

WITHDRAWN

IEEE Std 1596.5™-1993 (Withdrawn 2005)

IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors

Sponsor

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

Withdrawn 25 January 2005

IEEE-SA Standards Board

* Withdrawn standards are standards are no longer useful or contain significant obsolete or erroneous information. All amendments or corrigenda associated with this standard have also been withdrawn. This standard has been withdrawn as an American National Standard (ANSI).

WARNING: In using this standards, you do so AT YOUR OWN RISK, as the technology contained herein may contain significant obsolete or erroneous information, or may have been superseded since the date of withdrawal. IEEE neither represents nor warrants that any information in this standard is either useful, in use by industry, current, without error, or applicable for any particular purpose.

Information on withdrawn standards is available in the IEEE-SA Standards Board Operations Manual
<<http://standards.ieee.org/guides/opman/index.html>>.

WITHDRAWN

Abstract: Formats for interchanging integer, bit-field, and floating-point data between heterogeneous multiprocessors in a Scalable Coherent Interface (SCI) system are specified. The defined data formats can also be used to share data among multiprocessors on other bus standards that support the read, write, and lock transaction set defined by IEEE Std 1212-1991 CSR Architecture. The intent is to support efficient data transfers among heterogeneous workstations within a distributed computing environment.

Keywords: data formats, data interchange, floating-point, heterogeneous multiprocessors, integer, shared data.

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

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

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Incorporated.

ISBN 1-55937-354-7

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

WITHDRAWN

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

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

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

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

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

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

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

Introduction

[This introduction is not a part of IEEE Std 1596.5-1993, IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors.]

The demand for more processing power continues to increase and apparently has no limit. The resources of any computer can be exhausted by merely specifying a finer mesh or higher resolution for the solution of some physical problem (hydrodynamics, for example), with the result that engineers and scientists need increasingly more powerful computers.

To get this kind of computing power, it seems necessary to use a large number of processors cooperatively. IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface (SCI), specifies a physical interconnect that supports large numbers of closely coupled or tightly coupled multiprocessors. Since many SCI systems are expected to contain heterogeneous multiprocessors, there is a need to standardize the formats of shared data structures. Otherwise, computational power could be limited by the inefficiency of performing complex conversions between different data formats.

Although heterogeneous shared multiprocessors benefit most from this standard, it should also be valuable for others who routinely distribute or share data. Many signal-processing programmers have wasted innumerable hours converting data from one ambiguously defined data format to another. This standard provides a framework for having compilers automatically support conversions.

This standards activity was initiated after discussions between the Chair of the IEEE P1596.5 Working Group, David V. James, and Ronald G. Minnich of the Supercomputer Research Center. Minnich observed that a tightly-coupled heterogeneous multiprocessor would be complex and inefficient if special routines were needed to convert between different data types.

David B. Gustavson, chairman of the P1596 scalable coherent interface (SCI) project, provided invaluable administrative assistance as well as technical review and encouragement for the P1596.5 working group.

Andre Bogaerts provided valuable input, observing that the least significant bits of virtual and physical addresses ought to be the same.

Jerry Huck provided valuable input on the complexities involved with standardizing virtual addresses.

Professor Kahan identified the need to document commercially significant data formats. He also provided documentation and contacts to assist the editor in generating this standard. These contacts, Mary H. Payne, Fred Ris, and James M. Kiernan, reviewed the initial drafts of the vendor-dependent floating-point data formats.

Mario Nemirovsky provided valuable off-line review of this document, including the development of the aligned/unaligned and atomic/permissive data format concepts.

Most importantly, IEEE Std 1212-1991, IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses, and IEEE Std 1596-1992 provided a well-defined framework upon which this document is based. The members of those working groups have indirectly contributed to making it possible for this related document to be developed in a short period of time.

The following is a list of participants in the IEEE P1596.5 Working Group. Voting members at the time of publication are marked with an asterisk (*).

David V. James*, *Chair*

Duane L. Anderson*
Sam Babb
Andre Boegarts*
Del Cecchi*
Mike Chastain*
Pete Fenner*
Stein Gjessing*
James Goodman*

David B. Gustavson*
Sverre Johansen*
Ross Johnson*
Ernst Kristiansen*
Stein Krogdahl*
Gerry Laws
Rogerio Miranda*
Ellen Munthe-Kaas*
Mario Nemirovsky*

Dan O'Connor*
Keith Rich
Tim Scott*
Joanne Spiller*
Glen Stone*
Hans Wiggers*
Janusz Zalewski
Mike Ziegler*

WITHDRAWN

The following persons were on the balloting committee:

William B. Adams
Knut Alnes
Kurt Baty
Andre Bogaerts
Haakon Bugge
Robert Crowder
Dante Del Corso
Roberto Divia
Mike Dorsett
Sourav Dutta
Wilhelm P. Evertz
Stein Gjessing
Julio Gonzalez-Sanz
Willard Graves
David B. Gustavson
Jerry Huck
Zoltan R. Hunor

David V. James
Kenneth A. Jansen
Ross Johnson
W. Kahan
Ramesh Krishnaiyer
Ernst H. Kristiansen
Thomas M. Kurihara
Glen Langdon
Gerry Laws
Rollins Linser
Stephen Mabbs
William McDonald
James D. Mooney
Klaus Dieter Mueller
Mario Nemirovsky
Daniel C. O'Connor
Tsugihiko Ohno

Elwood Parsons
Robert K. Rouse
Ravi Shankar
Kerry Shore
Michael Teener
Manu Thapar
Michael G. Thompson
Robert Tripi
Rudolf Usselmann
Yoshiaki Wakimura
Eike Waltz
Richard J. Westmore
C. Whitby-Stevens
Andrew Wilson
David L. Wright
Oren Yuen
Janusz Zalewski

When the IEEE Standards Board approved this standard on June 17, 1993, it had the following membership:

Wallace S. Read, *Chair*
Donald C. Loughry, *Vice Chair*
Andrew G. Salem, *Secretary*

Gilles A. Baril
José A. Berrios de la Paz
Clyde R. Camp
Donald C. Fleckenstein
Jay Forster*
David F. Franklin
Ramiro Garcia
Donald N. Heirman

Jim Isaak
Ben C. Johnson
Walter J. Karplus
Lorraine C. Kevra
E. G. "Al" Kiener
Ivor N. Knight
Joseph L. Koepfinger*
D. N. "Jim" Logothetis

Don T. Michael*
Marco W. Migliaro
L. John Rankine
Arthur K. Reilly
Ronald H. Reimer
Gary S. Robinson
Leonard L. Tripp
Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
James Beall

Richard B. Engelman

David E. Soffrin
Stanley I. Warshaw

Paula M. Kelty
IEEE Standards Project Editor

WITHDRAWN

CLAUSE	PAGE
1. Overview	1
1.1 Document scope and purpose	1
1.2 Document structure	2
1.3 Historical perspective	2
1.4 Objectives	5
1.5 Other data formats	6
1.6 Processor design models	6
1.7 Byte-ordering models	10
2. References, glossary, and notation	12
2.1 References	12
2.2 Conformance levels	12
2.3 Glossary of terms	12
2.4 Bit and byte ordering	14
2.5 Integer format notation	15
2.6 Real format notation	16
2.7 Numerical values	16
2.8 Glossary of data formats	16
2.9 Glossary of data-format containers	17
2.10 Data-format collections	18
3. Big-endian formats	22
3.1 Big-endian signed integers	22
3.2 Big-endian unsigned integers	24
3.3 BigSignedField values	26
3.4 BigUnsignedField values	28
3.5 Big-endian real numbers	30
4. Little-endian formats	32
4.1 Little-endian signed integers	32
4.2 Little-endian unsigned integers	34
4.3 LittleSignedField values	37
4.4 LittleUnsignedField values	38
4.5 Little-endian real values	40
5. Data format summary	42
5.1 Integer formats	42
5.2 Real formats	46
5.3 AlignedLinelet	47
6. Data-access operations	48
6.1 Supported data-access operations	48
6.2 Atomic and permissive data-access operations	49
6.3 Atomic data-access inheritance	51
7. Conformance levels	51

WITHDRAWN

CLAUSE	PAGE
7.1 Transaction set conformance	51
7.2 Subset conformance	52
Annex A (informative) Bibliography.....	54
Annex B (normative) Common test&lock data-access capabilities.....	55
Annex C (informative) Expected C-language support.....	60
Annex D (informative) Other floating-point formats	64
Annex E (informative) C-code overview	71

WITHDRAWN

formats and vendor-dependent data formats. In the long term, compilers and processors are expected to efficiently support these shared-data formats.

This standard can also be used by systems based on other bus standards. The standard's scope is restricted to SCI in order to provide a well-understood design framework. The results, however, are applicable to any bus standard that supports the read/write/lock transaction set defined by IEEE Std 1212-1991 [B4].

1.2 Document structure

This document is partitioned into clauses that serve several distinct purposes.

Clause 1, Overview, provides background for understanding the goals and objectives for this document.

Clause 2, References, glossary, and notation, gives the references referred to in this standard and provides a glossary, which defines the terms used by later clauses within this document.

Clause 3, Big-endian formats, define sets of data objects optimized for use by big-endian processors.

Clause 4, Little-endian formats, define sets of data objects optimized for use by little-endian processors.

Clause 5, Data-format summary, provides concise formulae for specifying the defined data formats.

Clause 6, Data-access operations, specifies which data-access operations are supported by each data format and specifies how uncached read, write, and lock transactions are used.

Clause 7, Conformance levels, specifies levels of conformance for the data formats defined by this standard.

Annex A, Bibliography, lists background material for this standard.

Annex B, Common test&lock data-access capabilities, defines the behavior of the test&lock data-access operation (a common semaphore operation for multiple instruction sets) and defines the behavior of uncached bus transactions. These behavior characteristics are formally specified using C code.

Annex C, Expected C-language support, illustrates how C programs are expected to support the data formats and data-access operations defined within this shared-data formats standard.

Annex D, Other floating-point formats, describes commonly used Digital Equipment Corporation, Cray, and IBM data formats.

Annex E, C-code overview, describes the contents of the C code on the enclosed diskette.

1.3 Historical perspective

1.3.1 Multicomputer data sharing

Historically, most computation was performed on a single processor, or by cooperating processors of the same type. In that environment, the processor that produced the data was identical to the processor that consumed the data. Within such constrained environments, there was little need to standardize data formats; many of the default data-format properties (such as endian order or alignment) were chosen by the compiler, which typically selected the most efficient option.

Within large installations, different types of computers are now being purchased to perform different tasks. On large physics experiments, for example, the computer that performs the data acquisition is often different from the computer that performs the data analysis. To avoid confusion, the acquisition and analysis computers agree on the format of header and data blocks for tape storage, as illustrated in the left half of figure 1.

A similar problem exists when multiple processors are connected to a shared network, as illustrated in the right half of figure 1. Higher level network protocols have standardized the contents of packet headers so

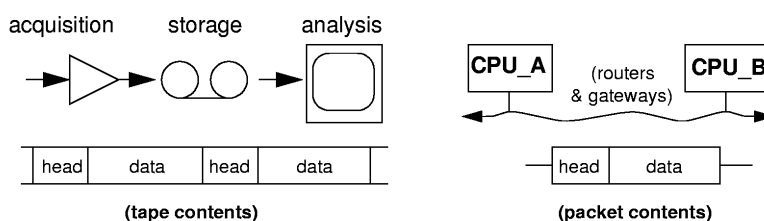


Figure 1— Data interchange models

that packets can be efficiently processed by generic routers and gateways. The data-packet formats need not be standardized when application-specific data is sent, but are often standardized when the data block contains network-service commands.

Although network standards have specified the format of packet headers, special routines are needed to translate between these standard formats and vendor-dependent data formats. As an example, the `htonl()`, `htons()`, `ntohl()`, and `ntohs()` routines in the Berkeley version of the UNIX^{®2} network software are used to convert between network and VAX-11/780 data formats.

1.3.2 Multiprocessor data sharing

Data interchange problems also arise within the context of a single system, when the I/O device or the media are provided by different vendors, as illustrated in figure 2. To ensure the early availability of the widest range of I/O interface boards, SCI and other standards have been heavily influenced by the need to support other I/O buses.

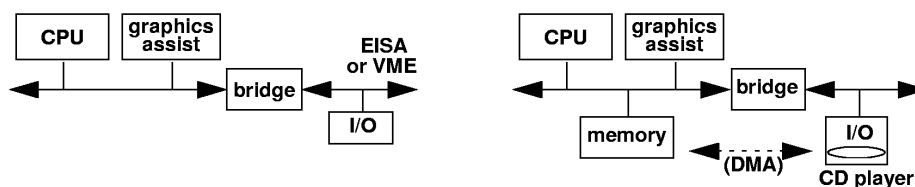


Figure 2— Multiprocessor sharing models

The availability of I/O functions is increasingly mandating the selection of I/O buses. Big-endian processors sometimes need to access little-endian control registers on I/O buses like EISA and little-endian processors sometimes need to access big-endian control registers on I/O buses like VME.

If supported by the C compiler, the data formats defined by this standard could simplify the creation of processor-independent I/O driver software. Data structures would accurately specify the format of an I/O device's control registers, so that memory-mapped accesses of 2-byte control and status registers could be performed directly and unambiguously. Indirect manipulation of fields (within the context of I/O driver storage) and sharing of FIFO data buffers would also be possible, as illustrated by the C-code header of table 1.

The data format specifications affect how the data is accessed (a 2-byte load is used to fetch a `BigUnsignedDoublet`) as well as the interpretation of the data. For example, the absolute value of an integer depends on whether a signed or unsigned data format is assumed.

Similarly, data interchange problems arise when the media and the processor's natural data formats differ. For example, the data formats on mass-distributed CD disks are likely to conflict with the natural data formats assumed by some processors.

²@UNIX is a registered trademark of AT&T.

WITHDRAWN

Customers are expected to support tightly coupled, shared-memory multiprocessors within future system configurations. The selection of the optimal processor instruction-set architecture may depend on its attachment point (SCI [B6], VMEbus [B3], Serial Bus [B5], etc.) as well as its application (for example, does it execute the operating system code, execute special graphics software, or is it an I/O processor?). Thus, even tightly coupled multiprocessors are likely to include collections of heterogeneous processors.

1.3.3 Data-format conversions

IEEE Std 754-1985³ simplified the task of converting between different floating-point data formats, by standardizing the radix and size of the exponent, the format of denormalized numbers, and the formats of the special zero and infinity values. Although that standard accurately defines the meaning of several floating-point formats, it refrains from specifying the alignment or byte-order characteristics of the floating-point values.

A similar ambiguity exists for integer data, whose alignment and byte-order characteristics have not been standardized. This complicates the data interchange problem, because the default data formats assumed by compilers are instruction-set dependent and poorly documented. Application programmers have had to explicitly translate between external data formats and the “natural” formats supported by their compiler. These conversion programs often have to be rewritten for each new machine, since the new compiler’s *natural* formats are often different.

Table 1— Defining I/O control registers

```

/* This code fragment illustrates how the I/O device's control and status
 * registers could be declared in a instruction-set insensitive manner.
 * The final IoRegisters typedef defines the control register structure.
 * The previous typedefs describe the fields within this structure. */

typedef struct {                /* Control register fields */
    unsigned reset:1;          /* as used within the context */
    unsigned command:15;       /* of local I/O driver variables */
} ControlFields;

/* The control register is always accessed as a whole entity
 * but the control register fields are used within I/O driver software.*/
typedef union {                /* Specifies fields and size */
    AlignedBigDoublet full;    /* Read/write access of all fields */
    ControlFields ctrl;        /* For internal driver usage */
} ControlType;

typedef struct {                /* Status register fields */
    unsigned ready:1;          /* as used within the context */
    unsigned others:15;       /* of local I/O driver variables */
} StatusFields;

typedef union {                /* Status register definitions */
    AlignedBigDoublet full;    /* Read/write access of all fields */
    StatusFields stat;         /* For internal driver usage */
} StatusType;

```

³Information on references can be found in 2.1.

Table 1— Defining I/O control registers (Continued)

```

typedef struct {
    /* Stereo data samples are */
    BigUnsignedDoublet left;    /* represented as big-endian */
    BigUnsignedDoublet right;   /* signed 2-byte integers */
} StereoSamples;

typedef struct {
    /* I/O device's CSR structure */
    ControlType control;        /* Control register fields */
    StatusType status;          /* Status register fields */
    BigUnsignedDoublet
    dataIndex;                  /* Last location written by device */
    BigUnsignedDoublet
    userIndex;                  /* Last location read by system */
    StereoSamples data[512];    /* FIFO data storage itself */
} IoRegisters;

```

1.4 Objectives

The efficiency of sharing data will increasingly impact system performance within such heterogeneous, shared-memory, multiprocessor environments. To efficiently share data, a standard set of data formats must be defined. The applications must then agree on the data structures (collections of these data formats) that are to be used. Also, compiler support is needed to efficiently support these data formats. This standard represents the first step in this process, by defining a standard set of data formats.

The primary purpose of this standard is to unambiguously specify the formats of data shared among heterogeneous multiprocessors. This specification should allow future compilers and processors to efficiently support these formats. The following lower-level objectives are intended to clarify this higher-level objective:

- a) *Passively shared.* When their data types are properly specified, data structures that are written by one processor may be read unambiguously (although not concurrently) by another. Passively shared data is expected to be a *permissive* data type, which means that a multistep update sequence is allowed.
- b) *Actively shared.* When their shared data types are properly specified, control variables (i.e., semaphore data values) may be concurrently shared by other processors. Actively shared data is expected to be an *atomic* data type, which means that the data update is performed indivisibly.
- c) *I/O device support.* When their data types are properly specified, control registers on existing memory-mapped I/O devices (such as VME [B3]) may be directly accessed at memory-mapped addresses. Note that read1, read2, read4, write1, write2, and write4 transactions are needed to access existing 1-byte, 2-byte, and 4-byte VME control registers.
- d) *Extendable.* The defined data types should support the next generation of processors. This includes support for 64-bit addresses, 128-bit integers, and 128-bit floating-point values.
- e) *Consistent.* The defined data types should be compatible with the transaction set defined by IEEE Std 1212-1991 [B4].
- f) *Cache transparency.* The data types should support the same data-access operations in uncached and coherently cached address spaces.
- g) *Efficiency.* Formats can be grouped into cache lines, to minimize the effects of false sharing and semaphore thrashing.

WITHDRAWN

1.5 Other data formats

1.5.1 Character strings and arrays

This standard does not define the format of multiple-element structures, such as of character strings or arrays. Characters are expected to be represented as unsigned 1-byte or 2-byte integers, which are used as an index for specifying a printable or control character. Other standards (such as ISO 646:1991 [B7]) define the meaning and function of these table entries, which are beyond the scope of this standard.

Character strings are expected to be an array of characters. However, the format of these arrays (termination character, length-specification formats, etc.) is language dependent and beyond the scope of this standard. However, the data types defined in this standard support unambiguous specification of the individual elements within arrays.

Arrays are expected to be collections of data elements, where each element is a collection of data in the formats defined by this standard. The address and size of an array element is expected to be a multiple of c , where c is the size of the largest aligned data format within that element.

1.5.2 Address pointers

This standard restricts the address of some data items, in that the address of an item with an aligned format is a multiple of the data format's size. The term *address* is used to ambiguously reference the data format's virtual, offset, or physical address.

The format and size of a data item's virtual, offset, and physical address may differ, but all are expected to be 4- or 8-byte integer values. This expectation is reflected in the size of the 4-byte and 8-byte compare&swap/fetch&add operations that are supported.

For any one data format, the 6 least significant bits of all address variants (which represent the address-offset within a cache line) are expected to be the same. This expectation is reflected in the definition of the AlignedLinelet, whose base address is always 64-byte aligned.

Processor-resident software is expected to use **virtual addresses**. In some cases, sign extensions could be used to convert a 32-bit virtual address into its 64-bit virtual address equivalent. In other cases, the upper portion of the 32-bit address could be a segment number that specifies the most significant portion of an internal 64-bit logical address. Given the variety of virtual addressing conventions, detailed specifications of the size and format of virtual addresses are beyond the scope of this standard.

An **offset address** is expected to be used within some shared-data objects; the offset address of a format is its byte offset from the 64-byte-aligned base of the shared-data object. Endian order is expected to be consistent with the data structures being shared, or big-endian address offsets are expected to be used when mixtures of little- and big-endian data formats are being shared.

All bus transactions are expected to use **physical** addresses, and DMA data transfers are expected to be specified as collections of byte-aligned physical address segments.

1.6 Processor design models

1.6.1 Data-access paths

In defining the access capabilities for shared-data items, the processor design model in figure 3 has been assumed.

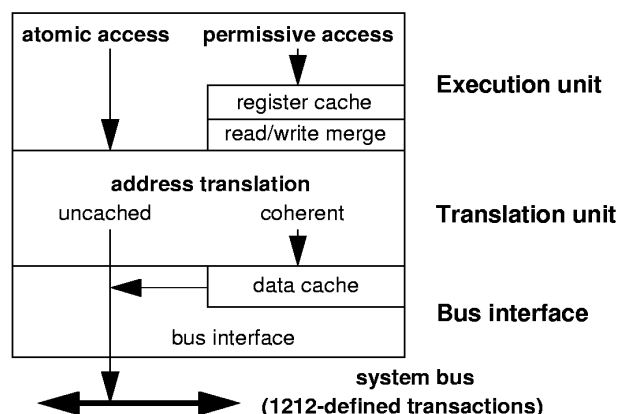


Figure 3— Processor design mode

When data is accessed, processors are expected to provide hardware/software directives to determine whether an atomic or permissive access should be used. An **atomic** access directly generates one (and only one) data-access operation; software-managed register caching, and read/write merge buffers, are inhibited. A **permissive** access may be performed by updating a software-managed register cache, fetching or merging the data in a read/write buffer, or generating multiple data-access operations.

Special tagged-memory architectures are not needed for distinguishing between atomic and permissive data types. The distinction between these data types is expected to be reflected in higher-level languages, page table structures, and/or the processor's instruction set.

A data-access operation is expected to involve either uncacheable or coherently cacheable data. An **uncached** data-access operation generates an uncached read, write, or lock bus transaction. A **coherent** data-access operation operates on coherently cached data; in some cases a sequence of bus transactions is needed to fetch the requested data, to convert the cache-line state, or to purge other copies after the data has been modified.

1.6.2 Cache-line alignment

Since cache-line placement can significantly affect system performance, a cache-line collection (called an AlignedLinelet) is defined and is expected to be utilized by applications and/or compilers in the following ways:

- To minimize performance losses caused by false sharing, data formats are expected to be grouped within or outside of selected AlignedLinelet locations.
- Stack base addresses are expected to be 64-byte aligned, so that stack-local data can be cache-line aligned.
- Applications using cache-line migration directives, such as SCI's *enqolb* and *deqolb* operations, are expected to apply the QOLB-lock to a cache line containing the affected lock-variable and (when possible) to place the affected data within the same cache line.
- An atomic write to an uncached AlignedLinelet address generates a write64 transaction, which is needed to send messages between the nodes. This message-passing capability is an option defined within the CSR Architecture.

1.6.3 Data-access operations

The data-access operations assumed by this standard include load, store, and lock operations. These data-access operations are performed on a *data* value and (for loads and locks) involve returning the *last* data value. For store and lock operations, the processor-provided *next* and *test* values are used to update the addressed data value, as illustrated in figure 4.

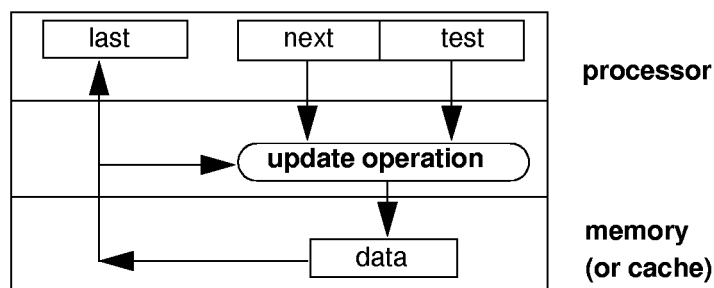


Figure 4— Data-access operation model

In all cases, the formats of the data-access parameters (*next*, *test*, and *last*) are the same as the accessed *data* value. The update operations assumed by this standard are listed in table 2. Some of the data formats do not support all of these; see 6.1 for details.

Table 2— Data-access operation updates

Data access	Update operation
load	<code>last = data;</code>
store	<code>data = next;</code>
mask&swap	<code>last = data; data = (next & test) (data & ~test) ;</code>
compare&swap	<code>last = data; if (test != data) data = next;</code>
fetch&add	<code>last = data; data += next;</code>

The mask&swap operation is expected to be used for setting lock bits in shared-data structures or for adding entries to shared (multiple producer, single consumer) linked lists. The compare&swap operation is expected to be used for deleting entries from linked lists. The fetch&add operation is expected to be used for dispatching tasks and performing barrier synchronizations when the tasks complete.

1.6.4 test&lock data-access operation

The mask&swap, compare&swap, and fetch&add data-access operations discussed in 1.6.3 cannot be easily supported by existing processor instruction sets and bus standards. Some microprocessors support restricted subsets of mask&swap instructions and others support only compare&swap. For these reasons, a minimal data-access lock capability, called test&lock, is provided for use within the subsistence set (as specified within 7.1.1).

The test&lock operation is performed on an aligned quadlet, sets a lock condition, and returns a Boolean locked/unlocked condition. See B.1 of annex B for specifications of the test&lock implementation options. The test&lock assumes that a shared semaphore has a locked or unlocked data value. Two locked value ranges are allowed: the addressed byte is zero (locked0) or the most significant bit of the addressed byte is one (locked1). Only one unlocked value is allowed, as illustrated in figure 5. Within this figure, the ‘X’ character labels a bit that may have a 0 or 1 value.

0	00000000	0	1XXXXXXXX	0	01000000
1	XXXXXXXX	1	XXXXXXXX	1	00000000
2	XXXXXXXX	2	XXXXXXXX	2	00000000
3	XXXXXXXX	3	XXXXXXXX	3	00000000
locked0 values		locked1 values		unlocked value	

Figure 5— test&lock data values

With these locked and unlocked value constraints, several different lock instructions can be used to support test&lock operations on atomic uncached data, as illustrated in table 3.

Table 3— Instructions that support test&lock

Lock instruction	Data size (in bytes)	Update operation
bitSet1	1	data = 0x80;
fullSet1	1	
fullSet4	4	data = ~0;
fullClear1	1	
fullClear4	4	data = 0;
compareSwap1	1	
compareSwap4	4	data = (data == test) ? next : data;

1.6.5 Coherent data-access capabilities

Special transactions are not needed to support coherent data-access operations. Coherent accesses are expected to be performed by fetching or converting a cache line into the proper state, executing one or several instruction on the affected data, and releasing the cache line for other uses. To ensure forward progress, processors are expected to temporarily inhibit cache interrogations while the cache-line accesses are performed.

If normal ALU instructions are used to implement coherent data-access operations, a wide variety of access operations are theoretically possible. However, a primary design objective is to define the same data-access operations for uncached and coherent addresses. Thus, the data-access operations are limited to the set provided for uncached data-access operations.

1.6.6 Uncached transaction support

The split-transaction protocols used by high-speed buses restrict the ways that data can be accessed. Special operations, such as indivisible read-modify-write sequences (that lock the bus between reads and writes) are impractical. Therefore, a limited transaction set (that is defined by the CSR Architecture and supported on the SCI standard) is assumed.

The uncached transactions assumed by this standard include read, write, and lock operations. These transactions involve an addressed *data* value and (for reads and locks) return the *last* data value. The requester-provided *next* and *test* values are used in the update operation, as illustrated in figure 6.

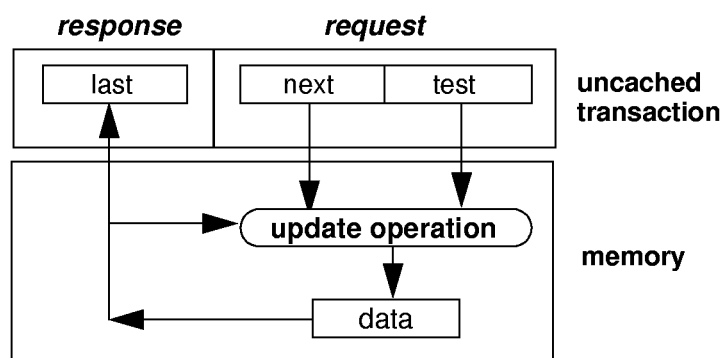


Figure 6—Uncached transaction design model

The available set of bus transactions thus limits the data-access operations that can be supported. The bus transactions of table 4 are assumed by this standard. In all cases, the formats of the transaction parameters (*last*, *next*, and *test*) and the addressed data format shall be the same. See 8.2 for an exact specification of the behavior of these transactions.

Different names are used to distinguish the data-access operations (which are supported by higher-level languages and compilers) from the bus transactions (maskSwap, compareSwap, bigAdd, and littleAdd) that are generated by an atomic access of uncached data.

These transactions are defined by the CSR Architecture and supported by the SCI standard. Most of them involve only transfers or bit-equality tests and are not affected by the bit/byte ordering of the addressed data. The bigAdd and littleAdd transactions are different, in that the direction of byte-carry propagation depends on the byte significance of the data. Thus, two versions of the uncached add transaction are assumed.

1.7 Byte-ordering models

This standard expects data objects to be byte-addressable. Since many data objects have multiple bytes, one needs to specify whether the data byte with the lowest byte address is the least or most significant byte of the data. Two types of data formats are commonly supported: big-endian and little-endian.

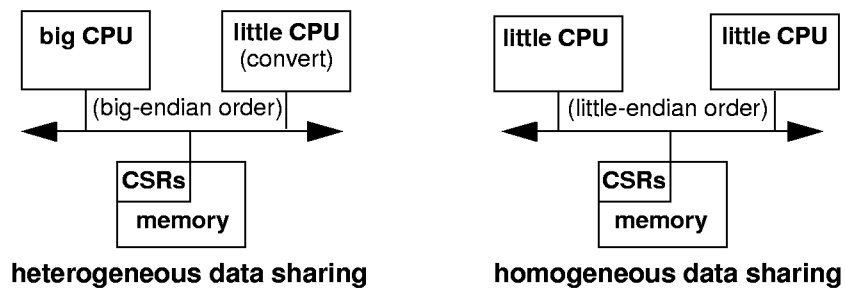
Rather than attempting to resolve which endian order is best (a technically admirable but unrealistic goal), both are defined by this standard. Other data-interchange standards, whose scope includes specifying the formats of shared-data structures, have the option of selecting the data formats that are most expedient for their expected operating environment.

However, within heterogeneous multiprocessor systems, shared-data structures are expected to be based on big-endian data formats, as illustrated by the left half of figure 7. This expectation is reflected in the data formats included within the *functional set*; see 7.1.2 for details.

Selecting one endian ordering simplifies the conversion costs (since some of the processor never have to convert) as well as the data structures (a dynamic byte-order flag bit is not required). The big-endian ordering is consistent with established network header formats, the control registers defined by the CSR Architecture, the transaction set supported by SCI memories (bigAdd is mandatory, littleAdd is optional) and the notation used throughout the SCI standard.

Table 4— Uncached bus transactions

Transaction	Data size (in bytes)	Update operation
read 1	1	last = data;
read2	2	
read4	4	
read8	8	
read16	16	
read64	64	
write1	1	data = next;
write2	2	
write4	4	
write8	8	
write16	16	
write64	64	
maskSwap4	4	last = data; data = (next & test) (data & ~test);
maskSwap8	8	
compareSwap4	4	last = data; if (data == test) data = next;
compareSwap8	8	
bigAdd4	4	last = data; data = AddHiToLo(data,next);
bigAdd8	8	
littleAdd4	4	last = data; data = AddLoToHi(data,next);
littleAdd8	8	

**Figure 7— Data-sharing models**

However, when exchanging data among themselves, little-endian processors are expected to minimize their conversion costs by using the little-endian data formats, as illustrated in the right half of figure 7.

2. References, glossary, and notation

2.1 References

This standard shall be used in conjunction with the following publications. When they are superseded by an approved revision, the revision shall apply:

ANSI/ISO 9899-1990, Programming Language—C.⁴

IEEE Std 754-1985 (Reaff 1991), IEEE Standard for Binary Floating-Point Arithmetic (ANSI).⁵

2.2 Conformance levels

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

2.2.1 expected: A keyword used to describe the behavior of the hardware or software in the design models assumed by this standard. Other hardware and software design models may also be implemented.

2.2.2 may: A keyword that indicates flexibility of choice with no implied preference.

2.2.3 shall: A keyword indicating a mandatory requirement. Designers are required to implement all such mandatory requirements to ensure interoperability with other products conforming to this standard.

2.2.4 should: A keyword indicating flexibility of choice with a strongly preferred alternative. Equivalent to the phrase “is recommended.”

For SCI processors and memory, the conformance levels with this standard are dependent on which set of data formats is supported, as defined below and specified in clause 7.

2.2.5 complete set: Should be supported by second-generation SCI processors. All data formats (little-endian as well as big-endian) and data-access capabilities are supported.

2.2.6 functional set: Shall be supported by second-generation SCI processors. All big-endian data formats and data-access capabilities are supported.

2.2.7 subsistence set: May be supported by the first SCI processors, since compiler changes are sufficient to support this set on existing hardware architectures.

An exact data-format set is not defined for other unit architectures, since special-purpose memories, I/O devices, and control registers are expected to support only a small set of application-specific data formats. Therefore, another conformance level is defined; see 2.2.8.

2.2.8 subset conformance: Those data formats or data-access operations that are supported conform with the notation, format, and value specifications contained within this standard.

2.3 Glossary of terms

A large number of bus and interconnect-related technical terms are used in this standard. These terms are defined below:

⁴Replaces ANSI X3.159-1989. ANSI/ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse. ANSI/ISO 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.

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

WITHDRAWN

2.3.1 aligned: A term that refers to the constraints placed on the address of the data; the address is constrained to be a multiple of the data format size.

2.3.2 atomic: A term that describes the constraints placed on accesses of a data format; the read, write, and lock accesses are performed indivisibly.

2.3.3 bigAdd: A bus transaction that adds an integer *addend* argument to a specified data address and returns the previous data value from that address. All values in this transaction are assumed to be big-endian integers. In the CSR Architecture this is called a *fetch_add* transaction.

2.3.4 byte: A set of 8 bits of data, used as a synonym for octet.

2.3.5 cache line: Often called simply a “line.” The 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.

2.3.6 coherent data-access operation: A data-access operation, when used to access coherently cached data.

2.3.7 compare&swap: A data-access operation that conditionally stores a *next* value to a specified data type and returns the previous data value. The store occurs when the addressed memory value and a second *test* value are equal. When accessing uncached data, this data-access operation generates a *compareSwap* bus transaction.

2.3.8 compareSwap: A bus transaction that conditionally stores a *next* argument to a specified data address and returns the previous data value from that address. The store occurs when the addressed memory value and a second *test* value are equal. In the CSR Architecture, this is called a *compare_swap* transaction.

2.3.9 container: An ordered set of 1, 2, 4, or 8 contiguous bytes fully packed with one or more signed or unsigned field formats.

2.3.10 CSR Architecture: Refers to IEEE Std 1212-1991 [B4].

2.3.11 data-access operation: A processor-initiated load, store, or lock that involves a data-format copy and (for lock operations) a data-update action (such as swap or add).

2.3.12 doublet: A data format or data type that is 2 bytes in size.

2.3.13 Futurebus+: Refers to IEEE Std 896.1-1991 [B1] and IEEE Std 896.2-1991 [B2]. Futurebus+ supports cache-coherent multiprocessing with physical buses on the backplane. SCI may be used to interconnect Futurebus+ systems, since they share the same line size and CSR Architecture.

2.3.14 hexlet: A 16-byte data format or data type. The name hexadecllet would more accurately describe these 16-byte formats, but for notational convenience this abbreviated term is used throughout this standard.

2.3.15 indivisible access: A data access for which the entire datum is read or written as a whole, with no possibility of partial interleaved access by another processor.

2.3.16 line: See: **cache line**.

2.3.17 linelet: A 64-byte collection of other aligned data formats and/or field containers.

2.3.18 littleAdd: A bus transaction that adds an integer *next* argument to a specified data address and returns the previous data value from that address. All values in this transaction are assumed to be little-endian integers. In the CSR Architecture this is called a *little_add* transaction.

2.3.19 load&add: A data-access operation that adds a *next* value to a specified data type and returns the previous data value.

2.3.20 mask&swap: A data-access operation that stores a *next* value to the *test* specified bits within a specified data type and returns the previous data value.

2.3.21 maskSwap: A bus transaction that stores bits of a *next* argument to a specified data address and returns the previous data value from that address. The affected bits are specified by a *test* argument. In the CSR Architecture this is called a *mask_swap* transaction.

WITHDRAWN

2.3.22 node: The entity associated with a particular set of control register addresses (including identification ROM and reset command registers) that is initially defined in a 4 Kbyte (minimum) initial node address space. 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).

2.3.23 octlet: An 8-byte data format or data type. Not to be confused with an octet, which has been commonly used to describe 8 bits of data. In this document, the term byte, rather than octet, is used to describe 8 bits of data.

2.3.24 permissive: A term that describes the constraints placed on accesses of a data format; the read, write, and lock accesses can be performed divisibly (i.e., using multiple transactions) using a multiphase fetch and/or update process.

2.3.25 quadlet: A 4-byte data format or data type.

2.3.26 SCI: See: **SCI standard**.

2.3.27 SCI standard: Refers to IEEE Std 1596-1992 [B6], which provides computer-bus-like services using a collection of point-to-point unidirectional links.

2.3.28 Serial Bus: Refers to the IEEE P1394 project, which defines an inexpensive serial network that can be used as an alternate control or diagnostic path, as an I/O connection, or in place of a parallel bus in some systems.

2.3.29 shared-data formats: Refers to this standard.

2.3.30 transaction: An information exchange between two nodes. A transaction consists of a request subaction and a response subaction. The request subaction transfers commands (and possibly data) from a requester to a responder. The response subaction returns status (and possibly data) from the responder to the requester.

2.3.31 unaligned: A term that refers to the constraints placed on the address of the data; the address is unconstrained and may be any integer value.

2.3.32 uncached data-access operation: A data-access operation, when used to access data that is not cached.

2.3.33 VMEbus: Refers to IEEE Std 1014-1987 [B3], which defines a 32-bit backplane bus.

2.4 Bit and byte ordering

2.4.1 Endian-order options

Two types of data formats are commonly supported within the computing industry: big-endian and little-endian. Both of these orderings are therefore defined by this standard. For consistency, a common template is used to illustrate both endian orderings, but the ordering of bits and bytes within formats is different.

When illustrating any format, the most significant and least significant components are always placed on the left and right sides of the figure, respectively. Within multiple-byte data objects, the addresses of the data-byte components are illustrated above the data object; these data-byte addresses are specified as offsets from the base address of the data object, as illustrated in figure 8 and figure 9.

2.4.2 Big-endian notation

When describing big-endian data objects, bit[0] is always the most significant bit of the data and byte[0] is always the most significant byte of the data, as illustrated in figure 8. The use of this consistent notation avoids confusion when specifying the addresses of bit fields, which sometimes cross byte-address boundaries.

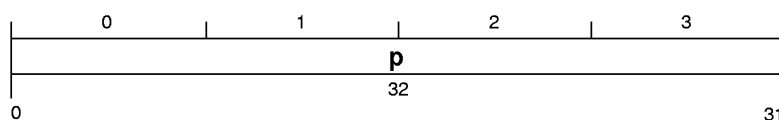


Figure 8— Big-endian notation (BigUnsignedQuadlet example)

WITHDRAWN

In these illustrations, the address of each byte (0, 1, 2, 3, as measured from the format's base address) is illustrated on the top. The width of data fields (32) is illustrated immediately below the data format. The bit labels for the most and least significant bits (0 and 31) are placed below these bit locations.

2.4.3 Little-endian notation

When describing little-endian data objects, bit[0] is always the least significant bit of the data and byte[0] is always the least significant byte of the data, as illustrated in figure 9. The use of this consistent notation avoids confusion when specifying the addresses of bit fields, which sometimes cross byte-address boundaries.

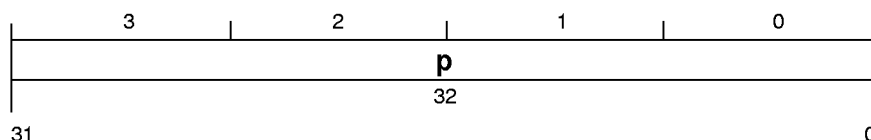


Figure 9— Little-endian notation (LittleUnsignedQuadlet example)

The bit labels for the least significant and most significant bits (0 and 31) are placed below these bit locations.

2.5 Integer format notation

Signed integers are represented as two's complement binary numbers and have two components: s and p . The sign bit s is the most significant bit and the remaining p field represents a positive (unsigned) integer, as illustrated in figure 10.

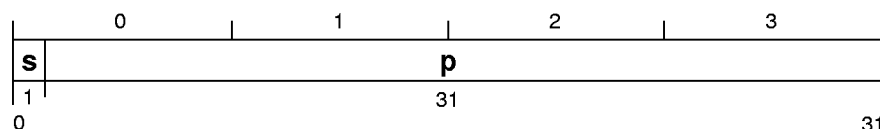


Figure 10— Signed Integer field notation (BigSignedQuadlet example)

Within this standard, illustrations of field data formats assume one or more fields are packed within a containing object, as illustrated in figure 11. However, the number of fields within a containing object may be as small as one, or may be as large as the number of bits (for this BigQuadletSignedField example, 32 fields could be supported).

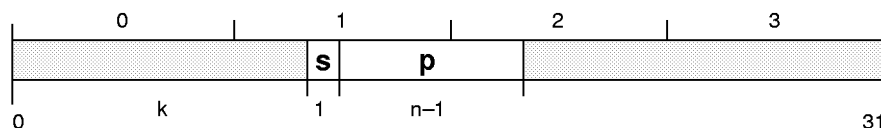


Figure 11— SignedField notation (BigQuadletSignedField example)

WITHDRAWN

Fields shall be one or more bits wide, fields within a container shall be non-overlapping, and fields shall fully populate their container. A field is parameterized by the name and address of its container along with its bit-position and width. The bit position shall be measured in bits from the most or least significant end of the data, for big-endian and little-endian formats, respectively.

Fields may be either signed or unsigned, and a container may be 1, 2, 4, or 8 bytes.

2.6 Real format notation

The floating-point number representations (which are called *reals* within this standard) are specified by IEEE Std 754-1985, which defines three components: *s*, *e*, and *f*. The *s* bit specifies the number's sign (positive or negative); the *e* and *f* fields specify the number's magnitude. The sign bit *s* is the most significant bit, the exponent field *e* contains the more significant bits, and the fraction field *f* contains the least significant bits, as illustrated in figure 12.

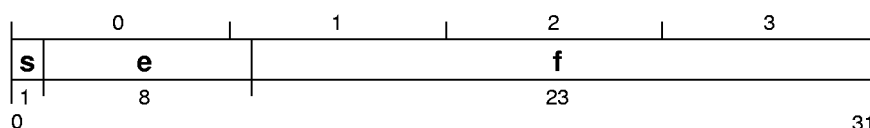


Figure 12— Floating point field notation (BigRealQuadlet example)

For NaN data values, the most significant bit of the fraction shall be used to differentiate between signaling not-a-number (sNaN) and quiet not-a-number (qNaN) data values, for which the bit is 0 or 1, respectively. This convention simplifies the conversion between signaling and quiet NaN values, which involves setting the most significant bit of *f* to 1.

2.7 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 usual 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 “1A₁₆” or “11010₂”.

2.8 Glossary of data formats

A large number of data formats are defined within this document. The data-format names contain one or more components. Each component describes a property of the data format and begins with a capital letter. These data-format names are listed in table 5 through table 9, along with their name components. The tables also list the number of bytes (although this is implied by the name's object component) and the subclause where the data format is defined.

The terms *aligned* and *unaligned* refer to the constraints placed on the address of the data. The term *aligned* describes formats for which the address is constrained to be a multiple of the size. The term *unaligned* describes formats without address restrictions (the address may be any integer).

Processors are expected to check for consistency between the address and the data format. When accessing an aligned data type, an unaligned address is expected to generate an illegal-access trap. When accessing an unaligned data type, the success of the access shall not be affected by the address. However, the performance of accessing unaligned data may be reduced if the address is not aligned to the size of the data, since this may require extra memory transactions or may cause traps to special handling software. Data alignment also affects the indivisibility of accesses; the loads and stores of unaligned data are not necessarily indivisible.

WITHDRAWN

The terms *big* and *little* refer to the ordering of bit labeling as well as the ordering of bytes within multibyte formats. The term *big* is used when the significance of bytes decreases as the byte address increases. The term *little* is used when the significance of bytes increases as the address increases.

The terms *signed*, *unsigned*, and *real* refer to the values represented by data formats. The term *signed* is used to describe formats that specify positive and negative integer values. The term *unsigned* is used to describe formats that specify positive integer values. The term *field* is used to describe contiguous bits located within larger containers. The term *real* is used to describe floating-point formats.

The terms *doublet*, *quadlet*, *octlet*, *hexlet*, and *linelet* refer to the size of the data-format objects. The term *doublet* describes a 2-byte format. The term *quadlet* describes a 4-byte format. The term *octlet* describes an 8-byte format. The term *hexlet* describes a 16-byte format. The term *linelet* describes a collection of 64-byte formats (the cache-line size on SCI).

2.9 Glossary of data-format containers

Signed and unsigned integers may be fields within larger containers. Each container has a name and each container name has one or more components. The components describe the alignment, endian order, and byte-size properties of the containers, as listed in table 10.

Table 5— Aligned big-endian formats

Format name	Address	Order	Type	Object	Bytes	Subclause
BigSignedByte	aligned	big	signed	byte	1	3.1.1
AlignedBigSignedDoublet				doublet	2	3.1.2
AlignedBigSignedQuadlet				quadlet	4	3.1.3
AlignedBigSignedOctlet				octlet	8	3.1.4
AlignedBigSignedHexlet				hexlet	16	3.1.5
BigUnsignedByte			unsigned	byte	1	3.2.1
AlignedBigUnsignedDoublet				doublet	2	3.2.2
AlignedBigUnsignedQuadlet				quadlet	4	3.2.3
AlignedBigUnsignedOctlet				octlet	8	3.2.4
AlignedBigUnsignedHexlet				hexlet	16	3.2.5
BigByteSignedField			signed	bit field	1	3.3.1
AlignedBigDoubletSignedField					2	3.3.2
AlignedBigQuadletSignedField					4	3.3.3
AlignedBigOctletSignedField					8	3.3.4
BigByteUnsignedField			unsigned	bit field	1	3.4.1
AlignedBigDoubletUnsignedField					2	3.4.2
AlignedBigQuadletUnsignedField					4	3.4.3
AlignedBigOctletUnsignedField					8	3.4.4
AlignedBigRealQuadlet			real	quadlet	4	3.5.1
AlignedBigRealOctlet				octlet	8	3.5.2
AlignedBigRealHexlet				hexlet	16	3.5.3

WITHDRAWN

2.10 Data-format collections

Several more-concise terms are used to describe collections of one or more data formats. These terms are derived from the names of the data formats that they describe. The name of the collection is generated by retaining the terms that are shared by the specified collection of data formats. These data-format names are listed in table 11, along with the collection of data-format names they describe.

Table 6— Aligned little-endian formats

Format name	Address	Order	Type	Object	Bytes	Subclause
LittleSignedByte	aligned	big	signed	byte	1	4.1.1
AlignedLittleSignedDoublet				doublet	2	4.1.2
AlignedLittleSignedQuadlet				quadlet	4	4.1.3
AlignedLittleSignedOctlet				octlet	8	4.1.4
AlignedLittleSignedHexlet				hexlet	16	4.1.5
LittleUnsignedByte			unsigned	byte	1	4.2.1
AlignedLittleUnsignedDoublet				doublet	2	4.2.2
AlignedLittleUnsignedQuadlet				quadlet	4	4.2.3
AlignedLittleUnsignedOctlet				octlet	8	4.2.4
AlignedLittleUnsignedHexlet				hexlet	16	4.2.5
LittleByteSignedField			signed	bit field	1	4.3.1
AlignedLittleDoubletSignedField					2	4.3.2
AlignedLittleQuadletSignedField					4	4.3.3
AlignedLittleOctletSignedField					8	4.3.4
LittleByteUnsignedField			unsigned	bit field	1	4.4.1
AlignedLittleDoubletUnsignedField					2	4.4.2
AlignedLittleQuadletUnsignedField					4	4.4.3
AlignedLittleOctletUnsignedField					8	4.4.4
AlignedLittleRealQuadlet			real	quadlet	4	4.5.1
AlignedLittleRealOctlet				octlet	8	4.5.2
AlignedLittleRealHexlet				hexlet	16	4.5.3