# 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



ISBN 1-55937-448-9



*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.**

American National Standard

# 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 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

In 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

# 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-Gôran 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
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 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

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

This page intentionally left blank

## Contents

# 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 bused 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 II[4]:** 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 NuBus®[5]:** 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 generated 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.

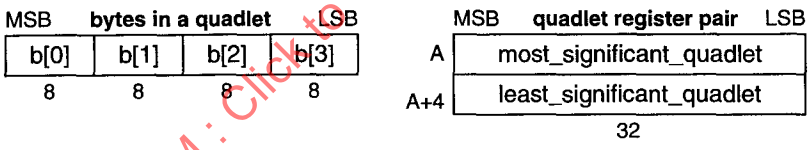



**Figure 1—Byte and quadlet ordering**

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 "$1A_{16}$" or "$11010_2$".

## 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 |
|---|---|
| `~c` | bitwise complement of integer *c* |
| `c^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, 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.

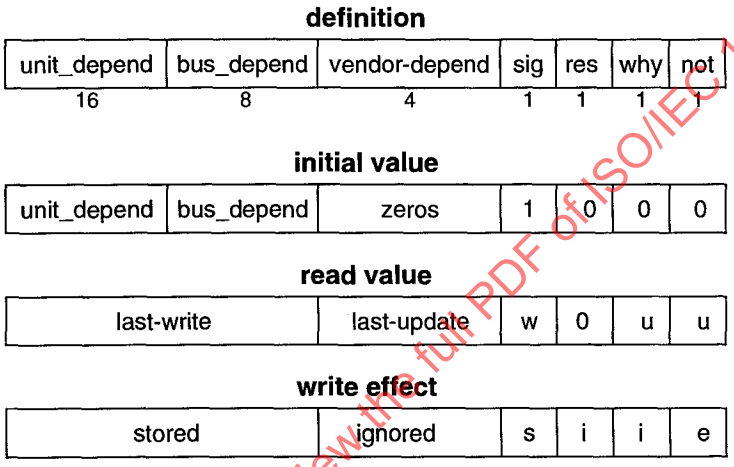



**Figure 2—CSR format specification (example)**

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

***last_update:***

| | |
|---|---|
| Abbreviation | *u* |
| 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 | *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 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:

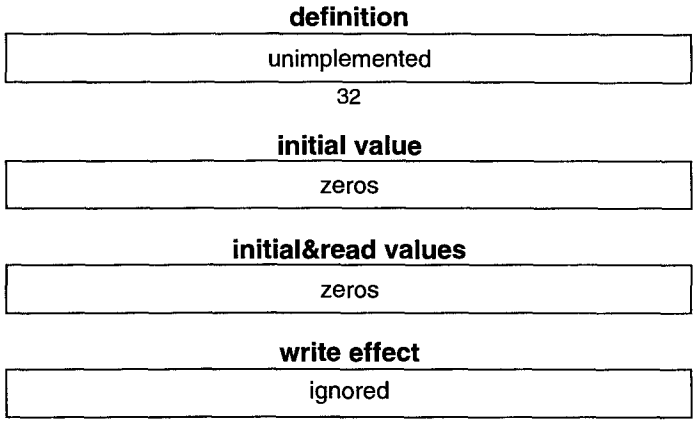



**Figure 3—Unimplemented register (hardwired to zero)**

***reserved:***

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

This page intentionally left blank

# 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 Futurebus+ (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.

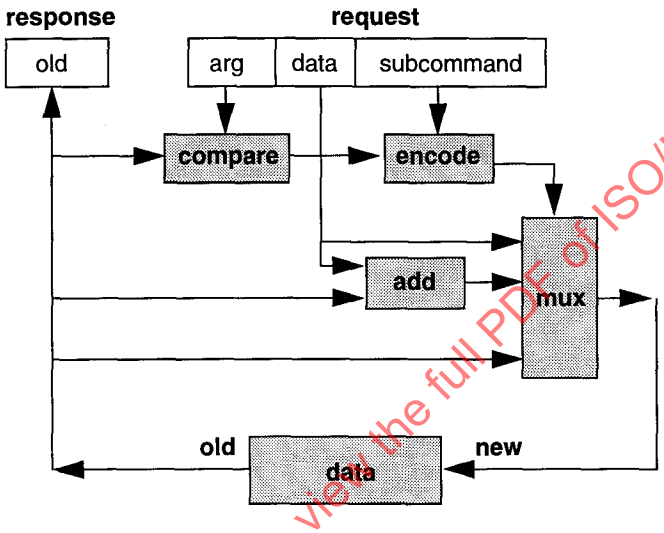



**Figure 4—Simplified lock model**

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 | `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.

## 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 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 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.

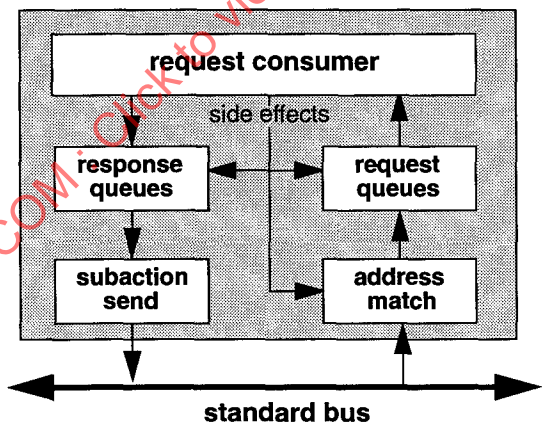



**Figure 5—Responder CSR processing model**

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 *off* bit discards queue entries, *lost* bit disables address spaces. |
| RESET_START | May discard request-queue and response-queue entries. |
| NODE_IDS | Changes address-match (all addressing models). |
| UNITS_BASE<br>UNITS_BOUND<br>MEMORY_BASE<br>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:

a) *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.

b) *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.

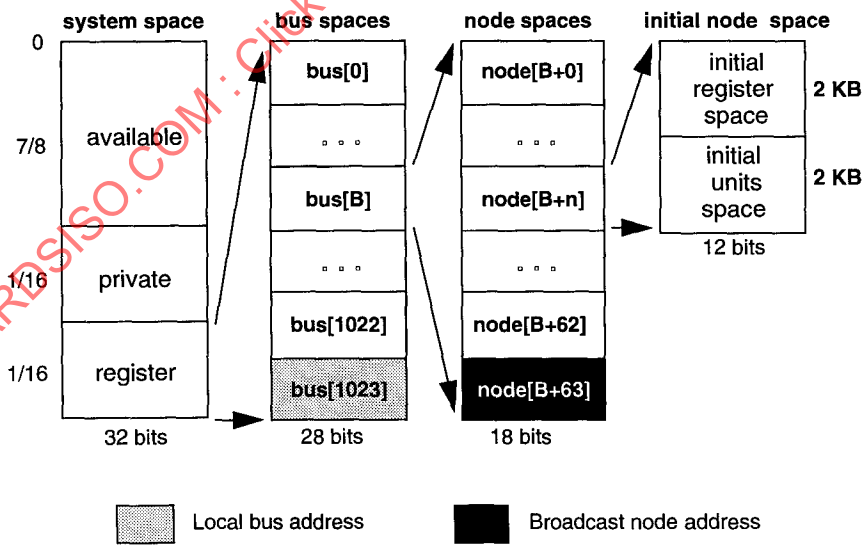



**Figure 6—32-bit extended addresses**

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 | EFFFFFFF | 32-bit private space |
| F0000000 | FFFFFFFF | 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 broadcall 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).

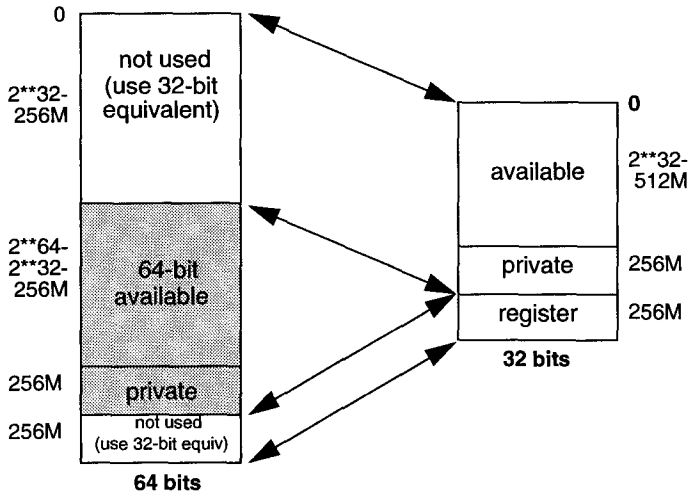



**Figure 7—64-bit extended addresses**

**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.

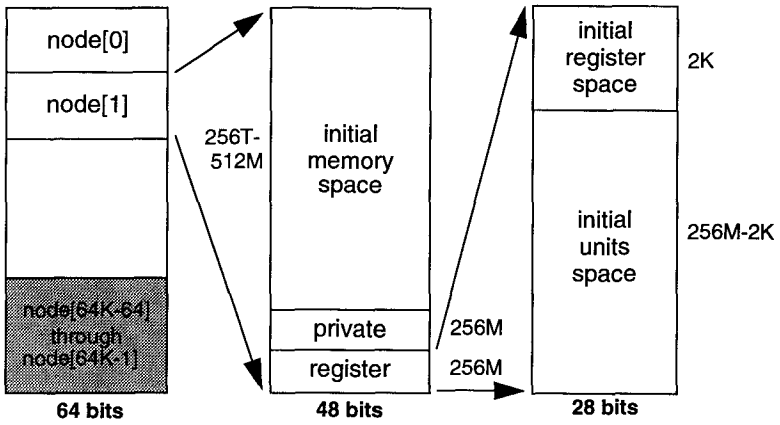



**Figure 8—64-bit fixed addressing**

## 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.

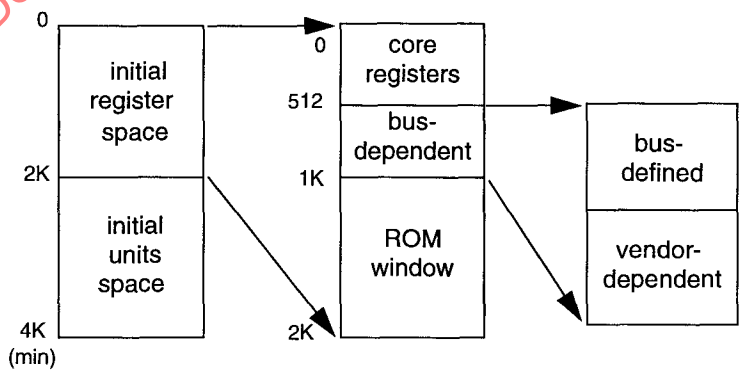



**Figure 9—Initial node space components**

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.

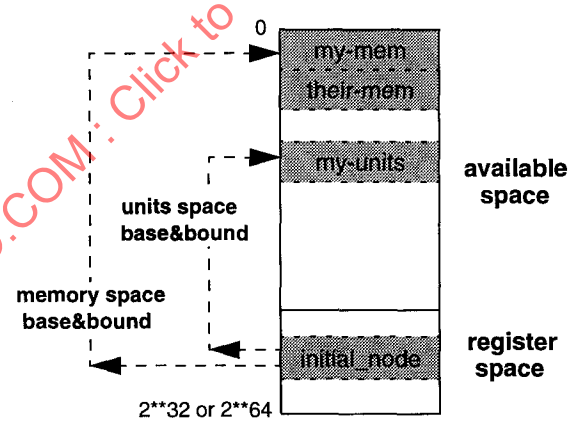



**Figure 10—Configurable extended addresses**

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.

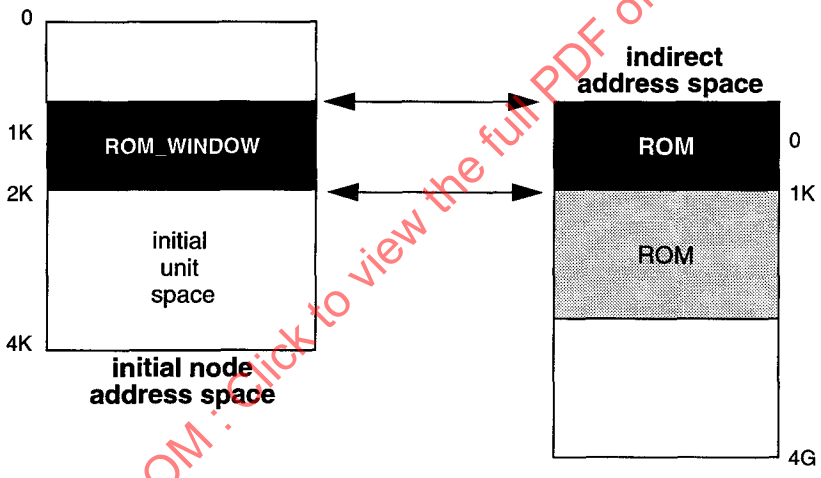



**Figure 11—Indirect space mapping (address less than 1 kbyte)**

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.

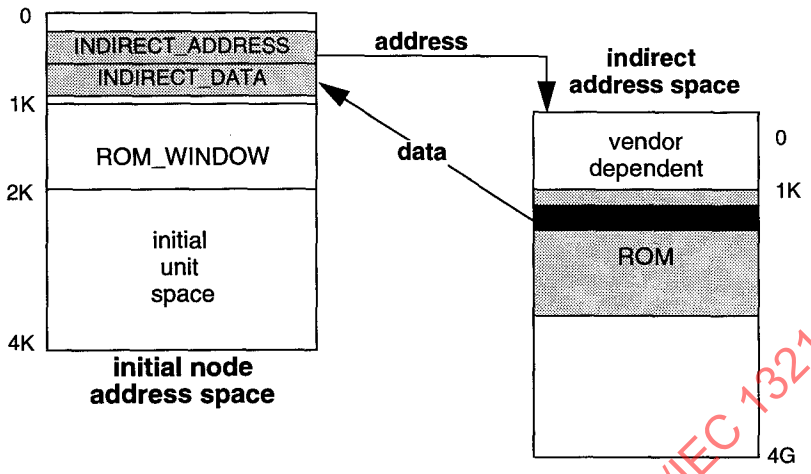



**Figure 12—Indirect space mapping (address greater than or equal to 1 kbytes)**

## 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 figure 13.

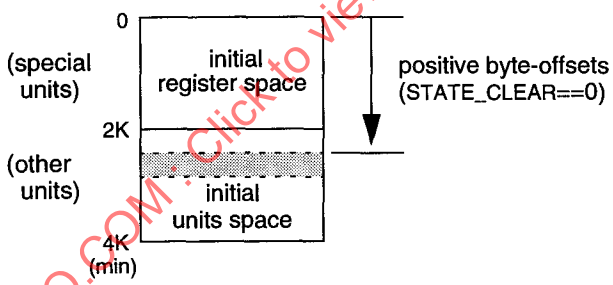



**Figure 13—ROM-specified offsets in the initial units space**

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 figure 14.

Since these ROM addresses are measured as offsets from the beginning of the address space, their meanings are not affected by the (software configured) location of the initial node or the extended units spaces.

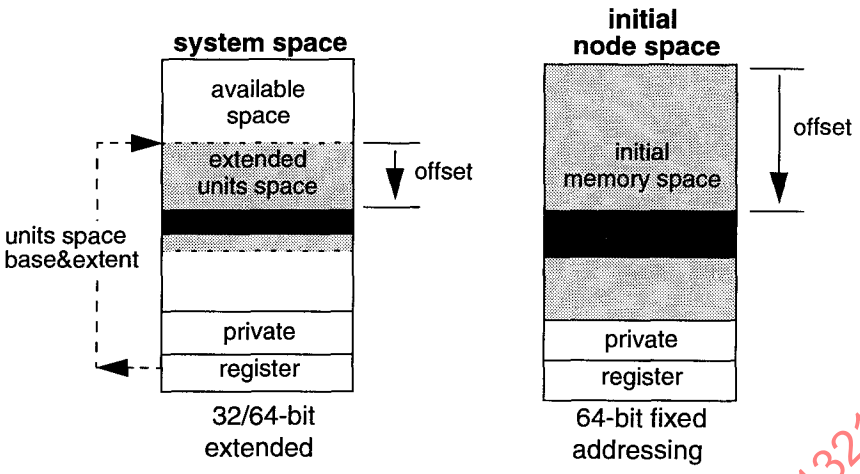



**Figure 14—ROM-specified offsets in extended units space**

# 5. Node architectures

## 5.1 Modules, nodes, and units

A simple configuration supported by the CSR Architecture includes modules attached to a shared bus, as illustrated in figure 15.

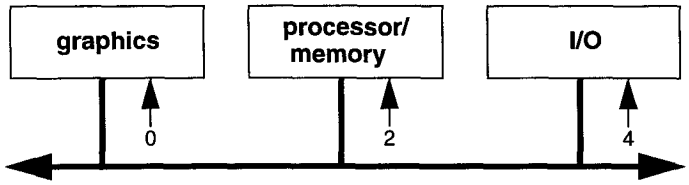



**Figure 15—Simple single-bus system**

To uniquely identify each module, a standard bus may provide geographical ID signals, which provide a unique code to each backplane slot (0, 2, and 4 as illustrated above). The module uses these backplane-provided signals and hardwired printed circuit metal-etch traces (which provide node-offsets +0 and +1, as illustrated in figure 16) to distinguish each node on the module from other nodes on the bus.

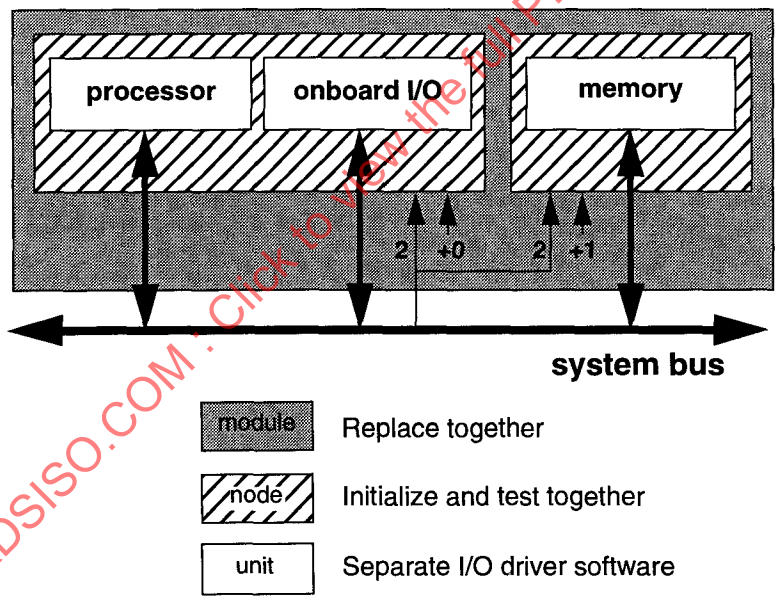



**Figure 16—Physical CSR component hierarchy**

Each module consists of one or more nodes, which are independently initialized and configured by operating-system software. Note that modules are a physical packaging concept and nodes are a logical addressing concept. A module is a board or board set, consisting of one or more nodes that share a physical interface. In normal operation, a module is not visible to software. Of course, this is not true when the module is replaced, when the shared bus interface fails, or when specialized module-specific diagnostic software is invoked.

A node is a logical entity with a unique bus-offset address. It provides an identification ROM and extended address-space registers, and it can be reset independently. A node has an initial address space (4 kbytes min-

imum), of which 2 kbytes are defined by the CSR Architecture. Since the definition of nodes is standardized, their initialization can be handled by generic system software.

The address space provided by a node can be directly mapped to one or more units. A unit is a logical entity, such as a disk controller, which corresponds to unique I/O driver software. On a multifunction node, for example, the multiprocessor, memory, and SCSI interfaces could be different units on the same node. Unless the node is reset or reconfigured, the I/O driver software for each unit can operate independently.

Within a unit there may be multiple subunits, which (after being configured by their shared I/O driver software) can be accessed through independent control registers or uniquely addressed DMA-command sequences. Although unit architectures should use the subunit concept to simplify I/O driver software, the definition of subunit architectures is beyond the scope of the CSR Architecture.

## 5.2 Node states

The definition of many CSRs is influenced by the concept of node states. There are four software-visible node states: initializing, running, testing, and dead. The running state is required; the other node states may be optionally implemented. All of the node's registers are accessible while the node is in the running state. Only a small portion of the node's registers is accessible while the node is in the initializing, testing, or dead states.

After a power_reset or command_reset (and if the initializing state is implemented), a node enters the **initializing** state. A node autonomously leaves the initializing state and moves to the **running** state (no fatal errors) or the **dead** state (fatal error(s) detected). A power_reset or command_reset is needed to change from the dead to the running or initializing states.

The start of a disruptive test (which is invoked through a write to the TEST_START register) moves the node from the running or dead state to the **testing** state and initiates a test sequence. At the completion of a disruptive or nondisruptive test, the node enters the running state or (if the node is non-operational) dead state.

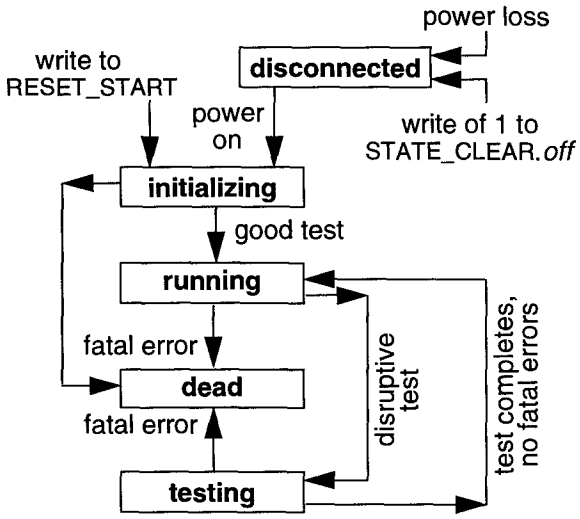
These node states and the events that trigger transitions between them are illustrated in figure 17.

A *disconnected* node-state concept is illustrated, but has a minimal effect on the CSR definitions. In the disconnected state, the node is not visible on the bus. A node may be in the disconnected state because its primary power has failed (it has no power to respond), because it has been disconnected from the bus by software (which sets the STATE_CLEAR.*off* bit to one, in preparation for on-line replacement), or when its hardware failures prevent it from responding to bus transactions.

Note that the CSR Architecture defines two forms of reset that may be used to enter the initializing state. A power_reset is an event that occurs when power is restored (or after a simulated loss of power, as indicated by power-status signals). A command_reset is an event that occurs during a write to the node's RESET_START register. These two forms of reset are summarized in table 9.

With one exception, the effects of a command_reset on the CSR Architecture shall be the same as power_reset. This exception is that on a power_reset the node_id shall be set to its initial value, and on a command_reset the node_id value shall be unchanged.

The side effects of a command_reset and power_reset may be different for bus-dependent and unit-dependent registers and fields. A bus standard may also define other forms of node reset, which are beyond the scope of the CSR Architecture.




NOTE—If implemented, lost bit should be set on transition to the dead state.

**Figure 17—Node states**

**Table 9—Types of node reset**

| Reset type | Description |
|---|---|
| power_reset | restoration of primary power, primary state is initialized |
| command_reset | write to RESET_START, most primary state is initialized |

## 5.3 Node testing

### 5.3.1 Access-path tests

Before using any node, a processor may verify the integrity of the node's CSR-access path. To assist in this task, a read/write ARGUMENT_LO register may be provided. Since the data value in this register has no direct side effects, it can be safely read and written to verify the CSR-access path before other node tests are remotely initiated.

### 5.3.2 Reset test

A simple reset test can be performed after the node is reset by writing to the RESET_START register. The reset test is initiated by writing a command to the node's **RESET_START** register; test completion is detected by polling the **STATE_CLEAR** register. Although the test interface is standard and the tests have some constraints, the tests may be radically different on different nodes.

Errors in the default reset test may prevent an update of the STATE_CLEAR register when the error is detected. Since reset tests are required to complete within ten seconds, a timeout can be used to detect such catastrophic errors, as illustrated in figure 18.

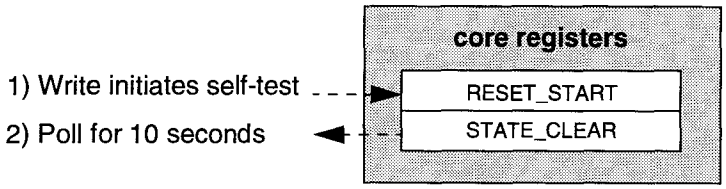



**Figure 18—Initialization test interface**

In the event of a reset-test failure, a standard mechanism for invoking other standard tests is provided, using another set of registers (TEST_START and TEST_STATUS).

### 5.3.3 Diagnostic tests

Standard but more complex diagnostic tests can be performed when the node is in the running or dead states. The diagnostic-test interface includes the TEST_START and TEST_STATUS registers, which allow an operator to select a specific test or set of tests to be run. The test options also allow these tests to be run in a tight loop, so that a logic analyzer or oscilloscope may be used effectively to isolate the defect.

Although the node-local **initialization** tests can test internal components, other **extended tests** are needed to test the bus-interface logic. The extended tests allow the node to access a remote address range while the test is being performed. The location of the remote address range is written to the node's argument registers before the extended test is started.

The CSR Architecture supports other optional diagnostic tests, which are useful for isolating the defects to a smaller number of components or connections. These **system** and **manual** tests, which are most useful in the repair depot, provide access and control features that make the test useful for defect isolation and repair. The four types of tests and their time-length constraints are as follows:

a) *Initialization tests.* The default set of tests shall complete in less than 10 seconds.
b) *Extended tests.* The default set of tests shall complete in less than 10 seconds.
c) *System tests.* There is no recommended time limit for system tests.
d) *Manual tests.* There is no recommended time limit for manual tests.

Note that system and manual tests are unit-specific and are not expected to be invoked by generic diagnostic control programs.

Errors in the default initialization and extended tests may prevent an update of the TEST_STATUS register when the error is detected. Since these tests are required to complete within 10 seconds, a timeout can be used to detect such catastrophic errors, as illustrated in figure 19.

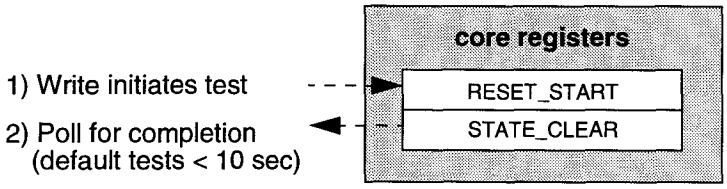



**Figure 19—Diagnostic test interface**

The default initialization and extended tests, as well as many of the other standard tests, are expected to be disruptive. A disruptive test may make the node nonoperational, in the sense that the node registers behave differently than in the running state. A disruptive test is initiated by a write to the TEST_START register, which immediately sets the TEST_STATUS.*active* bit to one, as illustrated in figure 20.

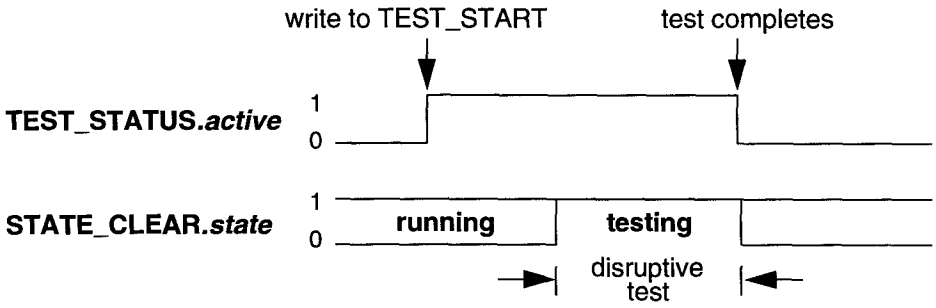



**Figure 20—Disruptive test interface**

The node may transition to the testing state when the TEST_START register is written, but shall transition to the testing state before the node's registers are disrupted. The node remains in the testing state until the disruptive test completes. At that time, the STATE_CLEAR.*state* field and the TEST_STATUS.*active* bit are simultaneously changed. The node state is set to the **running** (no fatal errors) or **dead** (fatal error(s) detected) and the TEST_STATUS.*active* bit is cleared to zero.

Other vendor-dependent tests may be nondisruptive. Diagnostic tests that are invoked when the node is in the dead state are also nondisruptive. A nondisruptive test has no effect on the node's operation, in the sense that the behavior upon access to the node registers does not change. A nondisruptive test is initiated by a write to the TEST_START register, which immediately sets the TEST_STATUS.*active* bit to one, as illustrated in figure 21.

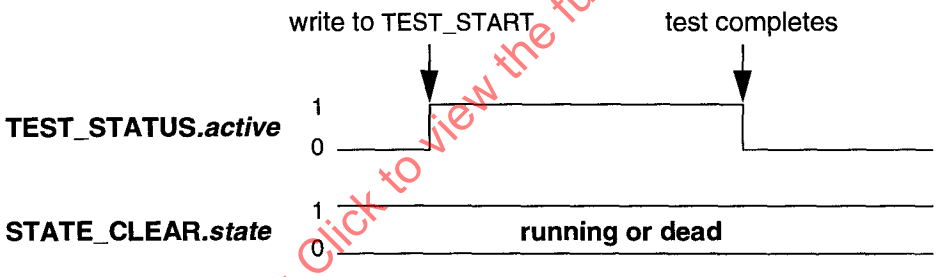



**Figure 21—Nondisruptive test interface**

The node remains in its previous state during the nondisruptive test. At that time, the TEST_STATUS.*active* bit is cleared to zero.

### 5.3.4 Non-standard diagnostic tests

The node's reset and standard tests may be insufficient to test the bus interface, which could be shared by other nodes on the same module. More-extensive module or system-level tests may be required to test this interface and to find other problems invoked by heavy system loads (stress tests). These extensive tests may be performed by vendor-dependent diagnostic software.

The CSR Architecture provides ROM values for associating the proper diagnostic software with the module under test. However, the diagnostic registers and test procedures are vendor-dependent.

## 5.4 Multinode modules

Multiple nodes can be located in a single module, which may display a summary of the node's states as illustrated in figure 22.

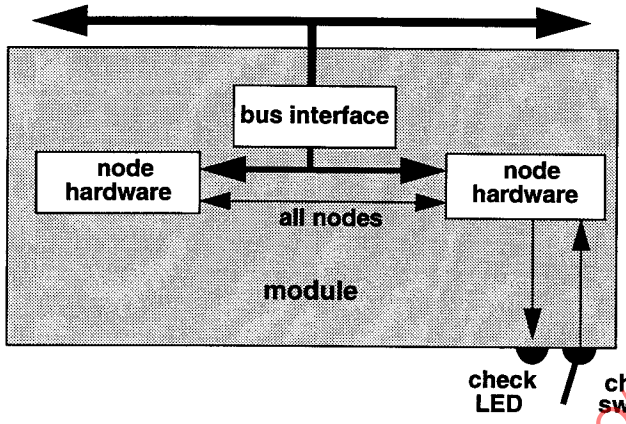



**Figure 22—Multinode module**

The shared **bus interface** is visible to vendor-dependent diagnostics, but is not normally visible to system software. For example, there are no defined registers for accessing the bus interface directly.

Each of the nodes has an operational **state** (such as initializing or running). To summarize its node's states, a module may have one or several state-displays. The details of the state display codes, as well as the state-summary algorithms (for consolidating the node's state information) are bus-dependent.

To support on-line replacement, a module may have an attention switch. Changing the attention switch position sets an attention bit in one of the node's status registers, and (depending on the details of additional unit-specific registers) may optionally send an interrupt. After observing the set attention status bit, software is expected to deconfigure the nodes on the module and disconnect the module from the bus. See 5.5 for details.

## 5.5 On-line replacement (OLR)

A bus standard may provide mechanisms for supporting on-line replacement (OLR). The CSR Architecture defines two bits to support OLR: the STATE_CLEAR.*atn* (attention) and the STATE_CLEAR.*off* bit. The remainder of this subclause defines how these bits are expected to be used.

A replacement warning signal event (1) is expected to be generated before a defective module is replaced (for example, the signal could be generated by a front-panel switch or a transaction sent over the diagnostic Serial Bus). This event is logged by setting the STATE_CLEAR.*atn* bit on one or more of the nodes on the module (2) and the processor is optionally interrupted (3).

After checking the STATE_CLEAR.*atn* bit (4), operating-system software is expected to idle the I/O driver software (5) associated with all nodes on the module (this could, for example, involve copying state to system memory). After the I/O driver software has been idled, the I/O driver software is expected to disconnect the module by setting the STATE_CLEAR.*off* bit on one of the nodes on the defective module (6). These steps are illustrated in figure 23.

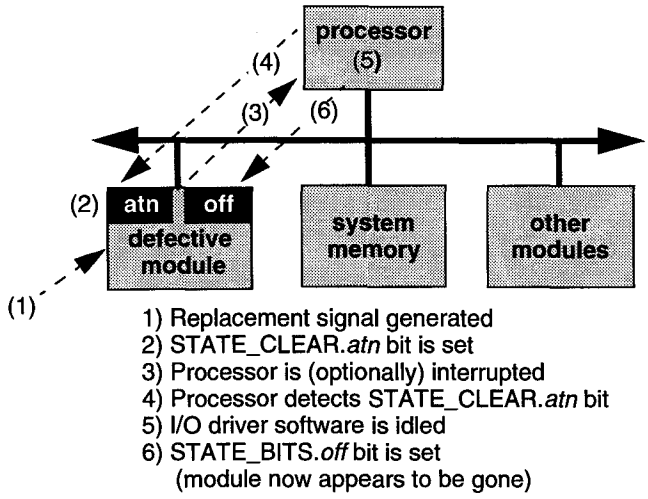



**Figure 23—OLR—defective module removal**

Once the STATE_CLEAR.*off* bit is set, all nodes on the module appear to immediately disconnect from the address space on the bus. However, an additional (bus-dependent) time may be required to reduce the module's power consumption before the module is replaced.

Since the STATE_CLEAR.*atn* bit, or the hardware used to access it, could be defective, modules are expected to provide alternative mechanisms for manually setting the STATE_CLEAR.*off* bit when software disconnection protocols fail.

This page intentionally left blank

# 6. Unit architectures

## 6.1 Unit architecture overview

The node's CSRs provide standard mechanisms for addressing, identifying, and testing hardware resources. After the system has been initialized, most of these registers are not accessed by the I/O driver software. Other register sets are provided to access processor, memory, or I/O device functions; these register sets define one or more unit architectures on the node.

The node provides address spaces that can be mapped to a unit architecture. Most of the unit's registers are expected to be mapped to a contiguous range of addresses within one of these (initial units, indirect or extended units) spaces. In normal operation, each unit has its own register set, and register sets of different units can be accessed concurrently by their I/O driver software.

In addition, a small number of CSRs on the node may be shared by its units. A few of the CSRs are used to route broadcast transactions (interrupts and messages) to the units through standardized CSR addresses on the node. One of the other CSRs, STATE_CLEAR, provides unit-dependent status bits, so the state of the node and its attached units can be quickly checked with one status register read.

To facilitate understanding these shared CSR resources, the interrupt and message-passing mechanisms are described in 6.2 and 6.3. Two of the partially standardized unit architectures (global clock and memory controller) are described in 6.4 and 6.5. Subclause 6.6 illustrates how other vendor-dependent unit architectures could be designed.

## 6.2 Interrupts

### 6.2.1 Interrupt-target registers

A standard CSR address offset, INTERRUPT_TARGET, is provided to support broadcast interrupts. A write to this address offset with the broadcast node address (node #63) is broadcast to all units on the local bus. A write to this address offset with a directed node address is nodecast to all units on the selected node. Other (unit-dependent) interrupt addresses may be provided for interrupts that are directed to a specific unit.

To improve efficiency and ensure forward progress, the standardized INTERRUPT_TARGET register supports the immediate acceptance of an arbitrary number of interrupt-write transactions. Although all broadcast interrupt events are immediately queued, the processing of the queued interrupts may be delayed (based on vendor-dependent interrupt-processing protocols).

The INTERRUPT_TARGET register provides one bit of storage for each of 32 interrupt groups; an interrupt is queued by setting the corresponding interrupt-group bit. Although this ensures that interrupt events can always be queued, there is no mechanism for determining the number of interrupt events that may have set a shared interrupt-group bit. When interrupt-group bits are shared, other polling mechanisms are needed to determine the source of the broadcast interrupt or to provide interrupt-service-routine parameters.

To ensure interoperability among processors and I/O controllers provided by different vendors, the CSR Architecture also restricts the functionality of directed interrupt registers. Directed interrupts are constrained to be **write4** transactions, whose address is a register within the vendor-defined processor unit architecture (note that a 64-bit address may sometimes be needed to address this register).

As with all CSRs, congested processors may assert a busy status to delay the acceptance of directed interrupts until storage space is available. However, to avoid the generation of system deadlocks, the processor is not expected to additionally delay the acceptance of the interrupt until other bus transactions are completed.

### 6.2.2 Interrupt-poll registers

When multiple nodes share an interrupt-group bit, polling is required to identify the interrupt source. Note that polling may never be required for processors whose **vendor-dependent directed-interrupt** definitions provide a sufficient number of interrupt-group bits.

To support such polling, a simple (non-DMA) node is expected to set a "service-request" bit within its STATE_CLEAR register before interrupting the processor. Alternatively, the processor may be programmed to periodically poll the STATE_CLEAR register of active I/O nodes.

To simplify and enhance the performance of the interrupt dispatch routine, 16 bits within the node's STATE_CLEAR register are reserved for units and expected to be used for this purpose. Note that other node-status bits are included in the same register, to minimize the overhead of simultaneously checking for other node-error conditions.

For shared interrupt-group bits, previously initialized memory tables or lists are expected to identify these interrupting nodes. In addition to a pointer to the node's STATE_CLEAR register, these data structures are expected to provide an interrupt mask value and interrupt-dispatch parameters, as illustrated in figure 24.



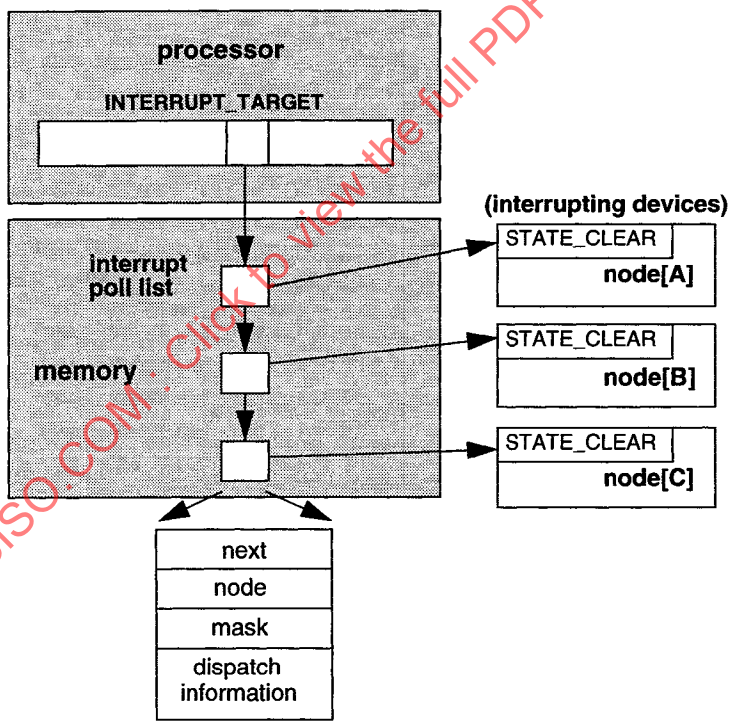



**Figure 24—Polled-CSR interrupt dispatch model**

An interrupting node produces a nonzero value when its STATE_CLEAR register is ANDed with its specified mask value. After the interrupting node has been identified, the selected status bits can be cleared by writing this nonzero value to the STATE_CLEAR register. The location, format, and functionality of the STATE_CLEAR register is standardized, to improve the performance of the latency-sensitive interrupt-polling software.

For higher-performance nodes with processor or DMA-like capabilities, the interrupting node can return completion-status reports to arrays or queues in memory. Nodes sharing one interrupt-group bit could share one completion-status list; each interrupt-group bit and its corresponding completion-status list could correspond to a different interrupt priority level. However, the details of these DMA-related data structures are beyond the scope of the CSR Architecture.

## 6.3 Message passing

Two standard CSR address offsets are provided to support broadcast message passing. A write to either of these address offsets with the broadcast node address (node [63]) is broadcast to all units on the local bus. A write to either of these address offsets within a directed node address is nodecast to all units on the selected node. Other unit-dependent message-passing addresses may be provided for messages that are directed to a specific unit.

Properly sized and aligned write transactions to these target addresses shall return a **done_correct** (if message queue space is available) or a **conflict_error** status (if no message queue space is available). All message-passing nodes are required to support 64-byte messages. Nodes may optionally support other message sizes (such as 16 or 256 bytes) as well, but larger message sizes are not transported indivisibly across all standard buses.

Separate target addresses are provided for request and response messages, to avoid message-queue deadlocks. A request message is defined as a message that could generate one or more response messages when processed. A response message never generates additional messages when being processed. Request and response messages may be directed to the MESSAGE_REQUEST address. Only response messages shall be directed to the MESSAGE_RESPONSE address.

Hardware is expected to provide a minimum of one queue entry for each message-passing address. The queue entry is updated and inserted into one of two message-received queues by writing to the corresponding message-passing address. The queues are emptied by processor firmware or software, which processes the queued messages. The protocols used by the processors to empty and interpret the contents of these messages are beyond the scope of the CSR Architecture.

Several techniques can be used to transfer larger messages using the 64-byte message-transfer primitives:

a) *Concatenated messages.* The larger message is split into smaller 64-byte messages. Messages contain identifiers that distinguish among the parts of the large message and smaller messages that may be concurrently accepted.
b) *Indirect messages.* The small message provides the address of a larger message, which can be DMA'd from memory-mapped data structures.

The format and meaning of 64-byte messages, as well as the techniques used to transmit large messages, are beyond the scope of the CSR Architecture.

## 6.4 Globally synchronized clocks

### 6.4.1 Clock overview

A bus standard may provide support for physically distributed yet globally synchronized clocks. Since the synchronization process involves a broadcast address and the clock values have a uniform format across bus standards, the register interface for the clock-unit architecture is partially standardized.

Real time systems often require a sense of the current physical time. The current time may be needed to record the time that events occur in the external world, to measure the elapsed time between events, to con-

trol the sequence in which event processing takes place, or to schedule the initiation of new events in the external world.

When only one processor requires physical time, this service is easily provided by a real time clock directly accessible to the processor. Although a central clock may be highly accurate internally, its accuracy is subject to loss in transit through the bus (due to variable arbitration and transmission latencies). A central clock also constitutes a single point of possible failure. Thus there is a need to provide distributed clocks for each node, and in order for the decentralized clocks to provide a common sense of time, the respective clocks must be synchronized.

A clock is expected to consist of a *clock_value* register and an oscillator. The *clock_value* register is updated by adding a *clock_tick* value to it at the completion of each cycle of the oscillator. The *clock_value* register may be initialized by software to any value within its range. To provide a common interface, all clock values are presented to the system scaled to the same *clock_value* format. The *clock_value* is represented as a 64-bit unsigned fixed-point number. The binary radix point divides the 64-bit integer into two 32-bit portions. The 32-bit most-significant portion represents seconds. The 32-bit least-significant portion represents a binary fraction of one second. The least-significant bit represents approximately 233 picoseconds. The 64-bit unsigned number overflows approximately once every 136 years.

The number of oscillations per second is called the frequency or rate of the clock. The completion of one oscillator cycle is called a tick. The tick period represents the time interval between two ticks and determines the resolution or granularity of the clock.

### 6.4.2 Clock synchronization

Clock synchronization is expected to involve the following steps:

a) *Clock sampling.* The values of all clocks are sampled at the same (or nearly the same) time, by distributing a clock_strobe signal to all nodes on the same bus. The sampled values are saved in the CLOCK_ARRIVED registers.
b) *Reference update.* A reference clock value (which corresponds to the time of the previous clock_strobe signal) is calculated. The reference clock may be derived in one of two ways:
   1) *Master reference.* The master's CLOCK_ARRIVED register is used as the value of the reference time.
   2) *Averaged reference.* The values of the CLOCK_ARRIVED registers from two or more nodes are averaged to provide the reference time.
c) *Clock adjustment.* The frequencies of clocks are changed to compensate for the errors between their sampled values and the reference time. The frequency adjustments may be performed in the following ways:
   1) *Master update.* The slave nodes provide external access to their clock-adjustment and clock-calibration registers. The clock master computes the clock slave clock errors and compensates for these errors through writes to their clock-adjustment registers.
   2) *Local updates.* The clock master distributes the reference time (as sampled on the last clock_strobe signal) to the clock slaves, using directed or broadcast writes to bus-dependent memory-mapped control registers. The clock slaves locally compensate for their own errors based on the differences between their internal clock_value register and the reference value received from the clock master.

On a traditional physically bussed bus standard, the clock_strobe is a well-defined phase of a write to a specific control register address. This event is nearly simultaneously observed by local nodes, which latch their impression of the current clock value in the CLOCK_ARRIVED register, as illustrated in figure 25.

On other bus standards based on point-to-point physical links, such as SCI, the clock_strobe signal must be forwarded through multiple nodes on the ringlet. Since the time delay through each node is variable, sepa-

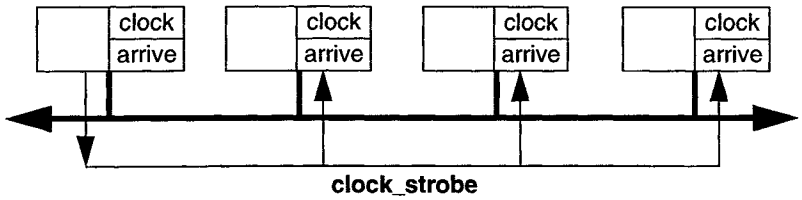



**Figure 25—Synchronized node clocks (broadcast backplane mode)**

rate registers are required to latch the time the clock_strobe signal arrives and the accumulated time between its arrival and departure (delta). Only the nodes with local clocks are required to have an arrival-time register, but (to accurately synchronize clocks) all nodes are required to provide the delta register. These registers are illustrated in figure 26.

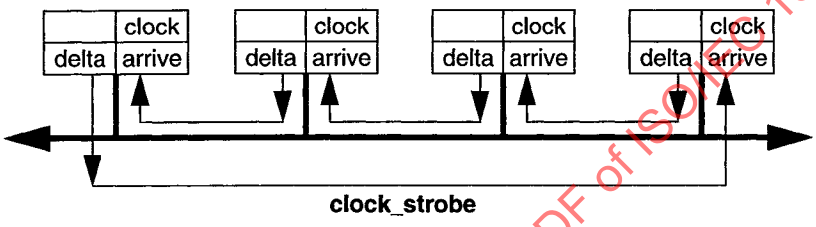



**Figure 26—Synchronized clocks (pipelined backplane model)**

The delta register capability may also be used on bridges between buses, to calibrate the time taken by the clock_strobe signal when passing through the bridge. However, the detailed functionality and format of the delta registers are beyond the scope of the CSR Architecture.

### 6.4.3 Clock update models

Several of the (optional) clock registers are needed to support a master update clock adjustment model. Six of the clock-related registers have standard formats and meanings; four of the clock-related registers are reserved for bus-dependent uses. The optionality of these ten clock-related registers, as well as additional definitional details, are bus-dependent.

The CLOCK_VALUE registers are used to initialize and monitor the current clock value, the CLOCK_TICK_PERIOD registers are used to adjust the current clock value, and the CLOCK_STROBE_ARRIVED registers are used to calibrate the current clock value. The definition of these registers is based on the clock design model illustrated in figure 27.

The upper 64 bits of the clock_value register are externally visible as the CLOCK_VALUE register pair. In figure 27, these registers are not shaded, to illustrate that register storage is not required to provide access to the internal clock_value register value. The clock_value register is expected to have additional less-significant bits, which cannot be accessed directly through CSR registers.

Although 32 least-significant bits of clock_value register are architecturally supported, an implementation may omit unnecessary least-significant bits in the adder and clock_value registers; the shading of these components is intended to illustrate this optionality. However, the resolution in the register and adder shall be sufficient to meet the node's clock-adjustment accuracy specification.

The adder is used to compute the next clock_value register value, which is updated at each tick of the oscillator. The difference between new and old values of the clock_value register is determined by the pair of

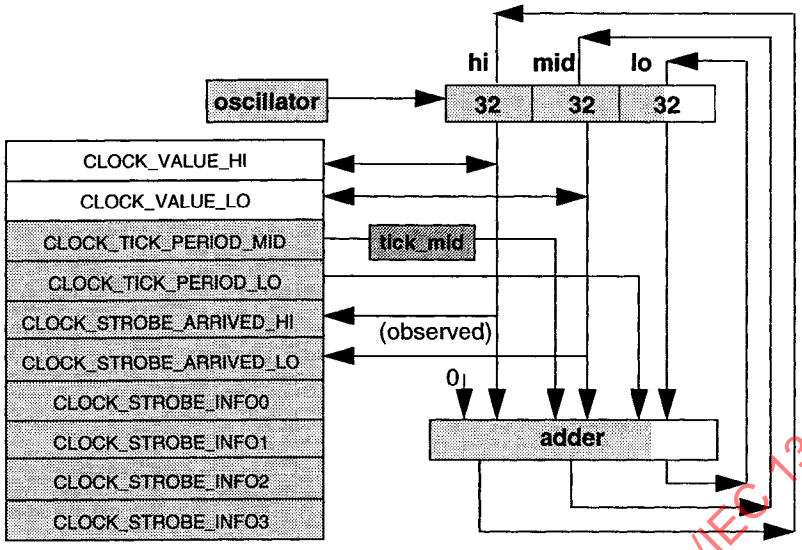



**Figure 27—Synchronized clock registers**

CLOCK_TICK_PERIOD CSRs, which supply the middle-significant and least-significant quadlets of the added value, respectively (the most-significant quadlet of the added value is zero). Software is expected to control the rate of the clock_value increase by setting the values in the CLOCK_TICK_PERIOD registers.

The CLOCK_STROBE_ARRIVED registers record the value of CLOCK_VALUE during the time that a strobe signal is observed on the bus. The CLOCK_STROBE_ARRIVED_MID register is sufficient to detect synchronization error differences up to half of a second; the CLOCK_STROBE_ARRIVED_HI register is needed to detect synchronization error differences that may be larger than half a second.

A bus standard may also support a local-update clock-adjustment model. Although a similar implementation model is expected, a local-update node is not required to provide CSR access to these resources; the details of their implementation are beyond the scope of the CSR Architecture.

### 6.4.4 Updating clock registers

Software is expected to use sequences of 32-bit accesses to access the larger 64-bit clock-register values. Although this complicates some of the register-access protocols, supporting the larger 64-bit read and write transactions would complicate the hardware designs.

Reading the CLOCK_VALUE register is expected to involve three reads of two quadlet registers. Software reads the CLOCK_VALUE_HI register and the CLOCK_VALUE_MID register before re-reading the CLOCK_VALUE_HI register. If the first and second values read from CLOCK_VALUE_HI are different (a carry condition), the second CLOCK_VALUE_HI time is used and the previously read value of CLOCK_VALUE_MID is cleared. The resultant 64-bit time value reflects a valid time between the first and second read of the CLOCK_VALUE_HI register.

Writing the CLOCK_VALUE register is expected to involve two quadlet register writes. Software is expected to set the time by writing zero to the CLOCK_VALUE_MID register and time value to the CLOCK_VALUE_HI register when the time is an integer multiple of seconds. Since the CLOCK_VALUE register need only be updated when the system is initialized, these difficulties in accurately setting the initial time value do not complicate the on-line use of these registers.

Writing the CLOCK_TICK_PERIOD_MID register is expected to involve two quadlet register writes. Software is expected to write to the CLOCK_TICK_PERIOD_MID register before writing to the CLOCK_TICK_PERIOD_LO register. The second write simultaneously updates the CLOCK_TICK_PERIOD_LO register and transfers the value of the CLOCK_TICK_PERIOD_MID register to the internal tick_mid register.

### 6.4.5 Clock accuracy requirements

The clock-calibration and clock-adjustment protocols shall minimally meet the following functional requirements (tighter restrictions on accuracy and drift may be specified by the bus standard):

a) *Clock tick period.* The internal *clock_value* register shall be updated at a nominal frequency of no slower than once per microsecond.

b) *Clock tick resolution.* The effective resolution of the clock-adjustment protocol shall be sufficient to adjust the clock drift rate to within 10 parts per million.

c) *Oscillator accuracy.* The frequency accuracy of the clock shall be 100 parts per million or less in the product's operating environment.

## 6.5 Memory unit architectures

The CSR Architecture provides the framework for the definition of memory unit architectures and constrains or supports its definition in the following ways:

a) *Read and write transaction set.* The CSR Architecture defines standard transactions that shall be implemented within the extended memory address space (see 3.2). The memory unit architecture may optionally support locks (see 3.3) or other bus-dependent transactions as well.

b) *RAM addressing.* The node provides MEMORY_BASE and MEMORY_BOUND registers that define the base and bound of the node's extended memory space. Addresses within this space are mapped through the node and processed by the memory unit.

c) *Address-space constraints.* The node's ROM provides standard mechanisms for specifying the alignment and size requirements for the node's extended memory space (see 8.4.14).

A memory controller with error-detection circuitry (EDC) can detect errors; a memory controller with error-correction circuitry (ECC) can also correct errors. A memory unit is expected to log detected and corrected errors in a unit-dependent error log and to set its allocated unit-dependent bit in the STATE_CLEAR register when the error is logged. The location of the memory-error bit and the details of the memory controller's error log are dependent on the memory-controller architecture.

A memory unit is expected to provide a set of registers for accessing the memory-controller functions. These registers could be used to initialize and test the contents of RAM and to select alternate memory banks. However, the detailed definition of these memory-unit registers is beyond the scope of the CSR Architecture.

## 6.6 Unit architecture environment

Unit architectures may be standardized or may have vendor-dependent definitions. A unit corresponds to a piece of I/O driver software and (in normal operation) the accesses to one unit do not affect the node or other units. A unit can be (optionally) reset without affecting the other units or the node to which it is attached.

The node architecture provides a framework for the definition of unit architectures; this framework is summarized below:

a) *Address space.* A unit (other than a memory unit) is located in a contiguous range of addresses within one of the initial units, extended units, or indirect address spaces. To support autoconfiguration of the units, the offset and size of the unit's address space is specified by an entry in the node's ROM.

b) *Initialized states.* The node command_reset and power_reset events also initialize the registers of their attached units. The initial state of the unit's registers is defined when the node enters the running state. Although their initial values are beyond the scope of this standard, one of the following conventions is expected:
   1) *Running.* The unit architecture is minimally initialized and left in the running state. I/O driver software is expected to invoke a unit-specific initialization or extended test.
   2) *Initializing.* The unit is in the initializing state and remains in this state until the initialization test has completed. I/O driver software is expected to poll periodically until the initialization test has completed.

c) *Messages.* Two standard message-passing target addresses are defined on the node. These addresses may be used to direct a message to all units on one node or to broadcast a message to all units on the bus.

d) *Interrupt target.* A standard INTERRUPT_TARGET address is defined on the node. This address may be used to broadcast an interrupt to all units on the bus or to send an interrupt to all units on one node.

e) *Interrupt poll.* Sixteen (16) bits of a standard interrupt-poll register (STATE_CLEAR) may be used by unit architectures on the node. This provides one register that can be efficiently polled to check the unit and node states.

f) *No-access spaces.* If the optional STATE_CLEAR.*lost* bit is implemented, accesses to registers in the initial units and extended units space are blocked when the node enters the initializing or dead states. This blocks access to the unit registers after a loss of power or after fatal errors.

g) *Power-fail warning.* On standard buses that provide centralized power, a (minimum) 4 ms power-fail warning should be provided. The warning is expected to be sufficient for most nodes to save their critical primary state (such as processor registers) in node-local secondary state (such as battery-backed RAM).

Nodes with requester capabilities are expected to pass the error status of bus transactions to the units that initiated them. Simple unit architectures (such as DMA or non-monarch-capable processors) are expected to log the bus transaction errors and set one of their allocated bits in the STATE_CLEAR register before halting. A processor is expected to periodically poll the STATE_CLEAR register to detect these halted-DMA error conditions.

Polling of the STATE_CLEAR register can be avoided if the unit provides a DMA heartbeat capability, which periodically returns its status to memory. The definition and use of such heartbeat capabilities are unit-dependent and beyond the scope of the CSR Architecture.

# 7. CSR definitions

## 7.1 Register names and offsets

This clause defines the format and meaning of the control and status registers (CSRs) that provide a framework for the design of vendor-dependent unit architectures (such as memory, I/O, or processors).

The CSR Architecture is not intended to provide a complete register definition for all node types—the vendor is expected to define additional unit-specific register sets. The intent is to standardize the registers that are accessed by generic software for initializing, configuring, and testing the node. Also, the intent is to specify broadcast addresses, to avoid incorrect interpretations of broadcast transactions directed to inconsistently defined addresses.

Many of the CSRs are optional and need not be implemented. However, other uses for a register address shall not be defined (the behavior of an unimplemented register is exactly defined). Thus, the cost of the implementation depends on the functionality provided by the node.

The location of a node-control register is defined by its byte-offset from the beginning of the initial register space. The CSRs needed to support the basic and optional capabilities are listed in table 10.

Note that some of the registers are affiliated with standard unit architectures; they provide extended address spaces for the units (UNIT_BASE through MEMORY_BOUND), a target address for broadcast/nodecast interrupts (INTERRUPT_TARGET), target addresses for broadcast/nodecast messages (MESSAGE_REQUEST or MESSAGE_RESPONSE), or they define a standard unit with special broadcast capabilities (CLOCK_VALUE through CLOCK_INFO).

The ROM_WINDOW specifies a range of register addresses that are mapped to the first 1 kbytes of the node's internal address space. The first portion of the internal address space shall contain ROM data (a minimum of 4 ROM bytes is required, see clause 8 for details).

A rather large space is allocated for bus-dependent registers; the bus standard is expected to define some of these to be **vendor-dependent**.

The read4 transactions are supported at all CSR addresses. The write4 transactions are supported to most CSR addresses, and the write64 transaction is supported to the message-target registers. For each of the CSRs, table 11 lists the expected software uses of the register (RO for read-only, WO for write-only, and RW for read/write) and the transaction types that are supported (read4, write4, etc.).

Access to the previously listed registers is supported when the node is in the running state. When a node is in the initializing, testing, or dead states, only limited access to the registers is supported, as described in table 12.

Only the first quadlet address of the ROM_WINDOW location always returns a defined result. A node's ROM is not required to be accessible while in the initializing state; if not accessible, a read of the first ROM_WINDOW address shall return a zero value. Once the ROM_WINDOW becomes accessible (a read of the first ROM_WINDOW address returns a nonzero value), the entire 1-kbyte ROM_WINDOW shall be accessible while the node is in the initializing state.

Accesses to the node's initial units and initial memory spaces are blocked when the STATE_CLEAR.*lost* bit is set to one; when blocked, an access of an address within these spaces completes with a type_error status.

**Table 10—CSR locations**

| Offset | Register name | Optional uses |
|---|---|---|
| 0 | STATE_CLEAR | state and control information |
| 4 | STATE_SET | sets STATE_SET bits |
| 8 | NODE_IDS | **required** |
| 12 | RESET_START | **required** (command_reset) |
| 16 | INDIRECT_ADDRESS | large ROM (>1 kbyte) |
| 20 | INDIRECT_DATA | " |
| 24 | SPLIT_TIMEOUT_HI | split requester (long timeout) |
| 28 | SPLIT_TIMEOUT_LO | split-requester (all timeouts) |
| 32 | ARGUMENT_HI | extended tests (64-address) |
| 36 | ARGUMENT_LO | diagnostic test interface |
| 40 | TEST_START | " |
| 44 | TEST_STATUS | " |
| 48 | UNITS_BASE_HI† | extended units space (64-address) |
| 52 | UNITS_BASE_LO† | extended units space |
| 56 | UNITS_BOUND_HI† | extended units space (64-address) |
| 60 | UNITS_BOUND_LO† | extended units space |
| 64 | MEMORY_BASE_HI† | extended memory space (64-address) |
| 68 | MEMORY_BASE_LO† | extended memory space |
| 72 | MEMORY_BOUND_HI† | extended memory (64-address) |
| 76 | MEMORY_BOUND_LO† | extended memory space |
| 80 | INTERRUPT_TARGET† | broadcast/nodecast interrupt |
| 84 | INTERRUPT_MASK† | broadcast/nodecast interrupt |
| 88 | CLOCK_VALUE_HI‡ | remote clock_unit read/write |
| 92 | CLOCK_VALUE_MID‡ | " |
| 96 | CLOCK_TICK_PERIOD_MID‡ | remote clock_unit calibration |
| 100 | CLOCK_TICK_PERIOD_LO‡ | " |
| 104 | CLOCK_STROBE_ARRIVED_HI‡ | " |
| 108 | CLOCK_STROBE_ARRIVED_MID‡ | " |
| 112 | CLOCK_INFO0‡ | bus-dependent clock_unit uses |
| 126 | CLOCK_INFO1‡ | " |
| 120 | CLOCK_INFO2‡ | " |
| 124 | CLOCK_INFO3‡ | " |
| 128–188 | MESSAGE_REQUEST† | target address for messages |
| 192–252 | MESSAGE_RESPONSE† | " |
| 256–380 | reserved | future IEEE Std 1212 definitions |
| 384–508 | ERROR_LOG_BUFFER | bus-dependent error log |
| 512–1020 | bus_dependent | bus-dependent |
| 1024–2044(max) | ROM_WINDOW | **required** |

† Access path to unit architectures.
‡ Unit architecture at standard locations.



Accesses to the node's extended units space is blocked when the STATE_CLEAR.*lost* bit is 1 or the UNITS_BASE.*enb* bit is zero; when blocked, an access of an address within this space completes with an address_error status.

**Table 11—CSR access modes**

| Register name | Uses‡ | Transactions |
|---|---|---|
| STATE_CLEAR | RW | read4,write4 |
| STATE_SET | WO | read4,write4 |
| NODE_IDS | RW | read4,write4 |
| RESET_START | WO | read4,write4 |
| INDIRECT_ADDRESS | RW | read4,write4 |
| INDIRECT_DATA | RO (RW*) | read4(write4) |
| SPLIT_TIMEOUT_HI | RW | read4,write4 |
| SPLIT_TIMEOUT_LO | RW | read4,write4 |
| ARGUMENT_HI | RW | read4,write4 |
| ARGUMENT_LO | RW | read4,write4 |
| TEST_START | RW | read4,write4 |
| TEST_STATUS | RO | read4 |
| UNITS_BASE_HI | RW | read4,write4 |
| UNITS_BASE_LO | RW | read4,write4 |
| UNITS_BOUND_HI | RW | read4,write4 |
| UNITS_BOUND_LO | RW | read4,write4 |
| MEMORY_BASE_HI | RW | read4,write4 |
| MEMORY_BASE_LO | RW | read4,write4 |
| MEMORY_BOUND_HI | RW | read4,write4 |
| MEMORY_BOUND_LO | RW | read4,write4 |
| INTERRUPT_TARGET | RB | read4,write4 |
| INTERRUPT_MASK | RW | read4,write4 |
| CLOCK_VALUE_HI | RW | read4,write4 |
| CLOCK_VALUE_MID | RW | read4,write4 |
| CLOCK_TICK_PERIOD_MID | RW | read4,write4 |
| CLOCK_TICK_PERIOD_LO | RW | read4,write4 |
| CLOCK_STROBE_ARRIVED_HI | RW | read4,write4 |
| CLOCK_STROBE_ARRIVED_MID | RW | read4,write4 |
| CLOCK_INFO0 | RW† | read4,write4 |
| CLOCK_INFO1 | RW† | read4,write4 |
| CLOCK_INFO2 | RW† | read4,write4 |
| CLOCK_INFO3 | RW† | read4,write4 |
| MESSAGE_REQUEST | WB | read4,write64 |
| MESSAGE_RESPONSE | WB | read4,write64 |
| ROM_WINDOW | RO | read4 |

* RW access expected when non-ROM units are located in indirect space, otherwise only read accesses are necessary.
† The CLOCK_INFO registers may require broadcast-transaction support.
‡ Uses keys:
RW Read/write: software is expected to read and write this register.
WO Write-only: software is only expected to write this register.
RO Read-only: software is only expected to read this register.
WB Write-broadcast: software is expected to write this register (directed and broadcast).

Accesses to the node's extended memory space is blocked when the STATE_CLEAR.*lost* bit is 1 or the MEMORY_BASE.*enb* bit is zero; when blocked, an access of an address within this space completes with an address_error status.

**Table 12—CSRs access modes in non-running states**

| Register name | initializing | testing | dead | Comments |
|---|---|---|---|---|
| STATE_CLEAR | R* | | | only *state* field |
| STATE_SET | W† | | | only *off* field |
| NODE_IDS | RW‡ | | | node configuration OK |
| RESET_START | W | | | resets always work |
| ARGUMENT_HI | no-access | RW | no-access | used by testing |
| ARGUMENT_LO | no-access | RW | no-access | " |
| TEST_START | no-access | RW | W | starts testing |
| TEST_STATUS | no-access | RO | no-access | needed when testing |
| ROM_WINDOW (minimal) | RO(zero) | RO | | ROM mostly available |
| Other CSRs within the initial register space | no-access | | | access limited |
| Initial units space | no-access§ | | | when space is enabled |
| Initial memory space | no-access§ | | | " |
| Extended units space | no-access** | | | " |
| Extended memory space | no-access** | | | " |

* *state* field only, other fields are no-access.
† *off* field only, other fields are no-access.
‡ *bus_id* and *offset_id* only, other fields are bus-dependent.
§ Returns type_error if space is disabled.
** Returns address_error if space is disabled.

To support multiple implementation technologies and testing strategies, access to most of the CSRs is only defined in the running state. However, bus- or vendor-dependent standards may require additional registers to be accessible while in these states. Special hardware is not required to detect CSR accesses while in these states, since registers which still meet the CSR specification are in a legal no-access state.

When a register access is no-access, reads return undefined data. A write to a no-access register may corrupt the node's state. While in the initializing or testing states, a write to a no-access register may cause the return of an incorrect test result when the node enters the running or dead states.

## 7.2 Minimal implementations

Only a small portion of these registers is required on a minimal node. If none of the node options are implemented, table 13 specifies the set of CSRs that shall be implemented.

**Table 13—Minimal CSR register set**

| Register name | Requirements |
|---|---|
| NODE_IDS | sets reconfigurable *bus_id* and *offset_id* values |
| RESET_START | software trigger for command_reset |
| ROM_WINDOW | one quadlet is required |

A standard Node_Capabilities field in ROM is provided to identify which options are implemented. See Clause 8 for details.

## 7.3 Unsupported register accesses

In this 2-kbyte initial register space, read4 transactions are supported. The CSR definitions also support other transaction types or sizes (for example, write4). Unless explicitly stated otherwise, these transactions return a done_correct status or a bus-dependent error when an unrecoverable transmission error is detected.

Within the initial register space, an unsupported write4 transaction shall have no side effects; it shall return a **type_error** status (so the error can be quickly detected) or a **done_correct** status (a lower-cost option). Within the initial register space, other unsupported transactions (which are not read4 or write4) shall return a type_error status.

The returned status for an unsupported transaction in other spaces is vendor- or unit-dependent. However, unsupported transactions are constrained to return values derived from the data that is addressed and their side effects shall be equivalent to writes of arbitrary data to the addresses that are accessed.

## 7.4 Register definitions

### 7.4.1 STATE_CLEAR

The STATE_CLEAR register provides bit fields, which are used to log special node-related, bus-dependent, and unit-dependent events. The format of the STATE_CLEAR register is illustrated in figure 28.

**definition**

| unit_depend | bus_depend | lost | dreq | r | elog | atn | off | state |
|---|---|---|---|---|---|---|---|---|
| 16 | 8 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |

**initial values**

| zeros | bus_depend | 1 | 0 | 0 | 0 | 0 | 0 | 00,01 |
|---|---|---|---|---|---|---|---|---|

**read value**

| unit_depend | bus_depend | w | w | 0 | w | w | 0 | last-update |
|---|---|---|---|---|---|---|---|---|

**write effect**

| clears selected (writeable) state_bits | ignored |
|---|---|

**Figure 28—STATE_CLEAR format**

**STATE_CLEAR:**

| | |
|---|---|
| Optional(RW) | Shall be implemented if any of its field values are implemented. |
| Initial value | See field definitions. |
| Read4 value | Returns the internal state_bits value. |
| Write4 effect | Shall update writeable fields by ANDing the current state_bits value with the complement of the write-data value. |

Sixteen of these bits are allocated for unit-dependent uses. A unit-dependent bit may be unused, may be assigned to one unit, or may be shared by multiple units. If shared by multiple units, these bits shall only be used for interrupt-status reporting; a bit shall only be set when the unit generates an interrupt. To provide a

uniform environment for the design of bus-independent unit architectures, the unit-dependent fields **shall not** be used for **bus-dependent** purposes.

Eight of these bits are allocated to bus-dependent uses. For example, these bits could be used to sense front-panel events or to illuminate LEDs on the front panel.

One of these bits (***r***) is reserved for future definition by the CSR Architecture.

The ***lost*** bit is set to its initial value of 1 by a command_reset, a power_reset, or when the node enters the dead state. Operating system software is expected to clear the lost bit after the node has been initialized and the unit's I/O drivers have been informed of the reset or fatal event. Software is not expected to set the *lost* bit; on a split-response bus, such accesses may have bus-dependent effects on request subactions that have been queued but have not been processed.

The ***dreq*** bit is expected to be set by software, to disable requests from unreliable nodes. Other unit-dependent bits may also be used to initially disable requests from DMA controllers or bridges until they have been properly initialized by system software. Nodes may provide a back-door access to the *dreq* bit, which may be used by general or special-purpose (remote diagnostic interface) processors to clear their own *dreq* bit when it has been improperly set by others.

The ***elog*** bit is expected to be set by node-internal hardware and is intended to inform software when an error has been detected and the node's error log has been updated. The *elog* bit shall be set to one when the error log is updated. Operating system software is expected to clear the *elog* bit after saving the contents of the node's error log. A bus standard may specify mechanisms for clearing the elog bit as a side effect of saving the error-log data. This is a status bit that has no effect on the node's operation.

The ***atn*** bit is expected to be set by node-internal hardware and is intended to inform software when the module containing this node should be prepared for on-line replacement. The bit may be set by a (momentary throw) front-panel switch or by a special command on a redundant diagnostic bus. This is a status bit that has no effect on the node's operation.

The ***off*** bit is expected to be set by software to disconnect a board from the bus resources before the board is replaced. Software is not expected to otherwise access the node while the *off* bit is being set; on a split-response bus, setting the *off* bit may discard the responses to such accesses (which are expected to generate response_timeouts) and the transaction that sets the *off* bit may also be terminated with a response_timeout.

The formal definitions of these fields follows.

***unit_depend:***

| | |
|---|---|
| Optional | The implemented bits shall be defined by the node's unit architectures. |
| Initial value | Zero. |
| Cleared | By STATE_CLEAR write, when the corresponding bit is one. |
| Set | (1) By STATE_SET write, when the corresponding bit is one.<br>(2) By unit-dependent events. |
| Effects: | None. These bits shall not affect the operation of the node or its units. |

***bus_depend (writeable bits):***

| | |
|---|---|
| Optional | The implemented bits shall be defined by the bus standard. |
| Initial value | Bus-dependent, but should be zero. |
| Cleared | By STATE_CLEAR write, when the corresponding bit(s) are one. |
| Set | (1) By a STATE_SET write, when the corresponding bit(s) are one.<br>(2) By bus-dependent events. |
| Effects | Bus-dependent. |

***bus_depend (read-only bits):***

| | |
|---|---|
| Optional | The implemented bits shall be defined by the bus standard. |
| Initial value | Bus-dependent, but should be zero. |
| Cleared | By bus-dependent events. |
| Set | By bus-dependent events. |
| Effects | Bus-dependent. |

***lost:***

| | |
|---|---|
| Optional | Required if the node is independently powered or if the dead state is supported. |
| Initial value | One. |
| Cleared | By STATE_CLEAR write (if STATE_CLEAR.*lost* is one). |
| Set | (1) By STATE_SET write (if STATE_SET.*lost* is one).<br>(2) By a power_reset.<br>(3) By a command_reset. |
| Effects(0) | None. |
| Effects(1) | (1) Accesses to the initial units space shall return a type_error.<br>(2a) For 32-bit and 64-bit extended address models, an access of the extended units and extended memory spaces returns an address_error status.<br>(2b) For 64-bit fixed address model, access of the initial memory space returns a type_error status. |

***dreq:***

| | |
|---|---|
| Optional | Required if bus-dependent mechanisms are not provided to disable the requester's initiation of bus transactions. |
| Initial value | Zero. |
| Cleared | By STATE_CLEAR write (if STATE_CLEAR.*dreq* is one). |
| Set | By STATE_SET write (STATE_SET.*dreq* is one). |
| Effects(0) | None. |
| Effects(1) | Initiation or retry of request subactions is inhibited. |

***elog:***

| | |
|---|---|
| Optional | Required when the optional error-log is implemented. |
| Initial value | Zero. |
| Cleared | (1) By STATE_CLEAR write (if STATE_CLEAR.*elog* is one)<br>(2) By saving the contents of the error log. |
| Set | (1) By STATE_SET write (if STATE_SET.*elog* is one)<br>(2) By an update of the node's error log. |
| Effects | None. |

***atn:***

| | |
|---|---|
| Optional | Required to support on-line replacement. |
| Initial value | Zero. |
| Cleared | By STATE_CLEAR write (if STATE_CLEAR.*atn* is one). |
| Set | (1) By STATE_SET write (if STATE_SET.*atn* is one).<br>(2) By the node upon observation of an on-line-replacement signal. |
| Effects | None. |

***off:***

| | |
|---|---|
| Optional | Required for supporting on-line replacement or board disconnects. |
| Initial value | Zero. |
| Cleared | Not applicable (always read as 0). |
| Set | (1) By STATE_SET write (if STATE_SET.*off* is one).<br>(2) By node-local disconnect signal. |
| Effects(0) | None. |
| Effects(1) | All nodes on the module shall immediately enter the disconnected state (they shall no longer respond to bus transactions). |

***state:***

| | |
|---|---|
| Optional | Required for supporting the initializing, testing, and dead node states. |
| Initial value(0) | If the node is initially in the running state. |
| Initial value(1) | If the node is initially in the initializing state. |

The *state* field is a read-only field that specifies one of four node states, as specified in table 14.

**Table 14—Node *state* values**

| *state* | Name | Description |
|---|---|---|
| 0 | running | initialization complete, now running |
| 1 | initializing | initialization reset&test in progress |
| 2 | testing | testing in progress (TEST_START invoked) |
| 3 | dead | fatal error, node is nonoperational |

### 7.4.2 STATE_SET

Any writeable bits in the internal state_bits register can be set by a write of the STATE_SET register, with a one in the corresponding bit locations. The format of this register is illustrated in figure 29.

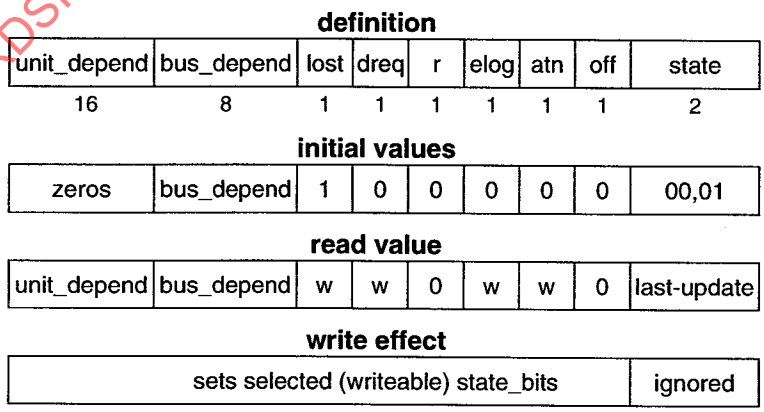



**Figure 29—STATE_SET format**

**STATE_SET:**

| | |
|---|---|
| Optional(W) | Required if STATE_CLEAR is implemented. |
| Initial value | Same as STATE_CLEAR. |
| Read4 value | Same as STATE_CLEAR. |
| Write4 effect | The writeable bits in the STATE_CLEAR register are indivisibly updated by ORing their current values with the corresponding write-data value. |

See 7.4.1 STATE_CLEAR for definition of individual fields.

### 7.4.3 NODE_IDS

The NODE_IDS register is used to identify and modify the current node_id values, which directly affect the initial node address. The NODE_IDS register contains the read/write node_id value and a 16-bit bus-dependent field, as illustrated in figure 30.

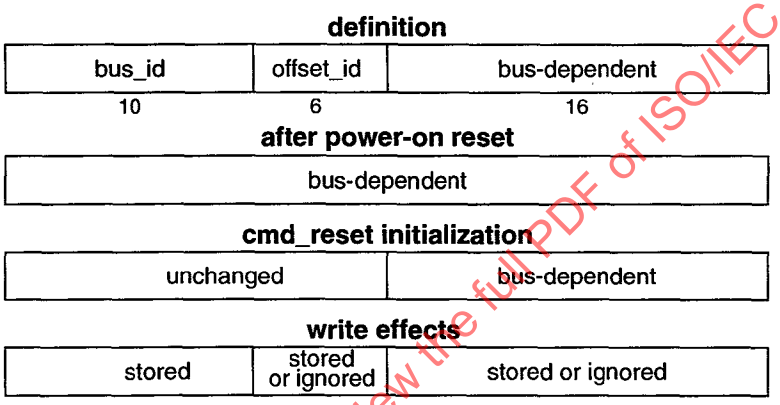



**Figure 30—NODE_IDS format**

**NODE_IDS:**

| | |
|---|---|
| Required(RW) | Used to configure the 4-kbyte initial node address space. |
| Initial value | See field definitions. |
| Read4 value | The writeable fields shall return the last-write value. |
| Write4 effect | Shall be stored to the writable fields. |

The 16-bit node_id value, which is the concatenation of the *bus_id* and *offset_id* fields, shall be used to specify the initial node space and may also be used to specify the return address for split-response bus transactions. Note that the node_id value has no effect on the location of the extended units or extended memory spaces.

When the extended address space is used, the 10-bit read/write ***bus_id*** field provides software with a mechanism for reconfiguring the bus address portion of the node's 4-kbyte initial node address space. The bus standard shall define the 6-bit ***offset_id*** field to be either a read/write or read-only field, but shall not allow both options.

Note that the initial node_id values after a power_reset and after a command_reset can be different. After a loss of power, the initial *bus_id* and *offset_id* values are bus-dependent. After a command_reset initialization, the *bus_id* and *offset_id* values remain unchanged. These distinctions are expected to simplify operating-system code by allowing a node to be reset without moving its initial node space.

Software is not expected to otherwise access the node while the NODE_IDS register is being changed; on a split-response bus, such accesses may have a bus-dependent effect on request subactions that have been queued but have not been processed.

The *bus_id* and *bus_dependent* fields have the following definitions:

***bus_id:***

| | |
|---|---|
| Required | This field shall be implemented. |
| Initial value | (a) For power_reset, bus-dependent (should be 1023).<br>(b) For command_reset, shall return the last-write value. |
| Read value | Shall return the last-write value. |
| Write effect | Shall be stored. The value in the *bus_id* field affects the base address of the initial node space. |

***bus_depend:***

| | |
|---|---|
| Optional | This field shall be defined by the bus standard. |
| Read value | The writeable fields shall return the last-write value. |
| Write effect | The writeable fields shall be stored. |

When the *offset_id* field is writeable, it has the following definition:

***offset_id:***

| | |
|---|---|
| Required | This field shall be implemented. |
| Initial value | (a) For power_reset, bus-dependent. However, all *offset_id* values on the bus shall be different.<br>(b) For command_reset, shall return the last-write value. |
| Read value | Shall return the last-write value. |
| Write effect | Shall be stored. |

When the *offset_id* field is hardwired, it has the following definition:

***offset_id:***

| | |
|---|---|
| Required | This field shall be implemented. |
| Initial value | Shall be hardwired to a bus-unique value. |
| Read value | Shall return the hardwired value. |
| Write effect | Shall be ignored. |

Nodes responsible for initialization shall provide a vendor-dependent mechanism for accessing their own node_id value. This allows these nodes to determine which of the bus-visible node_id addresses is their own.

### 7.4.4 RESET_START

A write to the RESET_START register performs an immediate command_reset. The command_reset initializes the node's primary state and optionally invokes an initialization test. If the initialization is immediate and successful, the node is left in the running state. If the initialization is delayed the node is temporarily left in the initializing state. Software is not expected to read the RESET_START register during normal operation.

Other request or response subactions may be queued in the node when a command_reset is performed. The node may discard these queued subactions as a side effect of the reset command. The node may also process these queued subactions, after the immediate effects of the write to the RESET_START register have been completed and the node is left in the initializing state (or running state, if the initializing state is not implemented).

Software is not expected to otherwise access the node while the RESET_START register is being changed; on a split-response bus, such accesses may have a bus-dependent effect on request subactions that have been queued but have not been processed.

The format of the RESET_START register is illustrated in figure 31.

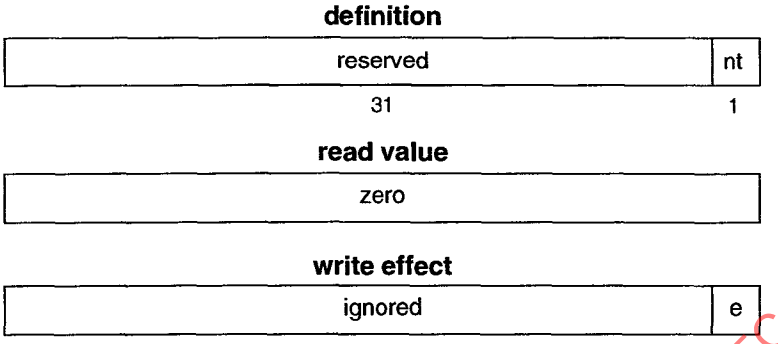



**Figure 31—RESET_START format**

**RESET_START:**

| | |
|---|---|
| Required(WO) | A write to this register performs an immediate command_reset. |
| Write4 effects | (1) The *nt* bit may be used as a parameter to the command_reset.<br>(2) The read-only STATE_CLEAR.state field shall be set to the initializing state (or running if the initializing state is not implemented).<br>(3) A command_reset shall be performed. |

The no-test bit (***nt***) can be use to selectively disable the initialization tests, which shall take less than 10 s to complete. Note that the 10-second limit applies to node tests; unit-dependent tests (such as memory-controller RAM initialization) may take longer than the 10-second limit. This *nt* bit is defined below:

***nt:***

| | |
|---|---|
| Optional | Required if an initialization test is provided and the node has the ability to not run it. |
| Write effects(0) | The node shall be reset and an initialization test should be performed. |
| Write effects(1) | The node shall be reset and an initialization test should not be performed. |

### 7.4.5 INDIRECT_ADDRESS

A node shall locate its ROM entries and may locate its specialized units in its indirect space; indirect-space addresses above the first 1 kbyte are accessed indirectly through the INDIRECT_ADDRESS and INDIRECT_DATA registers. Note that the indirect address space is different from the node's **private** or **internal** spaces, which may be used by a processor to access its node-local memory-mapped resources.

The format of the INDIRECT_ADDRESS register is illustrated in figure 32. Since indirect addresses are assumed to be quadlet aligned, the two LSBs (which should normally be zero) are reserved.

**INDIRECT_ADDRESS:**

| | |
|---|---|
| Optional(RW) | Required if ROM or units are at indirect addresses greater than 1 kbyte. |
| Initial value | Zero. |
| Read4 value | The *indirect_address* field returns the last-write value. |
| Write4 effect | The *indirect_address* field shall be stored. |

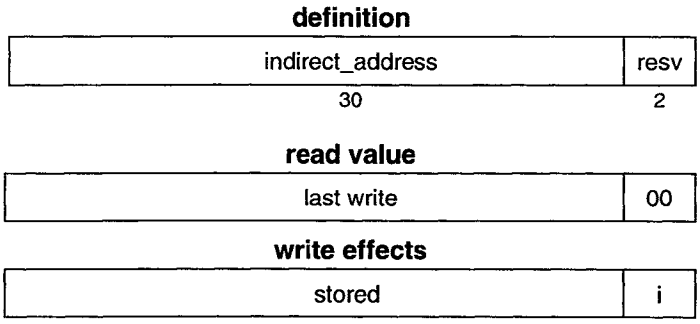



**Figure 32—INDIRECT_ADDRESS format**

The *i* field in figure 32 illustrates that the 2-bit reserved field (*resv*) is ignored when the INDIRECT_ADDRESS register is written.

See 8.2 for the structure and meaning of the first addresses in the indirect space. Other unit architectures may be located in the remainder of the indirect space.

### 7.4.6 INDIRECT_DATA

A read of the INDIRECT_DATA register returns a quadlet data value from the node's indirect space, based on the address in the INDIRECT_ADDRESS register. A write of the INDIRECT_DATA register stores a quadlet to the same indirect-address. The format of the INDIRECT_DATA register is illustrated in figure 33.

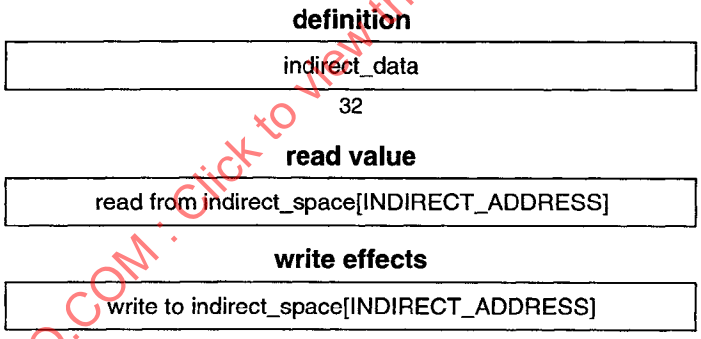



**Figure 33—INDIRECT_DATA format**

**INDIRECT_DATA:**

| | |
|---|---|
| Optional(RO/RW) | Required if the INDIRECT_ADDRESS is implemented. |
| Read4 value | Shall return the indirect_space[INDIRECT_ADDRESS] value. |
| Write4 effect | Shall be stored to indirect_space[INDIRECT_ADDRESS] if that location is writable. |

### 7.4.7 SPLIT_TIMEOUT

The SPLIT_TIMEOUT registers set the default timeout value for detecting split-transaction errors. After a request subaction is sent, the value of SPLIT_TIMEOUT sets the time within which the response subaction should be received. After this time, a requester is expected to terminate the transaction with a response_timeout status.

This format is consistent with the optional clock registers in this standard and other timeout register formats in the bus standards. The integer and fractions-of-a-second format allows a range of up to $2^{32}-1$ s (approximately 136 years) with a resolution of $2^{-32}$ s (the value of the LSB is approximately 233 ps).

The **zero value** has special meaning: the split-response timeout value is ignored. The timeout value is effectively infinite.

The bus standard shall specify the range of values in the split-transaction register that shall be implemented (which is expected to be much less than 136 years and much more than 233 ps). If the specified timeout value is outside this range, the operation of the timeout-detection hardware may be undefined.

Software is not expected to update the SPLIT_TIMEOUT register while a split-transaction is active on the same node. Such updates may have a bus-dependent effect on the effective timeout value used to generate the response_timeout status.

The format of the SPLIT_TIMEOUT register is illustrated in figure 34.

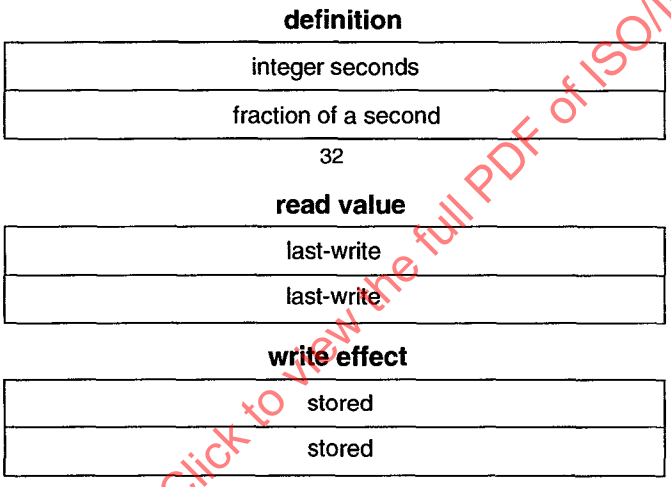



**Figure 34—SPLIT_TIMEOUT register**

**SPLIT_TIMEOUT:**

| | |
|---|---|
| Optional(RW) | Required for requesters supporting split-response transactions. |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. |

### 7.4.8 ARGUMENT

Software is expected to use write4 and read4 transactions to the ARGUMENT_LO register to test the node's basic bus interface. After this access path has been tested, the values of the ARGUMENT_HI and ARGUMENT_LO registers may be used to pass arguments to the extended tests that are initiated through writes to the TEST_START register.

The meaning of these argument values is dependent on the value written to the TEST_START register. The ARGUMENT_HI and ARGUMENT_LO registers are read/write registers, with the format illustrated in figure 35.

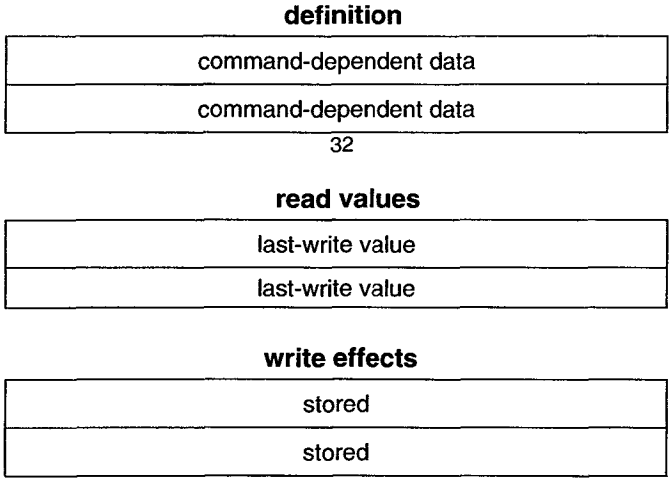



**Figure 35—ARGUMENT formats**

**ARGUMENT_HI:**

| | |
|---|---|
| Optional(RW) | Required if extended tests are implemented and the node supports 64-bit addressing. May also be required for vendor-dependent tests that are initialized through writes to the TEST_START register. |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. |

**ARGUMENT_LO:**

| | |
|---|---|
| Optional(RW) | Required if extended tests are implemented.<br>May also be required for vendor-dependent tests that are initialized through writes to the TEST_START register. |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. |

These registers should not be updated while a test is in progress. If the register values are updated during this time, the side effects of the test may be undefined.

The values of the ARGUMENT_HI/ARGUMENT_LO registers have a special meaning when an extended test is invoked (a write to the TEST_START register when TEST_START.*extended* is one); these two registers specify the address of a 4-kbyte memory space that may be accessed during the extended test to exercise the bus interface. In this case, the format of these registers is illustrated in figure 36.

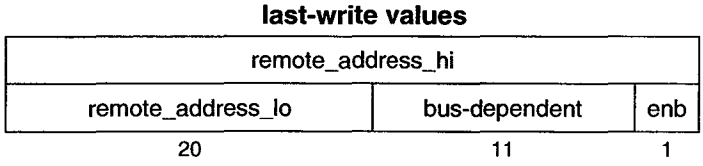



**Figure 36—ARGUMENT register remote address format**

Software is expected to load the ARGUMENT_HI and ARGUMENT_LO registers with a 64-bit address. However, if the node only supports 32-bit addressing, the upper half of the 64-bit address is ignored. This remote address space is expected to support the noncoherent read and write transactions specified in table 3.

The value of the enable remote access bit, ***enb***, specifies whether remote bus transactions can be generated, using the address in the ARGUMENT_HI and ARGUMENT_LO registers.

### 7.4.9 TEST_START

A write to the TEST_START register initiates one of the node's built-in test sequences. A write to the TEST_START register shall immediately set the TEST_STATUS.*active* bit to one.

The TEST_START register includes a 4-bit test category field (***cat***), a 16-bit field that identifies a specific test sequence or test step (***test_step***), and a 3-bit field that specifies test options (***tops***), as illustrated in figure 37.

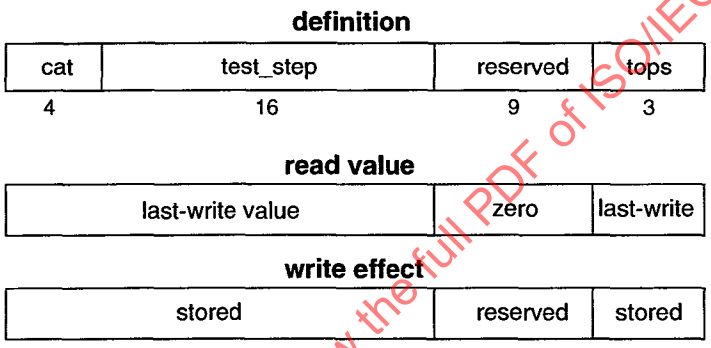



**Figure 37—TEST_START format**

**TEST_START:**

| | |
|---|---|
| Optional(RW) | Required to initiate extended, system, or manual tests. Also required to initiate initialization tests beyond those run during a command_reset and power_reset. |
| Initial value | Zero. |
| Read4 value | Non-reserved fields shall return the last-write value. |
| Write4 effects | (1) Shall be stored.<br>(2) The STATE_CLEAR.*state* field shall be set to the testing value.<br>(3) A test or a test-step activity shall be invoked. The *cat*, *test_step*, and *tops* fields are used as arguments for the test that is invoked. |

The *cat* and *tops* fields contain single-bit arguments for the test. A *cat* field value of zero has a special meaning; this is interpreted as a test-stop command (any tests that are running are stopped at the next available opportunity). The format of the argument bits within these fields (which have defined meanings when *cat* is nonzero) is illustrated in figure 38.

These argument bits and the *test_step* field are defined below:

***cat:***

| | |
|---|---|
| Required | |
| Effects(0) | Halts all tests. |
| Effects(not 0) | Starts tests, see following bit definitions for details. |

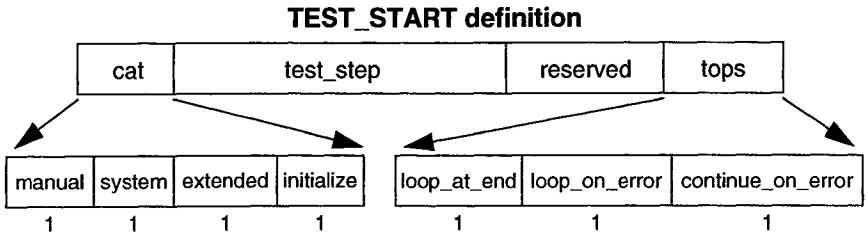



**Figure 38—*cat* and *tops* formats**

***manual:***

| | |
|---|---|
| Optional | Required to support manual tests. |
| Effects(0) | Manual tests shall not be performed. |
| Effects(1) | Manual tests should be performed. |

***system:***

| | |
|---|---|
| Optional | Required to support system tests. |
| Effects(0) | System tests shall not be performed. |
| Effects(1) | System tests should be performed. |

***extended:***

| | |
|---|---|
| Optional | Required to support extended tests. |
| Effect(0) | Extended tests shall not be performed. |
| (1) | Extended tests should be performed. |

***initialize:***

| | |
|---|---|
| Optional | Required to support initialization tests. |
| Effect(0) | Initialization tests shall not be performed. |
| (1) | Initialization tests should be performed. |

***test_step:***

| | |
|---|---|
| Optional | Required to select a specific test or set of tests to be run. |
| Effects | Shall specify which test sequence or group of sequences is run. The *test_step* values are vendor-dependent except when one or both of the *initialize* and *extended* bit are 1; for these exceptions the *test_step* values have special meanings, as specified in table 15. |

**Table 15—Special *test_step* values**

| test_step | Description |
|---|---|
| $0000_{16}$ | default tests shall be run. |
| - other - | vendor-dependent definitions. |
| $FFFF_{16}$ | all tests in this category should be run. |

***loop_at_end:***

| | |
|---|---|
| Optional | Required for repeating the requested test sequence automatically (restart from the beginning). |
| Effects(0) | Shall be ignored if *loop_on_error* is 1 or *continue_on_error* is 0. Otherwise, shall stop testing at the end of the test sequence. |
| Effects(1) | Shall be ignored if *loop_on_error* is 1 or *continue_on_error* is 0. Otherwise, shall restart the test sequence at the end of the test sequence. |

***loop_on_error:***

| | |
|---|---|
| Optional | Required for looping on the failed portion of a test sequence automatically after one or several errors have been detected. |
| Effects(0) | Action after an error shall be specified by *continue_on_error*. |
| Effects(1) | Shall repeatedly rerun the portion of the sequence that failed. |

***continue_on_error:***

| | |
|---|---|
| Optional | Required for continuing the requested test sequence automatically after one or several errors have been detected. |
| Effects(0) | Shall be ignored if loop_on_error is 1.<br>Otherwise, shall halt testing when an error is detected. |
| Effects(1) | Shall be ignored if loop_on_error is 1.<br>Otherwise, should continue testing after an error is detected. |

Tests are allowed to ignore any of the bits within this register. However, if any bit or field is ignored, the test shall behave as though the bit or field were zero.

Combinations of bits are expected to be set in the test options field. For example, a temporary (intermittent or transient) fault may only be detected after running a test sequence many times. To cause the test to capture the symptoms of the infrequent temporary fault, the operator would set *loop_at_end* and no other bit in the *tops* field. Table 16 describes the interpretation of the *tops* field for all single and multiple bit-set values.

**Table 16—Expected interpretation of *tops* values**

| tops | Expected interpretation |
|---|---|
| $000_2$ | In the absence of an error, testing shall stop at the end of the test sequence. When an error is detected, testing shall halt. (!*loop_at_end* && !*loop_on_error* && !*continue_on_error*) |
| $001_2$ | Testing shall stop at the end of the test sequence. (!*loop_at_end* && !*loop_on_error* && *continue_on_error*) |
| $010_2$<br>$011_2$ | In the absence of an error, testing shall stop at the end of the test sequence. When an error is detected, testing shall loop, repeating the failed portion of the test sequence. (!*loop_at_end* && *loop_on_error*) |
| $100_2$ | In the absence of an error, testing shall repeat the test sequence after reaching the end of the test sequence. When an error is detected, testing shall halt. ( *loop_at_end* && !*loop_on_error* && !*continue_on_error*) |
| $101_2$ | Testing shall repeat the test sequence after reaching the end of the test sequence. ( *loop_at_end* && !*loop_on_error* && *continue_on_error*) |
| $110_2$<br>$111_2$ | In the absence of an error, testing shall repeat the test sequence after reaching the end of the test sequence. When an error is detected, testing shall loop, repeating the failed test. ( *loop_at_end* && *loop_on_error*) |

### 7.4.10 TEST_STATUS

A read of the TEST_STATUS register provides access to the node's current test state values. Its format is dependent on the value of the *test_state* field, as illustrated in figure 39. Software is not expected to write to the TEST_STATUS register during normal system operation.

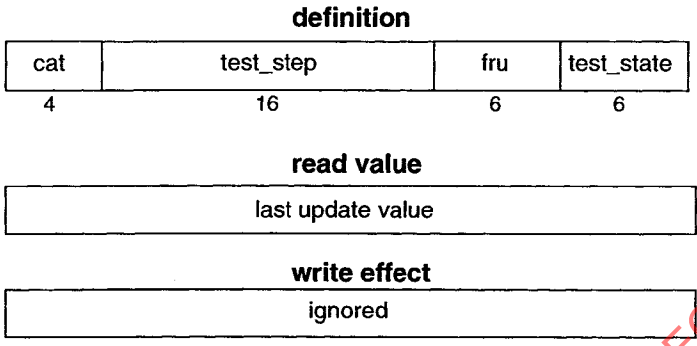



**Figure 39—TEST_STATUS format**

**TEST_STATUS:**

| | |
|---|---|
| Optional(RO) | Required if TEST_START is implemented. |
| Initial value | Shall return zero. |
| Read4 value | Shall return the last-update value. |
| Write4 effect | Shall be ignored. |

The format and meaning of bits within the *cat* fields in TEST_START and TEST_STATUS are the same. The ***test_state*** field provides 6 bits of status information, as illustrated in figure 40.

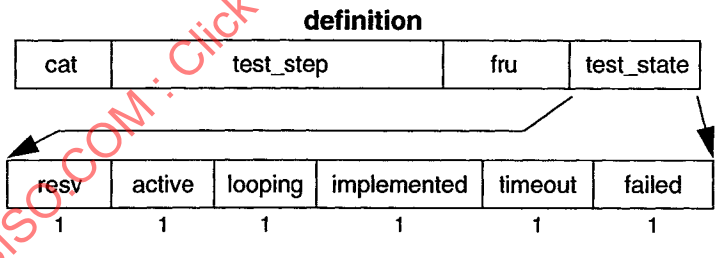



**Figure 40—*test_state* format**

After a write to the TEST_START register, the testing phase is either active, successful, or unsuccessful. The testing phase is specified by the values of the TEST_STATUS.*active* field and the logical OR of the TEST_STATUS.*timeout* and TEST_STATUS.*failed* values, as illustrated in table 17.

Within the context of table 17, a nonfatal error allows the test to complete as specified by this standard. Fatal errors are those that halt the test immediately, without updating the value of the TEST_STATUS register. Software is expected to detect fatal errors by timing the length of active tests; a fatal error is assumed after a timeout interval has passed. For that reason, the length of several standard tests is constrained to not exceed 10 s (see 5.3 for details).

The definitions of the TEST_STATUS fields are dependent on the test phase, as specified below.

**Table 17—Test status value**

| Testing phase | *active* | *timeout* or *failed* | Description |
|---|---|---|---|
| active | 1 | 0,1 | testing being performed |
| successful | 0 | 0 | testing completed, no errors |
| unsuccessful | 0 | 1 | testing completed, nonfatal error(s) |

***cat:***

| | |
|---|---|
| Optional | Useful for identifying the type of test that failed. |
| Active | Should identify the type of test that is being performed. |
| Successful | Should identify the type of test that detected the error. |
| Unsuccessful | Vendor-dependent. |

***test_step:***

| | |
|---|---|
| Optional | Useful for identifying the specific test step that failed. |
| Active | Shall be updated at least once every 10 s, with different, nonrecurring values. |
| Successful | Should identify the final test step in the test sequence. |
| Unsuccessful | Should identify the test step that detected the failure. |

***fru:***

| | |
|---|---|
| Optional | Useful for identifying the field-replaceable unit that failed. |
| Active | Should identify the unit under test. |
| Successful | Vendor-dependent. |
| Unsuccessful | Should help identify the failed field-replaceable unit. |

***active:***

| | |
|---|---|
| Required | |
| Value(0) | Shall mean a test was successful or unsuccessful. |
| Value(1) | Shall mean a test is active. |

***looping:***

| | |
|---|---|
| Optional | Required if TEST_START.*loop-on-error* is implemented. |
| Active(0) | Shall indicate the testing continues. |
| Active(1) | Shall indicate testing is looping on a failed test. |
| Successful | Vendor-dependent. |
| Unsuccessful | Vendor-dependent. |

***implemented:***

| | |
|---|---|
| Required | Confirms that the selected test is implemented. |
| Active(0) | Shall indicate that the specified test might not be implemented. |
| Active(1) | Shall indicate that the specified test is implemented. |
| Successful(0) | Shall indicate that the specified test is not implemented. |
| Successful(1) | Shall indicate that the specified test is implemented. |
| Unsuccessful(0) | Shall be an illegal value. |
| Unsuccessful(1) | Shall indicate the specified test is implemented. |

***timeout:***

| | |
|---|---|
| Optional | Required if errors are detected by timeouts. |
| Active(0) | Vendor-dependent if *looping* is 1. If *looping* is 0:<br>Shall indicate no timeout errors have been detected. |
| Active(1) | Vendor-dependent if *looping* is 1. If *looping* is 0:<br>Shall indicate one or more timeout errors have been detected. |
| Successful | Not applicable (zero by definition). |
| Unsuccessful | One or more errors have been detected by a timeout. |

***failed:***

| | |
|---|---|
| Optional | Required if errors are detected by fault (non-timeout) mechanisms. |
| Active(0) | Vendor-dependent if *looping* is 1. If *looping* is 0:<br>Shall indicate no fault errors have been detected. |
| Active(1) | Vendor-dependent if *looping* is 1. If *looping* is 0:<br>Shall indicate one or more fault errors have been detected. |
| Successful | Not applicable (zero by definition). |
| Unsuccessful | One or more errors have been detected by a timeout. |

When tests have completed (either successfully or unsuccessfully), the 5 defined bits in the 6-bit *test_state* field shall provide an overview of how the test completed, as specified in table 18.

**Table 18—Test status values (successful and unsuccessful phases)**

| *test_state* | Name | Description |
|---|---|---|
| $00000_2$ | not implemented | Successful unimplemented test. |
| $00001_2$–$00011_2$ | illegal | Unimplemented tests do not have errors. |
| $00100_2$ | implemented | Successful implemented test. |
| $00101_2$ | failed | Unsuccessful test, fault error. |
| $00110_2$ | timeout | Unsuccessful test, timeout error. |
| $00111_2$ | failed/timeout | Unsuccessful test, timeout&fault errors. |
| $01000_2$–$01111_2$ | illegal | Completed tests are not looping. |

When tests are active, the five LSBs of the *test_state* field shall provide an overview of how the test is progressing, as specified in table 19.

### 7.4.11 UNITS_BASE

For some unit architectures (such as graphics frame buffers), the size of the node's initial address space may be insufficient. A node can be assigned additional address space for its unit architectures by writes to the UNITS_BASE and UNITS_BOUND registers. These writes set the base address and bound of the extended units address space. Multiple units may be mapped to non-overlapping portions of the extended units address space.

Software is not expected to otherwise access the node while the UNITS_BASE or UNITS_BOUND registers are being changed; in a split-response bus, such accesses may have a bus-dependent effect on request subactions that have been queued but have not been processed.

The UNITS_BASE registers are either 32 bits or 64 bits, as illustrated in figure 41.

**Table 19—Test status values (active phase)**

| *test_state* | Name | Description |
|---|---|---|
| $10000_2$ | checking | Test implementation has not been verified. |
| $10001_2$–$10011_2$ | illegal | Unimplemented tests do not have errors. |
| $10100_2$ | implemented | Test implementation verified. |
| $10101_2$ | failed | Fault error detected, testing continues. |
| $10110_2$ | timeout | Timeout error detected, testing continues. |
| $10111_2$ | failed/timeout | Timeout and fault errors detected, testing continues. |
| $11000_2$–$11011_2$ | illegal | Unimplemented tests do not loop. |
| $11100_2$ | looping | Initial tests complete, test is now looping. |
| $11101_2$–$11111_2$ | looping on error | Error detected, test is now looping. |

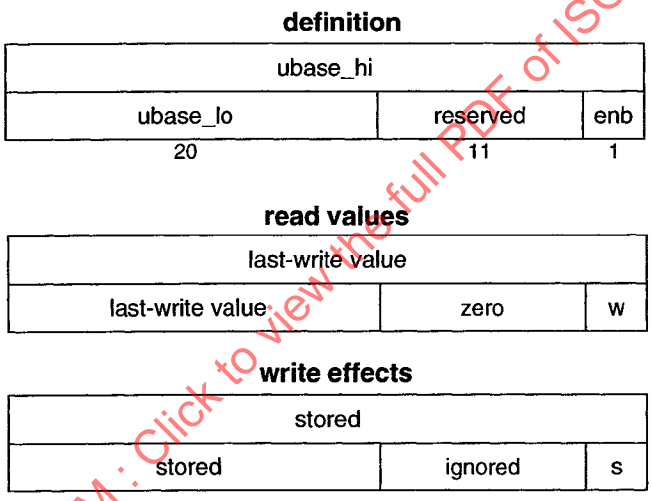



**Figure 41—UNITS_BASE formats**

**UNITS_BASE_HI:**

| | |
|---|---|
| Optional(RW) | Required on 64-bit nodes if UNITS_BASE_LO is implemented. |
| Initial value | Zero. |
| Read4 value | The last-write value shall be returned. |
| Write4 effect | Shall be stored. |

**UNITS_BASE_LO:**

| | |
|---|---|
| Optional(RW) | Required for supporting extended units space. |
| Initial value | Zero. |
| Read4 value | For the nonreserved fields, the last-write value shall be returned. |
| Write4 effect | For the nonreserved fields, shall be stored. |

Within the UNITS_BASE_LO register, the following fields are defined:

***ubase_lo:***

| | |
|---|---|
| Required | Specifies the least-significant bits of the extended units space. |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. The stored value specifies (part or all of) the base address for the extended units space. |

***enb:***

| | |
|---|---|
| Required | Selectively enables the extended units space. |
| Initial value | Shall be zero. |
| Read value | Shall return the last-write value. |
| Write effect | Shall be stored. The extended units space is only enabled when STATE_CLEAR.*lost* is 0 and *enb* is 1. |

### 7.4.12 UNITS_BOUND

The UNITS_BOUND registers specify an address that is one more than the largest address within the node's extended units space. The UNITS_BOUND_HI and UNITS_BOUND_LO register formats support 64-bit addresses, as illustrated in figure 42.

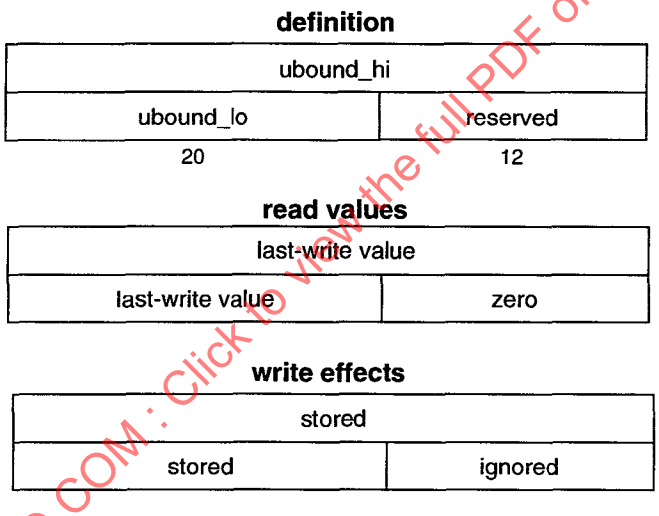



**Figure 42—UNITS_BOUND formats**

**UNITS_BOUND_HI:**

| | |
|---|---|
| Optional(RW) | Required if UNITS_BASE_HI is implemented. |
| Initial value | Zero. |
| Read4 value | Shall return last-write value. |
| Write4 effect | Shall be stored. |

**UNITS_BOUND_LO:**

| | |
|---|---|
| Optional(RW) | Required if UNITS_BASE_LO is implemented. |
| Initial value | Zero. |
| Read4 value | For nonreserved fields, shall return the last-write value. |
| Write4 effect | For nonreserved fields, shall be stored. |

Within the UNITS_BOUND_LO register, the following field is defined:

***ubound_lo:***

| | |
|---|---|
| Required | |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. The stored value partially or fully specifies the next larger address after the extended units space. |

When the value of base is equal to UNITS_BASE and bound is equal to UNITS_BOUND and these registers do not conflict with the node's initial node or extended memory spaces, table 20 specifies which addresses are within the node's extended units space.

**Table 20—Extended space alignment and size**

```
/* Returns 1 if address is within the node's extended address space.
 * Depending on the node architecture, int is 32 bits or 64 bits. */
#define ENB 1
ExtendedAddress(int address, int base, int bound)
{
  if ((base & ENB) == 0)                    /* Address space is */
    return (0);                             /* enabled when ENB is 1 */
  if (address < (base & ~0XFFF))            /* Ignore 12 LSBs */
    return (0);                             /* Below extended space */
  if (address >= (bound & ~0XFFF))          /* Ignore 12 LSBs */
    return (0);                             /* Above extended space */
  return (1);                               /* Within extended space */
}
```

### 7.4.13 MEMORY_BASE

A node is assigned additional address space for its memory controller by writes to the MEMORY_BASE and MEMORY_BOUND registers. These writes set the base address and bound of the extended memory address space. The MEMORY_BASE_HI and MEMORY_BASE_LO register formats support 64-bit addresses, as illustrated in figure 43.

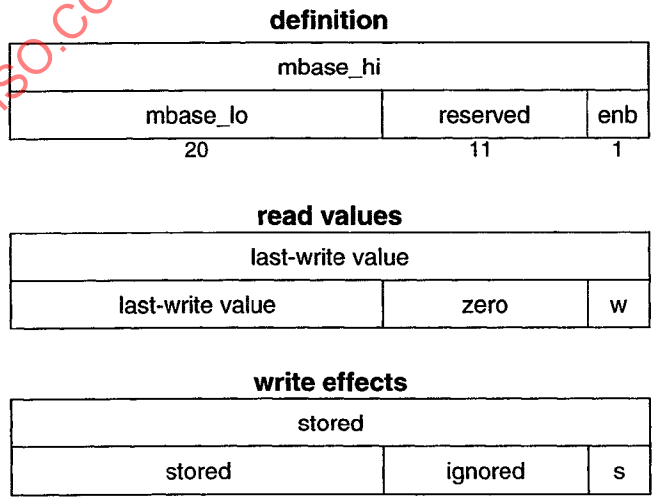



**Figure 43—MEMORY_BASE formats**

Software is not expected to otherwise access the node while the MEMORY_BASE or MEMORY_BOUND registers are being changed; on a split-response bus, such accesses may have a bus-dependent effect on request subactions that have been queued but have not been processed.

**MEMORY_BASE_HI:**

| | |
|---|---|
| Optional(RW) | Required on 64-bit nodes if MEMORY_BASE_LO is implemented. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. |

**MEMORY_BASE_LO:**

| | |
|---|---|
| Optional (RW) | Required by memory-controllers units with 32/64-bit extended addressing. |
| Read4 value | For nonreserved fields, shall return the last-write value. |
| Write4 effect | For nonreserved fields, shall be stored. |

Within the MEMORY_BASE_LO register, the following fields are defined:

***mbase_lo:***

| | |
|---|---|
| Required | |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. The value specifies (part or all of) the base address for the extended memory space. |

***enb:***

| | |
|---|---|
| Required | This bit shall be implemented. |
| Initial value | Shall be zero. |
| Read value | Shall return the last-write value. |
| Write effect | Shall be stored. The extended memory space is only enabled when STATE_CLEAR.*lost* is 0 and *enb* is 1. |

### 7.4.14 MEMORY_BOUND

The MEMORY_BOUND registers specify an address that is one more than the largest byte address within the node's extended memory space. The MEMORY_BOUND_HI and MEMORY_BOUND_LO register formats support 64-bit addresses, as illustrated in figure 44.

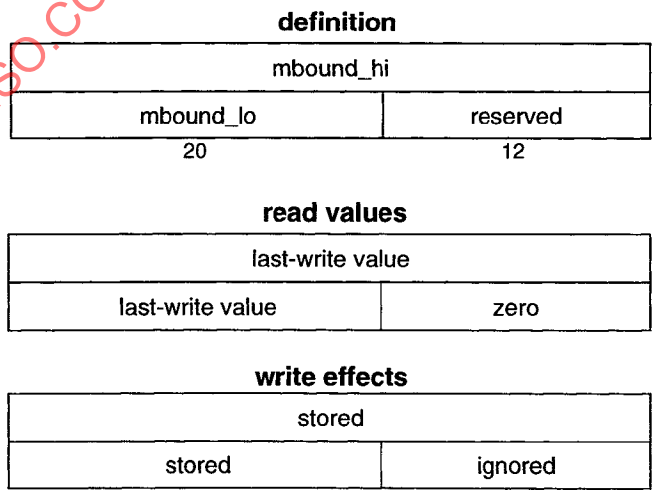



**Figure 44—MEMORY_BOUND formats**

**MEMORY_BOUND_HI:**

| | |
|---|---|
| Optional(RW) | Required if MEMORY_BASE_HI is implemented. |
| Read4 value | Shall return last-write value. |
| Write4 effects | Shall be stored. |

**MEMORY_BOUND_LO:**

| | |
|---|---|
| Optional (RO) | Required if MEMORY_BASE_LO is implemented. |
| Read4 value | Nonreserved fields shall return the last-write value. |
| Write4 effects | Nonreserved fields shall be stored. |

Within the MEMORY_BOUND_LO register, the following field is defined:

***mbound_lo:***

| | |
|---|---|
| Required | |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. This value specifies (part or all of) the size of the extended memory space. |

When the value of base is equal to MEMORY_BASE and bound is equal to MEMORY_BOUND and these registers do not conflict with the node's initial node or extended units spaces, table 20 specifies which addresses are within the node's extended memory space.

### 7.4.15 INTERRUPT_TARGET

The node's INTERRUPT_TARGET register provides an address for nodecasting interrupts to all units on the node. A write to this target address is distributed to all units on the node. Software is not expected to read the INTERRUPT_TARGET register during normal system operation.

The write4 transaction data correspond to 32 interrupt-event priorities, where the most- through least-significant bits of the data correspond to the highest-through-lowest-priority interrupt event, respectively. When responding to these writes, units are expected to provide bits to save the 32 interrupt events. When all 32 bits of this register are implemented, each bit shall correspond to an event type, and each occurrence of an event shall set the corresponding bit in the register.

When the INTERRUPT_TARGET register is mapped to a unit with less than 32 interrupt priority levels, each priority bit in the unit shall be mapped to a contiguous range of bits within the INTERRUPT_TARGET register, the mapping shall be monotonic (higher priority interrupt bits should be mapped to more-significant bits within the INTERRUPT_TARGET register), and all of the INTERRUPT_TARGET bits shall be mapped to a unit interrupt bit.

When the INTERRUPT_TARGET register is written, the write-data is ANDed with the current value of the INTERRUPT_MASK register, and the result is distributed to the units on the node. The format of the INTERRUPT_TARGET register is illustrated in figure 45.

**INTERRUPT_TARGET:**

| | |
|---|---|
| Optional(WO) | Required to support nodecast interrupts. |
| Initial value | Zero. |
| Read4 value | Shall return zero. |
| Write4 effect | The write-data value shall be ANDed with the value of the INTERRUPT_MASK register; the resulting data shall be immediately nodecast to the units. |

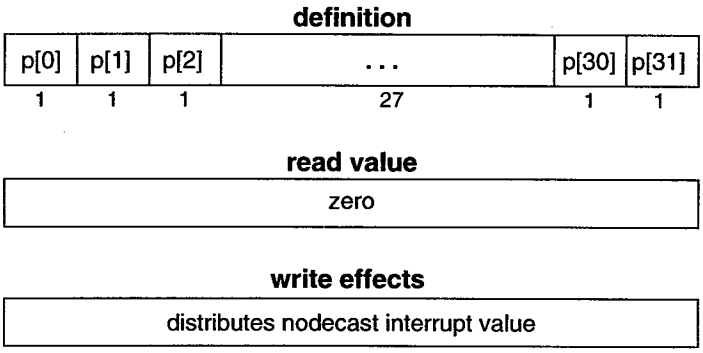



**Figure 45—INTERRUPT_TARGET format**

### 7.4.16 INTERRUPT_MASK

The INTERRUPT_MASK register is used to selectively enable the INTERRUPT_TARGET's interrupt bits. The format of the corresponding INTERRUPT_MASK register is shown in figure 46.

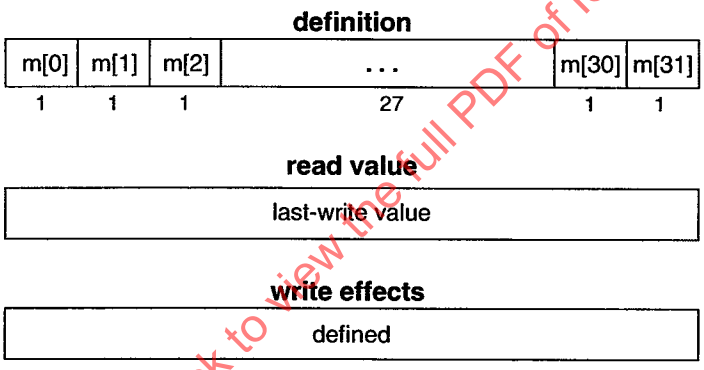



**Figure 46—INTERRUPT_MASK format**

**INTERRUPT_MASK:**

| | |
|---|---|
| Optional(WO) | Required if INTERRUPT_TARGET is implemented. |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. |

Note that the INTERRUPT_MASK register is only used as a mask for writes to the INTERRUPT_TARGET register; a write to the INTERRUPT_MASK register **does not** have the effect of forwarding interrupt data to the units on the node.

### 7.4.17 CLOCK_VALUE

The optional CLOCK_VALUE registers provide access to the 64 most-significant bits of an internal clock_value register. Software is expected to read these registers to determine the current time. Software is only expected to write to these registers when the global system time is being initialized. The most-significant register holds the integer portion of time, which is measured in seconds. The least-significant register provides the fractions-of-a-second portion.

The CLOCK_VALUE register may be read-only or read/write; the format of the read-only version is illustrated in figure 47.

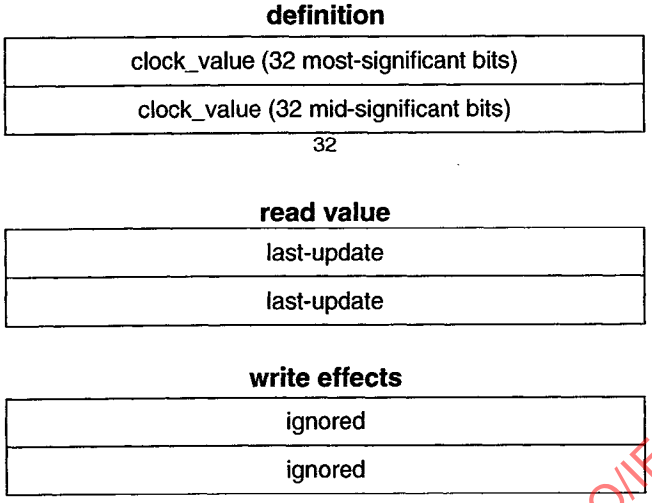



**Figure 47—CLOCK_VALUE formats (read-only)**

**CLOCK_VALUE_HI:**

| | |
|---|---|
| Optional(RO) | Required if CLOCK_VALUE_MID is implemented. |
| Initial value | Zero. |
| Read4 value | Shall return the last-update value. |
| Write4 effect | Shall be ignored. |

**CLOCK_VALUE_MID:**

| | |
|---|---|
| Optional(RO) | Required for some clock unit architectures. |
| Initial value | Zero. |
| Read4 value | Shall return the last update value. |
| Write4 effect | Shall be ignored. |

The format of the read/write version of the CLOCK_VALUE register is illustrated in figure 48.

**CLOCK_VALUE_HI:**

| | |
|---|---|
| Optional(RW) | Required if CLOCK_VALUE_MID is implemented. |
| Initial value | Zero. |
| Read4 value | Shall return the most recent of the last-update and last-write values. |
| Write4 effect | Shall be stored. |

**CLOCK_VALUE_MID:**

| | |
|---|---|
| Optional(RW) | Required for some clock-unit architectures. |
| Initial value | Zero. |
| Read4 value | Shall return the most recent of the last-update and last-write values. |
| Write4 effect | Shall be stored. |

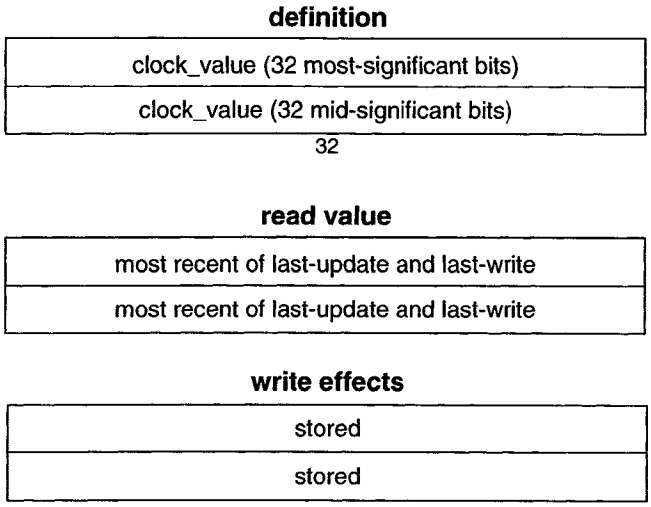



**Figure 48—CLOCK_VALUE formats (read/write)**

### 7.4.18 CLOCK_TICK_PERIOD

The CLOCK_TICK_PERIOD registers determine the amount by which the 96-bit clock-timer is incremented on each cycle of its oscillator. When implemented, software is expected to write to the CLOCK_TICK_PERIOD registers to control the rate of increase of the clock_value register. The format of these registers is illustrated in figure 49.

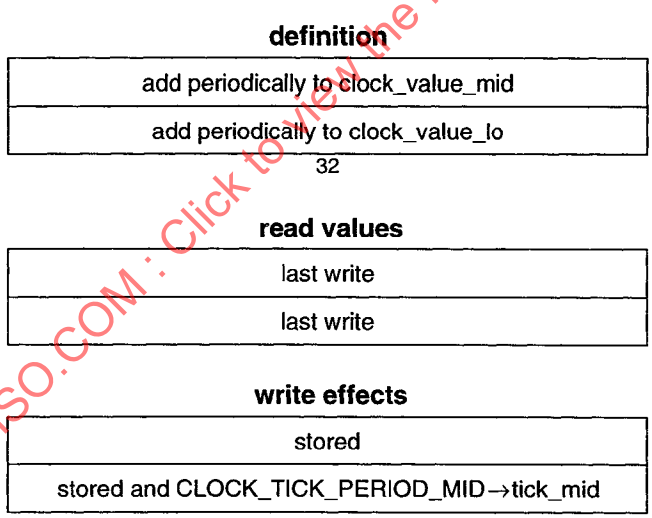



**Figure 49—CLOCK_TICK_PERIOD formats**

**CLOCK_TICK_PERIOD_MID:**

| | |
|---|---|
| Optional(RW) | Required if CLOCK_TICK_PERIOD_LO is implemented. |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effect | Shall be stored. |

**CLOCK_TICK_PERIOD_LO:**

| | |
|---|---|
| Required(RW) | Required for some clock-unit architectures. |
| Initial value | Zero. |
| Read4 value | Shall return the last-write value. |
| Write4 effects | 1) Shall be stored.<br>2) The value of the CLOCK_TICK_PERIOD_MID register is simultaneously transferred to the internal tick_mid register. |

### 7.4.19 CLOCK_STROBE_ARRIVED

The CLOCK_VALUE_HI and CLOCK_VALUE_MID registers are simultaneously transferred to the CLOCK_STROBE_ARRIVED_HI and CLOCK_STROBE_ARRIVED_MID registers during the time that a bus-dependent clock_strobe is observed. Software is expected to read these registers when the distributed clocks are being synchronized. Software is not expected to write these registers during normal system operation.

The CLOCK_STROBE_ARRIVED registers may be implemented as read-only or read/write register pairs. The format of the read-only version of these registers is shown in figure 50.

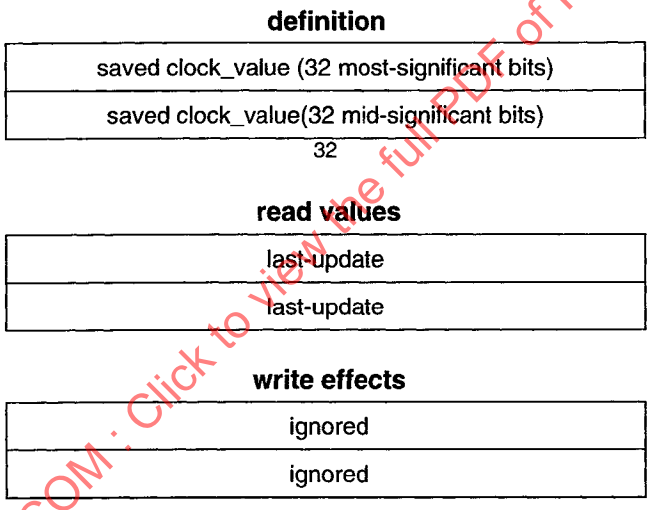



**Figure 50—CLOCK_STROBE_ARRIVED formats (read-only)**

**CLOCK_STROBE_ARRIVED_HI:**

| | |
|---|---|
| Optional(RO) | Required if CLOCK_STROBE_ARRIVED_MID is implemented. |
| Initial value | Zero. |
| Read4 value | Shall return the last-update value. |
| Write4 effect | Shall be ignored. |

**CLOCK_STROBE_ARRIVED_MID:**

| | |
|---|---|
| Optional(RO) | Required by some clock-unit architectures. |
| Initial value | Zero. |
| Read4 value | Shall return the last-update value. |
| Write4 effect | Shall be ignored. |

The format of the read/write version of the CLOCK_STROBE_ARRIVED registers is in figure 51.

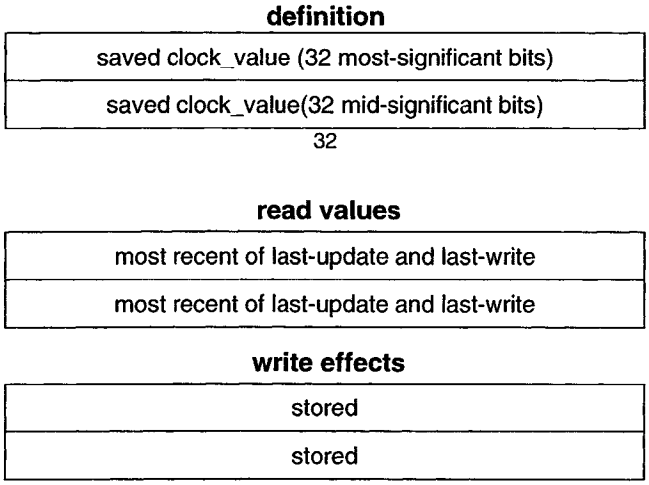



**Figure 51—CLOCK_STROBE_ARRIVED formats (read/write)**

**CLOCK_STROBE_ARRIVED_HI:**

| | |
|---|---|
| Optional(RW) | Required if CLOCK_STROBE_ARRIVED_MID is implemented. |
| Initial value | Zero. |
| Read4 value | Shall return the most recent of the last-write or last-update values. |
| Write4 effect | Shall be stored. |

**CLOCK_STROBE_ARRIVED_MID:**

| | |
|---|---|
| Optional(RW) | Required by some clock-unit architectures. |
| Initial value | Zero. |
| Read4 value | Shall return the most recent of the last-write or last-update values. |
| Write4 effect | Shall be stored. |

Writes to the CLOCK_STROBE_ARRIVED_HI and CLOCK_STROBE_ARRIVED_MID registers are only expected to be used for diagnostic purposes, since these registers are updated by the clock_strobe signal during normal system operation.

### 7.4.20 CLOCK_STROBE_INFO

The CLOCK_STROBE_INFO0 through CLOCK_STROBE_INFO3 registers are bus-dependent quadlet registers. For example, a backplane bus could use these registers to transmit the time associated with the previous strobe and (on a bus-bridge or point-to-point interconnect) the registers could log the time taken to forward the strobe through the node. The format and detailed definition of these registers is bus-dependent.

### 7.4.21 Message targets

The two 64-byte-aligned message addresses, MESSAGE_REQUEST and MESSAGE_RESPONSE, provide target addresses for broadcast and nodecast message transactions.

Logically, the message-passing window (these register addresses) is part of the node, but the messages are processed or filtered (selectively discarded, based on their content) by the units. Physically, either the node or the units may do the message filtering.

The MESSAGE_REQUEST and MESSAGE_RESPONSE registers respond to at least aligned 64-byte write transactions. Support of other transaction sizes and unaligned addresses is bus-dependent.

**MESSAGE_REQUEST:**

| | |
|---|---|
| Optional (WO) | Required for supporting nodecast or broadcast message passing. |
| Read4 values | A read shall return zeros. |
| Write64 effect | If message-passing request-queue space is available, the transaction's data are saved and a *done_correct* status is returned. If request-queue space is not available, an error-status code (*conflict_error*) is returned. |

**MESSAGE_RESPONSE:**

| | |
|---|---|
| Optional (WO) | Required for supporting nodecast or broadcast message passing. |
| Read4 values | A read shall return zeros. |
| Write64 effect | If message-passing response-queue space is available, the transaction's data are saved and a *done_correct* status is returned. If response-queue space is not available, an error-status code (*conflict_error*) is returned. |

### 7.4.22 ERROR_LOG registers

The error log contains 128 bytes of bus-dependent registers. Node hardware is expected to update these registers when a node-related error is detected; the ERROR_LOG registers shall be updated before the STATE_CLEAR.*elog* bit is set to one. The subsequent errors are not expected to overwrite the critical error-log fields until the STATE_CLEAR.*elog* bit has been cleared. The format of these registers and the conditions under which they are updated are bus-dependent.

Operating system software is expected to read these registers and save their contents in system memory. Software is expected to clear the STATE_CLEAR.*elog* bit after reading the ERROR_LOG_BUFFER registers. Bus standards are expected to provide overflow bits or error count fields, to detect duplicate errors that could not be properly logged.

The error log minimizes the need to provide special error-related errors in the STATE_CLEAR register; the standard location for the error log simplifies error-logging software, which is expected to copy the data from the error log to a system-memory log after the STATE_CLEAR.*elog* bit is set.

This page intentionally left blank

# 8. ROM specification

## 8.1 Introduction

### 8.1.1 ROM design assumptions

The CSR ROM Directory structure is based upon several assumptions:

a) The portion of the ROM containing critical boot information should be easily accessed by initialization firmware or software.
b) The ROM is expected to be scanned during system initialization, to identify the node and to identify the appropriate driver software to be loaded.
c) A minimal ROM structure is necessary for inexpensive nodes or for vendors who wish to define their own data structures.
d) Optional ROM structures should support large numbers of bus-, unit-, or vendor-dependent parameters.

Note that the term ROM does not imply a physical implementation; see 1.4 for the definition of this term.

### 8.1.2 ROM formats

Two ROM formats are supported: minimal and general. A minimal ROM format provides a 24-bit vendor identifier; the vendor may provide additional vendor-dependent information.

The general ROM format provides additional information in a bus_info_block and a root_directory containing entries. Each entry within the root_directory may provide information, or may contain a pointer to another directory (which has the same structure as the root_directory) or a leaf (which contains information).

The unit directories contain information about the units, such as their software version number and their location within the node's address space. These directories and leaves are illustrated in figure 52.

### 8.1.3 Driver and diagnostic identifiers

The ROM structures provide identifiers for associating the appropriate I/O driver software and diagnostic software with a particular module, or a particular node or unit on the module. The identifiers and their hierarchical relationship to each other are illustrated in figure 53.

The arrows in figure 53 illustrate the default values for various company_id values. For example, when Node_Spec_Id is not provided, its assumed value shall be equal to Node_Vendor_Id. Similarly when Node_Vendor_Id is not provided, its assumed value shall be equal to Module_Vendor_Id.

The Module_Hw_Version and Node_Hw_Version values (when concatenated with their respective vendor identifier, such as Module_Vendor_Id) are expected to uniquely identify the appropriate diagnostic software for the module or node respectively. The unit-dependent diagnostics are expected to be contained within the node's diagnostic software.

The Module_Sw_Version, Node_Sw_Version, and the Unit_Sw_Version values (when concatenated with their respective specifier identifier, such as Module_Spec_Id) are expected to uniquely identify the appropriate I/O driver software for the module, node, or unit, respectively.

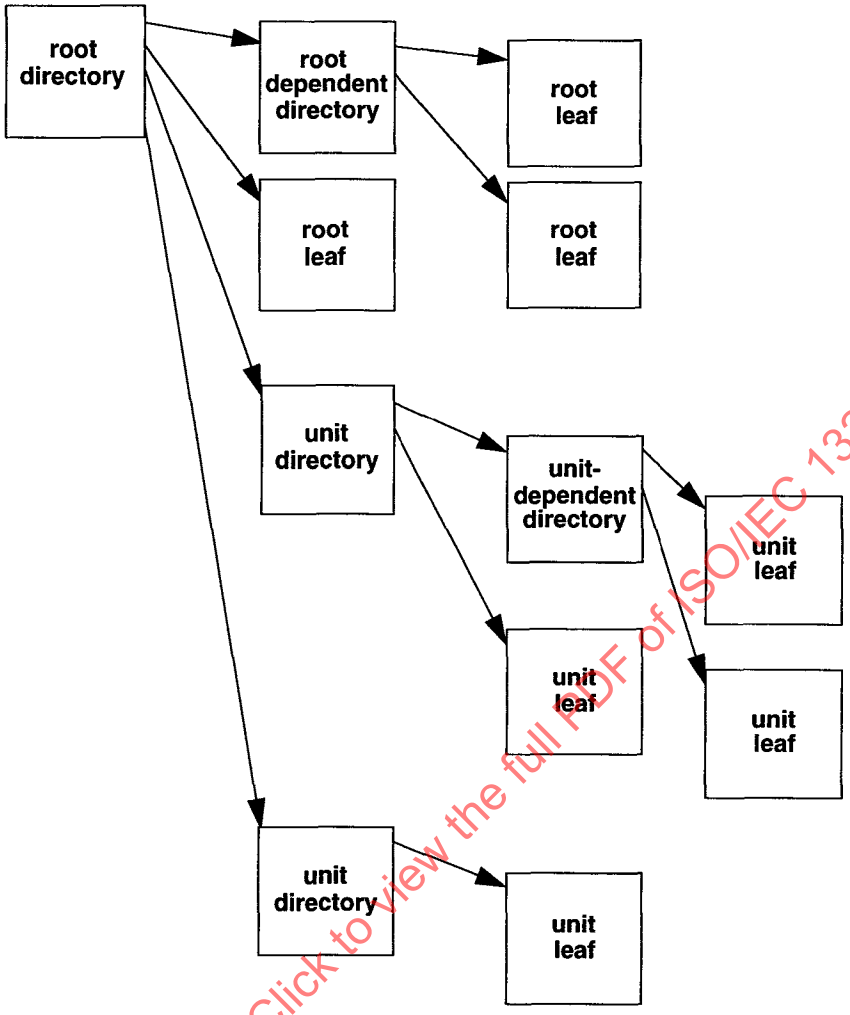



**Figure 52—ROM hierarchy**

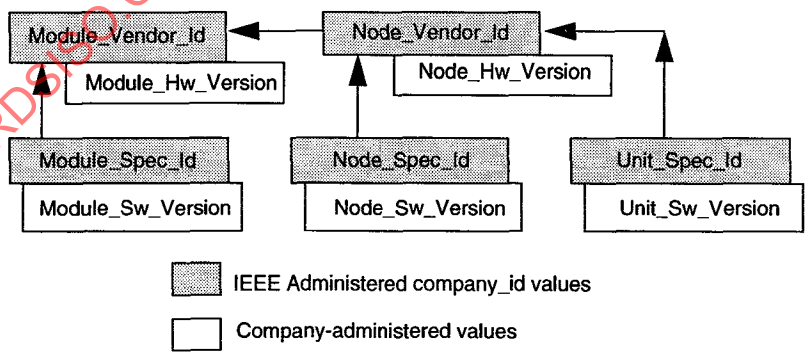



**Figure 53—Software identifier hierarchy**

### 8.1.4 ASCII text

The CSR Architecture defines a minimal ASCII subset for use within the ROM format and for simple, generic unit architectures that may be defined in the future. The minimal ASCII subset, specified in table 21, is derived from ISO/IEC 646:1991. Within this table, each of the 7-bit codes is followed by the associated character or control action.

**Table 21—ASCII subset (number values are hexadecimal**

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | 07 | BEL |
| 08 | BS | | | 0A | LF | | | 0C | FF | 0D | CR | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| 20 | SP | 21 | ! | 22 | " | | | | | 25 | % | 26 | & | 27 | ' |
| 28 | ( | 29 | ) | 2A | * | 2B | + | 2C | , | 2D | - | 2E | . | 2F | / |
| 30 | 0 | 31 | 1 | 32 | 2 | 33 | 3 | 34 | 4 | 35 | 5 | 36 | 6 | 37 | 7 |
| 38 | 8 | 39 | 9 | 3A | : | 3B | ; | 3C | < | 3D | = | 3E | > | 3F | ? |
| 40 | @ | 41 | A | 42 | B | 43 | C | 44 | D | 45 | E | 46 | F | 47 | G |
| 48 | H | 49 | I | 4A | J | 4B | K | 4C | L | 4D | M | 4E | N | 4F | O |
| 50 | P | 51 | Q | 52 | R | 53 | S | 54 | T | 55 | U | 56 | V | 57 | W |
| 58 | X | 59 | Y | 5A | Z | | | | | | | | | 5F | - |
| | | 61 | a | 62 | b | 63 | c | 64 | d | 65 | e | 66 | f | 67 | g |
| 8 | h | 69 | i | 6A | j | 6B | k | 6C | l | 6D | m | 6E | n | 6F | o |
| 70 | p | 71 | q | 72 | r | 73 | s | 74 | t | 75 | u | 76 | v | 77 | w |
| 78 | x | 79 | y | 7A | z | | | | | | | | | | |

See ISO/IEC 646:1991 for the definition of the characters and control actions. The backspace character encoding is provided for future unit architectures and shall not be used in the ROM format. All codes not explicitly defined are reserved.

The minimal ASCII subset for the ROM format complies with ISO/IEC 646:1991 with the exception that the following characters defined by ISO/IEC 646:1991 are not included in the ASCII subset of table 21:

a) Subclause 4.1.1, Transmission control characters.
b) Subclause 4.1.2, The format effectors HT and VT (horizontal & vertical tabulation).
c) Subclause 4.1.3, Code extension control characters.
d) Subclause 4.1.4, Device control characters.
e) Subclause 4.1.5, Information separators.
f) Subclause 4.1.6, Other control characters.
g) Subclause 4.3.2, Alternative graphics character allocations.
h) Subclause 6.4, IRV graphics character allocations.

When used within the context of a unit architecture, the expected display size is 80 characters by 24 lines.

### 8.1.5 CRC calculations

The CRC check is based on the ITU-T CRC-16 code (ITU-T Recommendation V.41 [B2]), which is performed on the most-significant bits first. This specification is based on the generation of a 16-bit CRC value (call this *check*). The CRC calculation shall start with an initially-zero *check* value and shall be successively

updated by each doublet protected by the CRC. The final check value provides the CRC value for the protected data.

The update algorithm shall calculate the new check doublet (C00–C15) based on the values of the previous check doublet (c00–c15) and the protected data value (d00–d15), using the algorithm provided by table 22. Note that a big-endian convention is used to label the bit positions; 00 labels the most-significant bit.

**Table 22—CRC-16 update algorithm**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C00 = | e04 | ⊕ | e05 | ⊕ | e08 | ⊕ | e12; | | | | | | | | |
| C01 = | e05 | ⊕ | e06 | ⊕ | e09 | ⊕ | e13; | | | | | | | | |
| C02 = | e06 | ⊕ | e07 | ⊕ | e10 | ⊕ | e14; | | | | | | | | |
| C03 = | e00 | ⊕ | e07 | ⊕ | e08 | ⊕ | e11 | ⊕ | e15; | | | | | | |
| C04 = | e00 | ⊕ | e01 | ⊕ | e04 | ⊕ | e05 | ⊕ | e09; | | | | | | |
| C05 = | e01 | ⊕ | e02 | ⊕ | e05 | ⊕ | e06 | ⊕ | e10; | | | | | | |
| C06 = | e00 | ⊕ | e02 | ⊕ | e03 | ⊕ | e06 | ⊕ | e07 | ⊕ | e11; | | | | |
| C07 = | e00 | ⊕ | e01 | ⊕ | e03 | ⊕ | e04 | ⊕ | e07 | ⊕ | e08 | ⊕ | e12; | | |
| C08 = | e00 | ⊕ | e01 | ⊕ | e02 | ⊕ | e04 | ⊕ | e05 | ⊕ | e08 | ⊕ | e09 | ⊕ | e13; |
| C09 = | e01 | ⊕ | e02 | ⊕ | e03 | ⊕ | e05 | ⊕ | e06 | ⊕ | e09 | ⊕ | e10 | ⊕ | e14; |
| C10 = | e02 | ⊕ | e03 | ⊕ | e04 | ⊕ | e06 | ⊕ | e07 | ⊕ | e10 | ⊕ | e11 | ⊕ | e15; |
| C11 = | e00 | ⊕ | e03 | ⊕ | e07 | ⊕ | e11; | | | | | | | | |
| C12 = | e00 | ⊕ | e01 | ⊕ | e04 | ⊕ | e08 | ⊕ | e12; | | | | | | |
| C13 = | e01 | ⊕ | e02 | ⊕ | e05 | ⊕ | e09 | ⊕ | e13; | | | | | | |
| C14 = | e02 | ⊕ | e03 | ⊕ | e06 | ⊕ | e10 | ⊕ | e14; | | | | | | |
| C15 = | e03 | ⊕ | e04 | ⊕ | e07 | ⊕ | e11 | ⊕ | e15; | | | | | | |

where:

| | |
|---|---|
| C00 – C15 | are the most- through-least-significant bits of the new check symbol |
| e00 – e15 | are the most- through-least-significant bits of an intermediate value: e15 = c15 ⊕ d15; e14 = c14 ⊕ d14; . . . e00 = c00 ⊕ d00; |
| d00 – d15 | are the most- through-least-significant bits of the data symbol |
| c00 – c15 | are the most- through-least-significant bits of the old check symbol |
| ⊕ | Exclusive OR |

The checking of a quadlet data value is performed by sequentially checking both doublets; the most-significant doublet is checked first. The equivalent CRC value can be calculated in an efficient nibble-sequential fashion, as illustrated in table 23. The CrcQuadlet routine is initially called with a check value of 0 and *data* is the first quadlet being checked. For additional data values *check* is the returned value from the previous call.

**Table 23—CRC-16 calculation routine**

```
/* CRC-16 calculation, performed on quadlets at a time */
Doublet
CrcQuadlet(unsigned data, unsigned check)
{
  int shift, sum, next;                          /* Integers are >= 32 bits */

  for (next = check, shift = 28; shift >= 0; shift -= 4) {
    sum = ((next >> 12) ^ (data >> shift)) & 0XF;
    next = (next << 4) ^ (sum << 12) ^ (sum << 5) ^ (sum);
  }
  return(next & 0XFFFF);                         /* 16-bit CRC value */
}
```

## 8.2 ROM formats

### 8.2.1 First ROM quadlet

The first (most-significant) byte of the first ROM quadlet is used to specify which ROM format is implemented. After a node is initialized, ROM access may be unavailable while the node is being tested; during this time, the quadlet shall be zero. When the ROM access is enabled, the first byte is 1 for the minimal ROM implementation option, and is greater than one for the general ROM option, as illustrated in figure 54.

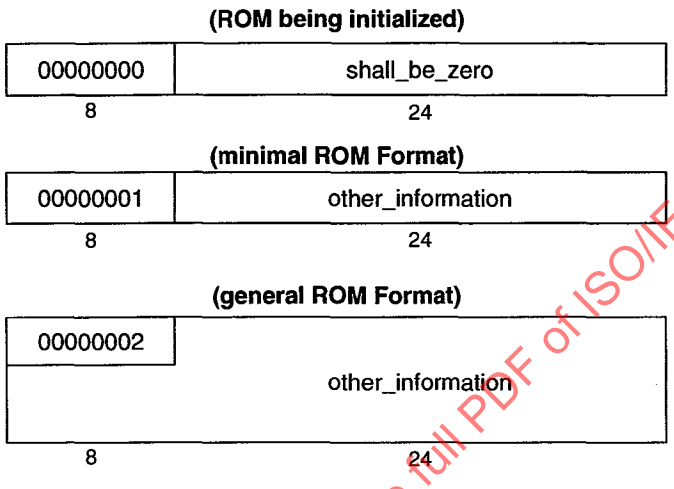



**Figure 54—First ROM quadlet**

### 8.2.2 Minimal ROM format

The minimal ROM implementation has a single quadlet, which provides a 24-bit *vendor_id* value as illustrated in figure 55.

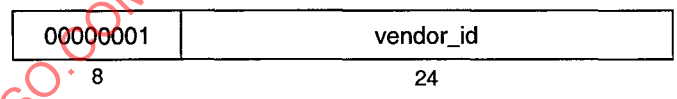



**Figure 55—Minimum ROM implementation**

The 24-bit immediate *vendor_id* value provides the 24-bit company_id of the vendor that manufactured the module. The company_id value is uniquely assigned to each module vendor by the IEEE; see 8.7. Note that the minimal ROM format does not preclude the presence of additional vendor-dependent information in the ROM.

### 8.2.3 General ROM format

In its more general form, the ROM directory structure is a hierarchy of information blocks; the blocks higher in the hierarchy point to the blocks beneath it. The locations of the initial blocks (which include the bus_info_block and root_directory) are fixed. The location of other entries (unit_directories, root and unit leaves) is vendor-dependent, but shall be specified by entries within the root_directory or its affiliated directories.

A general ROM implementation is shown in figure 56. Note that there may be multiple copies of the unit_subdirectories, one for each unit on the node.

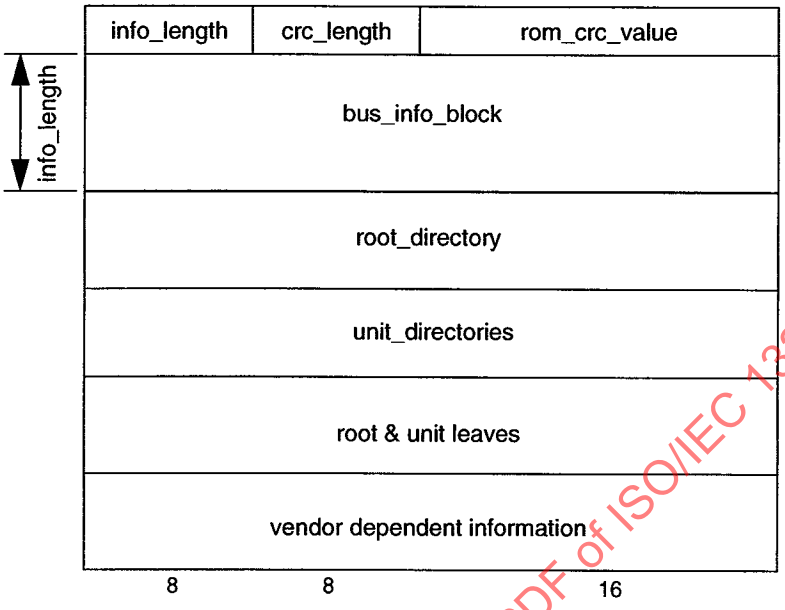



**Figure 56—Fully implemented CSR ROM directory**

The *info_length* byte (which is greater than one) distinguishes the general from the minimal ROM format (whose first byte equals one). The *info_length* value specifies the number of quadlets contained in the following bus_info_block data structure.

The *crc_length* parameter specifies how many of the following quadlets are protected by the *rom_crc_value* (the number of protected bytes is four times the *crc_length* value). For example, the *crc_length* field would be 11 if 44 bytes were protected. The minimum number of protected quadlets is the number within the bus_info_block.

The maximum number of protected bytes for *rom_crc_value* is 1020 bytes; the length field would equal 255 for this maximum coverage length. Thus, the entire ROM window within the node's initial address space can be protected by one CRC.

If the complete ROM is less than 1 kbyte in length, the initialization software is expected to check only the *rom_crc_value* protecting the entire ROM. However, the other CRC values (contained within the bus_info_block, the root_directory, directories, and leaves) may be checked to localize an error when the overall *rom_crc_value* is incorrect.

The *rom_crc_value* is calculated by the method described in 8.1.5.

### 8.2.4 Directory formats

All directories (including the root_directory) shall have the format specified in figure 57.

The directory_length parameter specifies the number of following quadlet entries in the directory; the total number of bytes in the directory is 4+(4 × directory_length). The CRC-16 value covers all of the following entries.

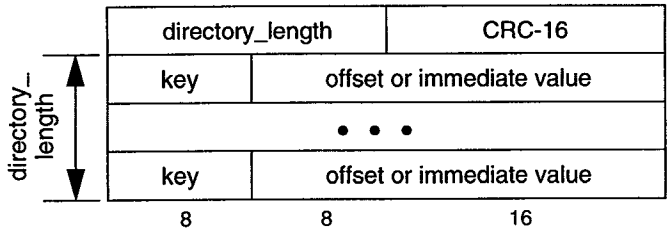



**Figure 57—Directory structure**

Each directory entry has the format shown in figure 58.

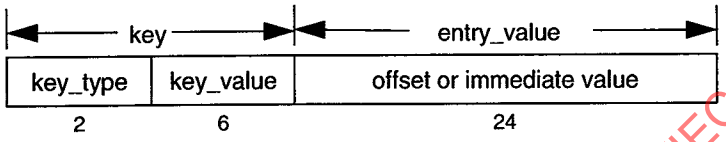



**Figure 58—Directory entry format**

The key is broken down into two fields. The *key_type* indicates the type of directory entry as shown in the table 24.

**Table 24—*key_type* definitions**

| Reference name | *key_type* | Meaning of least-significant 24-bits |
|---|---|---|
| immediate | 0 | immediate value |
| offset | 1 | initial-register-space offset for an immediate value |
| leaf | 2 | indirect-space offset for a leaf |
| directory | 3 | indirect-space offset for a directory |

The *key_value* specifies the particular directory entry, e.g., Module_Spec_Id, Unit_Sw_Version, etc.

For an **immediate** entry (*key_type* is 0), the *entry_value* shall be the 24-bit value for that directory entry, as illustrated in figure 59. Its meaning is dependent on the type of entry.


**Figure 59—Immediate entry format**

For an **offset** entry (*key_type* is 1), the *entry_value* shall contain an 24-bit offset field, as illustrated in figure 60. The offset value specifies a CSR address, as a quadlet offset from the base address of the initial register space.

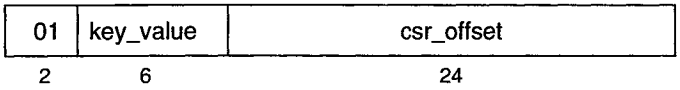



**Figure 60—Offset entry format**

For the **leaf** and **directory** entries (*key_type* is 2 or 3), the *entry_value* shall provide a 24-bit *indirect_offset* value, as illustrated in figure 61. The *indirect_offset* value specifies the the address of the leaf or directory in the indirect space.

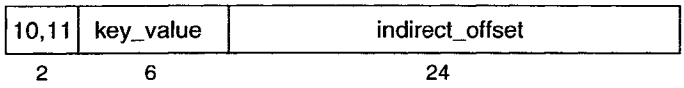



**Figure 61—Leaf and directory formats**

The *indirect_offset* value indirectly specifies the ROM-offset address of the leaf or directory entry (offsetOfValue) by specifying the number of quadlets between the entry point (offsetOfEntry) and the ROM-offset address. The relationship of these byte-address offsets within the indirect space is specified in table 25.

**Table 25—Calculating address of entry value**

```
/* Calculate ROM-offset address (offsetOfValue), based on
 * the address of the referencing entry (offsetOfEntry) and
 * the contents of the entry (indirect_offset). Note that
 * offsetOfValue and offsetOfEntry are integer multiples of 4 */

offsetOfValue = offsetOfEntry + 4 * indirect_offset;
```

The following text clarifies how data is fetched from ROM, once the offsetOfValue parameter is known. If the ROM-offset is less than 1024, a (properly adjusted) ROM-offset address shall be used to directly access the corresponding ROM location. If the ROM-offset is equal to or larger than 1024, the corresponding ROM location shall be accessed indirectly. The C code in table 26 illustrates the algorithm for performing these accesses.

### 8.2.5 Leaf format

All leaves have the format shown in figure 62.

The *leaf_length* parameter specifies the number of following quadlets in the leaf; the total number of bytes in the leaf is $4+(4 \times leaf\_length)$. The CRC-16 value covers all of the following quadlets.

### 8.2.6 Textual_descriptor

The Textual_Descriptor entry provides a textual descriptor of a ROM directory entry and may be used for describing a portion of a module, node, or unit. The Textual_Descriptor entry, if present, occurs directly after the directory entry that it describes. In figure 63, the first Textual_Descriptor is used to describe the Module_Dependent_Info entry and the second Textual_Descriptor is used to describe the Node_Software_Version entry. A Textual_Descriptor entry may follow any entry other than another Textual_Descriptor.

**Table 26—Fetching ROM from a desired offset address**

```
/*   Routine for accessing ROM, based on base address of the node's CSRs
 * csrBase    - base address of node being accessed
 * romOffset - offset address of Quadlet being accessed */
#define INDIRECT_ADDRESS 16
#define INDIRECT_DATA 20
Quadlet
FetchRomQuadlet(Quadlet csrBase, Quadlet romOffset)
{
  Quadlet *csrAddress, *regAddress, *regData;

  assert((romOffset % 4) == 0);                /* Offset is multiple of 4 */
  if (romOffset < 1024) {
    /* Internal address is mapped directly (but offset from csrBase) */
    csrAddress = (Quadlet *) ((char *)csrBase + 1024 + romOffset);
    return(*csrAddress);
  } else {
    /* Otherwise, address is mapped indirectly */
    regAddress = (Quadlet *) ((char *)csrBase + INDIRECT_ADDRESS);
    regData = (Quadlet *) ((char *)csrBase + INDIRECT_DATA);
    *regAddress = romOffset;
    return(*regData);
  }
}
```

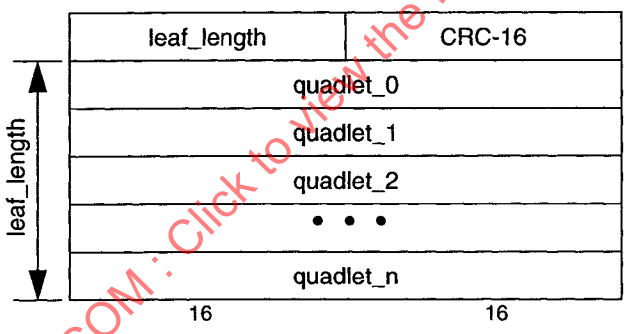



**Figure 62—Leaf format**

| directory_length | CRC-16 |
|---|---|
| key | Module_Dependent_Info |
| key | Textual_Descriptor(1) |
| key | Module_Hardware_Version |
| key | Node_Software_Version |
| key | Textual_Descriptor(2) |

**Figure 63—Textual descriptor locations**

The Textual_Descriptor entry may either point to a leaf or a directory (which provides descriptions for data entries in one or more languages). A leaf provides a single human-readable character string. A directory can have multiple leaves, each of which is a human-readable character string. When the Textual_Descriptor points to a directory, the directory contains entries that point to leaves; each leaf is expected to provide the same information, but would support a different language (and possibly different character sets).

The format of a generic Textual_Descriptor Leaf is shown in figure 64. The *specifier_id* field identifies the entity that defines the language defined by the *language_id*. The *string_info* quadlets contain the text description of the data entry. The format of the *string_info* information is *specifier_id/language_id* dependent.

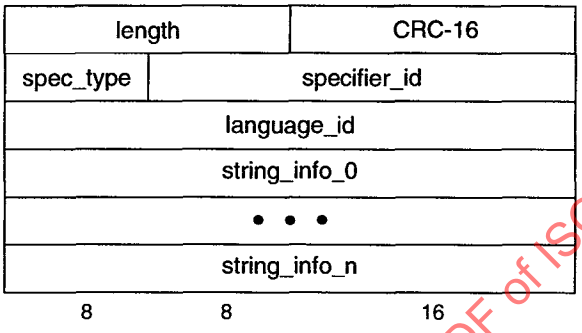



**Figure 64—Textual descriptor leaf**

Vendors who wish to supply languages to be used in the CSR Architecture's standard directories, other than the minimal ASCII, shall provide the *spec_type* and *specifier_id* values and shall usefully specify the format and meaning of the *string_info_0* through *string_info_n* quadlet values. For vendor-dependent entries, the value of *spec_type* shall be 1 and the *specifier_id* field shall be the vendor's company_id value.

The *spec_type* value of zero is used when the language is specified by a standards body, rather than by a vendor with a company_id value. For this *spec_type* value, the *specifier_id* value of 0 is reserved for use by the CSR Architecture. If *spec_type*, *specifier_id*, and *language_id* are all zero, this shall indicate that the minimal ASCII set specified in this document is being used and the language of the textual descriptor is English. The textual descriptor leaf format for the minimal ASCII set is as shown in figure 65.

The above format shows a string n characters long, char [0] to char [n-1]. If the length of the string is not quadlet aligned, the last quadlet shall be null extended, NUL = $00_{16}$.

A Textual_Descriptor directory contains one or more Textual_Descriptor leaves and no other entries.

## 8.3 bus_info_block

The format of the bus_info_block is specified in figure 66.

The bus_info_block shall contain the 4-character **bus_name** value and may contain additional bus-dependent information. For example, the bus-dependent information may be used to specify which profile or profiles are implemented. The "ABCD" characters correspond to the IEEE PAR number assigned to the bus

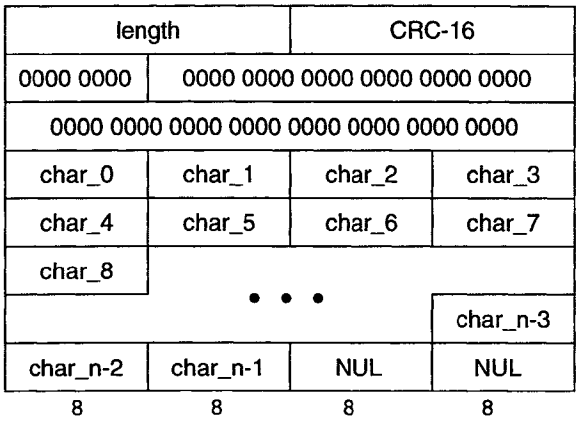



**Figure 65—Minimal ASCII textual descriptor leaf format**

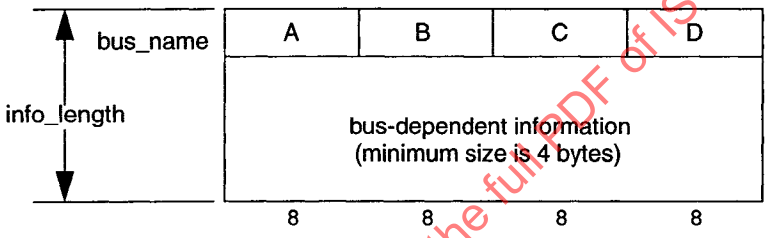



**Figure 66—bus_info_block format**

standard and are stored in the first four characters. For example, an "0896" number would correspond to the IEEE Futurebus+ specifications ([B3] and [B4]), as illustrated in table 27.

**Table 27—Bus_name example values**

| Position | A | B | C | D |
|---|---|---|---|---|
| ASCII representation | '0' | '8' | '9' | '6' |
| Hexadecimal value | $30_{16}$ | $38_{16}$ | $39_{16}$ | $36_{16}$ |

The optionality, format, and meaning of the bus-dependent information shall be defined by the bus standard corresponding to the **bus_name** value. The ASCII string containing four zero characters, "0000," should be used on buses that are compliant with the CSR Architecture but do not have an IEEE PAR number (because the bus is not yet or may never be an industry/IEEE standard). Vendor-dependent processor/memory buses are a good example of this.

## 8.4 Root directory entries

The root_directory contains entries that describe the module and node and provides pointers to other information provided by a general ROM format. The defined root_directory entries are illustrated in table 28.

**Table 28—Root directory entries**

| Entry name | Entry description |
|---|---|
| Bus_Dependent_Info | bus-dependent info |
| Module_Vendor_Id | module's vendor provides company_id |
| Module_Hw_Version | module's diagnostic software identifier |
| Module_Spec_Id | module's architect provides company_id |
| Module_Sw_Version | module's software interface identifier |
| Module_Dependent_Info | module's additional information |
| Node_Vendor_Id | node's vendor provides company_id |
| Node_Hw_Version | node's diagnostic software identifier |
| Node_Spec_Id | node's architect provides company_id |
| Node_Sw_Version | node's software interface identifier |
| Node_Capabilities | node's option-bits specification |
| Node_Unique_Id | node's unique 64-bit identifier |
| Node_Units_Extent | node's extended units space size |
| Node_Memory_Extent | node's extended memory space size |
| Node_Dependent_Info | node's additional information |
| Unit_Directory | unit-specific information (one per unit) |



### 8.4.1 Bus_Dependent_Info

(Optional.)
Used to provide additional bus-dependent information.

The Bus_Dependent_Info entry specifies a directory or a leaf that contains information not contained in the bus information block. The meaning of the directory's *key_value* is bus-dependent and the meaning of quadlets within the leaf is bus-dependent; the bus_name value specifies the bus standard that defines them.

### 8.4.2 Module_Vendor_Id

(Required.)
Provides the company_id of the vendor manufacturing the module.

The 24-bit immediate Module_Vendor_Id value provides the company_id of the vendor that manufactured the module. The company_id value is uniquely assigned to each module vendor by the IEEE. See 8.7 for details.

### 8.4.3 Module_Hw_Version

(Optional.)
Used to identify the appropriate diagnostic software for the module.

The 24-bit immediate Module_Hw_Version value provides a module-hardware identifier. The Module_Hw_Version number, when prepended with the Module_Vendor_Id value, is expected to uniquely identify the appropriate diagnostic software for extensive module-level tests.

### 8.4.4 Module_Spec_Id

(Optional.)
If not implemented, its assumed value shall be equal to Module_Vendor_Id.

The 24-bit immediate Module_Spec_Id value provides the company_id of the vendor that defined the architectural interface for the module.

### 8.4.5 Module_Sw_Version

(Optional.)
Identifies the I/O driver software interface architecture for the module. Shall not be present if either Node_Sw_Version or Unit_Sw_Version is present.

The 24-bit immediate Module_Sw_Version value provides a module-software identifier. This Module_Sw_Version number, when prepended with the Module_Spec_Id value, is expected to uniquely identify the appropriate I/O driver software for the module.

### 8.4.6 Module_Dependent_Info

(Optional.)
Used to provide additional information about the module.

The Module_Dependent_Info entry specifies a directory or leaf containing module-dependent information. The meaning of the directory's *key_value* is vendor-dependent and the meaning of quadlets within the leaf is vendor-dependent; the module's Module_Vendor_Id value specifies the vendor that defines them.

### 8.4.7 Node_Vendor_Id

(Optional.)
If not implemented, its assumed value shall be equal to Module_Vendor_Id.

The 24-bit immediate Node_Vendor_Id value provides the 24-bit company_id of the vendor that manufactured the node.

### 8.4.8 Node_Hw_Version

(Optional.)
Used to identify the appropriate diagnostic software for the node.

The 24-bit immediate Node_Hw_Version value provides a node-hardware identifier. The Node_Hw_Version number, when prepended with the Node_Vendor_Id value, is expected to uniquely identify the appropriate diagnostic software for extensive node-level tests.

### 8.4.9 Node_Spec_Id

(Optional.)
If not implemented, its assumed value shall be equal to Node_Vendor_Id.

The 24-bit immediate Node_Spec_Id value provides the 24-bit company_id of the vendor defining the architectural interface for the node.

### 8.4.10 Node_Sw_Version

(Optional.)
Identifies the I/O driver software interface architecture for the node. Shall not be present if Module_Sw_Version or Unit_Sw_Version is present.

The 24-bit immediate Node_Sw_Version value provides a node-software identifier. This Node_Sw_Version number, when prepended with the Node_Spec_Id value, is expected to uniquely identify the appropriate I/O driver software for the node.

### 8.4.11 Node_Capabilities

(Required.)
Shall identify which options are implemented.

The 24-bit immediate Node_Capabilities value indicates which options in the CSR Architecture are implemented in this node. The Node_Capabilities entry's format is shown in figure 67.

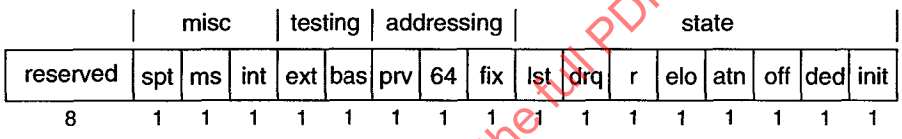



**Figure 67—Node capabilities entry format**

The nonzero bits within the Node_Capabilities entry specify which options are implemented, as described in table 29.

### 8.4.12 Node_Unique_Id

(Optional.)
Used to uniquely identify the node.

The leaf Node_Unique_Id entry provides a 64-bit binary number, which, if present, shall be uniquely assigned by the vendor to each node. For example, this entry could provide a serial number. The format of the Node_Unique_Id value leaf is illustrated in figure 68.

The technique used to assign the *unique_id* values is vendor-dependent and beyond the scope of the ROM format. If the Node_Unique_Id entry is present, the 88-bit number formed by concatenating the 24-bit Node_Vendor_Id value with the 64-bit *unique_id* value shall be unique for any node.

### 8.4.13 Node_Units_Extent

(Optional)
Required if one or more units are located within the extended units space.

Specifies the size and alignment constraints for the node's extended units space. The Node_Units_Extent entry may have an immediate or offset value format.

**Table 29—Node capabilities bit field definition**

| Bit field | Description |
|---|---|
| *spt* | The SPLIT_TIMEOUT register is implemented. |
| *ms* | The messages-passing registers are implemented. |
| *int* | The INTERRUPT_TARGET and INTERRUPT_MASK registers are implemented. |
| *ext* | The ARGUMENT registers are implemented. |
| *bas* | Node implements TEST_START&TEST_STATUS registers and testing state. |
| *prv* | The node implements the private space. |
| *64* | The node uses 64-bit addressing (otherwise 32-bit addressing). |
| *fix* | The node uses the fixed addressing scheme (otherwise extended addressing). |
| *lst* | The STATE_BITS.*lost* bit is implemented. |
| *drq* | The STATE_BITS.*dreq* bit is implemented. |
| *r* | This bit shall be zero, but shall be ignored when read. |
| *elo* | The STATE_BITS.*elog* bit and the ERROR_LOG registers are implemented. |
| *atn* | The STATE_BITS.*atn* bit is implemented. |
| *off* | The STATE_BITS.*off* bit is implemented. |
| *ded* | The node supports the *dead* state. |
| *init* | The node supports the *initializing* state. |

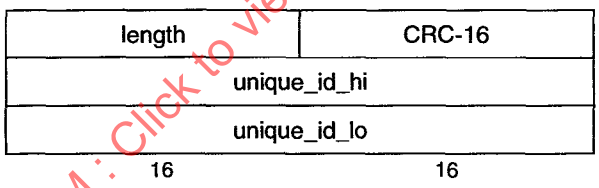



**Figure 68—Unique_Id leaf format**

### 8.4.13.1 Node_Units_Extent immediate format

The immediate Node_Units_Extent value format (*key_type* is 0) provides *align* and *number* parameters within a 24-bit value. The immediate value format is illustrated in figure 69.

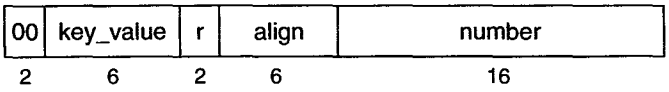



**Figure 69—Extent entry format, immediate form**

The fields in the immediate Node_Units_Extent entry are defined in table 30.

**Table 30—Extent entry fields, immediate form**

| Bit fields | Description |
|---|---|
| *r* | reserved 2-bit field, shall equal $00_2$ |
| *align* | the align parameter for the extended space calculations |
| *number* | the number parameter for the extended space calculations |

The *align* parameter specifies the address-alignment restrictions for the base address of the node's extended units space. The *number* parameter, when multiplied by the alignment size, specifies the size of the extended units space.

The location and size of the node's extended units space shall be defined when the baseAddress and boundAddress values are in the UNITS_BASE and UNITS_BOUND registers, respectively. The values of baseAddress and boundAddress are based on the *align* and *number* parameters provided by this entry, and the count value provided by system software, as specified by the C code of table 31.

**Table 31—Extended space alignment and size**

```
/* ROM specifies align, software specifies count.
 * The count argument selects the base addresses location.
 * This code assumes that the default integer size is 32 or 64 bits, for
 * nodes supporting 32-bit and 64-bit extended addressing, respectively.
 */
#define ENB 1
int
CalculateBaseAddress(int align, int count)
{
  return ((1 << align) * count);                /* Return baseAddress */
}

/* The baseAddress, align and number arguments are provided by ROM */
int
CalculateBoundAddress(int baseAddress, int align, int number)
{
  return (baseAddress + (1 << align) * number); /* Return boundAddress */
}
```

### 8.4.13.2 Node_Units_Extent offset format

The offset Node_Units_Extent value format (*key_type* is 1) provides a 24-bit *offset* parameter. The offset value format is illustrated in figure 70.


**Figure 70—Node_Units_Extent format, offset form**

The *csr_offset* value specifies a CSR address, as a quadlet offset from the base address of the initial register space. An EXTENT register is located at the *csr_offset*-specified address and shall have the format specified in figure 71.

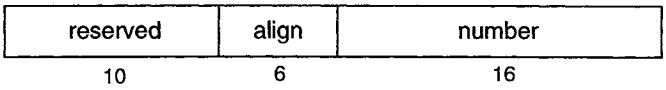



**Figure 71—EXTENT register format**

The fields within the EXTENT register are defined in table 32.

**Table 32—EXTENT register fields, offset form**

| Bit field | Description |
|---|---|
| *r* | reserved 10-bit field, shall be zero. |
| *align* | the align parameter for the extended units space calculation. |
| *number* | the number parameter for the extended units space calculation. |

The location and size of the node's extended units space shall be defined when the baseAddress and boundAddress values are in the UNITS_BASE and UNITS_BOUND registers, respectively. The values of baseAddress and boundAddress are based on the *align* and *number* parameters provided by the EXTENT register, and the count value provided by system software, as specified by the C code of table 31.

### 8.4.14 Node_Memory_Extent

(Optional.)
Required if a node contains a memory controller that is expected to be mapped to the available address space.

Specifies the size and alignment constraints for the node's extended memory space. The Node_Memory_Extent entry may have an immediate or offset value format.

#### 8.4.14.1 Node_Memory_Extent immediate format

The immediate Node_Memory_Extent value format (*key_type* is 0) provides *align* and *number* parameters within a 24-bit value. The immediate value format is illustrated in figure 69. The fields in the immediate Node_Memory_Extent entry are defined in table 30.

The *align* parameter specifies the address-alignment restrictions for the base address of the node's extended memory space. The *number* parameter, when multiplied by the alignment size, specifies the size of the extended memory space.

The location and size of the node's extended memory space shall be defined when the baseAddress and boundAddress values are in the MEMORY_BASE and MEMORY_BOUND registers, respectively. The values of baseAddress and boundAddress are based on the *align* and *number* parameters provided by this entry, and the count value provided by system software, as specified by the C code of table 31.

#### 8.4.14.2 Node_Memory_Extent offset format

The offset Node_Memory_Extent value format (*key_type* is 1) provides a 24-bit *offset* parameter. The offset value format is illustrated in figure 70.

The *offset* value specifies a CSR address, as a quadlet offset from the base address of the initial register space. An EXTENT register is located at this address and has the format specified in figure 71. The fields within the EXTENT register are defined in table 32.

The location and size of the node's extended memory space shall be defined when the baseAddress and boundAddress values are in the MEMORY_BASE and MEMORY_BOUND registers, respectively. The values of baseAddress and boundAddress are based on the *align* and *number* parameters provided by the EXTENT register, and the count value provided by system software, as specified by the C code of table 31.

### 8.4.15 Node_Dependent_Info

(Optional.)
Used to provide additional information about the node.

The leaf or directory Node_Vendor_Dependent_Info provides vendor-dependent information. The format and meaning of this information is dependent on the 48-bit value produced by prepending the 24-bit Node_Vendor_Id value to the 24-bit Node_Hw_Version number.

### 8.4.16 Unit_Directory

(Optional.)
Used to provide additional information about a unit. If either the Module_Sw_Version or the Node_Sw_Version are provided, this entry shall not be present.

The Unit_Directory entry points to a lower-level directory that contains unit-specific information. There may be several Unit_Directory entries in each root_directory, one for each unit on the node.

## 8.5 Unit directories

If any unit subdirectories are present in the ROM, then one unit directory is required for each unit on the node. All of the defined unit directory entries are shown in table 33.

**Table 33—Unit directory entries**

| Entry name | Entry description |
|---|---|
| Unit_Spec_Id | unit's vendor provides company_id |
| Unit_Sw_Version | unit's I/O driver software identifier |
| Unit_Dependent_Info | unit's additional information |
| Unit_Location | unit's address-space location |
| Unit_Poll_Mask | unit's bits within STATE_CLEAR |

A unit directory shall not be provided if either Module_Sw_Version or Node_Sw_Version is present.

### 8.5.1 Unit_Spec_Id

(Optional.)
If not implemented, its assumed value shall be equal to Node_Vendor_Id.

The 24-bit immediate Node_Spec_Id value provides the 24-bit company_id of the vendor defining the architectural interface for the unit.

### 8.5.2 Unit_Sw_Version

(Required if the unit directory is present.)
Identifies the I/O driver software interface architecture for the unit.

The 24-bit immediate Unit_Sw_Version value provides a unit-software identifier. This Unit_Sw_Version number, when prepended with the Unit_Spec_Id value, is expected to identify uniquely the unit architecture and therefore the appropriate I/O driver software for the unit.

### 8.5.3 Unit_Dependent_Info

(Optional.)
Used to provide additional information about the unit.

The leaf or directory Unit_Dependent_Info provides additional unit-dependent information. The format and meaning of this information is dependent on the 48-bit value produced by prepending the 24-bit Unit_Spec_Id value to the 24-bit Unit_Sw_Version number.

### 8.5.4 Unit_Location

(Optional.)
Required if two or more unit subdirectories are present.

The Unit_Location entry provides a *tag*, *base64*, and *bound64* values. A unit is mapped to a contiguous range of addresses within one of the node's address spaces (initial node, indirect, initial memory, or extended memory spaces). The ***tag*** value selects one of 4 address spaces; the *base64* and *bound64* values specify the range of byte addresses that is mapped.

The *base64* and *bound64* pairs specify the quadlet-aligned base and bound addresses of the unit, measured as byte offsets from the base address of the specified space. The address specified by *bound64* is one greater than the last unit address.

The format of these two values is illustrated in figure 72.

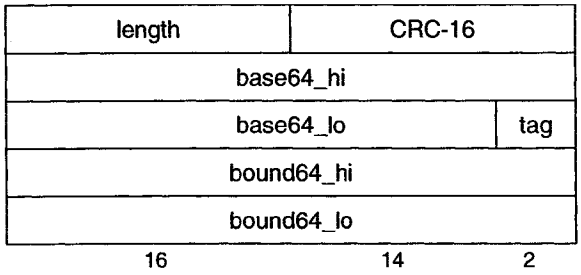



**Figure 72—Unit_Locate leaf format**

The value of ***tag*** specifies where the address space is located, as defined in table 34.

**Table 34—Unit_Locate.*tag* bit field definition**

| *tag* | Unit register location |
|---|---|
| $00_2$ | initial register space |
| $01_2$ | initial memory or extended units space |
| $10_2$ | indirect space |
| $11_2$ | reserved |

Note that the initial memory space and the extended units space are mutually exclusive; the initial memory space is only defined for the 64-bit fixed address model and the extended units space is only defined for the 32- and 64-bit extended address models. The node's address space model is specified by its node's **Node_Capabilities** entry.

### 8.5.5 Unit_Poll_Mask

(Optional.)
Required if the unit affects one or more bits in the STATE_CLEAR.*unit_depend* field.

The immediate Unit_Poll_Mask entry provides a 16-bit *poll_mask* value that is shifted left by 16 bits to define the bits within the STATE_CLEAR register that are affected by this unit. The format of the entry is shown in figure 73.

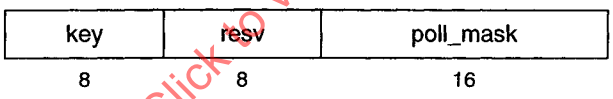



**Figure 73—Unit_Poll entry format**

## 8.6 Key definitions

The *key_type* and *key_value* values for the entries in the general ROM format are specified in table 8-17. The *key_type* is a 2-bit field and the *key_value* is a 6-bit field; all values are hexadecimal numbers.

With the exception of the Textual_Descriptor, there are no constraints on the order in which these entries appear within a directory, and (depending on its definition) some of the entries may appear multiple times within a directory.

The remaining *key_value* values are reserved as follows:

- $17_{16}$ to $2F_{16}$ are reserved for future definition by the CSR Architecture.
- $30_{16}$ to $37_{16}$ are reserved for definition by the bus standard identified in the Bus_Info_Block.
- $38_{16}$ to $3F_{16}$ are allocated for definition by vendors. Vendor-dependent *key_value* values may be position- and context-dependent. Within a vendor-dependent directory, the meaning of all *key_value* parameters is also vendor-dependent.

**Table 35—*key_type* and *key_value* (hexadecimal values)**

| Entry name | key_type(s) | key_value |
|---|---|---|
| Textual_Descriptor | leaf or directory | 01 |
| Bus_Dependent_Info | leaf or directory | 02 |
| Module_Vendor_Id | immediate | 03 |
| Module_Hw_Version | immediate | 04 |
| Module_Spec_Id | immediate | 05 |
| Module_Sw_Version | immediate | 06 |
| Module_Dependent_Info | leaf or directory | 07 |
| Node_Vendor_Id | immediate | 08 |
| Node_Hw_Version | immediate | 09 |
| Node_Spec_Id | immediate | 0A |
| Node_Sw_Version | immediate | 0B |
| Node_Capabilities | immediate | 0C |
| Node_Unique_Id | leaf | 0D |
| Node_Units_Extent | immediate or offset | 0E |
| Node_Memory_Extent | immediate or offset | 0F |
| Node_Dependent_Info | leaf or directory | 10 |
| Unit_Directory | directory | 11 |
| Unit_Spec_Id | immediate | 12 |
| Unit_Sw_Version | immediate | 13 |
| Unit_Dependent_Info | leaf or directory | 14 |
| Unit_Location | leaf | 15 |
| Unit_Poll_Mask | immediate | 16 |



## 8.7 company_ids

### 8.7.1 company_id assignments

The term "company_id" is used throughout this clause to identify uniquely vendors that manufacture or specify components. To obtain an Organizationally Unique Identifier (OUI), contact:

Registration Authority Committee
The Institute of Electrical and Electronic Engineers, Inc.
445 Hoes Lane
Piscataway, NJ 08855-1331
USA
(908) 562-3812

Ask for an Organizationally Unique Identifier (OUI) for your organization. See 8.7.2 for a description of how the IEEE/RAC-administered company_id value is used in the ROM entries defined by this standard.

### 8.7.2 company_id mappings

A direct or indirect read of the ROM quadlet containing the Module_Vendor_Id returns the company_id. For example, a company_id value of $ACDE48_{16}$ (which has a binary representation of $101011001101111001001000_2$) is returned as illustrated in figure 74.

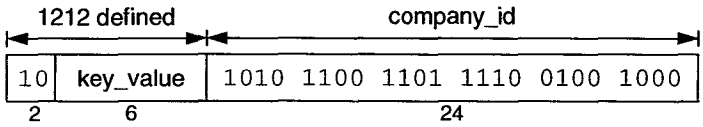



**Figure 74—Mapping of company_id bits**

Other ROM entries, such as Module_Spec_Id, Node_Vendor_Id, Node_Spec_Id and Unit_Spec_Id may also contain the company_id value, but have a different *key_value*.

Your company need not purchase a company_id if it has been previously assigned an IEEE *48-bit Globally Assigned Address Block* or an IEEE-assigned *Organizationally Unique Identifier (OUI)* for use in network applications. However, beware that the (left through right) order of the bits within the company_id value is not the same as the (first through last) network-transmission order of the bits within these other identifiers. Consult the IEEE Registration Authority for clarifying documentation.

# 9. Bus standard requirements

The CSR Architecture requires the bus standard to define the following:

a) **Addressing**. The bus standard shall specify which of the following address models are supported:
   1) 32-bit extended
   2) 64-bit extended
   3) 64-bit fixed

b) **Transaction definitions.** The bus standard shall define the following:
   1) Encoding. The bus standard shall specify the form of the required transactions (read1-read64, write1-write64, lock4, and lock8).
   2) Clocks. If synchronized clocks are supported, the bus standard shall specify the form of the clock_strobe signal.

c) **Error status codes**. The bus standard shall define signaling protocols for returning the following error-status values:
   1) Type_error
   2) Address_error
   3) Conflict_error

d) **Resets**. This specification defines two types of resets: power_reset (primary power is lost) and command_reset (write to RESET_START). The bus standards shall define how the power_reset event is signaled. A bus standard that defines other forms of reset shall define their side effects on the registers defined by the CSR Architecture.

e) **STATE_CLEAR** (*bus_dependent* field). Eight bits of the STATE_CLEAR register are bus dependent. These are expected to be used for specialized node status and control purposes.

f) **NODE_IDS** (bus-dependent fields). A bus standard shall define:
   1) *offset_id*. What is the initial *offset_id* value; is the *offset_id* a read/write or read-only value?
   2) *bus_dependent*. The format and meaning of this field shall be defined by the bus standard.

g) **ARGUMENT_LO** (extended tests). Eleven of the bits in the ARGUMENT_LO register are bus-dependent. These are expected to specify the access modes (32-bit, 64-bit, compelled, split/unified, etc.) or sizes (can 256-byte transactions be used?) for transactions that can be used by the node's extended test routines.

h) **SPLIT_TIMEOUT** (defined ranges). The bus standard should define the range of bits that shall be implemented in the SPLIT_TIMEOUT register.

i) **Bus-dependent registers**. The bus standard shall specify the content and size of the bus-dependent portion of the initial node space.

j) **Bus-dependent ROM entries**. The bus standard shall specify the following parts of ROM:
   1) Bus Name. The bus standard shall specify the last four characters of the *bus_name* component within the Bus_Information_Block.
   2) Bus_Information_Block. The bus standard shall specify the size of the Bus_Information_Block, plus any contents after the *bus_name* component (if any).
   3) Bus_Dependent entry point. If one is provided, the bus standard shall specify the type of the Bus_Dependent entry point (leaf or directory). The bus standard shall also specify the format and meaning of the contents within the Bus_Dependent entry point.

This page intentionally left blank

# Annexes

## Annex A
## Bibliography

(informative)

The CSR Architecture has involved the extensive cooperation of several bus standards. The standards listed below are cited as illustrations:

[B1] CCITT Recommendation V.41, Code-Independent Error-Control System. Geneva: International Telecommunications Union, 1988.

[B2] ISO/IEC 10857:1994 [ANSI/IEEE Std 896.1, 1994 Edition], Information technology—Futurebus+ — Logical protocol specification.

[B3] IEEE Std 896.2-1991, IEEE Standard for Futurebus+ — Physical Layer and Profile Specifications (ANSI).

[B4] IEEE P1014.1, Standard for a Futurebus+/VME64 Bridge (Draft No. 2, March 1993).

[B5] IEEE Std 1196-1987, IEEE Standard for a Simple 32-Bit Backplane Bus: NuBus (ANSI).

[B6] ISO/IEC 10861:1994 [ANSI/IEEE Std 1296-1987], Information technology—High-Performance Synchronous 32-Bit Bus: MULTIBUS II.

[B7] IEEE P1394, High Performance Serial Bus (Draft No. 6.6, March 1992).

[B8] IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface (SCI) (ANSI).