



IEEE Standard for Scalable Storage Interface (S²I)

IEEE Computer Society

Sponsored by the
Microprocessor and Microcomputer Standards Committee

1285TM

IEEE
3 Park Avenue
New York, NY 10016-5997, USA
22 March 2006

IEEE Std 1285TM-2005

Recognized as an American National (ANSI)

IEEE Standard for Scalable Storage Interface (S²I)

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

Approved 29 December 2005
Reaffirmed 28 June 2011

American National Standards Institute

Approved 22 September 2005
Reaffirmed 8 December 2010

IEEE-SA Standards Board

Abstract: A scalable interface between mass-storage devices and controlling hardware/software is specified in this standard. The interface is optimized for low-latency interconnects, and assumes that the processor/controller and the storage device can often be co-located on the same printed-circuit board. The interface can also be used with longer-distance bus-like interconnects, including (but not limited to) IEEE Std 1394^a-1995 Serial Bus and IEEE Std 1596^a-1992 Scalable Coherent Interface.

Keywords: controller, mass-storage, storage interface

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

Copyright © 2006 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 22 March 2006. 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.

Print: ISBN 0-7381-4798-2 SH95371
PDF: ISBN 0-7381-4799-0 SS95371

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

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied “**AS IS.**”

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, or every ten years for stabilization. When a document is more than five years old and has not been reaffirmed, or more than ten years old and has not been stabilized, 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.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon his or her independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

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 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 societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal interpretation of the IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

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. Recommendations to change the status of a stabilized standard should include a rationale as to why a revision or withdrawal is required. Comments and recommendations on standards, and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854-4141
USA

Authorization to photocopy portions of any individual standard for internal or personal use is granted by The Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

This introduction is not a part of IEEE Std 1285-2005, IEEE Standard for Scalable Storage Interface (S²I).

Technology is now enabling the miniaturization of non-volatile storage units such as magnetic disk drives that can now be directly mounted on printed circuit boards. Existing cable-type storage interfaces, developed to span long distances, are not cost-effective for these small packages. The Scalable Storage Interface (S²I) is intended to provide a simpler interface for “in the box” applications.

Recognizing the need for leverage across product families, the S²I interface maintains a generalized DMA transfer model, while leveraging previous related standards (IEEE Std 1212.1™-1993). Key concepts from IEEE Std 1212.1 (such as mailbox-based status reports) are leveraged, while attempting to simplify the command-entry delivery process based on experiences from IEEE 1212-related bus standards.

Although intended to support storage-device interfaces, the S²I interface defines a scalable DMA architecture that is applicable to a wide range of input/output (I/O) devices and network interfaces. Implementing a largely physical-layer independent interface allows the S²I definition to be used with a wide range of interconnect technologies. Implementing a largely device-independent interface allows new (and surprisingly different) storage devices to be supported in the future. There is also the possibility of simplifying system software since one architecture can be leveraged across different classes of I/O devices.

As the work on this standard progressed, Martin Freeman stressed the importance of using a RAM-like interface to leverage existing industry-standard buses. Chris Hamlin emphasized the need to provide a simple disk-attach interface. Andy Hospodor helped us better understand disk-drive requirements. Richard Wilmont helped define the power-management structure. Mike Wenzel provided a detailed review of shared list-handoff variables, to ensure correctness in the presence of read/write timing hazards.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/updates/errata/index.html>. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/interp/index.html>.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Participants

The following is a list of participants in the IEEE Project 1285 Working Group. Voting members at the time of publication are marked with an asterisk (*).

Martin Freeman, *Chair**

Andy Hospodor, *Vice-chair**

David V. James, *Editor**

Duane Anderson*
Haakon Bryhni
Bob Davis*
Stein Gjessing
David Gustavson*
Chris Hamlin

Anatol Kaganovich*
Sam Karunanithi
Beomsup Kim
Dennis Pak
Ron Roberts
R. J. Safranek
Wink Saville

Tim Scott
Vivian Shen*
Fred Shirley
Joanne Spiller
Russ Tuck
Richard Wilmot*

The following members of the individual balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

James Carlo
Keith Chow
John Cole
Bob Davis
Guru Dutt Dhingra
Sourav Dutta
Martin Freeman
Ron Greenthaler

David Gustavson
Andy Hospodor
David James
Gregory Luri
Joseph Marshall
Narayanan Murugesan
Matthew Noury
Klaus Rapf

Gary Robinson
Michael Scholles
Stephen Schwarm
Veselin Skendzic
Robert Snively
Thomas Starai
Srinivasa Vemuru
Oren Yuen
Janusz Zalewski

When the IEEE-SA Standards Board approved this standard on 22 September 2005, it had the following membership:

Steve M. Mills, *Chair*

Richard H. Hulett, *Vice Chair*

Don Wright, *Past Chair*

Judith Gorman, *Secretary*

Mark D. Bowman
Dennis B. Brophy
Joseph Bruder
Richard Cox
Bob Davis
Julian Forster*
Joanna N. Guenin
Mark S. Halpin
Raymond Hapeman

William B. Hopf
Lowell G. Johnson
Herman Koch
Joseph L. Koepfinger*
David J. Law
Daleep C. Mohla
Paul Nikolic

T. W. Olsen
Glenn Parsons
Ronald C. Petersen
Gary S. Robinson
Frank Stone
Malcolm V. Thaden
Richard L. Townsend
Joe D. Watson
Howard L. Wolfman

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Satish K. Aggarwal, *NRC Representative*
Richard DeBlasio, *DOE Representative*
Alan H. Cookson, *NIST Representative*

Michelle Turner
IEEE Standards Project Editor

Contents

1. Overview.....	1
1.1 Scope	1
1.2 Purpose	1
1.3 Background.....	1
1.4 Scalable storage interface properties	2
1.5 Historical perspective	2
1.6 Non-storage applications	3
1.7 S ² I command execution.....	4
1.8 S ² I topologies	6
1.9 S ² I interconnects.....	9
1.10 Memory-mapped addresses	9
1.11 Shared memory-mapped registers	10
2. Normative references.....	11
3. Glossary and notation	12
3.1 Definitions	12
3.2 Field names.....	15
3.3 C-code notation	15
3.4 Bit and byte ordering.....	16
3.5 Generic delta-level command lists	18
3.6 Data formats	18
4. Abbreviations and acronyms	20
5. Alpha-level interface	21
5.1 Alpha-level overview	21
5.2 Command execution.....	22
5.3 Alpha4 command and status entries	23
5.4 Alpha8 command and status entries	25
5.5 Command processing	26
5.6 Alpha-level CSRs	27
6. Delta interface properties.....	30
6.1 Delta-level distinctions.....	30
6.2 Delta-level memory-mapped addresses.....	30
6.3 Address formats.....	32
6.4 Command lists	34
6.5 Legacy command-set support.....	37
6.6 Delta-level command processing	37
6.7 Command sequence ordering	41
6.8 Input/output transfers.....	42
6.9 Scatter/gather transfers	43
6.10 System-bus transactions	44
6.11 Status reports	46
6.12 Command synchronization.....	48
6.13 Error conditions	50
6.14 Delta-level mover CSRs	51

6.15 Operational mover states	54
7. Delta interfaces	56
7.1 Delta16 overview.....	56
7.2 Delta32 overview.....	67
7.3 Delta32 Mover CSRs.....	75
7.4 Delta64 overview.....	76
Annex A (informative) Bibliography	85
Annex B (informative) Illustrative applications	86
B.1 Secure logins	86
B.2 Scattered buffer allocation.....	87
B.3 Heartbeat timers.....	88
B.4 Event reports.....	88
B.5 Power-level management	88
B.6 Timed command execution	89
B.7 DiskLite architecture	90
Annex C (informative) Software based RAID.....	93
C.1 RAID I/O driver software.....	93
C.2 Single-block OUTPUT.....	93
C.3 Striped-block OUTPUT.....	95
C.4 Erasure-correcting single-block INPUT.....	96
C.5 Erasure-correcting multiple-block INPUT	97
Annex D (informative) Physical interface possibilities	98
D.1 Delta16-level backplane interface.....	98
D.2 Delta64 Serial Bus interface	99
Annex E (informative) C-code illustrations.....	103

IEEE Standard for Scalable Storage Interface (S²I)

1. Overview

This is an informative introductory clause. Normative specifications are contained within following clauses and normative annexes.

1.1 Scope

This standard defines a scalable interface for use with memory-mapped storage units and other devices. The term “storage unit” can encompass rotating, non-rotating, volatile, and non-volatile storage. Issues of concurrency, latency, bandwidth, extensibility, and negotiation will be addressed. The interface is intended for use with either a single storage unit or with many coordinated storage units.

1.2 Purpose

This standard provides an efficient physical-layer independent interface architecture for storage units that attach directly to memory access buses. In this interface model, storage units have an attachment to main memory, rather than an attachment to an input/output (I/O) channel. This simplifies the storage unit design and supports scheduling of data transfers spanning large numbers of units.

1.3 Background

The S²I interface is intended to be located within the processor’s memory-mapped address space, in order to support a wide range of interconnect technologies. The memory-mapped address space is the only interface universal to all computers based on Von-Neumann processors. All microprocessors, whether RISC or CISC, are stored program machines and can access either memory or memory-mapped I/O devices directly. The interface consists of registers that control data-transfer engines, called movers, which can transfer data from the I/O device to processor accessible memory.

The device-media addresses are not usually mapped directly into a memory-mapped address space, since the latencies associated with high-capacity disk or tape storage devices are much greater than the latencies of the resources normally mapped into the bus-address space. Dealing with these large latencies is not possible on buses with small fixed-timeout periods and, even when possible, would compromise the bus performance and error-detection capabilities.

For redundant array of inexpensive disks (RAID) applications, each storage unit could have local storage that is memory-mapped into system address space. Writes to that memory have EXOR effects on transaction writes. This memory facility and other command-synchronization facilities allows I/O devices to provide RAID storage facilities, without additional hardware support.

1.4 Scalable storage interface properties

The following items have been identified as properties of the delta-level S²I interfaces.

- a) *Scalable*. S²I is scalable. It is cost-effective for small systems and extensible to large (thousands of disks) systems. Low-cost or high-speed interconnect technologies can be used.
- b) *Bus independence*. Because the S²I interface is memory-mapped, standard memory interfaces can be used to support the logical protocols. The logical protocols can be readily supported on general-purpose buses (PCI Local Bus Specification [B8]¹), low-cost expansion buses (Serial Bus), or high-performance longer-distance interconnects (SCI).
- c) *High performance*.
 - 1) *Scattered addresses*. Indirect transfers involving lists of addresses efficiently support transfers:
 - i) An address/length array can specify scattered sets of system- or device-memory addresses.
 - ii) A list of commands can (somewhat less efficiently) specify scattered media addresses.
 - 2) *Command appending*. Additional command lists can be dispatched, while a current command list is active, without disrupting the in-progress transfers.
 - 3) *Synchronized transfers*. Device-to-memory or device-to-device transfers can be paced by using a LOOP command, whose completion is delayed until a test condition has been reached.
- d) *RAID friendly*. The target-paced data transfer capabilities allow the architecture to accommodate large numbers of storage devices.
- e) *Device independence*. The direct memory access (DMA) data-transfer architecture can be readily applied to a variety of storage and/or I/O device architectures.
- f) *Multiplexed connections*. There is no limit on the number of S²I interfaces that can attach to one bus (or bus-like interconnect). However, specific bus standards typically restrict the number of attached interfaces for speed or latency reasons.
- g) *Emulatable*. Control and Status Register (CSR) reads have no effects and writes need not have immediate side effects, allowing CSRs to be placed in a register file. The microprocessor's interpretation of recently-written CSRs can thus be done in a time insensitive fashion.

1.5 Historical perspective

Years ago on large mainframe systems, mass storage was provided by physically-large disk drives. Since these disk drives, as well as the CPU, were large, they were typically located in physically distinct racks. Bulky (and typically expensive) cables were initially used to connect the disk-drive and the CPU cabinets, as illustrated in the left side of Figure 1.1.

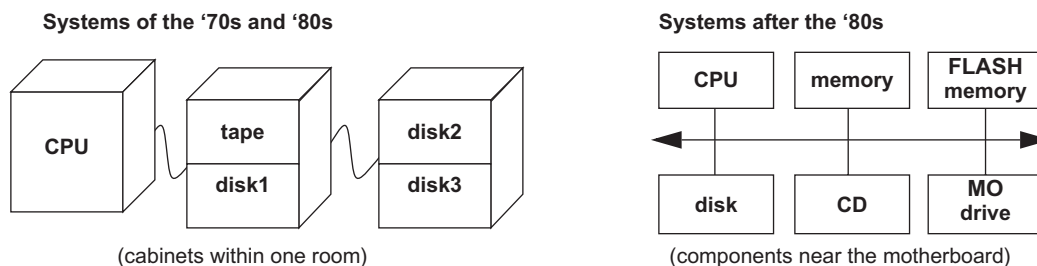


Figure 1.1—Evolving storage-device cabling strategies

Magnetic disks and tapes are no longer the only types of storage devices—optical disks (CD-ROM), non-volatile memories (FLASH), and magneto-optic drives (MO drives) now supplement them. Decreased

¹The numbers in brackets correspond to those of the bibliography in Annex A.

costs and improved densities of integrated circuits and disk storage now allow mainframe functionality to be provided by workstations and personal computers.

Within personal computer environments, the physical distance between the CPU and the storage devices has been reduced, and in most cases the storage devices are located adjacent to the CPU's motherboard. Thus, attaching storage devices to the processor/memory bus (or protocol-compatible, possibly switched, extensions of this bus) has become practical, allowing data to be directly transferred between storage devices and command-specified memory addresses. Thus, a distinct set of protocols for communicating through specialized storage-device connections is unnecessary and therefore is not assumed.

Given the wide range of bus and system topologies, several levels of storage-device interface were found to be appropriate, as listed below:

- Alpha4: Commands are placed in a device-resident array of 4-byte entries.
- Alpha8: Commands are placed in a device-resident array of 8-byte entries.
- Delta16: Commands are placed in memory-resident linked segments of 16-byte entries.
(Lists of Delta16 segments are illustrated in Figure 1.5.)
- Delta32: Commands are placed in memory-resident linked lists of 32-byte entries.
- Delta64: Commands are placed in memory-resident linked lists of 64-byte entries.
(Lists of Delta32/Delta64 entries are illustrated in Figure 1.6.)

Simple alpha-level interfaces have limited local-buffer DMA capability and are intended for embedded applications; delta-level interfaces allow commands and data to be located in system memory as well as in local memory. The Delta16 interface is optimized for uniprocessor applications, while the Delta32/Delta64 interfaces are optimized for multiprocessor systems. The *alpha* name is intended to imply simplicity; the *delta* name is intended to imply improvements beyond current designs.

These alpha-level and delta-level interfaces are logical-level interfaces describing device-resident registers and memory-resident data structures that are visible to software. All are intended to be applicable to a variety of communication technologies (called *transports*). Possible mappings of the delta-level interface to a computer's I/O bus (see D.1) as well as bus-like Serial Bus (see D.2) transports are illustrated in this document; mappings to other transports can be easily defined in the future.

1.6 Non-storage applications

Although optimized for storage devices, the S²I interface defines a general-purpose data-transfer architecture that allows the device to directly access memory. The data-transfer commands allow a wide range of storage devices to be easily supported; distinct transfer-unit architectures for each type of device architecture (disk, tape, FLASH memory, etc.) are not needed. The generality of the transfer-unit architecture allows it to be used with a wide variety of I/O devices, including FAX modems and network interfaces. However, such device-dependent architectures are beyond the scope of this standard.

To support multiple storage-device types, most data-transfer addresses and length counts are specified in terms of bytes, rather than media-dependent block sizes. This allows a uniform architecture to be shared by multiple storage devices, including variable-record-length devices. However, storage devices may use block addresses (as opposed to byte addresses) to increase the range of accessible media addresses.

1.7 S²I command execution

1.7.1 Shared command and status queues

Since the S²I device has the appearance of attaching to the system bus, I/O channel communication is unnecessary. The initiator (the processor) places a set of data-transfer commands in memory, so that the data transfers can be performed by the target (the storage device), as illustrated in Figure 1.2. After the commands are processed, the target updates memory-resident status reports to inform the initiator of data-transfer completion and exception conditions.

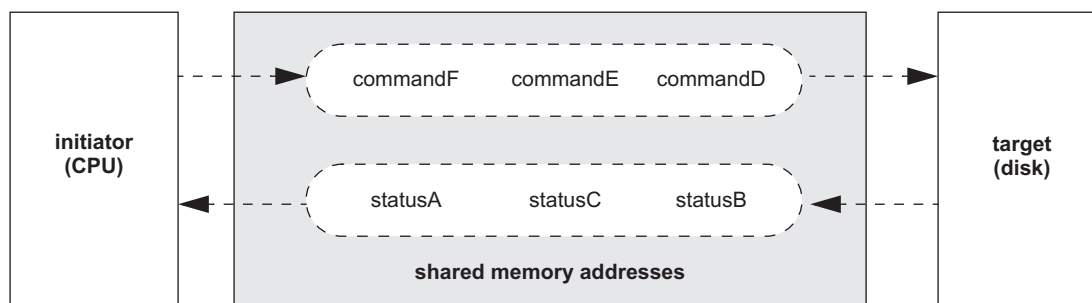


Figure 1.2—Command and status entry communications

Thus, data transfers from the target device involve the following steps.

- a) *Command creation.* The initiator creates a list of memory-resident command entries. Each command entry typically specifies the data-transfer direction (input or output) and provides media and system-memory addresses.
- b) *Target activation.* The initiator writes the first-command address to a memory-mapped location on the target, which activates processing of this and subsequent commands by the target.
- c) *Command processing.* Command-list entries are fetched and processed by the target, the command-specified data transfer is performed, and (if specified) an associated status entry is written.
- d) *Status processing.* The initiator (typically I/O driver software) processes the returned status-queue entries, releasing the affected data blocks, commands, and status entries. Completion status is then returned to the application.

The execution order of commands (and hence the ordering of status reports) is not necessarily the same as the command delivery order to the target.

From the perspective of the target, all system-memory addresses are physical and access errors are fatal. Fatal errors terminate further system-memory accesses by the I/O device; such errors typically generate a timeout and (via this timeout) cause the initiator to read the I/O device's status register, where an error indication is provided.

Although the I/O device appears to access system memory addresses, the system designer has the option of using address-mapping tables in the bridge (the hardware device that connects the system bus to the I/O bus), to restrict the target accesses to those pages actively involved in data transfers. Address-translation tables can also be provided to isolate the target from the system's internal addressing conventions.

Although all S²I interfaces behave in this fashion, the detailed structure of the command lists depends on which level of the S²I interface is implemented, as listed below:

- a) For the Alpha interfaces, the commands are placed in a small circular queue.
 - 1) The Alpha4 commands are 4 bytes in size.
 - 2) The Alpha8 commands are 8 bytes in size.
- b) For the Delta interfaces, command entries are linked, as follows:
 - 1) The 16-byte Delta16 commands are stacked into linked segments.
 - 2) The 32-byte Delta32 interface commands are placed into linearly linked lists.
 - 3) The 64-byte Delta64 interface commands are placed into linearly linked lists.

Data transfers involve processing of these shared command-list entries. An *initiator* is the function that generates command-list entries. In normal operation, the initiator dispatches commands by writing the appropriate location on the *target*. An active target normally copies the first and following command-list entries into local memory and invokes the command-entry-specified operation.

For the delta-level interfaces, the commands and status entries can be placed in the processor's local memory, reducing the frequency of processor accesses to the target's remotely located memory locations. After initialization, a single write transaction (that can be efficiently buffered within the processor and/or bus bridges) is sufficient to reactivate an idle device.

1.7.2 Command capabilities

In this standard, there are two basic types of data-transfer commands, listed below. For the alpha-level interface, memory addresses are restricted to be local-buffer addresses; for the delta-level interfaces, memory addresses may be system-memory or local-buffer addresses.

- INPUT: Data is transferred between a range of device and memory addresses.
- OUTPUT: Data is transferred between a range of memory and device addresses.

Several variants of these commands are supported, as listed below. For any specific INPUT or OUTPUT command, the preceding OUTPUT_PREP (if any) and following INPUT_POST (if any) are executed as an indivisible set.

- INPUT_FETCH: A speculative INPUT that terminates when another command becomes available.
- OUTPUT_PREP: A prepended OUTPUT command that establishes next-command context. This command immediately precedes an INPUT or OUTPUT data-transfer command.
- INPUT_POST: A postpended INPUT command that saves extended transfer-status context. This command immediately follows an INPUT or OUTPUT data-transfer command.

For the more capable delta-level interfaces, control commands also support synchronization and conditional execution, as listed below. Also, delta-level commands can be conditionally skipped, based on transfer-completion status.

- LOOP: Command execution is delayed based on command-specified device-register conditions.
- JUMP: Next command selection is based on command-specified device-register conditions.

Special device-dependent commands (such as format, rewind, or eject disk) are not provided. Rather than encoding these as special commands, writes to special device-dependent register locations can be used. This simplifies the data-transfer hardware (which only understands a few commands), while the large device-dependent address space provides the flexibility to define a (nearly) unlimited set of device-dependent capabilities. These device-dependent registers are beyond the scope of this standard.

1.7.3 S²I capabilities

To decouple the details of the initiator (typically the processor) from the target (the storage device), some higher-level capabilities are assumed to be located within the target device, as follows:

- a) *Logical address translation.* The target translates initiator-specified logical addresses (such as block addresses) to target-specific physical media address (such as cylinder/track/offset parameters).
- b) *Error correction.* The target is responsible for correcting recoverable errors, such as remapping defective block addresses to alternative media locations, in a functionally transparent manner.
- c) *Local buffering.* The target may optionally provide externally accessible data buffers, to support efficient RAID applications and device-to-device transfers.

Disk-drive vendors often prefer to use logical addresses and provide error correction capabilities, to isolate the customer from the drive's internals. This allows the disk vendors to quickly track technological advances without changing I/O driver software.

Data buffers are already located on most storage-device targets since they are needed to support a device's error-correction capabilities. These buffers decouple a device's data-streaming timings from the constraints of the attached interconnect (i.e., data transfers need not be performed at the exact head-transfer rate). Since error correction buffers are already assumed, the cost of local buffering is expected to be minimal.

1.8 S²I topologies

1.8.1 Alpha-level interface

The alpha-level interface is expected to be applied to embedded systems, where the initiator processor, memory, and target device are located on the same bus, as illustrated in Figure 1.3. The system is well protected from the target since all device-buffer to system-memory copies are performed by the processor.

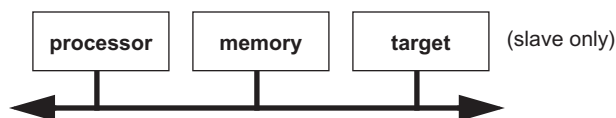


Figure 1.3—Expected alpha-level configurations

The alpha-level interface has low inherent cost, since no bus-master capabilities are assumed. Command and status entries are placed into a simple device-resident circular queue. Within this queue, 32-bit command entries specify the data-transfer operation being performed and provide a media-block address. All commands are assumed to invoke a fixed-size data transfer, between the media and one of the device-local buffers. Status is returned by overwriting the command location.

Command processing invokes device-to-target-buffer or target-memory-to-device-buffer copies. The target-local nature of these data buffers simplifies the target, but mandates additional data-memory-to-buffer copies. Depending on the system design, processor software or processor-managed DMA hardware could be responsible for these data-memory-to-buffer copies.

1.8.2 Delta-level interfaces

Delta-level interfaces are expected to be applied to 32/64-bit uniprocessor and multiprocessor systems. The interface is applicable to directly attached devices, as illustrated in the left portion of Figure 1.4.

If the target is located on a remote bus, an intermediate bridge provides speed-matching capabilities, as shown in the right portion of Figure 1.4. The bridge between the processor-bus and remote-bus may provide address-translation/protection, but these are transparent to the target. Thus, the delta-level target appears to directly access system-memory addresses.

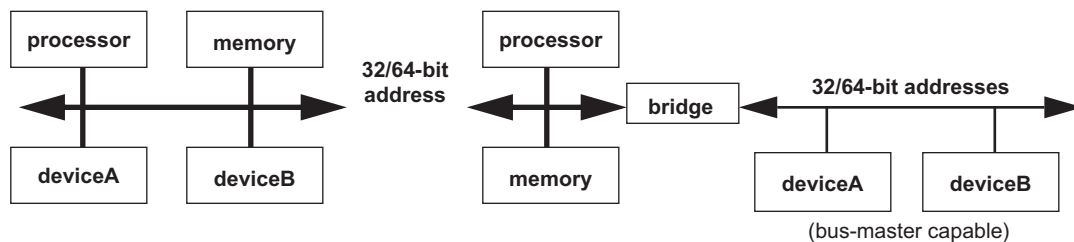


Figure 1.4—Expected delta-level configurations

The Delta16/Delta32 and Delta64 commands support 64-bit and 128-bit addresses respectively; on 32-bit systems, the more-significant portion of the address is expected to be ignored.

Storage-device interconnects are typically slower than processor/memory buses, and bridges have propagation delays. For these reasons, the memory-mapped processor-to-device accesses are expected to be much slower than the processor-to-memory accesses. To reduce the use of higher-latency processor-to-device accesses, delta-level command lists are normally placed in processor-local system memory.

Command entries are written by the initiator (typically a processor) and read by the target (typically the disk). For the Delta16 interface, commands are structured as linked segments and each segment contains one or more command entries, as illustrated in Figure 1.5. Intermediate command segments are terminated with a distinct JUMP command entry, which points to the next command entry. On the final command segment, the final command is used to suspend or terminate command-list execution.

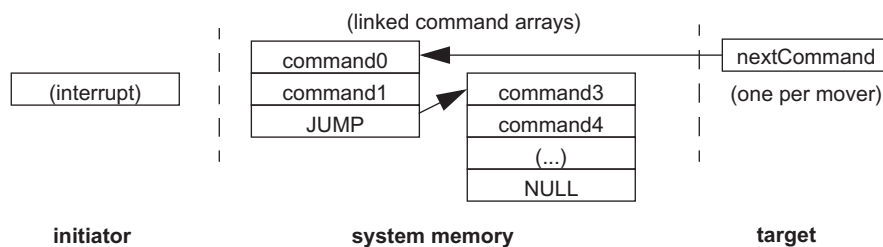


Figure 1.5—Delta16 command lists

For the Delta32/Delta64 interfaces, commands entries are linked into lists; each command entry provides a pointer to the next command entry, as illustrated in Figure 1.6.

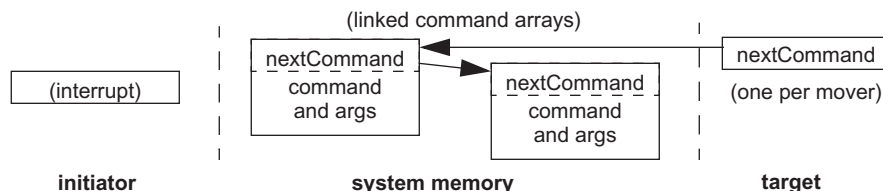


Figure 1.6—Delta32/Delta64 command lists

Simple targets are expected to process one command list; other targets are expected to process multiple command lists. Each command list has its own command-processing entity, called a *mover*. For example, a full-duplex channel interface would have two movers, one for input and one for output. Bus interface and DMA hardware can be time-shared by multiple movers, if the software interface model is maintained.

By software convention, two forms of Delta16/Delta32/Delta64 status reports are supported: mailbox and adjacent. With mailbox status reports, status is written to a command-entry-specified host-adaptor address, where the status information is saved and the processor is interrupted.

For adjacent status reports, the status is written to a fixed offset from the command entry and a distinct interrupt mechanism is used. This simple approach requires no host-adaptor hardware support, but complicates the processing of out-of-order status reports.

1.8.3 Multi-level RAID controllers

High-performance disk-array controllers may, for example, use the alpha-level interface as an internal connection to their attached targets, as illustrated in the left portion of Figure 1.7. Thus, the (potentially lower cost) alpha-level targets can be used within higher-performance disk arrays.

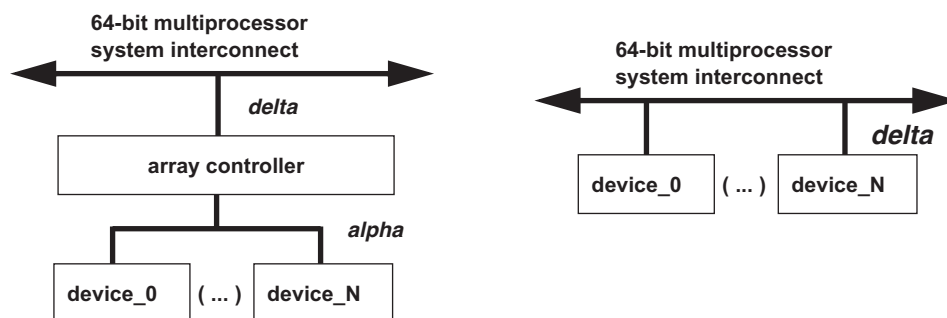


Figure 1.7—Expected delta-level multi-disk controller

Alternatively, the synchronization message capabilities of the delta-level interface allow disk arrays to be constructed from inexpensive disks, without an intermediate controller. In this case, I/O driver software generates multiple command lists (one command list for each device) providing synchronization points so these can be executed in a coordinated fashion. From a disk's perspective, only a small amount of additional buffer memory (with exclusive-OR capabilities) is required.

1.9 S²I interconnects

1.9.1 Interconnect capabilities

From the system's perspective, an S²I device behaves like a memory-mapped I/O device. The initiator broadcasts an address and the address is used to select the desired device as well as the device register within it. Data is either transferred to or from a range of device addresses, if write or read transactions are used respectively. These basic interconnect capabilities are illustrated in Figure 1.8.

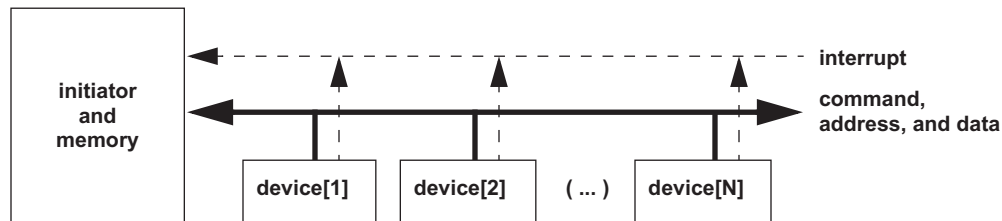


Figure 1.8—S²I device connectivity

The concepts of a multiplexed address/data bus and *interrupt* signaling are intended to illustrate the functional requirements of the interconnect, not its design. For example, buses implement interrupt indications as special backplane signals, memory-mapped transactions, or special transaction command codes.

On computer backplanes (PCI Local Bus Specification, [B8]), the interrupt capability may be implemented by special control signals. Alternatively, the CSR Architecture (see 3.1.2.12) defines interrupts to be 4-byte write transactions (to a previously-configured processor-interrupt address). S²I allows either approach to be used. The advantage of using a write transaction, rather than special interrupt signals, is that bridges can easily maintain the relative ordering of the 4-byte write transaction (the interrupt) and other concurrent data transfers, to ensure that interrupts do not pass previously posted data-transfer writes.

As examples, Serial Bus sends all data in a bit-serial fashion, but special packets (with unique command or address fields) transport interrupt information; some computer backplanes (PCI Local Bus Specification [B8]) provide each card with a distinct interrupt signal. Alternatively, any of the I/O buses and some of the synchronous DRAM interfaces developed in the 1990s may be used.

1.10 Memory-mapped addresses

From the perspective of I/O driver software on the processor, an S²I interface is a unit that is contained within a node, and is visible through a set of memory-mapped addresses. The memory-mapped addresses of a node contains ROM, registers, and device-buffer addresses, as illustrated in Figure 1.9.

Depending on the bus standard's addressing conventions, the memory-mapped addresses may be co-located within a range of contiguous node addresses (this is required on SCI and Serial Bus) or may be mapped to distinct software-configured portions of the address space (see D.1).

The ROM parameters are expected to specify the structure, function, and size of these memory-mapped resources. This information can be used to specify device parameters (such as sizes or speeds) or which optional features of the S²I interface have been implemented. The content and format of such information is bus-standard dependent and beyond the scope of this standard.

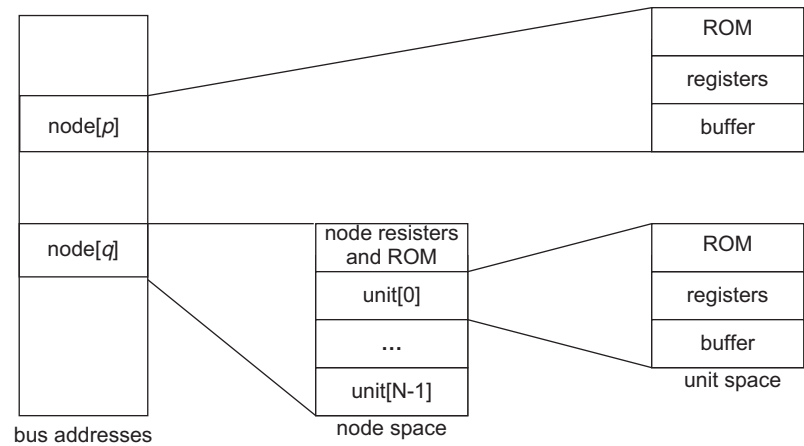


Figure 1.9—S²I’s memory-mapped addresses

Each node consists of one or more units, such as processor, memory, or S²I interfaces, which corresponds to unique I/O driver software. Thus, a single node could contain a processor unit as well as an S²I unit. Except for initialization and shared bus-interface tests (that can affect the entire node) the I/O driver software for each unit operates independently.

A single-unit node is illustrated by node[p] near the top of Figure 1.9. Within such single-unit nodes, distinct node resources are unnecessary and oftentimes only unit resources are provided.

A multiple-unit node is illustrated by node[q] near the bottom of Figure 1.9. Within such multiple-unit nodes, distinct node and unit resources are oftentimes provided. Depending on the standard, each unit’s ROM may be a subset of the node’s ROM, or may correspond to distinct unit-affiliated memory-mapped addresses.

Each unit is assumed to provide one or more movers. Each mover has the capability of transferring data to or from media addresses, without processor intervention. The structure and behavior of such movers is specified by this standard.

1.11 Shared memory-mapped registers

Some of the memory-mapped registers are shared and can be near simultaneously accessed by a bus-resident master and node-resident hardware. Access to these shared memory-mapped registers is assumed to be coordinated by access-control hardware, which fully completes an active access before initiating another potentially conflicting access.

This access-control hardware eliminates the possibility of reading metastable data or partially updated register contents. However, techniques for implementing such access-control hardware is beyond the scope of this standard.

2. Normative references

The following referenced document is indispensable for the application of this standard:

ANSI X3.159–1989, Programming Language—C.²

²ANSI publications are available from the American National Standards Institute, Sales Department, 11 West 42nd St., 13th Floor, New York, NY 10036, 212-642-4900.

3. Glossary and notation

3.1 Definitions

3.1.1 Conformance levels

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

3.1.1.1 expected: A keyword used to describe the behavior of the hardware or software in the design models assumed by this S²I standard. Other hardware and software design models may also be implemented.

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

3.1.1.3 shall: A keyword indicating a mandatory requirement. Designers are required to implement all such mandatory requirements to ensure interoperability with other S²I standard conforming products.

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

3.1.2 Definitions of S²I related terms

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

3.1.2.1 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.

3.1.2.2 alpha-level interface: A relatively simple (Alpha4 or Alpha8) device interface that supports transfers between device-buffer and device-media addresses (the processor must be used to copy data into system memory).

3.1.2.3 Alpha4: A device interface with 4-byte commands optimized for small embedded-system devices.

3.1.2.4 Alpha8: A device interface with 8-byte commands optimized for large embedded-system devices.

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

3.1.2.6 bus transaction: A data exchange between two nodes, consisting of a request subaction and a response subaction. The request subaction transfers a transaction code to a responder; the response subaction returns a completion-code to the requester. Depending on the transaction code, the data may be transferred in the request (for a write), in the response (for a read), or in both (for a compareSwap).

3.1.2.7 byte: An 8-bit entity. Within other standards, this is also called an octet.

3.1.2.8 command entry: An aligned location specifying operations to be performed by a device-local data-transfer engine.

3.1.2.9 command list: A singly-linked list of command entries. One or more initiators may append command entries to a command list. A single target is responsible for extracting command entries from a command list.

3.1.2.10 command set: A concatenated set of delta-level command-list entries, with leading OUTPUT_PREP and trailing INPUT_POST commands, that has the scheduling behavior of a single command.

3.1.2.11 compareSwap8 transaction: An indivisible bus transaction that conditionally stores an 8-byte *new* argument to a specified data address and returns the previous 8-byte data value from that address. The store occurs when the addressed memory value and a second 8-byte *arg* value are equal. In the CSR Architecture, this is called a compare_swap transaction.

3.1.2.12 CSR Architecture: ISO/IEC 13213, Control and Status Register Architecture.

NOTE— See ISO/IEC 13213: 1994 [B6].³

3.1.2.13 delta-level interface: A relatively sophisticated (Delta16, Delta32, or Delta64) interface that supports transfers between system-memory and device-media addresses, as specified by system-memory resident commands.

3.1.2.14 Delta16: A device interface with 16-byte commands optimized for uniprocessor systems.

3.1.2.15 Delta32: A device interface with 32-byte commands optimized for small multiprocessor systems.

3.1.2.16 Delta64: A device interface with 64-byte commands optimized for large multiprocessor systems.

3.1.2.17 device buffer: A set of dual-ported memory locations on an S²I device. These locations can be remotely accessed (by an attached processor or controller) or locally accessed by the local data-transfer-movement (i.e., mover) engines.

3.1.2.18 ignored; ign: A term used to describe the fields within unit-specific CSRs or command/status entries whose zero or last-written values shall be ignored.

3.1.2.19 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.

3.1.2.20 initiator: An entity that generates commands and delivers them to a target using the I/O process. The initiator may also receive status from the target upon command completion.

3.1.2.21 input operation: The data-transfer phase in the execution of a command entry that copies data from the device-specific space into bus-accessible space. A input operation consists of one or more write transfers.

3.1.2.22 I/O: An abbreviation for the terms *input and output*.

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

3.1.2.24 mover: An entity associated with a set of control registers. These control registers are typically used to initiate, manage, and monitor the transfer of data between an attached device and shared (bus-accessible) memory.

3.1.2.25 node: The entity associated with a particular set of (typically bus-dependent) control register addresses (including identification ROM and reset command registers). These registers are typically used to

³Notes in text, tables, and figures are given for information only, and do not contain requirements needed to implement the standard.

identify the node and to initialize the state of unit-dependent registers. 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).

3.1.2.26 output operation: The data-transfer phase in the execution of an OUTPUT command entry that copies data from the bus-accessible space into device-specific space. An output operation normally consists of one or more memory read transactions.

3.1.2.27 read transaction: A bus transaction that transfers data from the responder to the requester.

3.1.2.28 read transfer: A transfer of data from a contiguous range of bus addresses into device-specific address space. Read transfers are performed by the target in the process of executing an OUTPUT (memory-to-target) or COPY (memory-to-memory) command-entry.

3.1.2.29 requester: A term that describes the node that initiates a bus transaction and transfers an address, command, and (sometimes) data to the responder.

3.1.2.30 reserved; res or r: A term used to describe the fields within unit-specific CSRs. On a write, the reserved-field value shall be ignored. On a read, the reserved field value shall be zero.

3.1.2.31 responder: A term that describes the node that processes a request and returns a response to the requester.

3.1.2.32 Scalable Coherent Interface (SCI) standard: Refers to ISO/IEC 13961 Scalable Coherent Interface, which provides computer-bus-like services using a collection of point-to-point unidirectional links.

NOTE— See ISO/IEC 13961 [B7].

3.1.2.33 Scalable Storage Interface (S²I): This standards document that specifies an interface that can transfer data between storage devices and shared-memory locations.

3.1.2.34 scatter array: A contiguous array of elements that specifies a set of discontinuous address ranges to which a single logical data transfer is performed. A scatter array consists of one or more elements, where each scatter element contains an address pointer and an unsigned length.

3.1.2.35 Serial Bus: Refers to IEEE Std 1394™, 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.

NOTE— See IEEE Std 1394-1995 [B4].

3.1.2.36 status entry: A location within a device buffer that contains completion status from the device-local data-transfer engine.

3.1.2.37 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) between a requester and a responder. The response subaction returns status (and possibly data) from the responder to the requester.

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

3.1.2.39 write transaction: A bus transaction that transfers data from the requester to the responder.

3.1.2.40 write transfer: A transfer of data from a device-specific address space into a contiguous range of bus addresses. Write transfers are performed by the target in the process of executing an INPUT (memory-to-target) or COPY (memory-to-memory) command-entry.

3.2 Field names

This document describes the values in unit control registers, mover control registers, status-report parameters, command entries, status entries, and unit-specific control-and-status registers (CSRs). For clarity, names of these values have an italic font and contain the context as well as field names, as illustrated in Table 3.1.

Table 3.1—Names of command, status, and CSR values

Name	Description
<i>UnitCsr.timeOfDay</i>	The unit's synchronized time-of-day clock.
<i>MoverCsr.control</i>	The mover's control register.
<i>Command.code</i>	The <i>code</i> field within a command entry.
<i>Status.label</i>	The <i>label</i> field within a status entry

3.3 C-code notation

The behavior of data-transfer command execution is frequently specified by C code, such as Equation (3.1). To differentiate such code from textual descriptions, such C-code listings are formatted using a fixed-width Courier font. Similar C-code segments are included within some figures.

```
// Return maximum of a and b values
Max(a,b) {
    if (a<b)
        return(LT);
    if (a>b)
        return(GT);
    return(EQ);
}
```

(3.1)

Since the meaning of many C code operators are not obvious to the casual reader, their meanings are summarized in Table 3.2:

Table 3.2—C code expression summary

Expression	Description
$\sim i$	Bitwise complement of integer i
$i \wedge j$	Bitwise EXOR of integers i and j
$i \& j$	Bitwise AND of integers i and j
$i << j$	Left shift of bits in i by value of j
$i * j$	Arithmetic multiplication of integers i and j
$! i$	Logical negation of Boolean value i
$i \&\& j$	Logical AND of Boolean i and j values
$i j$	Logical OR of Boolean i and j values
$i += j;$	Equivalent to $i = i + j;$
$i == j$	Equality test, true if i equals j
$i != j$	Equality test, true if i does not equal j
$i < j$	Inequality test, true if i is less than j
$i > j$	Inequality test, true if i is greater than j

3.4 Bit and byte ordering

3.4.1 Endian ordering options

This document defines registers and memory locations that are 4 bytes in size. To ensure interoperability across bus standards, the ordering of the bytes within these locations is defined by their relative addresses, not their time slot or physical position on the bus.

The arithmetic significance of data bytes within multibyte data structures is assumed to be big- or little-endian, if the byte with the lowest address is the most- or least significant respectively. Distinct terms are used to describe a bus's address/data multiplexing convention, which is big- or little-addressan if the byte with the lowest address is multiplexed with the most- or least-significant portion of the address.

Thus, a big-endian architecture (byte 0 is the most significant of data bytes at addresses 0...3) can be easily supported on a little-addressan bus, where data-byte 0 is multiplexed with the least-significant byte of the address. The big- or little-addressan order of the bus is not visible to software and is not discussed within other clauses of this S²I standard.

3.4.2 Big-endian byte ordering

A big endian architecture has been assumed in this standard, because:

- a) I/O driver software writers have indicated familiarity with current SCSI commands, and
- b) Big endian order is assumed by CSRs (control and status registers) on Serial Bus and SCI.

For big-endian architectures, the data byte with the lowest address is the most significant, as illustrated in Figure 3.1.

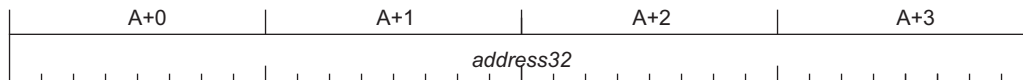


Figure 3.1—Byte ordering within 4-byte locations (big endian)

A 64-bit value is oftentimes represented as a pair of 32-bit values, as illustrated in the top of Figure 3.2. The more and less significant portions of this 64-bit value are mapped to the lower and higher register addresses respectively, and the lower address is always on the top, as illustrated in Figure 3.2. This makes the register-pair ordering consistent with the big-endian byte-ordering conventions.

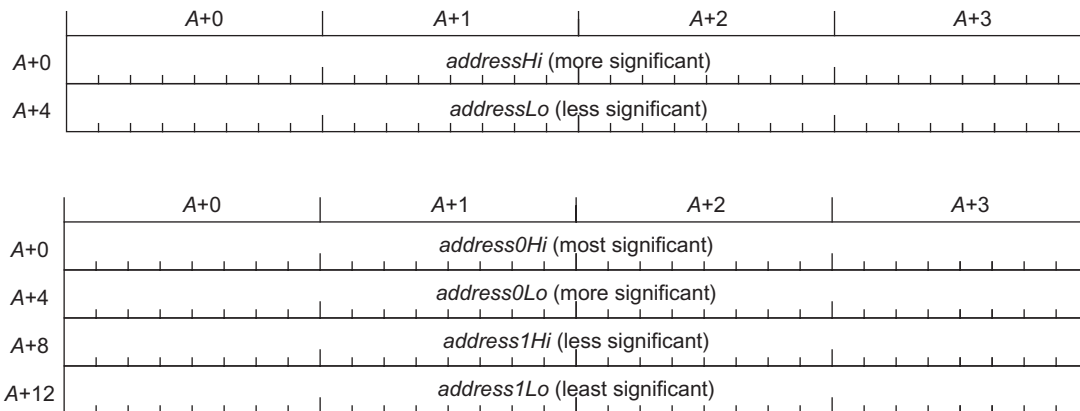


Figure 3.2—Byte ordering within 8-byte and 16-byte locations

A 64-bit value is sometimes represented as a single 64-bit value, rather than a pair of 32-bit values, as illustrated in the bottom of Figure 3.2.

To simplify the drawings and minimize confusion between alternative bit-labelling conventions, the widths of bit fields and their locations are implied by tick marks, where tick marks are placed on the right and left of each bit position.

3.4.3 Little-endian alternatives

From the processor's perspective, an S²I interface is often expected to be attached to a remote I/O bus, which connects to the central processor bus through an address-invariant bridge. In this case, the byte-ordering defined by the S²I interface is visible to processor software. Thus, a little-endian S²I architecture (as designed for a personal computer interface) would still appear to have bytes in a little-endian order, even when accessed from a big-addressan bus, such as the Scalable Coherent Interface.

Systems and software on other buses (PCI Local Bus Specification [B8]) have assumed that all data structures have a little-endian byte ordering (i.e., byte 0 is the least-significant byte of the data bytes at addresses 0...3). On such buses, an alternative little-endian architecture is likely to be mandated. However, the detailed changes required to support little-endian architectures are beyond the scope of this standard.

3.5 Generic delta-level command lists

The Delta16 commands are placed into contiguous arrays, called segments, and these segments may be linked by using a JUMP command as the final command of each segment. Delta16 command lists reflect the stacked nature of their commands, as previously illustrated in Figure 1.5.

The Delta32 and Delta64 commands are linked into linear lists, where each command entry provides a pointer to the next command entry. Illustrations of Delta32/Delta64 command lists reflect the linked entry nature of their structure, as previously illustrated in Figure 1.6.

Because of the common Delta16/32/64 feature sets, it is sometimes convenient to provide generic delta-level list illustrations, where command entries are vertically stacked command-entry lists, as illustrated in Figure 3.3. The vertical nature of the linked command entries is intended to communicate the possibility of a Delta16 command segment, where the command links are implied by the sequential command-entry addresses.

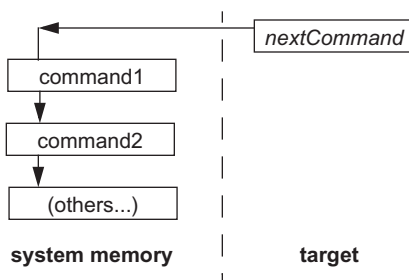


Figure 3.3—Generic delta-level command lists

3.6 Data formats

3.6.1 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”.

3.6.2 Reserved registers and fields

The portions of the command-entry fields, status-entry fields, and unit-specific CSRs that are not implemented have defined reserved values. This includes optional registers (when the option is not implemented) and reserved registers (which are required to be unimplemented), or unused portions of command/status entries. The capabilities of reserved fields are exactly defined, to minimize conflicts between current implementations and future definitions.

For command-entry and status-entry fields, which are located in system-memory or transfer-buffer spaces, the reserved fields and bits shall be zero when the command is written.

Unused fields within register locations may be either *reserved* or *ignored*. The reserved option allows these field locations to be hardwired-to-zero, so that physical memory elements are not required. Alternatively, an ignored field (a read of an ignored field returns the last-written value) allows registers to be special addresses within a general-purpose memory array.

4. Abbreviations and acronyms

This standard contains the following abbreviations and acronyms:

CISC	complex instruction set computer
CSR	control and status register
DRAM	dynamic random-access memory
EUI-64	extended unique identifier, 64 bits
EXOR	exclusive or
I/O	input/output
RAID	redundant array of inexpensive disks
RISC	reduced instruction set computer
SCI	Scalable Coherent Interface
S ² I	Scalable Storage Interface

5. Alpha-level interface

5.1 Alpha-level overview

5.1.1 Alpha-level distinctions

The alpha-level interface has been optimized for low-cost embedded systems, including consumer devices such as cameras, answering machines, and personal communications devices. This alpha-level interface is intended to be a minimal complexity interface with limited function. When compared to the delta-level interfaces, the alpha level has several distinct attributes, as follows:

- a) *Slave-only*. The device-resident mover transfers data between media spaces and device-resident buffers. The host is expected to transfer data between these buffers and system memory.
- b) *Small commands*. Basic commands are 4 bytes in size (although an 8-byte extension is possible).
- c) *Simple queues*. Command entries are written into a circular queue by the host. When the command is completed by the device, the command entry is overwritten with a status-entry value.

Applications requiring a DMA-capable interface (i.e., one that can transfer data directly to system memory) should use a delta-level interface.

5.1.2 Alpha-level components

An alpha-level target consists of physical media for nonvolatile data storage (typically a disk), a device-resident transfer buffer, and a *mover* that can transfer a fixed-sized data block between the buffer and media. Associated with a mover are *command/status* registers comprising a circular array, from which the device executes commands in sequence. Distinct device level *control* and *state* registers are provided, as illustrated in Figure 5.1.

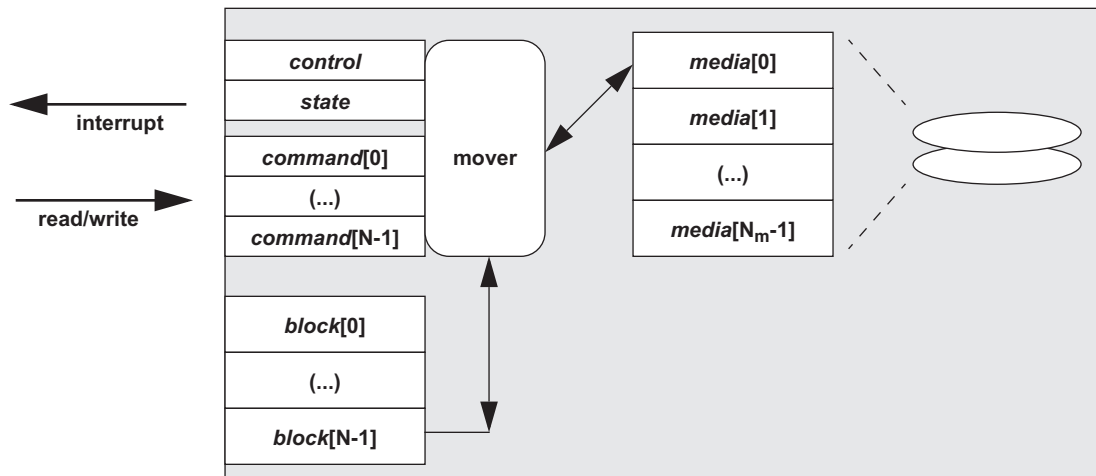


Figure 5.1—Alpha interface components

There are N ($N < 256$) command registers and each command register has an associated buffer block. Each buffer block holds S data blocks, where the block size B is constrained as specified in Equation (5.1).

$$B = 2^p, \text{ where } p \geq 0 \quad (5.1)$$

5.1.3 Alpha-level memory-mapped addresses

The alpha-level interface is expected to be incorporated in single-unit nodes, as illustrated in Figure 5.2. The *control*, *state*, and *commands* registers are mapped into the bus-address space to control and monitor the mover operations. The *block[]* storage buffers are mapped into the bus-address space to facilitate processor-invoked buffer-to-memory and memory-to-buffer data transfers.

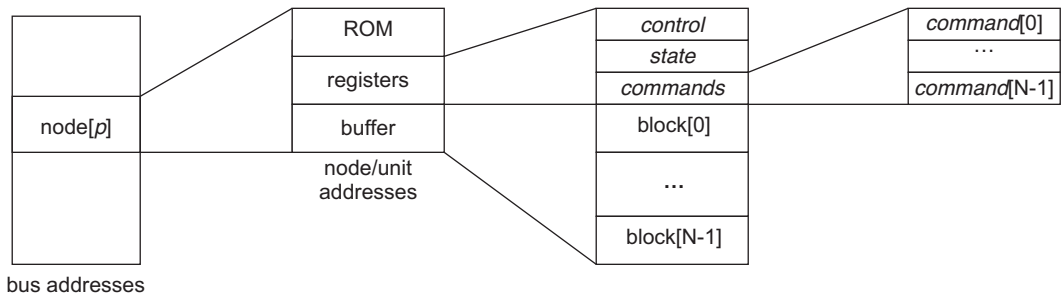


Figure 5.2—Alpha-level memory-mapped addresses

The base location of the descriptive ROM is typically bus dependent, but is expected to provide the information listed below. The detailed device properties d) could be specified by ROM, or could be discovered by using the alpha-level interface to transfer such information from a device-class dependent media location.

- a) *Interface*. Information that differentiates between Alpha4 and Alpha8 interface levels.
- b) *Locations*. The location of the alpha-level interface components.
 - 1) The control and state register locations.
 - 2) The N-element command-block array.
 - 3) The N-element data-block array
- c) *Capacity*. The value of N (this limits the number of simultaneously active commands).
- d) *Properties*. The detailed properties of the attached storage device.

5.1.4 Buffer block access

Each command entry is directly associated with a single buffer block: *command[n]*. Processing of *command[n]* transfers data between media block *media[m]* (where *m* is specified in the command) and buffer *block[n]*. Between the time that a command is written (by the host) and the time that the command is overwritten (by the device) with a status value, the contents of the associated buffer locations are owned by the device. Further changes by the host may yield unspecified results.

After the device writes its status to a command/status entry, ownership of that entry reverts to the host and the device shall ignore the buffer contents.

5.2 Command execution

The alpha-level command list consists of a circular queue, located on the device, which is shared by the host and the device. The host is responsible for writing new 4-byte command entries. The device is responsible for reading these command entries, processing them in FIFO order, and overwriting the command location with a completion-status entry.

5.2.1 Command-list additions

The host is expected to maintain a host-resident *index* value to identify the command-entry location being processed. Before writing a new command entry, the host is expected to check the next-command-entry, delaying the command-entry update if this entry contains status (that has not been processed by the host) or a pending command (that has not been processed by the device).

After the command entry has been written, the *index* value is incremented (modulo N, the number of command entries).

5.2.2 Out-of-order command execution

Out-of-order execution is possible if the illusion of in-order execution is maintained. Commands can therefore be reordered to improve throughput, although in-order command-completion reporting is required. Significant application-software latencies can be associated with these in-order alpha-level reporting requirements.

Applications that expect to benefit from command execution/reporting reordering are encouraged to use one of the more sophisticated Delta16/Delta32/Delta64 interfaces.

5.3 Alpha4 command and status entries

5.3.1 *command* entry format

A *command* entry is written by the initiator and is read and executed by the device. It contains a 4-bit *command* field, whose value affects the mover operations, as illustrated in Figure 5.3.

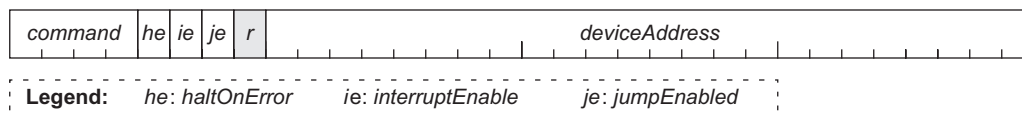


Figure 5.3—*command* entry format

The 4-bit *command* field affects the mover operations; different *command*-code values are used to perform data-transfer operations, as listed in Table 5.1. There are only a few data-transfer commands, which transfer zero or one fixed-sized block between the command's buffer block and the command-specified *deviceAddress* address. These commands are a subset of the Delta interface commands (see 6.6.2).

Table 5.1—Alpha4/Alpha8 *command* (transfer command) values

Value	Name	Description
0	NULL	Not used (corresponds to a status-block entry)
1	INPUT_FETCH	Ordered speculative input
2	OUTPUT_PREP	Setup data-transfer parameters
3	INPUT_POST	Return extended data-transfer status
4	OUTPUT_ORDER	Ordered output
5	INPUT_ORDER	Ordered input

Table 5.1—Alpha4/Alpha8 *command* (transfer command) values (continued)

Value	Name	Description
6	OUTPUT_QUICK	Unordered output
7	INPUT_QUICK	Unordered input
8–15	—	Reserved

The *haltOnError* bit specifies whether command processing should halt if a data-transfer error occurs. If *command.haltOnError* is 1, a data-transfer error sets an internal *halted* bit to 1 and command processing stops. If *command.haltOnError* is 0, *halted* is zero and command processing continues.

The *interruptEnable* bit specifies when interrupts should be sent. If *interruptEnable* is 0, an interrupt is only sent if *halted* is set when the command completes. If *interruptEnable* is 1, an interrupt is always sent when command processing completes.

The *jumpEnable* bit specifies one of two possible locations for the following command. The values of 0 and 1 correspond to the next and first command locations respectively. The *r* bit shall be reserved.

The 24-bit *deviceAddress* field shall specify the media block address being accessed.

NOTE—With a 4096-byte block size, the *deviceAddress* field size allows approximately 2^{36} bytes (more than 64 gigabytes) of data to be directly addressed.

5.3.2 status-entry format

The *status* entry overwrites a command entry when the mover’s command processing completes. It contains a 4-bit *stat* field, whose value reflects the status of the mover’s command processing, as illustrated in Figure 5.4.

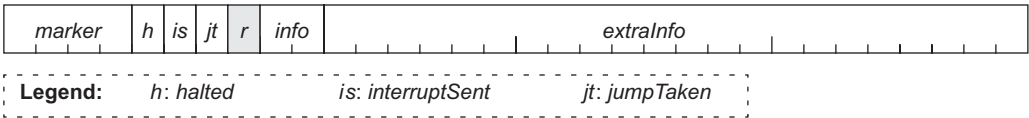


Figure 5.4—status entry format

The *marker* field shall be 0000₂ (zero), to distinguish status entries from command entries that can be placed in the same location.

The *halted* bit is a saved copy of the device’s internal *halted* bit. A one-valued *halted* bit indicates that the device’s *state.index* value has not been incremented, so the next command will be refetched from the same command/status-entry location.

The *interruptSent* bit is 1 if an interrupt event was sent; otherwise *interruptSent* shall be 0.

The *jumpTaken* bit is a copy of the like-named field from the corresponding command entry. The *r* bit shall be reserved.

The 2-bit *info* field reflects the status from the last mover operation, as listed in Table 5.2.

Table 5.2—*info* field values

Value	Name	Description
0	NORM	Normal successful operation
1	WARN	Abnormal successful operation
2	HOPE	Normal failure operation (retry likely to succeed)
3	FAIL	Abnormal failure operation (retry unlikely to succeed)

The 22-bit *extraInfo* field is expected to provide additional diagnostic information, such as the number of transferred bytes or diagnostic codes. The format and function of this field is beyond the scope of this standard.

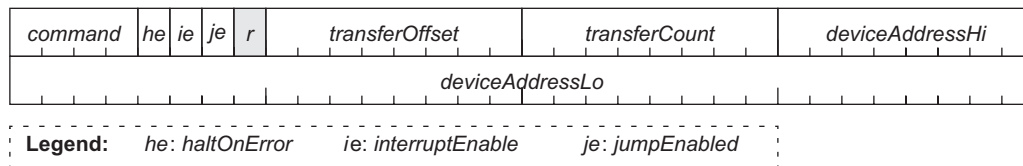
5.4 Alpha8 command and status entries

Although sufficient for some storage devices, the 24-bit *deviceAddress* field of the Alpha4 interface may be insufficient for larger device architectures. The Alpha8 command set option is intended to better fulfill these needs.

The Alpha8 interface provides twice-as-large 64-bit command and status entries, allowing the command to transport a much larger 40-bit *deviceAddress* parameter. The decision to support the Alpha4 or Alpha8 option is expected to be made at the time of device manufacture.

5.4.1 Extended *command* entry format

The Alpha8 interface supports a larger 40-bit *deviceAddress* space, as illustrated in Figure 5.5.

**Figure 5.5—Alpha8 *command* entry format**

The 4-bit *command* field, *haltOnError* bit, *interruptEnable* bit, *jumpEnabled* bit, and the *r* bit are defined in 5.3.1.

The 8-bit *transferOffset* field specifies the offset within the device's memory space. The *transferOffset* field is specified in relative 1/256 portions of the device's memory space. The intent is to support a wide range of device buffer sizes, while constraining the command's offset-address field size.

The 8-bit *transferCount* field specifies the number of transfer blocks, with the exception that 0 (zero) represents the 256 value. The intent is to support efficient transfers of multiple contiguous data blocks, as well as random transfers of individual blocks.

The 8-bit *deviceAddressHi* and 32-bit *deviceAddressLo* fields are concatenated to form the 40-bit *deviceAddress* value; the *deviceAddressHi* field is the more significant portion of the concatenated address.

5.4.2 *status-entry format*

The extended status entry has an additional 32 bits of diagnostic information, as illustrated in Figure 5.6.

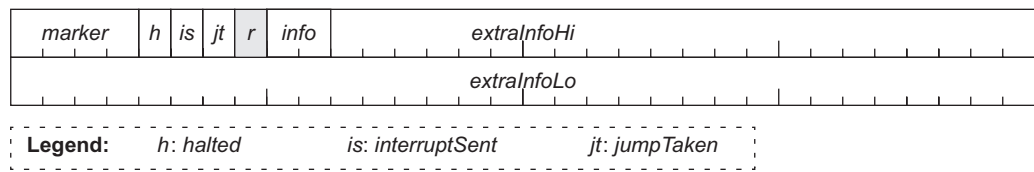


Figure 5.6—Extended *status* entry format

The *marker* field has a value of 0000₂ (zero), to distinguish it from commands placed in the same location. The *halted* bit, the *interruptSent* bit, the *jumpTaken* bit, and the 2-bit *info* field are defined in 5.3.2. The *r* bit shall be reserved.

The 24-bit *extraInfoHi* and the 32-bit *extraInfoLo* fields are expected to provide extended status information, such as the number of transferred bytes or diagnostic codes. The format and function of these fields is beyond the scope of this standard.

5.5 Command processing

Before processing a command entry, the device checks to see if this entry’s *command* field is nonzero; if zero, no command entry is available and the device’s command-processing is delayed. When the *command* field becomes nonzero, the command entry is processed.

Command processing typically involves a data transfer and returns completion-status information. Further command processing depends on the returned status information, as follows:

- a) *Check*. An internal *halted* bit is set if both of the following conditions are met:
 - 1) The command’s *command.haltOnError* bit is equal to one.
 - 2) An error is detected while the command is being processed.
- b) *Report*. The current command entry at the *index*-specified location is overwritten with transfer-completion status, with the effect of setting *status.halted*=*halted*.
- c) *Step*. If *halted* is zero, the current *index* value is incremented by 1 (modulo N).
- d) *Interrupt*. An interrupt is generated if either of the following conditions are met:
 - 1) The internal *halted* bit is one.
 - 2) The command’s interrupt bit, *command.interruptEnable*, is equal to one.

Commands that set *halted* never increment the *index* value, providing the host with two recovery possibilities:

- *Retry*. The host may rewrite the command location with its original value and restart the mover.
- *Continue*. The host may rewrite the command entry with a NOOP command.

For clarity, the command-execution conditions, as affected by the *commands.command*, *commands.haltEnable*, *commands.interruptEnable*, and *commands.jumpEnable* bits, are also illustrated in Annex E.

5.6 Alpha-level CSRs

5.6.1 Alpha-level CSR offset addresses

The alpha-level interface control and state registers (CSRs) are directly mapped into the system-memory address space. The location and names of these registers are summarized in Table 5.3.

Table 5.3—MoverCsr summary

Offset	Register name	Initial value	Read value	Write effect	Description
0	<i>control</i>	zero	data	update	write modifies operational state
4	<i>state</i>	zero	depends	ignore	read-access to state bits

These registers are cleared to their initial values whenever the node is reset. A node reset can occur when power is lost, the I/O card is reset, or a node-reset register is written. The details of such node-reset invocation mechanisms are beyond the scope of this standard.

The *control* and *state* registers control and monitor the device's state, as described in 5.6.4.

5.6.2 control register

The *control* register is written by the initiator and read by the mover hardware. It contains an *op*-code field, whose value affects the mover operations, as illustrated in Figure 5.7.



Figure 5.7—control register format

The 2-bit *op* field affects the mover operations; different *op*-code values are used to initiate, resume, and halt mover-subunit operations, as listed in Table 5.4 and described in 5.6.4. The 30-bit *reserved* field shall be reserved.

Table 5.4—*op* (mover operation) values

Value	Name	Description
0	RUN	Starts command processing
1	—	Reserved
2	STOP	Stops mover operations
3	RESET	Resets mover state to a known initial state

5.6.3 *state* register

The read-only *MoverCsr.state* register is written by the mover hardware and read by the initiator. It contains state fields, whose value reflects the current mover state, as illustrated in Figure 5.8.



Figure 5.8—state format

The 2-bit *stat* field reflects the mover’s operational state, as described in Table 5.5. The *stat* field value is typically changed in response to changes in the *control.op* field, but may also be changed by fatal exception conditions that force a transition to the STOPPED state. The 22-bit *reserved22* field shall be reserved.

Table 5.5—stat (mover operational state) field values

Value	Name	Description
0	RESETTING	Mover being reset
1	RUNNING	Mover operational, running
2	—	Reserved
3	STOPPED	Mover operations stopped

The 8-bit *index* field specifies which command entry is being processed. The initial value of this field shall be zero.

5.6.4 Operational states

When reset, or after power is restored, the transfer unit starts in the *MoverCsr.state.stat*=RESETTING state. The mover remains in this operational state until (power-reset or software sets) *MoverCsr.control.op*=RUN, which eventually changes *MoverCsr.state.stat* to RUNNING, as illustrated in Figure 5.9. Note that within this state-transition table, only the transitions to other operational states are illustrated.

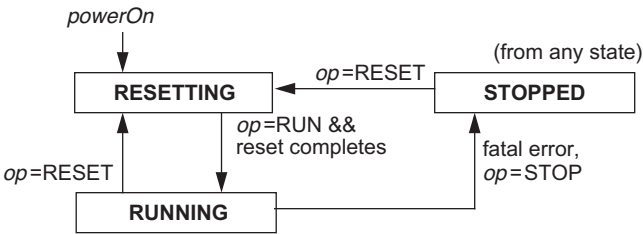


Figure 5.9—Alpha-level operational states

These states and the defined transitions have been designed to facilitate low-cost implementations. Since the writing of a new *MoverCsr.control.op* value is not required to have an immediate effect, the *MoverCsr.control* registers can be contained within a general-purpose register file.

These states affect the mover’s operation, as follows:

- **RESETTING.** The mover is deactivated: fetching and execution of command entries is stopped. The mover's registers are restored to their initial value.
- **RUNNING.** The mover fetches and executes command entries.
- **STOPPED.** The mover is deactivated: fetching and execution of command entries is stopped. The mover's registers retain their state, with the intent of assisting diagnostic software.

While in the **RUNNING**-state, command-list processing can be forcibly stopped, in an asynchronous fashion, by changing the *MoverCsr.control.op* value from **RUN** to **STOP**. Fatal errors encountered during command entry execution also change the *MoverCsr.state.stat* value from **RUNNING** to **STOPPED**.

A reset is the only way that the mover can exit from the **STOPPED**-state. A software invoked reset changes the *MoverCsr.control.op* value to **RESET**, which eventually changes the *MoverCsr.state.stat* value from **STOPPED** to **RESETTING**.

A data-transfer error is not a fatal error, but a *command.haltOnError* value of 1 effectively stops the fetching of additional command entries, as described in 5.5.

After a fatal error condition, the host is expected to change the *MoverCsr.control.op* value to **RESET** and wait until *MoverCsr.state.stat* changes from **STOPPED** to **RESETTING**. The *MoverCsr.control.op* value is then expected to be changed from **RESET** to **RUN**, whereupon *MoverCsr.state.stat* changes from **RESETTING** to **RUNNING** and normal operations can be resumed.

6. Delta interface properties

6.1 Delta-level distinctions

The Delta16, Delta32, and Delta64 interfaces are optimized for uniprocessor systems, multiprocessor systems, and large multiprocessor systems, respectively. Properties that may distinguish these delta-level interfaces from other device-resident DMA architectures include the following:

- a) *Long addresses.* Larger 64-bit or 128-bit system-memory address spaces are assumed, for compatibility with 64-bit Serial Bus and SCI interfaces, as well as extended computer backplane specifications.
- b) *Multiple movers.* Concurrency is supported by defining multiple movers, where each mover has its own command list, so that multiple command lists can be concurrently processed.
- c) *Scatter/gather.* System addresses for one segment can be directly specified; a set of arbitrary-length segments can be indirectly specified.
- d) *Status reports.* The status reporting mechanism is the writing of a 16-byte status block to a pre-specified address. The status block includes a 32-bit command-entry-specified label, so that system software can readily dispatch to the proper interrupt handler.
- e) *Multiple spaces.* Bus-dependent transfer modes/widths can be specified via address modifier bits.

The use of multiple spaces differentiates the delta-level interface from many device-dependent DMA architectures. With the proper access-space selection, the standard OUTPUT command can update mover hardware context (to change the default page size, for example) as well as device-media information. Using distinct addresses to select between different device-specific parameters, rather than defining distinct commands, has two advantages:

- *Extensible.* A large number of commands is easily supported.
- *Flexible.* A wide range of command lengths is readily supported.

6.2 Delta-level memory-mapped addresses

The delta-level interfaces are expected to be incorporated in multiple-unit (as well as single-unit) nodes, as illustrated in Figure 6.1.

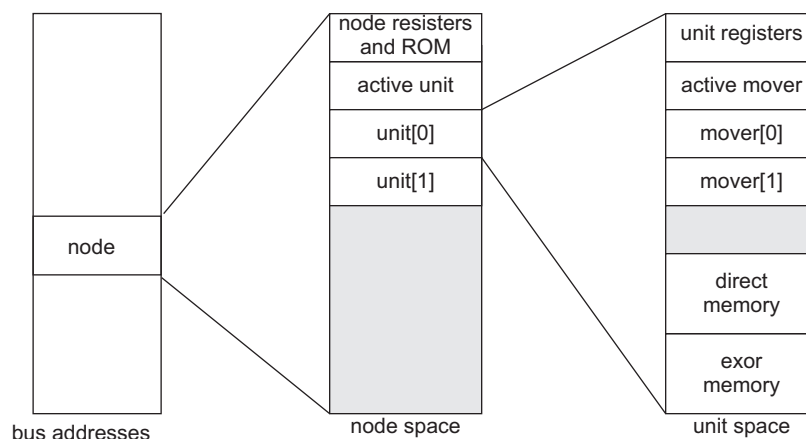


Figure 6.1—Delta-level memory-mapped addresses

For I/O-driver software convenience, portions of the mover-accessed address space correspond to the context-dependent resources associated with the currently active mover. The same resources could be less conveniently accessed using the context-independent unit-specific or mover-specific register set.

The base location of the descriptive ROM is typically bus dependent, but is expected to provide the information listed below. The detailed device properties c) could be specified by ROM, or could be discovered by using the primary delta-level mover to transfer such information from a device-class dependent media location.

- a) *Interface*. Information that differentiates between Delta16, Delta32, and Delta64 interface levels.
- b) *Location*. The base location of the primary delta-level mover registers.
- c) *Properties*. The detailed properties of the attached storage device.

6.2.1 Mover operations

A set of one-or-more movers may be associated with each I/O device. For a half-duplex I/O device, such as a disk, one mover is sufficient to support the device. For a full-duplex network interface, two movers might be associated with the device, so that packet transmissions and receptions can operate independently. In the most general multi-port case, there may be more than two movers per device: parallel input/output channels and control.

Since full-duplex and multi-device architectures are supported through distinct memory-mapped mover addresses, a simple half-duplex S²I disk-device architecture is not burdened with the cost or complexity of supporting full-duplex or multiple-access-port S²I device architectures.

In all cases, each mover is assumed to operate independently. Movers may be implemented as one hardware engine with multiple contexts or as independent hardware engines. When mover hardware is shared in a time-multiplexed on-demand fashion, there shall be no interlocking dependencies between distinct-mover operations.

6.2.2 Mover register addresses

Each mover is expected to be allocated 64 bytes of memory-mapped-register space: the initial portion of this space is defined by this standard; the remaining is allocated for device-specific registers. The configuration and I/O driver software is simplified by defining fixed-size mover-register sets; extra registers (when needed) can also be located in a special portion of the register address space. The defined registers include registers for event reporting, in addition to those providing data-transfer capabilities.

Device identification and configuration parameters oftentimes include device-specific parameters that specify software-driver interfaces, device capabilities, performance capabilities, storage size, implemented options, and access times. Such parameters are expected to be located in the device-ROM space, which can be accessed indirectly through one of the movers.

The device's power-control registers may be located in the device-register space, which is also indirectly-accessed. These registers may be accessed to set the desired power-level setting and to determine the current power-level setting. See B.5 for details.

6.2.3 Device-buffer addressing

Local memory is an option that is characteristic of special applications, such as RAID-supportive disk drive interfaces (see Annex C). Such interfaces may map the initial portions of their device-buffer memory into two address ranges, *buffer* (which starts at *A*) and *exorBuffer* (which starts at *A+size*), as illustrated in Figure 6.1. The bus-interface logic is responsible for checking the address and (when appropriate) performing an exclusive-or operation, rather than the normal update operation.

A write to an address within the *buffer* range places the write-data value into the specified location. A write to an address within the *exorBuffer* range exclusive-or's the data with the same-offset-location within the *buffer* space, as specified within Equation (6.1).

```

Write(addr, data) {
    if (addr < 2*size) {
        if (addr < size)
            buffer[addr]= data;
        else
            buffer[addr-size]^= data;
    }
}

```

(6.1)

NOTE—The specification/implementation of address-dependent write-transaction side effects has often been done for memory-mapped frame-buffers, where side effects have included {OR,EXOR,AND,NAND} for bit-mapped black-and-white displays, as well as color blending and depth-dependent updates on more sophisticated displays. The implementation involves read/modify/write memory cycles, using write-transaction data values.

The combination of the exclusive-or and mover-synchronization capabilities means that RAID-like functionality can be readily implemented by configuring multiple storage devices and customizing the I/O driver software to coordinate command-list processing. Although less efficient than specialized hardware, since bus bandwidth is consumed transferring data between devices, this RAID-storage system can be much less expensive.

6.3 Address formats

6.3.1 Delta16/Delta32 address formats

For Delta16 and Delta32 commands, 64-bit system addresses are assumed. Each *systemAddress* component consists of two 32-bit components, as illustrated in Figure 6.2.

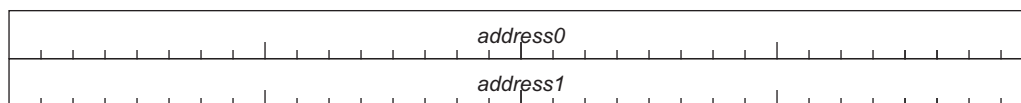


Figure 6.2—Unaligned Delta16/Delta32 address formats

Other address values are 16-byte aligned and the four otherwise unused least-significant bits of these addresses specify bus-dependent data-transfer modes, as illustrated in Figure 6.3.

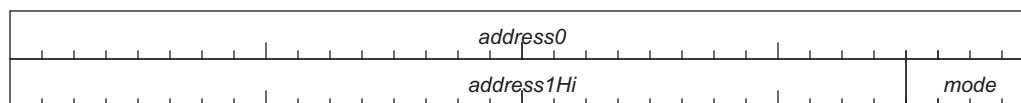


Figure 6.3—Aligned Delta16/Delta32 address formats

The bus-dependent 4-bit *mode* field may be used to specify data-transfer modes and/or speeds (see D.2.3).

6.3.2 Delta64 address formats

For Delta64 commands, 128-bit system addresses are assumed. Depending on the attached bus standard, these may be used to specify addresses in a much larger bus-address space; alternatively, the address may be partitioned into stable and semistable components (see D.2.1).

When directly addressing memory, the *systemAddress* component specifies portions of a 128-bit address. The *systemAddress* component consists of four 32-bit components, as illustrated in Figure 6.4.



Figure 6.4—Unaligned Delta64 address formats

Other address values are 16-byte aligned and the four otherwise unused least-significant bits of these addresses specify bus-dependent data-transfer modes, as illustrated in Figure 6.5.

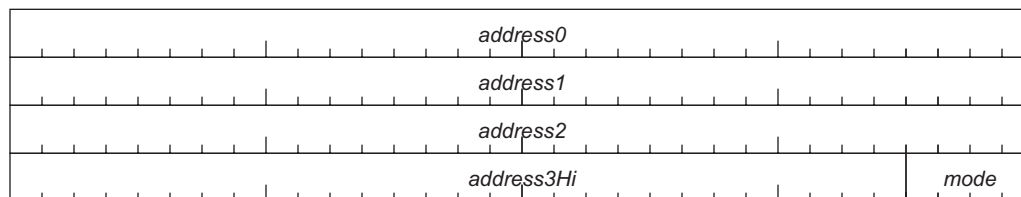


Figure 6.5—Aligned Delta64 address formats

6.3.3 Address modes

The bus-dependent 4-bit *mode* field is used to specify data-transfer modes, address/data widths, and/or data-transfer speeds (see D.2.3). Two of these mode values have special meaning, as specified by Table 6.1; the remaining values are available for bus-dependent definitions.

Table 6.1—Delta16 *mode* field values

Value	Name	Address	Description
0	NULL	NULL	Invalid address, operations are nullified
1	VOID	command	Fetch next-command address only
		scatter-array	Command relative addressing
		report	
2-15	—	—	Bus-dependent access <i>mode</i> specifier