INTERNATIONAL STANDARD

ISO/IEC 13961 IEEE Std 1596

First edition
2000-07

Information technology –

Scalable Coherent Interface (SCI)

Reference number
ISO/IEC 13961:2000(E)
IEEE Std 1596, 1998 Edition

Abstract: The scalable coherent interface (SCI) provides computer-bus-like services but, instead of a bus, uses a collection of fast point-to-point unidirectional links to provide the far higher throughput needed for high-performance multiprocessor systems. SCI supports distributed, shared memory with optional cache coherence for tightly coupled systems, and message-passing for loosely coupled systems. Initial SCI links are defined at 1 Gbyte/s (16-bit parallel) and 1 Gb/s (serial). For applications requiring modular packaging, an interchangeable module is specified along with connector and power. The packets and protocols that implement transactions are defined and their formal specification is provided in the form of computer programs. In addition to the usual read-and-write transactions, SCI supports efficient multiprocessor lock transactions. The distributed cache-coherence protocols are efficient and can recover from an arbitrary number of transmission failures. SCI protocols ensure forward progress despite multiprocessor conflicts (no deadlocks or starvation).

Keywords: bus architecture, bus standard, cache coherence, distributed memory, fiber optic, interconnect, I/O system, link, mesh, multiprocessor, network, packet protocol, ring, seamless distributed computer, shared memory, switch, transaction set.

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

ISBN 2-8318-5375-3

INTERNATIONAL STANDARD

ISO/IEC 13961 IEEE Std 1596

First edition
2000-07

Information technology –

Scalable Coherent Interface (SCI)

Sponsor

Microprocessor and Microcomputer Standards Subcommittee of the IEEE Computer Society

PRICE CODE XF

For price, see current catalogue

CONTENTS

Clause Page

Clause Page

Clause Page

Page

Page

Page

Page

Page

INFORMATION TECHNOLOGY –

SCALABLE COHERENT INTERFACE (SCI)

FOREWORD

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

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

3) Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 13961 was prepared by subcommittee 26: Microprocessor systems, of ISO/IEC joint technical committee 1: Information technology.

Annexes A and B are for information only.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute.

The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

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

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

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

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

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

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Foreword to IEEE Std 1596, 1998 Edition

[This foreword is not a part of ISO/IEC 13961:2000, Information technology – Scalable Coherent Interface (SCI).]

The demand for more processing power continues to increase, and apparently has no limit. One can usefully saturate the resources of any computer so easily by merely specifying a finer mesh or higher resolution for the solution of some physical problem (hydrodynamics, for example), that engineers and scientists are desperate for enormously larger computers.

To get this kind of computing power, it seems necessary to use a large number of processors cooperatively. Because of the propagation delays introduced when signals cross chip boundaries, the fastest uniprocessor may be on one chip before long. Pipelining and similar large-mainframe tricks are already used extensively on single-chip processors. Vector processors help, but are hard to use efficiently in many applications. Multiprocessors communicating by message passing work well for some applications, but not for all. The shared-memory multiprocessor looks like the best strategy for the future, but a great deal of work will be needed to develop software to use it efficiently.

It is important to support both the shared-memory and the message-passing models efficiently (and at the same time) in order to support optimal software for a wide range of problems, especially for a system that dynamically allocates processors and perhaps changes its configuration depending on the nature of its load.

SCI started from an attempt to increase the bandwidth of a backplane bus past the limits set by backplane physics in order to meet the needs of new generations of processor chips, some of which can single-handedly saturate the fastest buses. We soon learned that we had to abandon the bus structure to achieve our goals.

Backplane performance is limited by physics (distributed capacitances and the speed of light) and by a bus's one-at-a-time nature, an inherent bottleneck. To gain performance far beyond what buses and backplanes can do, one needs better signaling techniques and the concurrent use of many signaling paths.

Rather than using bused backplane wires, SCI is based on point-to-point interconnect technology. This design approach eliminates many of the physics problems and results in much higher speeds. SCI in effect simulates a bus, providing the bus services one expects (and more) without using buses.

Committee Membership

The specification has been developed with the combined efforts of many volunteers. The following is a list of those who were members of the Working Group while the draft and final specification were compiled:

David B. Gustavson, Chair

David V. James, Vice Chair

Nagi Aboulenein
Knut Alnes
Robert H. Appleby
Kurt Baty
Amir Behroozi
David L. Black
Andre Bogaerts
Paul Borrill
Patrick Boyle
David Brearley, Jr.
Charles Brill
Haakon Bugge
Jan Buytaert
Jay Cantrell
Mike Carlton
Fred L. Chong
Graham Connolly
James R.(Bob) Davis
W. Kenneth Dawson
Stephen R. Deiss
Gary Demos
Roberto Divia
Gregg Donley
Wayne Downer
Guy Fedorkow
Peter Fenner
David Ford
Stein Gjessing
Torstein Gleditsch
James Goodman
Robert J. Greiner
Charles Grimsdale
Emil N. Hahn
Horst Halling
Craig Hansen
Marit Jenssen
Rajeev Jog
Svein Erik Johansen
Sverre Johansen
Ross Johnson
Anatol Kaganovich
Alain Kagi
Hans Karlsson*
Tom Knight
Michael J. Koster
Ernst Kristiansen
Stein Krogdahl
Ralph Lachenmaier
Branko Leskovar
Dieter Linnhofer
Robert McLaren
Mark Mellinger
Svein Moholt
Viggy Mokkarala
John Moussouris
Hans Muller
Klaus D. Muller
Ellen Munthe-Kaas
Russell Nakano
Tom Nash
Steve Nelson
Julian Olyansky
Chris Parkman
Dan Picker
Phil Ponting
Steve Quinton
Jean F. Renardy
Randy Rettberg
Morten Schanke
Gene Schramm
James L. Schroeder
Tim Scott
Donald Senzig
Gurindar Sohi
Robert K. Southard
Joanne Spiller
Paul Sweazey
Lorne Temes
Manu Thapar
John Theus
Mike van Brunt
Phil Vukovic
Anthony Waitz
Richard Walker
Steve Ward
Carl Warren*
Steinar Wenaas
Mike Wenzel
Richard J. Westmore
Wilson Whitehead
Hans Wiggers
Mark Williams
Philip Woest
S. Y. Wong
Ken Wratten
Chu-Sun Yen

*deceased

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

M. R. Aaron
Scott Akers
Ray S. Alderman
John Allen
Knut Alnes
Richard P. Ames
Bjorn Bakka
David M. Barnum
Kurt Baty
Harrison A. Beasley
Amir Behroozi
Janos Biri
David Black
William P. Blase
Andre Bogaerts
W. C. Brantley
David Brearley, Jr.
Haakon Bugge
Kim Clohessy
Graham Connolly
Jonathan C. Crowell
W. Kenneth Dawson
Stephen Deiss
Dante Del Corso
Stephen L. Diamond
Jean-Jacques Dumont
William P. Evertz
Guy Federkow
Timothy R. Feldman
Peter Fenner
Gordon Force
Stein Gjessing
Andy Glew
Patrick Gonia
James Goodman
Charles Grimsdale
David Hawley
Phil Huelson
Zoltan R. Hunor
Edgar Jacques
David V. James
Kenneth Jansen
Rajeev Jog
Sverre Johansen
Ross Johnson
Jack R. Johnson
Anatol Kaganovich
Christopher Koehle
Michael J. Koster
Ernst H. Kristiansen
Ralph Lachenmaier
Glen Langdon, Jr.
Gerry Laws
Minsuk Lee
Branko Leskovar
Anthony G. Lubowe
Svein Moholt
James M. Moidel
James D. Mooney
Klaus-Dieter Mueller
Ellen Munthe-Kaas
Cuong Nguyen
J. D. Nicoud
Dan O Connor
Mira Pauker
Donald Pavlovich
Thomas Pittman
Steve Quinton
Richard Rawson
Steven Ray
Randy Rettburg
Hans Roosli
Paul Rosenberg
Carl Schmiedekamp
James L. Schroeder
Don Senzig
Philip Shutt
Michael R. Sitzer
Gurindar Sohi
Robert K. Southard
Joanne Spiller
David Stevenson
Robert Stewart
Paul Sweazey
Daniel Tabak
Daniel Tarrant
Lorne Temes
Manu Thapar
Michael G. Thompson
Chris Thomson
Joseph P. Trainor
Robert Tripi
Robert J. Voigt
Phil Vukovic
Yoshiaki Wakimura
Richard Walker
Eike Waltz
Carl Warren*
Richard J. Westmore
Hans A. Wiggers
Mark Williams
Andrew Wilson
Joel Witt
Ken Wratten
David L. Wright
Chu Yen
Oren Yuen
Janusz Zalewski

*deceased

When the IEEE Standards Board approved this standard on 19 March 1992, it had the following membership:

Marco W. Migliaro, Chair **Donald C. Loughry**, Vice Chair

Andrew G. Salem, Secretary

Dennis Bodson	Donald N. Heirman	T. Don Michael*
Paul L. Borrill	Ben C. Johnson	John L. Rankine
Clyde Camp	Walter J. Karplus	Wallace S. Read
Donald C. Fleckenstein	Ivor N. Knight	Ronald H. Reimer
Jay Forster*	Joseph Koepfinger*	Gary S. Robinson
David F. Franklin	Irving Kolodny	Martin V. Schneider
Ramiro Garcia	D. N. Jim Logothetis	Terrance R. Whittemore
Thomas L. Hannan	Lawrence V. McCall	Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

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

This standard was was approved by the American National Standards Institute on 23 October 1992. It was reaffirmed by IEEE in 1998.

– Blank page –

INFORMATION TECHNOLOGY –

SCALABLE COHERENT INTERFACE (SCI)

1 Introduction

1.1 Document structure

This International Standard describes a communication protocol that provides the services required of a modern computer bus, but at far higher performance levels than any bus could attain. Packet protocols on unidirectional point-to-point transmission links emulate a sophisticated bus without incurring the inherent bus physics or bus contention problems.

This International Standard is partitioned into clauses that serve several distinct purposes:

Clause 1: Introduction provides background for understanding the Scalable Coherent Interface (SCI) protocols, and may be skipped by those already familiar with these concepts. The descriptions in this clause are somewhat simplified, and should not be considered part of the SCI specification.

Clause 2: References, glossary and notation defines the terminology used within this standard and lists references that are required for implementing the standard.

Clause 3: Logical protocols and formats defines the packets and protocols that implement transactions (like reads and writes) between SCI nodes. This clause uses text and figures as introductory material, to establish a frame of reference for the formal specification.

Clause 4: Cache-coherence protocols provides background information for understanding the protocols used by two or more SCI nodes to maintain coherence between cached copies of shared data. The coherence protocols contain many options. This clause describes the minimal subset of these protocols, a typical set of options that are likely to be implemented, and also the full set of protocols.

Clause 5: C-code structure explains the structure of the C code that defines the logical (packet symbol processing) and cache-coherence protocols. The precise specifications of the logical-level packet protocols and the cache-coherence protocols, which involve a large number of state-transition details, are expressed in C code because it is difficult to state them unambiguously in English, and so that they can be tested thoroughly under simulation.

Clause 6: Physical layers defines a mechanical package and several physical links that may be used to implement the logical protocols. This clause uses text and figures to specify the mechanical and electrical characteristics of several physical links.

Annexes A and B: These annexes describe other system-related concepts that have influenced the design of this standard. These may be useful for understanding the rationale behind some of the SCI design decisions.

Bibliography provides a variety of references that may be useful for understanding the terminology, notation, or concepts discussed within this standard.

C code: The C code is published as a text file on an IBM-format diskette. This was done for the convenience both of the casual reader of this standard, who will not delve into the details of the C code, and also of the serious user, who will wish to understand the C code thoroughly, executing it on a computer. Though the C code takes precedence over this International Standard in case of inconsistency, this International Standard provides considerable explanation and illustration to help develop an intuitive understanding that will make the C code more comprehensible.

1.2 SCI overview

1.2.1 Scope and directions

Purpose: To define an interface standard for very high-performance multiprocessor systems that supports a coherent shared-memory model scalable to systems with up to 64 K nodes. This standard is to facilitate assembly of processor, memory, I/O, and bus adaptor cards from multiple vendors into massively parallel systems with throughputs ranging up to more than 10^{12} operations per second.

Scope: This standard will encompass two levels of interface, defining operation over distances less than 10 m. The *physical* layer will specify electrical, mechanical, and thermal characteristics of connectors and cards. The *logical* level will describe the address space, data transfer protocols, cache coherence mechanisms, synchronization primitives, *control* and status registers, and initialization and error recovery facilities.

The preceding statements were those submitted to and approved by the IEEE Standards Board as the definition of the SCI project. These goals have been met and exceeded: support for message-passing was added, and the operating distance is not limited to 10 m. (The intent of that limitation was to make clear that this is not yet-another Local Area Network.)

The real distinction between SCI and a network has more to do with the memory-access-based model SCI uses and the distributed cache-coherence model.

The practical operating distance depends more on the throughput and performance needed than on any absolute limit built into the specification. Very long links would yield unacceptable performance for many users (but perhaps not all).

In particular, the fibre-optic physical layer can extend the SCI paradigm over distances long enough to link a computer to its I/O devices, or to link several nearby processors. No arbitrary length limit would be appropriate, but practical considerations including the throughput requirements and the cost of transmitters and receivers will set the lengths that people consider useful.

A very-high-priority goal was that SCI be cost-effective for small systems as well as for the massively parallel ones mentioned in the purpose statement above. SCI's low pin count and simple ring implementation make medium-performance, few-processor systems easier to build with SCI than with bused backplane systems; a two-layer backplane should be sufficient, and three layers should be enough to support the optional geographical addressing mechanism. The SCI interface, complete with transceivers, fits into a single IC package that includes much of the logic needed to support the cache-coherence protocols. This economy for small systems leads to the expectation that SCI processor boards will be built in high volume, making them inexpensive enough to be assembled in large numbers for building supercomputers at low cost.

SCI also simplifies the construction of reliable systems. SCI Type 1 modules are well protected against electrostatic discharge and electromagnetic interference, and can be safely inserted while the remainder of the system remains powered. SCI supports live insertion and withdrawal by using a single supply voltage (with on-board conversion as needed) and staggered pin lengths in the connector to guarantee safe sequencing. Note, however, that system software plays an important role in live insertion or removal of a module because the resources provided by that module have to be allocated and deallocated appropriately.

In systems where several modules share a ringlet, the removal of one module interrupts all communication via that ringlet, so the resources on those modules also have to be deallocated. A similar situation arises in any system that may have multiple processors resident on one field-replaceable board: all have to be deallocated when any one is replaced. The system software for handling the deallocation and reallocation of these resources is outside SCI's scope.

Although SCI does not provide fault tolerance directly in its low-level protocols, it does provide the support needed for implementing fault-tolerant operation in software. With this recovery software, the SCI coherence protocols are robust and can recover from an arbitrary number of detected transmission failures (packets that are lost or corrupted).

The SCI paradigm removes the limits that bus structures place on throughput, but its latency is of course limited by the speed of signal propagation (less than the speed of light). Ever-increasing throughput can be expected as technology improves, but the organization of hardware and software will have to take into account the relatively constant latency (delay between request and response), which is proportional to the physical size of the system.

The last generation of buses approached the ultimate limits of performance, leading to the concept of an ultimate standard. However, the initially defined SCI physical layers are likely just the first of a series of implementations having higher or lower performance levels. The 1 Gbyte/s link speed specified for the initial ECL/copper-backplane implementation was chosen based on a combination of marketing and engineering considerations. From a marketing point of view, it was necessary to define a territory that did not disturb the markets for present 32-bit standards or present networks, and from an engineering point of view this link speed was near the edge of what available signalling technology and integrated circuit technology could support.

New technologies, such as better cables, connectors, transceivers; IC packages with more pins or higher power-dissipation capabilities; or faster ICs, could make it practical or desirable to implement SCI on new physical-layer standards. Such standards, with different link widths or bit rates, will be developed from time to time. However, packet formats and higher level coherence protocols will be the same across all these physical implementations. That should make the problem of interfacing one SCI system to another relatively simple – SCI already includes the necessary mechanisms to cope easily with speed differences.

1.2.2 The SCI approach

The objective of SCI was to define an interconnect system that *scales* well as the number of attached processors increases, that provides a *coherent* memory system, and that defines a simple *interface* between modules.

SCI developers initially hoped to make a better backplane bus to meet these goals, but soon realized no bus could do the job. Bus speeds are limited by the distance a signal must travel and the propagation delay across a backplane. In asynchronous buses, the limit is the time needed for a handshake signal to propagate from the source to the target and for a response to return to the source. In synchronous buses, it is the time difference between clock and data signals that originate in different places.

Transmission lines in a backplane bus are affected by reflections caused by multiple connectors, as well as by variations in loading as the number of inserted modules changes. This makes a backplane bus an imperfect transmission line at best.

Furthermore, a backplane bus can only handle one data transmission at a time and therefore becomes a bottleneck in multiprocessor systems. Although bridges can be used to extend the bus concept to a multiple-bus topology, these bridges are expected to be more costly and less efficient than SCI switches. Support for an efficient switch greatly influenced the design of the SCI protocols.

SCI solves these problems by defining a radically different interconnect system. SCI defines an interface standard that enables a system integrator to connect boards using many different interconnect configurations. These configurations may range from simple rings to complex multistage switching networks. SCI modules still may plug into a backplane – it holds the connectors in place; it is just not wired as a bus.

SCI uses point-to-point unidirectional communication between neighbouring nodes, greatly reducing the nonideal-transmission-line problems. The bandwidth of the point-to-point link depends on the transmission medium. A Type 18 DE 500 link is 2 bytes wide and data are transferred at 1 Gbyte/s, using differential ECL signalling and both edges of a 250 MHz clock.

The clock rate can be much higher for point-to-point links than for buses. For a given data rate this makes it possible to use faster clocking to reduce the link width. This reduces the pin count for bus interface logic, so that the entire bus interface can be integrated on a single chip. Thus, timing skews can be tightly specified, since components are inherently well matched in a single-chip design. A large number of requests can be outstanding at the same time, making SCI well suited for high-performance multiprocessor systems. SCI allows up to 64 K nodes to be connected in a single system. Since each node could itself be a multiprocessor, the SCI addressing mechanism should be sufficient to support the next generation of massively parallel computer systems.

Cache coherence is an important part of the proposed standard. Switching networks cannot easily provide reliable broadcast or eavesdrop capabilities. Hence the SCI coherence protocols are based on single-responder directed bus transactions and distributed directories, where processors sharing cache lines are linked together by pointers. Broadcasts are generally software, not hardware, operations, though the protocols do support some (noncoherent) broadcast transactions that may be useful in certain applications.

1.2.3 System configurations

An SCI node relies on feedback arriving on its input link to *control* its behaviour on its output link. Thus there must always be a ring-like connection, with the output of one node providing the input to another. Implementations of this structure range from a small ring connecting two nodes (one of which might be the port to a fast switch) to a large ring consisting of many nodes. The term ringlet is often used to imply a ring that has a relatively small number of nodes, up to perhaps half a dozen. Few applications will perform well with large rings because each node sees traffic generated by all the other nodes on the ring; for some I/O applications, however, large rings may be appropriate.

One node on each ring (called the scrubber) is assigned certain housekeeping tasks, such as initializing the ring to the point that each node is addressable, maintaining certain timers, and discarding damaged packets so they don't circulate indefinitely.

For performance, fault tolerance or other reasons many systems will require more than one ringlet. Agents, which consist of two or more SCI node interfaces to different ringlets, with appropriate routing mechanisms, are used to allow nodes on different ringlets to communicate with one another in a transparent way.

One can build useful switch fabrics consisting of many ringlets with a few processor nodes and agents on each. Or one can use more traditional switch mechanisms that have SCI interfaces at their extremities but transparently use whatever internal data transfer and switching techniques they prefer.

1.2.4 Initial physical models

The logical portion of the SCI specification defines the format and function of fields in packets that are sent from one SCI node to another over any one of several different physical link layers.

SCI links continually transmit symbols that contain 16 data bits plus packet-delimiter and clock information. The clock provides a precise timing reference that the receiver uses for extracting data from the incoming signals. A symbol is either part of a packet (a contiguous sequence of symbols marked by the packet delimiter) or an idle symbol (transmitted during the interval between packets to maintain synchronism between the link and the receiver). On a backplane, where signal wires are relatively inexpensive, an entire symbol may be sent each clock period. On longer-distance interconnects, where signal wires are relatively expensive, the symbols may be sent one bit at a time.

The notation used by SCI for names of link types is:

Type <number of signals> <kind of signals> <bit rate per signal in Mb/s>.

Type 18 DE 500 signals support high-performance boards plugged into a system backplane or cable links connecting proprietary physical packages. Symbols are sent bit-parallel, using differential drivers and receivers. High transmission rates can be achieved by having all signal drivers and receivers in the same integrated circuit package, which also contains high-speed queues, as illustrated in figure 1.

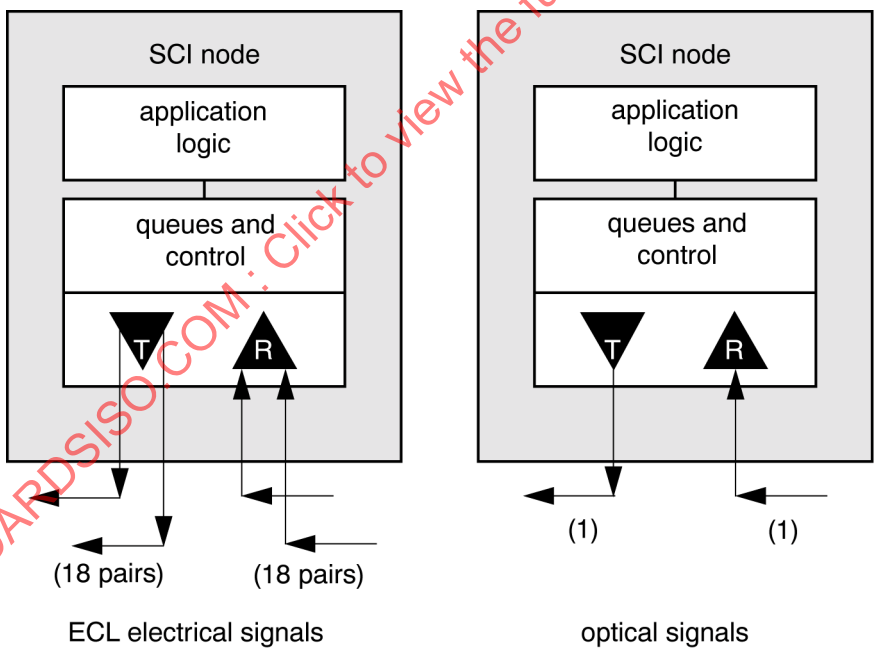


Figure 1 – Physical-layer alternatives

The initial interface chips were VLSI chips that included the transmitters, receivers, high-speed queues, and most of the cache-coherence protocols. Subsequent implementations generally removed the coherence logic, leaving that to the province of the system's memory controller. Several implementations initially ran at reduced speed for compatibility with standard CMOS processes. Some included the SCI interface as just a small part of a system chip that included processor and other application-specific logic. The complexity of the protocol is very low, with some implementors reporting that they used less than 25 k gates,

much less than some common low-end interface products. The inherent power dissipation of the SCI interface is less than that of interfaces to bused backplanes, since the differential signal levels are smaller (less than 1 V), there are fewer signals, and transmission impedances are significantly higher.

The Fiber-Optic Physical Layer Type 1-FO-1250 is intended to support longer-distance local communications (tens to thousands of meters). The fiber versions of SCI could be used to connect back-end peripherals to the central system, or could provide high-bandwidth communication between workstations and servers in a local computing environment. Packets are sent in a bit-serial fashion, as illustrated in figure 1.

Low-cost LEDs can support communication bandwidths of less than 1 Gb/s over short fiber hops. Higher-cost single-mode lasers and fibers are required for higher bandwidth communications over longer distances. Many applications will find it attractive to use coaxial cable instead of fiber for short hops, avoiding the optical/electrical conversion costs.

Fiber-optic interfaces are expected to consist of high-speed bipolar front-ends that convert between a high-bandwidth serial bit-stream and a lower-bandwidth symbol-stream. Lower-speed back-end circuits could be implemented in less expensive CMOS technologies.

New link standards will be defined from time to time to take advantage of advances in technology or to accommodate the needs of particular markets.

1.2.5 SCI node model

An SCI node needs to be able to transmit packets while concurrently accepting other packets addressed to itself and passing packets addressed to other nodes. Because an input packet might arrive while the node is transmitting an internally generated packet, FIFO storage is provided to hold the symbols received while the packet is being sent. Since a node transmits only when its bypass FIFO is empty, the minimum bypass FIFO size is determined by the longest packet that the node originates. Idle symbols received between packets provide an opportunity to empty the bypass FIFO in preparation for the next transmission.

Input and output FIFOs are needed in order to match node processing rates to the higher link-transfer rate. Since there is no facility for delaying the transmissions of symbols within a packet, each node ensures that all symbols within one packet are available for transmission at full link speed. Similarly the node is able to receive a packet at full speed. Since node application logic is not expected to match the SCI link speeds, FIFO storage is needed for both transmit and receive functions, as illustrated in figure 2.

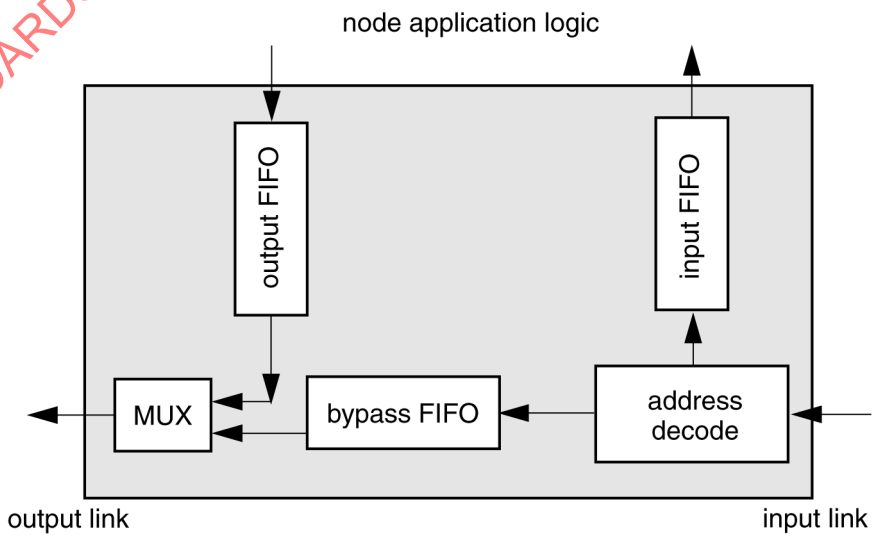


Figure 2 – SCI node model

1.2.6 Architectural parameters

The SCI system is generally considered to have a 64-bit architecture, because of its address size (16 bits for node selection plus 48 bits for use within each node). The data width is less constrained, however. SCI usually sends data in multiples of 16 bytes, and the most significant size assumption is the 64-byte coherence-line size.

SCI is described in terms of a distributed shared-memory model with cache coherence, because that is the most complex service SCI provides. However, SCI also provides message-passing mechanisms and noncoherent transactions for those who need or prefer them. All of these transactions can be dynamically mixed in one system as desired.

1.2.7 A common CSR architecture

Control and status registers (CSRs) are an important part of the proposed standard. The CSR definitions are essential for all initialization and exception handling. A few of the CSRs are SCI-specific, but the majority of the necessary definitions are provided by the CSR Architecture standard (IEEE Std 1212-1991)[1].

SCI uses the 64-bit-fixed addressing model defined by the CSR Architecture. The 64-bit address space is divided into subspaces, one for each of 64 K equal-sized nodes, as illustrated in figure 3. When compared to other address-extension schemes, the fixed address-field partitioning dramatically simplifies packet routing; however, it complicates software's memory-mapping model, since the memory addresses provided by different memory nodes can no longer be contiguous.

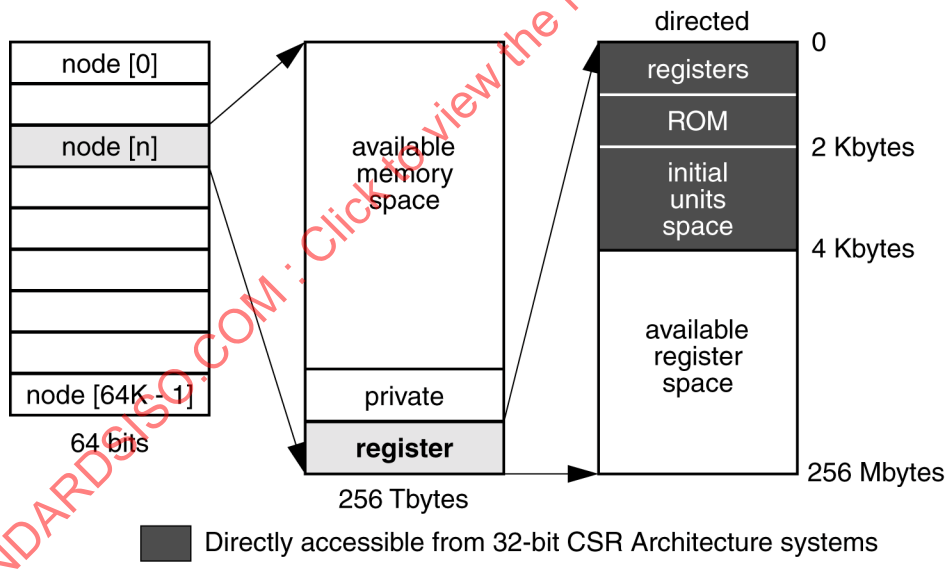


Figure 3 – 64-bit-fixed addressing

The upper 16 bits of the address specify the responder *nodeId* value; the remaining 48 bits specify the address-offset in the addressed node. The highest 256 Mbytes of each node's 256 Tbytes contain the CSR registers as defined in the CSR Architecture. Since SCI's broadcast transactions are block moves with no responses, only the directed (i.e., not broadcast) CSR registers are supported.

[1] Information on references can be found in 2.1.

Only a portion of the 64-bit address space is accessible from 32-bit systems bridged to SCI. The initial 4 Kbytes of each node's directed CSR address space as defined by the CSR Architecture could be directly mapped into 32-bit addresses, using the 10 bus-address and 6 module-address bits to form an SCI node address. In addition, a small portion (3.5 Gbytes) of the memory address space in node[0] could be directly mapped from the 32-bit memory address space. However, the address-map conventions used by bridges to other buses are beyond the scope of the SCI standard.

1.2.8 Structure of the specification

This specification covers a great deal of new territory, and has required some new approaches for presenting the material in a way that is precise and not easily misunderstood. Much of this International Standard is tutorial and explanatory in nature, to develop the way of thinking and the level of understanding needed to properly interpret and use the precise specification. The most important part of this standard is the packet protocol. Packet transmission is in turn implemented on some physical signalling layer, and that in turn may be incorporated into a standard mechanical package.

Except for the packet formats and physical implementation specifications, such as module, connector, power and signal levels, this specification is expressed in the C computer language. English text should be considered explanatory, and C listings, the definitive specification. Though C is known to have some ambiguities (such as the order of evaluation of parts of certain expressions), they are easily avoided in this application. In addition to making this specification unambiguous, another significant advantage of the C specification is that it is executable so that it can be incorporated into other software to test the operation of the specification under simulation or to test a real implementation of the specification.

1.3 Interconnect topologies

1.3.1 Bridged systems

To ensure the early availability of the wide range of I/O interface boards that any system needs in order to become accepted and useful, the SCI standard was heavily influenced by the need to bridge to other system buses. Conversions between SCI and other bus standards are performed by bus bridges, as illustrated in figure 4.

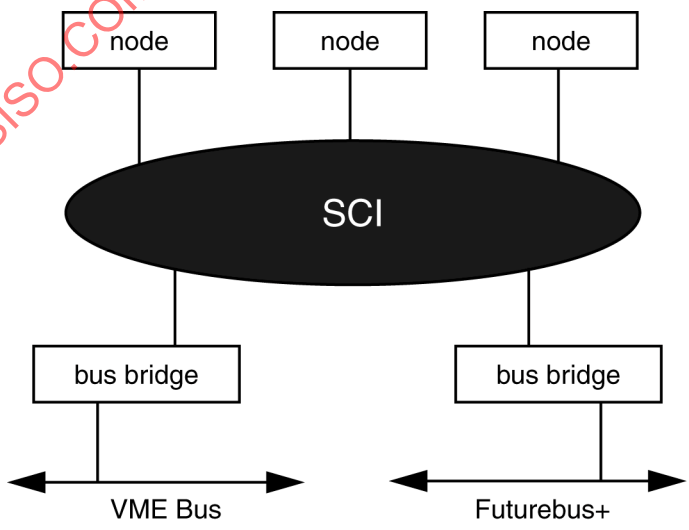


Figure 4 – Bridged systems

Indivisible uncached lock transactions (such as swap, compare&swap, fetch&add) are supported, but not implemented as indivisible read-modify-write transaction sequences. Since indivisible sequences are hard to implement in large switches, indivisible lock operations are performed at the responder upon request. A standard set of lock transaction subcommands is defined in order to communicate the intent of the requester to the hardware at the responder that will carry out the operation. Bus bridges may translate these lock transactions into indivisible sequences where appropriate.

Most remote DMA adapters generate uncached bus transactions; bus bridges can convert these into coherent transaction sequences. If the remote bus supports coherent transfers, the bus bridge can also convert between coherence protocols. Futurebus+R (see [B11][2], [B3], and [B4]) and SCI have the same coherence line size, which simplifies that conversion process.

1.3.2 Scalable systems

SCI protocols are scalable, which means that they are efficient and cost-effective for uses ranging from low-end desktop computers to high-end massively parallel processing (MPP) systems. One future vision of a massively parallel processor consists of large numbers of single-board computers connected through a high-performance switch.

To make this vision a reality, SCI is designed to be used in simple passive backplane configurations, or as the basis for constructing switches, or as the interface between multiprocessor boards and vendor-dependent proprietary high-performance interconnects. Such configurations are introduced in the following clauses.

1.3.3 Interconnected systems

SCI is based on packets sent from one node to another over unidirectional links. This specification defines a way to send these packets 16 bits at a time over short distances (on the order of meters), and one bit at a time over longer distances (on the order of a kilometer).

The bit-serial version of SCI makes use of fiber-optic links or short coaxial cables. It might be used as a high-performance peripheral bus connecting storage servers to back-end processors, or as a local-area bus connecting distributed workstations and file servers.

1.3.4 Backplane rings

The simplest SCI interconnect is a single ring. Larger configurations could consist of multiple rings connected through bridges. The highest performance configurations would probably be based on switching interconnects, like the butterfly switch. From a node interface perspective, the interface to a simple ring and to a complex switch is the same (one input link and one output link). The lowest-cost SCI configuration makes use of a passive backplane; the nodes are electrically connected as a ring. The ring connection could join adjacent slots (which results in one long link to connect the ends) or alternate slots (to shorten the maximum link length). On a sequential ring, a node's physical and electrical neighbours are the same, as illustrated in figure 5.

[2] The numbers in brackets preceded by the letter B correspond to those of the bibliography.

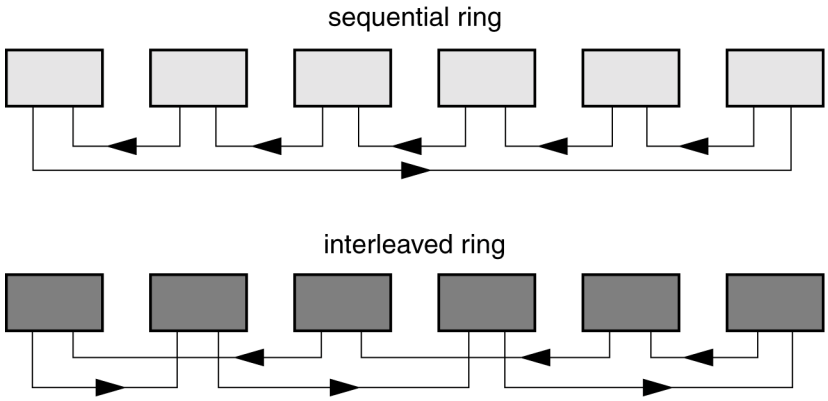


Figure 5 – Backplane rings

On an interleaved ring a node's physical and electrical neighbours differ. Even-numbered boards attach to one ring direction; odd-numbered boards to the other, thus minimizing the maximum distance between nodes. To support partially populated topologies, implementations are expected to use pass-through cards in empty slots, to provide jumper-card pairs for bypassing empty slots, or to use self-bridging connectors (that short inputs to outputs when the slot is empty).

There is also provision for doubled (or even trebled) SCI connections to a module, making bridges and redundant fault-tolerant systems possible. With multiple rings arranged so that at least one ring skips any given slot, one can maintain partial system operation even when one module is removed – the rings connected to that slot are broken, but the other rings can connect the remaining modules via bridges.

For some applications it may be desirable to use SCI signals on cable links to connect devices that do not fit conveniently into the standard SCI modules.

1.3.5 Interconnected rings

Since the SCI protocols have been designed to minimize the transit time for packets that pass through a switch from one ringlet to another, they can be readily applied to multiple-ring topologies. For example, a grid of processors can be easily and efficiently interconnected by horizontal and vertical ringlets, as illustrated in figure 6. In this illustration, each processor has two SCI interfaces, one interface attaches to the horizontal ringlet and the other attaches to the vertical ringlet.

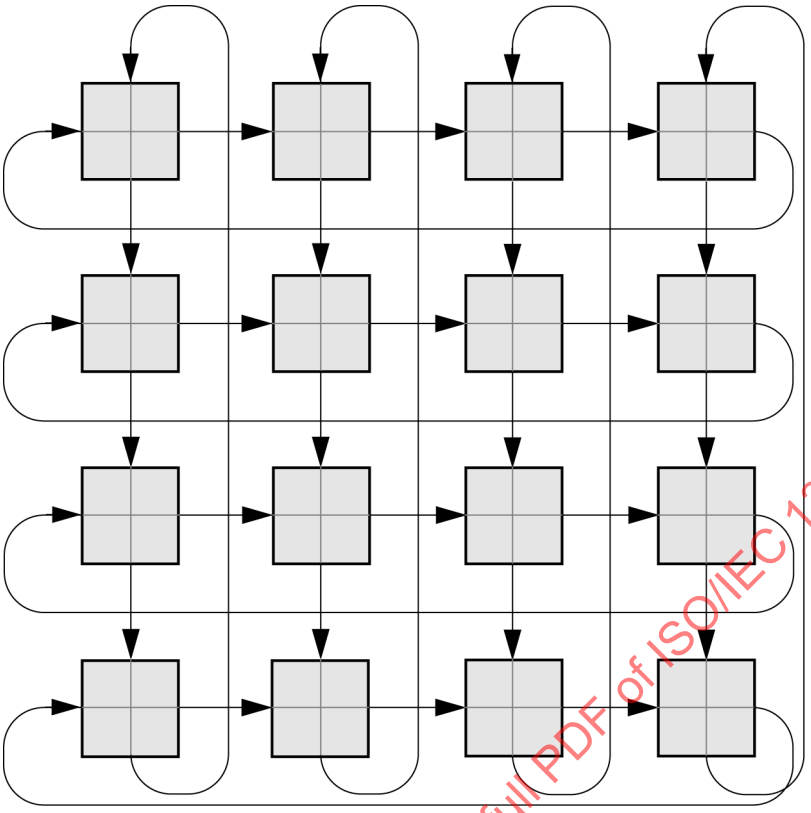


Figure 6 – Interconnected rings

Additional dimensions (for example, a 3-D cube) can be supported by increasing the number of ports on each processor (one for each dimension). Such structures are known as *k*-ary n-cubes, where *k* is the number of nodes on each ringlet and *n* is the number of dimensions. For a fixed number of processors, the number *k* can be increased to reduce the cost of the switch elements or may be decreased to reduce the contention on each ringlet.

1.3.6 Rectangular grid interconnects

SCI can also be used as an interconnect to form grids of processors. Nodes with four SCI interfaces can form a bidirectional interconnect, where different ringlets connect each node to its adjacent neighbours. Nodes with two SCI interfaces can form a unidirectional interconnect, where the ringlets form squares of nodes, as illustrated in figure 7.

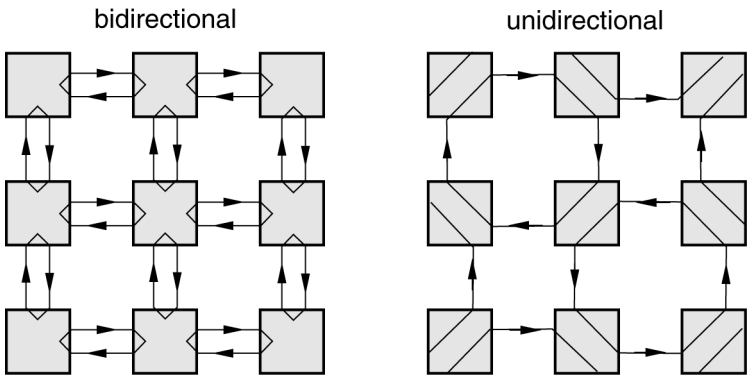


Figure 7 – 2-D processor grids

1.3.7 Butterfly switches

SCI can also be used to implement butterfly-like interconnects. Before SCI, these NlogN switches were generally implemented with a unidirectional data transfer and a reverse flow-*control* signal. The switch is wrapped around, so one processor node appears to connect to both sides of the switch.

SCI ringlets can be used to implement such switches by partitioning the transmission paths into separate ringlets, horizontal and diagonal, as shown in figure 8.

The dotted-line ringlet-completion path in this figure is an implied node-internal data path that connects one access port to another.

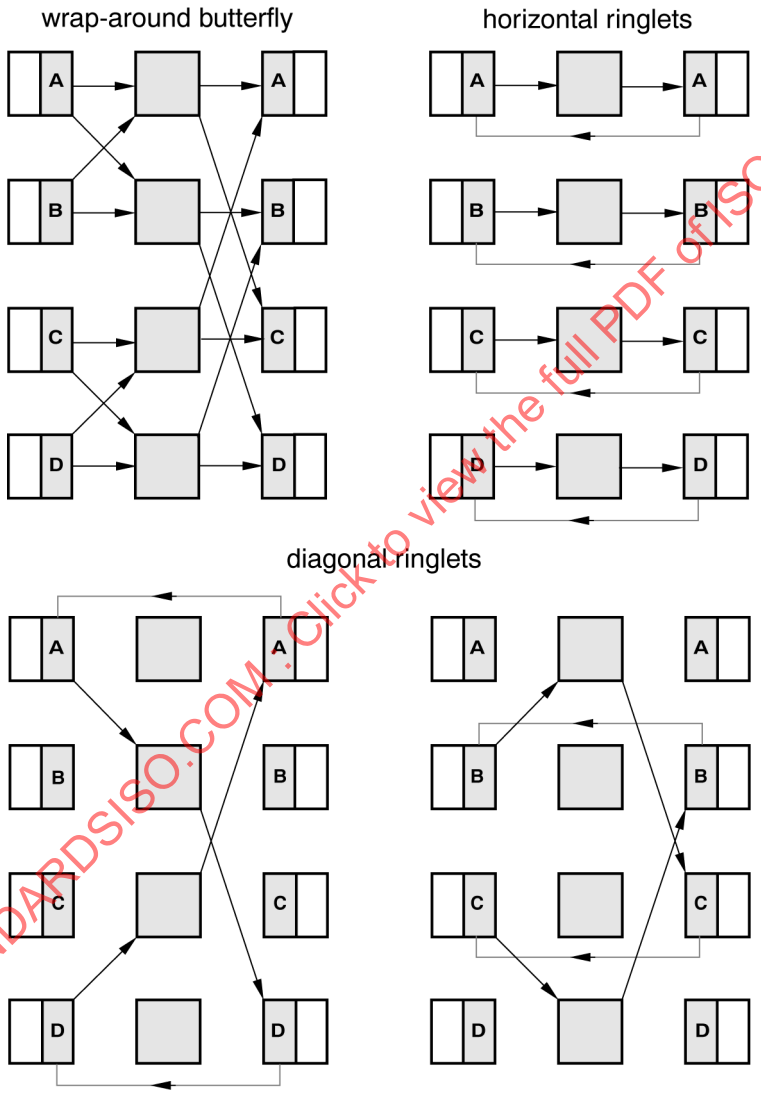


Figure 8 – Butterfly ringlets

1.3.8 Vendor-dependent switches

A switch may internally implement specialized vendor-dependent protocols to route SCI packets. Each node is attached to the switch by an SCI ringlet obeying normal SCI protocols, as shown in figure 9. SCI provides the interface between the nodes and the queues in the switch interfaces. To avoid deadlock, two queues are provided in each direction, one for requests and one for responses. This prevents requests from using up all the queue space and thus blocking completion of their responses. This strategy is followed throughout SCI.

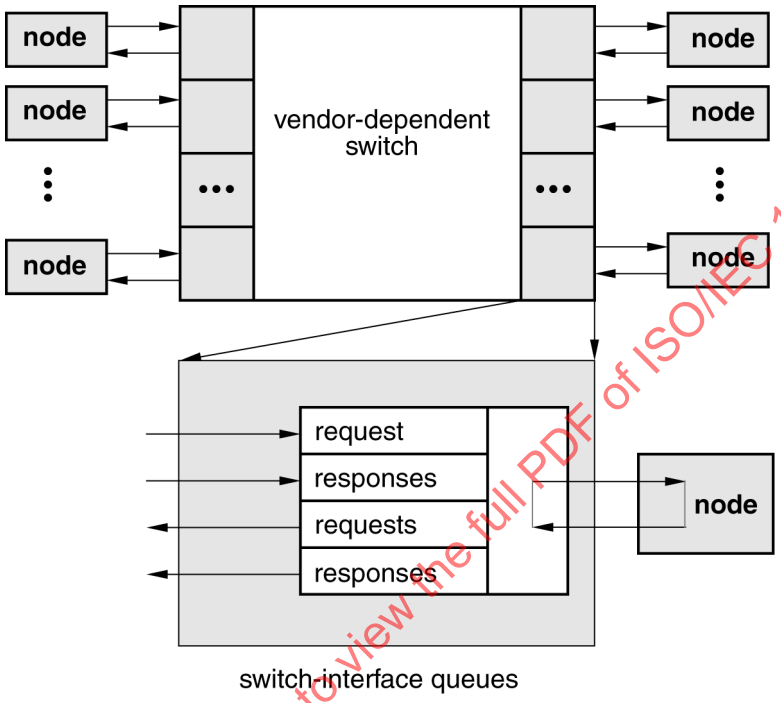


Figure 9 – Switch interface

1.4 Transactions

Transactions are performed by sending packets from a queue in one node to a queue in another. A packet consists of an unbroken sequence of 16-bit *symbols*. It contains address, command, and status information in a header, optional data in one of several allowed lengths, and a check symbol. When a packet arrives at a node to which it is not addressed, it is passed on to the next node with no change except possibly to the flow control information in the header. When a packet arrives at its destination address it is stored by that node for processing, and is not passed on to the next node.

An SCI packet originates at a source and is addressed to a single target. In going from source to target the packet may possibly pass through intermediate nodes or agents (explained later). Such single-requestor/single-responder protocols are highly scalable.

Transactions are initiated by a *requester* and completed by a *responder*. Transactions consist of two subactions. During the *request subaction* address and command are transferred from requester to responder. The *response subaction* returns completion status from responder to requester. Depending on the transaction command, data are transferred in the *request subaction* (writes), the *response subaction* (reads), or both subactions (locks).

A *subaction* consists of two packet transmissions, one sent on the output link and the other received on the input link. A subaction is initiated by a source, which generates a *send* packet. The subaction is completed by the destination, which returns an *echo* packet. Hence a typical transaction involves the transfer of four packets, as illustrated in figure 10.

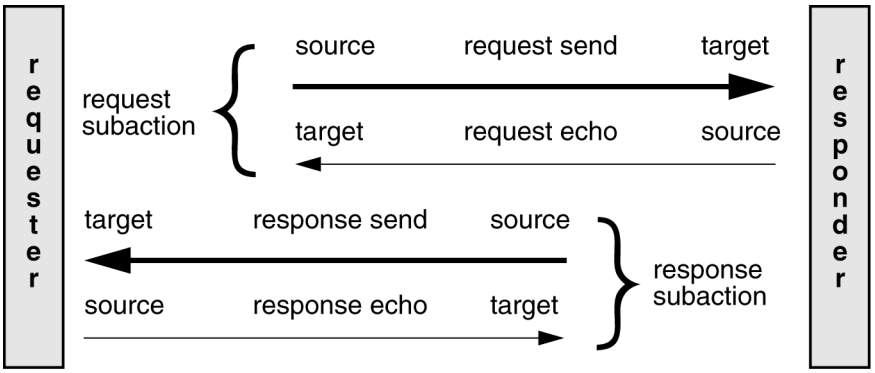


Figure 10 – Subactions

1.4.1 Packet formats

The first symbol of the header, *targetId*, contains the final target's nodeId, and is sufficient for a node to quickly recognize packets addressed to it. During the passage of a packet through an SCI system, intermediate agents look at the *targetId* symbol (and possibly other symbols) to route the packet, and intermediate nodes look at it to determine whether they should accept the packet. This and other packet symbols are shown, in simplified form, in figure 11.

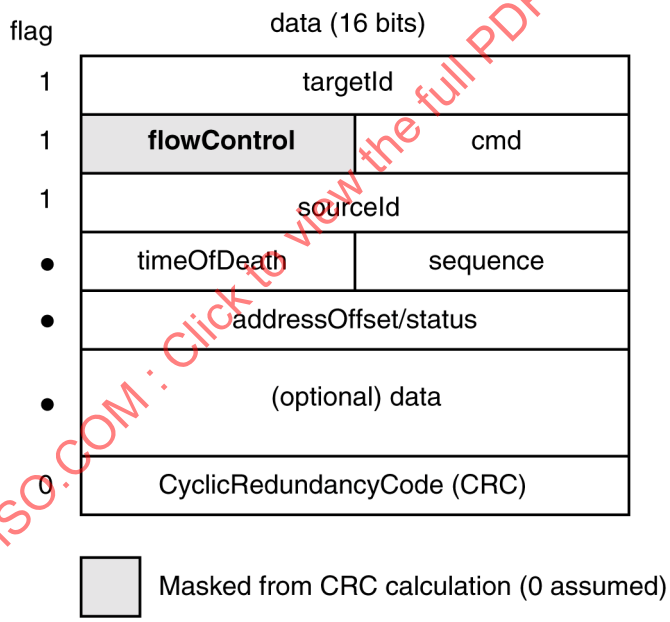


Figure 11 – Send-packet format, simplified

The second symbol, *command*, provides flow-control information and the transaction command field. The flow-control field, which contains localized flow-control information, may be changed many times before a packet reaches its destination. This information is excluded from the CRC calculation, so the CRC remains unchanged (and error coverage is not compromised) as the packet is routed toward its final destination.

The command field specifies the type of packet (read00, readsb, writesb, etc.). In a request-send packet, the command specifies the action to be performed by the responder. In a response-send packet, the command specifies the amount of data returned. In an echo packet, the command field indicates whether the corresponding send packet was accepted.

The third symbol contains the *sourceId*, allowing the target to identify the originator of the packet. All packets include a 6-bit sequence number (which distinguishes between multiple currently pending transactions from one requester). The location of this field differs for send and echo packets.

Appended to each packet is a 16-bit cyclic redundancy code (CRC), that is generated when the packet is created by the source, is optionally checked by agents, and is checked before the packet is processed by the target. The CRC is generated based on a parallelized version of the 16-bit ITU-T CRC.

Note that a flag bit is associated with each symbol. A zero-to-one transition of the flag bit is used to identify the first symbol of a packet. The one-to-zero transition of the flag bit occurs near the end of the packet (1 or 4 symbols before the packet's end, for echo and send packets respectively). A loss of link synchronization will generally cause improper flag patterns and CRCs.

Other information that is included in some packet types includes the following:

1) *Time of death*. The *timeOfDeath* is a time-stamp field in send packets, that specifies the time at which the packet should be discarded. This simplifies error recovery protocols by bounding the lifetime of all outstanding packets.
2) *Address offset*. The 48-bit *addressOffset* field in request-send packets transfers an address offset to the responder. Although this is often used to select specific memory or register locations, the interpretation of (most of) this field is responder-architecture dependent.
3) *Status.* The 48-bit status field in response-send packets returns the transaction status from the responder to the requester.
4) *Extended header.* A packet may include an additional 16 bytes of header. The presence of the extended header is signalled by a bit in the command field. A small portion (four bytes) of the extended header is defined for certain cache-coherence transactions. The remainder of the extended header is reserved for definition by future extensions to the SCI standard.
5) *Data bytes*. The data section contains a data block of 0, 16, or 64 bytes. SCI systems may optionally support 256-byte transfers for higher efficiency.

1.4.2 Input and output queues

Queues are used to hold SCI packets that cannot be immediately forwarded or processed at their intermediate or final destinations. The simplest responder node has two queues. The input queue holds request packets that have been stripped from the input link but have not yet been processed. The output queue holds response packets to be sent on the output link when bandwidth is available. These queues are illustrated in figure 12.

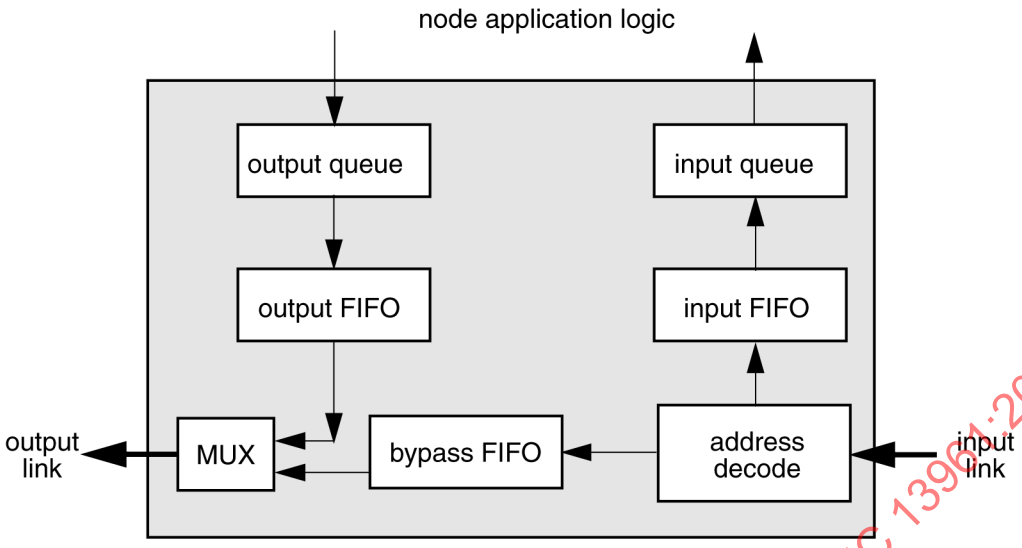


Figure 12 – Responder queues

Packets in the output queue are sent when the bypass FIFO is empty and the node's flow-control mechanism (see 3.6) permits it. Another packet (or packets) may arrive on the input link while an output packet is being sent. If they are not addressed to this node, the bypass FIFO holds these incoming packets for delayed transmission after the output queue packet has been sent. Thus, the bypass FIFO needs to be as large as the longest packet sent through the output queue.

While the bypass FIFO is nonempty, symbols arriving between packets (called idle symbols) are merged and their contents are saved for delayed retransmission. Thus, most idle symbols provide an opportunity to decrease by one the number of saved symbols in the bypass FIFO. When the bypass FIFO is empty, and the flow-control mechanism is re-enabled, another packet may be sent from the output queue.

When a send packet is emitted, the packet is saved in the output queue until a confirming echo packet is received. The addressed target node strips the send packet from the interconnect and creates an echo packet, which is returned to the source. There are two types of echo packet. If the target node can save the send packet, a done echo is returned. If the target node lacks queue space, it discards the send packet and returns a retry echo.

When a done echo is returned to the source the corresponding send packet is discarded (i.e., its queue space is freed for reuse). When a retry echo is returned to the source the corresponding send packet is resent. Resending after a retry echo packet is often called busy-retry, and the discarded send packet is said to have been busied by the destination node.

Note that send packets can be discarded by targets that have no space to save them, but returned echo packets are always accepted. Sources need to allocate space for echo packets before transmitting send packets.

1.4.3 Request and response queues

Many SCI nodes have requester as well as responder capabilities. To avoid system deadlocks on these full-duplex nodes, request and response subactions are processed through separate queues. Thus, each node logically has a pair of request and response subaction queues, as shown in figure 13.

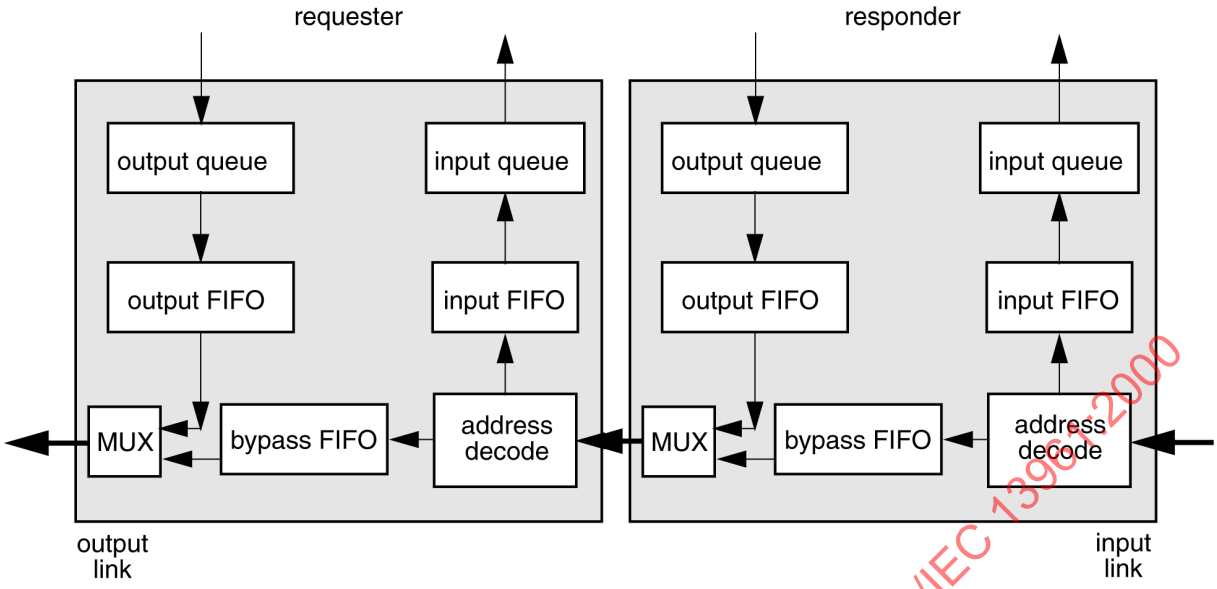


Figure 13 – Logical requester/responder queues

For performance and cost reasons, a single bypass FIFO is desirable. With suitable allocation protocols, the two bypass FIFOs can be merged into one, as illustrated in figure 14.

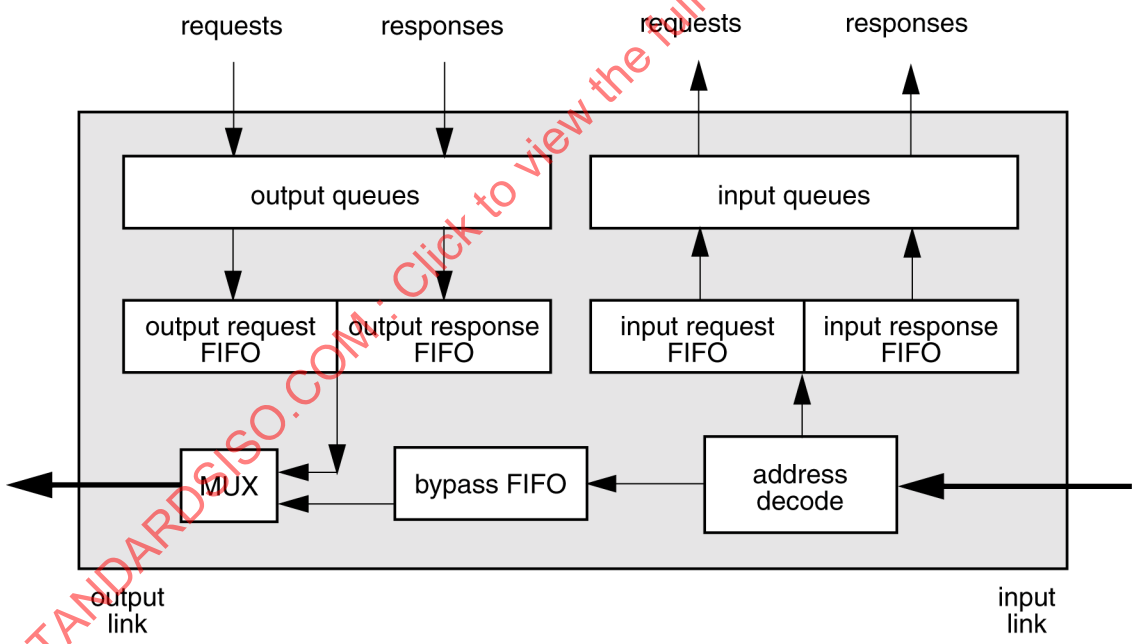


Figure 14 – Paired request and response queues

Pairs of input and output FIFOs are still required, to ensure that requests and responses can be processed independently. The input and output queues can be dynamically or statically allocated for holding requests and responses, if these queues can be bypassed when a FIFO entry is available. Forward progress is ensured because at least one entry is always available for holding input-request, input-response, output-request, and output-response packets respectively.

1.4.4 Switch queues

The concept of independent queue pairs can be extended to switches. For example, the queues in a simple bridge (suitable for use in hierarchical topologies) between two SCI ringlets are illustrated in figure 15.

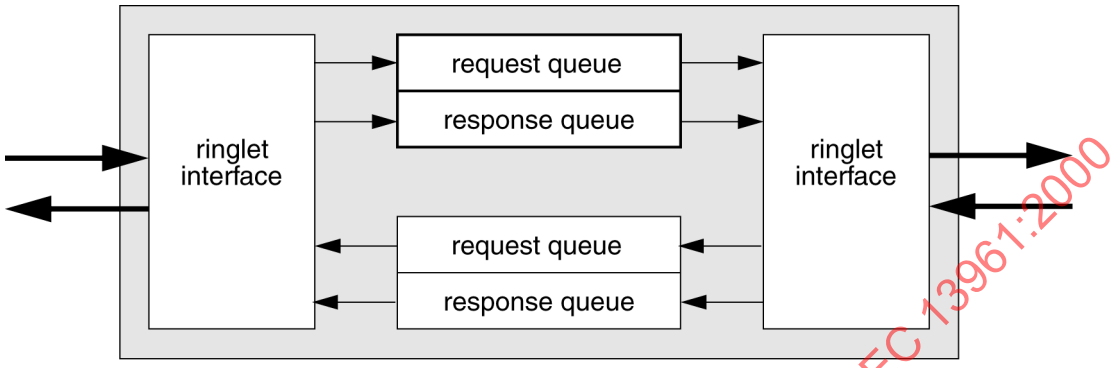


Figure 15 – Basic SCI bridge, paired request and response queues

More-complex topologies could have loops in the physical configuration (e.g., a toroidal topology formed by connecting the top and bottom edges and the right and left edges of a 2-D mesh). Additional queues may be needed to avoid hardware deadlocks due to possible circular dependencies in such systems.

1.4.5 Subactions

When requester and responder are on the same lightly loaded ringlet (i.e., local), a transaction involves four packet transmissions, as illustrated in figures 16 and 17. (Shading is used to indicate the queue that holds the relevant packet. The queue state in figure 16 is shown as it would be just before receipt of the illustrated packet.) The request subaction involves the transfer of a request packet from the requester to the responder (steps 1 and 2). The responder's processing involves the consumption of the request packet and the generation of a response packet. The response subaction involves the return of a response packet from the responder to the requester (steps 3 and 4).

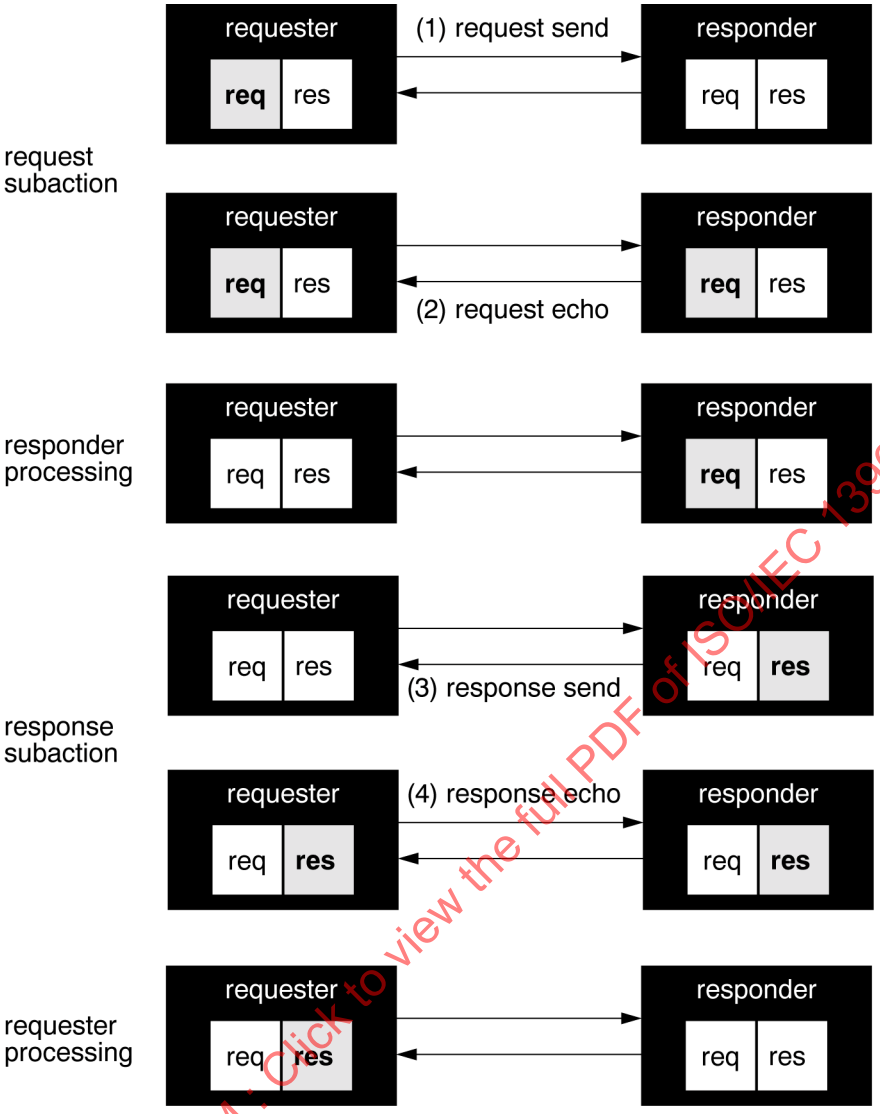


Figure 16 – Local transaction components

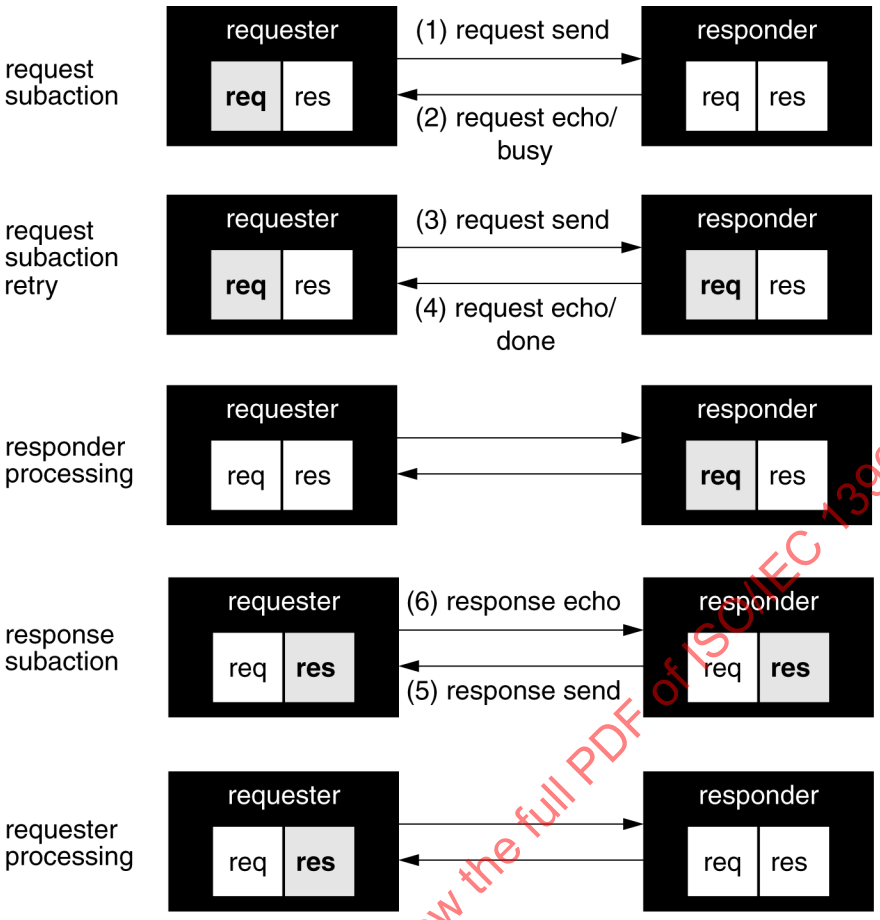


Figure 17 – Local transaction components (busied by responder)

Each subaction consists of a *send packet* (steps 1 and 3 in figure 16) that transfers information between a producer and a consumer and an *echo packet* (steps 2 and 4) acknowledging the receipt of the information. Each packet is sent between a *source* and a *target*. The producer is a source for request-send and response-echo packets and a target for request-echo and response-send packets.

The producer saves a copy of the request-send (or response-send) packet until a returned request-echo (or response-echo) packet confirms its acceptance at the consuming node. The echo packet may sometimes indicate that the consumer queues were busy (full) and that the send packet was discarded. These busied packets are retransmitted until they are accepted by the consumer. *Bandwidth allocation protocols* are used to guarantee that all producers will eventually transmit their send packets; *queue allocation protocols* guarantee that consumers will eventually accept these send packets (or a busied retransmission of them, see 3.7).

For example, consider a heavily loaded system, where there is contention for the shared responder subaction queues. If the responder's request queue is full, the first request-send packet may be busied and retransmitted as illustrated in figure 17. The queue state in this figure is shown as it would be just before completion of each illustrated subaction.

The first request-send packet (1) is busied by the responder, which initially has a full request queue. The request-send packet is discarded and the busy status (2) is returned in the first request-echo packet. Later another request-send packet (3) is sent from the requester to the responder and (in this example) is accepted; receipt of the request-send packet is confirmed by the status returned in the request-echo packet (4).

Although not illustrated in figure 17, either the request-send or the response-send packet may be busied many times, but will eventually be accepted. Simple ageing protocols guarantee that the oldest busied transactions are eventually accepted.

1.4.6 Remote transactions (through agents)

A packet starts at an original-producer (source) node, addressed to a final-consumer (target) node. For a remote transaction the source and target nodes are on different rings. The packet will then be accepted by consumer queues in intermediate agents (e.g., bridges or switches) for forwarding to the target. Each intermediate agent behaves like a producer when forwarding the packet to its final-consumer node. A given packet has only one final consumer, but may be processed by a number of consumer/producer pairs as it moves from agent to agent.

A remote transaction is initiated by the requester as though it were local. The packets forming the transaction are queued and forwarded by intermediate agents. To the requester, the agent behaves like a responder; to the responder, the agent behaves like a requester. An agent typically acts on behalf of many nodes, and thus accepts packets with any of a set of addresses (a different set on each side). The steps involved in the completion of a remote SCI transaction are illustrated in figure 18, for a lightly loaded system (no subaction queues are full) with a single intermediate agent. In this figure, the queue state is shown as it was before the start of each illustrated subaction.

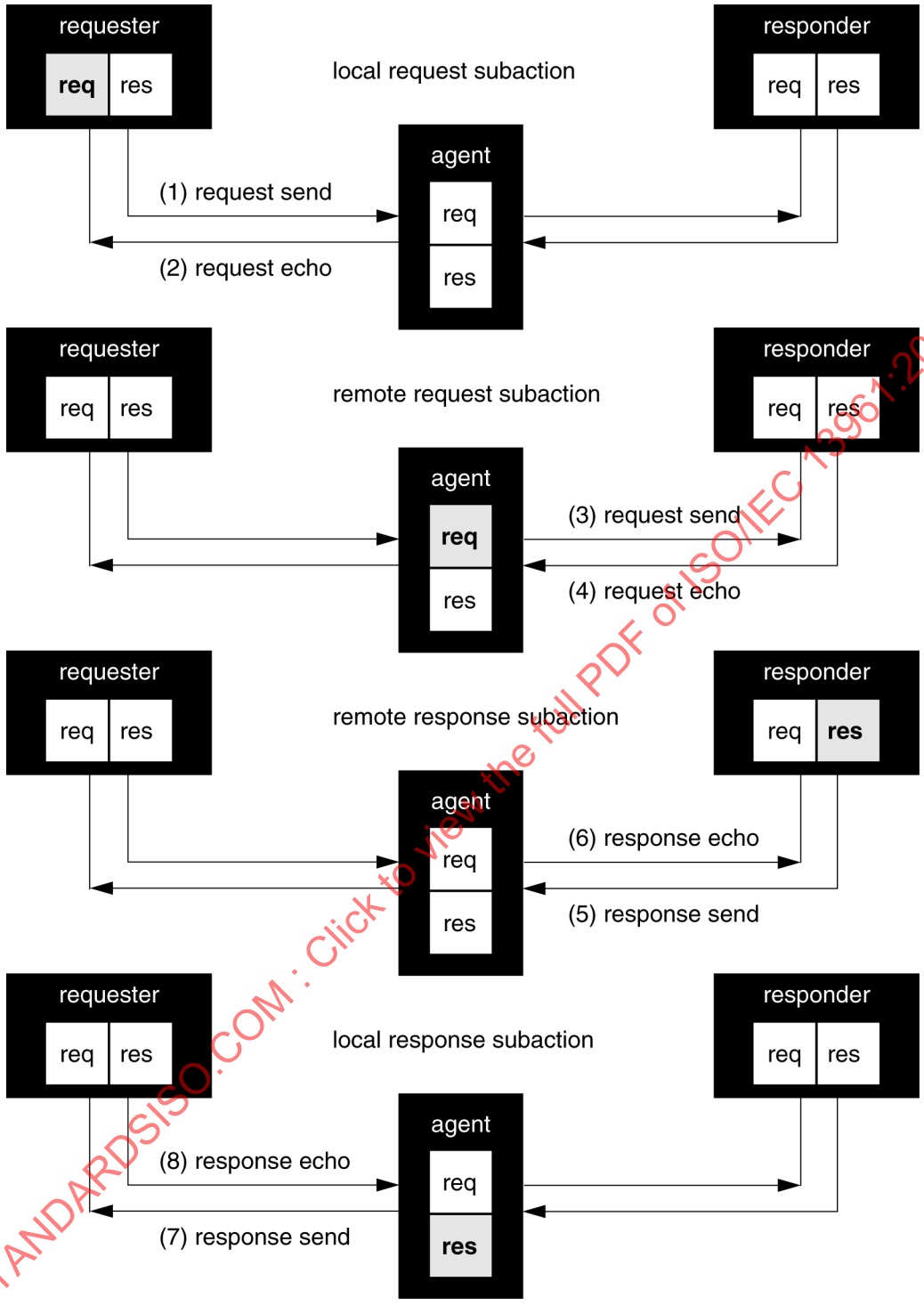


Figure 18 – Remote transaction components

The initial request subaction (1 and 2) transmits the request packet from the local requester to the intermediate agent. The remote request subaction (3 and 4) forwards the request packet from the agent to the remote responder. After confirmation that the request has been accepted by the responder, the intermediate agent discards subaction information (residual history); its send buffers can immediately be reused for other purposes.

Note that subactions do not care whether they are local or remote; only agents need know that the subaction is not local. Note also that echoes merely confirm delivery to the next agent, not necessarily to the final consumer, and that queues in agents take responsibility for further transmission.

After the request has been processed by the responder, the remote response subaction (5 and 6) transmits the response packet from the remote responder to the intermediate agent. The local response subaction (7 and 8) forwards the response packet from the agent to the original requester. After confirmation of the response being accepted by the requester, the responder and the intermediate agent have no queued send packets; their send buffers can be immediately reused for other purposes.

An active agent can be pipelined; forwarding of the request-send packet (3) can begin before the request-echo packet is returned (2) to the requester. The same is true for the response; the response-send packet (7) can begin before the response echo (6) is returned to the responder. Note that an agent must also keep a copy in its queue until echo confirmation has occurred.

This mechanism applies in general for any number of intermediate agents. The routing of packets in a system is determined by the set of agents, each with its own set of addresses to accept.

1.4.7 Move transactions

A *move* transaction is like a write transaction, with the exception that no response subaction is returned. A move transaction is expected to be used when large amounts of data are transferred and timeliness is more important than confirmed delivery, such as for repetitive data transfers to a video frame buffer. Although more efficient than a write transaction, the lack of a response (which provides the responder's completion status) limits move-transaction uses to specialized applications or constrained configuration topologies.

A move transaction is a specialized noncoherent write transaction that has a request subaction but (for improved efficiency) no response subaction. Flow control, performed at the subaction level, ensures that request-send packets are not discarded when attempting to enter congested queues. However, transmission errors (which are normally reported in response subactions) will not be detected by the standard lower-level protocols (but could be by application-specific higher-level ones). The steps involved in the completion of a remote SCI move transaction are illustrated in figure 19, for a lightly loaded system.

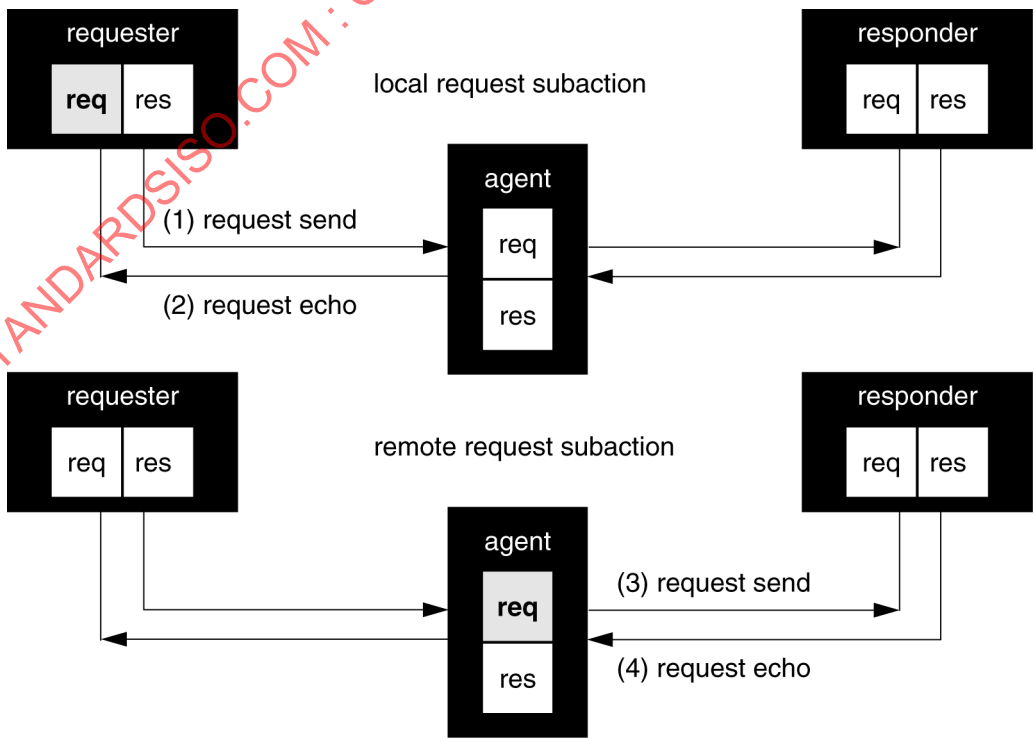


Figure 19 – Remote move-transaction components

The local request subaction (1 and 2) transmits the move-request packet from the requester to the intermediate agent. The remote request subaction (3 and 4) forwards the move-request packet from the agent to the responder. The final agent is informed when the request is queued in the responder, but the requester receives no such confirmation. Since transactions may be reordered while passing through an interconnect, there is no standard way for either the requester or the agent to confirm when or if the move transaction has completed.

Since move transactions have no response, there is no standard way to return agent or responder error status to the requester. Intermediate agents and responders are expected to provide mechanisms for logging these errors, but these error logging mechanisms are beyond the scope of the SCI standard.

Since move transactions have no confirming response, there is no reliable way to use their *transactionId* values to differentiate between distinct move transactions. Thus, producers with two or more active move transactions could become confused, when two or more active move transactions generated the same request-echo packet. To avoid these confusions, producers are expected to temporarily inhibit transmission of new move requests when their echoes could be confused with those that are already expected from other active requests. (An active request is one that has been sent but whose echo has not been returned).

1.4.8 Broadcast moves

Some applications can benefit from the optional capability of efficiently broadcasting a packet to multiple destinations using a single transaction. Application examples include some kinds of image processing such as HDTV (high-definition television) signal processing, systolic processing arrangements, and massively parallel architectures such as neural networks. Special protocols are used to ensure forward progress, since a move transaction might sometimes be accepted by some of the nodes but not all (when some of the consumer queues are temporarily full).

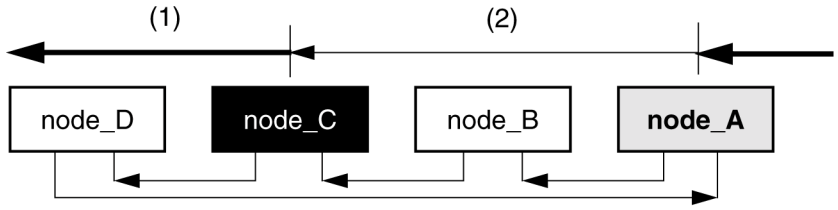
In the worst case, a broadcast consumes the same bandwidth as sending the packet repeatedly to all its *N* destinations. In the best case, it reduces the consumed bandwidth by a factor of *N*, when there are *N* broadcast-capable nodes on the ring. Note that broadcast transactions are ignored by nodes that do not support this optional capability.

Several subaction command codes are allocated for broadcast functions. Half of these codes are for starting broadcast messages; the other half are for the resumption of a previously initiated broadcast. Except for having multiple effective target addresses, broadcast (*start* and resume) transactions are functionally equivalent to directed moves (they do not have a response subaction and they do not participate in cache coherence).

On a local ringlet, a start-broadcast packet is sent from the broadcaster to itself, with a special start-move command code (*smove*) that enables the eavesdrop capability on other ringlet-local nodes. The command code for the broadcast is decoded by all those nodes that have broadcast capability; the *smove* is ignored by nodes that do not support broadcast, based on its target address.

If all acceptance queues are free, the *smove* packet returns to its source (*node_C*) and is stripped. The originating broadcaster *node_C* recognizes that no echo is needed, but updates its send queues as though one were received. The strategy of not echoing one's own send packets is efficient, simplifies the allocation-priority sampling protocols, and applies to directed send packets as well.

If an eavesdropper's acceptance queues are full, it strips (1) the packet and returns (2) an echo, as illustrated in figure 20.

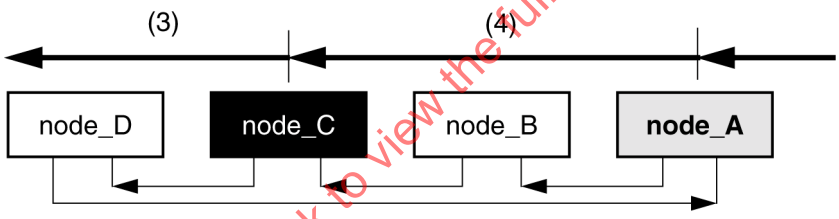


1) smove send packet, target = source = R
2) echo packet, target = R, source = A

Figure 20 – Broadcast starts

In this example, the broadcast transaction has been originated by a remote node (*node_R*) and is being forwarded to this ringlet through *node_C*. Just as for other SCI transactions, the send packet's sourceId field is provided by the original source, not the local agent.

When the busied transaction is re-sent (3) by *node_C*, the retried packet contains the resume-broadcast command (*rmove*) and is directed to the node that returned the echo (*node_A*). The resume-broadcast packet is directed in the sense that it is ignored by other ringlet-local nodes. While the acceptance queues are full, the *rmove* packets are stripped and echoed by *node_A* and re-sent by *node_C*. When the acceptance queues become free, *node_A* converts the packet into its original *smove* form (4) for distribution to the downstream nodes, as illustrated in figure 21.



3) rmove send packet, target = A, source = R
4) smove send packet, target = source = R

Figure 21 – Broadcast resumes

When the *rmove* transaction is accepted by *node_A*, its target address is restored to the value provided by the source and its command value is restored to the original *smove* value. When this queued packet is passed to an adjacent ringlet it looks like the original broadcast. Restoring the resume-broadcast to its start-broadcast form also requires regeneration of the CRC value since the target and command fields change. Note that waiting for the sourceId before converting the packet to its original form requires two extra levels of pipelining in the node's packet processing (one more than needed by a ring scrubber).

The *smove* transaction completes when it is stripped by the originating *node_C*. This broadcast is never busied, even if *node_C*'s acceptance queues are full. This is because the broadcast actions were already performed on *node_C*, before the send packet was originally transmitted.

1.4.9 Broadcast passing by agents

The routing algorithms for an agent's directed and broadcast transactions may differ, to prevent broadcasts from travelling from one ringlet through a switch or a bridge to another ringlet and back again, thereby circulating in the system indefinitely.

For example, consider two ringlets connected to each other via two distinct symmetric bridges. A broadcast could start on one ringlet, propagate to the first bridge, pass to the other ringlet, circulate around to the second bridge, and then propagate back onto the original ringlet. There would be an infinite loop and an increasing number of packets the original packet would go past the bridge while the bridge creates a new one.

Normally an agent needs only to look at the *targetId* and its own internal routing tables to decide whether or not to pass a packet to its remote side. That is, routing decisions depend entirely on packet destinations. However, agents that support broadcast transactions look at the *sourceId* field in broadcast packets, and broadcasts have a special routing table. The table indicates which broadcasts are to be passed, based on *sourceId* comparisons. When properly initialized, such tables prevent the return of broadcasts that previously left this ringlet.

Such broadcast routing tables need to be set up at initialization time. Proper setup of these routing tables involves treating each node in the system as the potential root of a tree whose branches are formed by the other ringlets and agents in the system. System initialization procedures are expected to put these broadcast tree routes into the broadcast tables with the specific purpose of creating efficient paths that have no loops. These procedures may optionally take into account traffic patterns in the system in order to optimize path assignments where path choices exist.

Note that the implementation of the broadcast routing table in an agent, like the normal routing table, need not be a table lookup. In some configurations, the routing can be done algorithmically with *sourceId* range-checking logic. However, the specification of the routing tables or range-checking logic is beyond the scope of this standard.

1.4.10 Transaction types

Several types of transactions are supported, including reads, writes, and locks. The primary difference between these transactions is the amount of data transferred, and in which subaction is as illustrated in figure 22.

	request		response	
readxx*	header		header	0,16,64,256
writexx*	header	16,64,256	header	
movexx*	header	0,16,64,256		
eventxx*	header	0,16,64,256		
locksb	header	16	header	16

Note: **xx** represent one of the allowed data block lengths
(number of data bytes, on the right after the header)

Figure 22 – Transaction formats

Readxx transactions copy data from the responder to the requester; writexx transactions copy data from the requester to the responder. Readxx and writexx transactions both have responses, which are used to return the completion status from the responder.

Movexx transactions copy data from the requester to the responder. Movexx transactions are more efficient than their nearly equivalent writexx transactions, but there is no provision for returning the completion status from the responder.

Eventxx transactions copy data from the requester to the responder. Eventxx transactions have no flow control, and are never rejected. Therefore, the protocols in this standard can provide no guarantee of data delivery. Event00 is used for synchronizing time-of-day clocks. If the other Eventxx transactions are used for moving data, the system designer must provide sufficient storage for that data outside the normal request-queue storage that is managed by SCI's flow control mechanisms.

Locksb transactions copy data from the requester to the responder. The responder indivisibly updates the affected address, based on the command value and the request-subaction's data. The response subaction returns the previous (unmodified) data and status. These non-coherent transactions support fetch&add as well as compare&swap update operations.

Shorter transactions, such as a 1-byte write transaction, are formatted as 16-byte transactions, but only a portion of the data is used. These selected-byte read and write transactions are useful when accessing *control* registers (which are less than 16 bytes in size, and whose side-effects are sometimes dependent on the transaction size).

1.4.11 Message passing

SCI supports message passing, as defined by the CSR Architecture. A standard noncoherent write64 transaction is used to send short unsolicited messages to a specified CSR register within the target node. Two techniques for sending longer messages can be used:

1) *Concatenated packets.* Two or more 64-byte write transactions are concatenated to form a longer message.
2) *Indirect pointer.* A long message transfer (from A to B) is initiated by a short unsolicited message from A to B. This message includes a pointer to the longer message, which remains stored in memory at A. After processing the message pointer, the processor on node B reads the long message from node A.

To simplify flow-control protocols (and buffer allocation), the indirect-pointer approach is recommended.

1.4.12 Global clocks

The SCI standard supports global time synchronization, as defined by the CSR Architecture. SCI nodes can maintain local clocks (formatted as 64-bit integer-seconds/fraction-seconds counters). Hardware provides mechanisms for detecting drifts between clocks, and software is responsible for correcting the drifts as they are detected. Several expected uses of the clocks are as follows:

1) *System debugging.* If the optional trace feature is implemented, the route of a packet with its trace-bit set can be reconstructed by logging (with an accurate time stamp) packet arrivals at switching points in the interconnect.
2) *Time of death.* If the optional *timeOfDeath* value is provided in the packet header, stale send packets can be safely discarded before they might be misinterpreted.
3) *Real-time data.* A global clock can be used to synchronize the activities of multiple data-acquisition nodes (such as A/D and D/A converters).

On a traditional backplane, a *clockStrobe* signal can be broadcast to synchronize clocks on observing nodes. Clock synchronization on SCI is more complex, since signal paths are daisy-chained or switched rather than bused (see 3.12.4.1).

1.4.13 Allocation protocols

Depending on system configurations and dynamic loading conditions, the cumulative bandwidth requirements of multiple requesters can exceed the capacity of a shared interconnect or the bandwidth of a shared responder. When the cumulative bandwidth exceeds the available bandwidth, allocation protocols apportion the oversubscribed resources to the multiple requesters.

Most of the bandwidth is (optionally) apportioned unfairly to the highest-priority transactions. However, a small portion of the bandwidth is always apportioned fairly, as illustrated in figure 23. There are four priority levels: 0 through 3 are the lowest through highest priority respectively. The allocation protocols allocate most of the bandwidth (approximately 90 %) to those transactions with the highest priority that is currently being used; the remaining bandwidth is allocated fairly to those transactions having priorities less than the current highest priority.

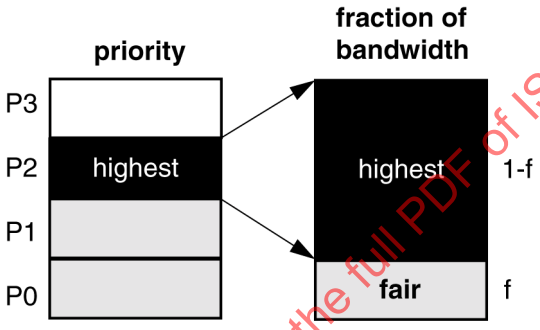


Figure 23 – Bandwidth partitioning

For the lower-priority nodes, the relative node priority has no effect on the allocation of this bandwidth. However, under dynamic loading conditions, the higher-priority nodes are likely to become the highest-priority nodes more often, which then increases their apportioned bandwidth.

Although this partial fairness scheme complicates allocation protocols, having even a little guaranteed bandwidth fairly allocated simplifies SCI in other ways, which include the following:

1) *Forward progress.* The impact of transient hardware or software priority inversions is minimized. A high-priority process can be temporarily blocked by a low-priority process without deadlocking the system.
2) *Deterministic timeouts.* For any system configuration, deterministic worst-case transaction timeout values can be calculated. These values are necessary for initializing the timeout hardware.
3) *Queue-allocation protocols.* Partial fairness bounds the time limit for retrying busied transactions. This simplifies queue-allocation protocols, which wait for retries of previously busied transactions.

Bandwidth allocation protocols apportion bandwidth on a local ringlet. When many *requesters* and many responders are on the same ringlet, allocation protocols apportion the shared ringlet bandwidth. SCI bandwidth-allocation protocols are similar in effect to bus arbitration protocols.

Queue allocation protocols allocate queue entries in a responder or switch component. When many requesters access the same responder, the responder's allocation protocols allocate the limited responder-queue bandwidth. SCI queue-allocation protocols and bus-bridge busy-retry protocols are similar in function.

Bandwidth allocation protocols apportion bandwidth when the interconnect is the bottleneck; queue allocation protocols apportion bandwidth when a shared responder (or intermediate agent) is the bottleneck. These bottlenecks are illustrated in figure 24. Shading indicates congestion.

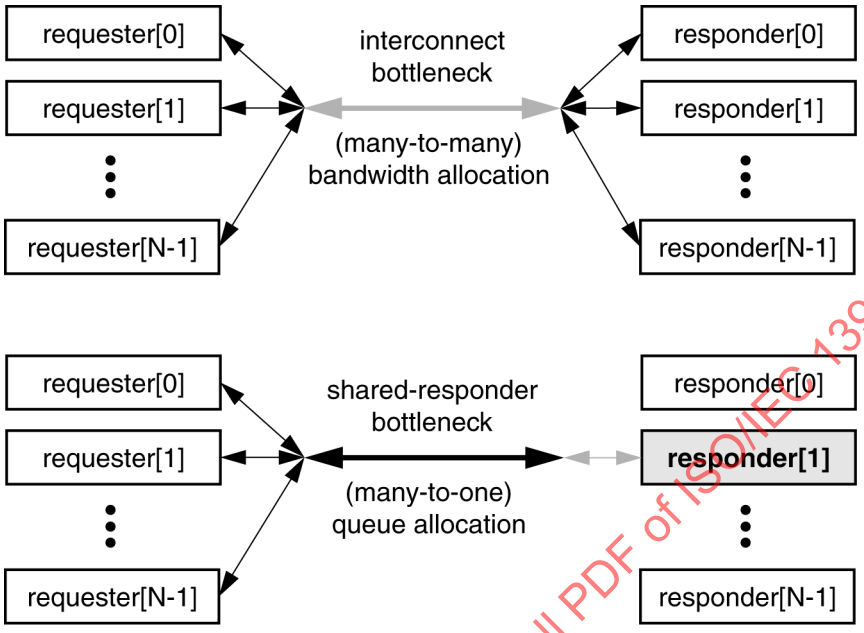


Figure 24 – Resource bottlenecks

Requester nodes assign a two-bit transaction priority to their transactions. This transaction priority affects the bandwidth and queue allocation protocols, which assign most of the available bandwidth to the highest-priority nodes. A send packet's effective priority is usually equal to its transaction priority, but may be temporarily increased because of higher-priority packets that are blocked behind it. This priority-modification process is called priority inheritance. Priority inheritance is supported by SCI, whose send packets contain the transaction priority as well as the effective priority.

Allocation of prioritized bandwidth has a delayed effect. Transmission of future packets is inhibited based on the state of other nodes in the recent past. On large systems, these protocols can effectively apportion bandwidth but have little effect on reducing the latency for random accesses.

Traditional backplane bus arbitration takes longer, but simultaneously senses the priority of all nodes, so priority information is more current and more directly affects latency. Note that this bus virtue comes at the price of severely limiting the bandwidth and the maximum number of nodes.

1.4.14 Queue allocation

Most bus designers are familiar with arbitration protocols, which are similar in function to SCI's bandwidth allocation protocols. When bus transactions are unified (not split into separate request and response subactions) and never busied, fair arbitration protocols are sufficient to ensure that all transactions eventually complete. However, when bus transactions are split into request and response subactions, many requesters may access a shared responder node, and its available queues may be filled. When queues are filled, request subactions are terminated with a busy status, which forces them to be retried until the queue eventually has space.

In the absence of queue-reservation protocols, some retried *request subaction*s could never be sent successfully. Although queues may be emptied quickly, they could consistently be refilled by one or several other requesters, while the one *requester* is continually busied, as illustrated in figure 25.

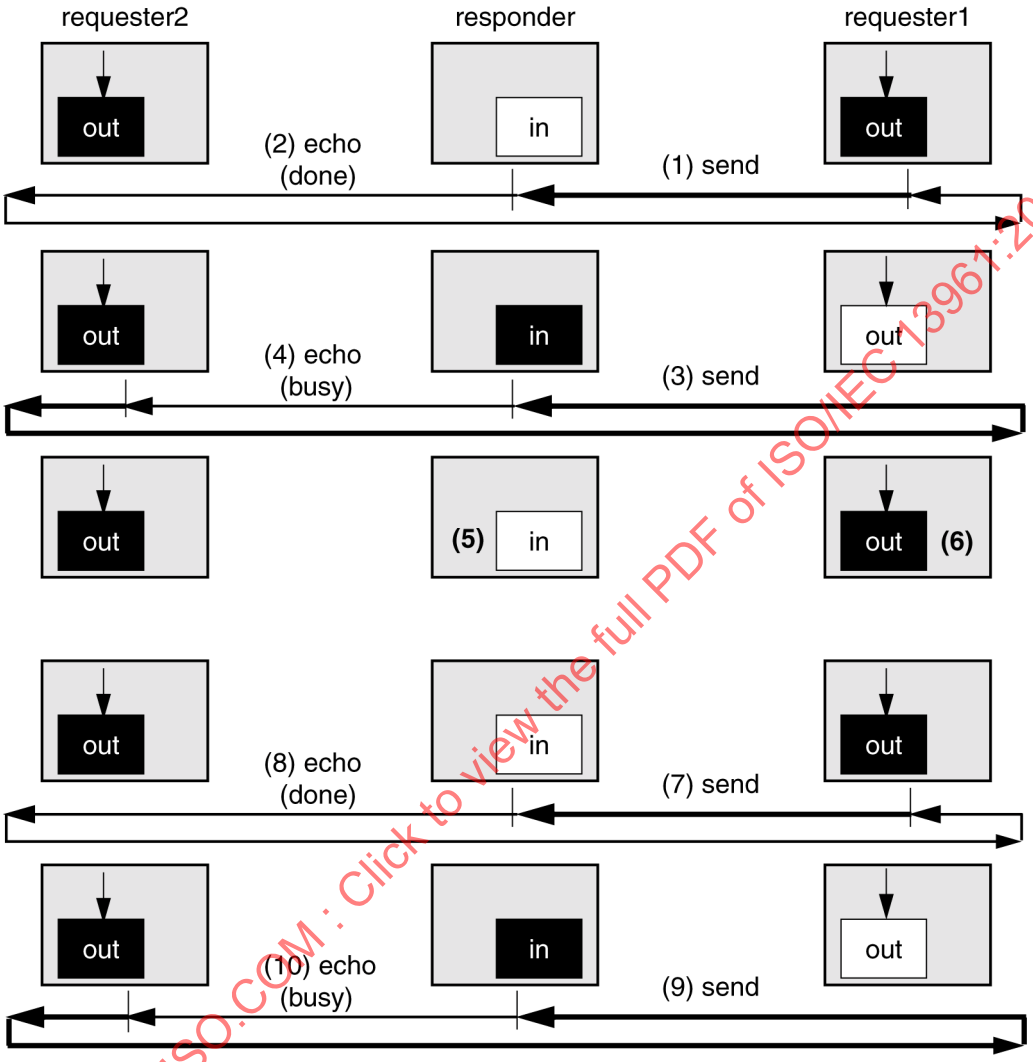


Figure 25 – Queue allocation avoids starvation

In this illustration, requester1 initially sends (1) a request-send packet to the responder; since the responder's queue is empty, the packet is accepted. The returned request-echo packet indicates (2) the request send was accepted without error. However, this request-send packet temporarily fills the responder's input-request queue.

Before the responder has processed its input-request queue, another request-send packet is sent (3) from requester2; since the responder's queue is full, the packet is rejected. The returned request-echo packet indicates (4) the subaction was busied and should be quickly retried.

Soon thereafter, the responder's input-request queue is emptied (5) and another request-send packet is generated (6) within requester1. The new request subaction is sent (7) from requester1; since the responder's queue is empty, the packet is accepted. The returned request-echo packet indicates (8) the request send was accepted without error.

Then requester2 resends (9) its previously busied request-send packet, but since the responder's queue is once again full the packet is rejected. The returned request-echo packet indicates (10) the subaction was busied and should be quickly retried.

If this cycle repeats, the less-fortunate requester2 could be forever starved by the activity of requester1. The SCI allocation protocols avoid such starvation conditions by reserving space for the older send packets that are busied. See 3.7 for details.

1.5 Cache coherence

1.5.1 Interconnect constraints

High-performance processors use local caches to reduce effective memory-access times. In a multiprocessor environment this leads to potential conflicts; several processors could be simultaneously observing and modifying local copies of shared data.

Cache-coherence protocols define mechanisms that guarantee consistent data even when data are locally cached and modified by multiple processors. The SCI cache-coherence protocol can be hardware based, thus reducing both the operating system complexity and the software effort to ensure consistency. Many cache-coherence protocols rely on the broadcasting of all transactions. This broadcasting allows use of eavesdropping and intervention techniques to achieve data consistency. Broadcast transactions are inherent in a bus-based system, but are not feasible for large high-speed distributed systems. Therefore, broadcast and eavesdropping mechanisms are not used by the SCI cache-coherence mechanism.

1.5.2 Distributed directories

SCI uses a distributed directory-based cache-coherence protocol. Each shared line of memory is associated with a distributed list of processors sharing that line. All nodes with cached copies participate in the update of this list.

Every memory line that supports coherent caching has an associated directory entry that includes a pointer to the processor at the head of the list. Each processor cache-line tag includes pointers to the next and previous nodes in the sharing list for that cache line. Thus, all nodes with cached copies of the same memory line are linked together by these pointers. The resulting doubly linked list structure is shown in figure 26.

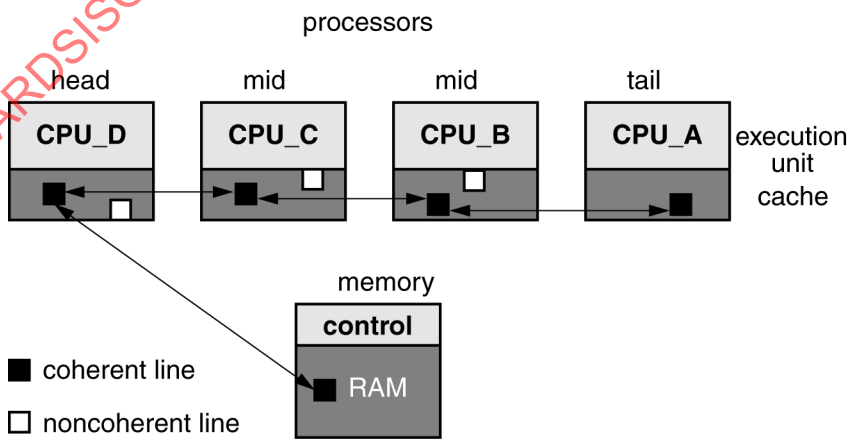


Figure 26 – Distributed cache tags

Note that this illustrates the logical organization of the directory's sharing-list structure for one line, which may be different for each line that is cached. The processors are always shown on the top and the shared memory location is shown on the bottom. These logical illustrations should not be confused with the physical topology of a system; SCI expects that processors and memory will often be found on the same node.

Coherence protocols can be selectively enabled, based on bits in the processor's virtual-address-translation tables. Depending on processor architecture and application requirements, pages could be coherently cached, noncoherently cached, or not cached at all.

This distributed-list concept scales well. Even when the number of nodes in a list grows dramatically, the memory-directory and processor-cache-tag sizes remain unchanged. However, memory-directory storage and processor-cache-tag storage represent extra fixed-percentage overheads for cache-coherence protocols.

The list pointer values are the node addresses of the processors (caches). When a node accesses memory to get a copy of coherently shared data, memory saves the requesting node's address. If there are currently no cached copies, the requesting node becomes the head of a new list. (The memory directory is updated with the new node address.) If other nodes have cached copies of the data, the pointer to the head of the sharing list is returned from memory. The requesting node inserts itself at the head of the list and gets its data from the previous head.

With the exception of the pairwise sharing option, write access is restricted to the node at the head of the list. To get write access, a requesting node creates an exclusive copy by inserting itself at the head of the list and purging the remainder of the list entries. SCI supports both weak and strong sequential consistency, as determined by the processor architecture. A weakly ordered write instruction can be executed before the sharing-list purge completes, while a strongly ordered write must wait for purge completion.

1.5.3 Standard optimizations

Standard optimizations are defined that improve the performance of common kinds of coherence updates, as follows:

1) *Fresh copies.* The *fresh* memory state indicates that all shared copies are read-only; the data can be returned from memory when a new processor is attaching to the head of the previous sharing list.
2) *DMA transfers.* DMA data can be read directly from the sharing-list head without changing the directory state. DMA writes (of full 64-byte lines) can be performed directly to memory, although a list of old copies (purge list) will be returned to the writer if the data were being shared.
3) *Pairwise sharing.* When data are shared by a producer (the writer) and a consumer (the reader), data are directly transferred from one cache to the other. The directory pointers need not be changed, and memory is not involved in the cache-to-cache transfer.

1.5.4 Future extensions

As well as supporting a wide range of interoperable options, the SCI standard intends to support several compatible future extensions. This allows implementations to quickly use the existing specification, while providing opportunities to expand the SCI capabilities when more experience is available. Although the future extensions are beyond the scope of the SCI standard, a short overview is intended to provide the reader with insights on how this standard may evolve in the future.

1.5.4.1 Out-of-band QOLB

The SCI standard supports the concept of delaying distribution of shared data, by queuing additional requesters until a cache line has been released by its current owner (queued on lock bit, called QOLB). The coherence protocols define the QOLB option to avoid transferring shared cache lines until the data can be used. Although QOLB controls the flow of cache lines between caches, an additional lock bit is needed to validate ownership of the cache-line data; within the SCI standard, this lock bit is expected to be contained within the 64 bytes of cache-line data.

A future extension to the SCI coherence protocols could implement a more-transparent lock bit, by providing an out-of-band lock bit for every 64-byte cache line. The advantage of using out-of-band lock bits is that compiler support of QOLB is made much easier. As an example, consider an array of objects, each of which needs a lock bit. The QOLB protocols assumed that the lock-bit and its affected data are contained within the same cache line. Although the compiler can make each object slightly larger, this would change the size of each array object.

If lock bits are implemented as a one-bit cache-line tag, which is located in an out-of-band data address, then the size of array elements is unaffected by the lock bits. To implement these lock bits, each cache line would be assumed to have a 513th bit associated with it. A reserved bit in the header could be used to efficiently transfer this bit in response-send packets; a bit in the extended header could be used to transfer this bit in request-send packets. Processors would be expected to provide special loadQolb and swapQolb instructions to read and modify this out-of-band lock bit based on the cache-line address being accessed. Special operating system software would be expected to save and restore these extra bits when the data is swapped to secondary storage.

The encoding of this out-of-band lock bit has been deferred, so that it can be reconsidered when the coding requirements of the logarithmic extensions (discussed in the following subclause) are known.

1.5.4.2 Logarithmic extensions

On a large heavily loaded system, hot spots may occur at or near a heavily shared memory controller. To eliminate such hot spots, coherence protocols should support the possibility of combining list-prepend requests in the interconnect. Such hot spots not only degrade the performance of the requesting processor, they degrade the performance of other transactions that share portions of the congested connection path. Although coherent combining is not defined in this specification, it is planned as part of P1596.2, a compatible extension to the SCI standard.

A possible way to support coherent combining is as follows. While queued in a switch buffer, two requests to the same physical memory address (read A and read B) can be combined. The combining generates one response (status A), that is immediately returned to one of the requesters, and one modified request (read A-B), that is routed toward memory. Additional requests (read C) can also be combined with the modified request, as illustrated in figure 27.

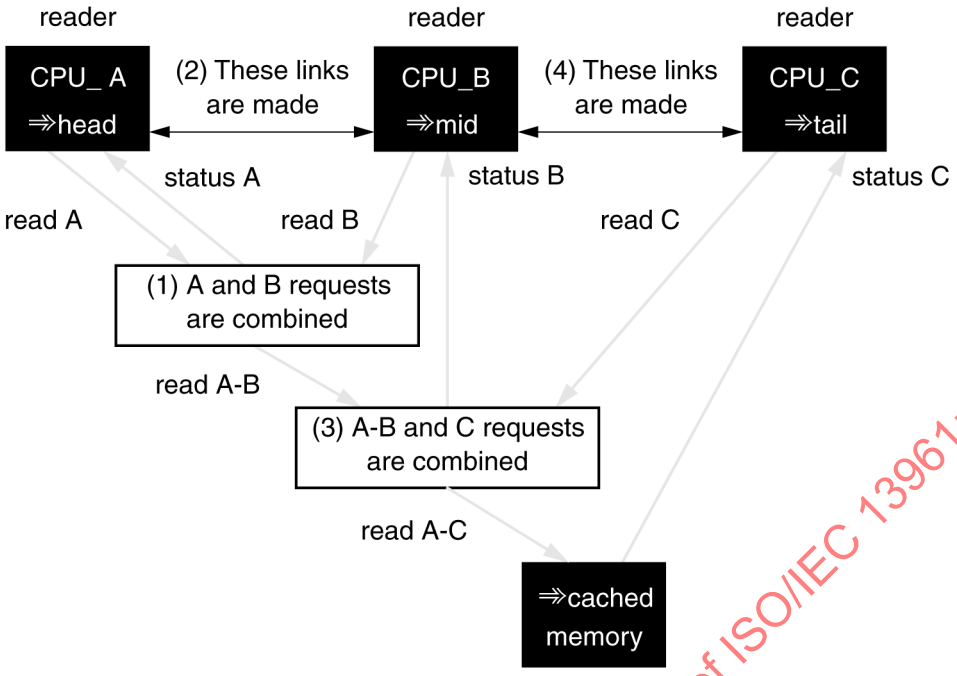


Figure 27 – Request combining

These read transactions can be combined in the interconnect or at the front-end of the memory controller.

When request combining reduces the hot spot latencies, the distribution of data to the other sharing-list entries may become the performance bottleneck. Extensions to the coherence protocols are being developed to reduce the linear latencies normally associated with data distribution and invalidations. Linear latencies can be reduced to logarithmic latencies by adding a third sharing-list pointer to SCI's forward and backward pointers to form a tree structure, as illustrated in figure 28.

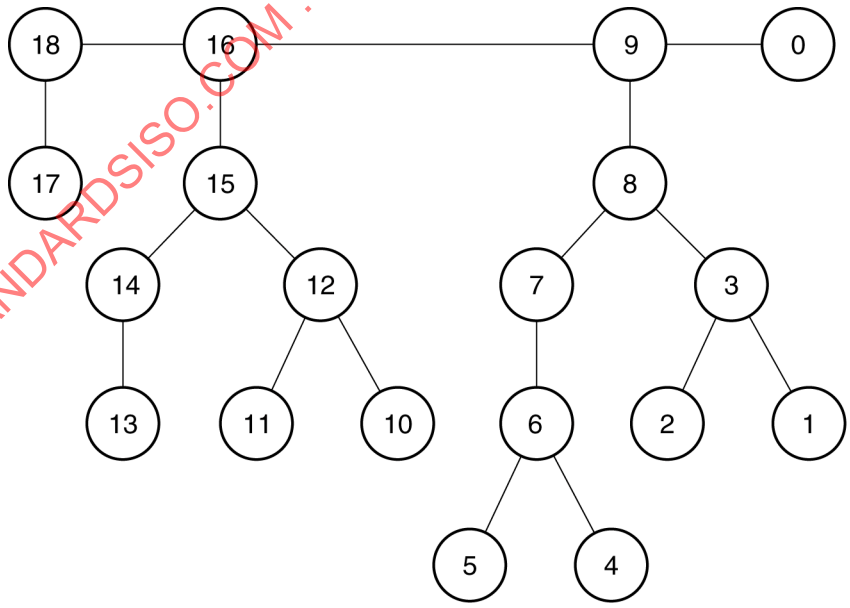


Figure 28 – Binary tree

The three pointers per cache line define a binary tree. Shared data can be routed through the tree to quickly distribute new copies of read-shared data. A writer can also route purges through the tree to quickly invalidate other read-only copies. Deadlock avoidance for forwarding of data and purges can be handled correctly.

The support for binary trees is planned as a compatible extension to SCI (P1596.2). It is an authorized standards project that has not been completed at the time of this International Standard's publication. For current information contact the chairman of that working group.

1.5.5 TLB purges

Most SCI systems will have processors that use virtual addressing. Such processors cache their most recent virtual-to-physical address translations in special translation lookaside buffers (TLBs). When page-table entries are changed, remotely cached TLB entries need to be purged.

TLB replacements are usually handled by software that purges the corresponding remote entries when page-table entries are changed. Three remote TLB purge mechanisms are supported by SCI:

1) *Indirect purging.* The TLB purge address is left (1) in a memory-resident message. Remote processors are interrupted (2), read their messages, purge their local TLB entries, and return their completion status to memory (3).
2) *Direct purging.* The TLB purge address is written to a *control* register on each remote processor. The response from the *control* register write is delayed until the TLB purge has completed.
3) *Coupled purging.* Physically addressed TLB entries can be implemented as cached versions of page-table entries. When the page table is modified the cache-coherence protocols are used to invalidate the TLB entries in the other processors.

The first two of these TLB-purge options are illustrated in figure 29, for processor P-1 purging a TLB entry in processor P-2. The third option has some dependency interlocks that must be clearly understood to ensure correctness while avoiding deadlock.

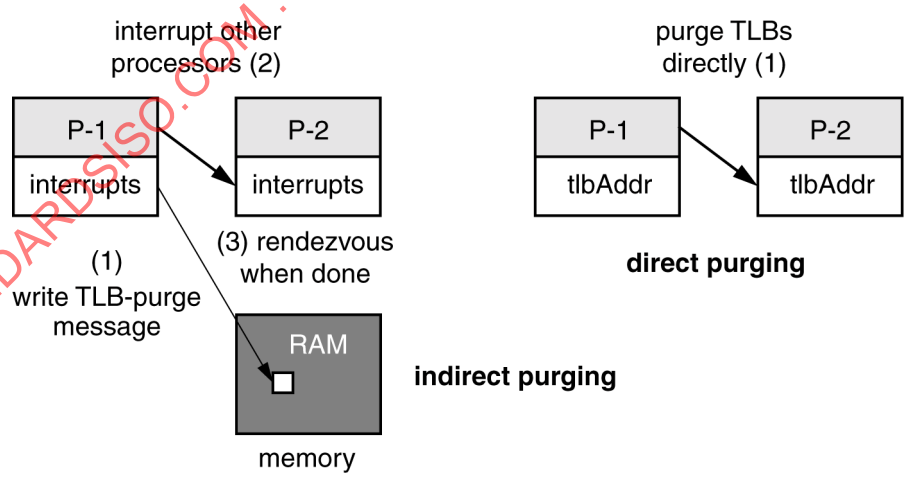


Figure 29 – TLB purging

1.6 Reliability, availability, and support (RAS)

1.6.1 RAS overview

Maintainability has been a primary concern in the design of SCI. To simplify maintenance, the SCI protocols have been defined with the following precepts in mind:

1) *Conceptual simplicity*. Although high-performance circuits may be complex when implemented, the functions provided by the SCI interconnect should be conceptually simple.
2) *Minimum options*. It is better to standardize on one nonoptimal option than to support a wide variety of options in the field.

Rather than describing a formal RAS strategy, this clause describes the major decisions (in the logical protocols) that were influenced by the RAS objectives and strategies.

1.6.2 Autoconfiguration

Each ringlet has a scrubber node that is responsible for monitoring ringlet activity and discarding stale or corrupted packets and idle symbols. To minimize human errors in the configuration process, the scrubber is automatically selected when the ringlet is initialized. This avoids the use of human-settable switches, which could accidentally be set to conflicting values.

The scrubber-selection process is based on an 80-bit unique identifier. The 16 most-significant bits of this identifier can be set manually, so that a pre-specified scrubber can be selected whenever the ringlet is initialized. The least-significant 64 bits of the number are used to break ties, when two or more nodes have the same value for the 16 most-significant bits. These 64 bits are assigned at node manufacturing time, or may be generated randomly (based on a real, not pseudo-, random number generator).

The initial addresses on each ringlet are automatically assigned by the scrubber, based on the distance of the node from the scrubber. In larger systems with multiple ringlets, each of the scrubbers initially assigns the same sequence of nodeId values to the nodes on its ringlet. Initialization software eventually overrides these initial values and assigns unique *nodeId* values to all nodes on all ringlets in the system.

1.6.3 Control and status registers

In the design of the control and status registers (as defined by the CSR Architecture), the following issues were considered:

1) *Autoconfiguration*. When new nodes are inserted, the old boot code should still work on the new system. The new configuration can be automatically detected and dynamically initialized. Autoconfiguration support includes the following features:
   a) *Standard ID-ROM*. Each node has ROM. A standard portion of the ROM identifies the node's name and initialization characteristics.
   b) *Standard selftests*. With standardized selftests, a node can be partially initialized before its I/O driver software is available.
2) *Distributed error logs.* The CSR Architecture provides the framework for implementing distributed error logs, one on each node in the system. These error logs supplement the standardized error status codes when attempting to isolate the source of an error.

See the CSR Architecture for details. Note that most of the definitions therein are shared by related buses (Futurebus+ and Serial Bus) as well.

1.6.4 Transmission-error detection and isolation

In a large system, a significant number of errors may occur during packet transmissions. SCI protocols are designed to detect these errors readily and isolate them. Although a small portion of each packet has no error detection coverage, these fields are only used for arbitration purposes; an error in them would affect only the packet's ringlet-local effective priority, not the packet's correct interpretation.

To reliably detect transmission errors, all packets are protected by a 16-bit ITU-T CRC code. The packet's flow-control information (which dynamically changes during packet routing) is excluded from the CRC calculation. Thus, the CRC is unchanged by intermediate (switch) hops between the original source and the final target. This simplifies implementation of switches and improves reliability of error checking (coverage is not compromised while a new CRC is being appended to unprotected data).

Timeouts are also used to detect transmission errors. Whenever possible, these timeouts are designed to be self-calibrating (so they cannot be incorrectly set). An exception is the response timeout, which has to be set by software (based on knowledge of system configuration and design parameters). Allocation protocols ensure a minimal amount of fairly apportioned bandwidth, so proper timeout values that detect hardware transmission errors are independent of the system's real-time software loading.

Addressing errors are a form of transmission error; although the data are not corrupted during transmission, there is no target to properly acknowledge the packet. These addressing errors are quickly detected and reported by ringlet scrubbers, so that these (software-related) errors will not be confused with other (hardware-related) types of transmission errors.

When possible, error status is returned to the *requester* in the response-send packet, using a 4-bit status code. The status code distinguishes among error categories. This helps isolate the cause of the problem (for an address-ID error), or the location of additional information (for a responder-data error).

1.6.5 Error containment

To simplify recovery from transmission errors, errors are contained (whenever possible). For example, the conversion of a send packet into an echo packet is delayed so that the integrity of the send packet can be reflected in its echo.

Often transmission of a packet or echo has begun before it is discovered to be invalid. This is commonly done to reduce latency. In such a case the correct CRC is computed for the data as transmitted, and then certain bits are complemented to produce a recognizable bad CRC value. This process is called stomping the CRC, and makes it possible to discriminate between packets newly discovered to be bad and those that have already been detected but are still propagating. Thus error logging can record the bad packet at only the first checking location after the failure, making discovery of the failure point easier. The stomped CRC is a bad CRC, and has the normal effect that the packet will eventually be discarded.

Error containment also influenced the *time-of-death* fields (which are optionally included in all send packets). When a response timeout is generated, the *time-of-death* value can be used to guarantee that residual send packets have been deleted. This simplifies error recovery, since stale packets (which could be confused with newly generated transactions) are never delivered.

1.6.6 Hardware fault retry (ringlet-local, physical layer option)

Ringlet-local hardware fault-retry may be supported (as a physical layer option) on individual ringlets. However, hardware fault-retry is not supported for end-to-end transmissions, since the failures introduced by the (much more complex) end-to-end retry hardware would most likely offset most of the benefits it could provide. For example, hardware fault retry could be used to improve the reliability of transmission over a less reliable intermediate medium, as illustrated in figure 30.

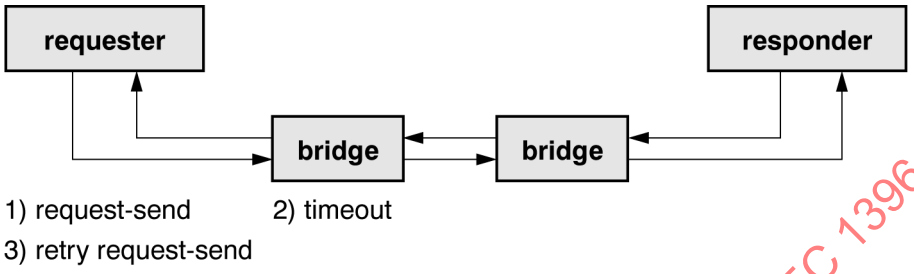


Figure 30 – Hardware fault-retry sequence

Hardware fault retry has significant costs; special accounting hardware is needed to log sequence numbers needed for duplicate suppression, and each packet is lengthened by prepending these sequence numbers. The SCI standard does not define a hardware fault-retry mechanism.

1.6.7 Software fault recovery (end-to-end)

Several forms of software fault recovery are well supported. When accessing noncoherent CSRs, many transactions can be safely retried by software. The retry is not as simple as it first sounds; after the failure, the success of the first transaction is unknown (it may have succeeded or failed). Reads (to SCI-defined registers) have no side effects, so reads of these registers can be safely retried (one and two reads are equivalent, they both have no side effects).

Many writes have side effects, but can safely be retried (the side effects of one and two identical writes are the same). Retrying writes to CSRs where one and two writes have different side-effects is harder. For these registers, the CSR Architecture recommends using sequence-number bits in the data; these bits can be used by software (to verify the success or failure of the initial transaction attempt). Designers should carefully consider these problems and avoid creating needless difficulties for error recovery.

Software can perform end-to-end fault retry on *coherent memory transactions*. Since coherent memory has a tag identifying the last owner, the previously dirty entries can be identified after the fault is detected. Transaction fault recovery involves flushing the old dirty copy to memory and destroying the (possibly now corrupted) sharing-list structure, as illustrated in figure 31. After the data have been flushed, the sharing list is rebuilt automatically using the standard coherence protocols.

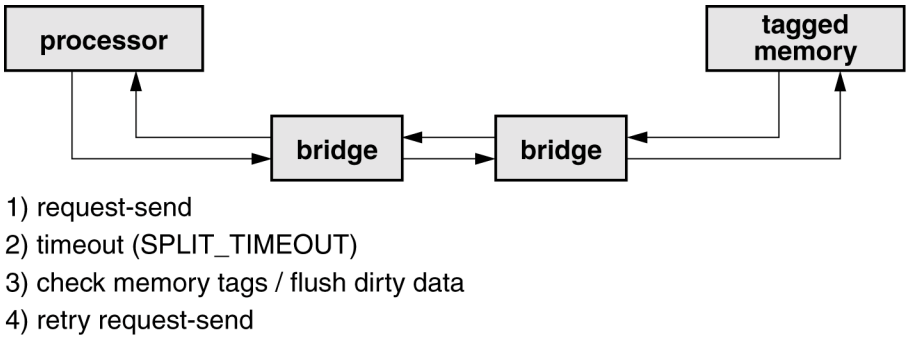


Figure 31 – Software fault-retry on coherent data

Although the error recovery is relatively inefficient, its infrequent use should have a minimal impact on system performance.

1.6.8 System debugging

A *trace* bit is provided to selectively enable *packet logging* as packets are routed through the system. Since a globally synchronized time-of-day clock is provided (see 3.4.6), packets can be accurately time-stamped as they are logged. The use of time stamps allows the route of the packet (at logging locations) to be reconstructed based on the log contents. The detailed implementation and use of the *trace* bit is beyond the scope of the SCI standard.

1.6.9 Alternate routing

On a single ringlet, the SCI protocols are intolerant to faults since one failure brings down the entire ringlet. However, redundant-ringlet systems are feasible. Switches or bridges between ringlets can isolate each ringlet from the failure of others.

Even though a ringlet has failed, its nodes could still be interrogated and diagnosed using a redundant low-cost diagnostic bus (Serial Bus). Although Serial Bus is not intended to be a redundant operational bus, it can assist in identifying the failed field-replaceable unit.

1.6.10 Online replacement

The SCI standard supports online replacement of modules, in that the full system need not be idled while a module is being replaced. Software is expected to isolate the module before it is replaced, taking account of any resources that that module was providing to the rest of the system. For example, coherently cached data has to be flushed to memory before a processor caching it can be replaced.

The physical specification section of the SCI standard defines mechanical and electrical interfaces that support online replacement. These specifications allow a module to be replaced without disrupting the electrical power supplied to other nodes in the system. The CSR Architecture defines the behaviour of modules during the on-line replacement process.

Replacing a module temporarily breaks the ringlet. A switch could isolate this ringlet from the remainder of the system while the module is being replaced. Alternatively, fault-recovery software could retry transactions that were lost while the module was being replaced. These ringlet-isolation and fault-recovery protocols are beyond the scope of the SCI standard.

2 References, glossary, and notation

2.1 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of IEC and ISO maintain registers of currently valid International Standards.

EIA IS-64 (1991), *2 mm Two-Part Connectors for Use with Printed Boards and Backplanes* [3)]

IEC 60793-1, *Optical fibres – Part 1: Generic specification* [4)]

IEC 60793-2, *Optical fibres – Part 2: Product specifications*

IEEE Std 1301-1991, *IEEE Standard for a Metric Equipment Practice for Microcomputers – Coordination Document) (ANSI)* [5)]

IEEE Std 1301.1-1991, *IEEE Standard for a Metric Equipment Practice for Microcomputers – Convection-Cooled with 2 mm Connectors (ANSI)*

ISO/IEC 13213:1994 [ANSI/IEEE Std 1212, 1994 Edition], *Information technology – Microprocessor systems – Control and Status Registers (CSA) Architecture for microcomputer buses* [6)]

ISO/IEC 9899:1990, *Programming languages – C*

2.2 Conformance levels

Several keywords are used to differentiate between different levels of requirements and options, as follows:

expected
a keyword used to describe the behaviour of the hardware or software in the design models assumed by the SCI standard. Other hardware and software design models may also be implemented

may
a keyword that indicates flexibility of choice with no implied preference

---

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

4) IEC publications are available from IEC Customer Service Centre, Case postale 131, 3 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse. IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

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

6) ISO publications are available from the ISO Central Secretariat, Case postale 56, 1 rue de Varembé, CH-1211 Genève 20, Switzerland/Suisse. ISO publications are also available in the United States from the American National Standards Institute.

shall
a keyword indicating a mandatory requirement. Designers are required to implement all such mandatory requirements to ensure interoperability with other SCI standard conformant products

should
a keyword indicating flexibility of choice with a strongly preferred alternative. Equivalent to the phrase is *recommended*

NOTE These terms are used infrequently in the introductory portions of this specification, which are nonbinding. This includes clause 1: Introduction, parts of clause 3: Logical protocols and formats, parts of clause 4: Cache-coherence protocols, parts of clause 5: C-code structure, and the Annexes. In these clauses, a less-precise writing style is used.

2.3 Terms and definitions

Many bus and interconnect-related technical terms are used in this document. These terms are defined as follows:

agent
switch or switch-like component or bridge between the requester and the responder. During normal operation the agent's intervention is transparent to the requester and responder

allocation protocols
protocols used to allocate resources that are shared by multiple nodes. These include bandwidth allocation protocols and queue allocation protocols

backplane
board that holds the connectors into which SCI modules can be plugged

In ring-based SCI systems, the backplane may contain wiring that connects the output link of one module to the input link of the next. In switch-based SCI systems, the backplane may merely provide mechanical mounting for connectors that are connected by cables to the switch circuitry; or, part of the switch circuitry may be implemented on the backplane. Usually the backplane provides power connections, power status information and physical position information to the module.

bandwidth allocation protocols
protocols used to allocate bandwidth on a ringlet. This involves inhibiting send-packet transmissions from one or more nodes when another node is being starved (never gets an opportunity to transmit its send packet)

board
physical component of an SCI module that is inserted into one of the backplane slots. A board may contain multiple nodes, and that nodes can be implemented without using boards or modules

bridge
pair of communicating nodes, each of which selectively (based on target address) accepts certain packets for retransmission by the other

For example, a symmetric bridge may be used to connect two SCI ringlets. Such a bridge (the simplest kind of switch) acts as an agent, taking the place of the target on one ringlet and of the source on the other. It acts like a node that has many addresses. Bridges may also connect dissimilar systems, such as SCI and VME. Such bridges are generally much more complex, because they must translate protocols.

broadcast transaction
transaction that may be processed by more than one *responder*.

Although a broadcast transaction is distributed to all nodes on the ringlet, it is only accepted by nodes that support the broadcast option. Broadcast transactions are flow-controlled, and bridges or switches may forward these transactions to other ringlets in the system. Only *move* transactions can be broadcast, so higher-level protocols are needed to confirm when all broadcast transactions have completed in a multiple-ringlet system.

busied
status indication returned in an echo packet that indicates to the sender that the send packet was not accepted (and was discarded), probably because there was no room in the destination queue. The sender should retransmit the packet later

byte
eight bits of data, used as a synonym for octet

cache line
often called simply a line

unit of data on which coherence checks are performed, and for which coherence tag information is maintained

In SCI, a line consists of 64 data bytes.

cleanse instruction
cleanse (cache-control) instruction converts a line to the clean state (the data in cache and memory are the same)

clear packet
packet used during initialization to empty linc buffers and initialize the linc. CSR state is unaffected; for example, the node's address is unchanged by a clear

Clear may be sent by any node that has lost synchronization in order to trigger reinitialization.

clockStrobe signal
packet that causes a node to record its time-of-day registers (if any) when it is received, and to record the duration of the propagation of the packet within the node. Used for precisely synchronizing multiple time-of-day clocks within a system

consumer
node on a ringlet that strips a send packet from the ringlet and creates the echo packet that is returned to the producer

consumption of idles
idle symbols arriving at a node may be discarded (after saving certain information) while other symbols that arrived earlier and were stored in the bypass FIFO are being transmitted

Consuming idles thus reduces the number of symbols stored in the bypass FIFO.

CRC
cyclic redundancy code used for error detection on each packet

CSR Architecture
See ISO/IEC 13213.

directed transaction
transaction that is processed by one and only one *responder*

The read, write, and lock transactions are always directed transactions.

doublet
two bytes of data

echo
second subaction packet

This 8-byte packet reports the status of the queueing of the corresponding send packet.

emperor
processor that has the responsibility for initialization of an entire multiprocessor system

flag
signal used to delimit packets in parallel signal transmission implementations

For example, in the 16-bit parallel implementation the flag is a 17th signal. In some serial implementations special symbols could be used in place of flag transitions.

flush instruction
flush (cache-control) instruction changes a line to the uncached state

If the data are dirty, they are copied back to memory before the old cache line is invalidated.

Futurebus+
refers to ISO/IEC 10857:1994 [B11] and IEEE Std 896.2-1991 [B3] which refine the earlier IEEE Std 896.1-1987. Those standards are intended for use with (or as an upgrade path from) MULTIBUS II (ISO/IEC 10861:1994 [B12]) systems, VME (IEC 60821:1991 [B1]) systems, and U.S. Navy next-generation hardware systems. They support cache-coherent multiprocessing with physical buses on the backplane. SCI may be used to interconnect Futurebus+ systems, since they share the same coherence line size and CSR Architecture.

global system time
SCI nodes may maintain time-of-day clocks as described in the CSR Architecture. Software may adjust each of these clocks in order to make them consistent to high accuracy. If this is done, the system is said to implement global system time. Otherwise each clock runs independently, which is sufficient for local timeout purposes but is not sufficient to implement the optional packet time of death feature.

go symbol
idle symbol that has been marked with the pertinent go bit (*idle.lg*=1 or *idle.hg*=1) to give permission to a waiting node to transmit

high symbol
idle symbol that has been marked for consumption by highest-priority nodes (Sometimes called high-idle symbol.)

idle symbol
symbol that is not inside a packet, and is therefore not protected by a CRC. Idle symbols serve to keep links running and synchronized when no other data are being transmitted. The idle symbol also contains flow-control information

linc
interface circuitry that attaches itself to an SCI ringlet

A linc typically contains control/status registers (including identification ROM and *reset* command registers) that are initially defined in a 4 kbyte (minimum) ringlet-visible initial node address space.

line
block of memory (sometimes called a sector) that is managed as a unit for coherence purposes; i.e., cache tags are maintained on a per-line basis. SCI directly supports only one line size, 64 bytes

low symbol
idle symbol that has been marked for consumption by lower-priority nodes. Sometimes called low-idle symbol. May also be consumed by a highest-priority node when it is taking its fair share of lower-priority bandwidth

module
board, or board set, consisting of one or more nodes, that share a physical interface to SCI

If a module has multiple boards with backplane-mating connectors, it only uses one for the logical connection to the node. The others may provide additional power or I/O for their associated boards, but otherwise merely pass the input link signals through to the output link to provide continuity in case the module is plugged into a ring-connected backplane.

monarch
processor that has the responsibility for initializing a part of the system, such as a ringlet

If a system has multiple monarchs, they eventually defer to an emperor that coordinates the initialization process.

node
entity associated with one or more interconnected links and optionally containing other functional units, such as cache and memory

In normal operation each node can be accessed independently (a control-register update on one node has no effect on the control registers of another node).

nodeId
16-bit number that determines the node address space

After system initialization, unique *nodeId* values have been assigned to all nodes within a tightly coupled system. The nodeId is the part of the 64-bit address that is used for routing packets.

noncoherent
transactions that do not participate in the cache coherence protocols, normally used when the data are known to be uncached or uncacheable, or in systems that do not implement cache coherence

NuBus®*
refers to IEEE Std 1196-1987, IEEE Standard for a Simple 32-Bit Backplane Bus: Nubus [B9]

NVM (nonvolatile memory)
memory that retains its contents even through power failures

octlet
eight bytes of data

packet
collection of symbols that contains addressing information and is protected by a CRC. A subaction consists of two packets, a send packet and an echo packet

---

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

packet symbol
symbol contained within a packet and protected by the packet's CRC

(Exception: part of the second symbol in a packet contains flow control information that is not covered by the CRC, but the symbol as a whole is still considered to be within the packet.)

physical interface
circuitry that interfaces a module's nodes to the input link, output link and miscellaneous signals

producer
node on a ringlet that transmits a send packet to the consumer and deletes the echo packet that is returned

purge instruction
purge (cache-control) instruction changes a line to the uncached state, invalidating the old cache line without copying dirty data back to memory

QOLB (queue on lock bit)
mechanism for efficiently sequencing the access to resources that are not to be used by more than one process at a time

quadlet
four bytes of data

queue allocation protocols
protocols used to allocate queue space when several nodes are sending packets to a shared node

This involves rejecting packets (with a busy status), but reserving future queue space; the reserved queue space is eventually used during one of the packet's retransmissions.

requester
node that initiates a transaction, by initiating a request subaction

request echo
echo packet generated by a responder or agent when it strips the request send packet

request send
packet generated by a requester to initiate an action in the responder, containing address, command, and, if appropriate, data

In a processor-to-memory read transaction, for example, the request send transfers the memory address and command from the processor to memory.

request subaction
request send and its echo. Often called simply a request

reset packet
packet used during initialization to reset the node's CSR state, empty ring buffers, initialize the ring interface and establish that ring closure has been achieved

responder
node that completes a transaction, by returning a response subaction

response echo
echo packet generated by a requester or agent when it strips the response send packet

response-expected request
request subaction component of a response-expected transaction

response-expected transaction
transaction that normally consists of a request subaction and a response subaction

For example, the read, write, and lock transactions are all response-expected transactions.

responseless request
request subaction component of a responseless transaction

responseless transaction
transaction that consists of only a request subaction (there is never any response subaction)

For example, the move and event transactions are responseless transactions.

response send
packet generated by a responder to complete a transaction initiated by a requester

In a processor's memory-read transaction, for example, the response send returns the requested data and related status information from the memory to the processor.

response subaction
response send and its echo. Often called simply a response

ringlet
closed path formed by the connection that provides feedback from the output link of a node to its input link

This connection may include other nodes or switch elements.

ROM (read-only memory)
memory on a node that provides storage locations for normally read-only data or code

The ROM data are maintained across losses of primary and secondary power. In some implementations ROM may be writable, using (normally disabled) vendor-specific protocols.

SCI
scalable coherent interface

SCI standard
refers to this document

scrubber
node that marks packets as they go past in a ringlet, and discards any previously marked packet

This prevents damaged or misaddressed packets from circulating indefinitely. The scrubber also performs other housekeeping tasks for the ringlet. There is always exactly one scrubber on a ringlet. Normal nodes may all have scrubber capability built in, but exactly one is enabled as scrubber per ringlet. Often the scrubber will take responsibility for initializing a ringlet, but this could be done by another (unique) node.

segment
portion of a ringlet between the producer and consumer along which a packet is sent

The segment traversed by a send packet is the send segment, and the segment traversed by an echo is the echo segment.

send
first of two packets within a request or response subaction (the second packet is an echo)

The send packet contains a 16-byte header (containing command and status) and may optionally contain a data component (up to 256 bytes).

Serial Bus
refers to the P1394 standards project that is defining an inexpensive serial network that can be used as an alternate control or diagnostic path, as an I/O connection, or even in place of a parallel bus in some systems

signal line
electrical or optical information-carrying facility, such as a differential pair of wires or an optical fiber, with associated transmitter and receiver, carrying binary true/false logic values

slot
module-insertion position provided by the backplane and associated card cage

source
node that creates a send or echo packet

The source nodeId is contained in the third symbol of the packet.

specialId
reserved nodeId value associated with special-send packets

special send
packet having one of a particular set of special addresses and a special format used for initialization, such as reset or clear

split transaction
transaction that consists of separate request and response subactions

On a backplane bus, for example, a split transaction is one in which bus mastership is relinquished between the request and response subactions. Few buses permit split transactions.

See also: unified transaction.

strip
to replace a received nonidle symbol by an idle symbol and hence to remove it from transmission on a ringlet

For example, a send packet is stripped by the receiving port of an agent or the target and replaced by idles (most of which may be consumed in the process of emptying the bypass FIFO) and an echo. Similarly an echo is stripped by the receiving port of an agent or the source and replaced by idles.

subaction
component of a transaction; a request or a response

switch
device that connects ringlets and has queues

It can behave as a consumer (when accepting remote subactions) and as a producer (when forwarding the subaction to another ringlet). It may be visible as a node, with a *nodeId*, or be transparent, with no *nodeId*. A switch differs from a bridge in that a switch may connect more than two ringlets, but a bridge connects only two. A switch is generally assumed to connect multiple instances of the same bus standard, while a bridge may connect different bus standards.

symbol
16-bit unit of data accompanied by flag information

The flag information may be explicitly present as a 17th bit, or implied by the context. Symbols are transmitted one after another to form SCI packets or idles. The particular physical layer used to transmit these symbols is not visible to the logical layer.

sync packet
special packet that is used heavily during initialization and occasionally during normal operation for the purpose of checking and adjusting receiver circuit timing

target
node addressed by the first symbol of a packet; i.e., the final destination of the packet

time of death
term used to describe a field within a send packet that is used to determine when a send packet is stale and should be discarded

training
process of synchronizing the receiver circuit of a linc to the incoming data stream during initialization

transaction
information exchange between two nodes

A transaction consists of a request subaction and a response subaction. The request subaction transfers commands (and possibly data) between a requester and a responder. The response subaction returns status (and possibly data) from the responder to the requester.

unified transaction
transaction in which the request and response subactions are completed in an indivisible sequence; i.e., no other subactions may be performed on the bus until this response subaction is complete

Most buses use unified transactions, but SCI uses only split transactions. The concept of a unified transaction is only relevant to SCI in the context of bridges to other buses.

2.4 Bit and byte ordering

The addressable unit in SCI is the byte. SCI is primarily defined in terms of packets, constructed from 2-byte (doublet) symbols, that contain a single data value or multiple items located in separate fields. For all packet headers, SCI defines the order and position of fields within the doublets. For data symbols in packets SCI defines the ordering of byte addresses within the symbols.

Bit zero is always the most significant bit of a symbol, byte zero is always the most significant byte of a symbol, and the most-significant doublet of the address always comes first. Bit[0] and byte[0] are sent first when packets are sent bit-serially. This notation is illustrated in figure 32.

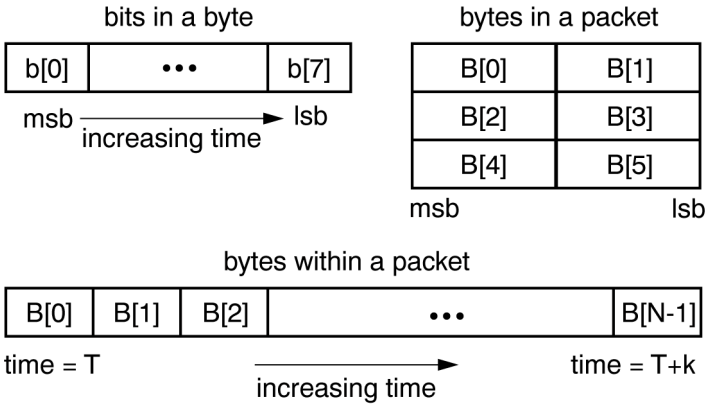


Figure 32 – Big-endian packet notation

The byte ordering defines the position of data bytes within a packet. This is the equivalent of the byte-position ordering on a multiplexed address/data bus. The format of a packet may be specified in terms of sequential symbols, when the next symbol is placed beneath the previous symbol. The format of a packet may also be specified in terms of a sequential byte stream, where the next byte is placed to the right of the previous byte. Both packet format conventions are illustrated in figure 32.

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

To support the address-invariance model, this standard specifies the mapping of data-byte addresses to bytes within a packet. For an access of an SCI-defined quadlet register, the data byte with the smallest address is the most significant. This big-endian byte significance option (which is also used by the CSR Architecture) is illustrated in figure 33.

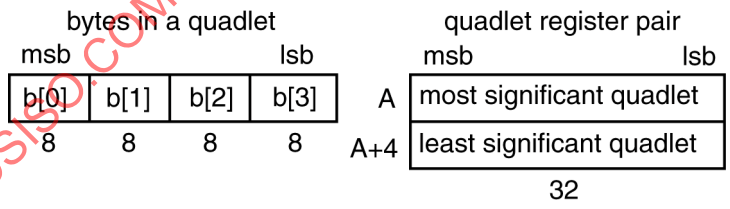


Figure 33 – Big-endian register notation

Since 64-bit addressing is used throughout this standard, some registers are implemented as quadlet-register pairs. For consistency, the quadlet register with the smaller address is also the most significant, as illustrated above.

For the defined packets and registers, 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 labelling 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.

NOTE 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 this standard. For example, the byte significance of a processor's general registers, or an I/O adapter's control and status registers, need not be the same as the byte-position ordering within a packet or within a defined SCI register.

2.5 Numerical values

Decimal, hexadecimal, and binary numbers are used within this document. For clarity, 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, except in C-code contexts, where they are written as 0x123EF2, etc. 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.

2.6 C code

The C code in this document is compatible with that specified by ANSI X3.159-1989, except for the use of long names. To use this code with a compiler that does not support long names, use a preprocessor to translate the long names into unique short names.

3 Logical protocols and formats

3.1 Packet formats

3.1.1 Packet types

The types of SCI packets directly involved in the logical protocols are listed in table 1. The first (*targetId*) symbol of a packet is used to route the packet to its destination. The second (*command*) symbol uniquely identifies one of the four packet types. Some switches may also use the *command* and the third (*sourceId*) symbol to make routing decisions.

Table 1 – Packet types

Group	Description
request send	request subaction content
request echo	request subaction local acknowledge
response send	response subaction content
response echo	response subaction local acknowledge

The 16 highest *targetId* values identify packets that have special properties, while all other *targetId* values are used for normal send or echo packets. The special packets include *init* and *sync* packets (see 3.2.7). Bits within the *command* symbol are used to distinguish the four types of send and echo packets: the *command.ech* bit distinguishes a send from an echo, the *command.cmd* field distinguishes a request send from a response send, and the *command.res* bit distinguishes a request echo from a response echo, as illustrated in figure 34.

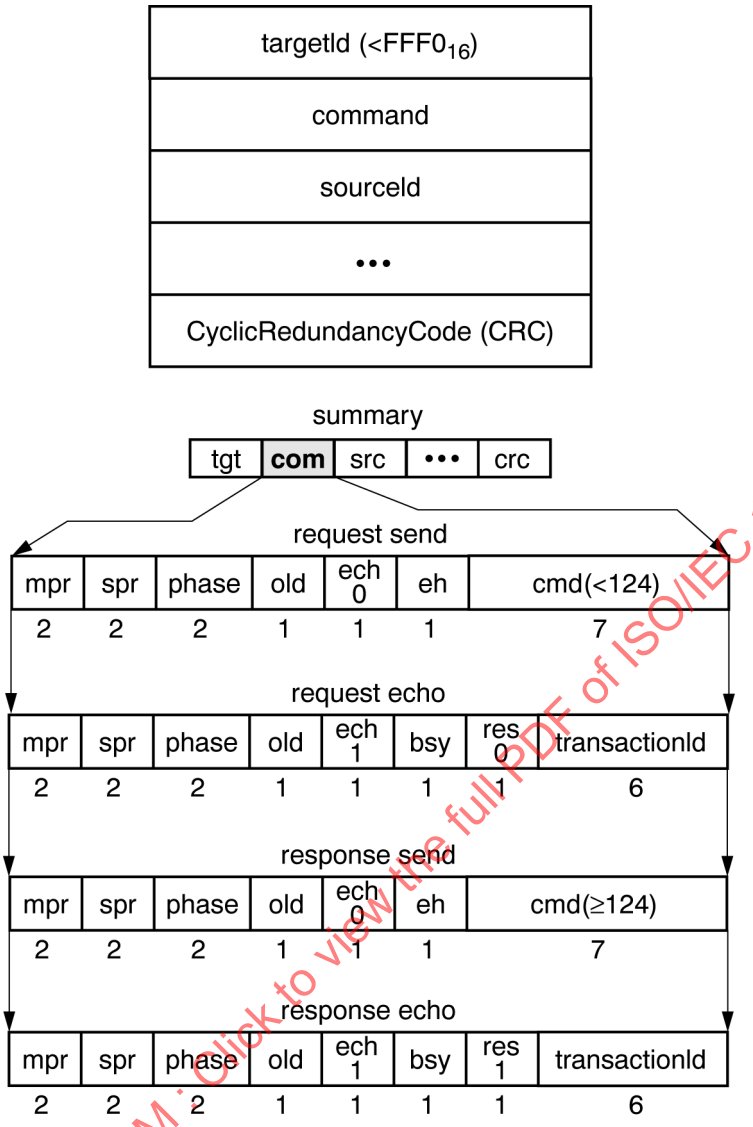


Figure 34 – Send- and echo-packet formats

3.2 Send and echo packet formats

3.2.1 Request-send packet format

The request packet contains *targetId*, *command*, *sourceId*, *control*, a 48-bit *addressOffset*, possibly a 16-byte extended header *ext*, *data* (0, 16, 64, or 256 bytes), and a CRC. These components are illustrated in figure 35.

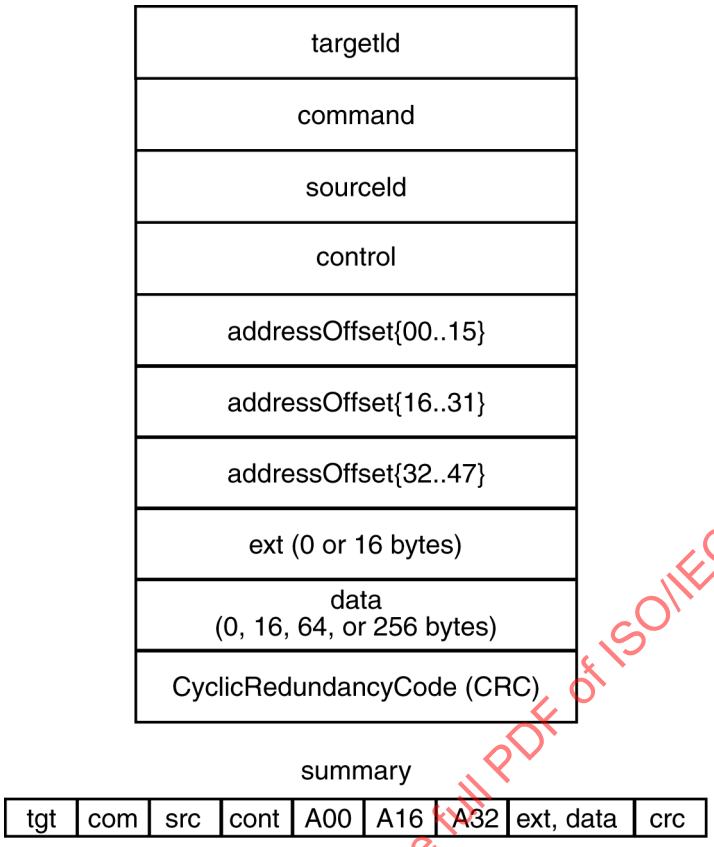


Figure 35 – Request-packet format

The *targetId* symbol is used to route the send packet from the requester to the responder. Some switches may also use the *cmd* field (in the following command symbol) and the *sourceId* symbol to make their routing decisions.

The *command* symbol provides flow control information and identifies the request-send packet type, as illustrated in figure 36.

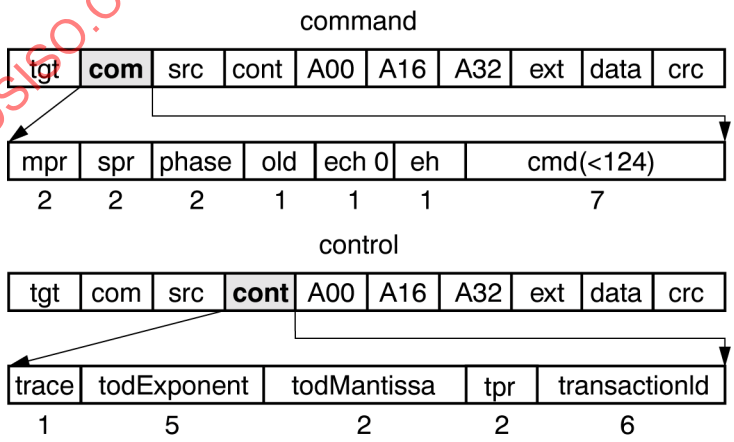


Figure 36 – Request-packet symbols

The *sourceId* symbol provides a *nodeId* address for the request echo that is created when the send request is consumed. The *sourceId* symbol also provides the *targetId* address for the subsequent response-send packet that may be created by the responder.

The 48-bit *addressOffset* field is interpreted by the responder. Some of these addresses have special meanings, as defined by the CSR Architecture.

A portion of the extended header *ext* (whose presence is signalled by the *command.eh* bit) is used by the cache-coherence protocols to pass a pointer while writing a cache line directly from one cache to another, as defined in the cache-coherence part of the SCI standard (see clause 4). Other uses are reserved for definition in future extensions to SCI. Likely contents include information needed for realtime scheduling applications.

For compatibility between designs, even nodes that do not generate request packets with extended headers shall accept and process request packets with extended headers if the request fits in their buffer. With the exception of the CRC calculations (which include the extended-header symbols), these packets shall be processed as though the extended header did not exist. Thus nodes that use 80-byte buffers to handle a header and 64-byte write data accept all transactions with extended headers except for write64. Nodes that handle 256-byte transactions shall include sufficient buffer storage to accept extended headers in all cases.

The *command.mpr* (maximum ringlet priority) field, which is initially zero when a send packet is produced, is modified by other nodes to determine the ringlet priority. The *command.spr* field (send priority) associates one of four effective send-priority levels with each request packet. The effective priority is set by the producer when the send packet is sent, based on the transaction priority (*control.tpr*) and the priority of other blocked subactions awaiting transmission in the producer's queue.

The *command.phase* field is used by the queue-control hardware, to enforce forward progress when packets are busied. When a request is busied, the phase field for the following retry is provided by the phase field of the echo packet, using the coding defined in table 2.

Table 2 – Phase field for send packets

value	name	description
00	NOTRY	send, on space-available basis
01	DOTRY	send, reserve space if busied
10	RETRY_A	retry, after BUSY_A status returned
11	RETRY_B	retry, after BUSY_B status returned

The *command.old* bit is used by the scrubber to identify and eventually discard stale send packets. The scrubber sets the *old* bit to 1 in all packets. When the scrubber observes an incoming *command.old* bit already set in a send packet, it strips the packet from the ringlet and creates an echo packet with a special error status.

Note that the *command.mpr*, *command.spr*, *command.phase*, and *command.old* fields are excluded from the request-send packet's CRC calculation (they are assumed to be zero when the CRC is calculated).

The *command.ech* bit identifies echo packets, and is 0 for send packets.

The *command.eh* bit is set to 1 if a 16-byte extended header is present. The coherence protocols define a few of the bytes within the extended header; the remainder are reserved. See 4.2 for details.

The *command.cmd* field specifies the transaction command being performed, as defined in 3.4.1. For request-send packets, the value of cmd is less than 124.

The *control.trace* bit (trace packet route) enables an optional hardware logging mechanism to monitor the progress of packets through the interconnect. When the *control.trace* bit is set, a node that produces or consumes this send packet places the packet header into a history log along with the current time. If the system clocks have been properly synchronized, this time is sufficiently accurate to ensure that the sequence of nodes processing the packet can be correctly determined later.

The *control.todExponent* and *control.todMantissa* fields specifies the global system time at which the send packet is to be discarded by agents. The zero value of *control.todExponent* corresponds to a never-die code, which prevents time-of-death discards. For example, the never-die code is used before the synchronized global time reference has been established, as described in 3.8.2.

The *control.tpr* field specifies a 2-bit transaction priority that assigns one of four priority levels to each request-send packet. This priority is set by the requester when the packet is created, and is used by the allocation protocols.

The *control.transactionId* field, when concatenated with the *sourceId* symbol, uniquely identifies each of the outstanding transactions. (Some nodes may be capable of issuing multiple requests before any responses are received.)

3.2.2 Request-echo packet format

A request-echo packet is created by a consumer when a request-send packet is stripped from the ringlet. The *targetId* and *sourceId* fields in the echo packet are generated by exchanging those in the stripped send packet. Request-echo packets are always four symbols long as illustrated in figure 37.

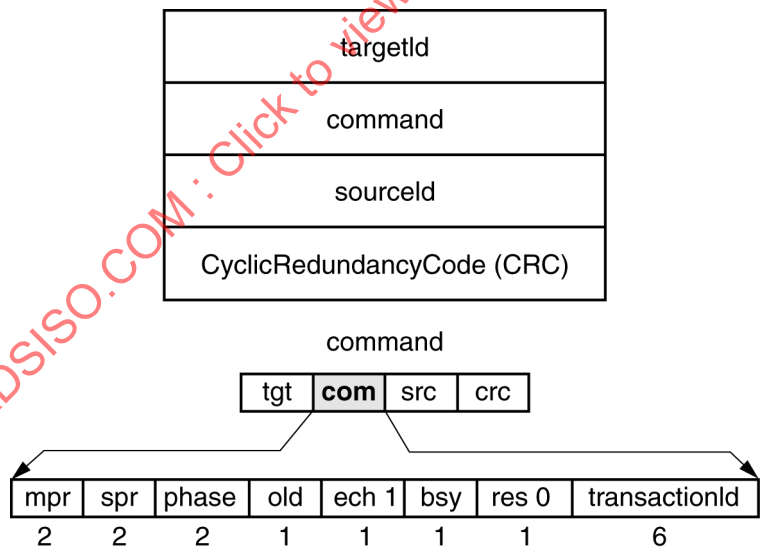


Figure 37 – Request-echo packet format

An echo *command* symbol contains part of the *command* symbol and part of the *control* symbol from the stripped send packet. The *command.mpr* and *command.spr* (maximum and send priority) fields are used by the bandwidth allocation mechanism. The *command.mpr* field is cleared to zero when the echo is created, and is updated when the echo passes through other nodes. The *command.spr* field is the *command.mpr* value contained in the send packet that was stripped from the ringlet when the echo was created.

The *command.phase* field is used by the consumer to return enqueue status to the producer; its meaning depends on the value of the busy-status bit (*command.bsy*). For completed send subactions (*command.bsy* is 0), the phase field values are defined in table 3. The scrubber uses the NONE status when stripping old send packets from the ringlet (within that generated echo packet, *command.bsy* is 0 and *command.phase* is NONE).

Table 3 – Phase field for nonbusied echoes

value	name	description
00	DONE	send queued successfully (safe to discard)
01	NONE	no local nodeId address (none responded)
10	DONE_A	reserved (equivalent to DONE)
11	DONE_B	reserved (equivalent to DONE)

For busied subactions (*command.bsy* is 1) the phase field values are defined in table 4 (see also table 2).

Table 4 – Phase field for busied echoes

value	name	description
00	BUSY_N	reserved (retry using NOTRY)
01	BUSY_D	retry, using DOTRY
10	BUSY_A	space reserved, retry using RETRY_A
11	BUSY_B	space reserved, retry using RETRY_B

The *command.old* bit is used by the scrubber to identify and eventually discard stale echo packets. A scrubber sets the *old* bit to 1 in all echo packets. When a scrubber observes an incoming *command.old* bit it discards the old echo packet from the ringlet.

Note that the *command.mpr*, *command.spr*, *command.phase*, and *command.old* fields are excluded from the request-echo packet's CRC calculation.

The *command.ech* bit is set to 1 to identify echo packets. The *command.bsy* bit is set to 1 when the send packet was rejected and needs to be resent. The *command.res* bit is used to discriminate between request- and response-echo packets and is 0 for request-echo packets.

The *command.transactionId* field is the value contained in the *control.transactionId* symbol of the send packet that was stripped from the ringlet when the echo was created.

3.2.3 Response-send packet

The response packet contains the *targetId*, *command*, *sourceId*, *control*, *status*, *forwId*, *backId*, possibly an extended header *ext* (0 or 16 bytes), *data* (0, 16, 64, or 256 bytes), and CRC. These components are illustrated in figure 38.

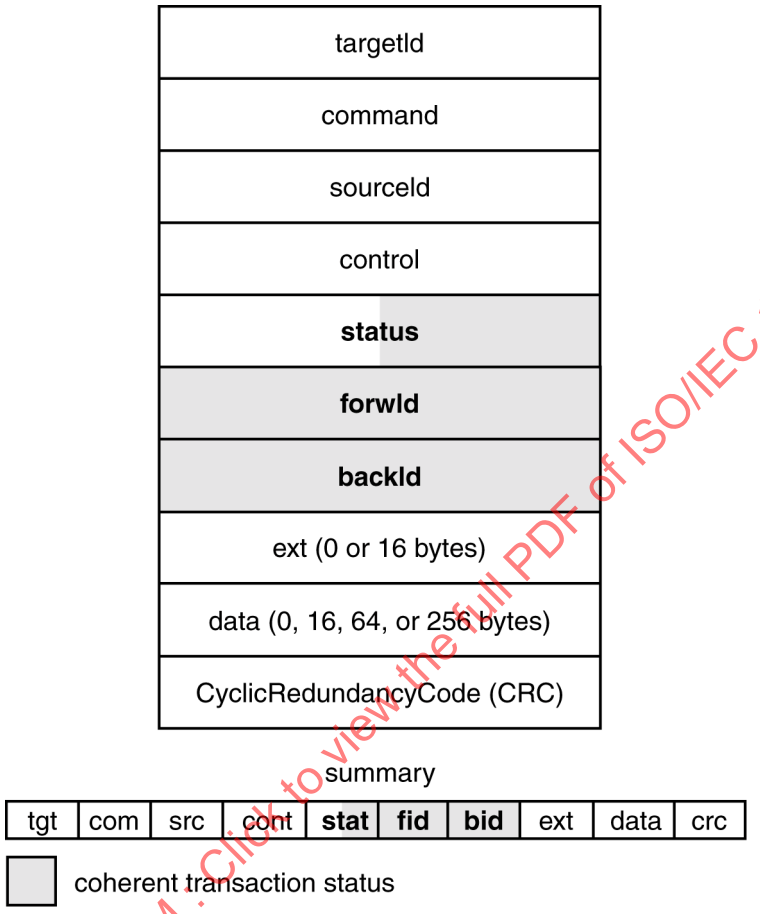


Figure 38 – Response-packet format

The *targetId* symbol is used to route the response-send packet from the responder to the requester. Switches may also use the *command.cmd* field (in the following command symbol) and the *sourceId* symbol to make their routing decisions.

The *command* symbol provides flow-control information and identifies the response-send packet type, as illustrated in figure 39.

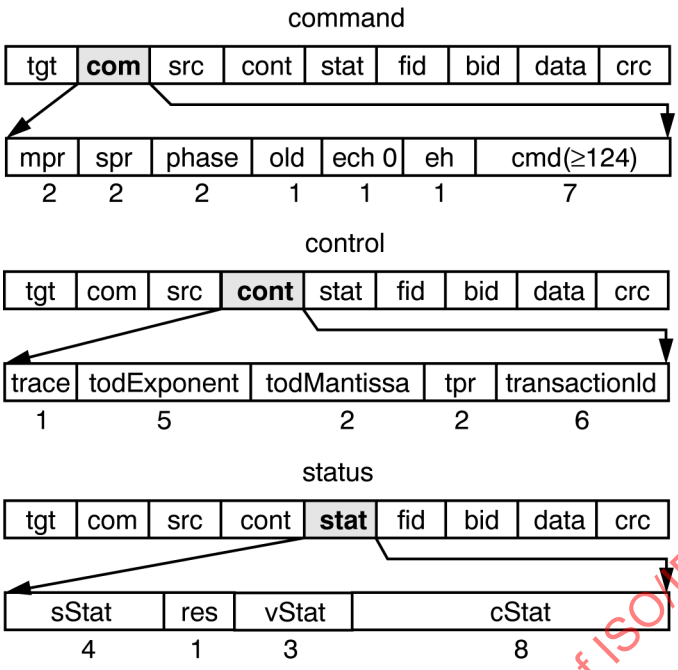


Figure 39 – Response-packet symbols

The *sourceId* symbol provides a *targetId* address for the response echo that is generated when the response send is stripped from the ringlet. The *sourceId* symbol also identifies the responder node that created the response-send packet, which may (after certain error conditions) be different from the addressed responder (as defined by the *targetId* value of the request-send packet). The *sourceId* field may be used to implement vendor-dependent protection or error-logging protocols, which are beyond the scope of the SCI standard.

Portions of the *status*, *forwId*, and *backId* symbols are used by the cache-coherence protocols to identify other nodes caching the same cache-line address. The *forwId* and *backId* fields are defined in clause 4.

The definitions of the *command.mpr*, *command.spr*, *command.phase*, *command.old*, *command.ech*, and *command.eh* fields are the same for request-send and response-send packets. Note that the *command.mpr*, *command.spr*, *command.phase*, and *command.old* fields are excluded from the response-send packet's CRC calculation.

The *command.cmd* field has the same format in the request-send and response-send packets, but a different range of command codes is used (the two least-significant bits of the command code specify the response-transaction size).

The definitions of the *control.trace*, *control.todExponent*, and *control.todMantissa* fields are the same for request-send and response-send packets.

The value of the *control.tpr* field specifies the priority of the *response subaction*. A responder should use the same *control.tpr* value in the response-send packet that it received in the corresponding request-send packet, but may use a larger value in the response-send packet.

The *control.transactionId* field, when concatenated with the *targetId* field, uniquely identifies each of the outstanding transactions.

The *status.sStat* (standard-status summary) field (defined in table 5) returns a summary of the transaction status from the addressed responder (or affected agent). Note that this field allows the requester to tell when an agent has provided the (unsuccessful) status information, implying that the request never reached its target. The following subclause defines the *status.sStat* code values.

The *status.res* (reserved) bit is reserved for future extensions to the SCI standard. The *status.vStat* (vendor-dependent status) field returns vendor-dependent transaction status details from the addressed responder (or affected agent). The definition of this field is beyond the scope of the SCI standard.

The *status.cStat* (coherence status) field returns the coherence-check status from the addressed responder. This field is defined in the cache-coherence specification (see clause 4).

3.2.4 Standard status codes

The *status.sStat* (standard status) field (defined in table 5) returns a summary of the transaction status from the addressed responder (or affected agent). Note that this field allows the requester to tell when an agent has provided the (unsuccessful) status information, implying that the request never reached its target. The following subclause defines the *status.sStat* code values.

Table 5 – *status.sStat* status summary codes

Responder-provided *status.sStat* codes		
code	***status.sStat* name**	**description**
0000	RESP_NORMAL	completion successful, normal operation
0001	RESP_ADVICE	completion successful, abnormal operation
0010	RESP_GONE	transaction OK, coherent data gone
0011	RESP_LOCKED	transaction OK, coherence-line is locked
0100	RESP_CONFLICT	conflict (end-to-end retry)
0101	RESP_DATA	unrecoverable failure
0110	RESP_TYPE	unsupported command or length
0111	RESP_ADDRESS	addressing error

Agent-provided *status.sStat* codes		
code	***status.sStat* name**	**description**
1000	AGENT_NORMAL	completion successful, normal operation
1001	AGENT_ADVICE	completion successful, abnormal operation
1010	AGENT_GONE	reserved for extensions to the SCI standard
1011	AGENT_LOCKED	reserved for extensions to the SCI standard
1100	AGENT_CONFLICT	reserved for extensions to the SCI standard
1101	AGENT_DATA	split-response timeout
1110	AGENT_TYPE	unsupported command or length
1111	AGENT_ADDRESS	addressing error

Requesters are expected to perform their normal transaction-completion processing for transactions with a RESP_NORMAL or RESP_ADVICE completion status. Requesters are expected to process transactions with a RESP_ADVICE or AGENT_ADVICE completion status similarly, but save the *status* symbol and other implementation-dependent information for later analysis. The other *status.sStat* error-status values are expected to generate a requester trap; the trap is expected to invoke error-isolation and/or error-recovery procedures.

A response-expected transaction (for which a response-send packet is expected) is normally completed when a response-send packet containing the RESP_NORMAL completion status is returned from the responder. A RESP_ADVICE completion status is returned when a response-expected transaction is completed successfully, but a recoverable error is detected (for example, a single-bit memory error).

A valid (correct address and type) noncoherent response-expected memory-access transaction (readsb, writesb, nread00, nwrite16, nwrite64, nwrite256) is terminated with a RESP_GONE status if the requested data is unavailable (memory is in the MS_GONE state). Note that a different completion status code (RESP_NORMAL) is used when data is available in memory, but may be stale (the most-recent copy is coherently cached); noncoherent requesters are expected to detect such coherent-data conflicts by checking the *status.cStat* field in the response-send packet.

A valid coherent memory-access or cache-access transaction (mread00, mwrite16, mwrite64, mwrite256, cread, or cwrite64) is terminated with a RESP_LOCKED status if the requested data is unavailable because the line's fault-recovery lock is set. This status is only expected to be observed while recovering from coherent transaction failures.

A valid (correct address and type) noncoherent response-expected memory-access transaction (readsb, writesb, nread00, nwrite16, nwrite64, nwrite256) is terminated with a RESP_CONFLICT status in situations where busy-retry protocols could generate system deadlocks, if the request cannot be queued. For example, bridges to I/O buses without split-response capabilities are expected to generate a RESP_CONFLICT error status when cross-bus access conflicts are detected.

A valid (correct address and type) response-expected transaction is terminated with a RESP_DATA status if the request cannot be completed correctly (for example, a double-bit memory error).

A correctly addressed response-expected transaction is terminated with a RESP_TYPE status if the request-send packet's *command.sCmd* is not supported, or if address ranges/alignments are incorrect. In case of a conflict, a RESP_TYPE command has precedence over a RESP_DATA command (i.e., a bad access is not detected if the access is not performed). For example, a RESP_TYPE status is generally returned if an nread64 transaction addresses a CSR address; a RESP_TYPE status is returned by a cache when processing the cwrite16 or cwrite256 transactions; a RESP_TYPE status is returned by a simple cache (that does not support pairwise sharing) when processing a cwrite64 transaction.

A response-expected transaction may be correctly routed to the *responder* based on the targetId value in the request-send packet. A RESP_ADDRESS status is returned if the send-packet's address is not recognized by the responder. In the case of a conflict, a RESP_ADDRESS error status has precedence over a RESP_TYPE error status (i.e., the validity of a command is not checked if the address is incorrect). For example, a RESP_ADDRESS status is returned if cache-access transactions (cread00, cwrite16, cwrite64, or cwrite256) are addressed to a responder that has no cache. Also, a RESP_ADDRESS status would be returned for a memory-access transaction (mread, mwrite16, mwrite64, or mwrite256) whose address-offset is larger than the size of populated RAM.

To improve system performance a response-expected transaction (for which a response-send packet is expected) may be completed by an agent rather than the addressed responder. For example, a bridge may combine sequential DMA requests into one larger request (to improve the transfer efficiency), or a switch may combine several coherent memory requests (to minimize interconnect traffic). A response-send packet containing the AGENT_NORMAL completion status is returned by such agents. An AGENT_ADVICE completion status is returned by such agents when a response-expected transaction is completed successfully but a recoverable error is detected (for example, a single-bit memory error).

The AGENT_GONE, AGENT_LOCKED, and AGENT_CONFLICT status values are reserved for definition by future extensions to the SCI standard.

When its response packet is excessively delayed or discarded (due to a transmission error or an excessive packet length), a response-expected transaction shall be terminated by a timeout. After the timeout the requester is expected to report the error condition using an internal AGENT_DATA error status.

A correctly addressed response-expected transaction is terminated by an agent with an AGENT_TYPE status if the request-send packet's *command.sCmd* is not supported on a remote bus, or if address ranges/alignments are found to be incorrect when the transaction is forwarded through a bridge.

A response-expected transaction may also be terminated when no other node responds to the targetId address specified within the request-send packet. These addressing errors are detected by the scrubber, which is indirectly responsible for creating a response-send packet containing the AGENT_ADDRESS error status.

3.2.5 Response-echo packet format

A response-echo packet is created by a consumer when the response-send packet is stripped from the ringlet. The *targetId* and *sourceId* fields in the echo packet are generated by exchanging those from the stripped response-send packet. Response-echo packets are always four symbols long, as illustrated in figure 40.

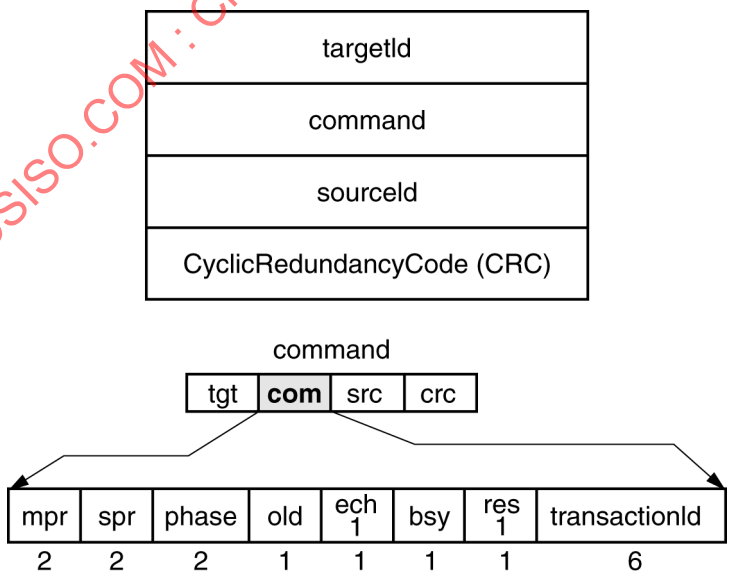


Figure 40 – Response-echo packet format

An echo *command* symbol contains part of the *command* symbol and part of the *control* symbol from the stripped send packet. The *command.mpr*, *command.spr*, *command.phase*, *command.old*, *command.ech*, and *command.bsy* fields are the same for request-echo and response-echo packets. Note that the *command.mpr*, *command.spr*, *command.phase* and *command.old* fields are excluded from the request-send packet's CRC calculation.

The *command.res* bit is 1 for response-echo packets. The *command.transactionId* field is the same for request-echo and response-echo packets.

3.2.6 Interconnect-affected fields

A vendor may modify the interpretation of a send packet's address-offset or data values, based on the needs of the node's requester and responder units. For example, several ranges of address-offset values could be mapped indirectly to the same graphics frame-buffer memory, to support multiple data-access formats (byte per color, bits per color, RGB and YIQ encodings, etc.). However, the other symbols within a send packet have a standard meaning that affects their routing and/or processing by the interconnect as follows:

1) *TargetId symbol*. The *targetId* symbol is used to route a packet from source node to target node.
2) *Command symbol*. A portion of the *command* symbol is used for ringlet-local flow control purposes, and may be changed while the packet is routed through the interconnect. The remainder of this symbol contains the *command.cmd* field, which has the following special properties:
   a) *Response generation*. Only subactions with *command.cmd* values 0 – 55 or 112 – 115 generate a response packet when an addressing error is detected.
   b) *Event processing*. The *command.cmd* values of 120 through 123 identify event subactions. Event subactions may change a node's state, but are discarded if the node does not have space to store them.
   c) *Response queueing*. The *command.cmd* values greater than 124 identify response subactions; requests and responses are placed into different queues while being routed through the interconnect.
   d) *Size restrictions*. The *command.cmd* value specifies the maximum packet size; larger packets may be truncated by intermediate nodes and switches. The actual packet size may be less than this value, but shall be an integer multiple of 16 bytes.
3) *SourceId symbol*. The *sourceId* symbol contains the *nodeId* of the creator of the send packet, which is also the *targetId* address used to return an echo from the consumer to the producer.
4) *Control symbol*. The *control* symbol contains the trace bit, the time-of-death field, the transaction priority field, and the *control.transactionId* field. These influence the subaction's processing by the interconnect as follows:
   a) *Tracing.* The *control.trace* bit enables the logging and time-stamping of packet headers in producers and consumers between the requester and responder.
   b) Time of death. The *control.todExponent* and *control.todMantissa* fields specify when send packets should be discarded.
   c) *Transaction priority*. The *control.tpr* field influences the speed (or at least the sequence) of processing send packets in the requester, intermediate producers and consumers, and the responder.
   d) *Transaction identifier*. The *control.transactionId* field is used to uniquely identify the transaction, so that multiple echoes and responses returning to the same requester can be correctly processed.

5) *Address offsets*. A small range of values within the *addressOffset* field in a request-send packet is reserved for control and status registers, as defined by the CSR Architecture.

6) *CRC calculations*. Only the flow-control information in a packet is excluded from the CRC calculation.

3.2.7 Init packets

There are several special packets, called *init* packets, that are only used during the system initialization process. The special packet format used by *reset*, *clear*, and *start* packets is illustrated in figure 41. These packets are identified by their use of defined special *targetId* values that are all in the range $FFF0_{16} \leq targetId < FFFF_{16}$.

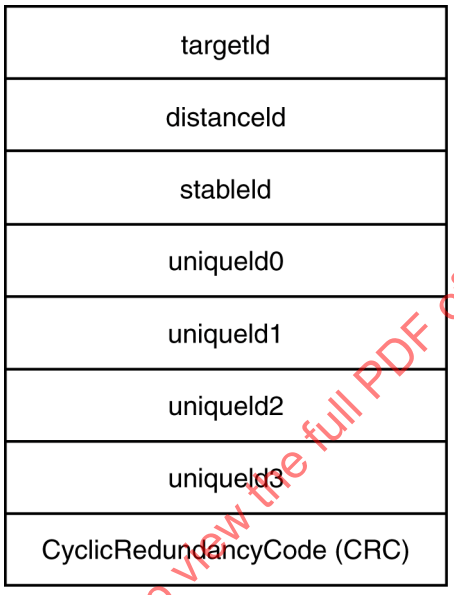


Figure 41 – Initialization-packet format

These special packets contain a *distanceId*, which measures the node's distance from the scrubber; it is decremented by one as the packet passes through each node. In reset packets, the *distanceId* value is used to set each node's initial nodeId value. This field is also used to detect stale *uniqueId* values (perhaps left by a node that started up briefly, then died and restarted again with a lower *uniqueId* number).

The *stableId* field, which sets the default scrubber selection order, is based on inputs from optional backplane-provided *geographicalId* signals or optional nonvolatile memory. The *uniqueId0* through *uniqueId3* fields are the most- through least-significant portions of a 64-bit *uniqueId* value. During ringlet initialization, the *uniqueId* value identifies the packets that each node generates. The *uniqueId* value may be randomly generated at startup (using an uncorrelated thermal noise source) or may be manufactured uniquely (a 24-bit *companyId* followed by a 40-bit *companyUnique* identifier).

For uniquely manufactured *uniqueId* values, the 24-bit *companyId* value is the most-significant portion of the 64-bit *uniqueId* value. The 40 least-significant bits of the *uniqueId* value are *companyUnique* bits that are assigned by the owner of the *companyId* value.

For example, a *companyId* value of $ACDE48_{16}$ (which has a binary representation of $101011001101111001001000_2$) is placed in an initialization packet as illustrated in figure 42. In this figure, the 40 *companyUnique* bits are labelled as # characters.

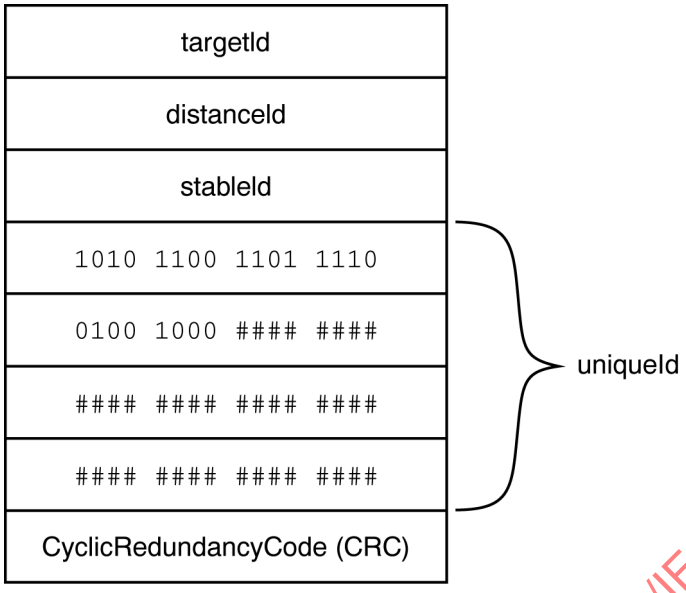


Figure 42 – Initialization-packet format example (*companyId*-based *uniqueId* value)

Note that the *companyId* as referenced in this clause is the same as the company_id as defined by the CSR Architecture.*

3.2.8 Cyclic redundancy code (CRC)

There is a 16-bit check symbol at the end of each packet. For good error coverage, a cyclic redundancy code (CRC) is used. The CRC efficiently detects errors but does not correct errors. Error recovery is performed at a higher level.

There are many 16-bit CRCs that detect all single-burst errors up to 16 bits (1 symbol) and any odd-number-of-bit errors, using various 16-bit polynomials. SCI uses the ITU polynomial $X^{16}+ X^{12}+ X^{5}+ 1$. This is one of the best documented 16-bit polynomials, able to detect the following:

- All odd numbers of bit errors
- All consecutive contiguous (single-burst) errors of 16 bits or less
- All single-, double-, and triple-bit errors

The probability that (given random errors other than the above) there will be combinations of errors that the code is not able to detect is less than 2^{-16} (15 PPM).

SCI presumes that the links are highly reliable and error free, and when error rates increase to the point that this presumption is invalid, the link should be shut down because it has a functional failure. Thus an undetected error should be rare, and probably only will occur during a transition into a shut-down state.

The serial implementation of the ITU-T CRC-16 polynomial is specified as shown in table 6 and figure 43.

---

* The term "company_Id" is used throughout the ROM format to uniquely identify vendors that manufacture or specify components used in the CSR Architecture. The 24-bit "company_Id" value is derived from the 24-bit Organizationally Unique Identifier (OUI) assigned by the IEEE Registration Authority Committee (RAC). To obtain an OUI, contact the Registration Authority Committee, The IEEE, 445 Hoes Lane, Piscataway, NJ 08855-1331, USA (908) 5623-3813.

Table 6 – Serial CRC-16 implementation

```
c15   .=c0 ⊕ d
c14   :=c15
c13   :=c14
c12   :=c13
c11   :=c12
c10   :=c11 ⊕  c0 ⊕ d
c9    :=c10
c8    :=c9
c7    :=c8
c6    :=c7
c5    :=c6
c4    :=c5
c3    :=c4 ⊕ c0 ⊕ d
c2    :=c3
c1    :=c2
c0    :=c1
```

where:

c0–c15	are the contents of the check work
d	is the data (1 bit for each strobe)
:=	Replaced by (after strobe)
⊕	Exclusive OR

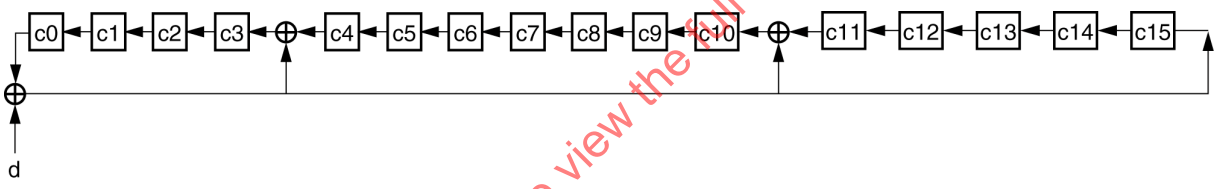


Figure 43 – Serialized implementation of 16-bit CRC

The check word is clocked for every data bit. After all data bits have been used to generate the check word, the check word is inserted in the data stream. The circuit effectively divides the data (bits taken as coefficients of a high order polynomial) by $X^{16} + X^{12} + X^5 + 1$ and uses the remainder as the check word.

3.2.9 Parallel 16-bit CRC calculations

Although the CRC is specified as a bit-serial computation, the CRC value can be computed in parallel as well. This is important for SCI, because CRCs have to be checked and regenerated at full SCI speed. Parallelizing the serial specification generates the set of equations shown in table 7.

Table 7 – Parallel implementation of 16-bit CRC

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

where

C00-C15 are the contents of the new check symbol

e00-e15 are the contents of the intermediate value symbol:
e15 = c15 ⊕ d15; e14 = c14 ⊕ d14; ... e00 = c00 ⊕ d00;

d00-d15 are the contents of the data symbol

c00-c15 are the contents of the old check symbol

All of the check-bits (c00–c15) on the right are before the symbol-clock strobe and all on the left are after the symbol-clock strobe. The assumed hardware model for this calculation is illustrated in figure 44.

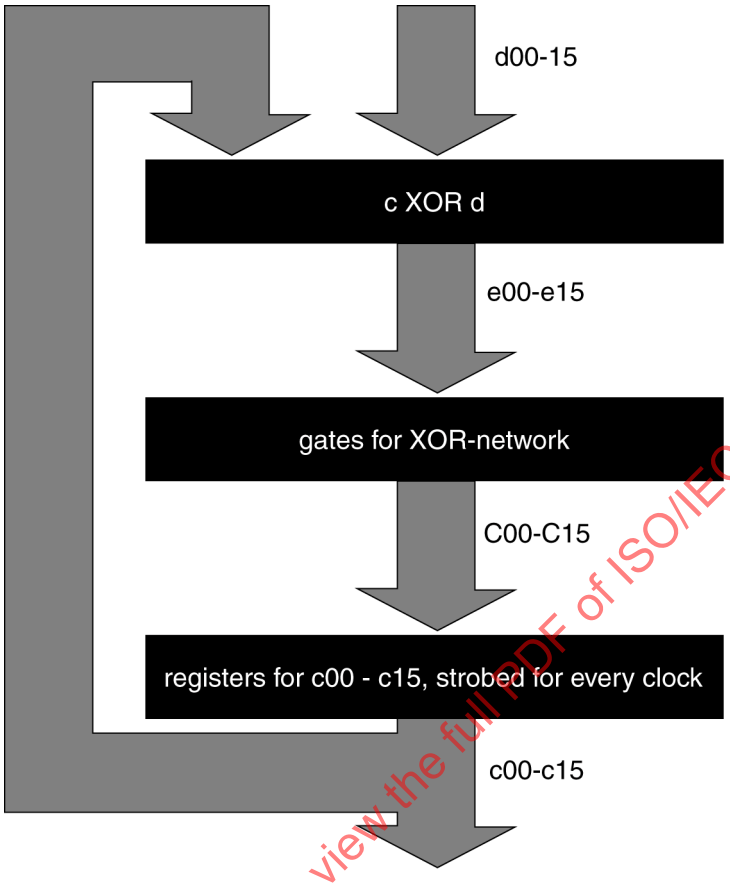


Figure 44 – Parallel CRC check

The maximum number of inputs to an XOR-term for one bit is 16. In advanced ECL technology this will take less than 1 ns (worst case). It is possible to calculate the CRC in real time while transmitting and receiving (independently). This mechanism makes it easy to modify control bits and calculate new check words on the fly when necessary, though generally a CRC is calculated at packet creation and propagates unchanged until the packet reaches its final target in order to ensure end-to-end coverage without gaps.

The 16-bit check word is cleared to begin the packet's CRC calculation. The accumulated CRC value is updated for each symbol in the packet, except for the final one (the CRC). The CRC in a received packet is compared to the computed value. If they are equal the packet is presumed to be error-free.

3.2.10 CRC stomping

The processing of a packet may be initiated before the packet's CRC is verified, but only if the side-effects of the processing can be nullified if a problem with the packet is eventually detected. For example:

1) an echo packet may be created by any requester or responder before the send packet's CRC is observed;
2) the data return from a memory controller may begin before the memory detects an error that affects part of the packet; and
3) the forwarding of a send packet by an agent may begin before the CRC has been observed.

To nullify the side-effects of partially sent send and echo packets, the CRC at the end of the damaged packet is "stomped". Stomping involves setting the new CRC value to the Exclusive Or of *good* and *stomp*, where *good* is what would have been the correct CRC value for the packet (as received) and *stomp* is a defined constant value ($874D_{16}$, which complements half of the bits).

An error is expected to be logged the first time (and only the first time) that a packet is stomped, to assist in isolating the source of the error. For example, if an error is introduced (1) in the request-send packet, several stomped transactions are generated (2, 3, 4), as illustrated in figure 45.

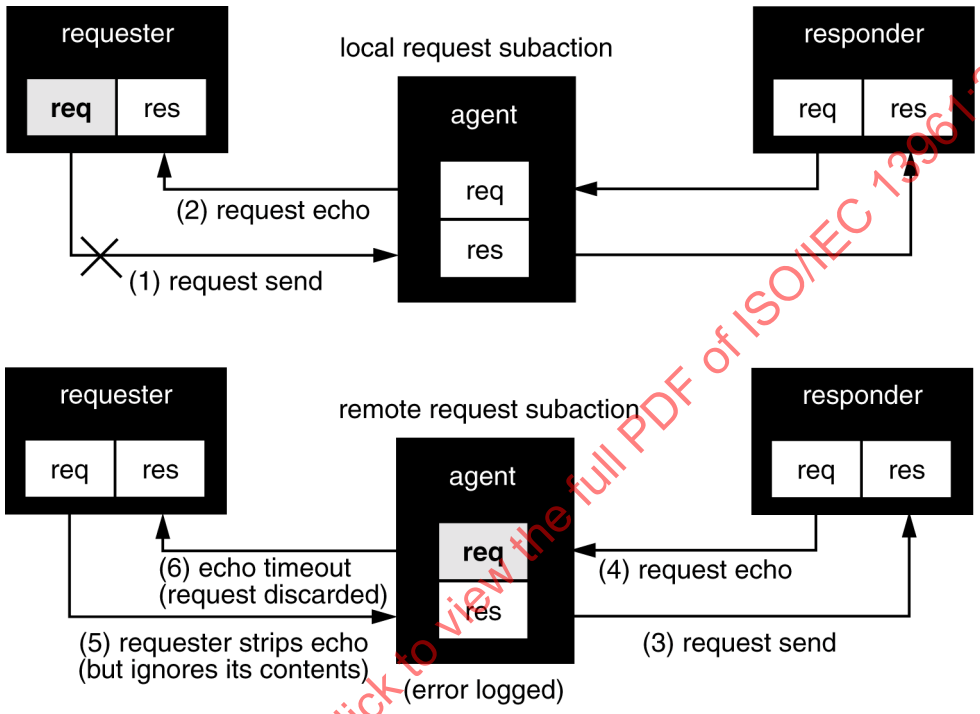


Figure 45 – Remote transaction components (local request-send damaged)

Note that both a stomped request-echo packet and a stomped request-send packet are generated by a high-performance pipelined agent (which begins to forward the send packet before checking its CRC). The remote responder could initiate its processing before the request-send packet's CRC is verified, but must nullify any side-effects when the bad CRC is observed.

If the request-send address has not changed (5), the damaged packet's echo is stripped by the requester, but its side-effects are nullified by the bad CRC value. Therefore, the request remains queued until an echo timeout (6) causes it to be discarded.

3.2.11 Idle symbols

Idle symbols fill the spaces between packets; they are created when request or echo packets are stripped from the ringlet. An idle symbol is any symbol that is not part of a send, sync or echo packet. Only eight bits of information are carried by a 16-bit idle symbol, whose other 8 bits provide a simple parity check, as illustrated in figure 46.

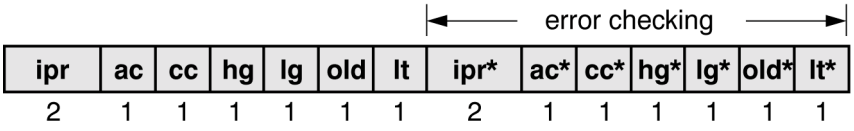


Figure 46 – Logical idle-symbol encoding

The idle-priority field, *idle.ipr*, is used to distribute the best current estimate of the ringlet's highest priority.

The allocation-count field, *idle.ac*, toggles when all nodes have had an opportunity to transmit a send packet. This field *idle.ac* is used to cancel target-queue reservations when busied send packets are never re-sent.

The circulation-count field, *idle.cc*, toggles when an idle has circulated around the ringlet. This field is used to detect lost echo packets and go bits.

The high-go and low-go bits, *idle.hg* and *idle.lg*, are the high priority and low priority bandwidth-allocation-control flags respectively. They enable allocation in an approximate round-robin order between nodes of the same priority class.

The age bit, *idle.old*, is set by the scrubber and cleared by other nodes under certain conditions, as part of a mechanism to monitor proper ringlet activity.

The low-type bit, *idle.lt*, allows the symbol to be consumed by nodes in the lower priority or highest-priority classes, when it is 0 or 1 respectively.

See 3.6 for an explanation of the use of these bits.

3.3 Logical packet encodings

3.3.1 Flag coding

SCI transactions are implemented as contiguous groups of nonidle symbols called packets, sent between a requester and a responder. All packets consist of an integer multiple of four symbols. Special sync packets are also used during initialization of each link; although the format of these sync packets is physical-layer dependent, they are always 8 symbols in length. Idle symbols (illustrated as i) are transmitted between packets to maintain synchronization and transfer flow-control information, as illustrated in figure 47.

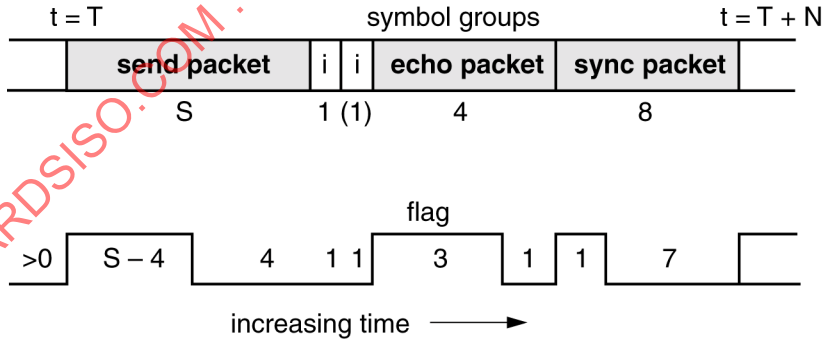


Figure 47 – Flag framing convention

One idle symbol is postpended to each send packet when it is produced. If there is more than one idle symbol between packets, the first is reserved for stripping by elastic buffers (to compensate for small differences in clock frequency at the various nodes) and the rest may be used to empty bypass FIFOs. Internal packets (after elasticity-buffer processing) may be back-to-back (as illustrated by the echo and sync packet), but there are enough idle symbols to handle the worst-case elasticity requirements.

The size of the fundamental SCI symbol is 16 bits. In addition, a clock signal is needed to define symbol boundaries (the data should be stable when sampled), and a flag signal is needed for locating the start and end symbols of a packet. No special start or stop symbols are needed or provided. Depending on the physical-layer encoding, some or all of these logical signals may be encoded and multiplexed onto one physical signal path.

A zero-to-one transition of the flag signal is used to mark the beginning of each packet, and the one-to-zero transition of the flag signal specifies the approaching end of each packet. The flag signal returns to zero for the final 4 symbols of send packets (to indicate when an echo should be created) and for the final symbol of an echo packet (to indicate when the CRC should be checked) as illustrated in figure 47. A zero always accompanies the CRC of any packet, so the zero-to-one transition can always be used to identify the start of the next packet (even when there is no idle symbol between them).

The first nonzero flag signal identifies the beginning of any packet. If the flag remains at one for at least four symbols, the packet is a send packet. The final symbol in the send packet is the CRC. It is identified by the fourth non-one flag bit. These send-packet framing conventions are illustrated in figure 48.

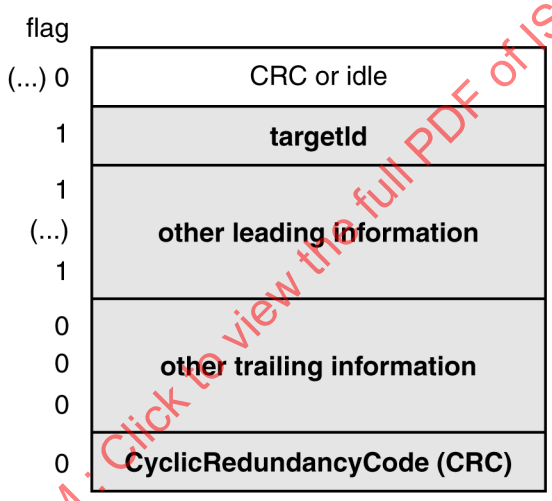


Figure 48 – Logical send- and init-packet framing convention

An echo packet has a sequence of three nonzero flag bits, while send packets always have four or more and sync packets have only one, as shown in figures 49 and 50. (For most purposes a bit in the command field is used to identify an echo for processing, because that bit provides earlier identification of the echo than the flag sequence does.)

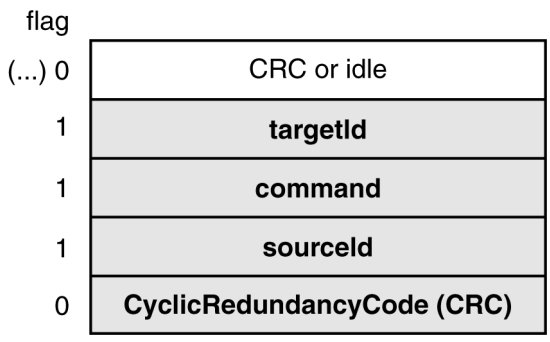


Figure 49 – Logical echo-packet framing convention

Sync packets are generated by each node during the initialization sequence and from time to time during normal operation so the downstream neighbour can deskew its receiver's data paths. The logical sync and send packets can easily be distinguished by the flag bit (which is high for only the first symbol of the sync packet), as illustrated in figure 50.

flag	
(...) 0	CRC or idle
1	**targetId**
1	**command**
1	**sourceId**
0	**CyclicRedundancyCode (CRC)**

Figure 50 – Logical sync-packet framing convention

For some physical encodings, the physical and logical encodings of the sync symbol are identical. These "simultaneous" transitions of the flag signal and all data signals (and the clock signal too, not shown) can be used for observing and compensating for differences in the signals' arrival times. The seven zero symbols are intended to provide a well-defined high-to-low transition for calibrating phase detection hardware in the data receivers. (Relatively large skews may be produced by inexpensive cables, and automatically compensated for by advanced SCI interfaces when circuit technology permits).

Abort packets are generated by any node that wishes to initiate a reset, in order to cleanly abort an arbitrary symbol stream being transmitted by the node (and cleanly stop packet processing on the downstream receiver). An *abort* packet is always *immediately* followed by a sync packet; this sequence generates a flag pattern that can never be interpreted as a valid packet. The *abort* packet uses a special *nodeId* address (table 13), repeated six times, followed by two zero symbols, with the flag set to one for the first six symbols and to zero for the last two, as illustrated in figure 51. The repetition of the *abort* address is for implementation convenience. Since this can never be interpreted as a valid packet, only the flag pattern is significant.

flag	
(...) x	any
1	**1111111111111011**
1	**1111111111111011**
1	**1111111111111011**
1	**1111111111111011**
1	**1111111111111011**
1	**1111111111111011**
0	**0000000000000000**
0	**0000000000000000**

Figure 51 – Logical *abort*-packet framing convention

3.4 Transaction types

3.4.1 Transaction commands

The command in a send packet falls into one of four main categories: response-expected request, move request, event request, and response. It is specified by a 7-bit command field and the 6 lsb's of the responder's offset address, as described in tables 8, 9, and 10.

Applications that might wish to have other (user defined, nonstandard) command types for special purposes should achieve their goals by using address bits in special ways rather than by redefining a standard command or using a reserved command. This provides sufficient flexibility for customization, while minimizing the risks of incompatibility. However, the length of these specialized subactions shall not be greater than the length specified by the send packet's *command.cmd* field, and shall be multiples of 16 bytes.

Table 8 – Response-expected-subaction commands (read, write, and lock)

command field	address lsb's	request transaction	req bytes	resp bytes	description
000ffff	aaaaaa	readsb	0	16	selected-byte read
001ffff	aaaaaa	writesb	16	0	selected-byte write
010ffSS	aaaass	locksb	16	16	selected-byte lock
0110000	0bbbbb	nread256	0	256	noncoherent memory read
0110000	1bbbbb	nread64	0	64	noncoherent memory read
0110001	aarrrr	nwrite16	16	0	noncoherent memory write
0110010	rbbbbb	nwrite64	64	0	noncoherent memory write
0110011	rbbbbb	nwrite256	256	0	noncoherent memory write
0110100	0rmmmm	mread00	0	0	coherent memory control
0110100	1rmmmm	mread64	0	0,64	coherent memory read
0110101	aammmm	mwrite16	16	0	coherent memory write (subline)
0110110	rrmmmm	mwrite64	64	0	coherent memory write (line)
0110111	rrmmmm	reserved	256	0	reserved for extensions
1110000*	0ccccc	cread00	0	0	cache-to-cache control
1110000*	1ccccc	cread64	0	0,64	cache-to-cache read
1110001	–	reserved	16	–	reserved for extensions
1110010*	cccccc	cwrite64	64	0	cache-to-cache write
1110011	–	reserved	256	–	reserved for extensions

NOTES

01110X0*	cread00, cread64, and cwrite64 transactions shall have extended headers
aaaaaa	least-significant address bits
cccccc	specify cache access codes
bbbbb	specify block data-transfer hints (for contiguous DMA transfers)
mmmm	specify memory-access codes
rrrr	specify reserved address bits
SSss	sub-command modifier bits, see lock and read transactions
ffff	final selected-byte address

Table 9 – Responseless-subaction commands (move)

command field	address lsb's	request transaction	req bytes	resp bytes	description
100ffff	aaaaaa	smovesb	16	–	start broadcast selected-byte move
101ffff	aaaaaa	rmovesb	16	–	resume broadcast selected-byte move
110ffff	aaaaaa	dmovesb	16	–	directed selected-byte move
0111000	rrrrrr	smove00	0	–	start broadcast 00-byte move
0111001	aarrrr	smove16	16	–	start broadcast 16-byte move
0111010	rrrrrr	smove64	64	–	start broadcast 64-byte move
0111011	rrrrrr	smove256	256	–	start broadcast 256-byte move
0111100	rrrrrr	rmove00	0	–	resume broadcast 00-byte move
0111101	aarrrr	rmove16	16	–	resume broadcast 16-byte move
0111110	rrrrrr	rmove64	64	–	resume broadcast 64-byte move
0111111	rrrrrr	rmove256	256	–	resume broadcast 256-byte move
1110100	rrrrrr	dmove00	0	–	directed 00-byte move
1110101	aarrrr	dmove16	16	–	directed 16-byte move
1110110	rrrrrr	dmove64	64	–	directed 64-byte move
1110111	rrrrrr	dmove256	256	–	directed 256-byte move

NOTES
aaaa least-significant address bits
rrrr reserved address bits

Table 10 – Event- and response-subaction commands

Event-subaction commands					
command field	**address lsb's**	**request transaction**	**req bytes**	**resp bytes**	**description**
1111000	rrrrrr	event00	0	–	clockStrobe signal
1111001	aarrrr	event16	16	–	reserved for 16-byte events
1111010	rrrrrr	event64	64	–	reserved for 64-byte events
1111011	rrrrrr	event256	256	–	reserved for 256-byte events
Response-subaction commands					
command field	**address lsb's**	**request transaction**	**req bytes**	**resp bytes**	**description**
1111100	--na--	several	–	0	status and 00-byte data return
1111101	--na--	several	–	16	status and 16-byte data return
1111110	--na--	xread64	–	64	status and 64-byte data return
1111111	--na--	xread256	–	256	status and 256-byte data return

NOTE
-na- not applicable (response has no address-offset field)
aaaa least-significant address bits
rrrr reserved address bits

The nread256 and nwrite256 transactions are provided to access large blocks of data. Although the accessed block address is 256-byte aligned, the starting address for the transaction is 64-byte aligned (so the most critical 64-byte line can be transferred first). That is, the 256-byte data transfer begins with the addressed 64 bytes and continues with the other 192 bytes of the aligned block in ascending address order (modulo 256). This is expected to reduce latency when efficiently accessing nonsequential data. However, the validity of the early data should not be assumed until the CRC has been checked, so the consumer should be able to undo side effects until the CRC has been verified.

Zero-length reads involve no transfer of data and cause no side effects on the data. However, the zero-length versions of the mread and cread transactions may have the side effect of updating the coherence state of the addressed cache line.

A move transaction shall have no response, but its acceptance may be delayed (its echo may have a busy status). Move transactions may be directed to one node, or to multiple nodes each of which has broadcast capabilities. The directed move transaction uses the dmovexx commands; the broadcast move transaction uses the smovexx commands (when first sent) and the rmovexx command (while being retried, after a busy status is returned).

An event transaction is special, in that its acceptance is never delayed. Event transactions shall have no response, shall be accepted immediately, and shall never generate an echo. Only one of the event transactions has a currently defined meaning – the event00 transaction is used to distribute the (time-of-day) clock-synchronization strobe signal, *clockStrobe*.

The four response subaction commands differ in the amount of data that follows. To avoid deadlocks, response subactions are expected to be placed in different queues from request subactions (reads, writes, locks, and moves).

Most SCI transfers use a small set of data sizes, for efficient queue storage management at high speed. However, odd-size or small transfers (1 to 16 bytes) are necessary to support I/O registers or adapters to existing bus designs. The selected-byte transfers (*readsb* and *writesb*) are implemented as ordinary 16-byte data transfers, with bits in the address and subcommand specifying the pertinent address range within the data transferred. The four least-significant bits of the address (called *a15*) specify the address of the first data byte; the four least-significant bits of the command (called *f15*) specify the address of the last data byte, as illustrated in figure 52.

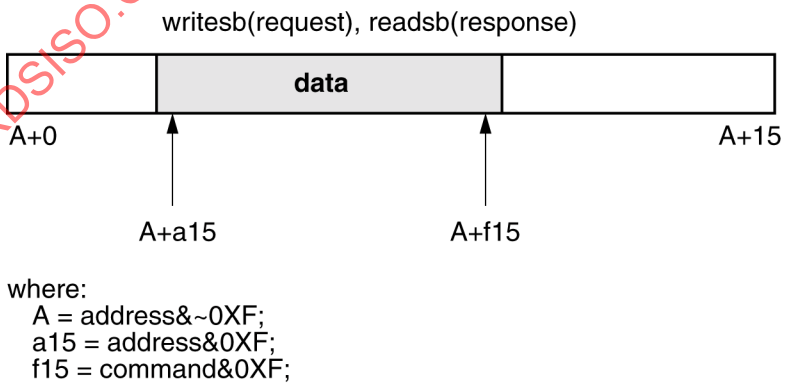


Figure 52 – Selected-byte reads and writes

The last byte address shall be equal to or larger than the first byte address. The unselected data may be undefined and shall be ignored.

3.4.2 Lock subcommands

Because of the distributed nature of SCI configurations, the interconnect cannot reasonably be locked while transaction sequences implement indivisible (e.g., test&set) operations on a memory location. Therefore, special lock transactions are defined that (for noncoherent accesses) communicate the intent from the requester to the responder, allowing indivisible updates to be performed at the responder. There is one standard lock transaction format, with several subcommands to define conditional and unconditional update actions that can be used for noncoherent memory accesses.

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

In this 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 53.

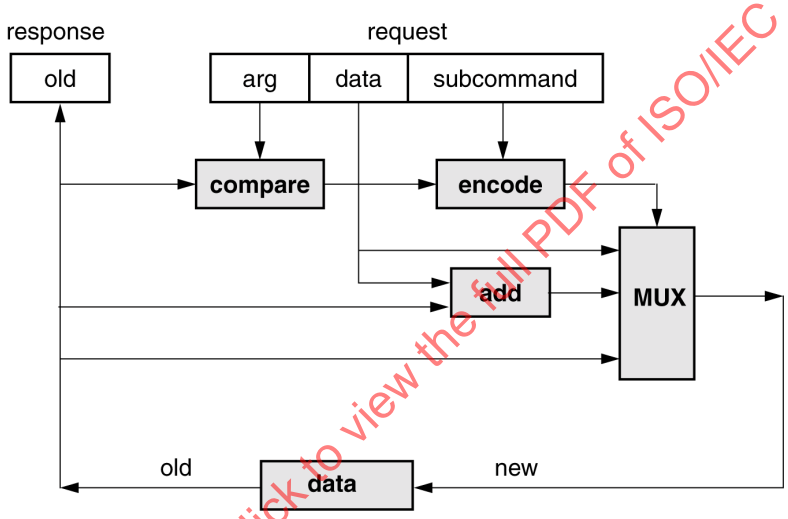


Figure 53 – Simplified lock model

The three data values (*data*, *arg*, and *old*) are all the same size, and are either quadlets or octlets. The specified set of locks, listed in table 11, is consistent with those defined by the CSR Architecture.

Table 11 – Subcommand values for Lock4 and Lock8

SSss	name	update*
0000	–	(not used)
0001	MASK_SWAP	new = (data&arg)|(old&~arg);
0010	COMPARE_SWAP	if (old==arg) new = data; else new = old;
0011	FETCH_ADD	new = old+data;
0100	LITTLE_ADD †	new = LittleAdd(old,data);
0101	BOUNDED_ADD	if (old!=arg) new = data+old; else new = old;
0110	WRAP_ADD	if (old!=arg) new = data+old; else new = data;
0111	vendor-dependent †	new = op(old,data,arg);
1000-	reserved[8] ‡	new = op(old,data,arg);
1111		

NOTES
* C-code notation used to define update actions
† Optional subcommands
‡ Reserved encodings for future definition by the CSR Architecture

Since the two least-significant bits of the address (ss) are not needed for addressing purposes, they are appended to the two least-significant bits of the command field (SS) to specify the 4-bit lock subcommand value. The two least-significant bits of the command specify the two most-significant bits of the lock subcommand; the two least-significant bits of the address specify the two least-significant bits of the subcommand.

The two next-more-significant bits in the address (aa in table 8) specify the first quadlet-aligned address of the data; the two next-more-significant bits in the command (ff in table 8) specify the last quadlet-aligned address of the data.

The MASK_SWAP, COMPARE_SWAP, FETCH_ADD, BOUNDED_ADD, and WRAP_ADD sub-commands shall be supported by SCI memory controllers. The LITTLE_ADD subcommand should be supported and one vendor-dependent subcommand may also be implemented.

Four lock subcommands involve addition of multiple-byte entities (quadlets or octlets); for these subcommands, a byte-significance specification is needed to correctly determine the direction of byte-carry propagation. For the FETCH_ADD, BOUNDED_ADD, and WRAP_ADD subcommands, a big-endian byte-significance is assumed (byte 0 is most significant). For the LITTLE_ADD subcommand, a little-endian byte-significance is assumed (byte 0 is least significant).

Lock transactions are constrained to access aligned quadlets and octlets. To simplify the hardware implementation, the least-significant bits of *data* and *arg* values are right justified within the two halves of the packet's 16-byte data field, as illustrated in the request portion of figures 54 and 55.

For a quadlet lock access, *old* is returned in one of four data positions. Figure 54 illustrates the format of the 16 bytes in the lock transaction request and response packets; the format of the request subaction is independent of the data address. Four formats for the response subaction are illustrated, one for each quadlet address.

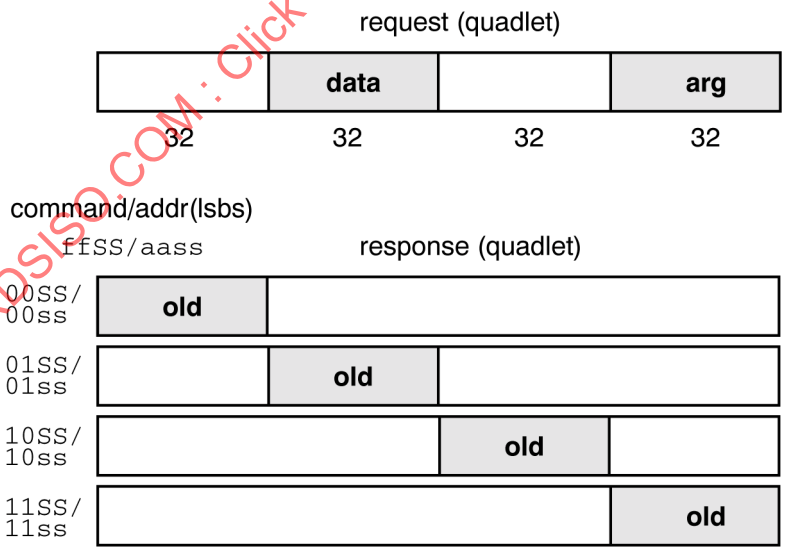


Figure 54 – Selected-byte locks (quadlet access)

For an octlet lock access, *old* is returned in one of two data positions, as illustrated in figure 55.

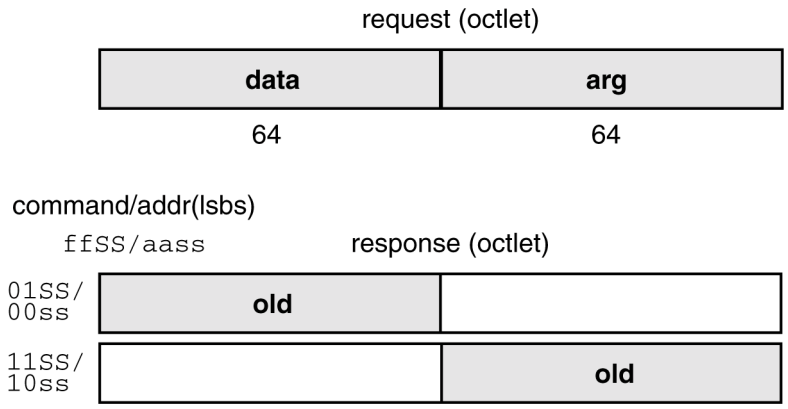


Figure 55 – Selected-byte locks (octlet access)

The unselected data in lock requests and responses (illustrated as blank boxes in figures 54 and 55) are undefined and shall be ignored.

3.4.3 Unaligned DMA transfers

The serial version of SCI is expected to be used for interconnecting distributed systems that may be based on the parallel version of the SCI standard or on (e.g.) bus-backplane standards. In distributed systems many of the transfers between subsystems may be large DMA-initiated transfers with cache- or page-aligned addresses and lengths. Although many DMA transfers are expected to access pages at page-aligned addresses, SCI also supports transfers to unaligned addresses, to efficiently support smaller transfers used for network- or terminal-transfer traffic.

When transferring noncoherent data from memory to a peripheral, a DMA controller is expected to primarily use nread64 transactions. For unaligned transfers the first and last nread64 transactions are likely to contain unneeded data, but using the smaller readsb transactions to transfer only the needed data would generally be more complex and less efficient. This use of nread64 transactions is illustrated in figure 56; the shaded portions of the blocks illustrate which data addresses are involved in the transfers.

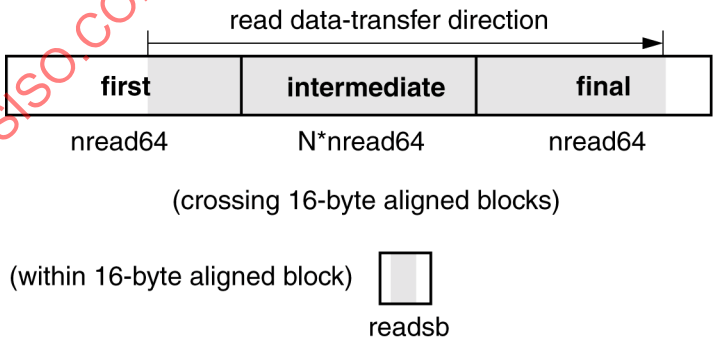


Figure 56 – Expected DMA read transfers

If nread256 transactions are supported, they may be used instead of nread64 transactions.

A single readsb transaction is expected to be used for small DMA transfers that are contained within a 16-byte aligned address block. The readsb transaction is more efficient than an nread64 transaction, and transparently supports DMA transfers to memory-mapped control registers (which may not support nread64 transactions).

When transferring noncoherent data into memory from a peripheral, a DMA controller is expected to use writesb, nwrite16, and nwrite64 transactions. For a poorly aligned transfer, the first transfer would use a writesb transaction to modify data up the next 16-byte-aligned address. The next transfer would use up to three nwrite16 transactions to modify the data up to the next 64-byte-aligned address, as illustrated in figure 57.

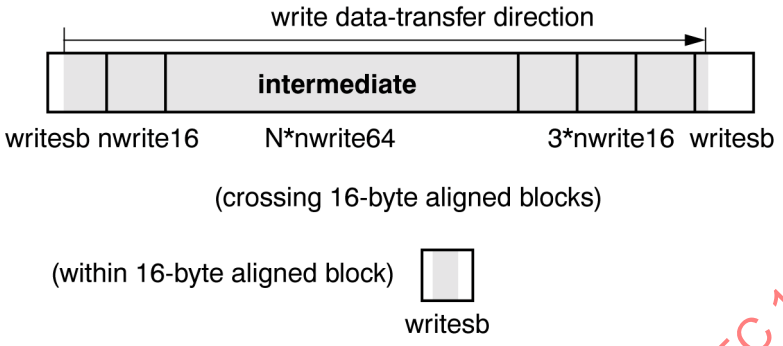


Figure 57 – Expected DMA write transfers

Intermediate transfers are expected to use the more efficient nwrite64 transactions. The final transfers are expected to use up to three nwrite16 transactions to modify data up to the final 16-byte-aligned block. The final writesb transfer is expected to modify data up to the final byte-aligned address.

A short transfer that is entirely contained within a 16-byte-aligned block shall be performed using a single writesb transaction.

Nodes that support nwrite256 may use them in the intermediate phase, using nwrite64 transactions to reach a 256-byte-aligned boundary.

Writesb transactions are sufficient to implement the entire write transfer, but are significantly less efficient. Similarly, writesb may be used with nwrite16 transactions to implement a less-efficient write transfer.

System efficiency can be improved in some applications if intermediate bridges are given some hints that let them prefetch or buffer data in the intermediate phase of a DMA transfer, as described in the following subclause.

3.4.4 Aligned block-transfer hints

To improve performance for intermediate 64-byte (or optionally 256-byte) aligned data transfers, transaction encoding space is provided for the DMA controller to communicate its intent to intermediate bridges, which may prefetch data, basing their prefetch decisions on the DMA controller's announced intent.

Hints also indicate that other nodes are not expected to use or modify the data while it is being transferred. For read transfers a bridge may safely prefetch data (which would become stale if they were modified later). For write transfers a bridge may pre-purge data (i.e., discard modified cached copies), if the purged data are eventually updated by the subsequent transactions.

The 5-bit data-transfer hints conveyed in the 5 least-significant address bits (see table 8) are provided for noncoherent 64-byte and 256-byte data transfers (*nread64*, *nread256*, *nwrite64*, and *nwrite256*). These long transfers have five phases, *first*, *start*, *continue*, *near*, and *last*. For the *first*, *start*, *continue*, and *near* phases, three of the data-transfer hint bits specify a phase-length parameter *N* (in multiples of the transaction size). The transaction command specifies the parameter *B*, which is the block size of the individual transactions.

A data-transfer hint shall only be used on a transfer to a contiguous range of physical addresses, and the phase-length parameter *N* shall be the same for all transactions within the transfer. When hints are provided, a normal transfer shall consist of a first transaction, a *start* phase containing *N* – 1 transactions, a variable-length *continue* phase, a *near* phase with *N* – 1 transactions, and a 1-transaction *last* phase, as illustrated in figure 58.

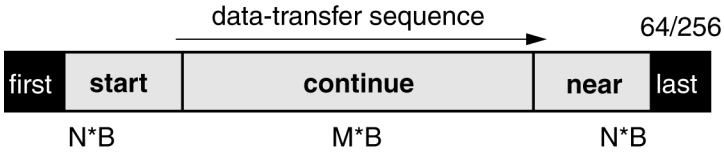


Figure 58 – DMA block-transfer model

While performing transactions within the transfer, the number of simultaneously outstanding transactions shall be limited to *N*, and the transaction at address *A+N*B* shall not be initiated until the transaction at address A has completed. A data transfer may be prematurely terminated at any point (the *start*, *continue*, *near*, and *last* phases are therefore optional).

Bridges may use these transaction hints to prefetch read data or to concatenate writes into larger blocks. Note that prefetch algorithms shall tolerate out-of-order transaction delivery; on a congested system, flow control mechanisms may reorder transactions before they are accepted by the bridge.

The behaviour of a simple prefetching bridge, when processing read transactions to address *A*, is expected to be as follows: During the *start* phase, previously prefetched data at address *A* are not used and previously prefetched data at addresses *A* and *A+N* are discarded. During the *continue* phase, previously prefetched data at address *A* are used and data are prefetched for address *A+N*. During the *near* phase, previously prefetched data at address *A* are used and data are not prefetched for address *A+N*. During the *last* phase, previously prefetched data at address *A* are used and all transfer-related prefetch resources are released.

The behaviour of a simple prefetching bridge, when processing write transactions to address A, is expected to be as follows: During the *start*, *continue*, and *near* phases, a write through the bridge is delayed until the next transaction (with the next larger address) is accepted by the bridge (or until a short timeout period); this provides the opportunity to merge four shorter 64-byte writes into one longer 256-byte write transaction, for example. The write transaction in the *last* phase is merged with previously buffered packets, but the emptying of this merged buffer is not delayed.

Note that any transfer may be terminated early, when a short data transfer terminates or after a DMA-controller failure. For a short DMA transfer (when the number of transferred bytes is less than the number requested) the *near* phase can be eliminated. In normal operation, the *last* phase is expected and can be used to improve bridge performance. However, the terminating *near* and *last* phases should only be used to improve the bridge's read-prefetch or write-merge performance, since they cannot be relied upon to correctly terminate transfer-related prefetch activity.

The 5 least-significant bits of the transaction address uniformly specify block-transfer hints for noncoherent nread64, nread256, nwrite64, and nwrite256 transactions, as specified in table 12.

Table 12 – Noncoherent block-transfer hints

bbbbb	phase	phase-length (N)	description
00000	–	none	no data-transfer hints
XX001	(below)	1	data transfer, shortest prefetch
XX010	(below)	2	data transfer, short prefetch
XX011	(below)	4	data transfer, short prefetch
XX100	(below)	8	data transfer, short prefetch
XX101	(below)	16	data transfer, short prefetch
XX110	(below)	32	data transfer, short prefetch
XX111	(below)	64	data transfer, longest prefetch
00nnn	first	nnn	first contiguous data transfer
01nnn	start	nnn	starting contiguous data transfer
10nnn	continue	nnn	continue contiguous data transfer
11nnn	near	nnn	near end of contiguous data transfer
11000	last	–	last contiguous data transaction
01000	reserved	–	not used, reserved for other types of hints
10000	reserved	–	not used, reserved for other types of hints

3.4.5 Move transactions

Move transactions are acknowledged by an echo (for the purposes of flow control), but have no end-to-end delivery confirmation. Since there is no response to confirm when or whether the request has been delivered, higher-level protocols are needed to confirm that move transactions have completed successfully. These higher-level protocols are beyond the scope of the SCI standard, but could include the following:

1) *Time delays*. Some types of data, such as video or some kinds of physics data, are loss tolerant, in that the data are still useful when small portions have been lost. A fixed time delay may be provided for such transfers to complete; after that time delay, late-arriving data will be discarded.
2) *Requester credits*. The requester moves the data to the responder, up to an amount established by its credit value. When a sufficient number of move transactions has been received and processed the responder directs a write transaction to the requester that increases the credits for the requester.
3) *Constrained ordering*. For certain configurations, all transactions with the same requester and responder ringlets will be routed on the same path through intermediate switches or bridges, called agents. The agents may be designed to flush previously-queued move subactions before forwarding read or write subactions with the same requester *nodeId* value (as specified by the request subaction's *sourceId* value). To preserve order, the move subactions would be forwarded (and discarded from the agent's queues) before the forwarding of the following read or write subaction is initiated.

To improve their performance when constrained ordering (3) is provided, DMA command architectures may have the capability of using move transactions for all but the last transaction in large DMA transfers. However, the DMA command architectures should allow this feature to be selectively disabled (on an address-block basis) in configurations without constrained ordering.

Errors are also harder to log and contain when move transactions are used. These error logging and containment strategies are agent-architecture dependent and beyond the scope of the SCI standard, but could include the following:

1) *Error logging*. The agent would log move-transaction errors when they are detected. The error log could be periodically polled, or the agent could periodically interrupt a pre-specified processor when this error log changes.

2) *Error containment.* After a move-transaction error is logged, an agent may disable further accesses to the now-corrupted data. Accesses may be disabled on a global basis (all accesses to the corrupted data are blocked) or on a selective basis (accesses from the same requester are blocked).

3.4.6 Global time synchronization

The SCI standard supports global clock synchronization (referring to time clocks, not data transmission clocking), within the framework provided by the CSR Architecture. All SCI nodes should maintain local timers (formatted as 64-bit integer-seconds/fraction-seconds counters). A *clockStrobe* (*event00*) packet provides the signal that maintains clock synchronization between SCI nodes on the same ringlet.

To support clock synchronization on SCI, all nodes provide a *through* register and the clock-capable nodes also provide an *arrived* register. The *clockStrobe* packet is generated by a clock-master (one on each ringlet, assigned by software), is routed through the other local nodes, and is ultimately stripped when it returns to the clock master, as illustrated in figure 59.

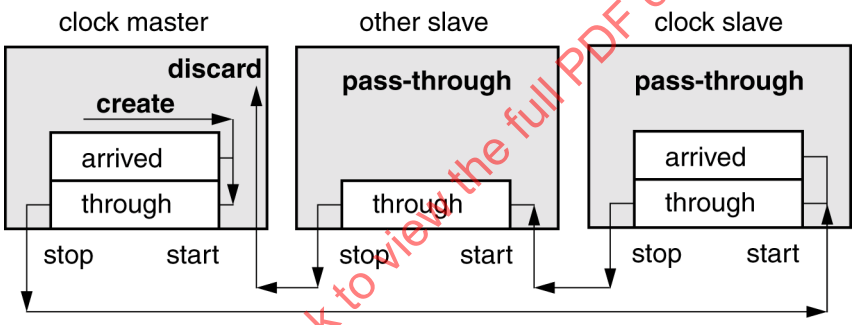


Figure 59 – Time-sync on SCI

For the clock master, the *through* register measures the time between the *clockStrobe* packet's creation and its transmission. For all other lincs, the *through* register measures the time between the arrival and departure of the *clockStrobe* packet. To minimize costs and improve accuracy, the *through* register is calibrated in terms of ringlet symbol times. For the clock master, the *arrived* latch saves the clock value when the *clockStrobe* packet is created. For other lincs, the *arrived* latch saves the clock value when the *clockStrobe* packet is received.

By analyzing the latched values of the *arrived* and *through* registers, and knowing the physical connection delays (cable lengths, etc.), software can make all clocks consistent and can compensate for frequency drifts between nodes in the system. For the purposes of clock-time logging, the precise arrival and departure time of the *clockStrobe* packet shall be defined to be at the trailing edge of its CRC symbol.

Multiple-linc switch nodes require additional resources for routing *clockStrobe* signals between SCI ringlets. See the C code for details.

3.5 Elastic buffers

3.5.1 Elasticity models

An SCI node usually has its own clock (referring now to data transmission clocking, not time clocks), which is approximately (but not exactly) the same frequency as the clock of any other node attached to the same ringlet. Since the clocks on separate nodes will drift in phase over time, symbols will sometimes need to be deleted (when the received clock is faster than the internal clock) or inserted (when the received clock is slower than the internal clock). The symbols that are inserted or deleted are idle symbols, which can only be between packets, (except that the last symbol of a sync packet may also be deleted). These are called *elasticity* symbols.

Note that SCI nodes are entirely synchronous, and that data transmission is "source-synchronous". The only asynchronous part of SCI is in the first stage of the receiver, where the incoming data may have an arbitrary (and slowly drifting) phase with respect to the remainder of the node. The sync packet provides enough information for a receiver to dynamically compensate for phase shifts on individual bits independently. However, this capability should not be needed in most systems.

The synchronous nature of an SCI node greatly simplifies operation at these very high speeds; for example, the FIFOs do not have to be concerned with metastability problems, which would slow them enormously. However, data receivers must be carefully designed to compensate for incoming clock phase drifts and to ensure that sampling latches never have their setup and hold time specifications violated. Metastable responses of receiver latches can last for many bit times, so they must be avoided except, of course, during initialization training of the link. Synchronous operation also greatly simplifies the operational description of the node, allowing its behaviour to be simulated with great confidence.

To guarantee a sufficient number of deletable idle symbols, a packet is never transmitted without at least one idle symbol separating it from the previous packet, and an idle symbol is postpended to any packet that is sent, unless the bypass buffer is full. Any idle symbol can be deleted as necessary, but some of the information it carried must be saved for use by the allocation protocols.

Idle symbols are deleted or inserted by means of an elastic buffer at the SCI input port as shown in figure 60.

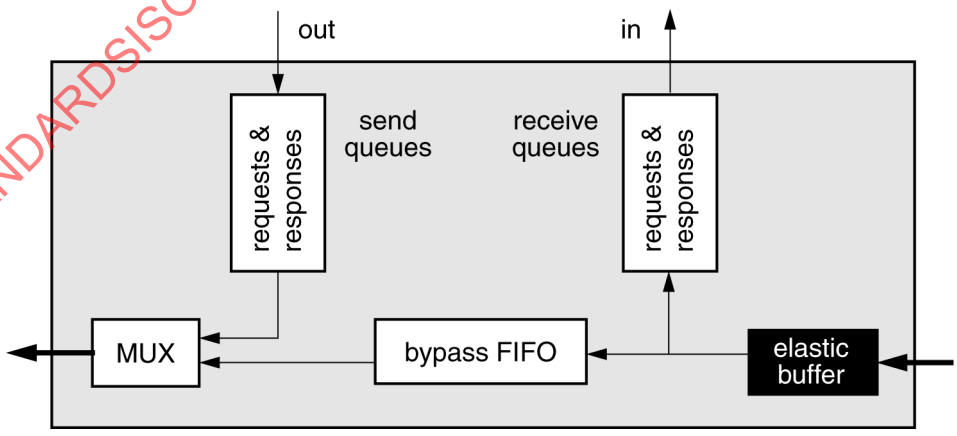


Figure 60 – Elasticity model

The input data synchronizer, which is responsible for the insertion and deletion of elasticity symbols, contains a multiple-tap two-symbol delay element (of length 2 T, where T is the duration of one symbol). A symbol is inserted when the delay is increased by T (tap *a* to *b*, as shown in figure 61), and a symbol is deleted when the delay is decreased by T (tap *b* to *a*). The shading illustrates the delay ranges that have different effects on the insertion and deletion of idles. A typical implementation of this delay element might be sixteen taps to provide a total delay of two symbols.

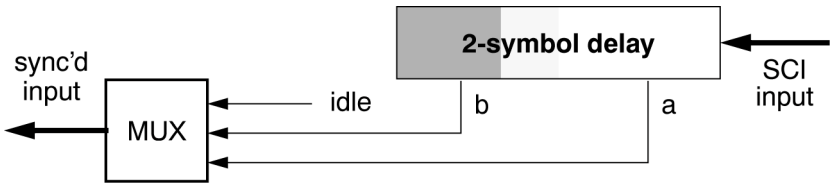


Figure 61 – Input-synchronizer model

During normal operation the received data clock is monitored and compared with the node's clock, and a different tap is selected from time to time to ensure that the input data are never sampled near a transition. Thus, as the relative phases of the two clocks drift, the tap changes and the delay in the elastic buffer varies. The elastic-buffer protocols are robust, in that they can support an arbitrary number of nodes on each ringlet.

3.5.2 Idle-symbol insertions

If the received clock is consistently slower than the node's clock, the delay will decrease and eventually reach zero. When this happens, the tap changes from *a* to *b*, which inserts an idle symbol and increases the delay by one symbol, as illustrated in figure 62. Idle insertion is inhibited within packets.

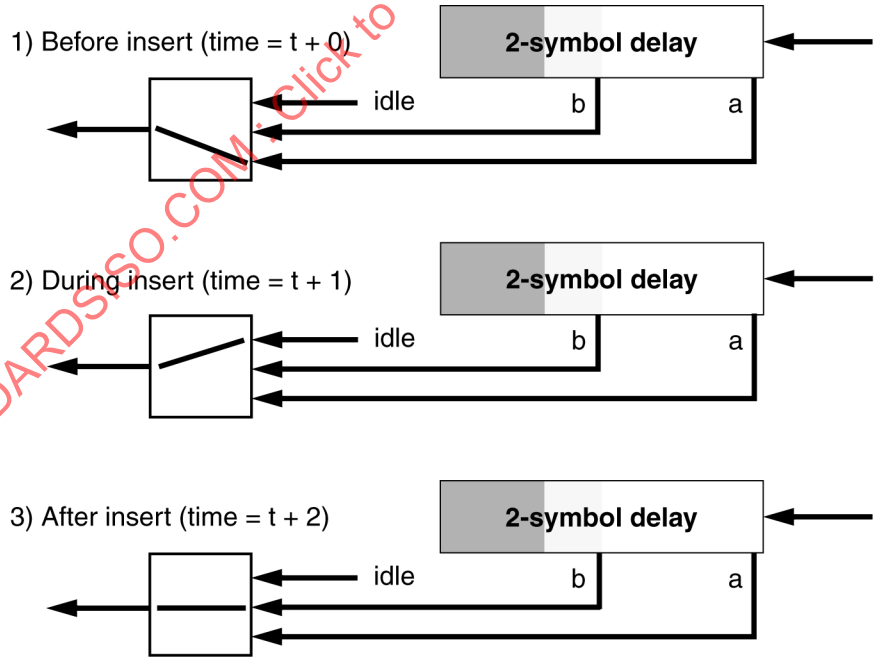


Figure 62 – Idle-symbol insertion

An idle symbol need not actually be inserted; an internal label (see 5.2.2) can be used to mark the input symbol for equivalent special processing by other parts of the node.

3.5.3 Idle-symbol deletions

Similarly, if the received clock is consistently faster than the node's clock, the delay will increase and could eventually reach two symbols. Before this happens, the tap changes from *b* to *a*, which deletes an idle symbol and decreases the delay by one symbol, as illustrated in figure 63. To avoid packet corruption, the idle deletion is only performed when the previewed symbol (at tap *a*) is known to be a deletable symbol (an idle or a sync-packet symbol). The go bits from deleted idles (which affect bandwidth allocation) are saved.

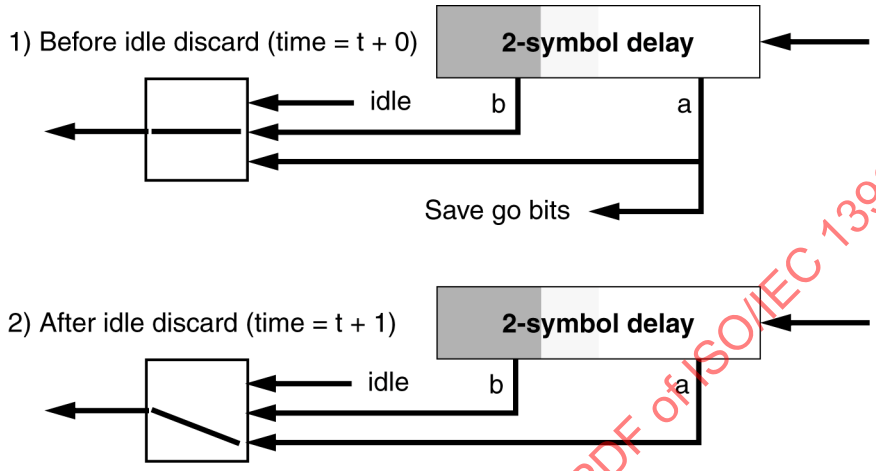


Figure 63 – Idle-symbol deletion

To avoid frequent insertions and deletions when the input and local clocks are closely matched, the idle deletion process is inhibited until the accumulated delay has exceeded 5/4 T.

A similar form of symbol deletion may be performed on the trailing symbol of a sync packet, which is sent periodically between adjacent nodes on the ringlet. Since this may provide a sufficient number of deletable symbols, particularly when nodes on one backplane share the same clock, the idle-symbol deletion capability is optional.

If sufficient idle symbols were not present, the deletion process could be inhibited until the two-cycle delay limit is exceeded. The node would lose data synchronization and the ring would need to be cleared and re-synchronized. To avoid this problem, the idle symbol at the end of every packet is reserved for elastic-buffer uses. (It is never deleted for allocation-related purposes.) Also, nodes should re-insert idles between back-to-back packets, unless their bypass buffers are full.

3.6 Bandwidth allocation

Computer buses use an arbitration mechanism to determine which processor gets exclusive use of the bus. SCI ringlets consist of a number of nodes connected by point-to-point links and performance would degrade if only one active transaction were allowed on each ringlet at any given time. Instead of arbitrating for exclusive access to the ringlet, SCI's protocols for allocating ringlet bandwidth allow multiple nodes to transmit packets concurrently. This bandwidth allocation protocol assures that all nodes are allocated at least a minimal bandwidth (independent of their priority), while most of the ringlet's bandwidth is reserved for nodes that have the highest active priority.

Bandwidth allocation protocols inhibit the transmissions of some nodes to ensure transmission opportunities for others. The allocation mechanism minimizes latency and overhead on an idle ringlet, while providing fairness and prioritized bandwidth partitioning.

Bandwidth allocation controls access to the ringlet, but if many producers direct send packets to the same consumer, the consumer may have insufficient space to queue all of these packets. For example, several processors may direct a sequence of requests to the same memory controller. In such a situation queue space, not bandwidth, is the limiting resource and queue-allocation protocols are needed to ensure that no producer is starved (i.e., there is forward progress for all). Queue allocation protocols are discussed in 3.7.

Nodes may be fair-only, incapable of using the prioritized bandwidth, or optionally they may be unfair-capable, capable of using the fair as well as the prioritized bandwidth. Note that a system containing exclusively fair-only nodes will share all the bandwidth fairly; the protocol does not reserve prioritized bandwidth if no one needs it.

3.6.1 Fair bandwidth allocation

Fairness (among nodes in the same allocation-priority group) gives each node equal access to the ringlet, with no node preferred over any other. Fairness is enforced by round-robin protocols, based on go-bits in idle symbols. Separate lo-go and hi-go bits are provided, so fairness between nodes in the lower and highest allocation-priority groups can be maintained independently. The low-go bit, *idle.lg*, maintains fairness among lower nodes; the high-go bit, *idle.hg*, maintains fairness among highest nodes. In this clause, fair bandwidth allocation is assumed and only one of the two go bits, *idle.lg*, is considered. Though every idle actually has an *idle.lg* bit, with value 0 or 1, for simplicity in the following discussion only the value 1 is referred to as an *idle.lg* bit and an idle that has an *idle.lg* value of 0 is referred to as not having an *idle.lg* bit.

For a producer, send packets can only be transmitted by postpending them to an idle symbol that has an *idle.lg* bit (step 1 in figure 64). The transmitted packet is also followed by another idle, which is reserved for elasticity uses. On a lightly loaded ringlet, this constraint rarely delays the transmission of send packets, since most symbols are idles and their low-go bits are usually set. Only when the bandwidth approaches saturation does this constraint begin to delay transmissions.

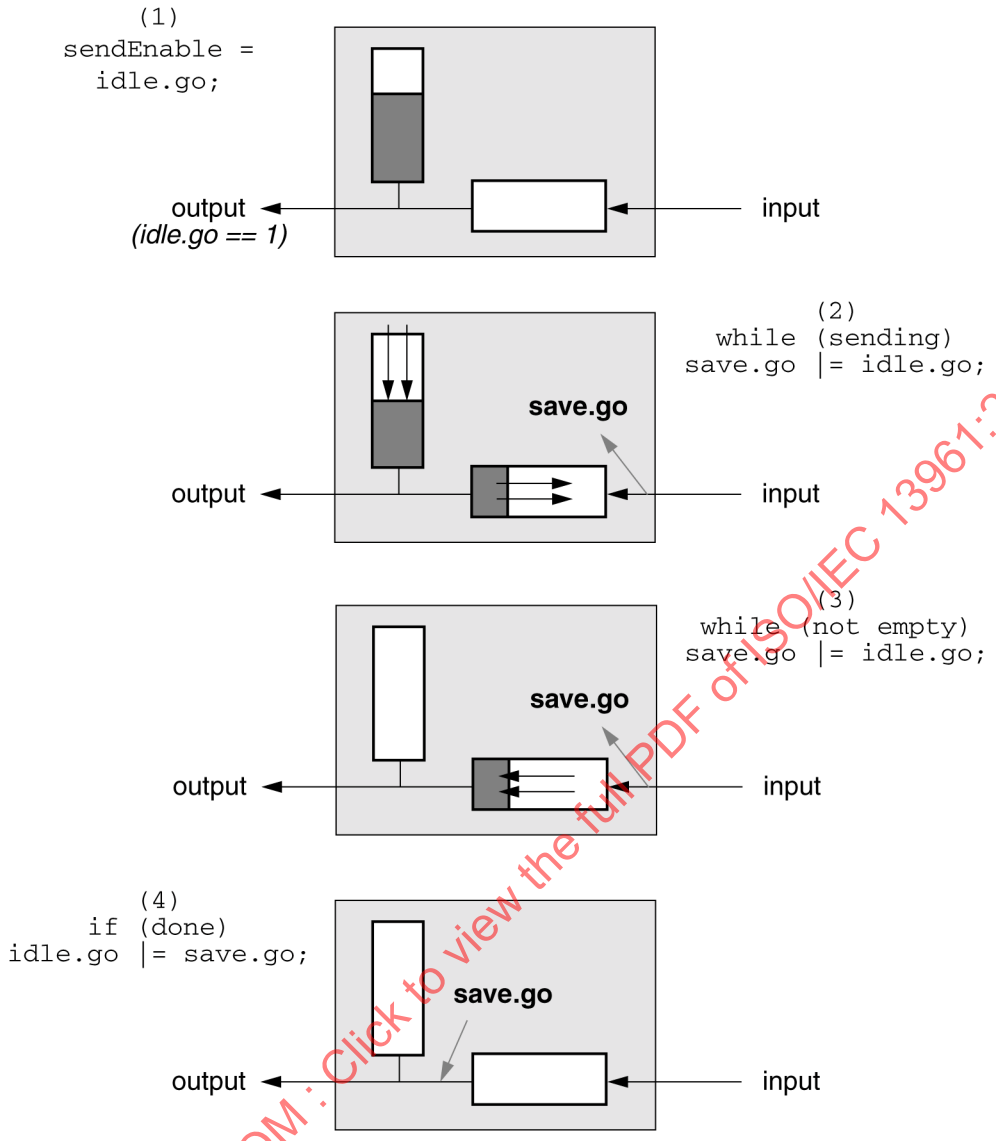


Figure 64 – Fair bandwidth allocation

When a transmission starts, forwarding of additional *idle.lg* bits is delayed (and the node is said to be blocked) until the node empties its transmit buffer and bypass FIFO. During this time consumable idles (those that follow another idle and have *idle.lt*==1 or *idle.ipr*==0) are discarded and their go bits are saved. Other packet symbols and nonconsumable idles are saved in the bypass FIFO, whose contents may increase while the send packet is being transmitted (step 2) and decrease after the transmission has ended (step 3).

In figure 64, ".go" refers to ".lg" or ".hg" , which are treated similarly. There is an internal go bit (*save.lg*) that is set to one when an idle with *idle.lg* set is consumed. Thus these incoming go bits are never discarded, but multiple go bits are sometimes merged into one.

The forwarding of *idle.lg* bits to the node's downstream neighbours is stopped (inhibiting additional transmissions from them) until the producer's transmission (of the packet it sent and of any symbols in its bypass FIFO) has ended. After the transmission ends, the saved go bit (*save.lg*) is released and put into the next idle symbol (step 4 in figure 64).

After a go bit has been released, it is extended into the immediately following idle symbol. A set *idle.lg* bit in one pass-through idle is extended so the *idle.lg* bit in the following output idle symbol is also set. These go bit extensions (which are also performed on the *idle.hg* bits) eventually fill the idle space between packets. Thus go bits fill an idle ringlet, reducing latency for access to a lightly loaded ringlet, yet act somewhat like a token that precedes a packet.

The go-bit extensions should be performed by the transmitter portion of a node, so the extended go bits can be used to quickly re-enable additional send-packet transmissions.

While the producer is transmitting (steps 2 and 3), it blocks the circulation of the allocation-count bit (*idle.ac*) within the idles that pass through it, by replicating the previous *idle.ac* value. The new allocation-count value passes through the node when the transmit buffer and bypass FIFO are empty and the node is re-enabled to send (step 1). By properly inhibiting the circulation of this *idle.ac* bit, changes in its value indicate that all local producers have had an opportunity to transmit (or re-transmit) their queued packets. These changes are monitored by the consumers, to detect failures in expected retransmissions.

3.6.2 Setting ringlet priority

The go bits are used to allocate bandwidth fairly among producers in the same bandwidth-allocation priority class. However, the highest-priority producers are allowed to consume more bandwidth than lower-priority producers. To implement this bandwidth partitioning, mechanisms are provided to dynamically determine the highest currently active producer priority, the *ringlet priority*. Each node maintains its own view of the current ringlet priority, based on its own priority and other priority information it observes in passing packets or idle symbols. The idle field *idle.ipr* distributes the ringlet priority determined by the node that has the most up-to-date information, i.e., a node that has just received an echo packet.

Request-send and response-send packets are assigned 2-bit priorities when created. This priority is stored in the *control.tpr* field of the packet's header (see 3.2), which specifies the packet's transaction-completion priority and is unchanged as the packet flows through the system. The strategy used to select the *control.tpr* values is beyond the scope of the SCI standard. A producer calculates a ringlet-local send priority, *command.spr*, based on the *control.tpr* value of its queued packets (some of which may have a higher priority).

The concept of using a ringlet-local send priority (*command.spr*) that may be higher than the transaction's priority (*control.tpr*) is often called priority inheritance. When transaction ordering cannot easily be changed, high priority packets temporarily increase the effective priority of the packets that block them. The original transaction priority is restored when the transaction moves to another consumer queue where it is no longer blocking other higher-priority packets.

The ringlet priority level is established by unfair-capable producers, based on their calculated *command.spr* values. As symbols pass through a blocked producer, this value is inserted in place of smaller *command.mpr* fields in send and echo packets and smaller *idle.ipr* fields in idle symbols, as illustrated in figure 65.

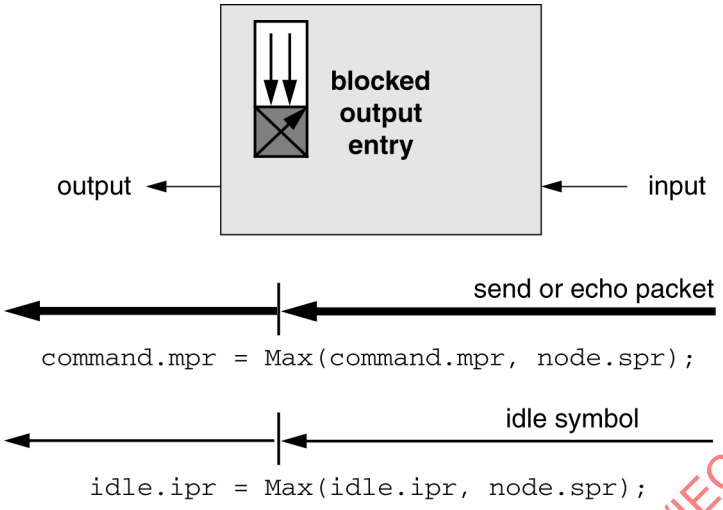


Figure 65 – Increasing ringlet priority

In the absence of some other priority-reduction mechanism, the ringlet priority would soon increase to the highest priority of any previously blocked producer. To avoid this priority escalation, nodes are responsible for restoring the ringlet priority when their echo packets are returned. When the producer's echo is returned, the maximum of its *command.mpr* and *command.spr* fields is saved for insertion into the *idle.ipr* field of subsequent idle symbols. This continues, as illustrated in figure 66, until the next send or echo packet is observed. This process quickly reduces the ringlet priority level to the most-recently sampled level.

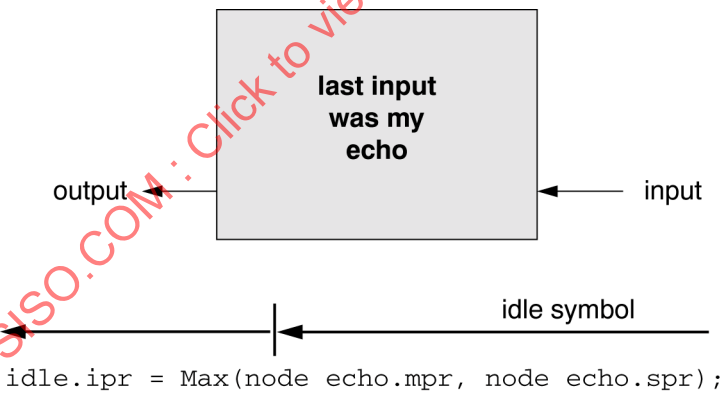


Figure 66 – Restoring ringlet priority

The *command.spr* field in the echo, which is set to the send packet's *command.mpr* value when the echo is created, provides the maximum of node priorities in the send-packet's path from the producer to the consumer. The *command.mpr* field in the echo, which is cleared when the echo is created, provides the maximum of the node priorities in the echo-packet's path from the consumer to the producer. (Note that event packets require special treatment because they have no echo. See the C code for details.)

These two segment priorities are kept separate to enable optimization of the performance of pipelined bandwidth allocation, where producers send most packets directly to their downstream neighbours. On such pipelined systems, the cumulative bandwidth may be much larger than provided by any individual link (the send-packet bandwidth can be reused after the send packet is stripped by the consumer). The *command.spr* and *command.mpr* fields are intended to support such pipelined bandwidth-allocation protocols, which are planned for future extensions of the SCI standard.

3.6.3 Bandwidth partitioning

The priorities of the nodes on the ringlet are divided dynamically into two allocation groups: the highest, consisting of all nodes having effective priority equal to or greater than their estimate of the ringlet priority, and the lower, consisting of all nodes. A goal is to apportion most of the bandwidth to the highest allocation group, while ensuring forward progress by leaving a residual bandwidth (which is partitioned fairly) for the lower group. Note that these groups are not mutually exclusive, as all nodes in the highest group are also in the lower group.

To implement the partitioning of ringlet bandwidth, two classes of idle symbols are created: high-type (*idle.lt*=0) and low-type (*idle.lt*=1). Allocation priorities restrict the consumption of these low- and high-type idles. The ratio of available low-type and high-type idles is influenced by the way these idles are created when send or echo packets are stripped from the ringlet. Fair-only nodes create only high-type idles, as illustrated in figure 67, where the quantities in square brackets represent the number of symbols.

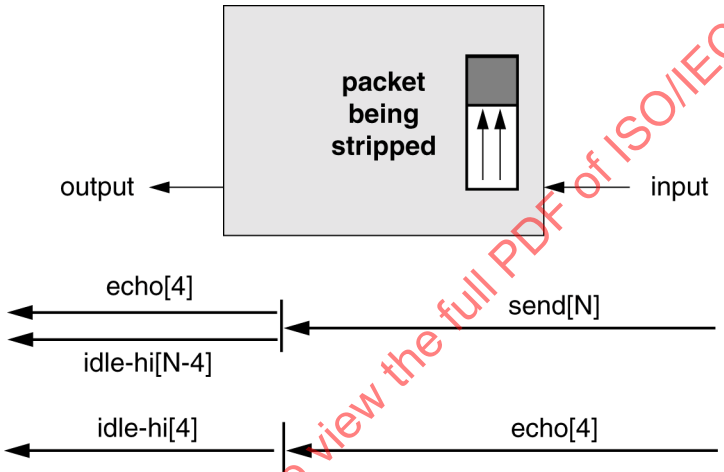


Figure 67 – Idle-symbol creation, fair-only node

Unfair-capable nodes are responsible for maintaining a mix of high-type and low-type idles. This mix is created by converting each subaction into many high-type idles and two low-type idles. A send packet, when stripped by the consumer node, creates an echo packet and many high-type idles, as illustrated in figure 68. An echo packet, when stripped upon returning to the producer node, is converted into a mix of high-type and low-type idles.

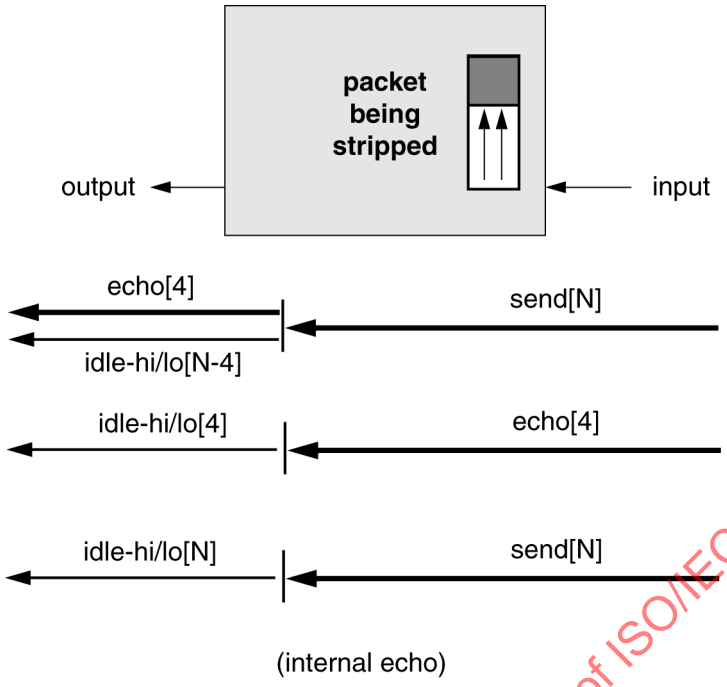


Figure 68 – Idle-symbol creation, unfair-capable node

When the producer and consumer are the same node (a node transmits a send packet to itself), stripping the send packet creates many high-idle symbols and a few low-idle symbols. This ensures the same ratio of low-type and high-type idles whether the producer and the consumer are the same or different nodes.

When prioritized packets are being sent, unfair-capable nodes (which consume most of the bandwidth and produce most of the idles) establish the ratio of low-type to high-type idles, which determines the proportion of the bandwidth available to the highest-priority and lower-priority producers.

Fair-only producers have to consume idle symbols to empty their bypass FIFOs, which may have accumulated part or all of an incoming packet while the node was transmitting. To avoid consuming bandwidth allocated to the highest priority group, fair-only nodes only consume high-type idles when the ringlet priority is zero, as illustrated in figure 69. Other (nonconsumable) idles are put into their bypass FIFOs, as are any packet symbols.

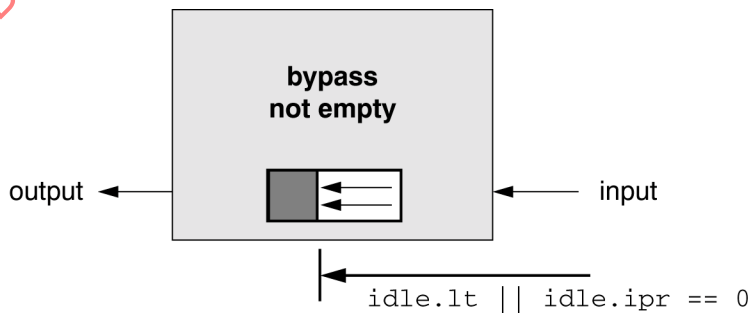


Figure 69 – Idle consumption, fair-only node

The consumption properties of an unfair-capable node are determined dynamically based on the node's consumption mode. Depending on its previous history, a producer may be able to only consume low-type idles (lower priority, fair), high-type idles (highest priority, unfair), or both (highest priority, fair), as illustrated in figure 70.

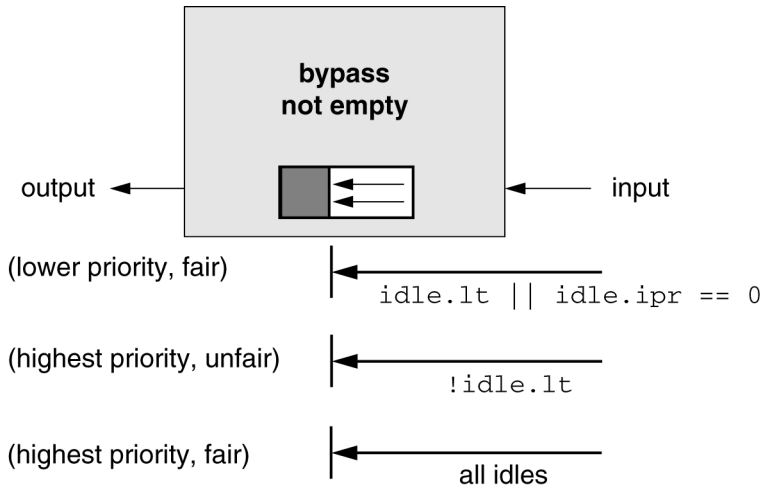


Figure 70 – Idle consumption, unfair-capable node

The preceding discussion was simplified for clarity. To further improve performance, an unfair-capable node consumes all idle symbols regardless of their type to empty its bypass FIFO, which may have accumulated part or all of an incoming packet. The desired selective-consumption behaviour is approximated by increasing a debt counter when a nonconsumable type of idle is consumed. After the bypass FIFO has emptied, the type of passing-through idles is converted until the debt has been repaid. This behaviour reduces the ringlet latency caused by storing symbols in the bypass FIFO.

A ringlet's priority and the availability of *go* bits can change dynamically, so the node's consumption restrictions may be changed while the send packet is being sent or before the echo packet is returned. Since any given node only knows the priority state of the ringlet as of some earlier time, it cannot make ideal allocation decisions. As a result, priority has only a minor effect on latency, but eventually affects the bandwidth allocation.

3.6.4 Types of transmission protocols

The simpler pass-transmission protocol may be used by nodes that support only fair transmissions (i.e., priority has no influence on these nodes). Low/high-transmission protocols shall be used by nodes that support unfair transmissions (priority influences these nodes). These protocols are interoperable, and provide cost/performance tradeoffs to the implementor. The pass transmission protocol is simpler, but limits the node to a single outstanding nonprioritized transaction, and does not support idle insertion/deletion (sync packets are the only source of elastic symbols). The low/high-transmission protocols are more expensive, but have higher performance and can support other options.

The pass-transmission protocol involves saving nonconsumable idles in the bypass FIFO. The debt-transmission protocol involves discarding idles after merging their critical information into one *saveIdle* symbol.

3.6.5 Pass-transmission protocol

For the pass-transmission protocol, an output buffer is used to hold a packet that is ready for transmission, a bypass FIFO holds portions of packets that arrive during the transmission, an output multiplexer selects between these two symbol sources, and an idleMerge block merges and/or saves bits from received idle symbols. The idleMerge block includes storage to save an idle symbol, *saveIdle*, as illustrated in figure 71.

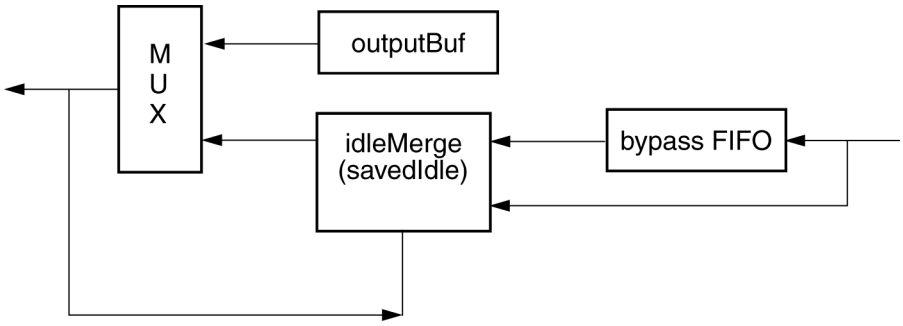


Figure 71 – Pass-transmission model (fair-only node)

When a fair-only node has recovered from its previous transmission, the node's next transmission may begin immediately after it has output an idle symbol having *idle.lg* set. That previous idle is copied into *saveIdle*, for creating an idle symbol that can be postpended to the transmitted packet. The ready-to-transmit condition and the saving of these idle-symbol parameters is illustrated in figure 72.

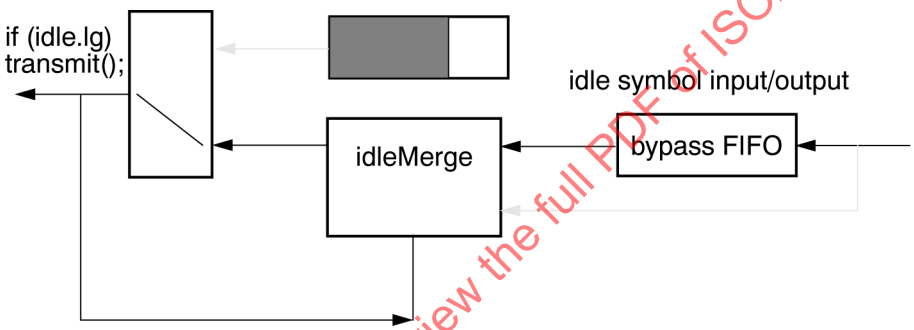


Figure 72 – Pass-transmission enabled

While the packet is being transmitted, arriving input symbols are saved for delayed transmission. A packet symbol or a nonconsumable idle is placed in the bypass FIFO (1), which may increase in stored-symbol content as the packet is being transmitted. A consumable idle (either *idle.lt* is 1 or *idle.ipr* is 0) is not inserted into the bypass FIFO, but its *idle.lg*, *idle.hg*, and *idle.old* bits are merged into the *saveIdle* symbol (2) and the number of symbols in the bypass FIFO remains unchanged, as illustrated in figure 73.

Merging the *idle.lg* and *idle.hg* bits involves ORing them into the *saveIdle* symbol (these bits are selectively set). Merging the *idle.old* bit involves ANDing the bit with the *saveIdle* symbol (this bit is selectively cleared).

Immediately after the packet has been transmitted, the current *saveIdle* symbol is postpended to it. From the perspective of incoming symbols, the *saveIdle* symbol extends the packet-transmission length by one symbol.

After the postpended idle has been sent, there may be one or more symbols in the bypass FIFO. The bypass FIFO is emptied until the node has recovered from its previous transmission. During this time, an incoming packet symbol or a nonconsumable idle is placed in the bypass FIFO (1), leaving the number of symbols in the bypass FIFO unchanged. A consumable idle (either *idle.lt* is 1 or *idle.ipr* is 0) is deleted (2), decreasing the number of symbols in the bypass FIFO. The idle deletion process involves saving the *idle.lg*, *idle.hg*, and *idle.old* bits, as illustrated in figure 74.

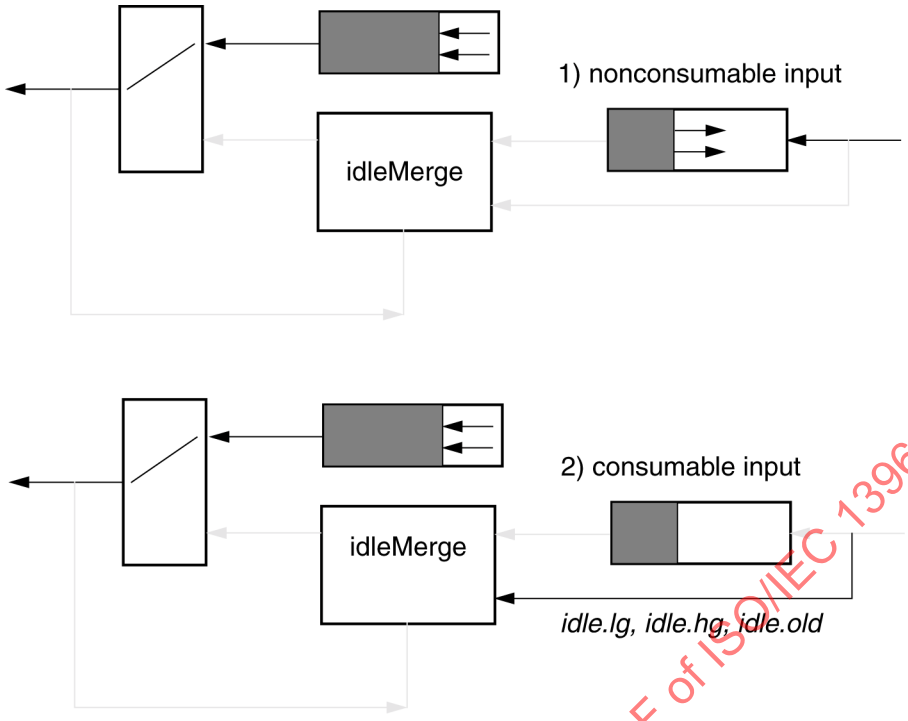


Figure 73 – Pass-transmission active

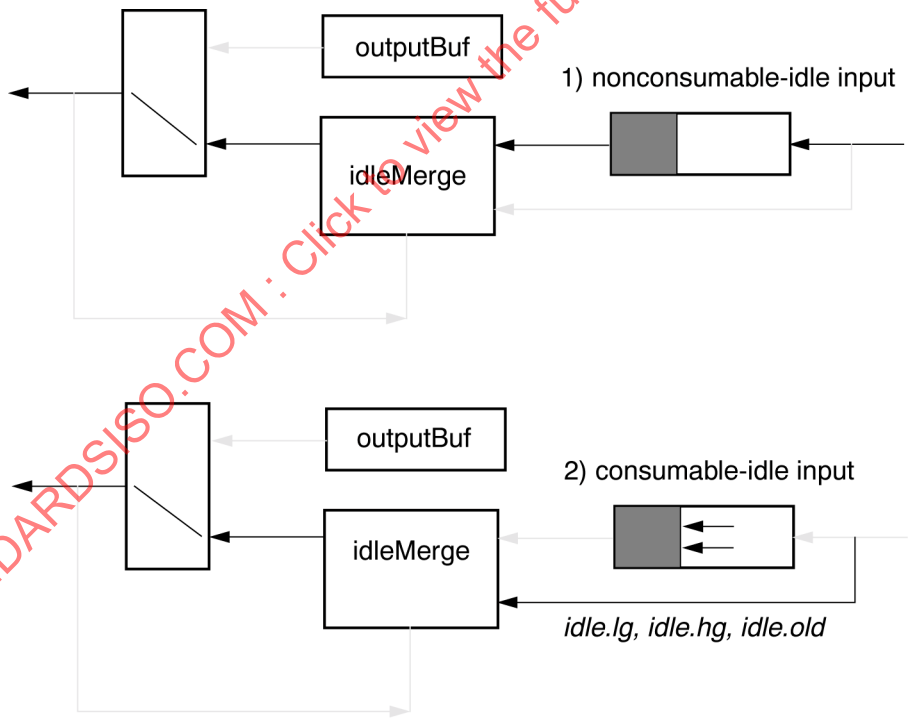


Figure 74 – Pass-transmission recovery

After the bypass FIFO has been emptied, the node is again free to transmit by postpending another packet to an output idle symbol, as previously illustrated in figure 74.

3.6.6 Low-transmission protocol

3.6.6.1 Low-transmission model

The low-transmission protocol is used for the lowest-priority packets and is occasionally used when sending the highest-priority packets (to ensure a small amount of fair bandwidth). A high-transmission protocol is generally used when sending the highest-priority packets. The low-transmission protocol utilizes similar components to the pass-transmission protocol, but they are used slightly differently.

During packet transmission, low- and high-transmissions delete incoming idles to quickly reduce the storage in the bypass FIFO (which reduces the ringlet latency). Thus, there is no need to modify the idles passing through the bypass FIFO, as illustrated in the functional block diagram of figure 75.

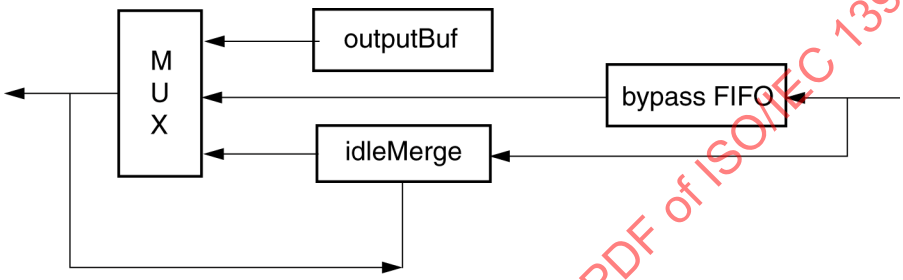


Figure 75 – Low/high-transmission model

When the producer sends its packet, its bypass FIFO may become nonempty because it stores packet symbols received while sending. The producer uses any idles it receives to empty the bypass FIFO quickly, but accumulates a debt when the wrong type of idle is consumed; subsequent producer transmissions are affected by the accumulated debt.

The behaviours of the low/high-transmission and pass-transmission protocols are similar, but the performance of the pass-transmission protocol is worse. The remainder of this clause discusses the low-transmission protocol; the high-transmission protocol is described in 3.6.8.

3.6.6.2 Low-transmission enabled

When a producer has recovered from its previous transmission, the low-transmission protocol allows the producer's next transmission to begin immediately after an idle symbol has been output and *idle.lg* was set. That previous idle is saved for post-pending to the transmitted idle. The ready-to-transmit condition and the saving of these idle-symbol parameters is illustrated in figure 76.

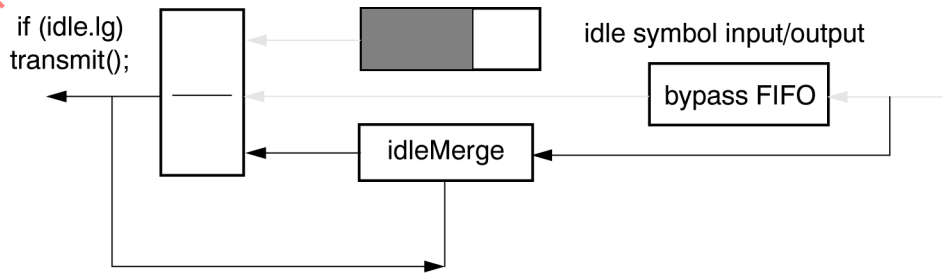


Figure 76 – Low-transmission enabled

3.6.6.3 Low-transmission active

While the packet is being transmitted, input symbols are saved for delayed transmission. A packet symbol is placed in the bypass FIFO (1), which increases the number of symbols saved in it. An idle symbol is not inserted into the bypass FIFO (2), but its *idle.lg*, *idle.hg* and *idle.old* bits are saved in the idleMerge block and the number of symbols in the bypass FIFO remains unchanged, as illustrated in figure 77.

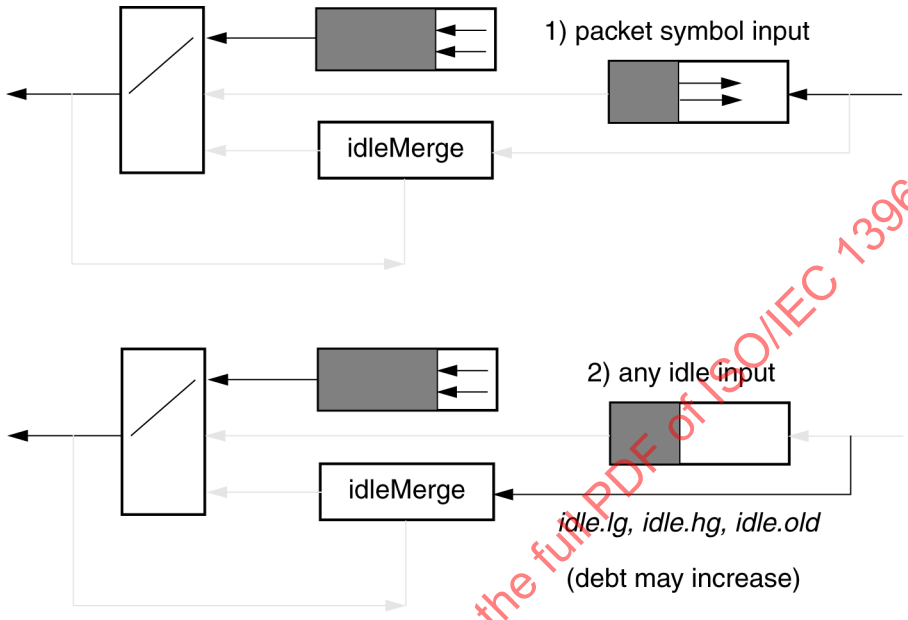


Figure 77 – Low-transmission active

A debt is accumulated that counts how many hi-type idles have been unjustly consumed.

Immediately after a packet has been transmitted, a *saveIdle* symbol is postpended to it. This *saveIdle* value was initialized by the idle symbol preceding the packet transmission.

3.6.6.4 Low-transmission recovery

After the postpend idle has been sent, there may be one or more symbols in the bypass FIFO. The bypass FIFO is emptied until the node has recovered from its previous transmission or another packet has been output. During this time, any incoming packet symbol is placed in the bypass FIFO (1) and the number of symbols in the bypass FIFO remains unchanged. An incoming idle is deleted (2) and the number of symbols in the bypass FIFO decreases. The idle deletion process involves saving the *idle.lg*, *idle.hg*, and *idle.old* bits, as illustrated in figure 78.

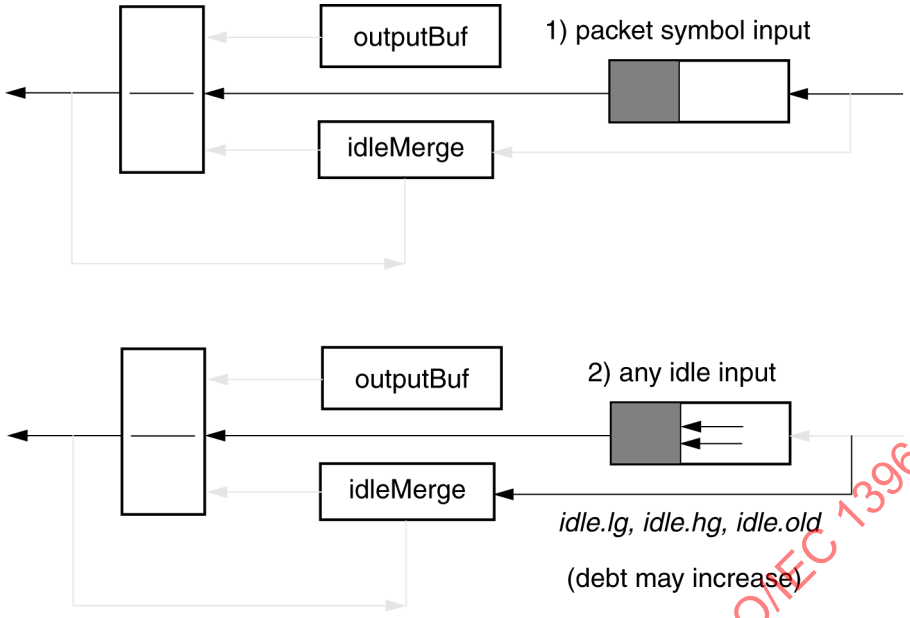


Figure 78 – Low/high-transmission recovery

The debt value is increased when consuming each second through final consecutive idle, if its *idle.It* is 0.

3.6.6.5 Low-transmission debt repayment

After the bypass FIFO has been emptied, accumulated low-consumption debts are reduced or cancelled. During the repayment phase, packet symbols pass (1) through the node unmodified. However, the type of idle symbols may be changed and the consumption debt reduced as idles pass (2) through the node, as illustrated in figure 79.

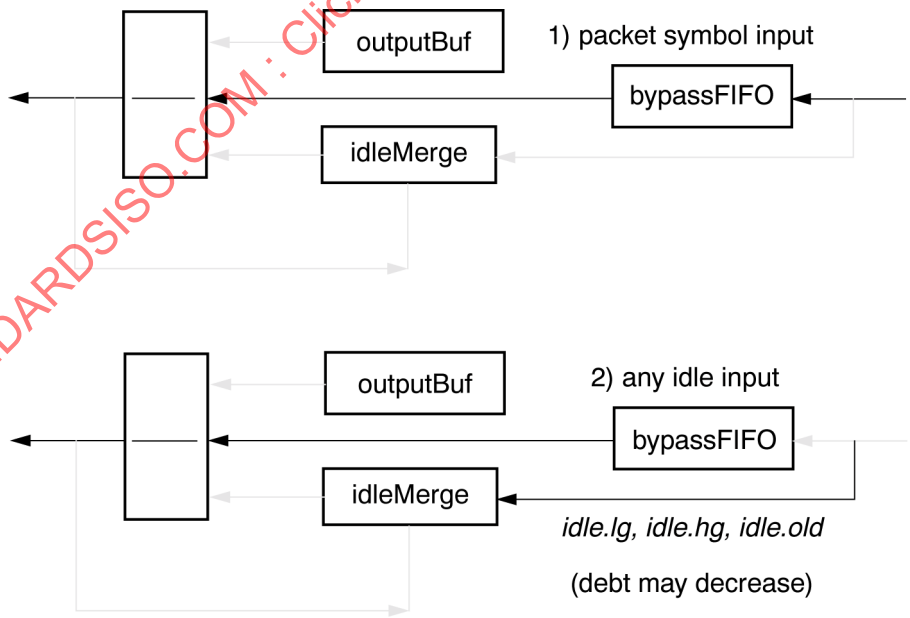


Figure 79 – Low/high-transmission debt repayment

A low-consumption debt is reduced by 1 when a low-type idle is converted into a high-type idle (by setting the previously zero *idle.lt* bit). This low-consumption debt is cancelled when the *idle.ipr* priority is no larger than the node's transmission priority. Further transmissions are disabled until the low-consumption debt has been reduced to zero.

3.6.7 Idle insertions

During the active and debt-repayment phases, idles are re-inserted as back-to-back packets pass through the bypass FIFO. The value of these *saveIdle* symbols may have been affected by the *idle.lg*, *idle.hg*, and *idle.old* bits within idles that were consumed during the packet transmission. When this post-pend occurs, an incoming packet symbol (1) increases the storage in the bypass FIFO, as illustrated in figure 80.

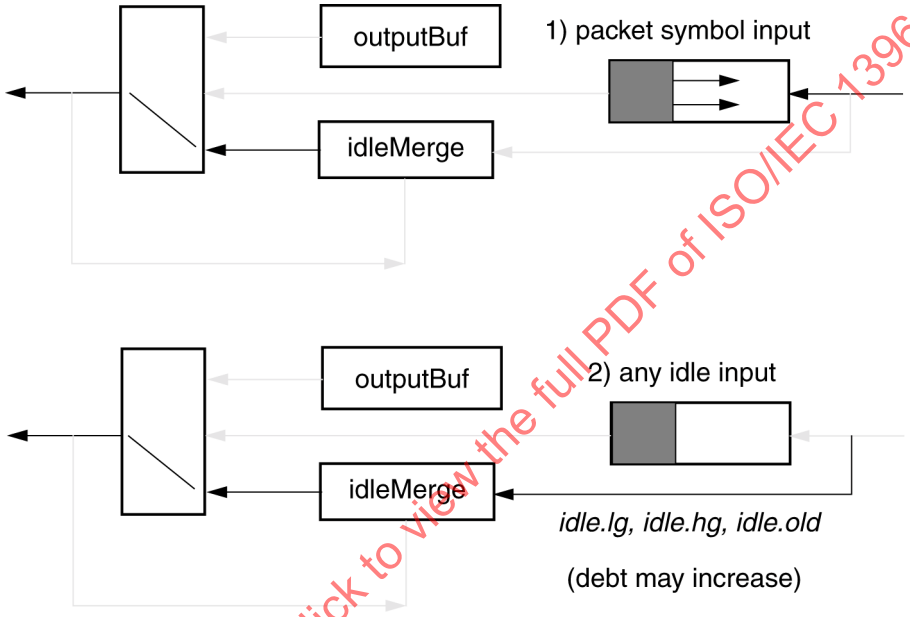


Figure 80 – Low/high-transmission idle insertion

An incoming idle symbol is merged (2) with the previously saved idle, and leaves the bypass FIFO storage depth unchanged, as illustrated in figure 80. However, the processing of this idle may increase the low-consumption debt.

3.6.8 High-transmission protocol

3.6.8.1 High-transmission enabled

The high-transmission protocol uses the same components as the low-transmission protocol, but the components are used slightly differently. When a producer has recovered from its previous transmission, the high-transmission protocol allows the producer's next transmission to begin immediately after an idle symbol has been output and *idle.hg* was set. That previous idle is saved for post-pending to the transmitted packet. The ready-to-transmit condition and the saving of this post-pend symbol are illustrated in figure 81.

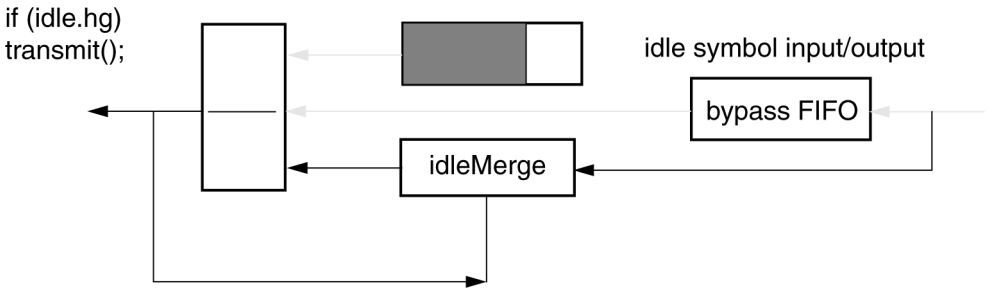


Figure 81 – High-transmission enabled

3.6.8.2 High-transmission active

While the packet is being transmitted, input symbols are saved for delayed transmission. A packet symbol is placed in the bypass FIFO, which increases the number of symbols saved in it. An idle symbol is not inserted into the bypass FIFO, but its *idle.lg*, *idle.hg* and *idle.old* bits are saved in the idleMerge block and the number of symbols in the bypass FIFO remains unchanged, as previously illustrated in figure 77.

A debt is accumulated that counts how many low-type idles have been unjustly consumed.

Immediately after a packet has been transmitted, the *saveIdle* symbol is postpended to it. This *saveIdle* symbol is a copy of the idle that immediately preceded the packet transmission. From the perspective of the input symbols, the transmission of this *saveIdle* symbol is treated like the transmission of the previous send-packet symbols.

3.6.8.3 High-transmission recovery

After the postpend idle has been sent, there may be one or more symbols in the bypass FIFO. The bypass FIFO is emptied until the node has recovered from its previous transmission or another packet has been output. During this time, any incoming packet symbol is placed in the bypass FIFO and the number of symbols in the bypass FIFO remains unchanged. An incoming idle is deleted and the number of symbols in the bypass FIFO decreases. The idle deletion process involves saving the *idle.lg*, *idle.hg*, and *idle.old* bits, as previously illustrated in figure 78.

The debt value is increased when consuming each second through final consecutive idle, if its *idle.lt* is 0.

3.6.8.4 High-transmission debt repayment

After the bypass FIFO has been emptied, accumulated high-consumption debts are evaluated. When this debt has exceeded a maximum threshold, high-transmission protocols are no longer used; a flag is set to ensure that low-transmission protocols are used on the next packet transmission.

3.7 Queue allocation

3.7.1 Queue reservations

When bus transactions are unified and never busied, fair arbitration protocols are sufficient to ensure that all transactions eventually complete. However, when bus transactions are split into request and response subactions, the queues on a shared consumer node may become filled. When this occurs, send packets are echoed with a busy status and must be re-sent until successful. Without a queue reservation mechanism, it would be possible for one producer to be starved, always failing to get queue space, while others successfully compete with its retries. This problem may occur with either request-send or response-send packets, which (from an acceptance-queue perspective) are processed independently.

The queue-reservation protocols, which are a subset of the queue-allocation protocols, ensure that retried send packets are eventually accepted. Input send-packet queues have a state register, whose state (SERVE_NA, SERVE_A, SERVE_NB, and SERVE_B) affects when subactions are accepted and how they are busied. To illustrate how reservations are utilized, consider a consumer node whose input queue is being actively shared by producer nodes requester1 and requester2, as illustrated in figure 82.

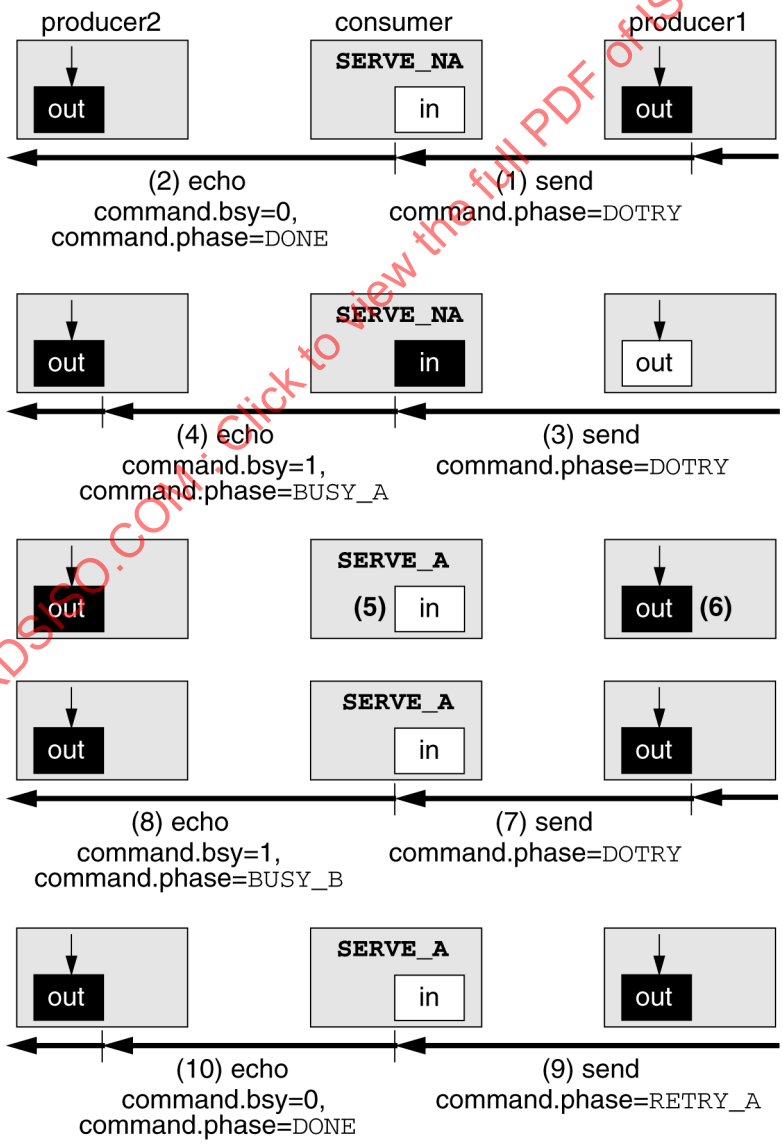


Figure 82 – Consumer send-packet queue reservations

In this illustration, producer1 initially sends a request-send packet (1) to the consumer, and it is accepted. The returned echo packet (2) indicates the send packet was accepted (*command.bsy* is 0) without error (*command.phase* is DONE). This send packet temporarily fills the consumer's input-send-packet queue.

Before the consumer empties its input-send-packet queue, another send packet (3) is sent by producer2. The returned echo packet (3) indicates the send packet was not accepted (*command.bsy* is 1) and should be retried with a RETRY_A command phase (*command.phase* is BUSY_A). The consumer's state is also changed from SERVE_NA (accepting new and RETRY_A commands) to SERVE_A (accepting only RETRY_A commands).

Shortly thereafter, the consumer's input-send-packet queue is emptied (5) and another send packet is generated within producer1 (6). When the new send packet is transmitted (7), the returned echo packet (8) indicates the send packet was not accepted (*command.bsy* is 1) and should be retried with a RETRY_B command phase (*command.phase* is BUSY_B). Although queue space is available, the send packet is not accepted while the queue space is reserved for the previously busied send packet (which has an A label).

Producer2 eventually resends its previously busied send packet, using a RETRY_A command phase. The consumer's state (SERVE_A) allows it to accept this re-sent packet, which ensures forward progress for producer2. When all previously busied RETRY_A requests have been accepted, the consumer's state is changed to SERVE_NB; new or RETRY_B requests (including those from producer1) will be accepted next.

The queue-reservation protocol cycles through the queue states SERVE_NA, SERVE_A, SERVE_NB, and SERVE_B. While in the SERVE_A and SERVE_B states, only RETRY_A and RETRY_B send packets are accepted respectively. This is a simple ageing protocol, where the re-sent packets from the oldest batch are accepted first and the A/B labels are used to identify the relative age of re-sent packets.

At any time, the relative age of a packet is dependent on the reservation state. In the SERVE_A state, the (older) RETRY_A packets are accepted before changing to the SERVE_NB state. In the SERVE_B state, the (older) RETRY_B packets are accepted before moving to the SERVE_NA state.

The queue-reservation algorithm is controlled by the consumer when the subactions are busied, as illustrated in figure 83.

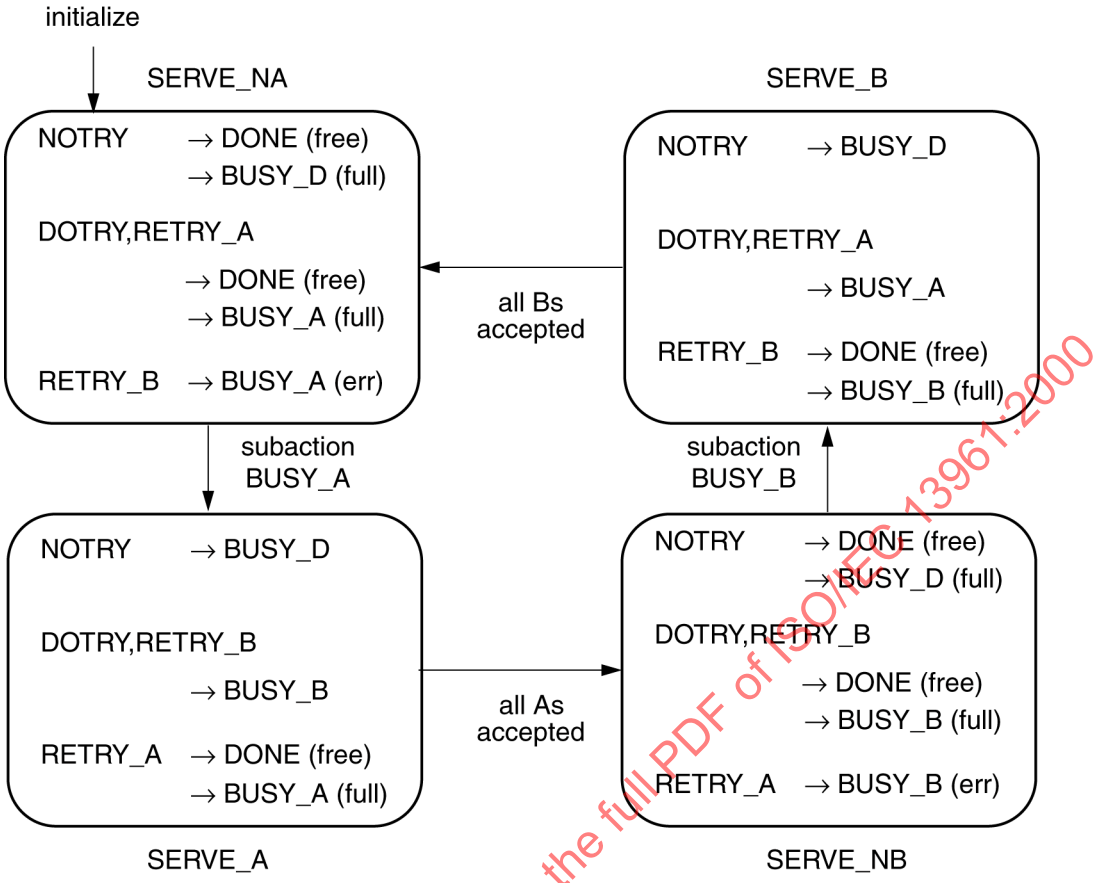


Figure 83 – A/B age labels

In the SERVE_NA state any transaction is accepted into an empty queue. However, as soon as a send packet is rejected (BUSY_A is returned), the queue state changes to SERVE_A and only RETRY_A requests (which are the oldest retries) are accepted.

Eventually all A requests are accepted. This condition can be reliably detected by the absence of another BUSY_A transaction within an allocation opportunity interval (a self-calibrating timeout, see 3.9.2). The delay in switching from SERVE_A to SERVE_NB can be minimized if a counter is used to count the number of busied sends. The (less efficient) timeout is still needed, but is only invoked when the consumer's reservation counter overflows or when one of the producer nodes is reset (which stops its retries).

After all (previously busied) RETRY_A transactions have been accepted, the queue state changes to SERVE_NB. In the SERVE_NB state any transaction is accepted into an empty queue. However, as soon as a packet is rejected (BUSY_B is returned), the queue state changes to SERVE_B and only RETRY_B requests (which are the oldest retries) are accepted.

Separate state machines are required for the request and response queues, which are processed independently to avoid queue-dependency deadlocks.

3.7.2 Multiple active sends

A high-performance producer may transmit more than one send packet before the first echo is returned. Many of these may be active (sent, but no echo returned) with a *command.phase* value of NOTRY. This allows a large number of new (not previously busied) send packets to be sent concurrently and accepted on a first-come first-served basis. However, when NOTRY send packets are busied by a consumer (which is in the SERVE_A or SERVE_B state or its input queue is full), no reservations are made.

Although no immediate reservations are made for a busied NOTRY send packet, this packet is eventually re-sent with a *command.phase* value of DOTRY. If this re-sent DOTRY send packet is then busied, it is assigned a reservation and will eventually be accepted. To avoid abuse of the reservation system (one producer reserving multiple queue entries) each producer shall have at most one DOTRY request-send and one DOTRY response-send packet active concurrently.

This constraint on re-send processing (which forces re-sent packets to be accepted sequentially), ensures that reservations are allocated fairly between contending producers (one reservation per producer) and limits the bandwidth overhead while send packets are being re-sent. Even with this implementation-model constraint, the NOTRY and DOTRY phases guarantee that a large number of sends can be concurrently active and every send will eventually be accepted.

3.7.3 Unfair reservations

The queue-reservation protocols restrict the use of free queue space to ensure fairness among contending nodes. On an unfair-capable node, these queue reservations may be bypassed for prioritized send packets, whose *command.spr* field is greater than zero. The algorithm used to select which prioritized send packets are accepted is beyond the scope of the SCI standard.

To ensure at least a minimal amount of fairness, the queue-reservation protocols shall be bypassed for a limited number of successive send packets. For example, by applying the reservation protocols to every sixteenth send-packet acceptance, every producer is ensured some small fraction of the consumer's packet-processing bandwidth.

3.7.4 Queue-selection protocols

The queue-selection protocols, which are a subset of the queue-allocation protocols, constrain the transmission order for send packets in a producer's output queues. For a fair-only node, the entries within the request-send and response-send queues are processed in FIFO order, with respect to other entries in the same queue. The producer is required to select output entries from the request-send and response-send queues in an alternating order, so that both queues are serviced equally.

For an unfair-capable node, these queue-selection protocols may be bypassed for prioritized send packets, whose *command.spr* field is greater than zero. When the queue-selection protocols are bypassed, which prioritized send packet is selected is based on priority rather than the entry's relative queue position.

To ensure a minimal amount of fairness, the queue-reservation protocols shall not always be bypassed. By applying the selection protocols to the selection of one request and one response packet out of every 16, for example, both output queues would be ensured a minimal fraction of the available producer's packet-transmission bandwidth. The C code illustrates a more efficient implementation.

An output send packet will sometimes have a lower priority than other send packets that are blocked behind it. When this occurs, the output send packet assumes (inherits) the highest priority of the packets that are queued behind it.

3.7.5 Re-send priorities

High-priority producers could easily saturate the ringlet by quickly retrying busied transmissions. To avoid such saturation, the node maintains two transmission priorities, *insertPriority* and *consumePriority*. The *insertPriority* value is used to increase the ringlet priority level; the previously busied packets are ignored when this value is computed. The *consumePriority* value is used to initiate a high-transmission or cancel low-consumption debt; all queued packets are checked when computing this value.

By using these two values, a retrying high-priority node is ensured a fair portion of the available ringlet bandwidth.

3.8 Transaction errors

3.8.1 Requester timeouts (response-expected packets)

For read, write, and lock transactions, the requester uses a response timeout to detect errors that result in the loss of request-send or response-send packets. The requester is expected to calculate a time interval within which a response is expected and when that time limit is exceeded the requester's linc synthesizes a response (or a response equivalent) to report the error to the attached hardware, as illustrated in figure 84.

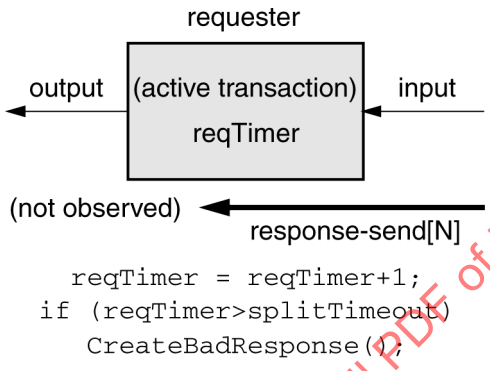


Figure 84 – Response timeouts (request and no response)

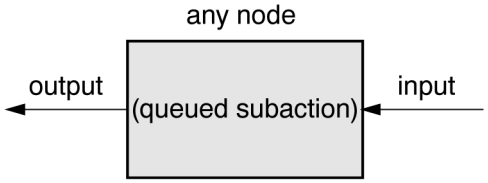
To implement these timeouts, each SCI node shall implement a timer and a SPLIT_TIMEOUT register (which sets the default time-limit for a response) as defined by the CSR Architecture. Vendor-dependent unit architectures may override the default time-limit value; for example, a processor could have a different time-for-response value for each of its virtually mapped memory pages. The response-timeout timers need not be coordinated with timers on other nodes; i.e., a globally consistent time is not needed for its operation.

A response timeout error is generated when either the request-send or response-send packet is damaged or lost. If the request subaction has side effects, the state of the responder is unknown and hardware retry protocols should not be used to automatically retry failed transactions. However, the cache-coherence protocols and the CSR Architecture support software-based fault-retry protocols.

Note that nodes must wait after a ringlet reset until the slowest possible response should have returned from elsewhere in the system before resuming normal operation, to ensure that old responses do not get confused with responses to new requests.

3.8.2 Time-of-death timeout (optional, all nodes)

When a transaction is initiated, a time-of-death value may be specified by setting the *control.todExponent* and *control.todMantissa* values in the request subaction. (These fields are also returned in the corresponding response subaction.) The *control* symbol is checked in send-packet queues and send packets are discarded when their time-of-death interval is reached. This *control*-symbol checking involves a decode step, which converts the compact *control*-symbol field into a normalized 64-bit *deathTime* value, based on the linc's current time-of-day value (*myTime*), as illustrated in figure 85.




```
deathTime = Decode(control,myTime);
if (myTime>deathTime) DiscardSendPacket();
```

Figure 85 – Time-of-death discards

The time-of-death value should be less than the response-timeout value, so that stale sends are safely discarded before the response-timeout occurs. Eliminating stale subaction packets simplifies error recovery protocols, which could otherwise confuse stale packets with recent ones generated during or after the error-recovery process.

The time used for time of death is based on an absolute time-of-day reference. This option therefore requires the implementation of globally synchronized time-of-day clocks in all participating nodes.

Send packets are only discarded from queues, and are not discarded while passing through normal nodes on a ringlet. In a highly pipelined switch the discard decision (which is based on values in the packet's *control* symbol) may be made while the packet is being transmitted; these packet transmissions shall be nullified by stomping the CRC, which will soon result in their being discarded.

To minimize the number of bits used within each send packet, the time-of-death protocols are based on life-intervals. A packet is born during life-interval *N* and is discarded by the interconnect during life-interval *N*+2. The requester's timeout is during the following interval *N*+3, as illustrated in figure 86.

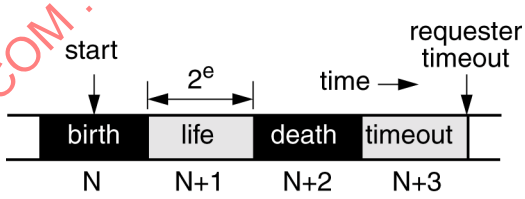


Figure 86 – Packet life-cycle intervals

The time-of-death value is condensed to an efficient floating-point format (5 bits of exponent and 2 bits of mantissa) that is only included in send packets. The *control.todExponent* specifies the length of the life-cycle interval (in powers of two) and the *control.todMantissa* specifies the date of death (one of four values). A zero value of *control.todExponent* (which is the default) inhibits any time-of-death-based discarding.

This 7-bit value and the node's 64-bit global time-of-day register (which is synchronized with the other global clocks) are used to generate a normalized 64-bit time-of-death value when a packet is enqueued, as illustrated in figure 87. This time-of-death value is checked when the packet is dequeued, and the stale packets are discarded. The time-of-death value need not be checked before the send-packet transmission starts, but has to be checked in time to stomp the CRC value.

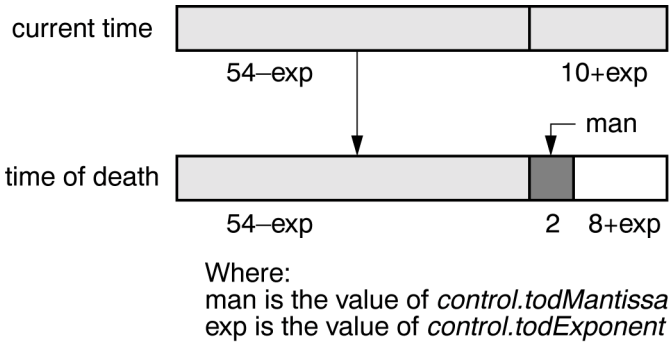
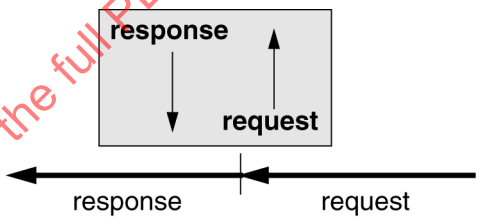


Figure 87 – Time-of-death generation model

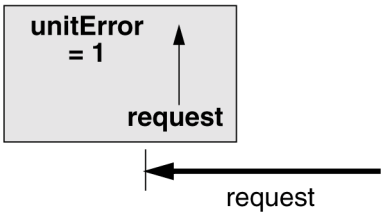
3.8.3 Responder-processing errors

Errors may be detected by the responder when its queued request packets are processed. When a response-expected request is processed, the responder processing status is returned as an error-status code within the returned response packet. When a responseless request is processed, the error status shall not be returned, but should be logged in a unit-dependent fashion (if the request's *addressOffset* corresponds to an implemented unit) or should be logged in the node by setting the errorLog bit (when no unit responds to the specified *addressOffset* value), as illustrated in figure 88.

```
if (!Responseless(command.cmd))
  ConvertRequestToBadResponse();
```




```
if (Responseless(command.cmd) &&
IsUnitAddress(addressOffset))
  UnitDependentErrorLog();
```




```
if (Responseless(command.cmd) &&
!IsUnitAddress(addressOffset))
  errorLog = 1;
```

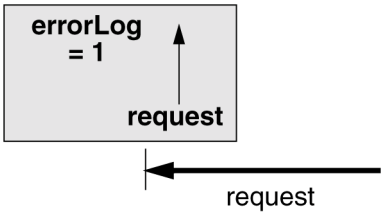


Figure 88 – Responder's address-error processing

A request packet with an invalid CRC value, an invalid length, or excessive length (which is not supported by the node) shall be discarded. The node's ERROR_COUNT register should be incremented (which shall have the side-effect of setting the node's STATE_CLEAR.*elog* bit). A unit-dependent error shall not be logged under these circumstances, since the integrity of the *addressOffset* field cannot be verified.

The excessive-length packets may always be discarded; this includes excessive-length request packets, for which a response packet is never returned. This dramatically simplifies consumer node hardware, which only needs to respond to packets that can be queued. Two errors are generated by such packet discards: The consumer increments its ERROR_COUNT when the packet is discarded, and the producer increments its ERROR_COUNT after an echo timeout (no echo is generated when the packet is discarded).

Note that there is no immediate way of distinguishing between the lack-of-response errors created by transmission errors and packet-size discards. In both cases the packet is lost and the loss is ultimately detected by the requester's response-timeout hardware. However, reading the CSRs that identify the node's capabilities can quickly identify packet-size-related errors.

A response-expected packet with an unsupported command or length value shall cause the return of a response packet with a RESP_TYPE status code. If the *addressOffset* value neither corresponds to a node CSR or a unit architecture on the node, a response with an RESP_ADDRESS status code is returned. If the *addressOffset* value corresponds to a unit architecture on the node, but the accepted request packet is rejected due to a transient queue-use conflict, a response with a RESP_CONFLICT status code is returned.

For a responseless packet, these errors are logged and no response packet is generated. Note that this includes move transactions as well as unexpected response-send packets (which cannot be associated with an outstanding request transaction). On a responder-only node, all responses are unexpected; these responses shall be accepted (a nodeId addressing error is not generated), but shall be discarded and should increment the node's ERROR_COUNT register when the response packet is processed.

3.9 Transmission errors

3.9.1 Error isolation

3.9.1.1 Error containment

To help locate the source of errors, nodes are responsible for detecting errors at their input, updating an error-count register when the error is detected, and "correcting" the error before it propagates to the node's output. ("Correcting" in this context does not repair the bad data, but changes the CRC so that the packet no longer causes errors to be detected.) Since the error condition is corrected on the node's output link, the error is logged at only one location, where the error was first detected. By reading the node's ERROR_COUNT register the link that generates the errors can be readily identified. The different forms of error correction are illustrated in figure 89.

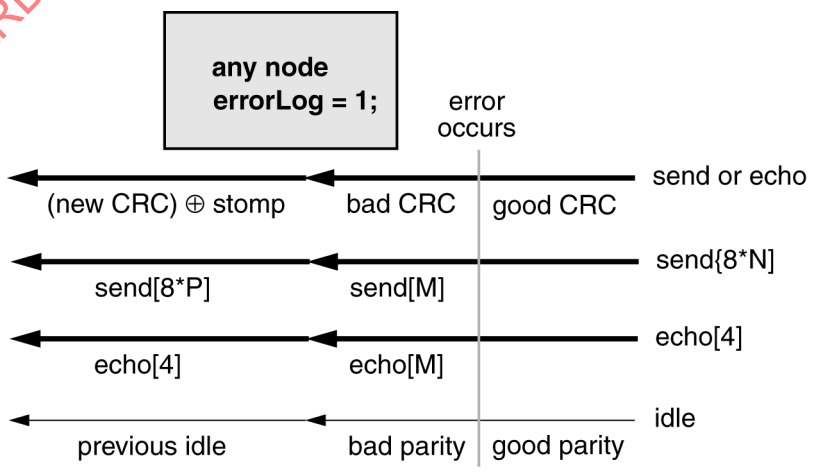


Figure 89 – Response timeouts (request and no response)

For correctly sized packets the CRC value is checked when the packet passes through intermediate nodes. If the CRC value is incorrect, a stomped value is substituted for the incorrect value. The stomped value is the expected CRC value (new_CRC) exclusive-ORed with a constant 16-bit STOMP value. An error is only logged when the incoming CRC value is incorrect and is different from the stomped CRC value. Only a few multiple-bit errors (about 1 in 216 error-burst sequences generate the "stomped" CRC value) will not be logged using this error-correction strategy.

Send packets are constrained to be multiples of 8 symbols in length and echo packets are constrained to be 4 symbols in length. If an error corrupts the flag signal, which marks the start and end of packets, the observed length of a send or echo packet may differ from one of these legal packet lengths. When passing through a node such packets are extended or truncated to a legal packet length and an error is logged.

Idle symbols are protected by parity, which is checked when the idle symbol is processed. When the parity is incorrect, the idle symbol is discarded and an error is logged. The previous idle symbol with good parity is substituted for a discarded idle symbol.

3.9.1.2 Error logging

An error condition may be detected during each symbol period. Errors are counted using the ERROR_COUNT register, but this register is only updated once every 64 symbol periods. The infrequent counter updates simplify the implementation of the ERROR_COUNT register (which changes at a slower clock rate), while providing a reasonable estimate of the number of errors.

To implement this slower counter, a detected error condition immediately sets an errorLog bit. Every 64 symbols the *errorLog* bit is checked and cleared. When the errorLog bit is one, the ERROR_COUNT register is incremented and an error status bit is set in the node's error-status summary register (STATE_CLEAR.*elog*), as illustrated in figure 90.

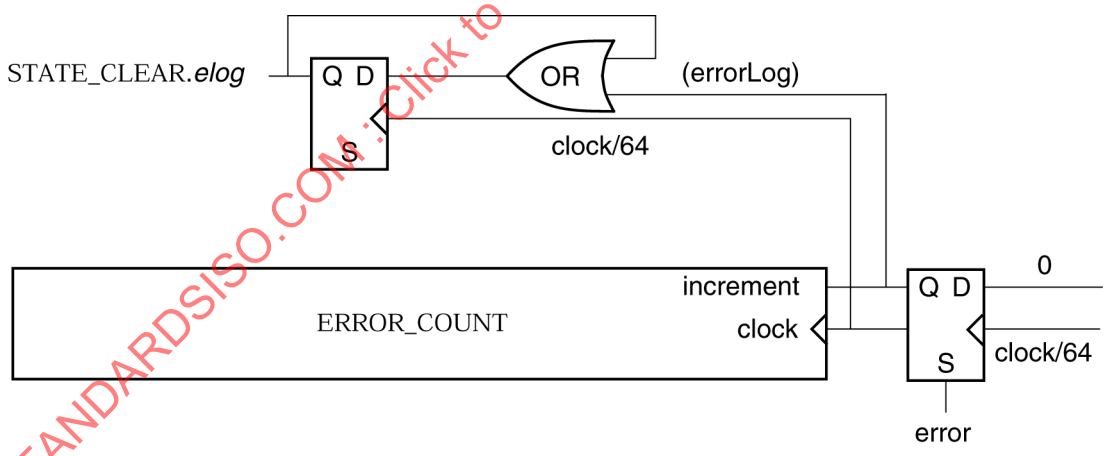


Figure 90 – Error-logging registers

The ERROR_COUNT register and bits within the STATE_CLEAR register may also be modified by CSR write transactions; these external-access update capabilities are not illustrated.

The ERROR_COUNT register should be implemented and the STATE_CLEAR.*elog* bit shall be implemented. Vendors may have additional vendor-dependent error logging registers to identify the error type and parameters (such as the packet's address). However, the definition of these registers is beyond the scope of the SCI standard.

3.9.2 Scrubber maintenance

3.9.2.1 Recoverable scrubber errors

A minimal ringlet has one requester, one responder, and one scrubber (although all three may be the same node). The scrubber is responsible for deleting damaged packets, returning nodeId addressing errors, and maintaining the self-calibrating ringlet timeout counters. Although many nodes can have scrubber capabilities, only one scrubber is activated on each ringlet; the ringlet initialization process assigns the scrubber node.

Although an implementation will generally combine scrubber functions with the normal node-function hardware, a node should act as though its scrubber functions are independent of other functions. Thus, the scrubber is logically a separate node but may be preceded or followed (internally) by a node that transmits and receives packets.

The scrubber is selected by a voting protocol during the ringlet initialization process. After the ringlet has been initialized, the scrubber is responsible for performing a variety of ringlet-maintenance functions, as illustrated in figure 91.

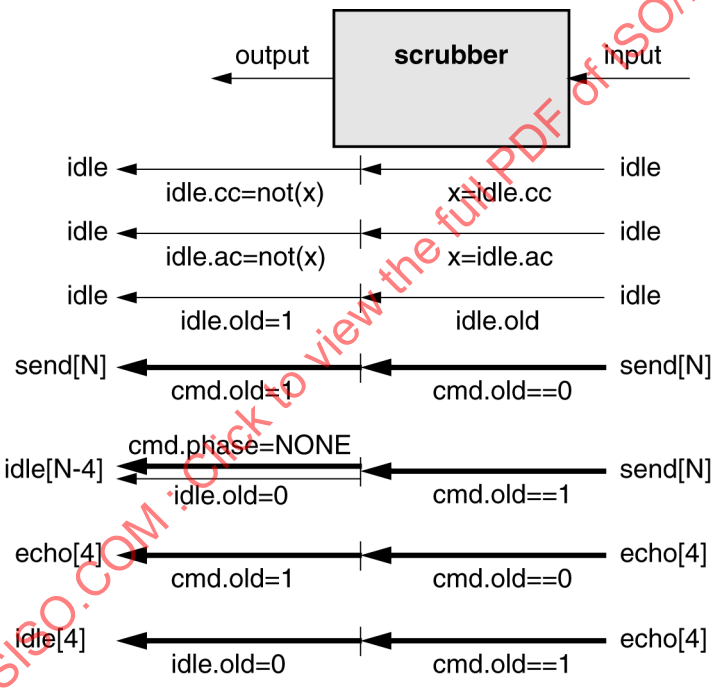


Figure 91 – Scrubber maintenance functions

When an idle passes through the scrubber, the values of its counters, *idle.cc* and *idle.ac*, are complemented. The ringlet effectively forms a variable-length ring counter, where the period of the counter depends on the delays of the *idle.cc* or *idle.ac* bits when passing through the other nodes. These counters are used for various timeouts that protect against waiting forever (deadlocking) after certain failures. For example, *idle.cc* is a ring circulation count, used to detect missing echoes (or dropped packets). The *idle.ac* counter keeps track of allocation opportunities, and is used in the busy-retry mechanism.

When an idle passes through the scrubber, its age bit *idle.old* is set to 1. Other nodes clear the *idle.old* bits to 0 while their FIFO is being emptied or their idle-consumption debts are decreasing. For certain timeouts, the absence of *idle.old* bits at the scrubber's input is interpreted as a sign of ringlet activity.

The scrubber's processing of send packets is influenced by the age bit in the command symbol, *command.old*. If *command.old* is 0, its value is set to 1 (the packet's age is increased). If *command.old* is 1, the send packet is stripped and replaced with idles and an echo. An error status is provided by the echo's command symbol: *command.phase* shall be set to NONE and the *command.mpr*, *command.spr*, and *command.old* fields shall be set to 0.

The scrubber's processing of echo packets is influenced by the age bit in the command symbol, *command.old*. If *command.old* is 0, its value is set to 1 (the packet's age is increased). If *command.old* is 1, the echo packet is discarded and replaced with idles.

3.9.2.2 Unrecoverable scrubber errors

Transmission errors may result in the loss or corruption of idle symbols. If all of the *idle.lg* (low-go) bits are lost, the ringlet activity will cease (permission to transmit is based on these go bits). A scrubber that detects this condition may optionally *clear* the ringlet (to discard corrupted idles and distributed allocation state) and inject new *idle.lg* and *idle.hg* bits when restarting the ringlet activity. A circulation-count-based timeout, a two-bit *lgTimer* counter, is used to detect this condition, as illustrated in figure 92.

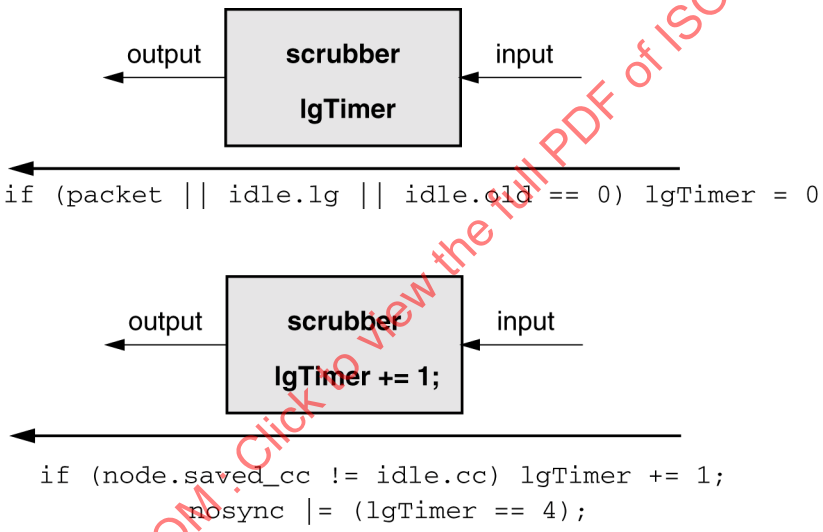


Figure 92 – Detecting lost low-go bits

The counter is cleared when a packet passes through, a low-go bit passes through (*idle.lg*==1), or other ringlet activity is sensed (*idle.old*==0). The counter is incremented every ringlet-circulation time, i.e., whenever *idle.cc* changes. The error condition is detected by an overflow of the *lgTimer* (counting past the value of 3).

Special timers are not needed to detect when *idle.hg* (high-go) bits are lost, since these bits are only used by unfair nodes, and the *idle.lg* bit regenerates an *idle.hg* bit when passing through unfair nodes.

3.9.3 Producer-detected errors

3.9.3.1 Ringlet-local address errors

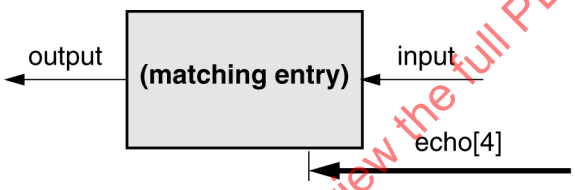
SCI protocols are optimized for directed transaction transmissions to existing node addresses. During system initialization, and after certain privileged-software (operating-system kernel) or hardware errors, transactions might be directed to nonexistent addresses.

If the nodeId portion of the address is correct, but the address offset is not implemented, then the responder status field *status.sStat* in the returned response packet informs the requester of the error. Packet transmission protocols are not affected by these address-offset errors; with the exception of the status code in the response-send packet and the lack of requested data in the response-send packet, these are normal transactions.

If the nodeId portion of the address corresponds to a nonexistent node address, the packet may be accepted by switches or bridges and forwarded to another ringlet, but at some point it will not be forwarded further and will circulate around that ringlet and be marked old by setting the *command.old* bit as the packet passes the ringlet scrubber. When this old packet returns to the scrubber, it is converted into an echo packet (with the NONE error status), as previously illustrated in figure 91.

Thus, the ringlet scrubber is a pseudo-responder for nonexistent nodeId addresses. Note that these addressing errors are quickly detected, since the delays in setting the *command.old* bit add only a small amount of latency compared to waiting for a response timeout.

When an echo with a NONE status is returned, the producer converts a response-expected request to a response packet that returns an error status to the source. Thus, addressing-error status is returned to the requester in two forms: a locally detected address error returns error status in an echo and a remotely detected address error returns error status in a response, as shown in figure 93.



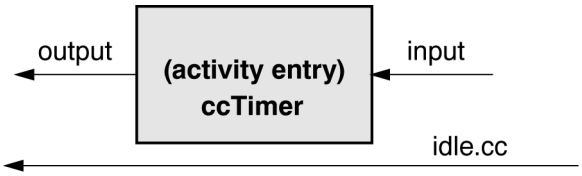

```
if (command.bsy == 0 && command.phase == NONE)
 if (Responseless(command)
  DiscardSendPacket();
else
 ConvertRequestToBadResponse();
```

Figure 93 – Producer's address-error processing

When an echo with a NONE status is returned, the producer discards responseless (move and event) requests.

3.9.3.2 Echo timeouts

A producer retains a send packet until the corresponding echo packet is returned. If the echo is damaged or lost, a timeout mechanism is needed to discard the producer's send entry, as illustrated in figure 94. This rapid timeout quickly releases expensive queue resources for use by other transactions.




```
if (node.saveCc != idle.cc) ccTimer += 1
if (ccTimer == 4) DiscardSend();
```

Figure 94 – Producer's echo-timeout processing

The echo timeout period is measured as four transitions of the ringlet-circulation count (the *idle.cc* bit in the idle symbol, see 3.2.11). This self-calibrating circulation count is maintained by the scrubber and propagated around the ringlet in idle symbols.

Except for optional error logging, the echo timeout only has the effect of discarding send packets. Thus, the higher-level response timeout (described previously in this clause) is still needed to detect the loss of corrupted send packets. A transaction may still complete normally after an echo timeout has occurred, if the send packet was transferred successfully but the returned echo was corrupted.

3.9.3.3 Fatal ringlet-state errors

An unexpected echo, which is returned with a good CRC and has no matching active send packet, is one form of ringlet-state error (that could be generated after a circulation-count failure). When attempting to transmit (and therefore preventing its output *idle.ac* value from changing), a node's input allocation-count (*idle.ac*) should change at most once; an additional change is a ringlet-state error (an allocation-count failure). A producer may optionally detect these ringlet-state errors and initiate an initialization sequence to clear the ringlet's interfaces, as illustrated in figure 95.

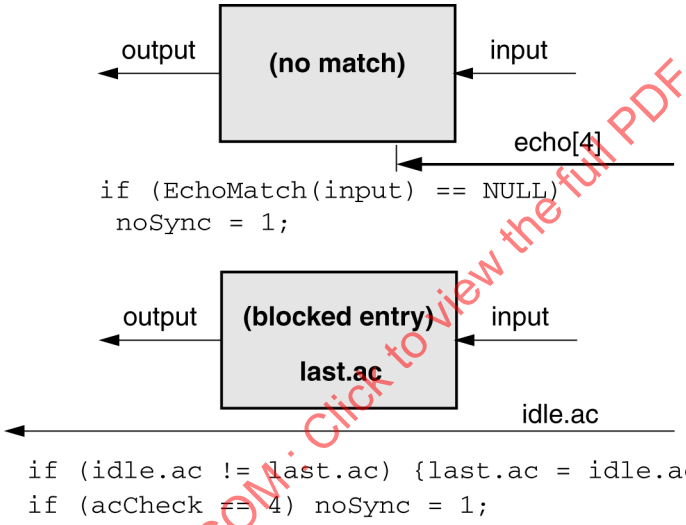


Figure 95 – Producer fatal-error recovery (optional)

3.9.4 Consumer-detected errors

When many nodes send packets to a congested consumer, they are stripped but retry echoes are returned to the producers. To ensure forward progress the consumer reserves its queue entries for the (older) set of the busied packets. When the source is reset or a re-sent packet is lost, these queue-entry reservations need to be cancelled quickly.

When space is reserved for it, a packet shall be re-sent within the next allocation-count interval, to ensure forward progress. Transactions not retried within four allocation-count intervals shall have their reservations cancelled.

The discard of a consumer's reservation is based on updating a reservation-confirmation timeout value (*acTimer*) while monitoring the allocation-count value. The *acTimer* value is cleared whenever a reservation is used, or when a reservation is confirmed (the send is resent, but is once again busied). If four arbitration counts are observed and no reservations are used or confirmed, the older reservations are cancelled, as illustrated in figure 96. This figure illustrates how the timeout applies to the RETRY_A state; a similar timeout is applied when in the RETRY_B state.

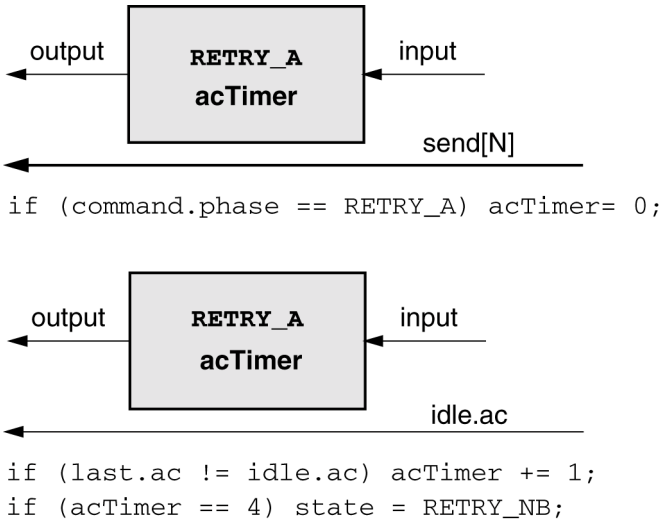


Figure 96 – Consumer error recovery

Every node shall use this reservation timeout to determine when its reservations are discarded. A higher-performance node may optionally maintain reservation counts, which are incremented when a reservation is made and decremented when a reservation is utilized. With a reservation count, the consumer reservation timeout need only be utilized when the reservation count overflows or when resets or errors cancel an expected send-packet retransmission.

Note that a reservation timeout when the counter did not overflow is an error that should be logged by incrementing the ERROR_COUNT register.

3.10 Address initialization

3.10.1 Transaction addressing

SCI uses the 64-bit-fixed addressing model defined by the CSR Architecture. The 64-bit address space is divided into subspaces, one for each of 64 K equal-sized nodes, as illustrated in figure 97. The highest 16 subspaces are reserved for special uses. The highest eight are used during system initialization.

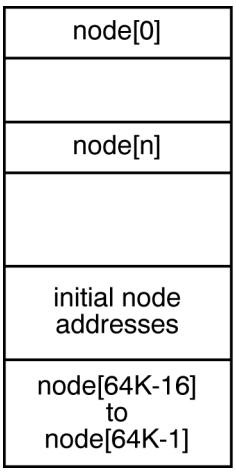


Figure 97 – SCI (64-bit fixed) addressing

The scrubber assigns sequential addresses to each node in its ringlet and then starts the ringlet operating normally. At this stage, each ringlet's nodes have the same set of sequential addresses. Software then initializes and starts the agents that connect the various ringlets and finally assigns a new address to each node so that node addresses will be unique in the system. These software address-assignment algorithms are beyond the scope of the SCI standard.

Defined nodeId addresses are used as the target address to label special-send (initialization-related) packets, as listed in table 13.

Table 13 – Defined SCI nodeId addresses

nodeId	name	description
FFFF16	SYNC	special synchronization format, all nodes strip on receipt
FFFE	CLEARH	clear packet, from fixed scrubber
FFFD	RESETH1	reset packet, from fixed scrubber, phase==1
FFFC	RESETH0	reset packet, from fixed scrubber, phase==0
FFFB	STOP	first symbol of stop and abort packets
FFFA	CLEARL	clear packet, from candidate scrubber or other
FFF9	RESETL1	reset packet, from scrubber or other, phase==1
FFF8	RESETL0	reset packet, from scrubber or other, phase==0
FFF7-FFF0	specialId	reserved nodeId addresses
FFEF	SCRUB_ID	scrubber's initial nodeId address
FFEElower	(other Ids)	initial nodeId addresses assigned by scrubber to others

The specialId values are reserved for future definition, and should not be assigned by system-configuration software. Note that the address decoders need not detect these specialId addresses, since they always pass through nodes that have their nodeIds properly assigned.

The sync packet (whose initial symbol is SYNC) and the abort packet (whose initial symbol is STOP) are both special, in that the final packet symbol (which would otherwise have been a CRC) is zero and the flag bit has a unique pattern. For the sync packet, the first symbol has *flag=1* and the final seven symbols have *flag=0*. For the abort packet, the first six symbols have *flag=1* and the final two symbols have *flag=0*. The abort packet is used to terminate packet transmissions unambiguously during link shutdown. It is always followed immediately by a sync packet.

The STOP symbol is also the first symbol of a stop packet, which has a CRC and the same flag-coding as the other send packets (four symbols *flag=1* and four symbols *flag=0*). The stop packet is used to force downstream nodes into the dead state (see figure 102).

A node design may allow one component (such as the processor) to access another node-local component (such as a memory) using node-local transport protocols. On such nodes, the SYNC address should be used to address other node-local components when the node's nodeId value may be unknown or changing. In normal operation, software is not expected to use the SYNC address, since this address cannot be shared by other nodes.

3.10.2 Reset types

There are several types of reset. A **power_reset** (which is triggered by a real or apparent loss of power for more than one second) discards all volatile node state. If random uniqueId values are used during ringlet initialization, a new uniqueId value is also generated. Note that one node's power_reset quickly initiates a linc_reset of other nodes on the same ringlet.

A **warm_reset** (which is triggered by a loss of power for less than one second) resets most of the linc and the node, but leaves the uniqueId and phase-bit values (which are used to select the scrubber) unchanged. A warm_reset is optional; if not implemented, the node's apparent power-loss shall always be greater than one second (i.e., a short power-recovery is not possible).

A **command_reset** (which is triggered by a write to the node's RESET_START register) has no effect on the queued packets or the control bits that affect their routing. However, the reset affects the node state, as defined by the CSR Architecture. From a software perspective, the command_reset differs from a power_reset or warm_reset in that 1) the reset only affects the addressed node and 2) the contents of the NODE_IDS and ERROR_COUNT registers are not changed.

A **linc_reset** (which is triggered by a vendor-dependent signal) is used to clear the link-interface-circuit (linc) queues, arbitration logic, arbitration counts, etc. From a software perspective, the linc_reset differs from a power_reset or warm_reset in that the contents of the ERROR_COUNT registers are not changed.

A **linc_clear** (which is triggered by a vendor-dependent signal) is used to clear the linc queues, arbitration logic, arbitration counts, etc. A linc_clear has no direct affect on register state. However, the clearing of the linc queue state may force the discard of previously queued packets, which may indirectly affect local and remote error-logging registers. A linc_clear is optional; if not implemented, a node shall enter the dead state when a clear initialization packet is observed.

These forms of reset are illustrated in figure 98.

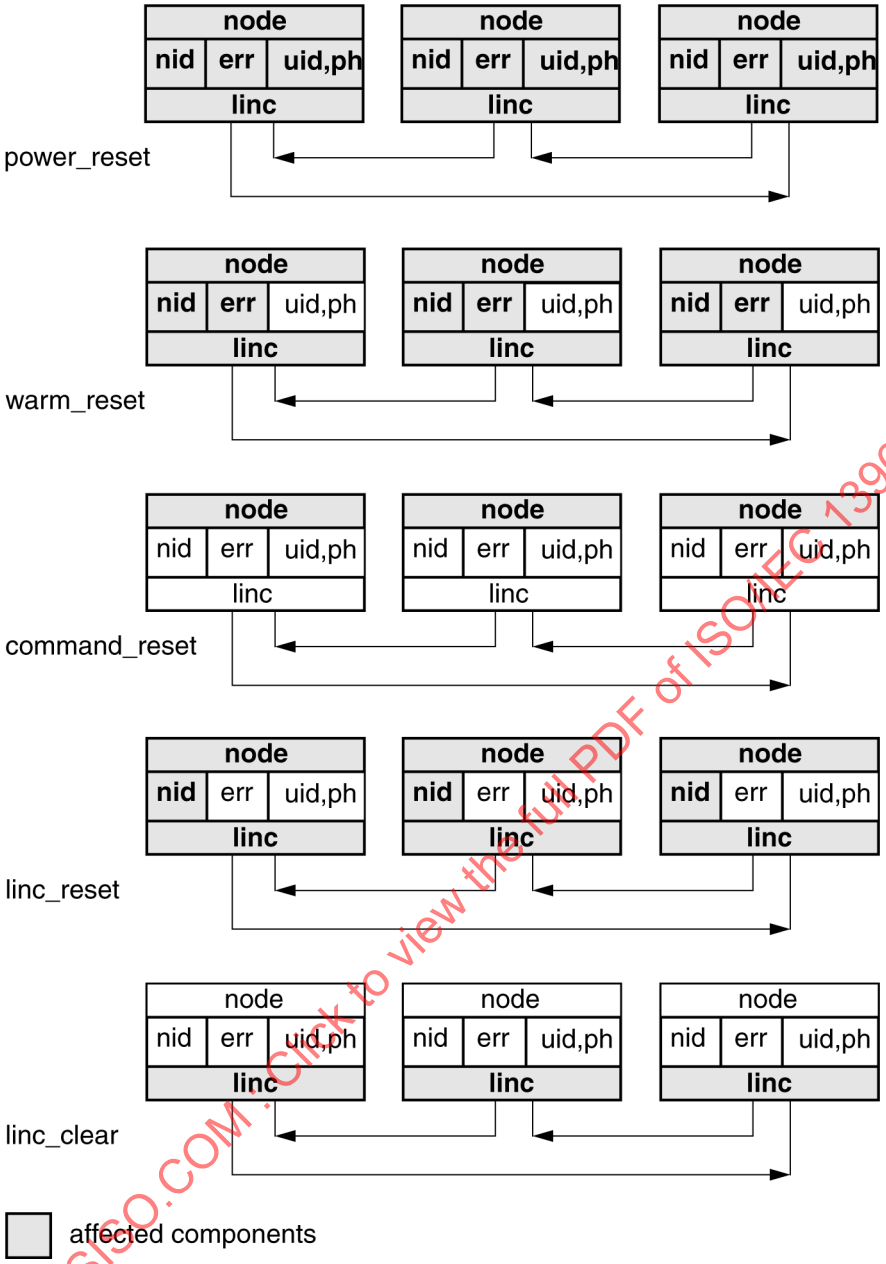


Figure 98 – Forms of node resets

3.10.3 Unique node identifiers

A working system needs each node to have a unique address and each ringlet to have exactly one node serving as scrubber. The initialization process SCI uses to meet these requirements needs a unique identifier for each node. This identifier is not used as the node's address; address assignment is a software responsibility that can be carried out once SCI has started up and made the nodes accessible.

Soon after the SCI links have started operating, and before normal transactions can begin, the ringlet scrubber node is selected. After initialization is complete the scrubber performs a variety of housekeeping functions: it maintains two self-calibrating ringlet-local time counters, detects and reports nodeId-addressing errors, and deletes damaged packets. Several of these functions depend on having only one scrubber per ringlet. For example, the scrubber sets a

command.old bit in each packet it sees go by, and if the *command.old* bit is set in an incoming packet, that packet is discarded (because it circulated more than once around the ringlet). If there were two scrubbers, every packet would be discarded by the second scrubber it encountered. However, if a second scrubber is somehow erroneously enabled, timeouts will cause the ringlet to be re-initialized (which will eliminate the problem).

On some backplane-based interconnects (not SCI) the assignment of unique identifiers is based on specialized backplane wiring, i.e., geographical addressing. Basing the scrubber-selection process on backplane wiring would complicate the design of serial versions of SCI, and would introduce the possibility of failure when the system is improperly configured (e.g., the special scrubber slot is not occupied).

Scrubber selection protocols are therefore based on 80-bit unique identifiers (called UIDs) contained on each node. During the initialization process, the node with the largest UID value is selected to be the scrubber. The most-significant part of UID (stableId) may be provided by 16 backplane signals or 16 bits of nonvolatile memory and the least-significant part of UID is provided by a 64-bit random number (randomId) or by a fixed number known to be unique.

When the most-significant 16 bits of the UID are uniquely assigned, the scrubber selection process is deterministic and the highest UID identifies the preferred scrubber node. When the most-significant 16 bits of the UIDs of two or more nodes are inadvertently assigned the same value (and this is the largest value on the ringlet) the less-significant portion of the UID ensures that the UIDs will (almost always) be unique.

3.10.4 Ringlet initialization

The SCI initialization protocols uniquely identify the scrubber (which has unique cleanup responsibilities on each ringlet) and assign initial nodeId values to all nodes on the ringlet. Initial (ringlet-unique) nodeId values are assigned during the scrubber-selection process, based on the distance of each node from the scrubber. The robust scrubber selection process avoids the use of specialized backplane wires or manual selection switches, since manual selection mechanisms are susceptible to human-induced configuration errors.

Each SCI ringlet is initialized independently, which will result in the same sequence of nodeId values being assigned to each of the various ringlets. Higher-level software that configures the bridge or switch address-acceptance lists is responsible for changing the initial nodeIds to nonconflicting values.

Initialization on each ringlet involves reset generation, input checking, nodeId checking, and startup steps (some of which are performed concurrently), as described in 3.10.4.1 to 3.10.4.4.

3.10.4.1 Reset generation

Each node generates a stream of training (sync) packets to synchronize the receiver of its downstream neighbour. The training packets are interleaved with *reset* packets. Nodes initially output reset packets with their own unique identifier (UID) values and a *distanceId* value of `SCRUB_ID`.

3.10.4.2 Input checking

Although all input packets are stripped during the initialization process, the nodes monitor incoming initialization packets and output the maximum of their UID value and the last UID value received, as illustrated in figure 99.

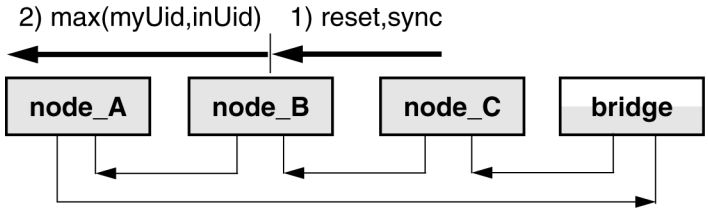


Figure 99 – Receiver synchronization and scrubber selection

Resets with a lower UID value than the node's own UID value are ignored. Observing a reset with a higher UID value than the node's own UID value removes the node from the scrubber-selection process, and the node forwards the higher UID value to the resets it sends in its output packet stream.

In most cases (all except perhaps those involving multiple rapid independent power up/down transitions at various nodes) ringlet closure is ensured when a node observes an input *reset* packet with its own UID value. That node then becomes the scrubber, and outputs idle symbols (with their *idle.lg* and *idle.hg* bits cleared), to flush reset packets from the ringlet, as illustrated in figure 100.

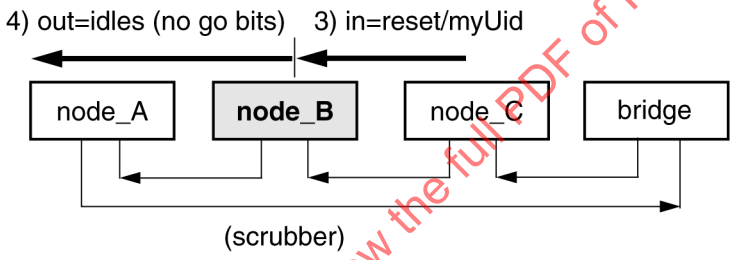


Figure 100 – Reset-closure generates idle symbols

3.10.4.3 NodeId assignment

The distanceId value in each init packet is decremented by each nonscrubber node, then saved as that node's (potential) nodeId value, as shown in figure 101. The decremented distanceId is sent on to the next node as the reset packet is passed on. The last distanceId value received comes from a reset generated by the scrubber (with nodeId `SCRUB_ID`), so it correctly sets all nonscrubber initial nodeId values. A distanceId value of zero indicates an error has occurred (perhaps a large UID was erroneously generated and circulated, blocking every node from becoming the scrubber) and the initialization sequence is restarted.

3.10.4.4 Startup

After receiving the first of its own reset packets, the scrubber outputs idle symbols. After an idle symbol is received, the scrubber changes to a running state and injects *idle.lg* and *idle.hg* bits into the idle symbols that it generates, as illustrated in figure 101.

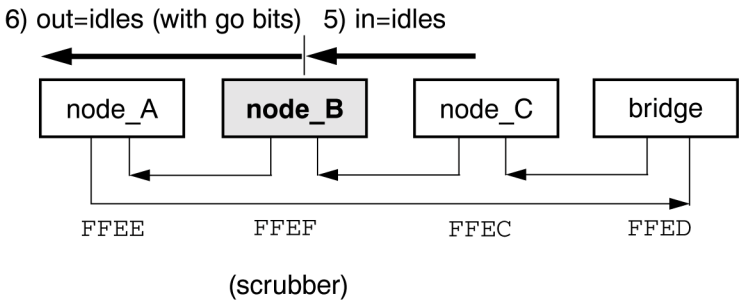


Figure 101 – Idle-closure injects go-bits in idles

3.10.5 Simple-subset ringlet resets

A simplified subset of this general model is also provided: one node on each ringlet may be configured by a means not specified by SCI (perhaps a jumper option) to be the scrubber. This node always considers itself to have the highest UID, and it emits RESETH and CLEARH packets that cause the other nodes to consider that they have lower UIDs. Thus this node always wins the scrubber competition, even if many other nodes have scrubber capability. Nodes of this type that are configured not to be scrubbers, and nodes with no scrubber capability, always lose the competition.

3.10.6 Ringlet resets

The previous subclauses illustrated how all nodes participate in the ringlet initialization activity. The following subclauses illustrate the behaviour of state machines in the individual SCI nodes.

Ringlet-local initialization begins when primary power is turned on. Each node generates a (presumably) unique 64-bit random number and concatenates this after a 16-bit value provided by nonvolatile storage (or the backplane status signals, if nonvolatile storage is not provided) to form its UID. The node then sets its phase bit *ph* to zero, and sends a reset packet (RESETL0) that contains this UID and the distanceId value SCRUB_ID, followed by sync (training) packets.

A node continues sending its reset and sync packets (state reset) waiting for its own RESETL0 packet to be returned. If its own RESETL0 packet is observed, the node changes its state (to winning) and outputs idle symbols. If a larger effective UID value is observed, the node changes state (to losing) and forwards reset packets (decrementing and saving their distanceId values) until an idle is received at its input, as illustrated in the reset state diagram of figure 102.

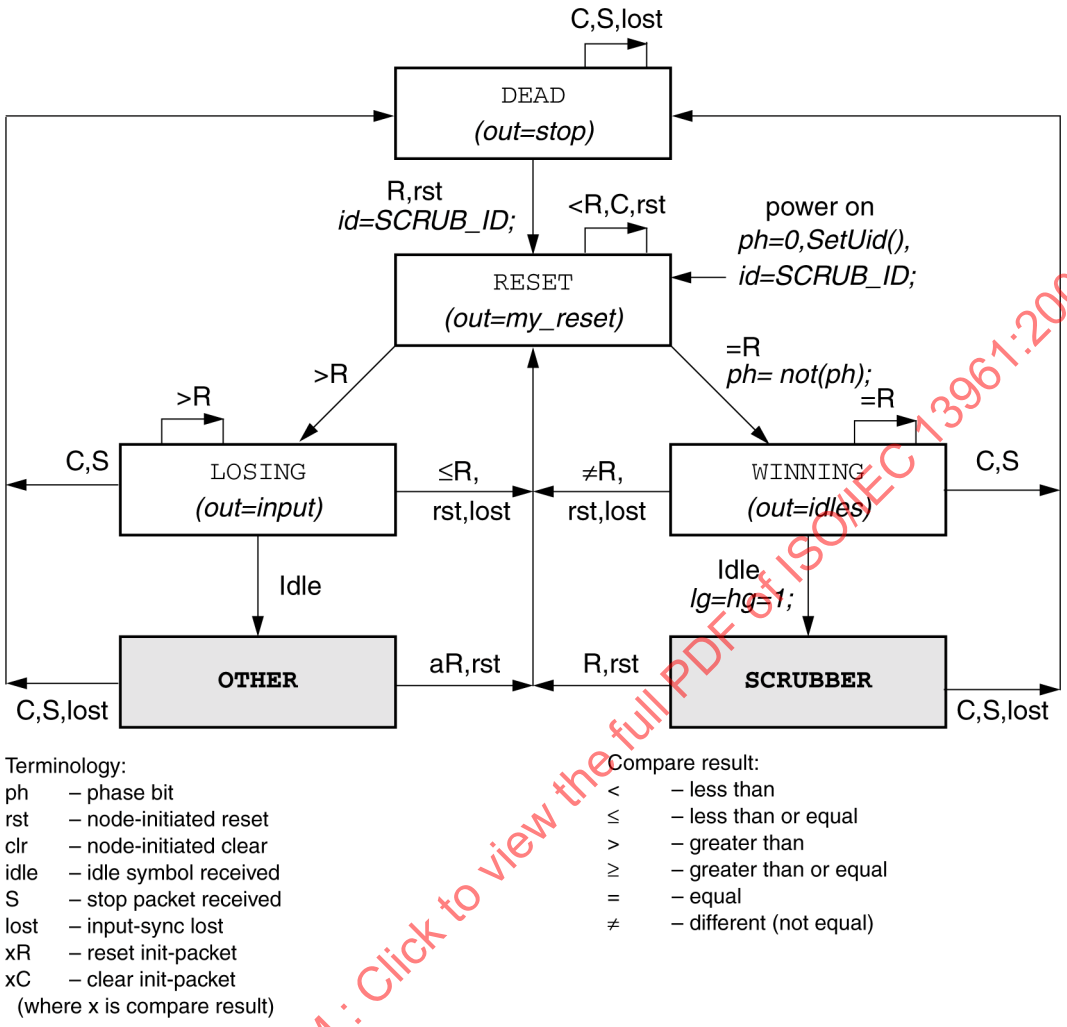


Figure 102 – Initialization states

The UID comparison is performed as follows: Nodes with scrubber-competition capability compare the incoming 80-bit UID with their own UID. If the incoming effective UID is greater than their own UID, the comparison is *greater than*. Nodes that have no scrubber capability or that are configured not to be scrubbers always generate *greater than*. Nodes that are configured to always be the scrubber generate *less than* until they receive a RESETH or CLEARH, whereupon they generate *equal*.

The chosen scrubber waits until its idle symbols return, then enters its operational state (winner), and injects *idle.lg*/*idle.hg* bits to enable transmissions by other nodes.

The reset sequence fails if a clear packet is observed in the LOSING or WINNING states (only reset packets or idles should be observed), and the node enters the DEAD state (an explicit reset is required to leave this state). Similarly, a node enters the DEAD state when input synchronization is lost.

If the reset process is restarted after a node has entered the WINNING or SCRUBBER states (perhaps because of power cycling), the node's phase bit (*ph*) has been complemented. The phase bit is copied to the LSB of the first reset packet symbol, to distinguish old reset packets (created before the reset was restarted) from the new (that are used to ensure ringlet closure). When looking for its own reset packet, reset packets with the incorrect phase bit are ignored.

3.10.7 Ringlet clears (optional)

Nodes may optionally provide a ringlet-clear capability that allows the flushing of packets and state bits from the linc when state-error or synchronization-error conditions have been detected. A ringlet clear may be initiated by the node's processor or other logic, or may be autonomously initiated by the node interface logic when a state error or synchronization error is detected, as illustrated in figure 103.

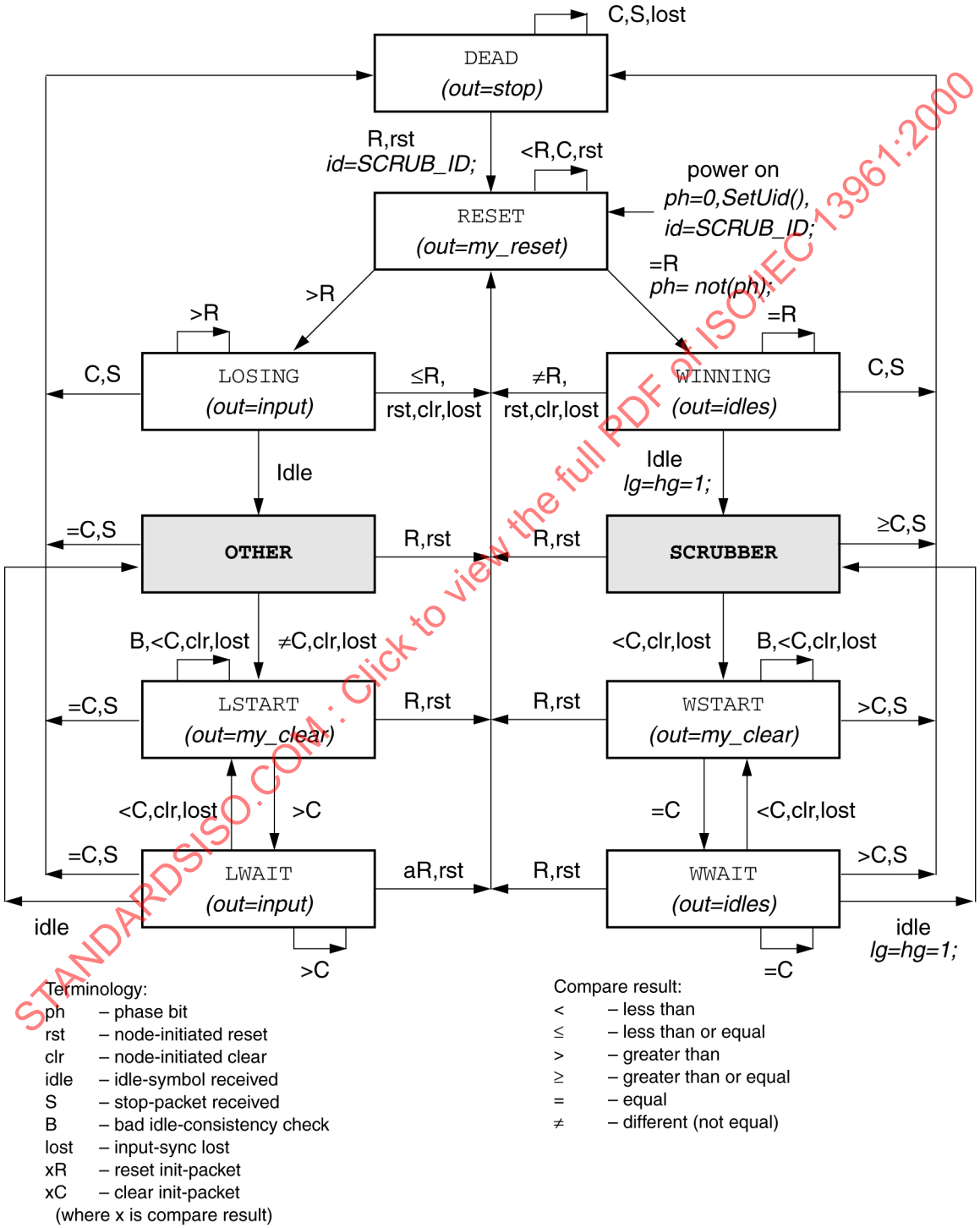


Figure 103 – Initialization states (*clear* option)

During the initial phase of a ringlet clear, nodes output clear packets containing their own UID values. This continues (in the state LSTART or WSTART) until the appropriate clear packet is observed at the input. A node in the LSTART state moves to the LWAIT state when a clear packet with a higher UID (which might be from the scrubber) is observed. UID comparisons are performed as explained in the previous subclauses.

The scrubber moves from the WSTART to the WWAIT state when ringlet closure is ensured a *clear* packet with an equal UID (from the scrubber itself) is observed. The scrubber then outputs idle symbols (with go bits cleared), until idle symbols are observed at its input. Ringlet operation is then activated by injecting go bits into the scrubber's output idles.

3.10.8 Inserting initialization packets

During the initialization process, input send and echo packets are discarded and a periodic sequence of init packets is output. Several types of init packets are output, as illustrated in figure 105. When a node is *reset* or cleared, it initially outputs an *abort* packet immediately followed by a SYNC packet to cleanly terminate any currently active packet transmissions.

The node then outputs packets containing its own UID, illustrated as *mine*. Each received *init* packet is initially stored in *save*, then transferred to the *pass* buffer after the CRC has been checked and the initialization state has changed appropriately. Nonscrubber nodes eventually output packets from *pass*, after they observe an incoming UID value higher than their own.

The initialization sequence involves inserting init packets between sequences of 1 023 sync packets, so the downstream neighbour can properly synchronize its receiver circuits before participating in the initialization process. This affects the data paths of the output multiplexer and the counters that are needed to properly sequence the output data symbols. The counters are expected to support sequencing through *init* and sync packets as illustrated in figure 104.

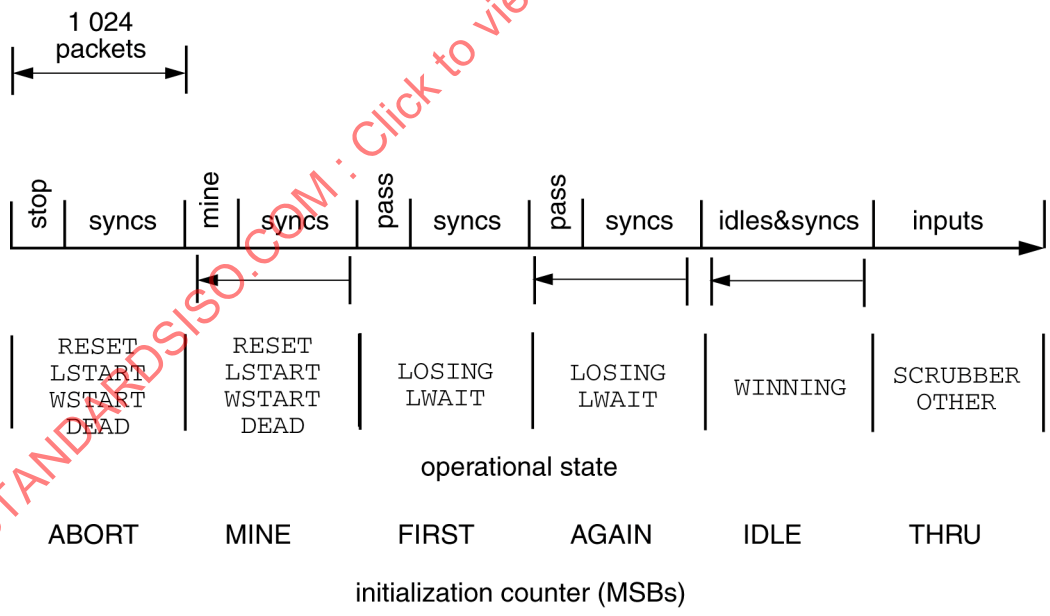


Figure 104 – Output symbol sequence during initialization

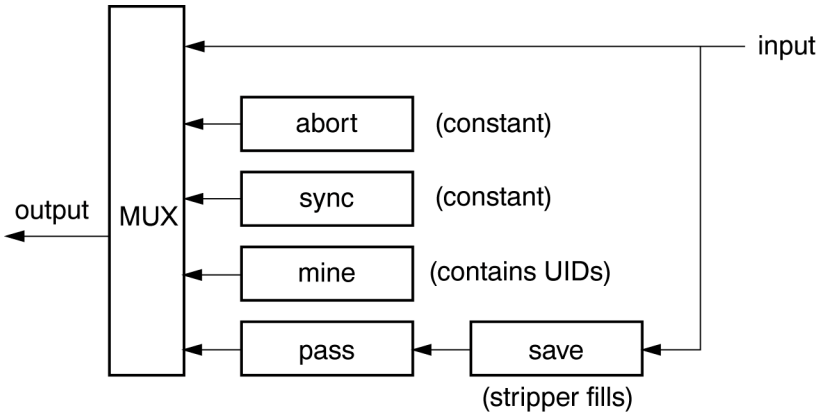


Figure 105 – Insert-multiplexer model

3.10.9 Address initialization

On a single- or multiple-ringlet system, the ringlet initialization process selects the scrubber, assigns the initial ringlet-local addresses and enables transmissions from nodes on the ringlet. In a multiprocessor system, processor firmware is responsible for selecting at most one processor on each ringlet to participate in the initialization process; this processor is called the *monarch*. Since memory is not yet available, special lock registers are expected to support the monarch selection process. However, the details of the special lock registers and the processor's access protocols are beyond the scope of the SCI standard. Note that the process for selecting the *monarch* processor is independent of the process used to select the scrubber.

Because switches or bridges between ringlets are initially disabled, the monarch selection process can be performed independently on each ringlet in the system. The node that is selected to be the monarch may be the same as or different from the node that was previously selected to be the scrubber. The initial nodeId values are distinct for all nodes on the same ringlet, but may be the same for nodes on different ringlets, as illustrated in figure 106.

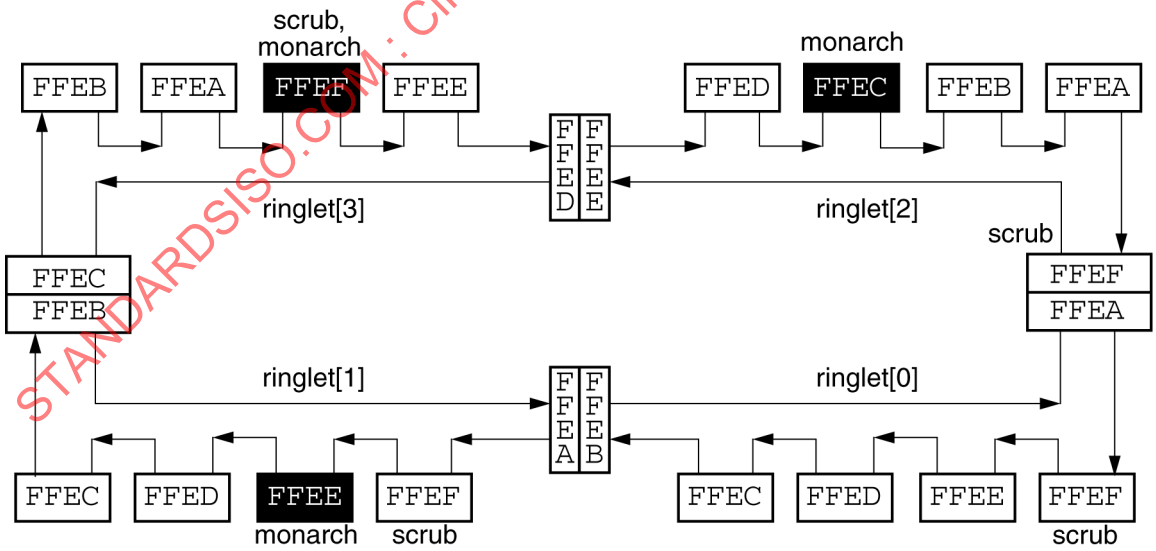


Figure 106 – NodeIds after ringlet initialization and monarch selection

Note that a bridge may have two or more node interfaces, one on each ringlet connection. Each of these node interfaces has a nodeId value that is initialized independently by the ringlet initialization process. The scrubber's initial nodeId address is `SCRUB_ID`, and the other initial nodeId values are assigned sequentially in decreasing order.

In a tightly coupled system, the monarch processors execute a distributed selection protocol to select one of the monarchs (called the *emperor*) that continues the initialization process. The emperor is responsible for establishing the ranges of addresses that is forwarded by each bridge and sets the initial nodeId addresses to be consistent with this address-forwarding plan, as illustrated in figure 107.

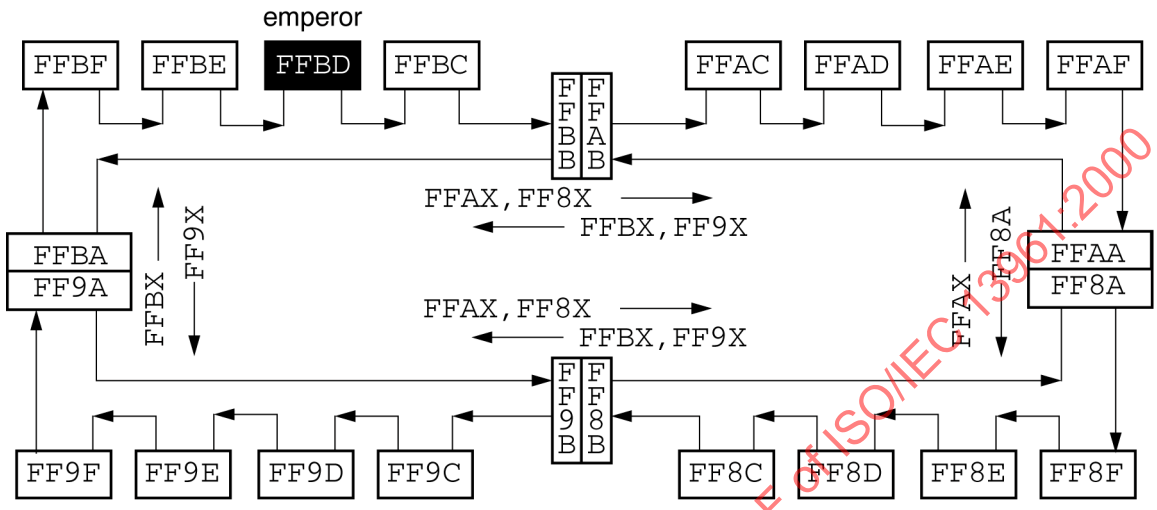


Figure 107 – NodeIds after emperor selection, final address assignments

In figure 107, a shorthand notation specifies which of two address ranges (FFAX or FF8X) is forwarded through the bridge from ringlet 3 to ringlet 2. The first three digits specify the 12 MSBs of the nodeId addresses that are forwarded to the adjacent ring; the X digit indicates that the 4 LSBs are ignored.

Note that more complex topologies (such as multidimensional grids) and different routing protocols are possible. The fastest routing decisions are based on the packet's targetId address, which is in the first symbol of every packet. Slower routing decisions may be based on the packet's *command*, *sourceId*, or *control* symbols as well. Other configurations and routing protocols may require additional request/response queue pairs to avoid queue deadlocks. However, the details of these alternate queue designs (often called virtual circuits) are beyond the scope of this standard.

Since the emperor is expected to fetch its address-configuration software from nonvolatile memory (such as disk storage), the address assignment protocols can be customized to meet the requirements of a particular system.

3.11 Packet encoding

The encoding specifies how packet types, packet lengths, and idle symbols are uniquely identified. For flexibility in the physical encoding layer (which might support any of several data-path widths), the logical encoding is specified in terms of the receiver output signals. For example, if the physical layer specifies that the data are transmitted 8 bits at a time, the receiver would be responsible for merging pairs of 8-bit data items into 16-bit SCI symbols.

3.11.1 Common encoding features (L18)

The size of the fundamental SCI symbol is 16 bits. In addition, a clock signal is needed to define symbol boundaries, and a flag signal is needed for locating the starting and ending symbols of packets. Depending on the physical-layer encoding, some or all of these logical signals may be encoded and sent on one physical signal path.

A zero-to-one transition of the flag signal is used to mark the beginning of each packet, and the one-to-zero transition of the flag signal specifies the approaching end of each packet. The flag signal returns to zero for the final 4 symbols of send packets and for the final symbol of an echo packet as illustrated in figure 108. A zero flag always precedes the CRC of any packet, so the zero-to-one transition can always be used to identify the start of the next packet (even when there is no idle symbol between them).

This logical encoding is the basis for the definition of the logical protocols defined by the SCI standard. A physical encoding may differ, but shall define the conversions necessary to convert between the physical encoding and this logical encoding.

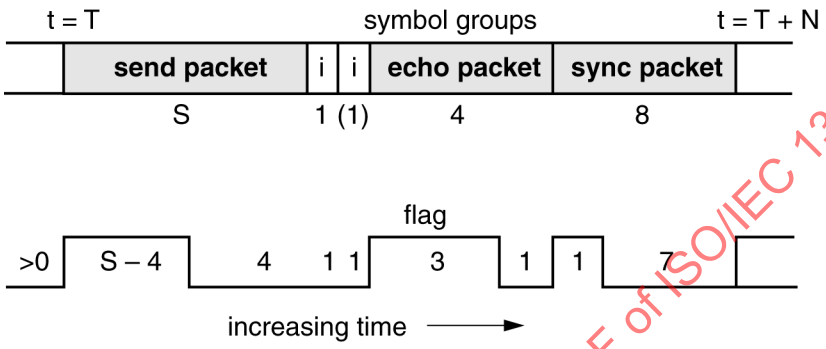


Figure 108 – Flag framing convention

3.11.2 Parallel encoding with 18 signals (P18)

The simplest encoding, called P18 encoding (parallel encoding, 18 signals) uses 18 signals to send one symbol (16 data bits, the flag, and the clock). The clock signal is used by the node interface to establish the phase difference between its internal clock and the clock associated with the incoming data. The flag signal uniquely delineates the start and end of SCI packets, so no special start or stop symbols are needed.

The flag signal directly transmits the logical flag signal, which is used to delimit the starting and ending symbols within a packet. The use of the data and flag signals in the P18 encoding is the same as in the logical symbol encoding L18.

The logical sync-packet encoding, which allows it to be readily distinguished from other send packets by its flag-bit transition, is also useful for synchronizing the P18 receivers. The all-ones symbol followed by the seven zero symbols provides a well-defined high-to-low transition for calibrating phase detection hardware in the data receivers. (Relatively large skews may be produced by inexpensive cables, and automatically compensated for by advanced SCI interfaces when circuit technology permits.)

3.11.3 Serial encoding with 20-bit symbols (S20)

For the S20 (serial, 20-bit symbols) encoding, 16 data bits, flag, and clock are encoded into one 20-bit unit that is transmitted one bit at a time, and the encoding ensures that the signal has no long-term d.c.-offset value. This encoding is much easier to map to its P18 equivalent than serial-encoding schemes that insert extra symbols to mark the transitions of the flag line. Thus, one may be able to use P18 chips within an S20-based node design.

A transition is guaranteed between the third- and second-from-last bits of each encoded S20 symbol. Implementations are expected to use this transition to maintain synchronization of the receiver with the data stream. During initialization this transition is always 0-1 and there is only one 0-1 transition per encoded symbol. During normal running the transition may be either 0-1 or 1-0.

With the exception of sync-packet symbols, the encoding of a 20-bit S20 symbol involves postpending the 16 bits of data with four additional bits, and complementing the 20-bit quantity as required to minimize the transmitter's cumulative d.c. offset value. If the flag bit is high, the 16 bits of data are postpended with a 1011 value before the complement decision is made. If the flag bit is low, the 16 bits of data are postpended with a 1101 value before the complement decision is made, as illustrated in figure 109.

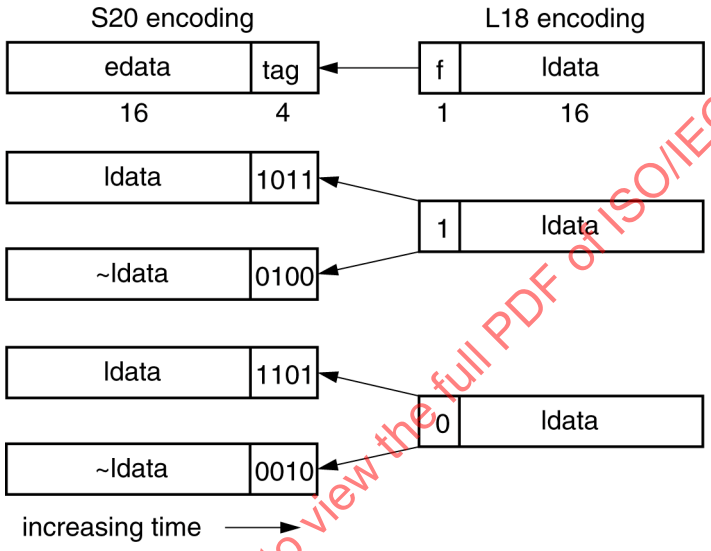


Figure 109 – S20 symbol encoding

In these figures, the left-most bit of each S20 symbol is always sent first. The 16 bits of encoded data are sent first (most-significant bit first) followed by the 4-bit tag value (whose bits have no arithmetic significance).

Either complement decision may be used when the symbol has a zero d.c.-offset value (the number of ones minus the number of zeros) or when the transmitter's accumulated d.c. offset is zero. When the intermediate 20-bit symbol has a nonzero d.c. offset and the transmitter's accumulated d.c.-offset value is also nonzero, the complement decision shall act to reduce the transmitter's accumulated d.c.-offset value when the symbol is sent. Thus, an all-zero or all-one symbol value may temporarily increase the magnitude of short-term excursions from the d.c. output value of zero, but does not cause any long-term imbalance to accumulate.

With the exception of sync-packet symbols, the decoding of a 20-bit S20 symbol is based on its postpended 4-bit data value, called the tag. If the tag is 1011 or 0100, the decoded flag bit is one; if the tag is 1101 or 0010, the decoded flag is zero, as illustrated in figure 110. For all legal SCI signal encodings, the tag shall be one of the 1011, 0100, 1101, 0010, or 0011 (sync packet symbol, see following discussion) values; other tag values are illegal and indicate the symbol value has been corrupted.

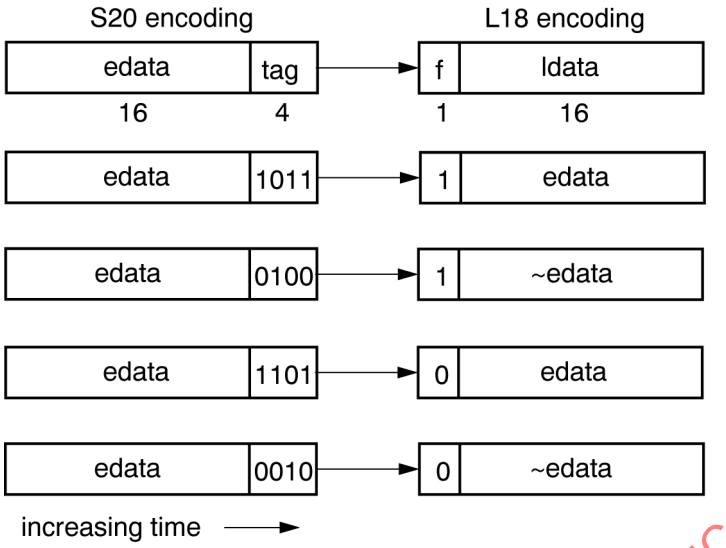


Figure 110 – S20 symbol decoding

The tag value also influences the interpretation of the encoded data. If the tag is 1011 or 1101, the decoded L18 data-bit values are the same as their corresponding S20 data-bit values. If the tag is 0100 or 0010, the decoded L18 data-bit values are the complement of their corresponding S20 data-bit values.

The encoded sync packet contains 8 repetitions of a unique encoded symbol value, sometimes called fill frame 0 (see 6.5.3.4). These symbol values are designed to simplify the receiver's phase-locked loops, which are provided with blocks of 1023 sync packets while the ringlet is being initialized. Phase-locked loops should easily be able to synchronize on these sync-packet sequences, since only one low-to-high (zero-to-one) transition occurs within each of these symbols, and the transition is always at the same place, as illustrated in figure 111.

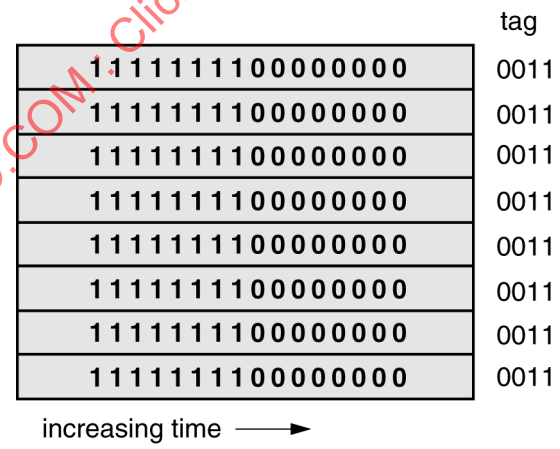


Figure 111 – S20 sync-packet encoding

The S20 sync packet is defined to be the same length as a P18 sync packet in order to make the interface between these two encodings as simple as possible, with a simple block substitution and a constant ratio of clocks between the two encodings.

3.12 SCI-specific control and status registers

3.12.1 SCI transaction sets

SCI follows the CSR Architecture. Certain transaction-set specifications are required by the CSR Architecture. Although the CSR Architecture specifies address-space formats and transaction-set requirements, a compliant standard is required to specify which address-space format is used and which optional transactions are implemented, as discussed in this clause.

The *64-bit fixed* address-space model is used. Note that the sixteen (16) highest node addresses, $FFF0_{16}$ $FFFF_{16}$, are used for specialized purposes (see table 1), and **shall never be assigned** by software. SCI uses command codes to specify whether a transaction is a broadcast, so special broadcast addresses are not required.

In addition to the transactions required by the CSR Architecture (table 8), the SCI transaction set also supports several optional transactions, as described below in table 14. Coherent read, write, and update transactions can be used to support the optional cache-coherence protocols. Selected-byte reads and writes can be used to access unaligned noncoherent data.

Table 14 – Additional SCI transaction types

transaction	size	align	description
cread	64	64	coherent processor-to-cache read transactions
cwrite64	64	64	coherent processor-to-cache write transactions
mread	00/64	64	coherent processor-to-memory read/control
mwrite16	16	16	coherent processor-to-memory write (subline)
mwrite64	64	64	coherent memory write (line)
readsb*	1-16	1	read selected (contiguous) byte addresses
writesb*	1-16	1	write selected (contiguous) byte addresses
nread256	256	64/256	read 256-byte block
nwrite256	256	64/256	write 256-byte block
lock4	4	4	indivisible 4-byte updates
lock8	8	8	indivisible 8-byte updates
event00	–	–	clockStrobe signal

NOTES

The read1, read2, read4, and read8 transactions are variants of readsb.

The write1, write2, write4, and write8 transactions are variants of writesb.

The nread256/nwrite256 transactions access an unaligned 256-byte block, but the starting address is 64-byte aligned.

SCI supports the 4-byte and 8-byte lock transactions defined by the CSR Architecture, as specified in table 11. In addition, SCI reserves 8 lock-transaction subcommands for possible future extensions to the SCI standard.

SCI also supports several types of responseless transactions, including directed moves, broadcast moves, and events. For move transactions, the command (rather than the address) is used to distinguish between the directed and broadcast versions. Events are a special form of directed move, which is used to transport signals and can never be busied. One of these transaction types (event00) provides the *clockStrobe* synchronization signal. These transactions are described in tables 9 and 10.

A response subaction (when provided) returns a 4-bit completion-status code to the requester. The various forms of error status are encoded into this 4-bit *status.sStat* field, as summarized previously in table 5.

3.12.2 SCI resets

SCI supports several types of reset, in addition to the power_reset and command_reset defined by the CSR Architecture. The warm_reset is a variant of power_reset, and is expected to be processed identically by software.

The linc_reset and linc_clear forms of reset are distinct SCI capabilities, which are initiated by sending special packets and affect all nodes on the attached ringlet. Both of these clear all queues and allocation state (i.e., they provide a bus-clear functionality). The linc_reset also resets node state.

Nodes are expected to initiate a linc_clear after fatal node-transmission failures. Nodes are expected to initiate a linc_reset if the milder linc_clear does not succeed. See 3.10.2 for details.

3.12.3 SCI-dependent fields within standard CSRs

SCI follows the CSR Architecture. Certain register fields in that standard are reserved for definition by the using standard. Such register fields and other SCI-specific details are given in this clause, which should be viewed as a supplement to the CSR Architecture and should be read in conjunction with it.

For all CSRs, including those fully defined by the CSR Architecture, SCI places some minimum performance constraints on CSR accesses. Without such performance constraints it would be impossible to accurately set the `SPLIT_TIMEOUT` register in SCI-based systems. When accessing a CSR-Architecture-standard-defined or SCI-defined CSR, the access should take no longer than 10 µs and shall take no longer than 100 µs.

3.12.3.1 NODE_IDS register

Initial *nodeId* values are assigned by the scrubber during the ringlet-initialization process, which is invoked when the system is powered on. After a power_reset, the ringlet scrubber is assigned an initial *nodeId* value of `SCRUB_ID`. Other *nodeIds* are assigned sequentially decreasing values, based on the node's distance downstream from the ringlet scrubber (the closest node has the highest initial *nodeId* value).

The *nodeId* value is not changed by a command_reset, but can be read or written when the node responds to its address space, as illustrated in figure 112.

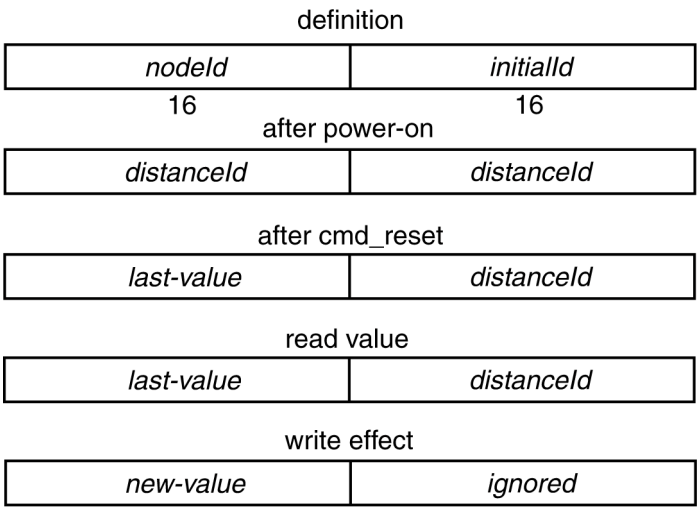


Figure 112 – NODE_IDS register

The initial values of the *nodeId* and *initialId* fields are the same and both are generated by a power_reset. The *initialId* field depends on the node's location relative to the scrubber; the scrubber's *initialId* field is equal to SCRUB_ID. The *initialId* value of the scrubber's downstream neighbour is one less, the next downstream neighbour is two less, etc. These *initialId* values are summarized in table 15.

Table 15 – Initial nodeId values

nodeId	name	description
$FFEF_{16}$	SCRUB_ID	scrubber's initial *nodeId* (address)
$FFEE_{16}$	down1-Id	first downstream neighbour, initial *nodeId* value
$FFED_{16}$	down2-Id	second downstream neighbour, initial *nodeId* value
. . .	. . .	other ringlet-local *nodeId*s

The *initialId* field is read-only from a software perspective, and is provided for discovering which nodes are on the same module. The *nodeId* field is compared to a packet's *targetId* symbol when selecting which packets are processed by the node. The nodeId field may be written as well as read, to relocate the node's initial address space.

3.12.3.2 STATE_CLEAR register

The STATE_CLEAR register provides bit fields that can be used to log special bus-dependent events. SCI reserves 8 bus-dependent bits, as illustrated in figure 113.

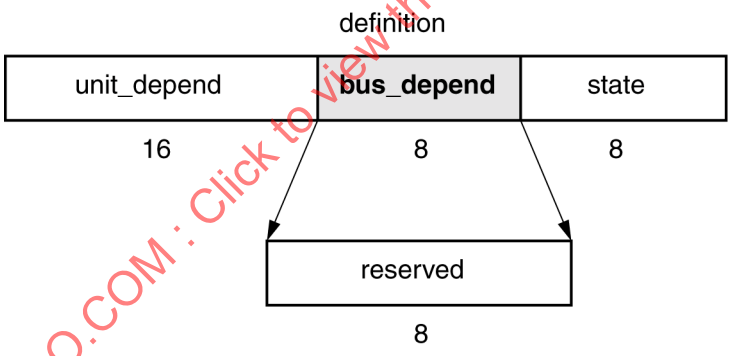


Figure 113 – STATE_CLEAR fields

The CSR Architecture defines several optional state bits, including *lost* and *dreq*. On SCI, the *lost* bit shall be implemented on all nodes and the *dreq* bit shall be implemented on all nodes that can generate request subactions.

Special bits are not required for identifying the scrubber on each ringlet, since the NODE_IDS.*initialId* addresses provide an equivalent functionality. When multiple nodes are implemented on one module, the matching of node addresses to module locations is assisted by a ROM entry that identifies the initial, intermediate, and final nodes on the module.

3.12.3.3 SPLIT_TIMEOUT register

The SPLIT_TIMEOUT register provides the default split-response timeout value for SCI nodes. On SCI, only the 16 most-significant bits of the SPLIT_TIMEOUT_LO register are required, as illustrated in figure 114.

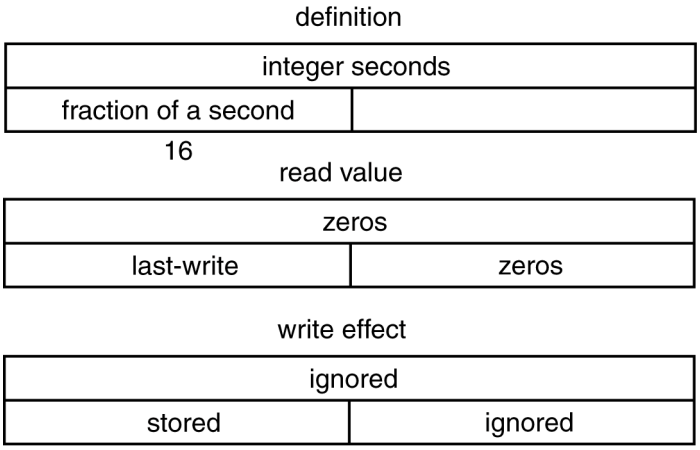


Figure 114 – SPLIT_TIMEOUT register-pair format

3.12.3.4 ARGUMENT register

The ARGUMENT register provides the address for a remote range of memory addresses that can be used during node tests. SCI reserves 11 of the bus-dependent bits within this register, as illustrated in figure 115.

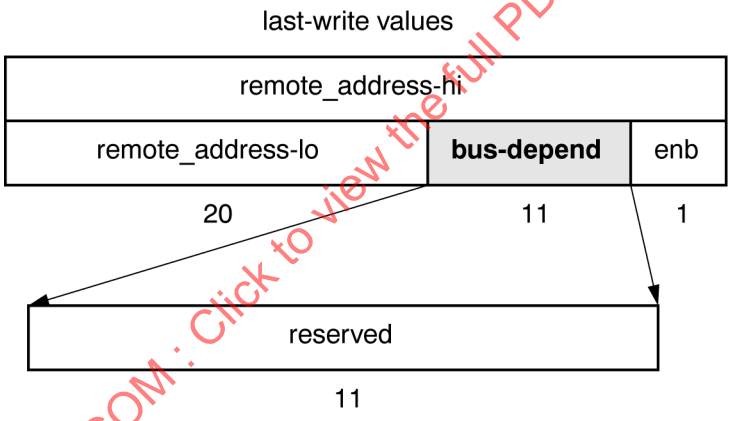


Figure 115 – ARGUMENT register-pair format

3.12.3.5 Unimplemented registers

None of the extended address registers is implemented. These unimplemented registers are listed in table 16.

Table 16 – Never-implemented CSR registers

register	description
UNITS_BASE_HI	unit address extensions (base registers)
UNITS_BASE_LO	unit address extensions (base registers)
UNITS_BOUND_HI	unit address extensions (bound registers)
UNITS_BOUND_LO	unit address extensions (bound registers)
MEMORY_BASE_HI	memory address extensions (base registers)
MEMORY_BASE_LO	memory address extensions (base registers)
MEMORY_BOUND_HI	memory address extensions (bound registers)
MEMORY_BOUND_LO	memory address extensions (bound registers)

3.12.4 SCI-dependent CSRs

Certain registers in the CSR Architecture are reserved for definition by the using standard. Those registers and other SCI-specific details are given in the following, which should be read in conjunction with the CSR Architecture.

3.12.4.1 CLOCK_STROBE_ARRIVED register

The optional CLOCK_STROBE_ARRIVED registers are defined by the CSR Architecture. In SCI, these registers sample the CLOCK_STROBE_VALUE registers at the time the *clockStrobe* signal is created or received by the node. See 3.4.6 and the CSR Architecture for further details.

3.12.4.2 CLOCK_STROBE_THROUGH register

The CLOCK_STROBE_THROUGH provides a measure of the time taken by the *clockStrobe* transaction to pass through the node. For the *clockStrobe* master, this register measures the time between the creation and transmission of the *clockStrobe* packet. For the *clockStrobe* slaves, this register measures the time between the arrival and departure of the pass-through *clockStrobe* packet. The format of the CLOCK_STROBE_THROUGH register is shown in figure 116.

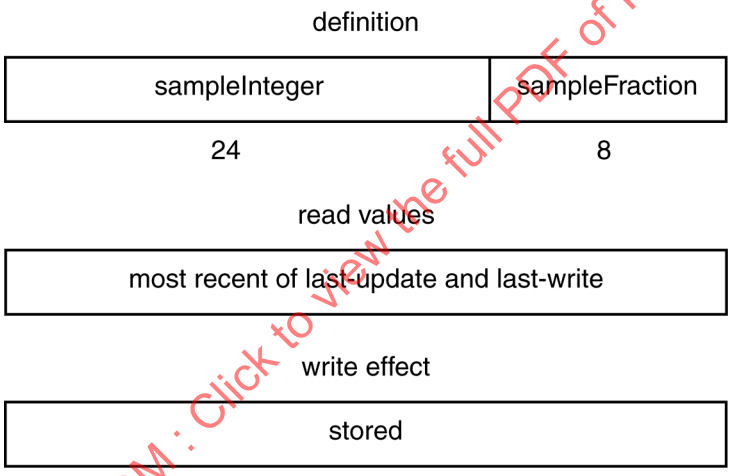


Figure 116 – CLOCK_STROBE_THROUGH format (offset 112)

CLOCK_STROBE_THROUGH:

Optional(RW): Required.

Initial value: 0

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_THROUGH register are expected to be used only for diagnostic purposes, since this register is updated by the *clockStrobe* packet during normal system operation.

3.12.4.3 ERROR_COUNT register

The optional ERROR_COUNT register, shown in figure 117, provides an inexpensive method of logging transmission errors that are not returned to the requester. The ERROR_COUNT register is incremented once for every error-interval (64 16-bit symbol times) during which an error was detected. Unlike most CSRs, this register is cleared by a power_reset or warm_reset, but is not affected by a command_reset, linc_reset, or linc_clear.

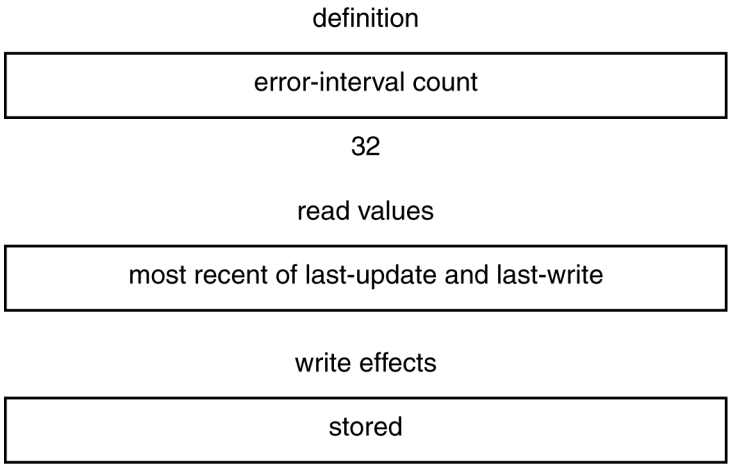


Figure 117 – ERROR_COUNT register (offset 384)

Note that the STATE_CLEAR.elog bit is also set whenever the ERROR_COUNT register is incremented. However, reads and writes of the ERROR_COUNT register have no effect on the state of the STATE_CLEAR.*elog* bit.

3.12.4.4 SYNC_INTERVAL register

The mandatory SYNC_INTERVAL register specifies the time interval at which the special sync packets should be generated. A default value is set during the ringlet initialization process. After ringlet initialization, software is expected to update this register with a time interval appropriate for normal operation of the particular implementation. Only the 24 most-significant bits of the SYNC_INTERVAL register are required, as illustrated in figure 118.

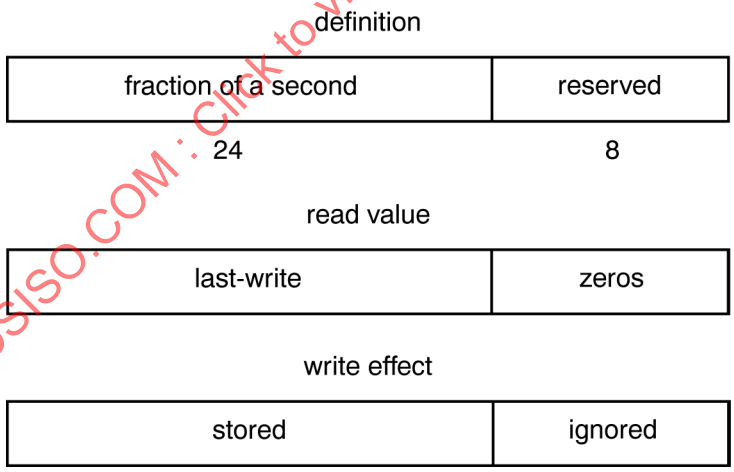


Figure 118 – SYNC_INTERVAL register (offset 512)

Unlike other CSRs, this register is initialized by a power_reset, warm_reset, or ringlet_reset and is unaffected by a command_reset or linc_clear. For the 18-DE-500 link, the initial value of this register is set to 00004000_{16} (and for the 1-FO-1250 or 1-SE-1250 links it is set to 00020000_{16}).

3.12.4.5 SAVE_ID register

The SAVE_ID register provides access to the 16 bit stableId value, which forms the most-significant portion of the 80-bit unique identifier (UID). This UID value is used during system initialization to determine which node becomes the ringlet scrubber. The most-significant 16 bits of this register are reserved. If the node supports a nonvolatile identifier, the format of this register is illustrated in figure 119.

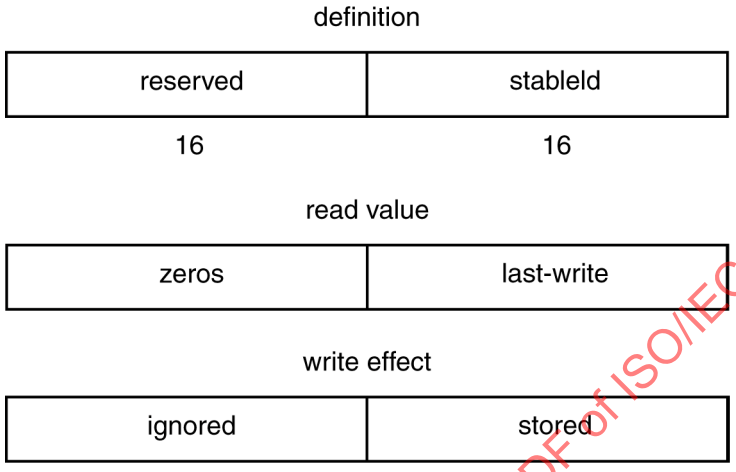


Figure 119 – SAVE_ID register (offset 520)

If a nonvolatile identifier is not supported, the behaviour of this register is identical to that defined for the SLOT_ID register (see 3.12.4.6).

Unlike other CSRs, this register is unaffected by a power_reset, warm_reset, command_reset, ringlet_reset, or linc_clear.

3.12.4.6 SLOT_ID register

The optional SLOT_ID register provides read-only access to a maximum of 16 backplane signal values. The *slotSignals* field is obtained from signals provided by the backplane. The most-significant bits of this field may be partially implemented; any unimplemented bits shall be zero. The format of this register is illustrated in figure 120.

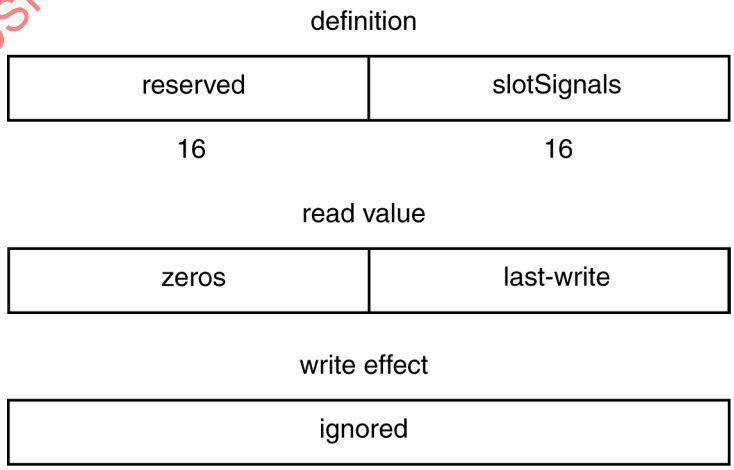


Figure 120 – SLOT_ID register (offset 524)

On the Type 1 Module version of the standard, these bits are hard-wired to zero or connected to backplane-provided geographical addressing signals. An implementation may connect less than 16 geographical address signals; the signals that are connected shall correspond to a contiguous range of least-significant bits within the *slotSignals* field.

On the bit-serial Type 1-FO-1250 or Type 1-SE-1250 versions of the standard, the *slotSignals* field shall be zero.

3.12.4.7 Vendor-dependent registers

Vendor-dependent registers may be placed in the CSR-offset range of 768 to 1020. These addresses are expected to be used for node-related purposes. Unit-specific registers are expected to be assigned to other register addresses (starting from address-offset 2048).

3.12.5 SCI-dependent ROM

3.12.5.1 Overall ROM format

The CSR Architecture provides a framework for defining the location, format, and meaning of node-supplied ROM information. The term ROM is used to describe the read-only nature of this information, which could be physically located in nonvolatile memory or could be initialized by a vendor-dependent support processor. The CSR Architecture defines a bus_info_block, whose length and format are bus-dependent; for SCI, this is 32 bytes in size and has the format illustrated in figure 121.

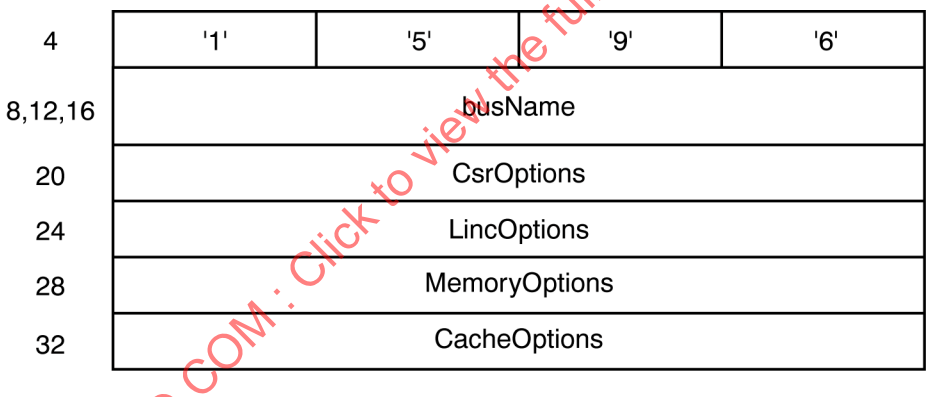


Figure 121 – SCI ROM format (bus_info_block)

The first four bytes are ASCII numerical characters that uniquely identify SCI by its project number. The following 12 bytes are null-terminated character strings that specify which physical standard is implemented, as shown in table 17.

Table 17 – Physical standard description

name	description
T-18-DE-500	18 signals, differential ECL, 500 Mperiods/second
T-1-SE-1250	1 signal, differential ECL, 1250 Mperiods/second

Note that the T-1-FO-1250 option is not explicitly supported in ROM, since its functional behaviour is expected to be identical to the defined T-1-SE-1250 option. Vendor-dependent implementations that intend to closely imitate the capabilities of the SCI standard should use names that begin with the two characters V– , to avoid confusion with the SCI names that begin with the two characters T–.

The bus_info_block contains four additional quadlets called CsrOptions, LincOptions, MemoryOptions, and CacheOptions, whose formats are specified in 3.12.5.2 to 3.12.5.5. These quadlets specify which SCI options are implemented. Although most of these options have no effect on system software, identifying the implemented options is expected to be useful for diagnostic, verification, and initial configuration purposes.

3.12.5.2 Format of CsrOptions

The CsrOptions quadlet specifies which of the optional CSR registers (or portions of registers) are implemented, as illustrated in figure 122.

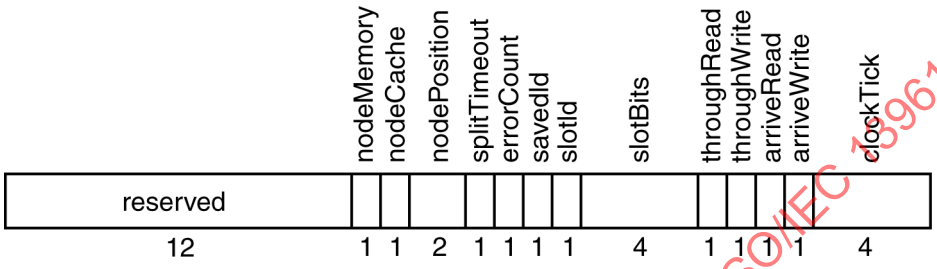


Figure 122 – ROM format, CsrOptions

If the *nodeMemory* bit is 1, the node supports SCI memory and the following MemoryOptions quadlet shall be nonzero. If the *nodeMemory* bit is 0, the node does not support SCI memory and the following MemoryOptions quadlet shall be zero. If the *nodeCache* bit is 1, the node supports a coherent SCI cache and the following CacheOptions quadlet shall be nonzero. If the *nodeCache* bit is 0, the node does not support a coherent SCI cache and the following CacheOptions quadlet shall be zero.

The 2-bit *nodePosition* field is provided to help identify nodes that are physically located on the same module. If the module has only one node on this ringlet, its *nodePosition* value is 0. If the module has two or more nodes attached to the same ringlet, the *nodePosition* value is 1 for the most-upstream node and 3 for the most-downstream node. If the module has three or more nodes attached to this node's ringlet, the *nodePosition* value is 2 for the other nodes attached to the same ringlet.

If the *splitTimeout* bit is 1, the 64 bits of the SPLIT_TIMEOUT register pair shall be implemented. If the *splitTimeout* bit is 0, only 16 bits of the SPLIT_TIMEOUT_LO register shall be implemented.

If the *errorCount* bit is 1, the optional ERROR_COUNT register shall be implemented. If the *saveId* bit is 1, the optional SAVE_ID register shall be implemented.

If the *slotId* bit is 1, a portion of the SLOT_ID register shall be implemented, and the value of *slotBits*+1 shall specify the number of implemented least-significant bits within this register. If slotId is 0, the SLOT_ID register shall not be implemented and the value of *slotBits* shall be 0.

If the *throughRead* bit is 1, the CLOCK_STOBE_THROUGH register shall be read-only. If the *throughWrite* bit is 1, the CLOCK_STROBE_THROUGH register shall be readable and writeable. The *throughRead* and *throughWrite* bits are mutually exclusive, in that one and only one of these two bits shall be 1.

If the *arriveRead* bit is 1, the CLOCK_STROBE_ARRIVED register pair shall be read-only. If the *arriveWrite* bit is 1, the CLOCK_STROBE_ARRIVED register shall be readable and writeable. The *arriveRead* and *arriveWrite* bits are mutually exclusive, in that at most one of these two bits shall be 1; if both bits are zero the CLOCK_STROBE_ARRIVED register is not implemented.

The *clockTick* field shall specify the approximate size of the clock-tick period (in 32-bit fractions of a second). If the *arriveRead* and *arriveWrite* bits are both zero, the value of *clockTick* shall be zero. Otherwise, the value of *clockTick* shall be the smallest integer for which the following inequality is true: *clockTickPeriod*<(1<<*clockTick*), where *clockTickPeriod* is the time period between clock updates measured in units of 2^{-32} s.

3.12.5.3 Format of LincOptions

The LincOptions quadlet specifies which of the optional linc capabilities are implemented, as illustrated in figure 123.

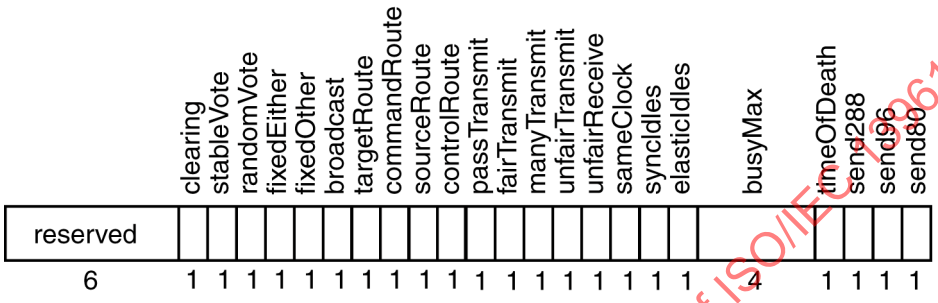


Figure 123 – ROM format, LincOptions

The *clearing* bit is 1 if the node supports the optional linc_clear capability. The *stableVote* bit is 1 if a 64-bit stable identifier is provided, and that identifier is used during system initialization to select the ringlet scrubber. The *randomVote* is 1 if a random 64-bit identifier is used during system initialization to select the ringlet scrubber. The *fixedEither* bit is 1 if the linc chip can be selectively configured to be either the scrubber or a nonscrubber node respectively. The *fixedOther* bit is 1 if the linc chip can only be a nonscrubber node. The *stableVote*, *randomVote*, *fixedEither*, and *fixedOther* bits are mutually exclusive, in that one and only one of these four bits shall be set to 1.

If the *broadcast* bit is 1, the node accepts broadcast send packets. If the *targetRoute* bit is 1, the node's routing decisions are only influenced by the packet's *targetId* symbol. If the *commandRoute* bit is 1, the node's routing decisions are only influenced by the *targetId* and *command* symbols (for example, requests and responses have different routes). If the *sourceRoute* bit is 1, the node's routine decisions are only influenced by the *targetId*, *command*, and *sourceId* symbols (for example, the packet's route depends on where it originated). If the *controlRoute* bit is 1, the node's routing decisions are influenced by the *targetId*, *command*, *sourceId*, and *control* symbols (for example, the packet's route depends on the send packet's *control.transactionId* field.

The *targetRoute*, *commandRoute*, *sourceRoute*, and *controlRoute* bits are mutually exclusive, in that one and only one of these four bits shall be set to 1. If the broadcast bit is 1, either the sourceId or the controlId bit shall be 1 (routing of broadcast packets is influenced by the third *sourceId* symbol).

If the *passTransmit* bit is 1, the node uses the pass transmission protocol and the *elasticIdles* bit shall be 0. If the *fairTransmit* bit is 1, the node uses only the low-transmission protocol, and only two packets (one request send and one response send) are simultaneously active (packets have been transmitted, but no echo has been returned). If the *manyTransmit* bit is 1, the node uses only the low-transmission protocol but more than one request send and one response send may be simultaneously active. If the *unfairTransmit* bit is 1, the node uses both the low- and high-transmission protocols to support prioritized send-packet transmissions. The *passTransmit*, *fairTransmit*, *manyTransmit*, and *unfairTransmit* bits are mutually exclusive, in that one and only one of these four bits shall be set to 1.

If the *unfairReceive* bit is 1, the node uses priority to selectively bypass the send-packet acceptance protocols (i.e., priority packets are busied less often). The *unfairReceive* bit shall be zero if the *unfairTransmit* bit is zero, and should be 1 if the *unfairTransmit* bit is 1 (the unfairReceive capability is an optional extension of the unfairTransmit capability).

The *sameClock* bit is 1 if the node's clock is the same as its input clock and insertion/deletion of idle symbols is not performed. The *syncIdles* bit is 1 if the node's clock may be different from its input clock, and insertion/deletion of symbols can occur only during sync packet inputs. The *elasticIdles* bit is 1 if the insertion/deletion of idle symbols can occur between any input packets as well as during input sync packets. The *sameClock*, *syncIdles*, and *elasticIdles* bits are mutually exclusive, in that one and only one of these three bits shall be set to 1.

If the *busyMax* bit is 0, the node's busy-retry protocols do not count the number of previously busied subactions. If busyMax is nonzero, the node's busy-retry protocols count the number of previously busied subactions, using a binary counter with *busyMax* bits.

If the *timeOfDeath* bit is 1, the node supports time-of-death checks on queued send packets and (if requester capabilities are provided) can initialize these to nonzero values when a request-send packet is generated. If the *send288* bit is 1, the node can accept the largest SCI packets (an extended header plus 256 bytes of data). If the *send96* bit is 1, the node can accept 64-byte SCI packets with extended headers. If the *send80* bit is 1, the node can accept 64-byte SCI packets without extended headers. The *send288*, *send96*, and *send80* bits are mutually exclusive, in that one and only one of these three bits shall be set to 1.

3.12.5.4 Format of MemoryOptions

If memory is not supported, as indicated by the *nodeMemory* bit within the CsrOptions quadlet, the MemoryOptions quadlet shall be zero. Otherwise, the MemoryOptions quadlet specifies which of the optional memory capabilities are implemented, as illustrated in figure 124.

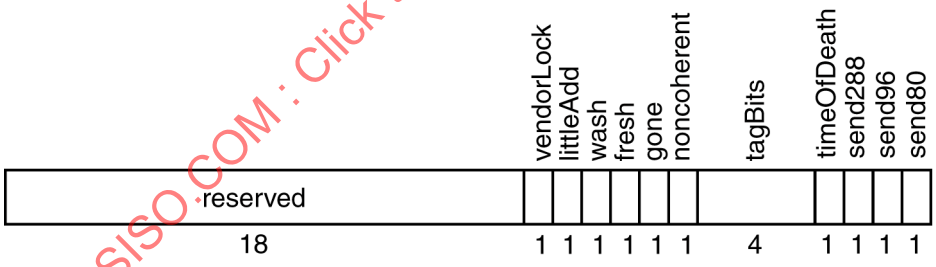


Figure 124 – ROM format, MemoryOptions

If the *vendorLock* bit is 1, the node's memory supports the vendor-dependent variant of the noncoherent locksb transaction. If the *littleAdd* bit is 1, the node's memory supports the LITTLE_ADD variant of the noncoherent locksb transaction.

If the *wash* bit is 1, the node's memory supports the coherent `MS_WASH` memory state. If the *fresh* bit is 1, the node's memory supports the coherent `MS_FRESH` memory state. If the *gone* bit is 1, the node's memory supports the coherent `MS_GONE` memory state. If the *noncoherent* bit is 1, the memory controller supports only the noncoherent accesses. The *wash*, *fresh*, *gone* and *noncoherent* bits are mutually exclusive, in that one and only one of these four bits shall be set to 1.

The *tagBits* field specifies the number of tag bits used to identify the owner of each coherently cached line; these nodeId values are saved as sign-extended values in a field that has *tagBits*+1 bits. If one of *gone*, *fresh*, and *wash* is 1, legal *tagBits* values shall include 7, 11, and 15; otherwise the *tagBits* field shall be 0.

If the *timeOfDeath* bit is 1, the memory controller supports time-of-death checks on queued send packets and (if requester capabilities are provided) can initialize these to nonzero values when a request-send packet is generated. If the *send288* bit is 1, the memory controller can accept the largest SCI packets (and extended header plus 256 bytes of data). If the *send96* bit is 1, the memory controller can accept 64-byte SCI packets with extended headers. If the *send80* bit is 1, the memory controller can accept 64-byte SCI packets without extended headers. The *send288*, *send96*, and *send80* bits are mutually exclusive, in that one and only one of these three bits shall be set to 1 if a memory controller is supported.

3.12.5.5 Format of CacheOptions

If cache is not supported, as indicated by the *nodeCache* bit in the CsrOptions quadlet, the CacheOptions quadlet shall be zero. Otherwise, the CacheOptions quadlet specifies which of the optional cache capabilities are implemented, as illustrated in figure 125.

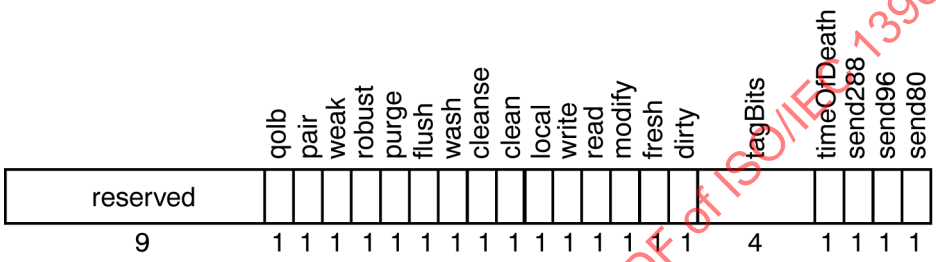


Figure 125 – ROM format, CacheOptions

The *qolb, pair, weak, robust, purge, flush, wash, cleanse, clean, local, write, read, modify, fresh,* and *dirty* bits specify which of the cache options are supported. See the C code for details.

The *tagBits* field specifies the number of tag bits used to identify the other entries in a coherence sharing list; these nodeId values are saved as sign-extended values in a field that has *tagBits*+1 bits. If cache is 1, legal *tagBits* values include 7, 11, and 15; otherwise *tagBits* shall be zero.

If the *timeOfDeath* bit is 1, the cache controller supports time-of-death checks on queued send packets. If the *send288* bit is 1, the node can accept the largest SCI packets (an extended header plus 256 bytes of data). If the *send96* bit is 1, the node can accept 64-byte SCI packets with extended headers. If the *send80* bit is 1, the node can accept 64-byte SCI packets without extended headers. The *send288*, *send96*, and *send80* bits are mutually exclusive, in that one and only one of these three bits shall be set to 1 if cache is supported.

3.12.6 Interrupt register formats

A single SCI system may include processors from many different suppliers. With shared memory, processors are expected to pass messages by writing the data to memory and interrupting another processor or processors. Software mailbox conventions, which are beyond the scope of this standard, are expected to standardize the format and meaning of the data structures in shared memory.

Although the architectures of various processors are likely to differ, standard interrupt and memory-controller architectures are intended to simplify the implementation of standard shared-memory-based message-passing protocols. Standardizing the processor's interrupt architecture is also expected to simplify monarch selection protocols, which may be defined in future extensions to this standard.

A node that contains one or more monarch-capable processors shall implement the INTERRUPT_TARGET register, as defined in the CSR Architecture. This provides an address for broadcasting interrupts to all monarch-capable processor units on the node. A write to this target address is distributed to all processors on the node, and shall be processed (as defined by the CSR Architecture) by all monarch-capable processors on the node. For interruptible uniprocessor nodes, this is the only required interface to the processor interrupt capability.

For nodes with two or more monarch-capable processors, a DIRECT_TARGET register shall be defined in the unit architectures of each monarch-capable processor (so that processors can be selectively interrupted).

A write to the processor's DIRECTED_TARGET register is routed to an individual processor on the node. For that processor, a write to the DIRECTED_TARGET register and a write to the node's INTERRUPT_TARGET register (with the INTERRUPT_MASK register set to all ones) shall be processed equivalently, as defined within this subclause.

The 32 data bits of a write4 transaction correspond to 32 interrupt-event priorities, where the most-through-least-significant bits of the data correspond to the highest (p[0]) through lowest (p[31]) priority interrupt-event respectively, as illustrated in figure 126.

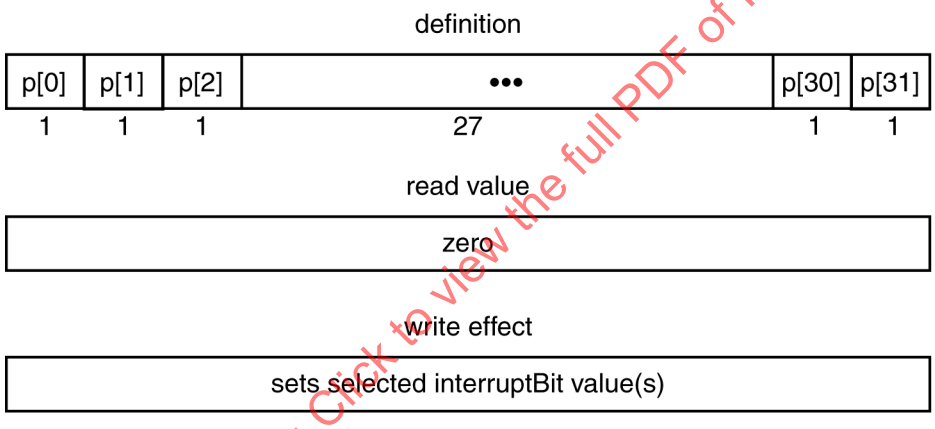


Figure 126 – DIRECTED_TARGET format

DIRECTED_TARGET:

Optional(WO): One should be provided on each interruptible processor.

Initial value: 0

Read4 value: Shall return 0.

Write4 effect: The write-data value is ORed with the processor's internal interruptBits.

Units that respond to DIRECTED_TARGET writes are expected to provide one bit to queue each interrupt event. When the DIRECTED_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 DIRECTED_TARGET register, the mapping shall be monotonic (higher-priority interrupt bits shall be mapped to more-significant bits within the DIRECTED_TARGET register), and all of the DIRECTED_TARGET bits shall be mapped to a unit interrupt bit.

When the DIRECTED_TARGET register is written, the write data are sent to the processor unit and may be ORed with the bits in the processor's internal interruptBits register. The enabling of processor interrupts is expected to be based on the bit position of the interrupt bit, and the processor is expected to provide mechanisms for selectively clearing bits within the internal interrupt-pending register. However, these internal processor-architecture details are beyond the scope of the SCI standard.

3.12.7 Interleaved logical addressing

Supporting interleaved addresses is an optional capability of a requester. However, having a common model supports interoperability between nodes made by different vendors.

On high-performance systems, memory interleaving is a cost-effective way of improving effective memory-access bandwidth. To simplify the hardware (and to improve burst-transfer rates), SCI supports interleaving on a cache-line (64-byte) granularity.

The interleaving is performed by a transformation of the logical DMA address (as specified in the DMA-command chain) to a physical DMA address (as used on SCI). Since the interleave transformation is performed inside the DMA controller, it has no effect on the data-transfer protocols defined by the SCI standard. Simple processors are assumed to use the same address-translation protocols, for compatibility with standard interleaved DMA controllers.

The interleaving involves an exclusive-OR of address bits. The width of the affected address bits is specified by the interleave width *w* and the location of the affected address bits is specified by the interleave shift parameter *s*, as illustrated in figure 127.

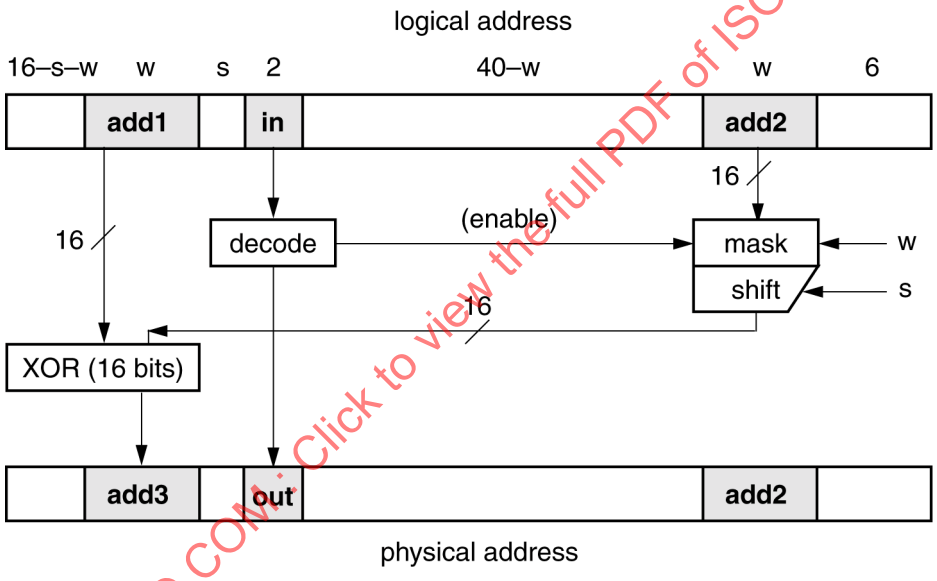


Figure 127 – Logical-to-physical address translation

The interleave shift field, *s*, provides flexibility for interleaving memory addresses from memory controllers with noncontiguous nodeId values. To make use of this interleave capability, initialization software is required to configure the nodeIds of the 2^n interleaved memory controllers to nodeId addresses that differ by only an *n-bit field*.

The upper two bits of the address offset field selectively enable the interleave operation. If the upper two bits are 00 or 11, interleaving is disabled. This supports noninterleaved access of the lowest or highest portion of the physical address space. If the upper two bits are 01, interleaving is enabled and the access address is mapped to the lower half of the address-offset space, as specified by table 18. If the upper two bits are 10, the address interpretation is vendor-dependent.

Table 18 – Interleave-control bits

in	out	enable
00	00	0
01	00	1
10	vd	vd
11	11	0

4 Cache-coherence protocols

4.1 Introduction

SCI supports multiprocessing with cache coherence for the very general distributed-shared-memory model. Use of cache coherence is optional, and there are also optional features within the cache-coherence model that interoperate compatibly but offer various tradeoffs of performance versus cost.

Some applications may choose to maintain cache coherence under software control instead of using SCI's automatic coherence mechanism, and others may prefer to use message-passing schemes. SCI supports all these styles efficiently and concurrently, so long as the system software correctly manages mixed-system operation.

4.1.1 Objectives

The set of cache-coherence states and transactions that is described in this document includes a number of optional subsets. These options are available to improve the performance of the frequent forms of cache sharing between entries in relatively short sharing lists. Performance enhancements for long sharing lists are under development for a future extension to SCI. The options included in this document are subject to the following constraints:

1) The coherence options can be implemented without significantly increasing the size of tags in the memory directory or caches.
2) The options work with the basic SCI transaction-set (request/response) definitions.
3) The options should not affect the correctness of the basic SCI cache-coherence specification.

4.1.2 SCI transaction components

SCI's high-performance design goals (1 Gbyte/s per node) forced a migration from bused backplanes to a unidirectional point-to-point-link interface. The interconnection possibilities for these links range from rings, through meshes of rings, to switch networks.

In order to support arbitrary interconnection mechanisms, SCI does not depend on broadcast transactions or eavesdropping third parties. Experienced switch-network designers claim that broadcasts are nearly impossible to route efficiently. Broadcasts are also hard to make reliable; with the large number of nodes on SCI (and therefore a high cumulative error rate) reliability and fault recovery are primary objectives.

Therefore, SCI cache-coherence protocols are based on directed point-to-point transactions, initiated by a requester (typically a processor) and completed by a responder (typically a memory or another processor). Most transactions consist of a request subaction followed by a response subaction. For example, the request subaction transfers the address to a memory controller and the response subaction returns data or caching status from the memory controller to the processor, as illustrated in figure 128.

Figure 128 – SCI transaction components

4.1.3 Physical addressing

For simplicity and interoperability, the SCI coherence protocols assume that a physical address is sufficient to extract cache entries from a cache. Although primary caches will often be virtually indexed, SCI expects that large secondary caches will isolate the interconnect from the virtual addresses generated by the processor.

Although virtually indexed caches are not supported by the SCI standard, a sufficient number of reserved fields is provided that such capabilities could be defined by extensions to the standard. With such extensions, virtual index bits could be transferred among compatible processors in fields reserved for vendor-specific uses. If standard DMA devices are used, explicit cache flushes may be required (in a virtual cache environment) before and after DMA transfers, as is done in some existing RISC architectures. Vendor-dependent DMA controllers that supply both the physical address and the virtual index bits could also be used.

4.1.4 Coherence directory overview

In buses that support caches, coherence is usually achieved by eavesdropping or snooping: all processors listen to the bus and invalidate or update their caches when data are written into memory. Noneavesdrop cache-coherence protocols, which scale beyond a bus, are generally directory-based. The following coherence properties form the basis for most of these schemes:

1) *Sharing readers.* Identical copies of a line of data may be present in several caches. These caches are called readers.
2) *Exclusive writer.* Only one cache at a time may have permission to write to a line of data. This cache is called the writer.
3) *Invalidate on write.* When a cache gains permission to write into a line, the writer notifies all readers to invalidate their copies.
4) *Accounting.* For each addressed line, the identity of all readers is stored in some kind of directory.

In limited configurations the directory could be centralized at the memory controller. However, SCI distributes the directory among the tags associated with coherently cached copies and the memory directory. By distributing the directory updates among multiple processors rather than using a central directory, SCI also distributes the housekeeping communication among the sharing processors. This is preferable to concentrating that communication at a heavily shared memory controller.

The SCI cache-coherence overview assumes that there is always one processor/CPU for each cache and that this processor executes the cache-coherence protocol. In an implementation there might, of course, be several processors with distinct primary caches that share a common secondary cache. In such configurations, the cache-coherence protocols are expected to be performed by a specialized cache controller, not the processor/CPU.

With SCI's distributed sharing lists each coherently cached line is entered into a list of processors sharing that line. Other lines may be locally cached, and are not visible to the coherence protocols. For illustrative purposes, both coherent and noncoherent lines are shown in figure 129.

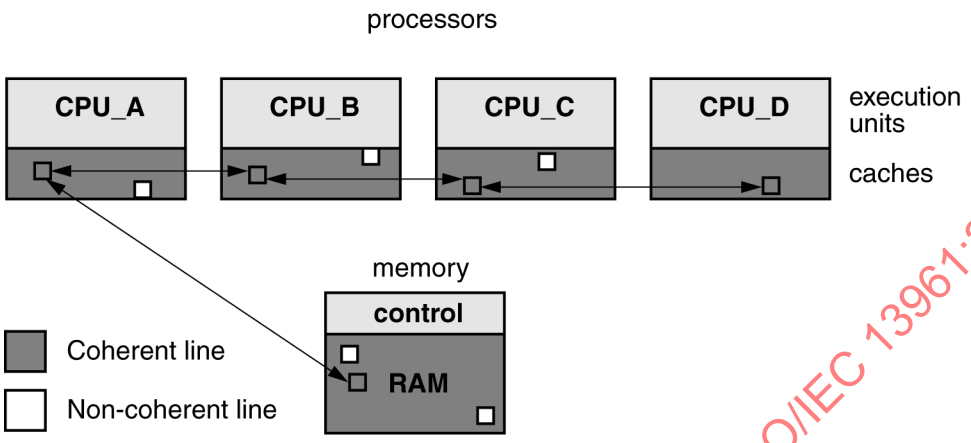


Figure 129 – Distributed sharing-list directory

Noncoherent copies may also be made coherent by higher-level software, perhaps on a page-level basis. However, the details of such software coherence protocols are beyond the scope of the SCI standard.

For every line the memory directory keeps associated tag bits. Some of these identify the first processor in the sharing list (called the head). Double links are maintained between other processors in the sharing list, using forward and backward pointers. The backward pointers support independent (and perhaps simultaneous) deletions of entries in the middle of the list, e.g., when a processor needs to free a cache line for use by a different address.

4.1.5 Memory and cache tags

Memory tags include a lock bit, a 2-bit memory-state field, *mState*, and a 16-bit *forwId* field. With the basic memory model, which only supports the caching of apparently dirty data, these bits may be located in the data store (which is not used when the data are cached). The *forwId* field specifies the first node in the sharing list in terms of the 16 most-significant bits (*nodeId*) of an SCI address.

Each cache entry contains the 7-bit cache state, *cState*, and two 16-bit pointer fields, *forwId* and *backId*, which usually point to the adjacent sharing-list entries. The extra memory-tag and cache-tag storage represent overheads of approximately 4 % and 7 %, respectively. These tag bits are illustrated in figure 130.

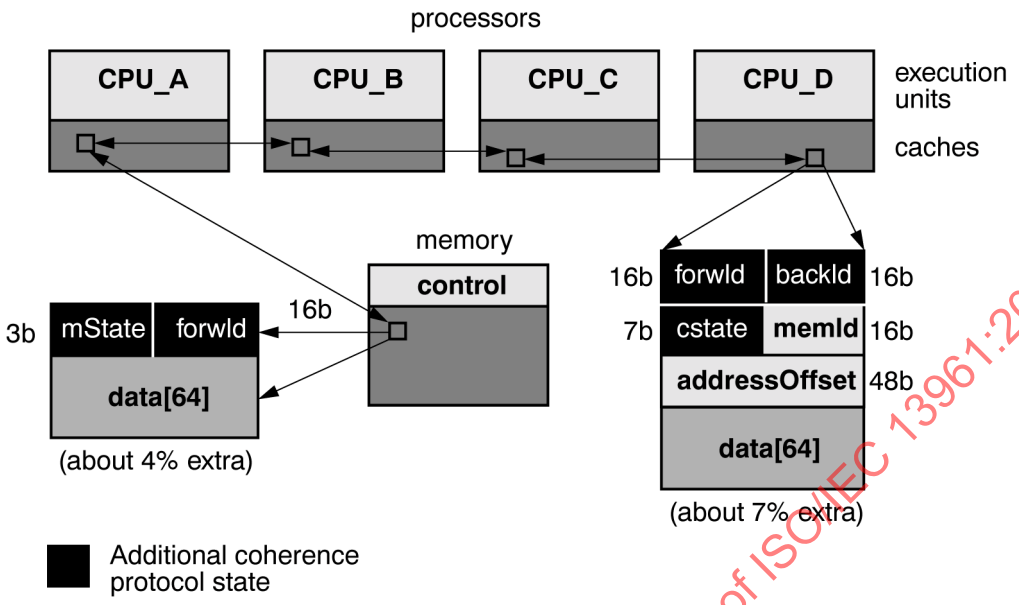


Figure 130 – SCI coherence tags (64-byte line, 64K nodes)

Each cached entry has an address that is partitioned into a 16-bit memory-controller identifier *memId* and 48 bits of *addressOffset*. For entries at the head of the list, the *backId* field is not needed, since the *memId* field is part of the line address. For these head-list entries, the *backId* pointer is not part of the basic sharing-list structure, but is used by an optional part of the coherence protocols.

SCI assumes a fixed 64-byte cache-line size, which is near optimal for most systems, for the following reasons:

1) *Small tag overhead.* The sizes of memory-directory and processor-entry tags are significantly less than the size of a line of data.
2) *Reasonable efficiency.* The 64-byte SCI transaction is relatively efficient; approximately two thirds of the consumed bandwidth is used for data.
3) *Uniformity.* The 64-byte size is shared by other bus standards (Futurebus+).

Having one fixed size dramatically simplifies the coherence protocols, which compensates for the use of a nonoptimal size on some systems. Although smaller line sizes could reduce the amount of false sharing (which can occur when two or more independent variables happen to be in the same line), smart compilers are a more effective solution to this problem.

4.1.6 Instruction-execution model

The cache-coherence protocols describe a set of actions used to change cache-line states. For a load instruction, the cache-line data must be converted to a readable state; for a store instruction, the cache-line data must be converted to an exclusive writeable state; for a flush instruction, the cache-line data must be returned to memory.

For this specification the processor's memory-access instructions are expected to have four phases: the allocate phase, the setup phase, the execute phase, and the cleanup phase. For example, the simplified C code of the following listing illustrates the four phases within a coherent store instruction.

```
/*              Listing 1: store_instruction illustration              */

void
ExecuteStore(ProcParameters *procPtr, AccessModes mode,
  Quads2 address, Byte *grBuf, int size)
{  CacheTags *cTPtr;
   int offset= address.Lo%64;
   cTPtr=FindLine(procPtr,mode,address);  /* Fetch matching entry */
   StoreSetup(procPtr,mode,address);      /* Setup cache-line state */
   Store(procPtr,mode,cTPtr,offset,
        grBuf,size);                      /* Execute phase */
   Cleanup(procPtr,mode,cTPtr);           /* Cleanup phase */
}
```

In this example, the allocate phase consists of `FindLine()`, which finds or fetches a cache-line entry for the addressed cache line. If a cache-line entry is found in a usable state (a cache hit), no transactions are generated.

The setup phase (which may involve the generation of multiple transactions) calls `StoreSetup()` to convert from the previous cache-line state to one of the instruction's usable cache states. For example, the setup phase of a load instruction would be used to convert a cache line from the state `INVALID` to one of the readable cache-line states.

The execute phase of an instruction calls Store() and might involve an immediate change of cache-line states. For example, the execute phase of a store instruction changes a modifiable cache-line entry (`ONLY_CLEAN`) to a modified cache-line entry (`ONLY_DIRTY`). The execute phase of a store instruction may also change a modifiable cache-line entry (`HEAD_DIRTY`) to a modified intermediate state (`HEAD_MODS`). Similarly, the execute phase of a flush instruction would mark the cache-line for flushing during the cleanup phase.

The cleanup phase of an instruction (which may involve the generation of multiple transactions) calls `Cleanup()` to change a transient cache-line state to one of the stable cache-line states. For example, after data in the sharing-list head is modified, the cleanup phase of a store instruction is responsible for purging the other sharing-list copies.

Processors may enforce weak or strong ordering constraints for the execution of memory-access instructions. Weak ordering constraints generally allow the pipelined execution of other instructions during the cleanup phase and strong ordering constraints do not. To support weak ordering constraints, the SCI C code updates a done code when the execute phase of an instruction completes. However, the details of how this affects other pipelined-instruction interlocks are beyond the scope of the SCI standard.

4.1.7 Coherence document structure

The coherence protocols support a rich set of interoperable performance enhancement options. These options have been designed so that nodes implementing different sets will interoperate correctly in all cases, but the enhanced performance the options offer may not be realized if they have not been implemented by all participating nodes.

The full set of options will probably not be used in initial implementations, but provides a rich set of design choices for customizing the protocols to meet specific system requirements. To simplify understanding of the cache-coherence protocols, three sets of implementation options are outlined in this overview: the minimal, a typical, and the full sets.

The *minimal set* can be used to maintain cache coherence in a trivial but correct way that has no provision for read sharing. This model could be useful for small multiprocessors where applications infrequently share data, and manage coherence of shared instruction pages by software.

The *typical set* has provisions for read sharing, robust recovery from errors, efficient read-only (fresh) data accesses, efficient DMA transfers, and local (noncoherent) data caching. This option set is likely to be implemented even in the first SCI systems.

The *full set* implements all of the defined options. In addition to the provisions of the typical set, the full set supports clean cache-line states, cleansing and washing of dirty cache-line states, pairwise sharing, and QOLB. This more complex option set is expected to be implemented on general-purpose processors as implementors gain experience with SCI.

These three option sets are described in 4.2 to 4.5.

4.2 Coherence update sequences

4.2.1 List prepend

To illustrate the coherence protocol components, consider the conversion of a sharing list from a one-entry (ONLYP_DIRTY) list to a two entry list (HEAD_EXCL and TAIL_STALE). If the entry in CPU_B is initially invalid, a modifiable cache line must be fetched from memory before the instruction can be executed. The *mread64* (coherent memory read) transaction consists of request and response components; the memory accepts the request Q1, performs an update action (A1) to update its cache-tag state and pointers, and returns the response S1.

The memory-tag-update action leaves the memory tag pointing to CPU_B and the old pointer value (which identifies CPU_A) is returned to CPU_B in the transaction response S1. While waiting for S1, CPU_B is left in the PENDING state. This sequence is illustrated in figure 131, using a shaded line (from requester to responder) to specify transactions and a solid line to specify sharing-list links.

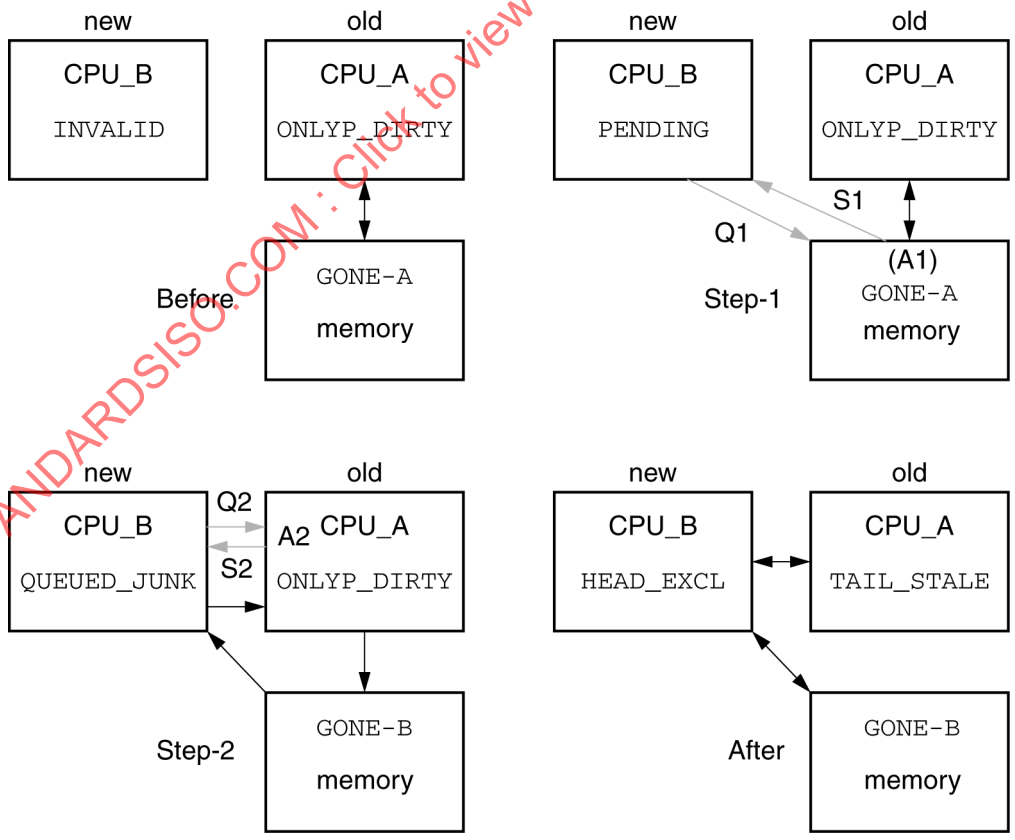


Figure 131 – Prepend to ONLYP_DIRTY (pairwise capable)

The response S1 returns to CPU_B the previous state of the memory-line and a pointer to CPU_A. Based on these values, CPU_B initiates a *cread64* (coherent cache read) transaction to the old sharing-list head (CPU_A). The old sharing-list head accepts the request subaction (Q2), performs an update action (A2), and returns a response subaction (S2). The update action (A2) leaves CPU_A in the TAIL_STALE state (tail of the list, data are stale and unusable). The processing of the response S2 leaves CPU_B in the HEAD_EXCL state (head of the list, data are exclusive and modifiable).

In this example the Q1 request is the first half of an *mread64* transaction; if the data had been uncached, this would have returned 64 bytes of data from memory. The *mread64* request is 16 bytes long; it contains the 16-bit nodeId of the memory controller responder (*resId*), the 7-bit transaction command (*cmd*), the 16-bit nodeId of the requester (*reqId*), and a 48-bit address offset (*A00,A16,A32*), which includes the 6-bit memory-update operand (*mop*). These transaction components, which are many of the request subaction fields, are illustrated in figure 132.

The S1 response is the second subaction in the *mread64* transaction. When data are unavailable, this response subaction returns the 4-bit storage-status (*sStat*), which is used to report data-storage and transmission errors, the 8-bit memory status (*mStat*), which is used to return the previous memory-tag state, a 16-bit forward pointer (*forwId*), which points to the previous sharing-list head, and a 16-bit reserved field (for future extensions to the coherence protocols). The *sStat* field is expected to be used for reporting ECC errors in RAM; the *mStat* field indicates how the sharing list was previously owned.

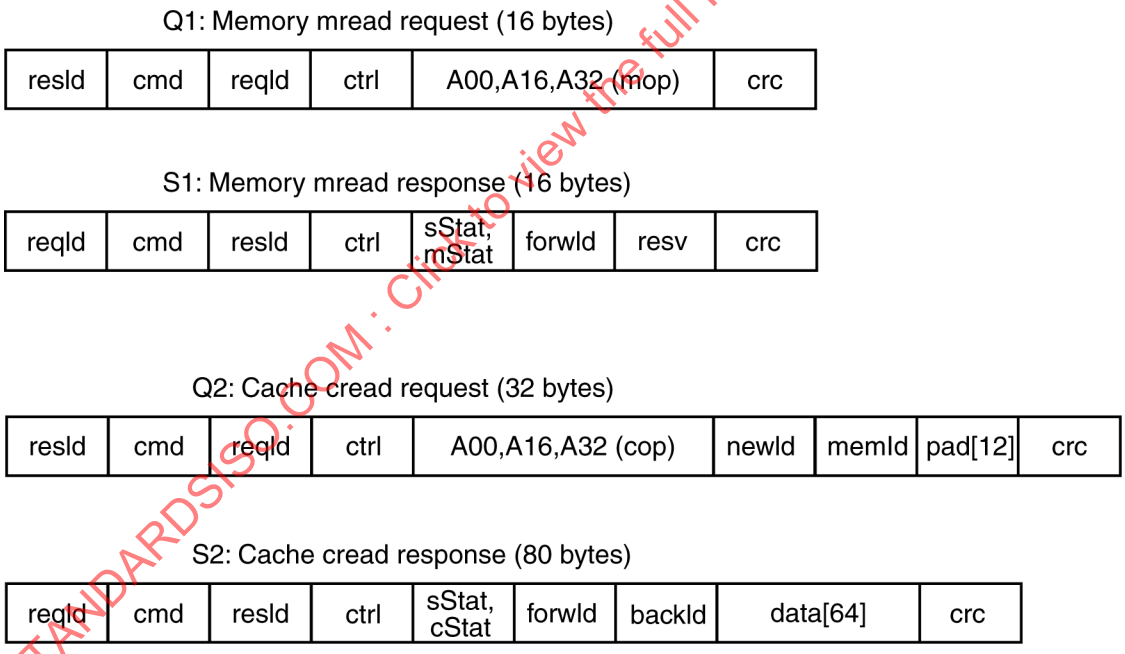


Figure 132 – Memory *mread* and cache-extended *cread* components

The Q2 request is the first subaction of an extended *cread64* transaction that requests data from the remote cache (CPU_A). The extended request is 32 bytes long; the first half contains the 16-bit address of the cache responder (*resId*), the 7-bit transaction command (*cmd*), the 16-bit address of the cache requester (*reqId*), the 48-bit address offset (*A00,A16,A32*), which includes the 6-bit cache-update operand (*cop*), and 16 bytes of extended-header information.

The extended portion of the header contains an unused 16-bit identifier (*newId*), a 16-bit memory identifier (*memId*), which provides the address of the memory controller, and 12 bytes of pad data. The pad data, which extends the packet to a uniform multiple of 16 bytes, contains reserved fields.

The S2 response is the second subaction in the *cread64* transaction. This response subaction returns the 4-bit storage status (*sStat*), which is used to report data-storage and transmission errors, the 8-bit cache status (*cStat*), which is used to return the previous cache-tag state, a 16-bit forward pointer (*forwId*), which points to the next sharing-list entry, a 16-bit backward pointer (*backId*), which points to the previous sharing-list entry, and 64 bytes of (optional) data.

Note that the memory controller can always add a requesting node to the *pending* queue, and ownership is then passed sequentially to the new heads of the queue. The addition of new sharing-list entries is thus performed in FIFO order, as defined by the arrival of coherent requests at the memory controller. Note that ownership implies that the cache-line's data may be immediately modified, although some delayed purging may be required after the data are modified.

4.2.2 List-entry deletion

To illustrate other coherence protocol components, consider the deletion of the initial sharing-list entry, which occurs when the cache-entry storage is needed for another cache-line address. If the cache line in CPU_B is initially in a head/exclusive state (HEAD_EXCL), an extended *cwrite64* transaction (see table 27) is used to return the data from CPU_B to CPU_A. The *cwrite64* transaction consists of request and *response subactions*; the remote cache accepts the request Q3, performs an update action (A3) to update its cache-tag state and pointers, and returns the response S3. The cache-tag-update action leaves the cache tag of CPU_A in the ONLYP_DIRTY state, as illustrated in figure 133.

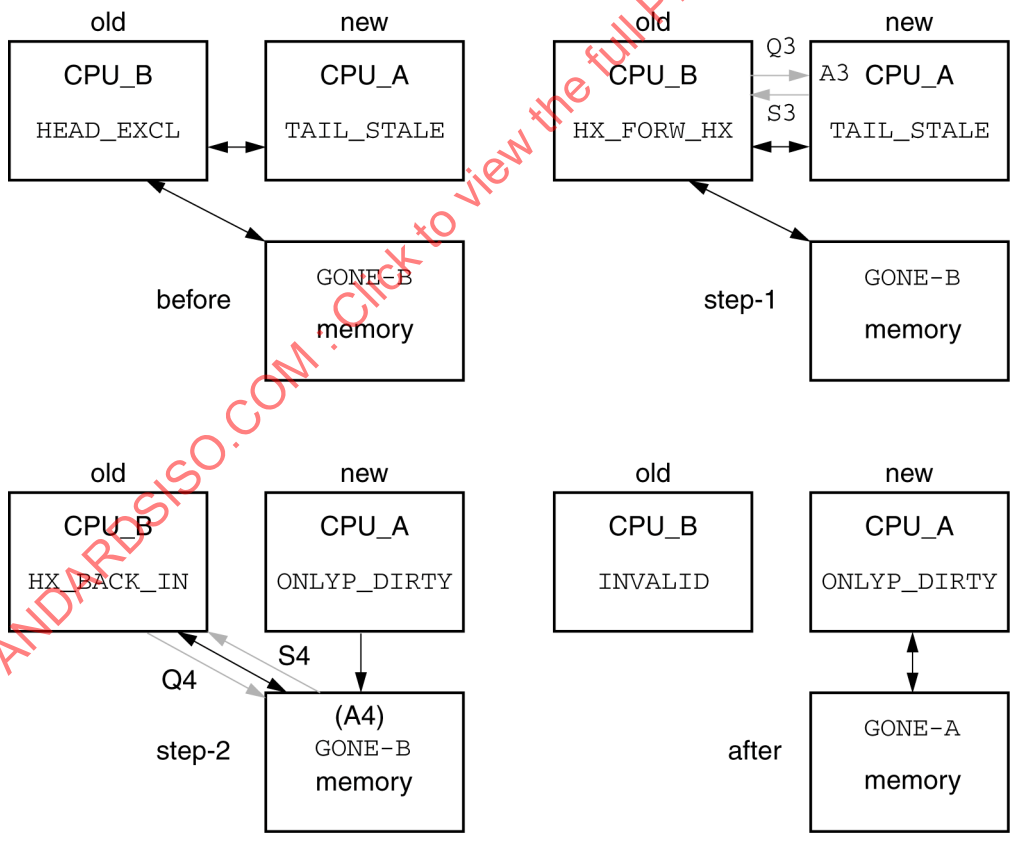


Figure 133 – Deletion of head (and exclusive) entry

The response to CPU_B returns the cache's previous cache-line state and pointer value. Based on these values, CPU_B initiates an extended *mread00* transaction to memory.

The memory accepts the request subaction (Q4), performs an update action (A4), and returns a response subaction (S4). The update action (A4) changes the memory pointer to point to the new sharing-list head (`CPU_A`). The processing of the response Q4 leaves the cache-line entry at `CPU_B` in the `INVALID` state, so it may be used to cache other cache-line addresses.

In this case, the extended *cwrite64* request (Q3) is the first subaction of a *cwrite64* transaction. The *cwrite64* request is 96 bytes long; it contains the 16-bit nodeId of the cache responder (*resId*), the 7-bit transaction command (*cmd*), the 16-bit nodeId of the cache requester (*reqId*), a 48-bit address offset (*add_offset*), which includes a 6-bit cache-update operand (*cop*), and 16 bytes of extended-header information, as shown in figure 134.

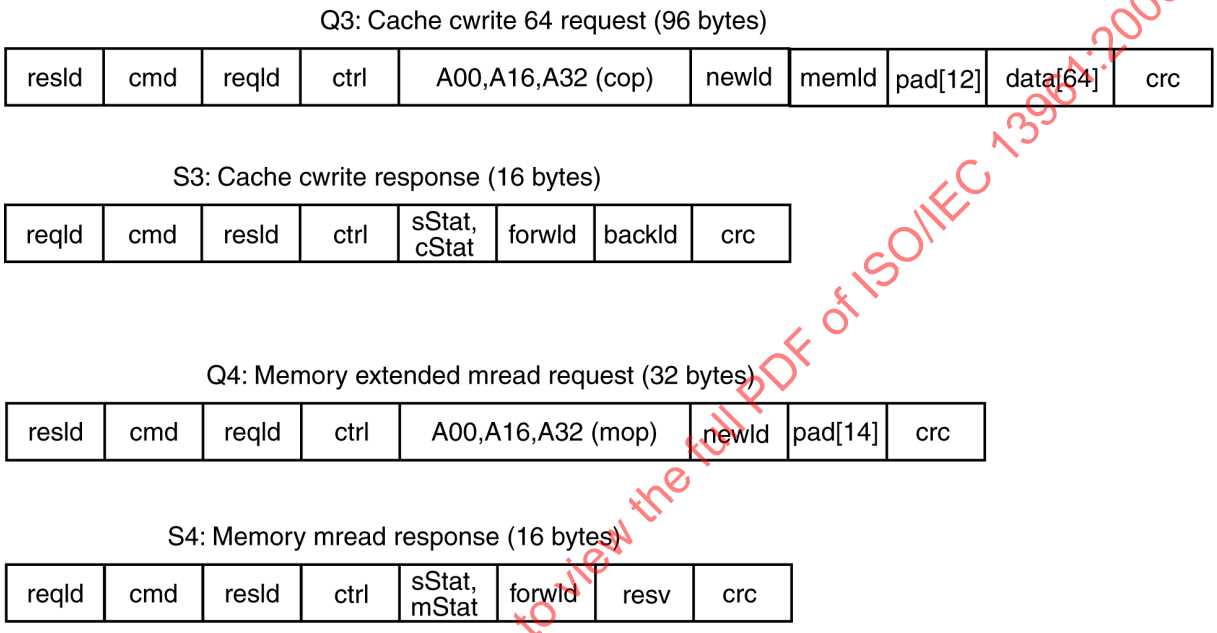


Figure 134 – Cache *cwrite64* and memory-extended *mread* components

The extended header contains an unused 16-bit identifier (*newId*), a 16-bit memory identifier (*memId*), which identifies the address of the memory controller, and 12 bytes of pad data. The pad data, which extends the packet to a uniform multiple of 16 bytes, contains reserved fields. The Q3 transaction is called an *extended cwrite64* because an extended 32-byte header is required to hold the extra *memId* value.

The *cwrite64* response (S3) is the second subaction in the *cwrite64* transaction. This response subaction returns the 4-bit storage status (*sStat*), which is used to report data-storage and transmission errors, the 8-bit cache status (*cStat*), which is used to return the previous cache-tag state, and two 16-bit pointers (*forwId* and *backId*), which point to the previous and following sharing-list entries. The *sStat* field is expected to be used for reporting ECC errors in cache-RAM; the *cStat* field indicates how the sharing list entry was previously used.

The Q4 request is the first subaction of an extended *mread00* transaction. The extended request is 32 bytes long; its first half contains the 16-bit nodeId of the cache responder (*resId*), the 7-bit transaction command (*cmd*), the 16-bit nodeId of the cache requester (*reqId*), the 48-bit address offset (*A00,A16,A32*), which includes a 6-bit memory-update operand (*mop*), and 16 bytes of extended-header information.

The extended header contains a 16-bit new-cache-nodeId identifier (*newId*), which identifies the new sharing-list owner, and 14 bytes of pad data. The pad data, which extends the packet to a uniform multiple of 16 bytes, contains reserved fields. The Q4 transaction is called an *extended mread00*, because an extended 32-byte header is required to hold the extra *newId* value. Note that *control* operations (which transfer no data) are called zero-length reads (*mread00* or *cread00*, when accessing memory or cache respectively).

The S4 response is the second subaction of an extended *mread* transaction. This response subaction returns the 4-bit storage-status (*sStat*), which is used to report data-storage and transmission errors, the 8-bit memory status (*mStat*), which is used to return the previous memory-tag state, a 16-bit forward pointer (*forwId*), which points to the next sharing-list entry, and a 16-bit reserved field.

4.2.3 Update actions

The responder's processing of each coherent request (Q1-Q4) initiates an indivisible action (A1-A4) in the responder. These actions conditionally update the responder's tag state, based on the parameters provided within the request subaction packet.

For this example, the subaction Q1 contains the memory-command value CACHE_DIRTY. The memory's processing of this command (A1) normally converts the memory state from HOME to GONE (if the data was previously uncached) and changes the *forwId* value to point to the sharing-list head.

Similarly, the subaction Q2 contains the cache-command value COPY_STALE. The (CPU_A) cache's processing of this command (A2) normally converts the cache state from ONLYP_DIRTY to TAIL_STALE, simultaneously changing the *backId* pointer in CPU_A to point to the requester (*reqId*).

Some of the update actions (A3 and A4) are conditional; these two update actions are nullified unless either the tag's *backId* or *forwId* value matches the request subaction's *reqId* field (the identity of the requester). These conditional actions make it possible to maintain consistency even though any or all processors may be concurrently trying to change the pointers and states in various ways. These (simplified) update actions are summarized in table 19.

Table 19 – Memory and cache update actions

	initial states			final states		
update_command	**state**	**forwId**	**backId**	**state**	**forwId**	**backId**
A1:CACHE_DIRTY	HOME	*forwId*	–	GONE	*reqId*	–
A1:CACHE_DIRTY	GONE	*forwId*	–	GONE	*reqId*	–
A2: COPY_STALE	ONLYP_DIRTY	*forwId*	*backId*	TAIL_STALE	*reqId*	*reqId*
A3: NEXT_EHEAD	TAIL_STALE	*forwId*	*backId*	ONLYP_DIRTY	*forwId*	*backId*
A4: PASS_HEAD	GONE	*forwId*	–	GONE	*newId*	–

Note that table 19 is an oversimplified update-action table; other possible initial states have not been included in the table. This simplified description does not include the effects of cache-line locks, which are used to block most memory- and cache-update actions during an error-recovery process.

Although it would be possible to specify the memory-update and cache-update actions as state-transition tables, particularly in these simplified cases, they have been specified by executable C routines instead. This simplifies the document considerably and provides a convenient mechanism for testing the specification by computer simulations.

The specification code includes tests of memory-tag and cache-tag lock bits. Also, a variety of implementation options is specified by execution-time conditional code execution. Execution-time conditionals are used rather than compile-time ones to make it easier to test the interactions of nodes that implement differing sets of options.

4.2.4 Cache-line locks

Error-recovery considerations have heavily influenced the design of the coherence protocols. During error recovery, software-based protocols utilize lock bits (one per cache line) to

stabilize the cache-line status. The error-recovery process (which is beyond the scope of this standard) is expected to proceed as follows:

1) *Lock lines.* The lock bits in the affected memory line and matching cache lines are set, to inhibit spontaneous state changes during the recovery process.
2) *Copy.* The currently cached (and now locked) entries are copied to a memory-resident table. After being copied, the previously cached entries are invalidated.
3) *Recovery.* Process the newly created memory-resident sharing-list table, in an attempt to recover the cached (and possibly modified) line. The recovery process completes with one of the following status codes:
   a) *Corrupted.* The sharing-list structure was corrupted (hardware failure).
   b) *Unrecoverable.* The possible locations for the most recently modified data are not unique; system software is expected to recover from a previous checkpoint.
   c) *Recovered.* The data has not been modified, or the most recently modified copy of the line was located. If modified, the dirty data was returned to memory.
4) *Unlock memory.* The memory line is unlocked, returning it to the HOME state.

Note that the *unrecoverable* status is only expected when the option called POP_ROBUST (which increases the complexity and latency of returning a dirty cache line copy) is not implemented.

To implement error recovery, there is one lock bit for each cache line. When set by a LOCK_SET command, the lock bit disables most changes to the associated state and data. Processors are expected to bypass the cache (to avoid generating additional, possibly dependent, errors) when executing the recovery software routines.

Except for other LOCK_SET and LOCK_CLEAR commands, accesses to these locked cache-line addresses return an error status in the response subaction *status.sStat* field. For the specialized LOCK_SET and LOCK_CLEAR commands, the error status is not returned, but the update-action status is returned in the response transaction's status fields.

4.2.5 Stable sharing lists

Each of the stable sharing-list states is defined by the state of the memory, *mState*, and the states of the entries in the sharing list, *cState*. In normal operation, the memory state is either HOME (no sharing list), FRESH (read-only sharing list), GONE (sharing list can be modified), or WASH (transition from GONE to FRESH). The minimal protocol uses the HOME and GONE states, the typical protocol uses only the HOME, FRESH, and GONE states, and the full coherence protocols use all of the memory-directory states. The stable and semistable memory-tag states are summarized in table 20.

Table 20 – Stable and semistable memory-tag states

name	description
HOME	no sharing list
FRESH	sharing-list copy is the same as memory
GONE	sharing-list copy may be different from memory
WASH [1)]	transitional state (GONE to FRESH)
[1)] WASH is a semistable state.	

The sharing-list state names have two components. The first component specifies the location of the entry in a multiple-entry sharing list (HEAD, MID, or TAIL), or identifies the only entry in the sharing list (ONLY). The second component specifies the entry's caching properties (FRESH, CLEAN, DIRTY, VALID, STALE, etc.). The stable and semistable cache-tag states are summarized in table 21.

Table 21 – Stable cache-tag states

name	description
ONLY_DIRTY	only one, writeable, modified
ONLYP_DIRTY	only one, writeable, modified (pairwise capable)
ONLY_CLEAN	only one, writeable, unmodified
ONLY_FRESH	only one, convertable, unmodified
HEAD_DIRTY	head of several, purgeable, modified
HEAD_CLEAN	head of several, purgeable, unmodified
HEAD_WASH [1)]	like HEAD_CLEAN (but list is in transition to HEAD_FRESH)
HEAD_FRESH	head of several, changeable, unmodified
MID_VALID	middle of many, readable, modified
MID_COPY	middle of many, readable, unmodified
TAIL_VALID	tail of several, markable, modified
TAIL_COPY	tail of several, markable, unmodified
HEAD_EXCL	head of two (exclusive), writeable, modified
HEAD_VALID	head of two (shared), markable, modified
HEAD_STALE0	head of two (stale), transferable, previously valid data
HEAD_STALE1	head of two (stale), transferable, previously valid data
TAIL_EXCL	tail of two (exclusive), writeable, modified
TAIL_DIRTY	tail of two (shared), purgeable, modified
TAIL_STALE0	tail of two (stale), transferable, previously valid data
TAIL_STALE1	tail of two (stale), transferable, previously valid data
ONLYQ_DIRTY	only one, writeable, modified (QOLB history)
HEAD_IDLE	head of several, transferable, waiting for data
MID_IDLE	middle of many, transferable, waiting for data
ONLY_USED	only one, writeable, lock set, none waiting
HEAD_USED	head of two, writeable, lock set, none waiting
HEAD_NEED	head of two, writeable, lock set, other is waiting
TAIL_IDLE	tail of two, transferable, waiting for data
TAIL_USED	tail of two, writeable, lock set, none waiting
TAIL_NEED	tail of several, writeable, lock set, others waiting

[1)] HEAD_WASH is a semistable state

NOTES

several	two or more sharing-list entries
many	three or more sharing-list entries
changeable	data may be read, but not written until memory is informed and rest of list is purged
convertable	data may be read, but not written until memory is informed
markable	data may be modified after other copy has been marked stale
purgeable	data may be read, but not written until rest of list is purged or marked stale
readable	data may be read immediately
transferable	data may not be read or written, until fetched from another entry
writeable	data may be read or written
unmodified	data are the same as memory
modified	data could be different from memory

Since the head normally administers the return of dirty data to memory, it differentiates between FRESH (must be the same as memory) and the other (can modify without informing memory) states. The protocols generate the stable sharing-list states shown in table 22.

Table 22 – Stable sharing lists

mem	head	(other)	tail	description
HOME	–	–	–	none or noncoherent copies
FRESH	ONLY_FRESH	–	–	convertable unmodified copy
FRESH	HEAD_FRESH	MID_BOTH	TAIL_BOTH	changeable unmodified copies
GONE	ONLY_CLEAN	–	–	writeable unmodified copy
GONE	ONLY_DIRTY	–	–	writeable modified copy
GONE	HEAD_DIRTY	MID_VALID	TAIL_VALID	purgeable modified copies
GONE	HEAD_BOTH [1)]	MID_BOTH [1)]	TAIL_BOTH [1)]	purgeable unmodified copies
GONE	HEAD_WASH [2)]	MID_VALID	TAIL_VALID	purgeable unmodified copies
(pairwise-sharing option)				
GONE	ONLYP_DIRTY	–	–	like ONLY_DIRTY, pairwise capable
GONE	HEAD_EXCL	–	TAIL_STALE0	writeable modified copy
GONE	HEAD_EXCL	–	TAIL_STALE1	writeable modified copy
GONE	HEAD_VALID	–	TAIL_DIRTY	purgeable modified copies
GONE	HEAD_STALE0	–	TAIL_EXCL	writeable modified copy
GONE	HEAD_STALE1	–	TAIL_EXCL	writeable modified copy
(QOLB option)				
GONE	ONLYQ_DIRTY	–	–	like ONLYP_DIRTY, QOLB history
GONE	ONLY_USED	–	–	writeable modified copy, locked
GONE	HEAD_USED	–	TAIL_STALE0	writeable modified copy, locked
GONE	HEAD_USED	–	TAIL_STALE1	writeable modified copy, locked
GONE	HEAD_STALE0		TAIL_USED	writeable modified copy, locked
GONE	HEAD_STALE1	–	TAIL_USED	writeable modified copy, locked
GONE	HEAD_NEED	–	TAIL_IDLE	writeable modified copy, locked, waiting
GONE	HEAD_IDLE	–	TAIL_NEED	writeable modified copy, locked, waiting
GONE	HEAD_IDLE	MID_IDLE	TAIL_NEED	writeable modified copy, locked, waiting

1) When heterogeneous options are implemented, unmodified lists may contain the following:

HEAD_BOTH either HEAD_CLEAN or HEAD_DIRTY;

MID_BOTH either MID_COPY or MID_VALID;

TAIL_BOTH either TAIL_COPY or TAIL_VALID.

2) Semistable state, transitioning between HEAD_DIRTY / HEAD_CLEAN and HEAD_FRESH states.

The processors within the sharing lists may implement different sets of optional cache capabilities. Thus, an entry at the head of the list may know that a cache line is fresh (HEAD_FRESH), while the other sharing-list entries believe the sharing-list could be dirty (MID_VALID or TAIL_VALID).

Note that two types of stale states (STALE0 and STALE1) are provided. The extra sequence bit that distinguishes these two states is needed to support software-based fault recovery protocols that are invoked after transmission failures.

4.3 Minimal-set coherence protocols

4.3.1 Sharing-list updates

The minimal set of coherence options supports the conversion of an invalid cache-line to the (modifiable) ONLY_DIRTY state. Fetching of read-only data (such as ONLY_FRESH) and support of multiple-entry sharing lists (HEAD_DIRTY, MID_VALID, TAIL_VALID) are not essential for loading or storing data, and can thus be viewed as optional performance enhancements. However, some additional states (ONLY_FRESH and TAIL_VALID) are needed in order to be compatible with the optional performance enhancements.

4.3.2 Cache fetching

Initially, memory is in the HOME state and all cache entries are INVALID (have no usable data). The sharing-list creation begins at the cache, where an entry is changed from the INVALID to the PENDING state. A dirty cache-line copy is fetched (1) from memory using an mread64.CACHE_DIRTY transaction, which leaves a newly created cache line in the ONLY_DIRTY state. This sequence is illustrated in figure 135, using a shaded line (from requester to responder) to specify transactions and a solid line to specify sharing-list links.

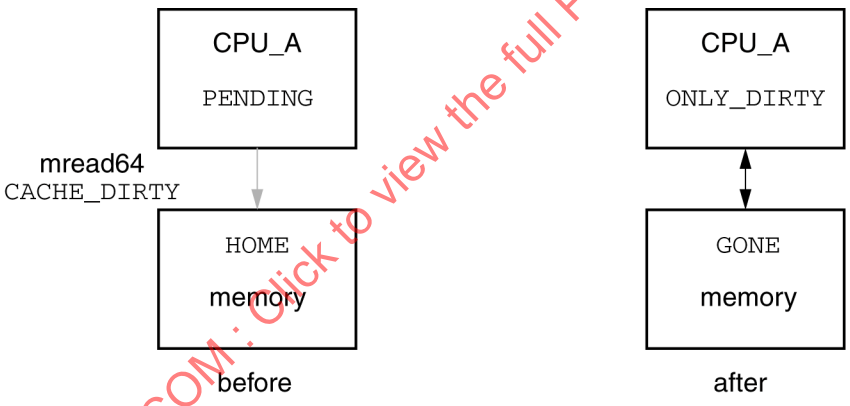


Figure 135 – ONLY_DIRTY list creation (minimal set)

Modifications of cache lines in the ONLY_DIRTY state can be performed immediately, without changing the cache-line state.

For subsequent accesses, the memory state is GONE and the head of the sharing list has the (possibly dirty) data. A request for data from memory provides (1) a sharing-list pointer and the new requester then prepends (2) to the old sharing-list head to get the data. After prepending has completed, the old sharing-list entries are invalidated (3) by the new head. These steps are illustrated in figure 136.

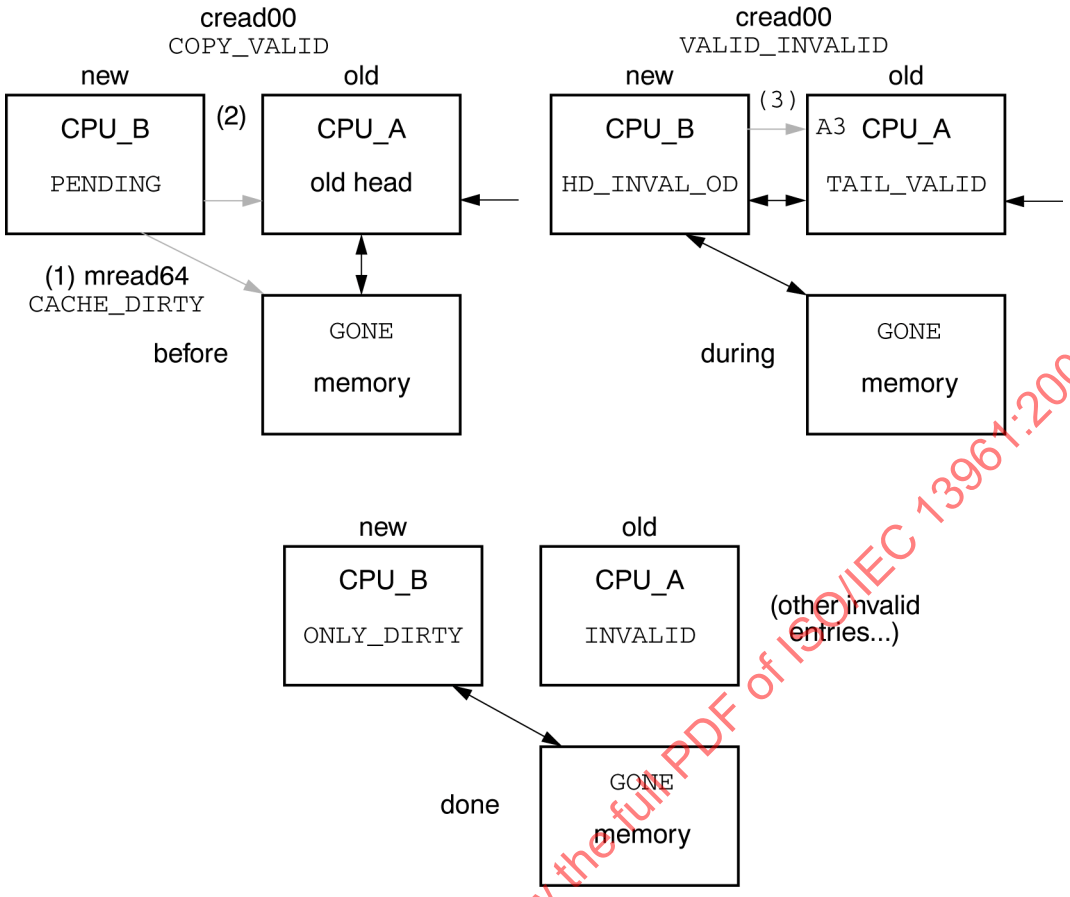


Figure 136 – GONE list additions (minimal set)

It might appear that the prepend and invalidation steps could be combined into a single transaction that returns the (possibly dirty) data and leaves the old head in the invalid state. However, separate prepend and invalidate transactions are needed for recovering from transmission errors. The performance penalty of this extra transaction can be avoided by implementing the pairwise-sharing option.

The minimal protocols need to interoperate with other options as well, and the typical protocols may leave the memory in the FRESH state. In this case, a new requester receives (1) the data directly from memory and invalidates (2) the old sharing-list entries, as illustrated in figure 137.

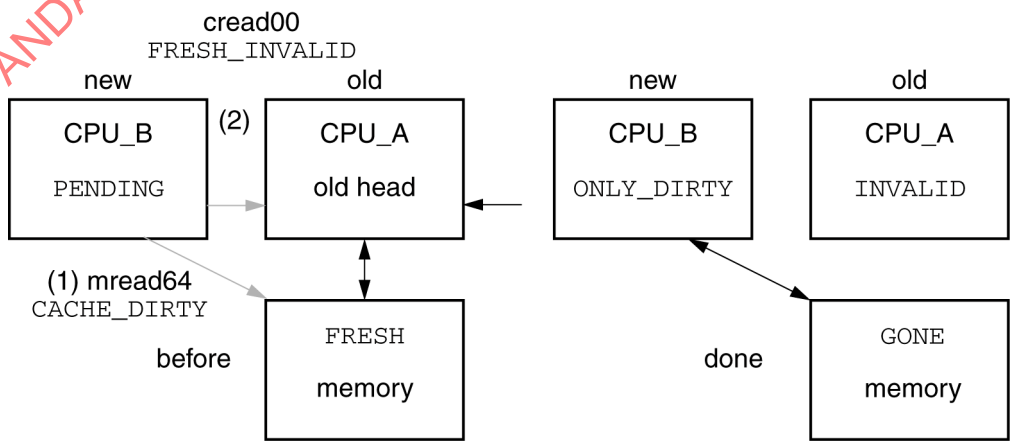


Figure 137 – FRESH list additions (minimal set)

An old head may also be in a PENDING state, in the process of adding itself back into the same sharing-list. In such cases, the transaction status returns the PENDING state from the next pending-queue entry. The new head's prepend transaction is retried until the old head's pending status changes.

4.3.3 Cache rollouts

An ONLY_DIRTY sharing list may be collapsed, e.g., when the cache-line storage is needed for use by another cache-line address (cache-line rollout). In the case of an ONLY_DIRTY entry, only one transaction (1a) is needed to collapse the sharing list. This transaction returns the dirty data to memory and updates the memory-tag state (from GONE to HOME), as illustrated in figure 138. To be interoperable with other options, the minimal option returns ownership of an ONLY_FRESH line to memory (1b) in a similar way.

The memory-directory update is nullified if the directory points to a previous or to a new sharing-list head. In this case, memory is polled until the sharing-list ownership is returned to CPU_A or until the cache-line in CPU_A is invalidated by another prepending processor.

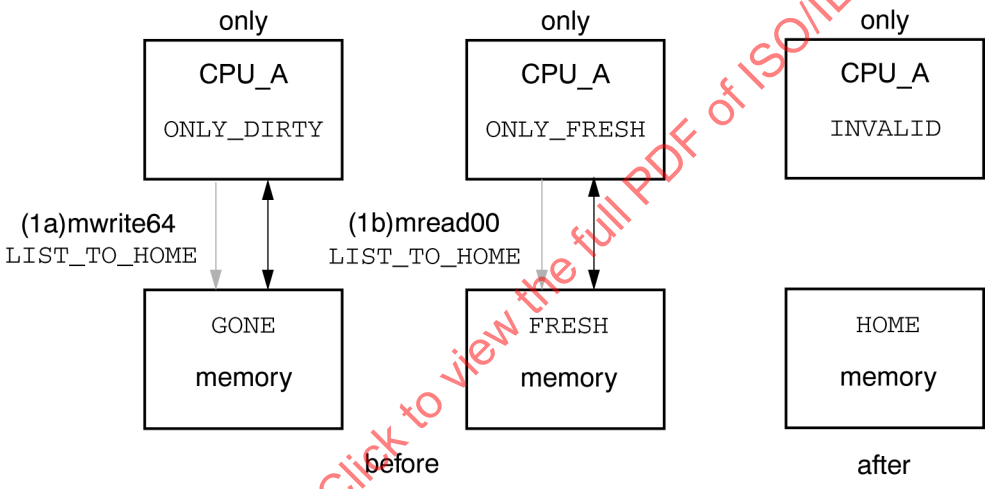


Figure 138 – Only-entry deletions

Recovery from an arbitrary number of detected transmission errors is not guaranteed when a single write transaction is used to collapse an ONLY_DIRTY sharing list. If one transaction is used to simultaneously return ownership and data, several transmission errors could leave the sharing list with one entry in the PENDING state and two entries in the OD_RETN_IN state. Although one of the OD_RETN_IN lines is known to have the valid data, it cannot be determined which one has the dirty copy and which one has a stale copy. To reliably return dirty data, one transaction is needed to cleanse the cache line (convert from ONLY_DIRTY to ONLY_CLEAN) and another is needed to convert from ONLY_CLEAN to INVALID, as described in 4.4.

Although the TAIL_VALID and ONLY_FRESH states are not directly generated by the minimal protocols, these states may be created after a more complex node prepends itself to an ONLY_DIRTY list. When deleting itself (1), a TAIL_VALID entry is converted into an intermediate state (called TV_BACK_IN), and one sharing-list transaction is used to delete the entry from the list, as illustrated in figure 139.

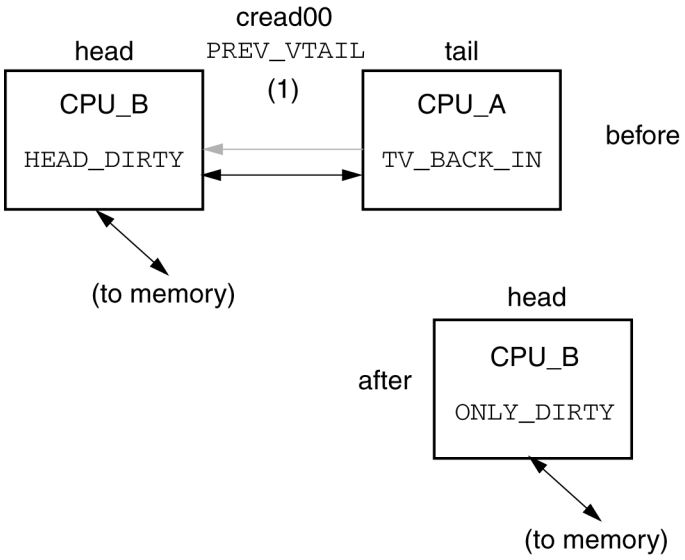


Figure 139 – Tail-entry deletions

Since the linked list is distributed and doubly linked, multiple entries can be deleting themselves concurrently. To ensure forward progress when adjacent deletions are initiated concurrently, the entry closest to the tail has priority and is deleted first.

4.3.4 Instruction-execution model

For efficient cache operation, a processor must communicate the nature of its access to data as well as the address of the data. Some processors at present lack the appropriate instructions for this and must simulate them by using special addresses or instruction sequences. Generic instructions that provide the needed information are assumed in the following.

The processor is expected to check and change cache-line states before and after instructions are executed. These checks and changes are modelled by the cache-execute routines listed in table 23.

Table 23 – MinimalExecute Routines

name	generated by the execution of
`MinimalExecuteLoad()`	a *load* memory-access instruction
`MinimalExecuteStore()`	a *store* memory-access instruction
`MinimalExecuteFlush()`	the *global flush* cache-control instruction (which collapses the sharing list)
`MinimalExecuteDelete()`	the *local flush* cache-control instruction (which deletes the local cache entry)
`MinimalExecuteLock()`	the *fetch&add*, *compare&swap*, and *mask&swap* instructions
NOTE The `MinimalExecuteLoad()` routine is equivalent to the `FullExecuteLoad()` routine, with the proper set of option bits. However, separate routines are provided so that this basic functionality is not obscured by the generality of the `FullExecuteLoad()` routine (which documents all options).	

4.4 Typical-set coherence protocols

4.4.1 Sharing-list updates

The typical set of coherence options supports the sharing of fresh or dirty data and provides special DMA read and write optimizations. This is a useful set of options that efficiently supports the sharing of read-only instructions/data as well as read/write data. This option set better illustrates the complexity of a typical implementation. Note that implementations are free to select other subsets of the coherence options, which might include fewer, more, or alternative options.

4.4.2 Read-only fetch

Initially, memory is in the HOME state and all caches are INVALID. When fetching a read-only copy, the sharing-list creation begins at the cache, where an entry is changed from the INVALID to the PENDING state, and an mread64.CACHE_FRESH transaction is generated to obtain a coherently cached copy. The read updates (1) the memory-directory state (from HOME to FRESH), and the new entry state is changed accordingly (from PENDING to ONLY_FRESH), as illustrated in figure 140.

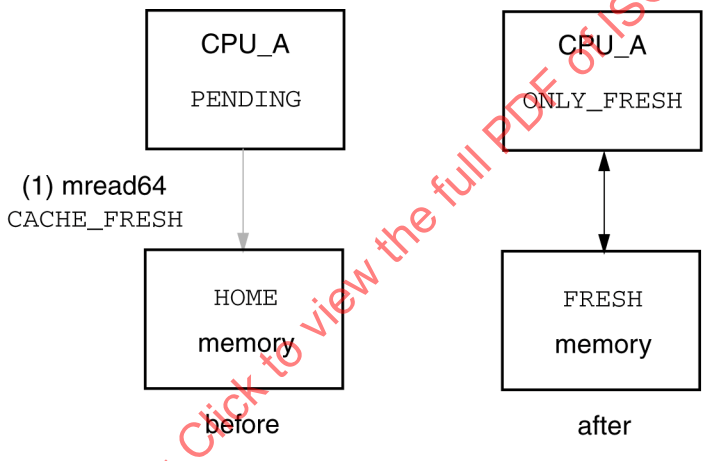


Figure 140 – FRESH list creation

Leaving the memory in a FRESH state minimizes the memory-access latencies for subsequent reads, since FRESH data can be provided by memory before the new sharing-list head attaches to the existing sharing list.

For subsequent accesses, the memory state is FRESH and the head of the sharing list has the unmodified data. When read-only data are accessed (1), fresh data are returned from memory and the new requester then attaches (2) to the old sharing-list head. These steps are illustrated in figure 141 for an mread64.CACHE_FRESH request when memory is in the FRESH state.

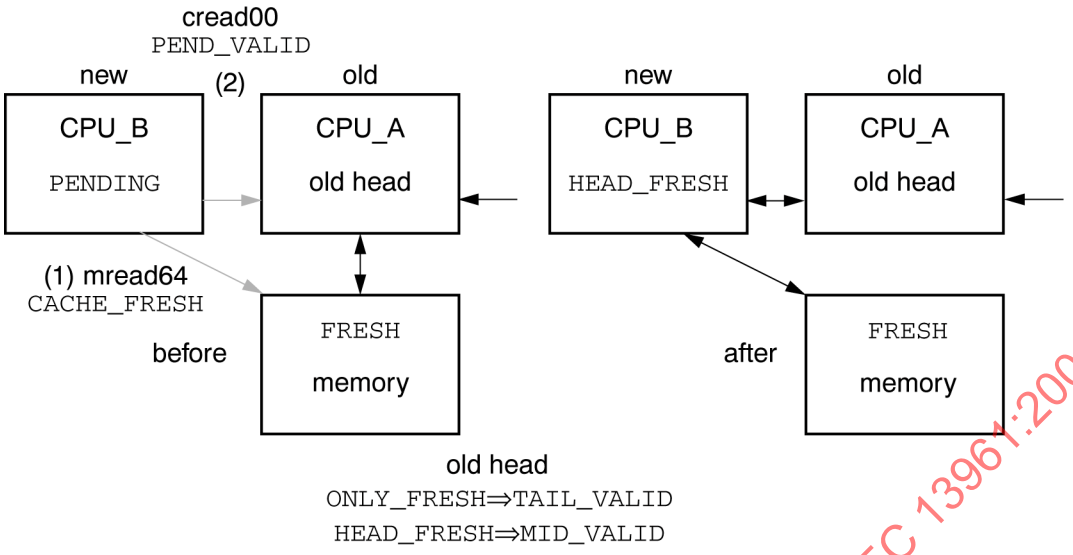


Figure 141 – FRESH addition to FRESH list

When the memory state is GONE, the head of the sharing list has the (possibly modified) data. The fresh data that is requested (1) cannot be returned from memory, but the dirty sharing-list copy is returned (2) when the new requester is attached to the old sharing-list head. These steps are illustrated in figure 142, for an mread64.CACHE_FRESH request when memory is in the GONE state.

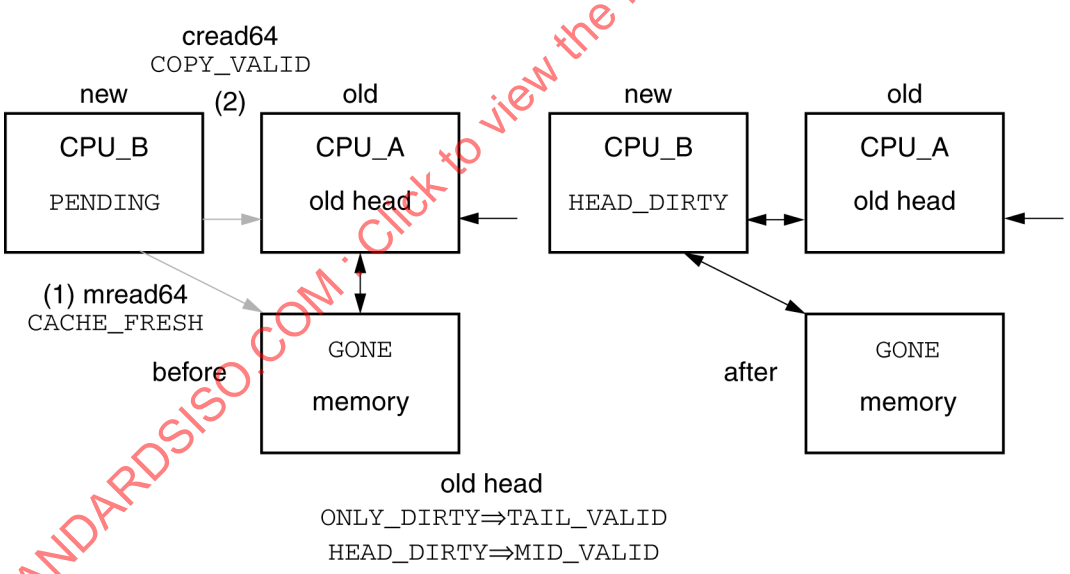


Figure 142 – FRESH addition to DIRTY list

The final state of the old sharing-list head is a function of the old head's initial state. The state of the new sharing-list head is HEAD_DIRTY. The states of the other mid and tail entries are unaffected by sharing-list additions.

4.4.3 Read-write fetch

If a later write is expected, a data-cache miss may be designed to fetch a modifiable (but not yet modified) copy. In this case, the read64.CACHE_CLEAN transaction is used (1) to fetch modifiable (but not immediately modified) data from memory. A FRESH memory state returns its data before the memory-tag state is changed to the GONE state. After prepending (2) to the old sharing list, the sharing list is left in the HEAD_DIRTY state, as illustrated in figure 143.

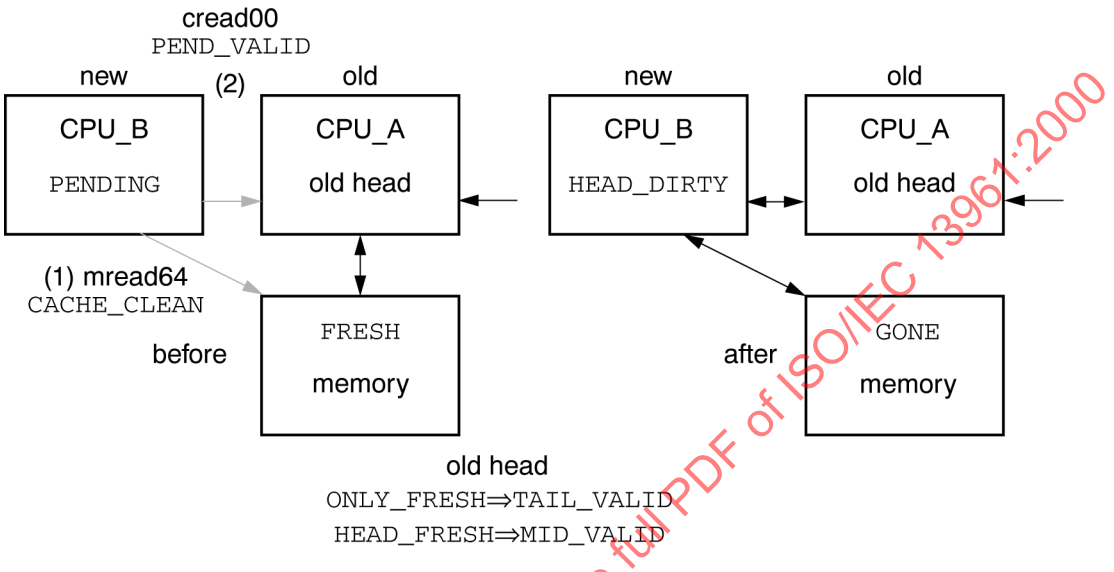


Figure 143 – DIRTY addition to FRESH list

The read64.CACHE_CLEAN transaction could access (1) a GONE memory state. In this case, the memory state is unchanged and no data are returned. The dirty data are eventually returned (2) when attaching to the old sharing list, as illustrated in figure 144.

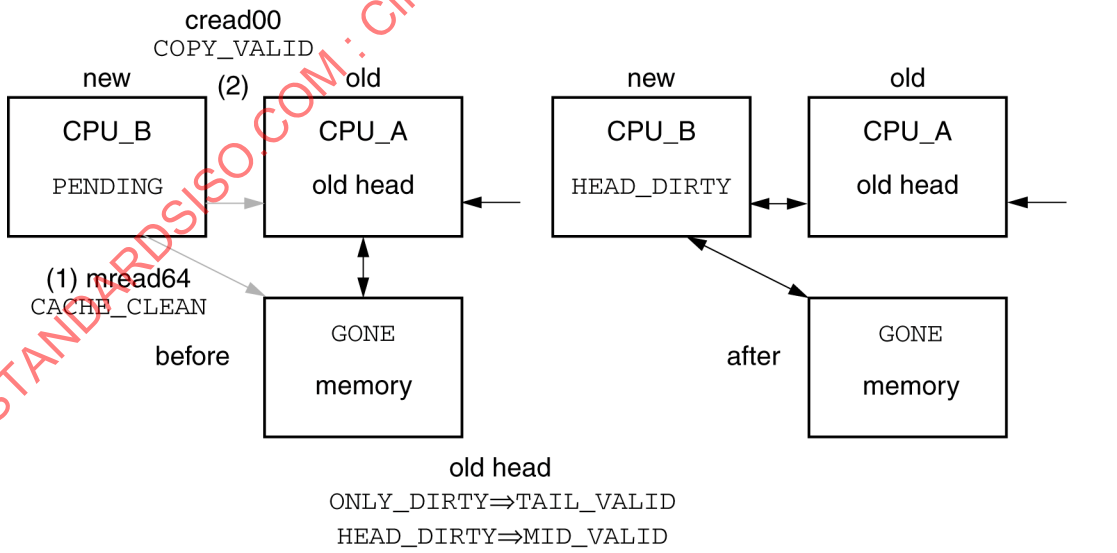


Figure 144 – DIRTY addition to DIRTY list

4.4.4 Data modifications

Data in the HEAD_DIRTY state may be modified immediately, before the remaining sharing-list entries are invalidated. After data are modified, the head of a modifiable sharing list (HEAD_DIRTY) purges the remaining sharing-list entries. For the typical set of options, the initial transaction to the second sharing-list entry purges (1) that entry from the sharing list and returns its forward pointer. The forward pointer is used to purge (2) the next (formerly the third) sharing-list entry. The process continues until the tail entry is reached, as illustrated in figure 145.

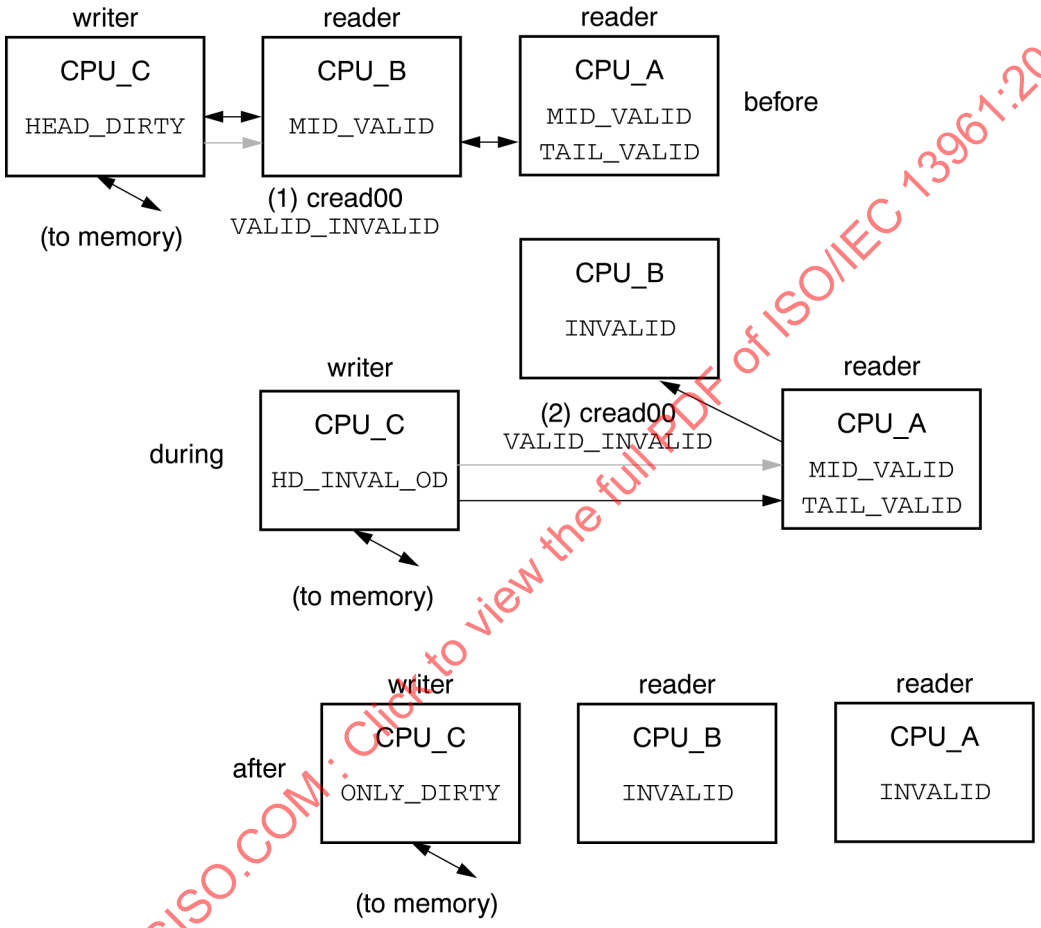


Figure 145 – Head purging others

Concurrent deletions may temporarily corrupt the *backId* pointers in one or more of the sharing-list entries. Since the head-initiated purge uses only the *forwId* pointers, the purges and deletions can safely be performed at the same time.

The purging state (HD_INVAL_OD) is similar to the PENDING state, in that new sharing-list additions are delayed while the purges are being performed. Note that purge latencies increase linearly with the number of sharing readers. Since purge lists are often short, the linear latencies may be acceptable in many systems.

An ONLY_FRESH entry is changed to the ONLY_DIRTY state before the data are modified. This requires an additional memory-access transaction (1) mread00.LIST_TO_GONE, which changes the memory-directory state from FRESH to GONE, as illustrated in figure 146.

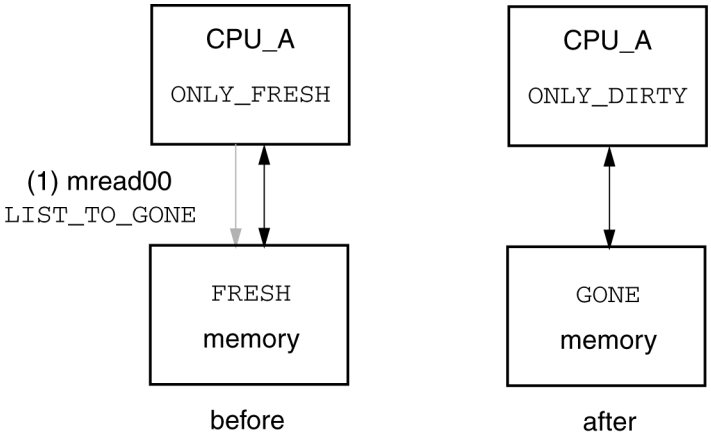


Figure 146 – ONLY_FRESH list conversion

Similarly, a HEAD_FRESH entry is changed to an intermediate modifiable (HEAD_DIRTY) state before the data are modified and the other sharing-list entries are invalidated. The memory-access transaction (1) mread00.LIST_TO_GONE is used to change from the HEAD_FRESH to HEAD_DIRTY state, the data modifications are performed, and the cache-line state is changed to an intermediate HD_INVAL_OD state. The other copies are then invalidated (2), as illustrated in figure 147.

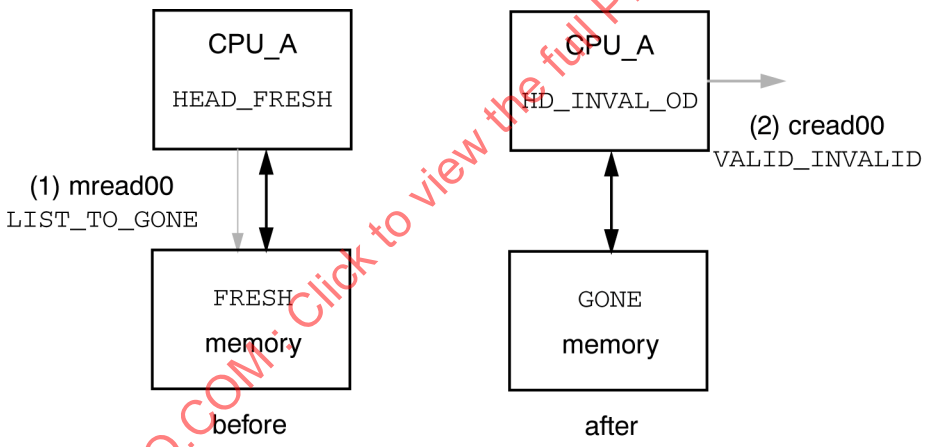


Figure 147 – HEAD_FRESH list conversion

The mread00.LIST_TO_GONE transaction's update of memory state is conditional; if the memory directory points to a newly queued cache entry the update is nullified. This nullification is detected by the sharing-list head, which then deletes itself from the sharing list and re-attaches in a modifiable (ONLY_DIRTY or HEAD_DIRTY) state.

4.4.5 Mid and head deletions

Entries can also be deleted from the list by their own controller when they are needed to cache data at other addresses (cache-line rollout). The sharing-list deletions involve the update of the *backId* in the next (closer to the tail) entry, and the *forwId* pointer in the previous (closer to memory) entry. Before the deletion begins the entry is converted into a locked state. A MID_VALID entry is converted into the locked MV_FORW_MV state and transactions (1 and 2) to the adjacent sharing-list entries are generated, as illustrated in figure 148.

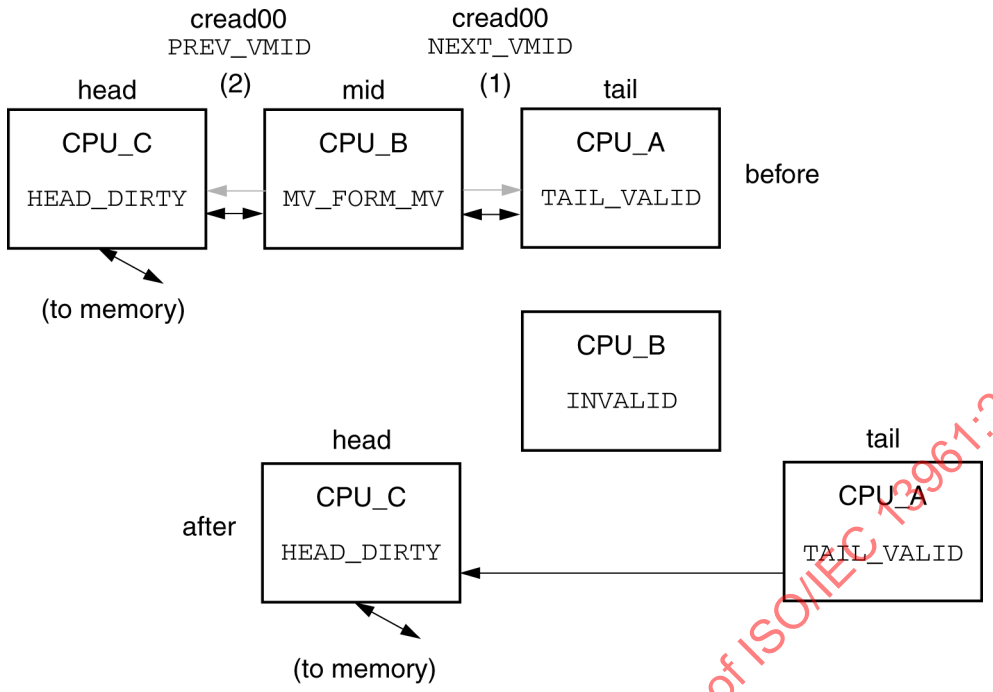


Figure 148 – Mid-entry deletions

Head entries can also delete themselves from the list, e.g., when they are needed to cache data at other addresses (cache-line rollout). The sharing-list deletions involve (1) the update of the backId in the next (closer to the tail) entry, and (2) the forwId pointer in the memory directory, as illustrated in figure 149.

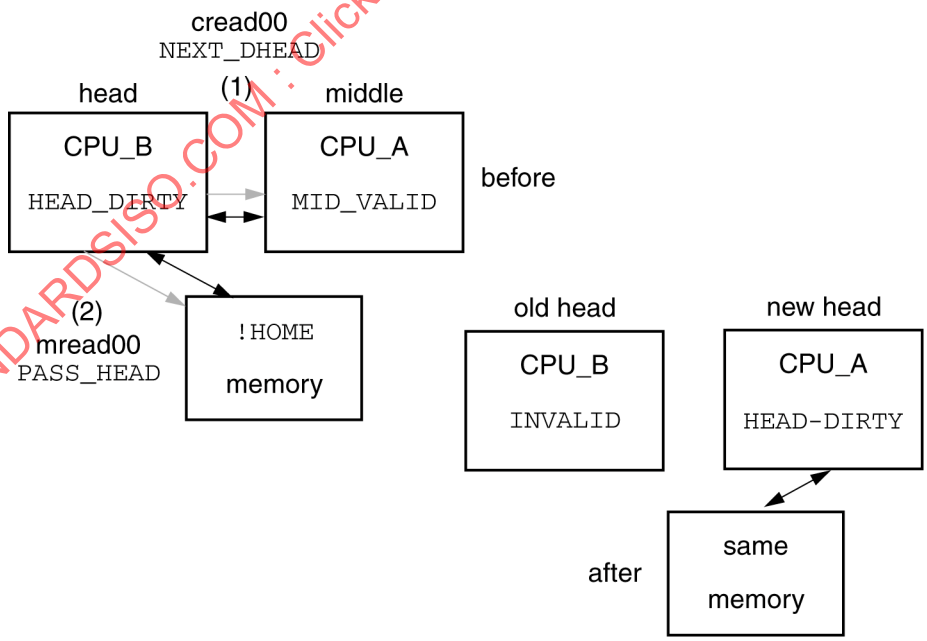


Figure 149 – Head-entry deletions

Recovery from detected transmission errors is usually possible when a single write transaction is used to collapse an ONLY_DIRTY sharing list, but cannot be guaranteed. Multiple transmission errors during a particular set of sharing-list transitions can leave the sharing-list in an uncorrupted (the data won't be incorrectly recovered) but unrecoverable (the correct data can't be recovered) state.

Therefore the fault-tolerance of the SCI system may optionally be improved by using two transactions: the first transaction returns (1) the dirty data and the second transaction collapses (2) the sharing list. These two steps are illustrated in figure 150.

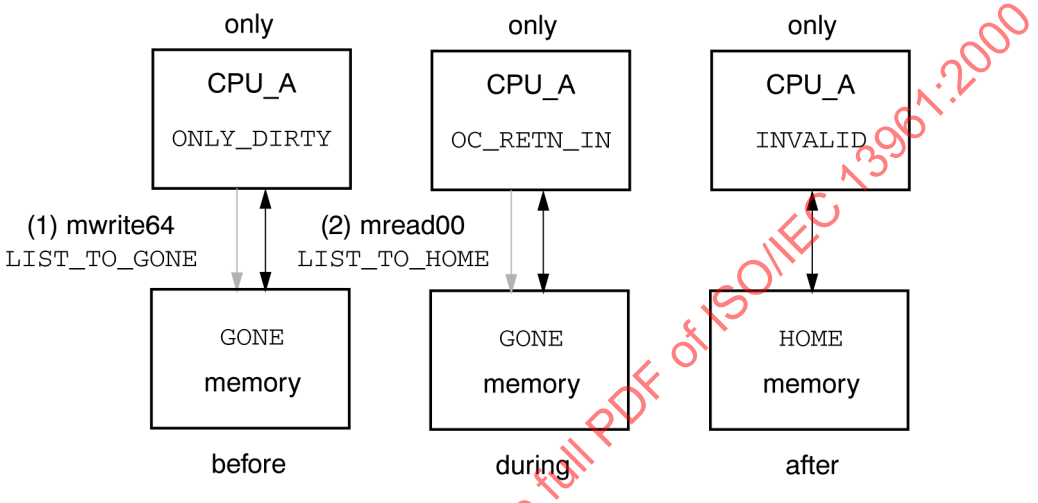


Figure 150 – Robust ONLY_DIRTY deletions

4.4.6 DMA reads and writes

On a read, a DMA controller needs a coherent copy of the data but has no need to cache the copy for future use. Therefore, a special read64.ATTACH_TO_GONE transaction is used (1) to fetch the data from memory. If the addressed location is HOME or FRESH, the data are returned directly from memory; otherwise the controller's cache is prepended to the previous sharing-list head, from which it fetches the most-recently modified data. Thus, the DMA controller can often fetch its data from memory (when it is in the HOME or FRESH states) without joining the sharing list, as illustrated in figure 151.

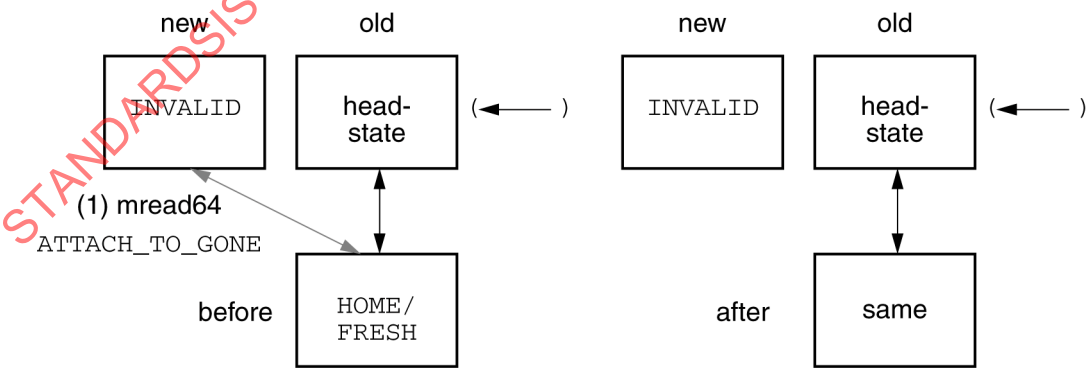


Figure 151 – Checked DMA reads

A DMA-write option supports writes of partial or full cache-lines that need not be cached by the DMA controller. The DMA controller writes (1) its data to memory using a mwrite64.FRESH_TO_HOME transaction. If the memory line was in the HOME state, the write is performed and the memory state remains unchanged. If the memory line was in the FRESH state, the memory state is changed to HOME and the pointer to the old sharing-list is returned (2, 3, ...) for purging by the DMA controller, as illustrated in figure 152.

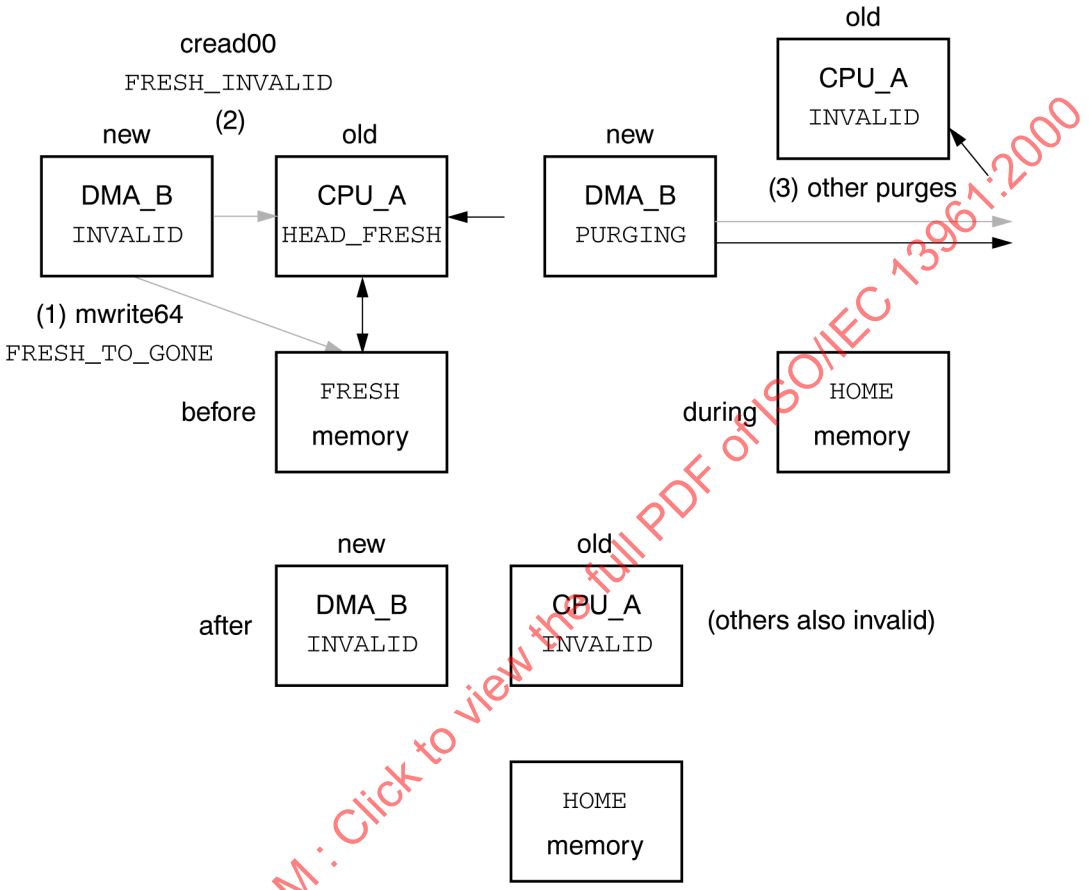


Figure 152 – Checked DMA write (memory FRESH)

If the memory state was GONE, the DMA controller attaches to the old sharing list (1 and 2), purges the remaining entries (3, ...), and (eventually) generates a transaction (N) to return its dirty copy to memory, as illustrated in figure 153.

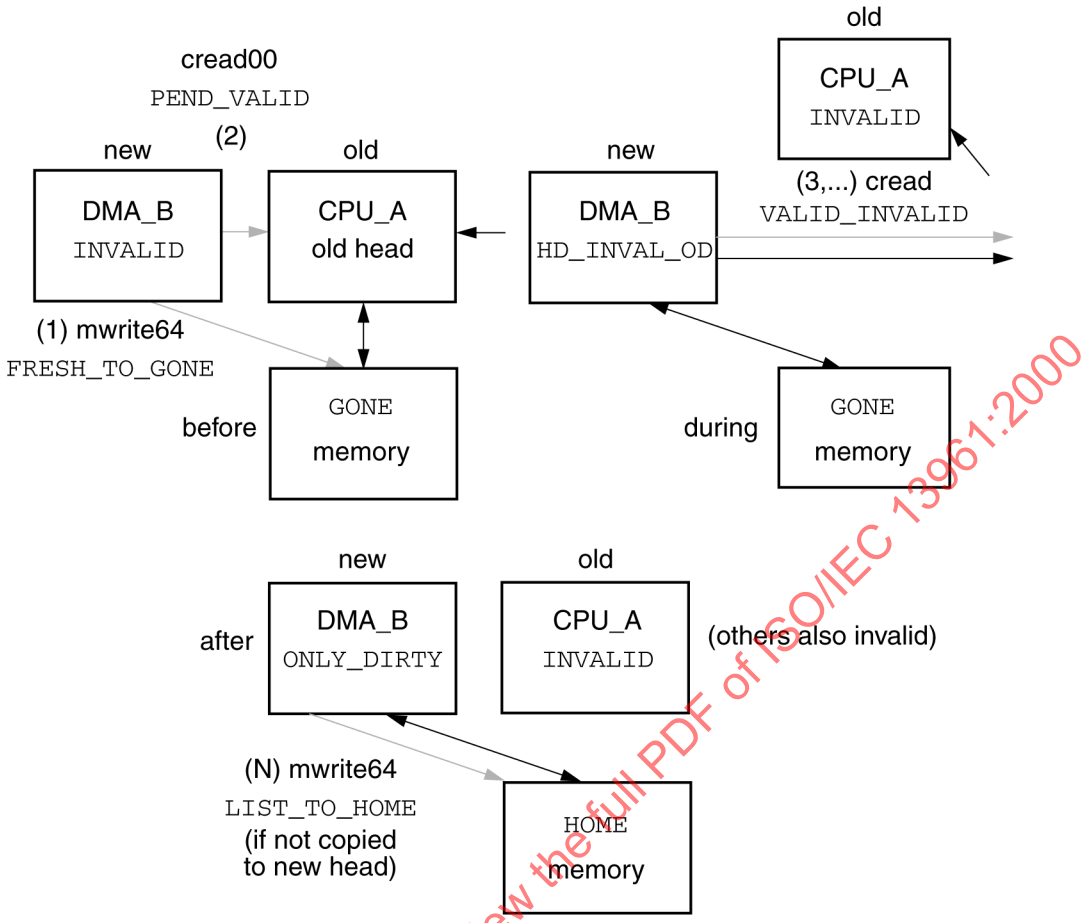


Figure 153 – Checked DMA write (memory GONE)

The mwrite64.LIST_TO_HOME transaction is not necessarily generated; the data may be fetched by another processor before being returned to memory.

The DMA-write optimization can generate a temporary condition where memory is in the HOME state while fresh copies of the data exist in caches. To ensure sequential consistency, higher-level I/O driver-software protocols (interrupts and DMA-completion messages) are expected to test for the completion of the purge process. Processors that use the DMA-write option are expected to provide equivalent forms of testing for the completion of the purge process.

4.4.7 Instruction-execution model

For efficient cache operation, a processor must communicate the nature of its access to data as well as the address of the data. Some processors at present lack the appropriate instructions for this and must simulate them by using special addresses or instruction sequences. Generic instructions that provide the needed information are assumed in the following.

The processor is expected to check and change cache-line states before and after instructions are executed. These checks and changes are modelled by the cache-execute routines listed in table 24.

Table 24 – TypicalExecute Routines

name	generated by the execution of
`TypicalExecuteLoad()`	a *load* memory-access instruction
`TypicalExecuteStore()`	a *store* memory-access instruction
`TypicalExecuteFlush()`	the *global flush* cache-control instruction (which collapses the sharing list)
`TypicalExecuteDelete()`	the *local flush* cache-control instruction (which deletes the local cache entry)
`TypicalExecuteLock()`	the *fetch&add*, *compare&swap*, and *mask&swap* instructions
NOTE The `TypicalExecuteLoad()` routine is equivalent to the `FullExecuteLoad()` routine, with the proper set of option bits. However, separate routines are provided so that this basic functionality is not obscured by the generality of the `FullExecuteLoad()` routine (which documents all options).	

4.5 Full-set coherence protocols

4.5.1 Full-set option summary

The full set of coherence options includes the typical set plus clean sharing lists, efficient cache control (flush, purge, and cleanse), pairwise sharing, and QOLB. Implementations are not expected to implement the full set of options. However, the full set is interoperable with any subsets (only the resulting efficiency varies) and provides a wide range of options from which to choose a nearly optimal subset.

The code for the full option set is part of the specification, from which the minimal and typical option sets can be derived. The special operations of this set are described in this subclause; the detailed specification can be found in the C code.

4.5.2 CLEAN-list creation

Initially, memory is in the `HOME` state and all caches are `INVALID`. The sharing-list creation begins at the cache, where an entry is changed from the `INVALID` to the `PENDING` state. To fetch a modifiable copy (which is not immediately modified), a clean copy is fetched (1) from memory using an mread64.`CACHE_CLEAN` transaction. This leaves a newly created cache line in the `ONLY_CLEAN` state, as illustrated in figure 154.

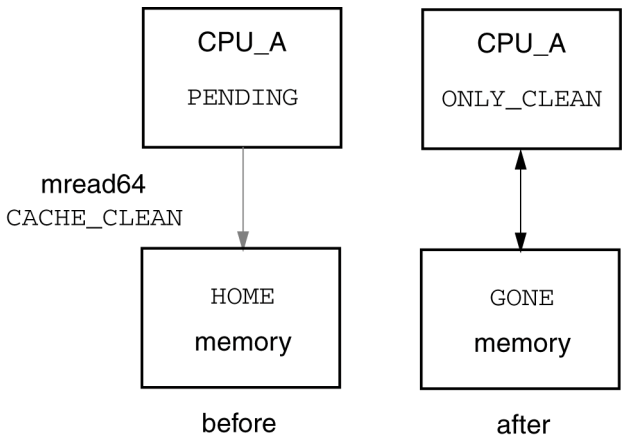


Figure 154 – CLEAN list creation

Clean sharing lists minimize the latencies for subsequent writes, since the data may be immediately written. An ONLY_CLEAN state is more efficient than the nearly equivalent ONLY_DIRTY state, since the data need not be returned to memory when the sharing list is collapsed.

4.5.3 Sharing-list additions

For subsequent accesses, the memory state is either FRESH or GONE and the head of the sharing list has the (possibly dirty) data. When fetching (1) read-only data from a (possibly) modified sharing list, a pointer is returned from memory and the new requester fetches (2) its data when attaching to the old sharing-list head. These steps are illustrated in figure 155.

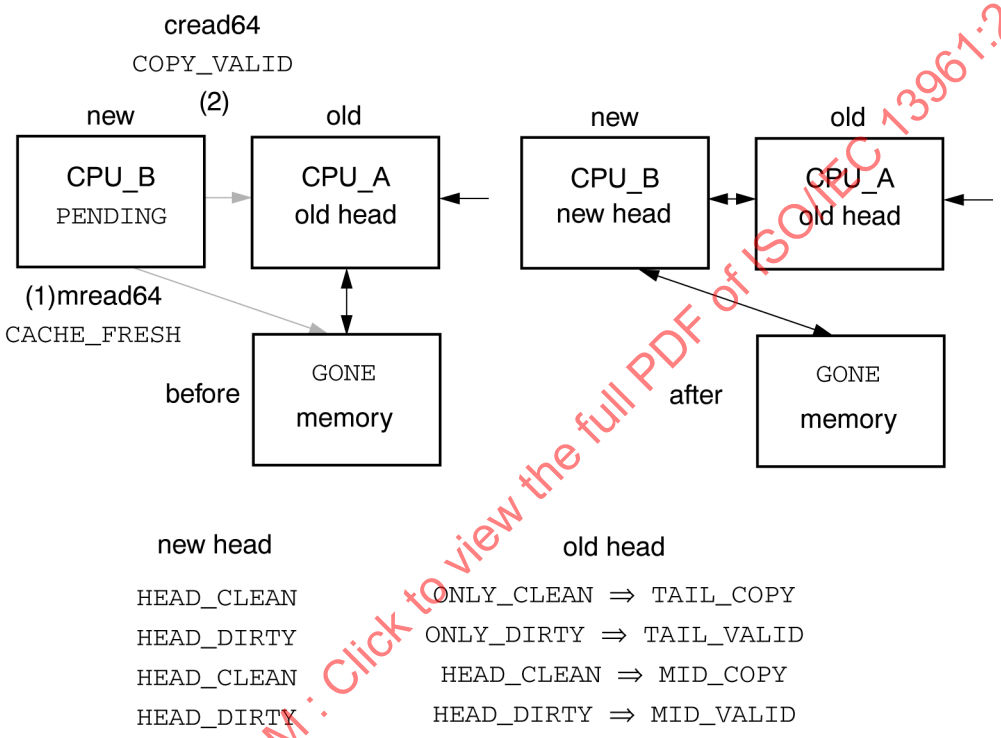


Figure 155 – FRESH addition to CLEAN/DIRTY list

Note that the previously clean head entries are left in the MID_COPY or TAIL_COPY state after the prepend completes. These optional copy states indicate that the data are the same as memory. This information may be used by the *cleanse* cache-control instructions (only dirty cache lines need be returned to memory).

When fetching (1) clean data from a fresh sharing list, the fresh data are returned from memory before the new head attaches (2) to the old sharing list as illustrated in figure 156.

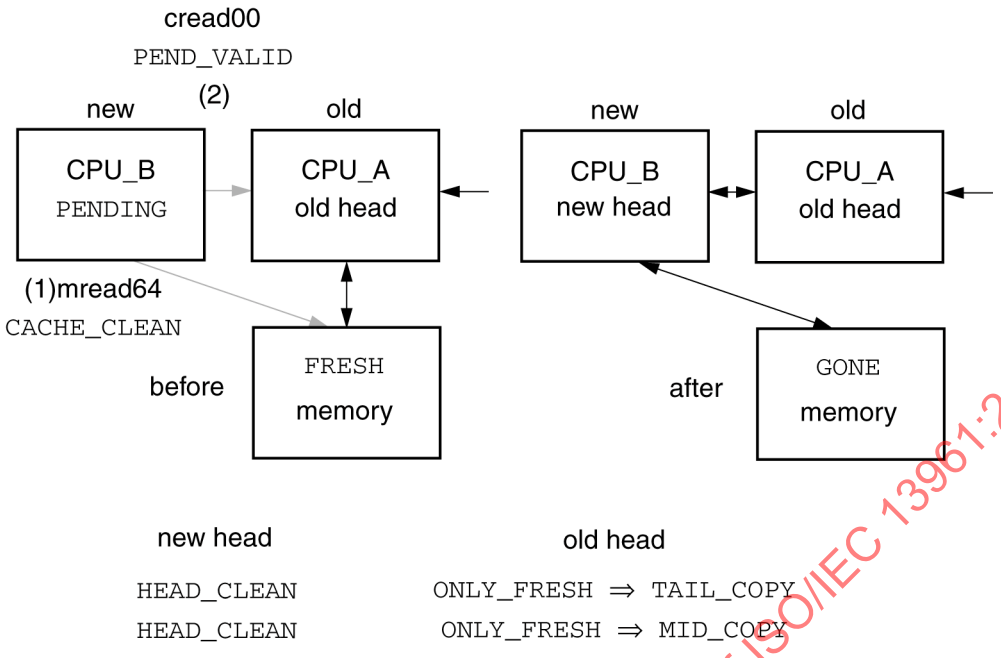


Figure 156 – CLEAN addition to FRESH list

When fetching (1) clean data from a clean or dirty sharing list, a sharing-list pointer is returned from memory and the data are fetched (2) when the new head attaches to the old sharing list as illustrated in figure 157.

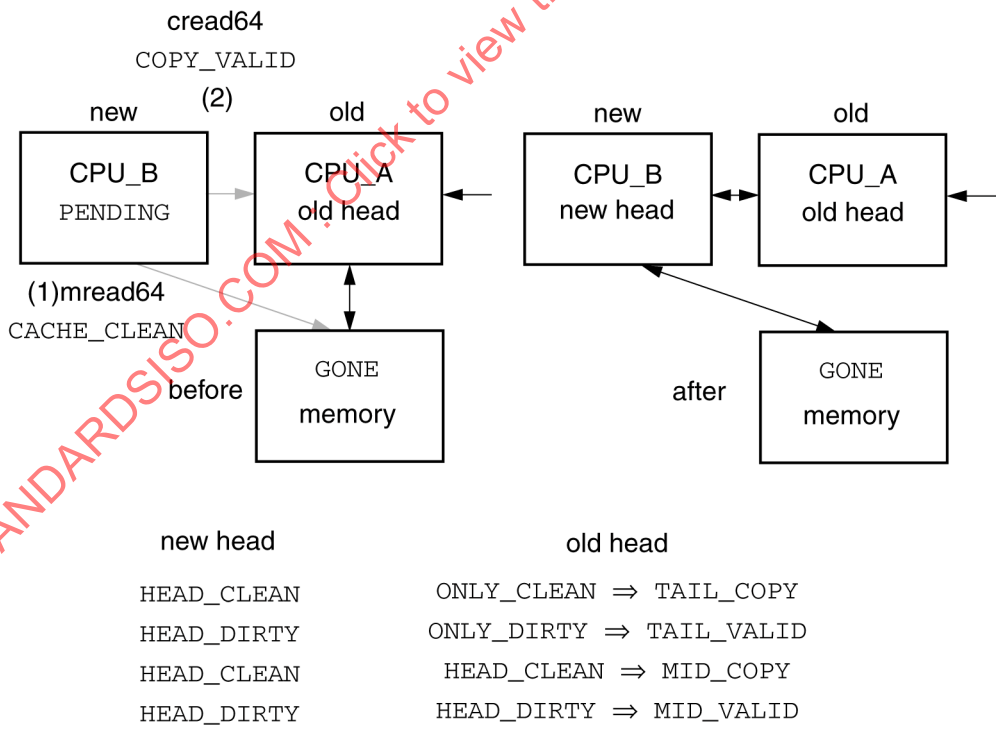


Figure 157 – CLEAN addition to CLEAN/DIRTY list

4.5.4 Cache washing

Read-only data are most efficiently accessed when in the fresh state; memory can return a data copy for use while a new head is prepending to an existing sharing list. However, most cache lines will be written before being used (for example, a cache line is written when pages are fetched from disk), and (if the cache line remains cached) a written cache line is left in the dirty state.

An optional washing protocol is provided to convert a dirty sharing list to the FRESH state, to improve the efficiency of accessing data that has become read-only. After a write has been performed, the washing protocol is performed by readers when they prepend themselves to the dirty sharing list.

A write will generally leave a previously written cache line in the ONLY_DIRTY state. The first read64.CACHE_FRESH of the ONLY_DIRTY line (which is not affected by the washing protocols) leaves the sharing list in the HEAD_DIRTY/TAIL_VALID states (steps 1 and 2 of figure 158. After the second successive read64.CACHE_FRESH attempt (3 and 4), a write64.LIST_TO_FRESH transaction returns the dirty data to memory (6). In the absence of additional reads, this would convert the memory and sharing-list states to *fresh*.

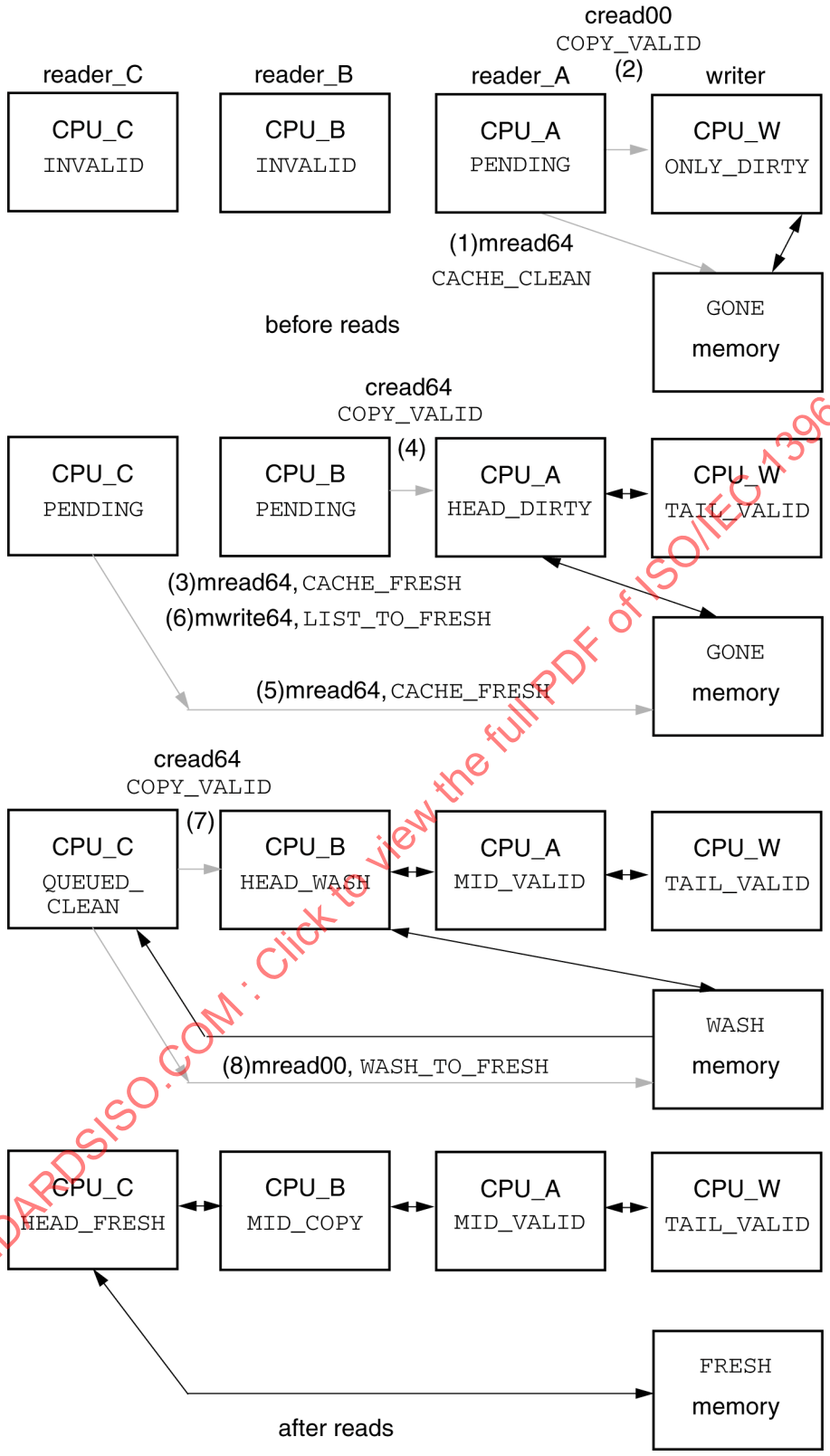


Figure 158 – Washing DIRTY sharing lists (prepend conflict)

However, another processor (CPU_C) may get a modifiable copy of the data from memory (5) before the LIST_TO_FRESH update (6) has been processed. This form of prepend conflict delays the washing process, leaving the memory and the sharing list in the WASH and HEAD_WASH states respectively. When entering the HEAD_WASH state, the sharing-list head (CPU_B) saves the identity of the conflicting reader (CPU_C) in its *backId* pointer.

After prepending to a HEAD_WASH list (7), the third reader (CPU_C) checks the returned *backId* value. If equal to its own *nodeId* value, CPU_C generates the read00.WASH_TO_FRESH transaction to convert memory from the WASH to FRESH states. Since memory's *forwId* value is ignored during the WASH_TO_FRESH conversion, the second wash cycle is not affected by additional readers that prepend to the same sharing list at nearly the same time.

Under light loading conditions, the washing process uses one extra transaction (mwrite64.LIST_TO_FRESH) to convert the sharing list from the GONE to the WASH state. Under heavy loading conditions (when the cache line is being concurrently accessed by multiple readers), the washing process uses two extra washing transactions (mwrite64.LIST_TO_FRESH and mread00.WASH_TO_FRESH) to convert sharing lists from the HEAD_DIRTY to the HEAD_FRESH state.

4.5.5 Cache flushing

A flush operation collapses the sharing list and returns dirty data (if any) to memory. After the flush has completed, the memory directory is normally left in the HOME state. When cache-line addresses are flushed, a memory transaction is necessary to confirm that copies that are locally invalid are globally invalid as well.

For example, a cache line of the flushing processor (not in the sharing list) could be in the INVALID state if the data are being read-shared by others. The flushing processor sends an mread64.ATTACH_TO_LIST transaction to memory, which prepends the flushing processor to an existing sharing list.

If memory is in the HOME state, no sharing list exists and the flush is completed when the memory response is returned. If the memory is in the FRESH state, the data are returned (1) from memory and the processor purges (2, 3, ...) the remaining sharing-list entries before returning the sharing-list ownership, as illustrated in figure 159. Data are requested from memory, in case another sharing-list prepend occurs (and requests the shared data) before the flush operation completes.

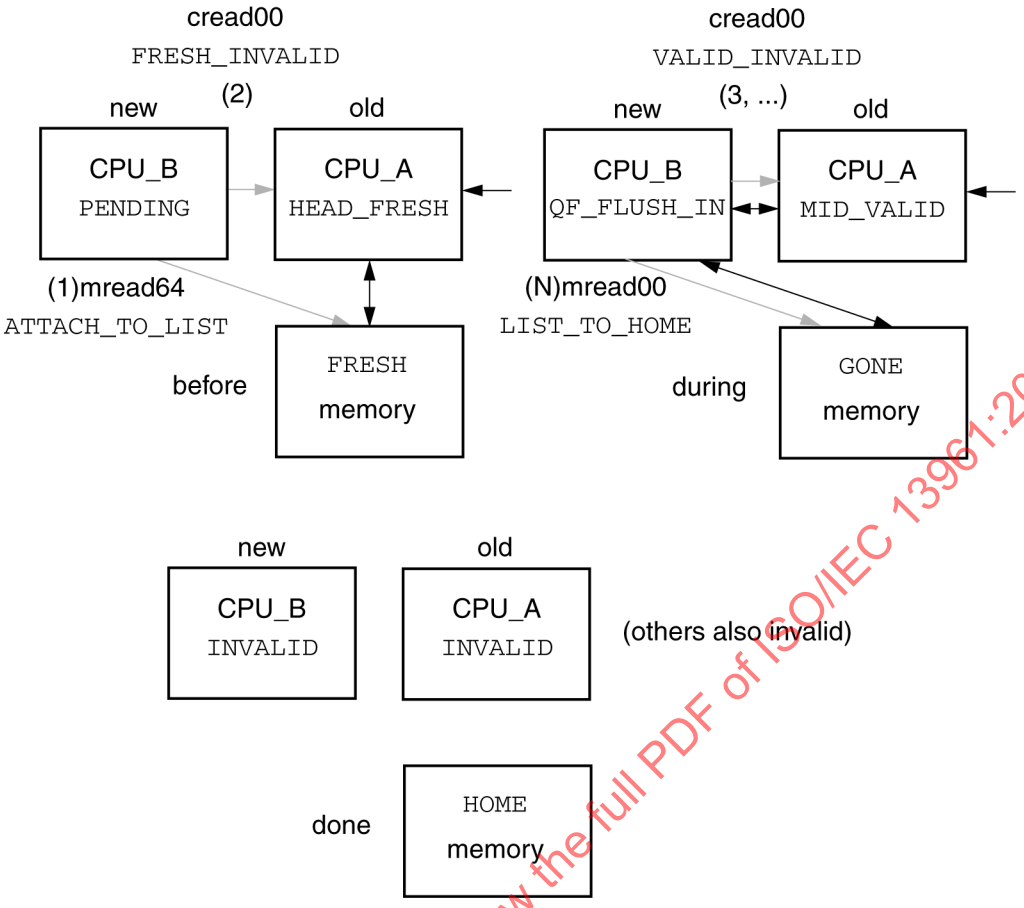


Figure 159 – Flushing a FRESH list

If memory is in the GONE state, a list pointer is returned (1) from memory and the data are fetched (2) when the processor attaches to the old sharing list head. After invalidating (3, ...) old entries, the old data (which may have been modified) is returned (N) to memory with the sharing-list ownership, as illustrated in figure 160.

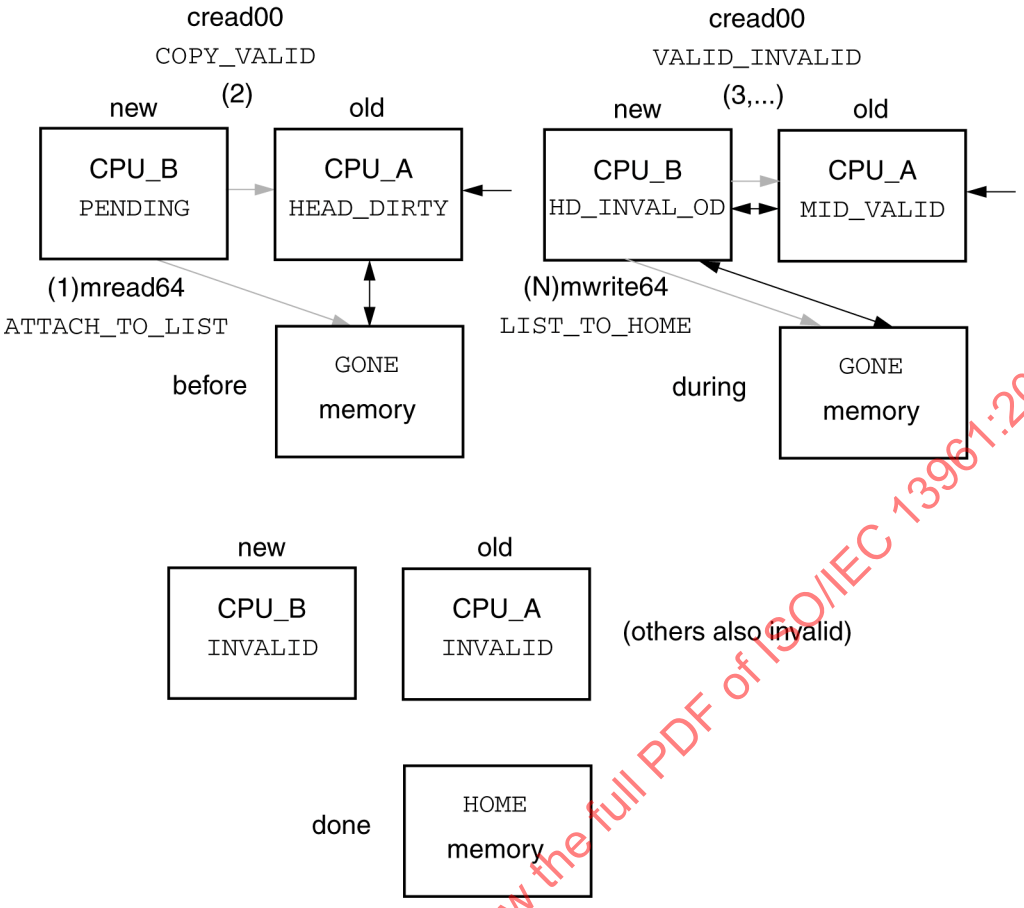


Figure 160 – Flushing a GONE list

The cache-purge instruction similarly collapses the existing sharing list, but discards dirty data rather than returning them to memory. A cache-purge instruction would be used to return ownership of coherent copies to memory before the data are noncoherently overwritten; for example, a purge instruction could be used to release the contents of a stack frame or a data buffer before a noncoherent DMA input transfer.

4.5.6 Cache cleansing

A dirty cache line may be cleansed by the execution of a *cleanse* cache-control instruction, which copies dirty data to memory, but does not necessarily collapse the sharing list. Cache cleansing is expected to be used with specialized memory, such as a graphics frame-buffer memory or nonvolatile memory (batteries maintain memory, but not the cache, when power is lost). For a graphics frame buffer or nonvolatile memory, cleansing a cache line puts the most recent updates on the screen or in checkpointable memory, while leaving the data efficiently cached for further updates.

The cleansing of an `ONLY_DIRTY` cache line involves a write to memory (1), during which the ownership is checked. If there is a new sharing-list owner (3), the cleansing cache then attempts to delete (2 and 4) itself from the list. This deletion is necessary to ensure forward progress, since otherwise writes and cleansing could constantly change the cache line between the `ONLY_DIRTY` and `ONLY_CLEAN` states and a new prepender could be delayed indefinitely by these continual changes. These cleansing steps are illustrated in figure 161.

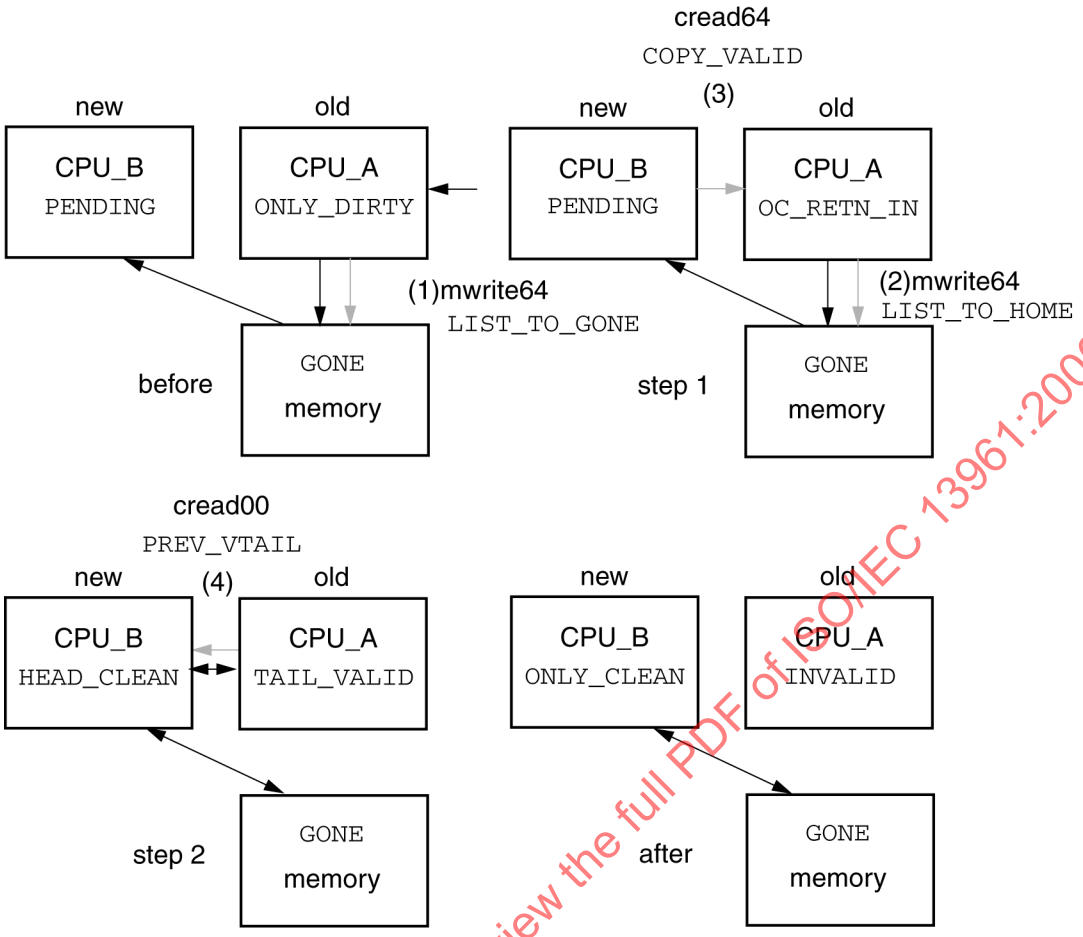


Figure 161 – Cleansing DIRTY sharing lists (prepend conflict)

A HEAD_DIRTY cache line is cleansed by copying the previously dirty data to memory. If the ownership has changed, the old head remains in the list (in the HEAD_CLEAN state) and forward progress is still guaranteed; the next write converts the head to ONLY_DIRTY, for which forward progress is assured (as described previously).

To minimize the number of cleansing-related transactions, special mid- and tail-entry states (MID_COPY and TAIL_COPY) are defined. When a new entry is prepended to a HEAD_CLEAN or ONLY_CLEAN entry, the head of the old sharing list is left in the MID_COPY or TAIL_COPY states, respectively. The MID_COPY and TAIL_COPY entries indicate that the sharing list is clean, so that cleansing instructions to these cache-line addresses need not change the sharing-list state.

4.5.7 Pairwise sharing

The pairwise-sharing option supports direct cache-to-cache transfers between the head and tail entries in a two-entry sharing list. The pairwise-sharing option reduces memory bottlenecks, since shared data can be transferred directly using cache-to-cache transfers.

Several types of cache-to-cache transfers are used to transfer data and ownership of cache lines between the head and tail sharing-list entries. For example, a store miss in a processor with a HEAD_DIRTY copy uses the cread00.TAILV_TO_STALE transaction (1) to convert the pair of sharing-list entries from the HEAD_DIRTY/TAIL_VALID to the HEAD_EXCL/TAIL_STALE0 states, as illustrated in figure 162.

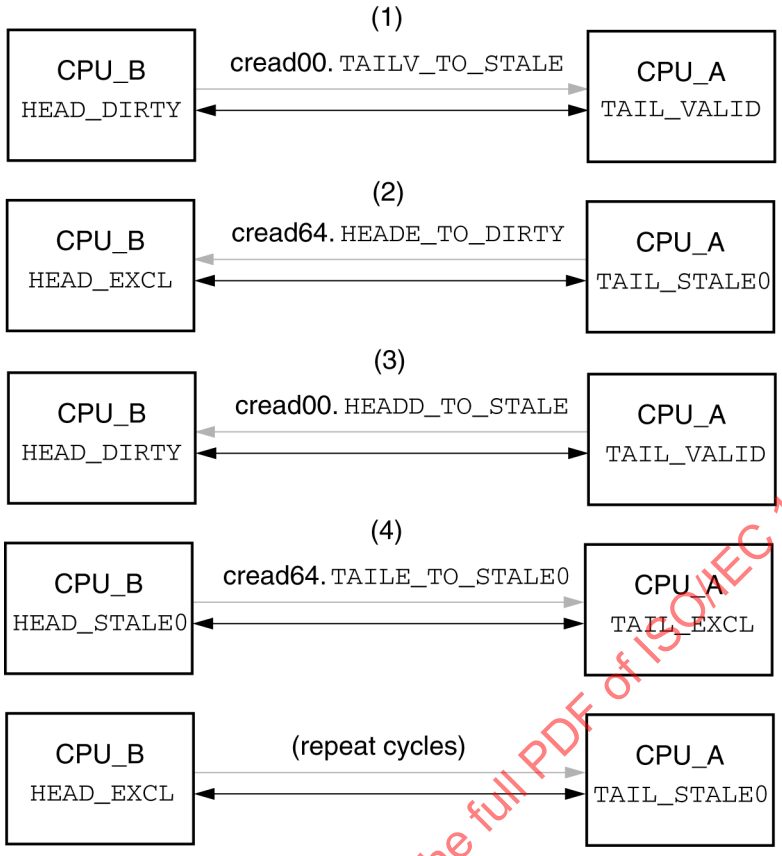


Figure 162 – Pairwise-sharing transitions

Similarly, a load miss in the TAIL_STALE0 state generates a cread64.HEADE_TO_DIRTY transaction (2), which converts the entries from the HEAD_EXCL/TAIL_STALE0 to HEAD_DIRTY/TAIL_VALID states; a store miss in the TAIL_VALID state generates a cread00.HEADD_TO_STALE transaction (3), which converts from the HEAD_DIRTY/TAIL_VALID to the HEAD_STALE0/TAIL_EXCL states. An exclusive copy can be directly transferred as well; a store miss in the HEAD_STALE0 state generates a cread64.TAILE_TO_STALE0 transaction (4) which converts from the HEAD_STALE0/TAIL_EXCL states to the HEAD_EXCL/TAIL_STALE0 states.

There is a potential for conflict if the cread00.TAILV_TO_STALE and cread00.HEADD_TO_STALE transactions are generated concurrently. When such conflicts occur, the dirty copy has precedence; the head and tail entries change to the HEAD_EXCL and TAIL_STALE0 states respectively. Similarly, when the cread00.HEADV_TO_STALE and cread00.TAILD_TO_STALE transactions are generated concurrently, the entries change to the HEAD_STALE0/TAIL_EXCL states.

The existence of pairwise sharing affects the prepend process, as illustrated in figure 163. After accessing memory (1), when prepending (2) to the HEAD_EXCL/TAIL_STALE0 or HEAD_EXCL/TAIL_STALE1 sharing-list states, an extra cread00.TAIL_INVALID transaction (3) is required in order to purge the old tail entry after the new head has prepended to the old sharing-list head.

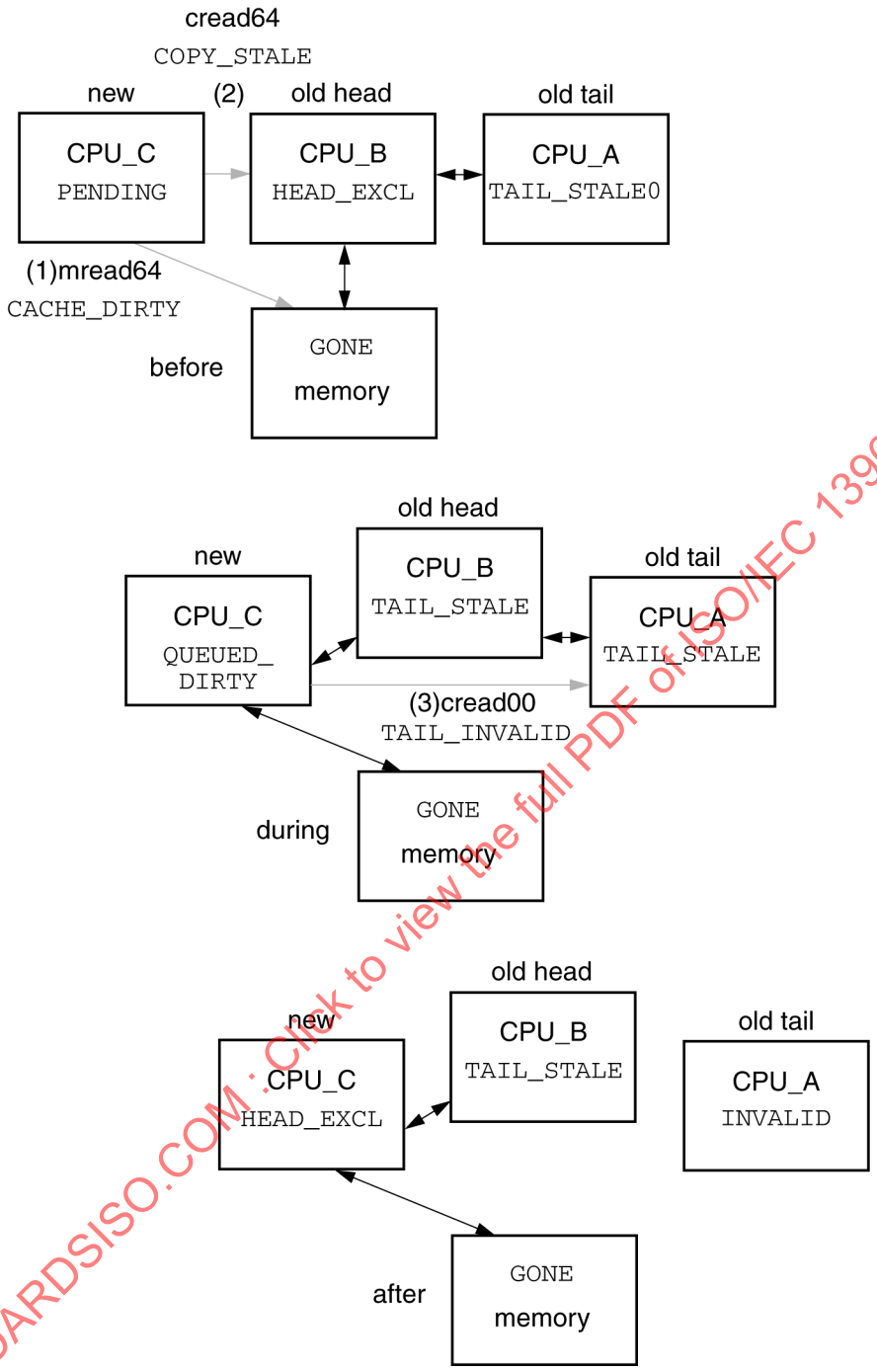


Figure 163 – Prepending to pairwise list (HEAD_EXCL)

Prepending to the HEAD_STALE0/TAIL_EXCL or HEAD_STALE1/TAIL_EXCL sharing list also takes one more step, as illustrated in figure 164. After accessing memory (1), when prepending (2) to a HEAD_STALE entry, the old head entry is changed to the SAVE_STALE state and provides the pointer to the tail, from which the data are returned. After prepending (3) to the old tail entry, the new head purges (4) the old sharing-list head (which was left in the transient SAVE_STALE state).

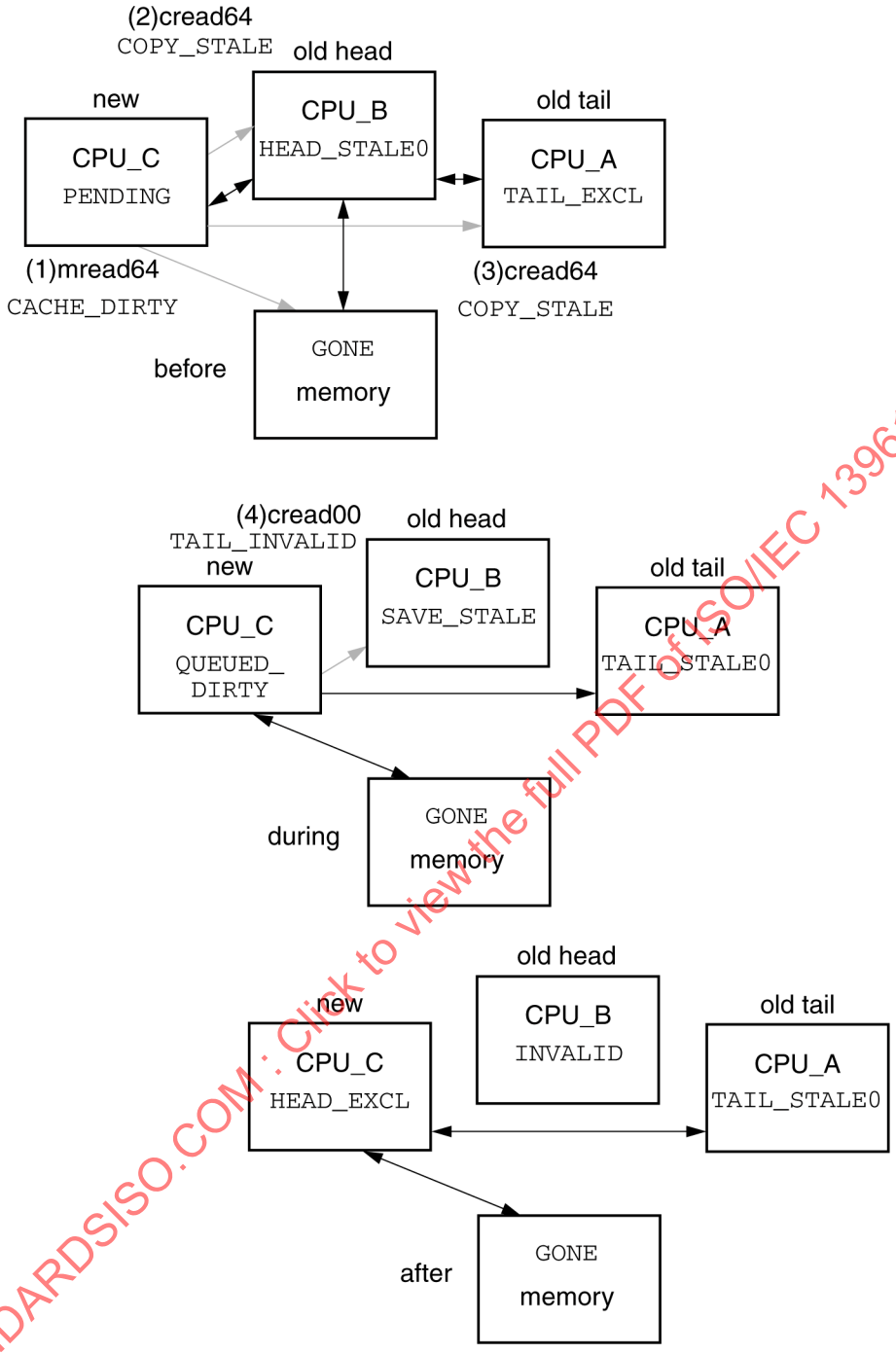


Figure 164 – Prepending to pairwise list (HEAD_STALE0)

A pairwise tail sets its *forwId* pointer equal to its *backId* value, to simplify the prepend process for the new sharing-list head. From the new head's perspective, the prepend process always involves the deletion of initial entries (which have invalid data), the transfer of (potentially dirty) data, and the post-invalidation of a remaining stale copy.

4.5.8 Pairwise-sharing faults

The pairwise sharing protocols support the efficient transfer of an exclusive (i.e., modifiable) cache line between the head and tail of a two-entry sharing list. In the event of transmission failures, the response (which contains the exclusive copy of data) may be dropped and the entries can both end up in similar externally visible stale states, as illustrated in figure 165.

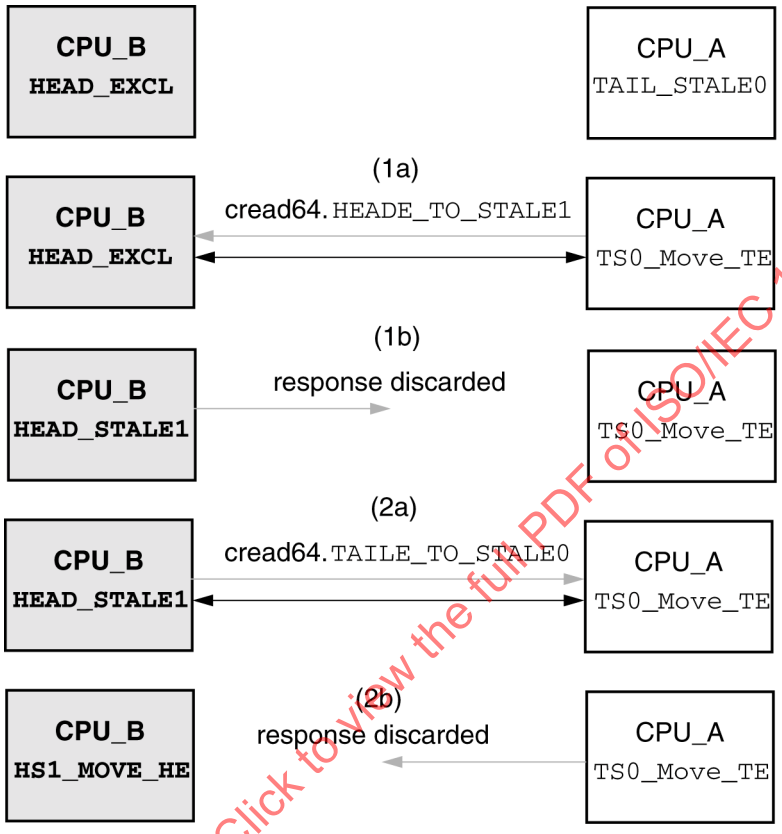


Figure 165 – Two stale copies, head is valid

In this example, the only valid copy is originally owned and retained by the head entry, as indicated by the shaded boxes. The tail entry initiated a cread64.HEADE_TO_STALE1 transaction to fetch the data from the head entry (1a). This transaction is processed by the head entry, which changes to the HEAD_STALE1 state and returns a response. The response is destroyed by a transmission failure (1b), and never returns to the tail entry. The head entry attempts to fetch its exclusive copy from the tail, using a cread64.TAILE_TO_STALE0 transaction, which is delayed by the tail waiting for its previous response (1b). This leaves the head and tail entries in the HS1_MOVE_HE and TS0_Move_TE states, respectively.

With these similar head and tail entry states, sequence bits (which differentiate between the HEAD_STALE0/HEAD_STALE1, TAIL_STALE0/TAIL_STALE1, HS0_MOVE_HE/HS1_MOVE_HE, and TS0_Move_TE/TS1_Move_TE state pairs) are necessary to identify the location of the most-recently modified data (which could be in the head or tail entry).

To illustrate the operation of the sequence bits, consider an example where the exclusive copy is originally owned by the tail entry and the head is in the HEAD_STALE0 state. In this case, the only valid copy is originally owned and retained by the tail entry, as indicated by the shaded boxes, in figure 166.

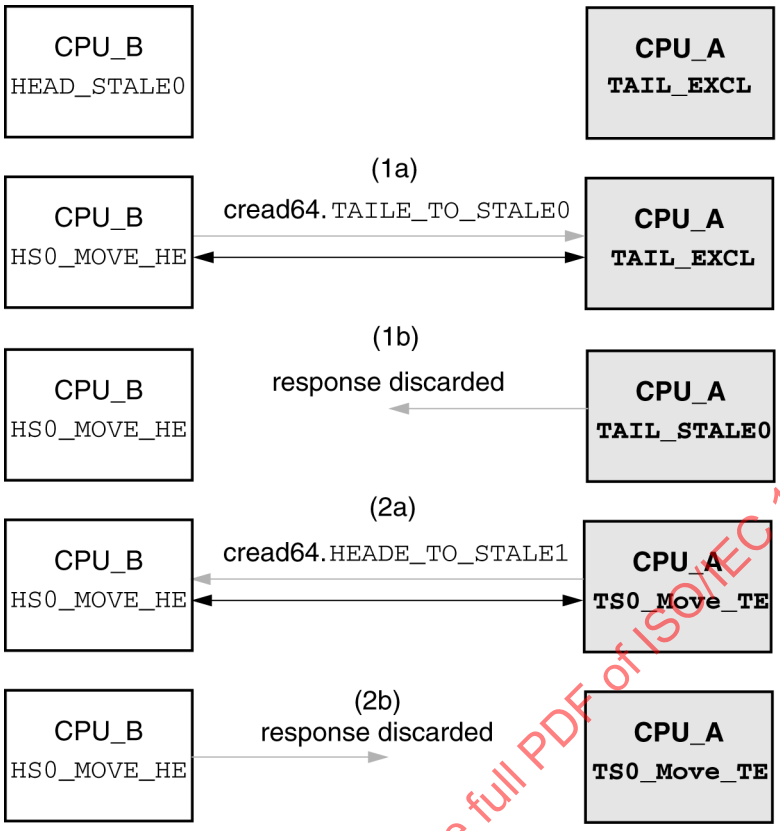


Figure 166 – Two stale copies, tail is valid

The head entry initiates a cread64.TAILE_TO_STALE0 transaction to fetch the data from the tail entry (1a). This transaction is processed by the tail entry, which changes to the TAIL_STALE0 state and returns a response. The response is destroyed by a transmission failure (1b), and never returns to the head entry. The tail entry attempts to retrieve its exclusive copy from the head, using a cread64.HEADE_TO_STALE1 transaction, which is delayed by the head waiting for its previous response (1b). This leaves the head and tail entries in the HS0_MOVE_HE and TS0_Move_TE states, respectively.

The sequence bits depend on whether the valid data are left in the head or in the tail states. Recovery software is expected to check the sequence bit in the head and tail entries. If the two values differ (as in the first of these two examples), the modified data are recovered from the head entry. If the two values are the same (as in the second of these two examples), the modified data are recovered from the tail entry.

4.5.9 QOLB sharing

SCI also supports efficient synchronization primitives for large scale multiprocessors. One, the queued-on-lock-bit (QOLB) concept, provides FIFO access to shared variables. The rationale for QOLB is to offer local spin-waiting on exclusive data structures. Since linked cache entries form the QOLB queue, little additional hardware is needed to implement this scheme.

The QOLB protocols offer synchronization on a per-memory-line basis. Using QOLB, a processor can request an exclusive copy of a memory line in its cache, and once granted, the line will stay in the cache until it is either rolled out or explicitly released. The processor that has such an exclusive copy of the line is called the QOLB owner of the line. If no other processor has requested the line, then the state of the exclusive copy is ONLY_USED. When a new processor uses QOLB to request the line for exclusive uses, it prepends to the sharing list, by sending a cread64.COPY_QOLB transaction to the old head.

When the cache line is owned by the old sharing-list head, the read64.COPY_QOLB transaction returns that status. The prepending processor then waits in the IDLE state, until the previously owned line is released (or rolled out). Other processors requesting QOLB access to the line will also join the sharing list as idle waiters. The sharing list will then consist of a head entry, mid-entries, and a tail entry, in the HEAD_IDLE, MID_IDLE, and TAIL_NEED states, respectively.

QOLB only guarantees exclusive use of cache lines that remain cached and are not rolled out for other uses. To ensure exclusive use despite rollouts, a lock bit is expected to be set in a line when the line is received for exclusive use. When the processor has completed its use, the lock bit is expected to be cleared. After the lock is cleared, the QOLB ownership is released and the line is sent to the next exclusive user.

The coherence protocols could support out-of-band lock bits, but such specifications have been deferred to extensions of this standard.

With QOLB the efficiency of shared locks is improved because the lock owner keeps the cache line while the lock is owned. The original QOLB concept used a special out-of-band (not part of a normal data item) lock bit. However, out-of-band bits would have complicated the I/O system, since a simple byte-sequential data access model is assumed by most I/O peripherals. SCI therefore implements the flow-control aspects of the QOLB scheme, but does not use memory-resident out-of-band lock bits.

To implement QOLB-like protocols we assume the availability of specialized *enqolb*, *deqolb* and *reqolb* instructions. The enqolb instruction is used to request ownership of a QOLB line; the cache-line is returned in an idle state when that access is delayed. The deqolb instruction is used to release ownership of a QOLB line. The reqolb instruction is a more efficient implementation of the deqolb/enqolb sequence; the ownership of a QOLB line is only released when needed by other idle lines waiting for its release.

The enqolb instruction leaves an unshared cache line in the ONLY_USED state; the distinction between ONLY_USED and ONLYP_DIRTY indicates the cache line has a QOLB owner. If another enqolb is executed while in the ONLY_USED state, the new processor joins the sharing list in the HEAD_IDLE state, as illustrated in figure 167.

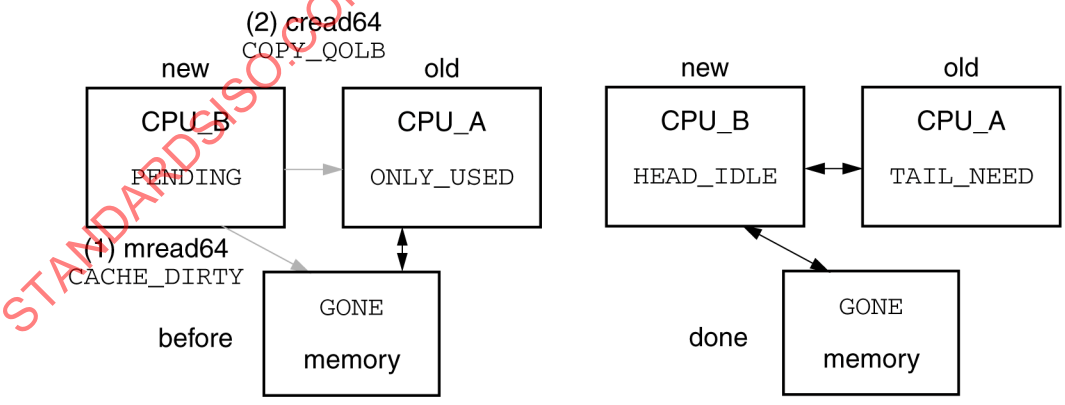


Figure 167 – Enqolb prepending to QOLB-locked list

While in the HEAD_IDLE state additional enqolb instructions are completed locally (with an *unsuccessful* status code). Polling of the TAIL_NEED node is not required, since the next entry is informed when the deqolb operation completes. For long QOLB lists, the deqolb operation converts an (*N*+1)-entry QOLB list into an *N*-entry QOLB list by transferring the tail's dirty data to the previous entry. This is illustrated in figure 168, for a three-entry sharing list.

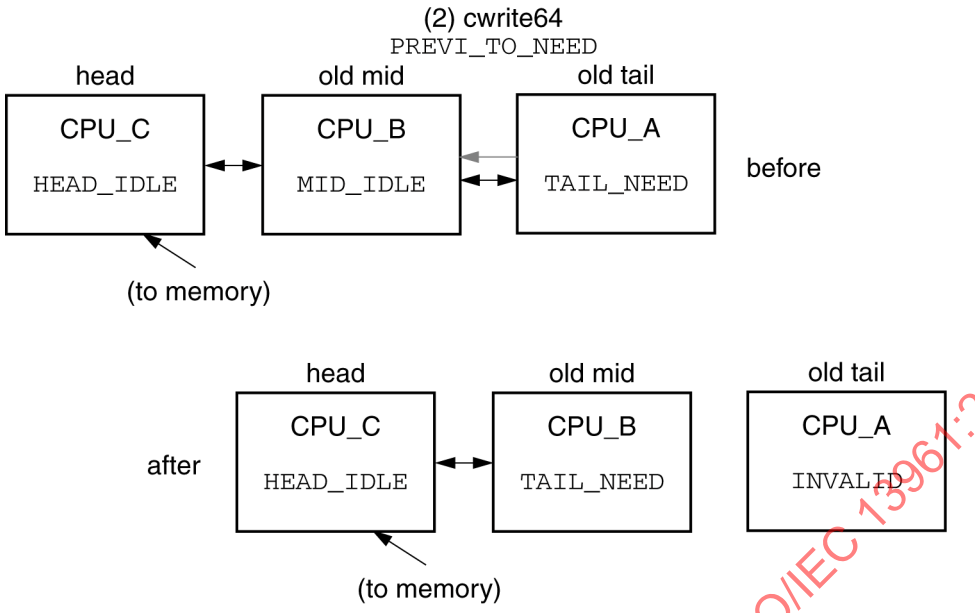


Figure 168 – Deqolb tail-deletion on QOLB sharing list

If there are only two entries in the QOLB list, the exclusive data ownership is returned but the tail remains in the TAIL_STALE state. Thus, the read/write performance advantage of pairwise sharing is applicable to QOLB lists as well.

For pairwise sharing, QOLB contention on a locked data structure generates at most two transactions for each transfer of ownership. The first transaction is generated by the checker, to fetch an exclusive or idle copy. The second transaction is generated by the owner (when the lock is released) and converts the remote copy from the IDLE to the USED or the NEED state. These steps are illustrated in figure 169.

The QOLB protocol also ensures graceful transformation between QOLB use and normal use of a memory line. A normal read/write prepend to a QOLB sharing list breaks down the list and leaves the prepender as the only member of the list (in the state ONLYQ_DIRTY). Additional read or write operations from other processors turn the list into a normal sharing list, while new QOLB operations turn the list back into a QOLB list (with one owner and zero or more waiters).

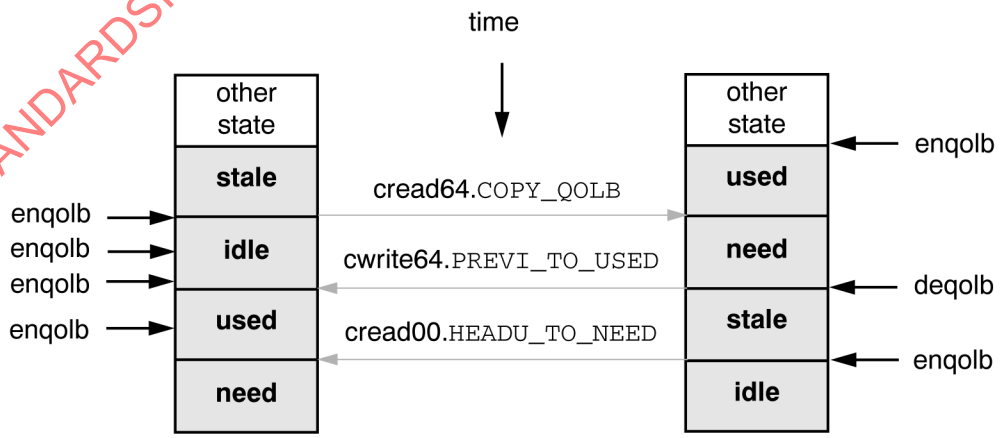


Figure 169 – QOLB usage

4.5.10 Cache-access properties

The read/write properties of the cache-line states, which affect their usefulness when read or written, are indirectly specified by the instruction execution model and associated routines. That specification is illustrated in a more human-readable form in table 25. Note that there are other (unreadable) states, like TAIL_STALE0, that are not included in this table.

Table 25 – Readable cache states

cache state	R/W	clean	write actions	post-write state
ONLY_DIRTY	RW	differs	done	same
ONLYP_DIRTY	RW	differs	done	same
ONLYQ_DIRTY	RW	differs	done	same
HEAD_EXCL	RW	differs	done	same
TAIL_EXCL	RW	differs	done	same
LOCAL_DIRTY	RW	differs	done	same
ONLY_USED	RW	differs	done	same
HEAD_USED	RW	differs	done	same
HEAD_NEED	RW	differs	done	same
TAIL_USED	RW	differs	done	same
TAIL_NEED	RW	differs	done	same
HX_XXXX_OD	RW	differs	done	same
TX_XXXX_OD	RW	differs	done	same
OD_spin_IN	RW	differs	done	same
ONLY_CLEAN	RW	same	change	ONLY_DIRTY [1)]
LOCAL_CLEAN	RW	same	change	LOCAL_DIRTY
QUEUED_MODS	RW	differs	purge	same
QUEUED_DIRTY	RW	differs	purge	same
HD_INVAL_OD	RW	differs	purge	same
HD_MARK_HE	RW	differs	purge	same
TD_mark_TE	RW	differs	purge	same
QUEUED_CLEAN	RW	same	purge	QUEUED_MODS
HEAD_CLEAN	RW	same	purge	HD_INVAL_OD [2)]
HEAD_WASH	RW	same	purge	HD_INVAL_OD [2)]
HEAD_DIRTY	RW	differs	purge	HD_INVAL_OD [2)]
TAIL_DIRTY	RW	differs	purge	TD_MARK_TE
TAIL_VALID	R	differs	NA	NA
MID_VALID	R	differs	NA	NA
HEAD_VALID	R	differs	NA	NA
OD_CLEAN_OC	R	differs	NA	NA
HD_CLEAN_HC	R	differs	NA	NA
HD_WASH_HF	R	differs	NA	NA
HV_MARK_HE	R	differs	NA	NA
HX_INVAL_OX	R	differs	NA	NA
MV_forw_MV	R	differs	NA	NA
HX_FORW_OX	R	differs	NA	NA
QUEUED_FRESH	R	same	NA	NA
ONLY_FRESH	R	same	NA	NA
HEAD_FRESH	R	same	NA	NA
MID_COPY	R	same	NA	NA
TAIL_COPY	R	same	NA	NA
HW_WASH_HF	R	same	NA	NA
OF_MODS_OD	R	same	NA	NA
HF_MODS_HD	R	same	NA	NA
QF_FLUSH_IN	R	same	NA	NA

[1)] Or state ONLYP_DIRTY, if pairwise sharing supported
[2)] Or state HD_MARK_HE, if pairwise sharing supported
NA means not applicable

The *R/W* column specifies whether the cache state is read-only (*R*) or readable and writeable (*RW*). The *Clean* column specifies whether the data may be different from the copy in memory (differs) or is the same as the copy in memory (same).

For the writeable cache-line states the *write actions* column specifies actions that must be performed on the cache-line state after the data has been modified. The write may require no further actions (*done*), may require a local cache-state change (*change*), or may require purging of other sharing-list entries (*purge*). For the read-only cache-line states this column may not be applicable (*NA*).

For the writeable cache lines the *post-write-state* column specifies the new cache-line state immediately after the write has been performed. Note that writes may be performed to some of the transient states; subject to the processor's (vendor-dependent) write-instruction execution-ordering constraints, data can be accessed while sharing-list purges are being performed.

4.5.11 Instruction-execution model

For efficient cache operation, a processor must communicate the nature of its access to data as well as the address of the data. Some processors at present lack the appropriate instructions for this and must simulate them by using special addresses or instruction sequences. Generic instructions that provide the needed information are assumed in the following.

The processor is expected to check and change cache-line states before and after instructions are executed. These checks and changes are modelled by the cache-execute routines listed in table 23.

Table 26 – FullExecute Routines

name	generated by the execution of
`FullExecuteLoad()`	a *load* memory-access instruction
`FullExecuteStore()`	a *store* memory-access instruction
`FullExecuteFlush()`	the *global flush* cache-control instruction (which collapses the sharing list)
`FullExecutePurge()`	the *global purge* cache-control instruction (which returns ownership to memory, but may discard the data)
`FullExecuteCleanse()`	the *global cleanse* cache-control instruction (which copies dirty data back to memory)
`FullExecuteDelete()`	the *local flush* cache-control instruction (which deletes the local cache entry)
`FullExecuteLock()`	the *fetch&add*, *compare&swap*, and *mask&swap* instructions
`FullExecuteEnqolb()`	the *enqolb* instruction, which converts a cache line to Qolb-owned (if unowned); otherwise adds the line to the Qolb queue
`FullExecuteDeqolb()`	the *deqolb* instruction, which releases Qolb ownership of a cache line
`FullExecuteReqolb()`	the *reqolb* instruction, a more efficient equivalent of a *deqolb*/*enqolb* instruction sequence
`FullExecuteAccess()`	the privileged cache-line/memory-line locking instructions, as used by fault-recovery software

4.6 C-code naming conventions

Previous subclauses have illustrated how subsets of the coherence protocol could be implemented. The coherence specification has provisions for supporting a wide range of implementation options, and all options are designed to interoperate with each other. Rather than providing a separate specification for each allowable subset, one coherence specification is provided and the vendor is allowed to specify several static and dynamic implementation options.

Most of the coherence protocols are defined in terms of C-code routines that specify changes between cache states. Routines that initiate transitions between states include the word To between the initial and final states; the library routines that are shared by two or more of these routines include the word Do between the initial and final states, as illustrated in listing 2.

```
/*                Listing 2: Routine_names.h illustration                */

OnlyDirtyToInvalid(procPtr,mode,cTPtr); /*transition specification */
OnlyDirtyDoInvalid(procPtr,mode,cTPtr); /* library routine */
```

The cache-state names are formatted in several ways, but always begin (in the C code) with the characters `CS_`, to distinguish them from other defined constants. The stable cache state names have two other character fields that describe the sharing-list position and the cache-access rights.

Transient cache states have three other character fields: the first and last of these are abbreviations of the initial and final cache states. The middle field describes the action that is being performed, and the capitalization of the middle name describes the response to prepend and invalidate actions. If the middle field is fully uppercase, prepends and invalidations to the transient cache state are delayed. If only the first letter is uppercase, prepends are delayed but invalidations are not. If none of the letters are uppercase, neither prepends nor invalidations are delayed. These naming conventions are illustrated in listing 3.

```
/*                    Listing 3: Code_notation.h illustration        */

enum CacheStates {
   CS_LIST_ACCESS,      /* Stable cache state */
   CS_L0_ACTION_L1,     /* Transient, nullifies prepends&invalidates */
   CS_L2_Action_L3,     /* Transient, nullifies prepends */
   CS_L4_action_L5      /* Transient, accepts prepends&invalidates */
};
```

With one exception, an implementation shall behave as though all of the C code were executed indivisibly. The exception allows others to access the cache while waiting for a transaction to complete (after a request subaction has been sent and before the response subaction has been returned) or while waiting for the cache-line state to change. Specifically, the routine of listing 4 need not be executed indivisibly.

```
/*                    Listing 4: Divisible routines                   */

ChipWaitsForEvent();       /* Waiting for time or queue-state change */
```

4.7 Coherent read and write transactions

The detailed format for the coherence transactions is specified in the logical protocols, clause 3. A subset of these transactions is used to maintain cache coherence, as illustrated in table 27.

Table 27 – Coherent transaction summary

transaction name	requester, responder	description
cread/cop	proc,proc	coherent cache control/read (for prepending to dirty list)
cwrite64/cop	proc,proc	coherent cache write (for exclusive-entry deletions)
mread/mop	proc,mem	coherent memory control/read (basic and extended header)
mwrite64/mop	proc,mem	coherent memory writes
NOTE The 6-bit cop and mop coherence-command codes are the 6 LSBs of the address.		

For the noncoherent response-expected transactions (readsb, writesb, locksb, nread64, nread256, nwrite64, nwrite256) the coherence checks are bypassed. Events and responseless transactions behave the same way, except that there is no provision for returning error status.

The coherent memory transactions are also directed to memory and transfer data to or from it. In addition the coherence mode also affects the memory access (the data transfer may be nullified) and the updates of the coherent memory tags.

For the coherent memory access transactions (mread00, mread64, and mwrite64), the least significant bits of the 48-bit addressOffset specify the coherence check mode. This information, in conjunction with the basic *command.cmd* field and the identity of the requester (*sourceId*), specifies which coherent actions are performed. These fields are illustrated in figure 170.

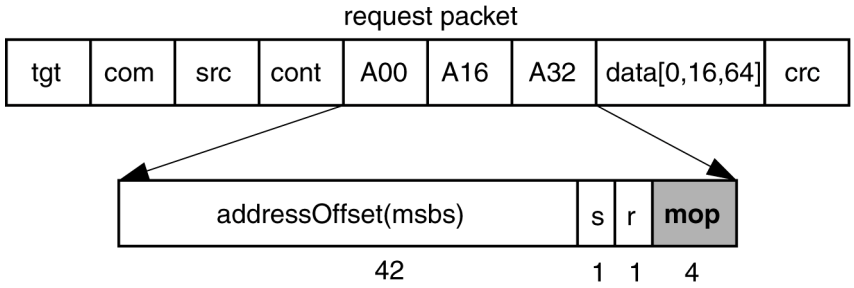


Figure 170 – Basic mread/mwrite request

The size bit, *s*, is 0 or 1 for the 0-byte mread00 or 64-byte mread64 transactions respectively. The reserved bit, *r*, is always set to zero when the request-send packet is created, but is ignored by the responder when the request-send packet is consumed.

In coherent memory transactions the previous state of the directory is always returned as status in the response subaction packet. The status includes a command-nullified field (*cn*), a 1-bit coherence-option bit (*co*), a 4-bit reserved field (*reserved*), the old memory-directory state (*mState*), and the old head pointer value (*forwId*), as illustrated in figure 171.

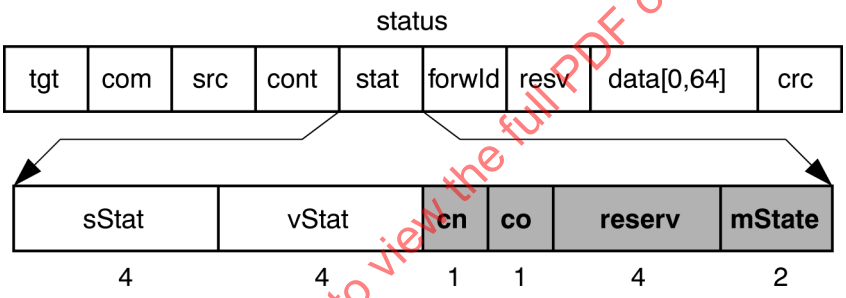


Figure 171 – Memory-access response

4.7.1 Extended mread transactions

The extended mread00 and mread64 transactions are directed to memory to perform the more complex updates of the sharing-list directory. These extended transactions pass an additional 2-byte *newId* parameter in the 16 bytes of extended header, as illustrated in figure 172. The value of the *newId* parameter influences the update of the memory tag.

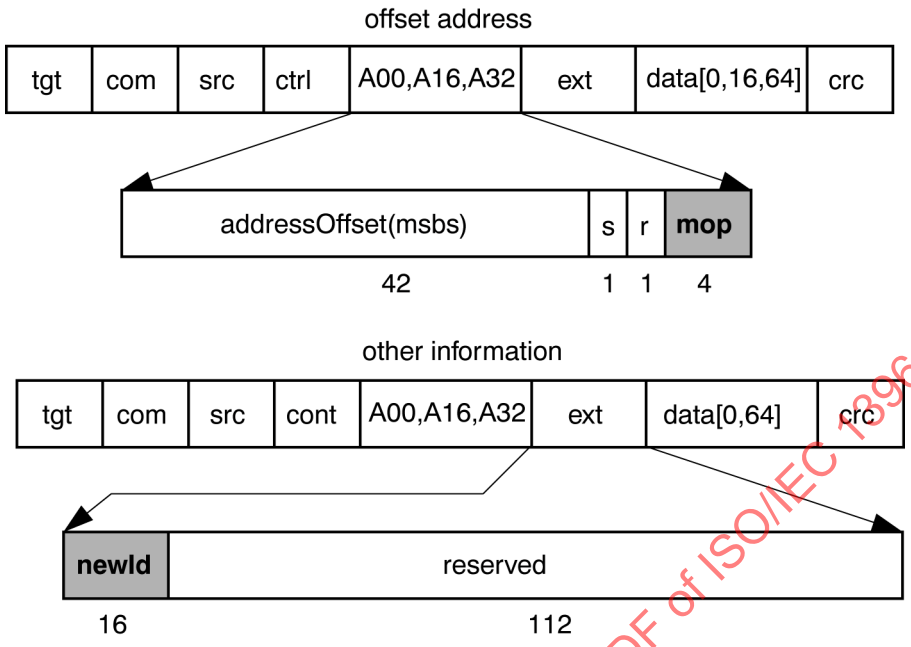


Figure 172 – Extended coherent memory read request

The *newId* parameter is used when the head deletes itself from a multiple-entry sharing list. In this case, the *newId* value is needed to identify the new sharing list head, which is different from the transaction's requester. The *newId* value is used to identify the new sharing list head in other transactions as well. This generalization is intended to support future extensions of the standard.

The *size* bit, *s*, is 0 or 1 for the 0-byte mread00 or 64-byte mread64 extended transactions respectively. The *reserved* bit, *r*, is always set to 0 when the request-send packet is created, but is ignored by the responder when the request-send packet is consumed.

4.7.2 Cache cread and cwrite64 transactions

The cache-access transactions are directed to other caches to perform the more complex updates of the distributed sharing-list. The cache-access transactions are implemented as extended cread transactions; two additional parameters are passed in the 16 bytes of extended header, as illustrated in figure 173.

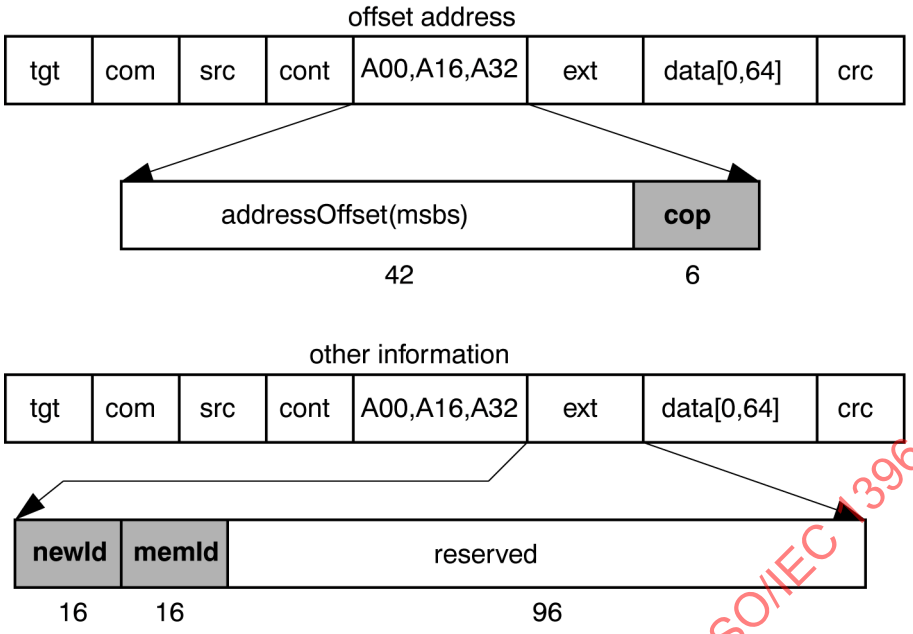


Figure 173 – Cache cread and cwrite64 requests

In the request subaction the physical cache address is specified by the addressOffset (*A00*, *A16* and *A32*) and the *memId* field. The tag update operation is specified by the cache-tag-operation, *cop*, and the *newId* field. The remaining 12 bytes of data contain 4 reserved bytes and 8 vendor-dependent bytes.

For the cread00, cread64, and cwrite64 transactions, the response status returns the previous tag state, as illustrated in figure 174.

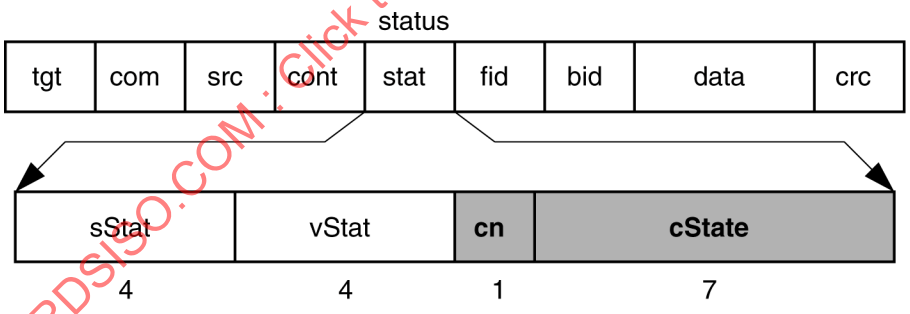


Figure 174 – Cache cread and cwrite64 responses

In addition to returning the cache's 7-bit coherence state (*cState*), two nodeId fields (*forwId* and *backId*, abbreviated as *fid* and *bid* respectively) are returned. The command-nullified bit (*cn*) is set to 1 when the cache command has not changed the cache-tag state.

4.7.3 Smaller tag sizes

Products may implement 8, 12, or the full 16 bits in the nodeId fields in their cache-tag and memory-tag storage, which are visible in several of the request and response fields (*newId*, *forwId*, and *backId*). If this subset is implemented, the upper (unimplemented) bits of the nodeId field shall be assumed to be all ones and the product shall be said to have an *n-bit nodeId configuration constraint*, where *n* is the number of bits implemented.

5 C-code structure

5.1 Node structure

5.1.1 Signals within a node

From the perspective of this International Standard, a node consists of a linc (link interface circuit) and one or more components. These components may be implemented as separate integrated circuits, or as separate portions of a single integrated circuit. This standard specifies the behaviour of these components; how they are implemented is beyond the scope of this standard.

On a node containing a single linc, the linc is expected to generate signals based on packets that it receives from its input port; these signals include *arrived* and *reset*. The *arrived* signal is generated when a *clockStrobe* packet is created or received at the input port (these may be used to synchronize time-of-day clocks on the attached units). The *reset* signal, which is generated while the linc is being *reset*, may be used to *reset* the state of attached components.

These signals are expected to be distributed to the other components on a node, as illustrated in figure 175.

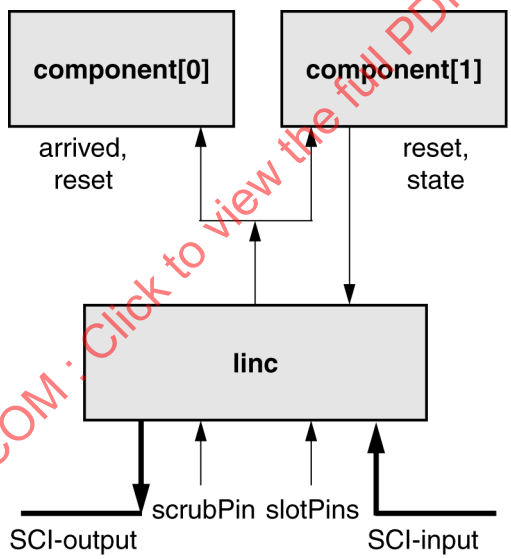


Figure 175 – Linc and component signals

From the linc's perspective, a unit may also generate signals directed to the linc. The *reset* signal is used to reset the interface to the linc. From the linc's perspective, this signal comes from one component; however, with the proper wired-OR circuitry, any component may initiate these ringlet activities.

Depending on the physical model, the linc may also be provided with static configuration-control signals, *scrubPin* and *slotPins*. The *scrubPin* signal is used to statically configure the scrubber or other node, when the node does not support a dynamic scrubber-selection process during system initialization. The (up to) 16 *slotPins* may be provided by the backplane, to provide the node with an identifier that can be read by software to determine the node's physical location.

5.1.2 Packet transfers among node components

During normal operation, these linc-related signals are not used. Communication is performed by passing packets among queues located on the linc and its associated components, using a vendor-dependent packet-transfer interconnect. Separate request and response queues are provided, to avoid queue-dependency deadlocks. The linc has receive queues, transmit queues, and CSR-access queues, as illustrated in figure 176.

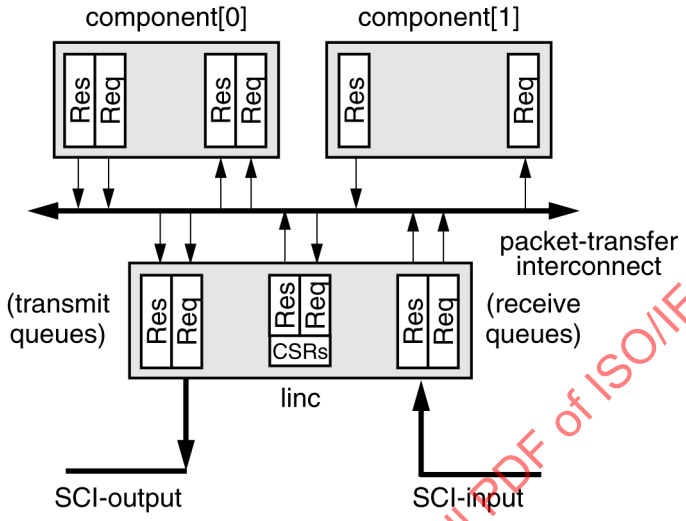


Figure 176 – Linc and component queues

In this illustration, component[0] has requester/responder capabilities and component[1] only has responder capabilities. As examples, component[0] could be a processor (which is a requester when fetching data and a responder when its cache is interrogated) and component[1] could be a memory controller (which never generates requests). Note that the linc also behaves like a responder unit, when packets are sent to its CSRs. Within the context of the logical code specification, the term "chip" is used to describe the linc and any of its attached components (which could be implemented on the same or different integrated circuits).

In many cases, a node component (which has its own queues) may be the same as a unit (which has its own control registers and state). However, a unit may consist of multiple components (a coherent multiprocessor has process and cache components) or the entire node may be viewed as one unit architecture.

5.1.3 Transfer-cloud components

The SCI standard specifies the behaviour of send packets in the linc and in a standard memory controller, as well as the behaviour of coherent traffic between processors and other caches. To accurately define the behaviour of these components without overspecifying the packet-transfer interconnect, the SCI standard is based on an abstract transfer-cloud that transfers signals and packets between the linc and other node components, as illustrated in figure 177. Within this illustration the source and destination for coherent request packets are shown with dark arrows.

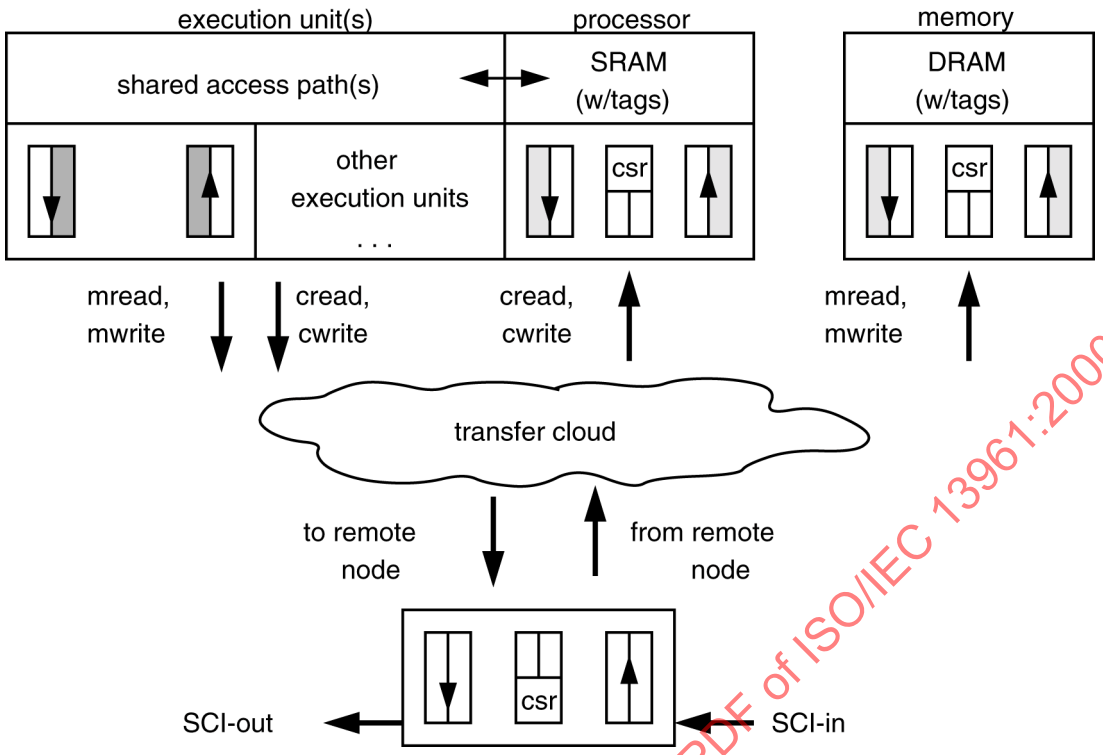


Figure 177 – One node's transfer-cloud model

The detailed specification of the cache-coherence mechanism is expressed primarily in C code so that it can be made unambiguous, can be tested thoroughly, and can be used to verify the behaviour of actual implementations. In its most general case, a coherent node contains processor, cache, and memory components. There may be one or more (superscalar) processor subcomponents that share the node's cache. Coherent memory may be located on the same node, which improves the performance of local-memory accesses.

Although some transfer-cloud behaviour is specified by the SCI standard, its detailed behaviour is vendor-dependent. A vendor may accurately simulate a variety of packet-transfer interconnects (dual request/response buses, cross-bar switches, etc.) by supplementing the code in this specification with different forms of transfer-cloud routines.

To simplify the interface between node components (such as processor, cache, memory, and linc), a standard queue structure is defined for all components (although not all queues are implemented on all components). The standard queue interface supports six queues: input-request (`IQ_REQ`), input-response (`IQ_RES`), CSR-input-request (`IQ_CSR`), CSR-output-response (`OQ_CSR`), output-request (`OQ_REQ`), and output-response (`OQ_RES`), as illustrated in figure 178.

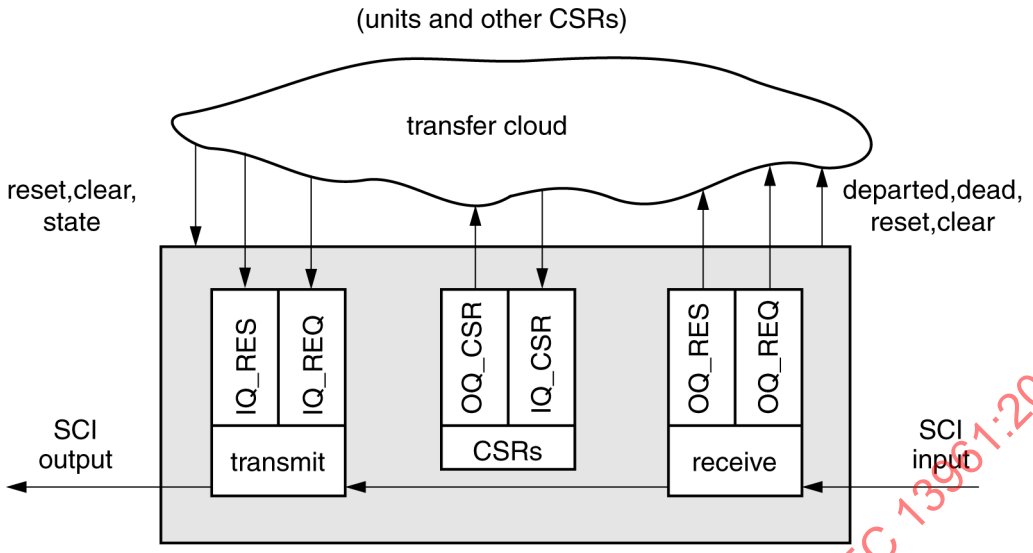


Figure 178 – The linc packet queues

The transfer cloud is expected to transfer request-send packets from the OQ_REQ queue of one component into the IQ_REQ or IQ_CSR queue of another component. Similarly, the transfer cloud is expected to transfer response-send packets from the OQ_RES or OQ_CSR of one component into the IQ_RES queue of another.

Note that a request-send packet may be transferred from an OQ_REQ queue to an IQ_CSR queue on the same component (as is done on the linc). Similarly, a response-send packet may be transferred from an OQ_CSR queue to an IQ_RES queue on the same component.

5.2 A node's linc component

5.2.1 A linc's subcomponents

This International Standard specifies the functional behaviour of an SCI node, but does not define how this behaviour shall be implemented. The operation of an SCI node is illustrated by the hardware blocks described in this clause; the functional behaviour is specified by the C code. Although the implementation shall have the same observable behaviour as the C-code specification, the implementation may implement other hardware structures than those illustrated in this clause or may partition the design differently from the C-code routines.

A linc is expected to have paired transmit and receive queues, each having one queue for requests and one queue for responses (as shown in figure 179). In addition, there is a bypass FIFO, which delays pass-through packets while a transmit-queue packet is being sent. To accept packets of *N* symbols the receive queues must be at least *N* symbols long. To send (request or response) packets of *N* symbols the corresponding (request or response) transmit queue must be at least *N* symbols long. For certain options, the bypass FIFO needs to be one symbol longer than the maximum transmitted packet. (The extra space is needed for inserting an idle between back-to-back packets, to support the elastic-buffer protocols).

The receive hardware (which is located near the upstream or incoming port) contains an elastic buffer, a parser and a stripper. The elastic buffer synchronizes the input clock to the local clock, inserting or deleting idle symbols as required. The parser uses the flag bits on incoming symbols to label symbol types within each packet. The stripper checks the nodeId and strips those packets addressed to this node.

The transmit hardware (which is located near the downstream outgoing port) contains a bypass FIFO, a *saveIdle* buffer, a multiplexer, and a CRC encoder. The bypass FIFO is used to save incoming packets while other symbols are being output. The *saveIdle* buffer is used to save the needed information from incoming idle symbols while other symbols are being output. Note that multiple incoming idle symbols can be merged and saved in the single-symbol *saveIdle* buffer.

Each node requires one CRC checker and one CRC generator. The stripper has a CRC checker, which delays processing of a node's received echo until its CRC has been verified. The stripper is also responsible for marking CRC errors in packets that pass through the node or are saved within it. The transmitter is responsible for creating new CRC values for new packets in the queue or for newly created echoes. The transmitter is also responsible for creating stomped CRC values when CRC errors are detected by the stripper. These functional components are illustrated in figure 179.

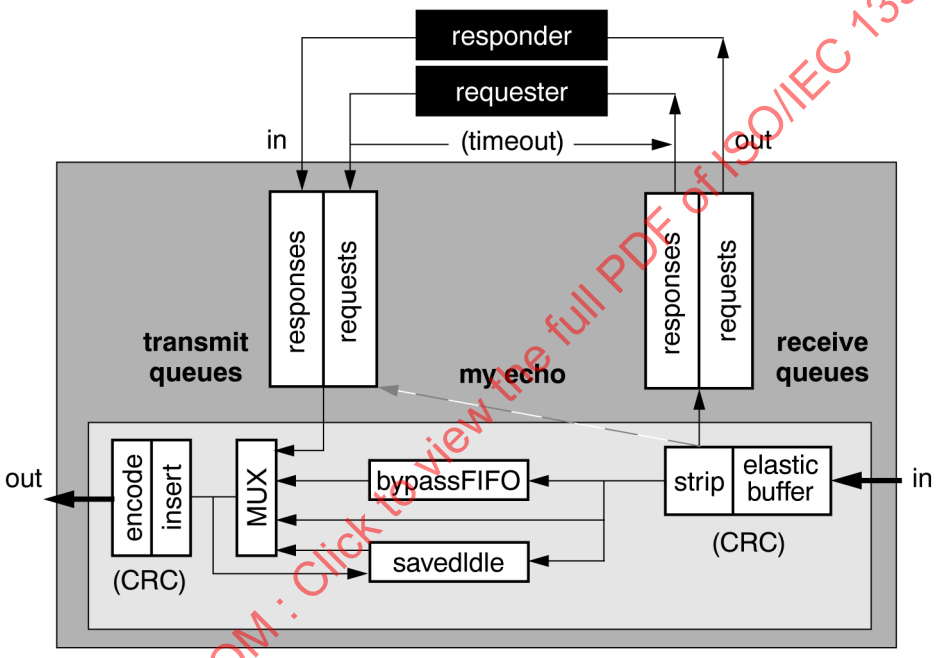


Figure 179 – Node interface structure

The node interface model has two high-speed receive queues, so that request and response packets can immediately be accepted before being processed. When receive queues are filled, fair nodes selectively accept packets based on their age, using a simple A/B ageing scheme.

Selective acceptance (queue allocation) protocols ensure that all packets are eventually accepted. Independent processing paths for requests and responses ensure that all queued sends are eventually processed. Thus, two receive queues are sufficient to ensure forward progress. Nodes may provide additional queue entries so that additional packets can be queued while others are being processed. The number of implemented receive queues is based on system performance requirements, which are beyond the scope of this International Standard.

Incoming data passes through an elastic buffer that inserts or deletes idle symbols based on the phase difference between the incoming data and the node-local clock. To avoid corruption of packets, only idle symbols (or an intermediate symbol of a sync packet) may be deleted, and only idle symbols may be created (and only between packets). Special symbol information is passed from the elastic buffer to the stripper, to identify symbols that have been inserted and to pass residual information from symbols that have been deleted.

The elastic buffer's output passes to the stripper, which labels and selectively strips packet symbols. The stripper uses the flag-signal value to identify and label the packet symbols (`SS_HEAD0`, `SS_HEAD1`, etc.). The stripper strips selected packets and, for robustness, truncates overly-long accepted packets (to avoid overflowing the initially allocated receive-queue space). The stripper is also responsible for detecting excessively long pass-through packets (generated by fatal transmission errors) and for clearing the ringlet when such errors are detected.

The stripped symbol stream is routed to a receive-queue entry or (if the queue is full) is discarded. Packet symbols that are not stripped are routed into the bypass FIFO or are multiplexed directly to the output. Idle symbols are saved, modified, or directly routed to the output. A separate *saveIdle* storage is used for saving the information from the idle symbols (which can be consumed and regenerated) and the packet symbols, which sometimes pass through the bypass FIFO.

Fair nodes are expected to use FIFO queues, processing the packets in the order in which they are received. Prioritized nodes are expected to change the processing order of packet entries in the receive or transmit queues, based on their packet priorities. When receive queues become full, prioritized nodes are also expected to selectively accept packets based on their priority and their age. Some fairness is required in the re-ordering and acceptance protocols to ensure forward progress for lower-priority transmissions.

5.2.2 A linc's elastic buffer

The elastic buffer code specifies how idle symbols are inserted or deleted when the input clock drifts from the local clock reference. The `Elasticity()` routine uses its previously calculated delay value (*myDelay*) to control a fraction-of-a-cycle delay block (*variableDelay*). The *flag* associated with the output symbol value provides an input to the control state machine which (along with bits within the next symbol) determines when idle symbols can be inserted and deleted. These functional blocks are illustrated in figure 180.

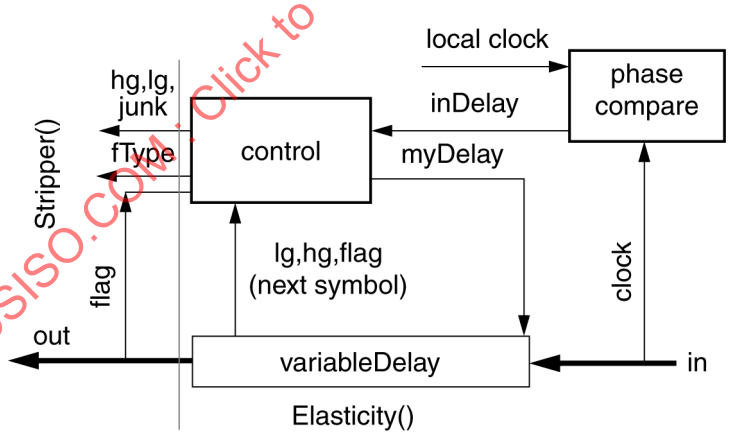


Figure 180 – Elasticity model

The `Elasticity()` routine's output symbol, *out*, is processed by the `Stripper()` routine. These symbols are labelled to identify the *out* symbols that should be ignored (they were created by idle-insertions or they are the tail of a stripped sync packet). When idles are deleted, their *idle.hg* and *idle.lg* bits are passed to the `Stripper()` routine and propagate to the transmitter.

5.2.3 Other linc components

The `Stripper()` is responsible for selectively copying the desired packets into the receive buffers and labelling packet symbols for other downstream components within the linc.

The `Inserter()` routine is responsible for inserting initialization packets in the output symbol stream during the node's initialization process.

The `Transmitter()` routine is responsible for selecting packets from the transmit buffers when the input link is idle and transmissions are enabled. On unfair nodes, the `Transmitter()` also updates the priority of the idles and selective send packets based on the linc's blocked-transaction priorities. Note that the priorities of packets emanating from the linc's own source FIFOs are not updated, so that the priorities on the local ringlet can be more accurately sampled.

The `Encoder()` routine is primarily responsible for re-inserting the flag symbol (`FT_LOW` or `FT_HIGH`), based on the symbol labels generated by the `Stripper()` and (possibly) modified by the `Inserter()` or `Transmitter()` routines. The `Encoder()` routine also provides the check value for idle symbols and generates the CRC for new packets.

While the node is initializing, the `Transmitter()` component is idle and the `Elasticity()`, `Stripper()`, and `Encoder()` components remain operational.

5.3 Other node components

5.3.1 A node's core component

Since the *cloud* is only an agent for performing data transfers, it has no I/O queues. However, each component on a cloud has a pointer to a shared node component, called the *core*. The core data structure also contains a bit map of the outstanding transaction identifiers (*transIdBits*), and provides queues for allocating this transaction-identifier resource.

The node's CSRs may be located on the linc or on other components connected to the transfer cloud. The CSRs that intimately affect the linc's behaviour (`STATE_CLEAR`, `STATE_SET`, `NODE_IDS`, `RESET_START`, `CLOCK_STROBE_THROUGH`, and `SYNC_INTERVAL`) are expected to be located on the linc. Other registers, such as the `SPLIT_TIMEOUT` and `CLOCK_STROBE_ARRIVED` CSRs, can be accessed through the transfer cloud and may be located on other node components.

5.3.2 A node's memory component

To provide common support for other SCI requesters (typically processors and DMA), the SCI standard specifies the functional behaviour (but not the performance) of standard memory unit architectures. The specification code assumes that two memory queues are needed, one to hold request-send packets (`IQ_REQ`) and another to hold response-send packets (`OQ_RES`), as illustrated in figure 181.

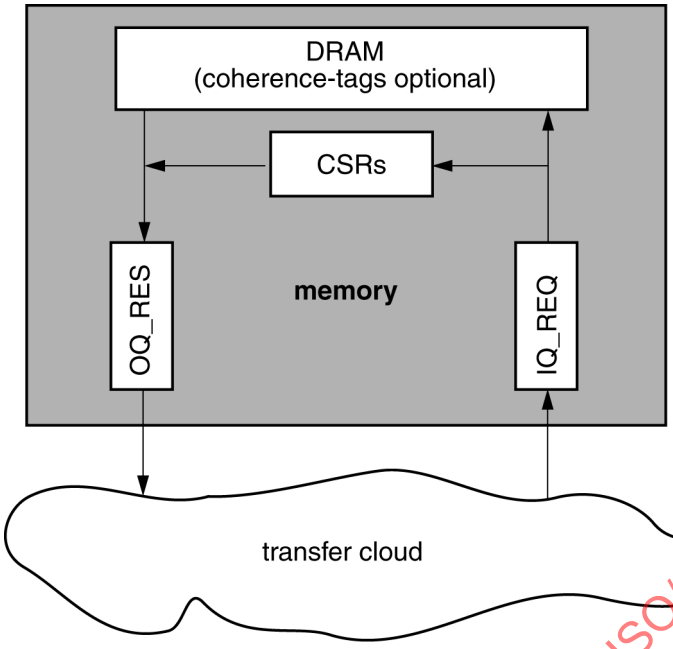


Figure 181 – A memory component's packet queues

A memory unit is not expected to have output-request or input-response queues, since it typically has no need to generate transactions. A minimal implementation shall provide at least one queue entry, to be shared between input-request and output-response queues.

One of the `GONE`, `FRESH`, and `WASH` option bits is set when a coherent-memory option is specified. The `GONE` option minimizes the tag storage requirements (the tag is saved as data when a line is coherently cached). The `FRESH` option supports efficient fetching of coherently cached read-only *fresh* data, which is returned immediately from memory. The `WASH` option supports the efficient conversion of data from the *dirty* state (the data must be fetched from a processor) to a *fresh* state (the data may be fetched from memory).

The `MemoryAccessBasic()` routine performs the requested RAM-update action and returns the requested data. The `DoLocks()` routine implements the coherent memory lock operations. An SCI memory controller shall implement the defined lock operations as specified by this routine, but may also provide additional support for a vendor-dependent lock operation.

5.3.3 A node's exec component

The SCI standard specifies the functional behaviour (but not the performance) of standard processor execution-unit architectures. The specification code assumes that two processor queues are needed to support each processor-initiated transaction, one to hold outgoing request-send packets (`OQ_REQ`) and another to hold incoming response-send packets (`IQ_RES`), as illustrated in figure 182. Note that one processor (proc) may have many attached execution units (execs).

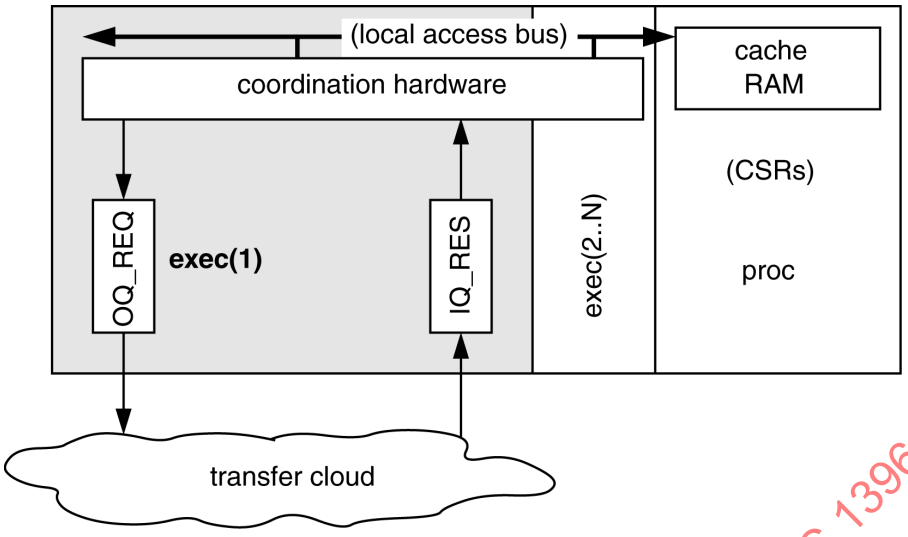


Figure 182 – An exec component's packet queues

5.3.4 A node's proc component

To provide common support for coherent SCI processors, the SCI standard specifies the functional behaviour (but not the performance) of a standard cached processor unit architecture, abbreviated as proc. The specification code assumes that two cache queues are needed, one to hold request-send packets (`IQ_REQ`) and another to hold response-send packets (`OQ_RES`), as illustrated in figure 183.

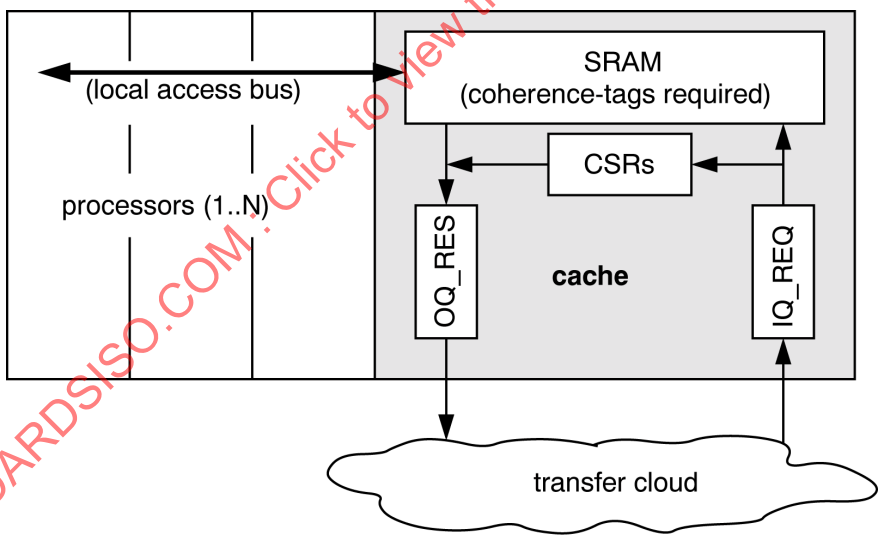


Figure 183 – A proc component's packet queues