Working Draft

T10
Project 1467D

Revision 3b
May 7, 2003

Information technology — Serial Bus Protocol 3 (SBP-3)

T10 Technical Editor:

Peter Johansson
Congruent Software, Inc.
98 Colorado Avenue
Berkeley, CA 94707
USA

(510) 527-3926
(510) 527-3856 FAX

PJohansson@ACM.org

Reference numbers
ISO/IEC xxxxx-xxx:200x
ANSI INCITS xxx-200x

Printed May 7, 2003

Points of contact

T10 Chair:

John B. Lohmeyer
LSI Logic, Inc.
4420 Arrows West Drive
Colorado Springs, CO 80907-3444
USA

(719) 533-7560
(719) 533-7036 FAX
Lohmeyer@T10.org

T10 Vice-Chair:

George O. Penokie
IBM/Tivoli
3605 Highway 52 North, MS 2C6
Rochester, MN 55901
USA

(507) 253-5208
(507) 253-2880 FAX
GOP@US.IBM.com

T10 URLs:

ftp://ftp.t10.org
http://www.t10.org

T10 Reflector:

T10@T10.org
Majordomo@T10.org (to subscribe)

IEEE 1394 Reflector:

STDS-1394@IEEE.org
ListServ@IEEE.org (to subscribe)

Document distribution:

INCITS Online Store
Techstreet
1327 Jones Drive
Ann Arbor, MI 48105

http://www.techstreet.com/ncits.html

(800) 699-9277
(734) 302-7811 FAX

Global Engineering
15 Inverness Way East
Englewood, CO 80112-5704
USA

http://global.ihs.com/

(800) 854-7179
(303) 792-2181
(303) 792-2192 FAX

INCITS Secretariat:

INCITS Secretariat
1250 I Street NW, Suite 200
Washington, DC 20005
USA

(202) 737-8888
(202) 638-4922 FAX

ANSI®
INCITS xxx-200x

American National Standard
for Information Systems –

Serial Bus Protocol 3 (SBP-3)

Secretariat

Information Technology Industry Council

Not yet approved

American National Standards Institute, Inc.

Abstract

This standard specifies a protocol for the transport of commands, data and status between devices connected by Serial Bus, a memory-mapped split-transaction bus defined by IEEE Std 1394-1995, Standard for a High Performance Serial Bus as amended by IEEE Std 1394a-2000 and IEEE Std 1394b-2002.

American National Standard

Approval of an American National Standard requires verification by ANSI that the requirements for due process, consensus and other criteria for approval have been met by the standards developer.

Consensus is established when, in the judgment of the ANSI Board of Standards Review, substantial agreement has been reached by directly and materially affected interests. Substantial agreement means much more than a simple majority, but not necessarily unanimity. Consensus requires that all views and objections be considered and that effort be made towards their resolution.

The use of American National Standards is completely voluntary; their existence does not in any respect preclude anyone, whether he has approved the standards or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standards.

The American National Standards Institute does not develop standards and will in no circumstances give interpretation on any American National Standard. Moreover, no person shall have the right or authority to issue an interpretation of an American National Standard in the name of the American National Standards Institute. Requests for interpretations should be addressed to the secretariat or sponsor whose name appears on the title page of this standard.

CAUTION NOTICE: This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken periodically to reaffirm, revise, or withdraw this standard. Purchasers of American National Standards may receive current information on all standards by calling or writing the American National Standards Institute.

CAUTION NOTICE: The developers of this standard have requested that holders of patents that may be required for the implementation of this standard, disclose such patents to the publisher. However, neither the developers nor the publisher has undertaken a patent search in order to identify which, if any, patents may apply to this standard.

As of the date of publication of this standard, following calls for the identification of patents that may be required for the implementation of the standard, notice of one or more claims has been received. By publication of this standard, no position is taken with respect to the validity of this claim or of any rights in connection therewith. The patent holders have, however, filed a statement of willingness to grant a license under these rights on reasonable and nondiscriminatory terms and conditions to applicants desiring to obtain such a license. Details may be obtained from the publisher.

No further patent search is conducted by the developer or the publisher in respect to any standard it processes. No representation is made or implied that licenses are not required to avoid infringement in the use of this standard.

Published by

American National Standards Institute
1430 Broadway, New York, NY 10018

Printed in the United States of America

Contents

Page

Tables

Figures

Annexes

Foreword (This foreword is not part of American National Standard INCITS xxx-200x)

This standard defines a transport protocol within the domain of Serial Bus, IEEE 1394, that is designed to permit efficient, peer-to-peer operation of input output devices (disks, tapes, printers, *etc.*) by upper layer protocols such as operating systems or embedded applications. Vendors that wish to implement devices that connect to Serial Bus may follow the requirements of this and other normatively referenced standards to manufacture an SBP-3 compliant device.

There are ten annexes in this standard. Annexes A, B, C, D and E are normative and part of this standard. Annexes F through J, inclusive, are informative and are not considered part of this standard.

Requests for interpretation, suggestions for improvement and addenda, or defect reports are welcome. They should be sent to the INCITS Secretariat, Information Technology Industry Council, 1250 I Street NW, Suite 200, Washington, DC 20005-3922.

This standard was processed and approved for submittal to ANSI by InterNational Committee for Information Technology Standardization (INCITS). Committee approval of this standard does not necessarily imply that all committee members voted for approval. At the time it approved this standard, INCITS had the following members:

Karen Higginbottom, Chair
Russ Richards, Vice-Chair
Jennifer Garner, Secretary

Organization Represented	*Name of Representative*
Apple Computer, Inc	David Michael
Hewlett-Packard Company	Karen Higginbottom
Hitachi America, Ltd.	John Neumann
IBM Corporation	Ronald F. Silletti
Institute for Certification of Computer Professionals	Kenneth M. Zemrowski
Microsoft Corporation	Mike Ksar
National Institute of Standards & Technology	Michael Hogan
Open Strategies, Inc.	John Neumann
Oracle Corporation	Donald R. Deutsch
Panasonic Technologies, Inc	Terence Nelson
SHARE Inc.	Dave Thewlis
Sony Electronics, Inc.	Ed Barrett
Sun Microsystems, Inc.	John Hill
Unisys Corporation	Arnold F. Winkler
US Department of Defense/DISA	Russ Richards
Xerox Corporation	Kathleen O'Reilly

Technical Committee T10 on Lower Level Interfaces, which developed and reviewed this standard, had the following members:

John B. Lohmeyer, Chair
George Penokie, Vice-chair
Ralph O. Weber, Secretary

P. Aloisi
C. Binford
T. Bradshaw
J. Breher
C. Brill
R. Cummings
Z. Daggett
C. DeSanti
R. Elliott
P. Entzel
M. Evans
B. Forbes
W. Galloway
E. Gardner
R. Griswold
R. Haagens
N. Hastad
E. Hill
G. Houlder
T. Hui
P. Johansson
S. Jones
T. Kasebayashi
E. Lew
K. Marks
W. McFerrin
K. Moe
C. Monia
D. Moore
J. Neer
T. Nelson
R. Nixon
M. O'Dell
E. Oetting
D. Peterson
D. Piper
B. Raudebaugh
R. Roberts
G. Robinson
C. Simpson
D. Wagner
M. Wingard

The T10 SBP-3 working group had the following participants:

Peter Johansson, Chair
Eric Anderson, Secretary

A. Green
R. Botchek
T. Bradshaw
D. Colegrove
B. Fairman
F. Farhoomand
L. Farrell
L. Flake
J. Fuller
P. Grunwald
K. Hasan
R. Haydt
D. Hunter
W. Jones
D. Knudson
R. Lash
R. Lawson
F. Nordby
N. Obr
S. Powers
C. Rice
R. Roberts
W. Russell
S. Smyers
M. Teener
T. Thaler
S. Ueda
D. Wooten

Revision history

Revision 1 (January 5, 2001)

First release of working draft. Isochronous material from SBP-2 Revision 3c, March 21, 1998, has been incorporated. Although the material received careful review by the SBP-2 working group, its inclusion in this draft is intended as a starting point and should not be considered prejudicial to new proposals.

Minor editorial corrections have been made throughout, in particular references to standards and draft standards have been revised to accurately reflect the current state of affairs.

Editorial changes have been made in the section on configuration ROM so that its terminology matches that of the revised CSR Architecture, draft standard IEEE P1212.

Revision 1a (February 20, 2001)

Added "fast start" facilities and methods described in 01-057r1.

Clarified mandatory *vs.* optional target implementation requirements for task management functions.

Emphasized that a status block is to be stored at the initiator's *status_FIFO* once, and once only, for the corresponding ORB.

Added the BROADCAST_CHANNEL register to the table of CSRs required if a target implements optional isochronous support. IEEE Std 1394a-2000 established this additional requirement for isochronous resource manager capable nodes.

Updated Annex F to use current terminology from SAM-2.

Created a bibliography for references of interest that are not necessarily normative inclusions within the standard.

Miscellaneous editorial clarifications and minor corrections have been made throughout.

Revision 1b (April 23, 2001)

The material in 01-070r0 concerning “bridge-aware” targets and node handles was incorporated into the draft.

Usage of the page table entries in the FAST_START register was clarified.

Instance directories and keyword leaves were added to the illustration of typical configuration ROM data structures.

A target may interpret a write to a fetch agent FAST_START register as if the first eight bytes of the data payload had been written to the ORB_POINTER register.

This does not result in the same increase in efficiency, but may be useful if full “fast start” functionality is not supported for all of a target’s fetch agents.

An initiator is required to report completion status to its application clients in the same order it is received at the *status_FIFO*.

Error recovery procedures applicable to the ORB_POINTER register are also valid for the new FAST_START register.

Revision 1c (May 31, 2001)

Created a new normal command block ORB that includes two data descriptors. This permits the use of two data buffers, each with a data transfer direction independent of the other.

The GET NODE HANDLE management function was extended to permit the release of a previously allocated node handle. As a consequence, the name was changed to NODE HANDLE.

The usage of instance directories was clarified and an instance directory was added to the configuration ROM examples in Annex F.

Revision 1d (August 10, 2001)

The password field in the login ORB is no longer overloaded with an EUI-64 value. As a consequence, the *aware* field was reduced in size to a bit.

Page tables and the data buffers they describe are not required to reside in the same node. Target support for this capability is optional and described by the Unit_Characteristics configuration ROM entry.

The clauses that describe configuration ROM entries in the unit and logical unit directories were reorganized to clarify which entries are mandatory or optional in these drectories. This rendered most of B.3 redundant and the affected text has been deleted.

The Revision entry has been added and the value of Specifier_ID changed to 01 0483_{16} (the same value used by SBP-2). This makes SBP-3 targets available for discovery by enumeration software written for SBP-2.

The Firmware_Revision entry is permitted in logical unit directories as well as the parent unit directory. Its value is not inherited.

Text that defines the cycle mark index was removed.

Revision 1e (October 1, 2001)

The FAST_START facility has been changed to include a *previous_ORB* pointer, as ratified by the working group in Cupertino, CA. The modification makes it simpler for multiprocessor initiators to use fast start.

Revision 1f (February 5, 2002)

Interim status, which may be stored no more than once for a particular ORB, was invented to support the transport of AV/C by SBP-3. The redefinition the *src* value previously assigned to isochronous error reports caused their deletion from the draft.

The configuration ROM section was revised to include new features from IEEE Std 1212-2001: instance directories and keyword leaves.

An editorial revision was made to Annex C in the hope of reducing reader confusion about password matching.

Former Annex H, AV/C Encapsulation, was revised and promoted from informative to normative, in accordance with 01-287r0. The new designation is Annex D.

Revision 1g (March 26, 2002)

Draft standards IEEE P1212 and IEEE P1394b were recently approved by IEEE RevCom as IEEE Std 1212-2001 and IEEE Std 1394b-2002, respectively. The references in SBP-3 have been revised accordingly.

Police the usage of “target fetch agent” *vs.* “logical unit fetch agent” throughout the draft.

Emphasize that targets shall not assume that the two least significant bits of a 48-bit Serial Bus address pointer are zero—even though they are reserved and that is the case at present.

Clarify the meaning of *page_size* in cases where no page table is referenced by the ORB.

Change the node handle management ORB to add an *allocate* bit.

Improve the definition of the *mgt_ORB_timeout* field in the Unit_Characteristics entry so as to clearly differentiate initiator and target usage.

In Annex D, explain in more detail how AV/C interim and final response frames may share the same buffer.

In Annex E, correct the value shown for the Version entry in configuration ROM and add the Revision entry to the examples.

In Annex F and elsewhere, differentiate between local bus split time-out and remote time-out.

Revision 2 (March 26, 2002)

Subsequent to a vote by the T10 plenary to stabilize portions of SBP-2, this revision has been prepared; it is essentially identical to Revision 1g but without the change bars.

The sections stabilized by the plenary cover the FAST_START facility. The stabilized sections are enumerated below:

Clause	Description
5.2.3	Node selectors
6.4.6	FAST_START register
7.7.11	Fast_Start entry
9.1.5	Fetch agent state machine

For readers unfamiliar with T10 process, stabilization is a significant milestone in the development of a standard. Once a document or portions thereof are stabilized they are not to be modified unless either a) there is a demonstrable flaw in the draft standard or b) the changes are agreed to by a two-thirds vote of the T10 plenary in which at least half of the membership votes.

Revision 2a (June 5, 2002)

Minor technical clarifications discussed and recorded in the minutes of the May 29 – 30 working group meeting at Timberline Lodge.

The login ORB is revised to include an *update* bit; this permits parameters of a login to be changed without a logout followed by a login. The feature may be useful during the control hand-off between BIOS and operating system during boot.

The text in 02-069r2, bridge-aware target operations, has been included in the draft as directed by the working group.

Revision 2b (July 25, 2002)

In accordance with a motion approved by the working group at the Colorado Springs meeting, the changes proposed in 01-287r1 have been incorporated in this revision of the draft. Minor changes to the proposal endorsed by the working group, *e.g.*, the alteration of CREATE STREAM to CREATE TASK SET, are reflected in the draft. Although 01-287r1 instructed the editor to delete former Annex H, “Common isochronous packet (CIP) format”, it has been retained pending further discussions on isochronous facilities.

The use of type error response to reject Serial Bus request subactions addressed to fetch agent CSRs from nodes other than the logged-in initiator has been clarified. Note that during a reconnect hold period, the source ID of the initiator is temporarily unknown and its request subactions will also be rejected.

After a discussion of possible failures in the analysis of self-ID packets, the working group agreed that correlation between EUI-64 and physical ID obtained from such an analysis should be considered provisional. The target is required to confirm the EUI-64 by reading the node's bus information block in configuration ROM.

Bridge-aware logins for which the initiator is connected to the same bus as the target are not affected by net update.

The login procedure described in C.2 has been expanded so that it more closely follows the analogous login procedure in 8.2.1.

Revision 2c (January 10, 2003)

Minor editorial clarifications and corrections have been made throughout the draft.

References to incorrect terminology (*e.g.*, kilobyte, 1000 bytes, which has been superceded by kibibyte, 1024 bytes) have been deleted.

The draft has been reviewed for correct usage of “logical unit” *vs*. target.

The circumstances in which a status block is stored after ORB completion have been clarified.

The clauses on bridge-awareness have been revised to reflect changes in IEEE P1394.1.

The description of the command block ORB fields affected by the *isochronous* bit has been corrected.

Task set IDs shall be unique within the context of the target, not simply within the context of the associated login.

The definition of the BUSY_TIMEOUT register specifies different initial values than IEEE 1394 and the register is not modifiable by write requests. Target single- and dual-phase behavior is specified and the same behavior is recommended for initiators.

Reconnect hold periods are timed separately for different fetch agents.

A FAST_START entry has been added to the configuration ROM examples.

Revision 2d (February 27, 2003)

Minor editorial clarifications and corrections have been made throughout the draft.

An extended status block format has been defined; it permits a maximum of 512 status bytes to be returned by target logical units. The login ORB has been modified to permit the initiator to enable the extended status block format. The configuration ROM Logical_Unit_Number entry has been modified so that a logical unit may advertise its extended status block capabilities.

Logical units that implement command sets that utilize both single and dual buffer descriptor command ORBs shall not reject these ORBs with an *sbp_status* of request type not supported. Instead, they shall be able to parse both ORB formats and deliver the command to the device, which shall then indicate its success or failure by command set-dependent means.

Rules that specify the order and presence of page table data when a dual buffer descriptor command block ORB references two page tables have been added to description of the FAST_START register.

The description of the *ORB_size* field in the Unit_Characteristics entry has been clarified; it applies to the largest ORB fetched by any of the target's logical units. The *ORB_size* field does not apply to management ORBs, who size is fixed at 32 bytes.

Added configuration ROM Plug_Control_Register entry as specified by T10/03-009r1.

An informative clause has been added to section 9 that describes how target fetch agents should parse ORBs and page tables, whether read from the initiator's system memory or written to the target's FAST_START register.

The distinction between interim and completion status has been clarified in 9.4.

Revision 2e (March 12, 2003)

Minor editorial clarifications and corrections have been made throughout the draft.

Added informative and normative descriptions of isochronous operations, as specified by T10/03-090r1.

The methods by which a logical unit may reject a command transported within a dual buffer descriptor command ORB are clarified. If the logical unit's command set specifies such commands, even if optional, the logical unit shall not use an *sbp_status* of one, request type not supported, to reject the command.

Revision 3 (March 12, 2003)

This revision has been prepared for ballot by T10 to approve or disapprove its forwarding to INCITS for further standards processing; it is essentially identical to Revision 2e but without the change bars.

Revision 3a (April 24, 2003)

This revision incorporates changes proposed by T10/03-166r1 in resolution of comments received during ballot on SBP-3 Revision 3.

Revision 3b (May 7, 2003)

After working group discussion in Nashua, clarifications were made to the algorithms for logical unit reset and target reset.

American National Standard for Information Systems –

Serial Bus Protocol 3 (SBP-3)

1 Scope and purpose

1.1 Scope

This standard defines a protocol for the transport of commands and data over High Performance Serial Bus, as specified by IEEE Std 1394-1995 [B9] as amended by IEEE Std 1394a-2000 [B10] and IEEE Std 1394b-2002 [B11] (collectively IEEE 1394). The transport protocol, Serial Bus Protocol 3 (SBP-3), requires implementations to conform to the aforementioned standards as well as to IEEE Std 1212-2001, Control and Status Register (CSR) Architecture for microcomputer buses [B6], and permits the exchange of commands, data and status between initiators and targets connected to Serial Bus.

This standard is an evolutionary extension of ANSI NCITS 325-1998, Serial Bus Protocol 2 (SBP-2) [B2], which revises and extends its protocols to take advantage of implementation experience gained subsequent to the development of SBP-2, the continued evolution of High Performance Serial Bus, IEEE 1394, as well as other IEEE Serial Bus standards in development.

1.2 Purpose

A T10 study group convened in Huntington Beach, CA on September 15, 2000 identified a number of areas for which enhancements or extensions to ANSI NCITS 325-1998, Serial Bus Protocol 2, were desired by the industry (see the scope below for details). The consensus of the study group was that a standard compatible with ANSI NCITS 325-1998 should be developed to meet these needs. This document, SBP-3, is the resultant standard.

The significant differences between SBP-2 and this standard are the result of revisions and extensions outlined below:

- methods to reduce a target's start-up latency from an idle condition;
- explicit description of the methods used to encapsulate 16-byte or larger command descriptor blocks (CDBs) within SBP-3;
- extensions necessary for initiators and targets to successfully interoperate across one or more Serial Bus bridges as specified by draft standard IEEE P1394.1 [B7];
- isochronous facilities and methods, with particular attention to data interchange formats that permit the use of removable media;
- definition of a new ORB type to permit bi-directional data transfer in the context of a single task;
- revisions necessary to utilize new Serial Bus features specified by IEEE Std 1394a-2000 and IEEE Std 1394b-2002; and
- clarifications and corrigenda applicable to ANSI NCITS 325-1998.

Although SBP-3 has been designed for Serial Bus as currently specified by IEEE 1394, the Technical Committee anticipates that it will be appropriate for use with future extensions to Serial Bus as they are standardized.

2 Normative references

2.1 Reference scope

The standards named in this section contain provisions which, through reference in this text, constitute provisions of this American National Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision; parties to agreements based on this American National Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

Copies of the following documents can be obtained from ANSI:

Approved ANSI standards;

Approved and draft regional and international standards (ISO, IEC, CEN/CENELEC and ITUT); and

Approved and draft foreign standards (including BIS, JIS and DIN).

For further information, contact the ANSI Customer Service Department by telephone at (212) 642-4900, by FAX at (212) 302-1286 or *via* the world wide web at http://www.ansi.org.

Additional contact information for document availability is provided below as needed.

2.2 Approved references

The following approved ANSI, international and regional standards (ISO, IEC, CEN/CENELEC and ITUT) may be obtained from the international and regional organizations that control them.

IEC 61883-1 (1998-02), Consumer audio/video equipment—Digital interface—Part 1: General

IEEE Std 1212-2001, Standard for a Control and Status Registers (CSR) Architecture for microcomputer buses

IEEE Std 1394-1995, Standard for a High Performance Serial Bus

IEEE Std 1394a-2000, Standard for a High Performance Serial Bus—Amendment 1

IEEE Std 1394b-2002, Standard for a High Performance Serial Bus—Amendment 2

INCITS 351-2001, SCSI Primary Commands 2 (SPC-2)

INCITS 366-2003, SCSI Architecture Model 2 (SAM-2)

ISO/IEC 9899:1999, Programming Languages—C

Throughout this document, the term “IEEE 1394" shall be understood to refer to IEEE Std 1394-1995 as amended by IEEE Std 1394a-2000 and IEEE Std 1394b-2002.

2.3 References under development

At the time of publication, the following referenced standard was still under development.

IEEE P1394.1, Draft Standard for High Performance Serial Bus Bridges

~~T10 Project 1157D, SCSI Architecture Model 2 (SAM-2)~~

3 Definitions and notation

3.1 Definitions

3.1.1 Conformance

Several keywords are used to differentiate 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 standard. Other hardware and software design models may also be implemented.

3.1.1.2 ignored: A keyword that describes bits, bytes, quadlets, octlets or fields whose values are not checked by the recipient.

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

3.1.1.4 reserved: A keyword used to describe objects (bits, bytes, quadlets, octlets and fields) or the code values assigned to these objects in cases where either the object or the code value is set aside for future standardization. Usage and interpretation may be specified by future extensions to this or other standards. A reserved object shall be zeroed or, upon development of a future standard, set to a value specified by such a standard. The recipient of a reserved object shall ignore its value. The recipient of an object defined by this standard as other than reserved shall inspect its value and reject reserved code values.

3.1.1.5 shall: A keyword that indicates a mandatory requirement. Designers are required to implement all such mandatory requirements to assure interoperability with other products conforming to this standard.

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

3.1.2 Glossary

The following terms are used in this standard:

3.1.2.1 byte: Eight bits of data.

3.1.2.2 command block: Space reserved within an operation request block to describe a command intended for a logical unit that controls device functions or the transfer of data to or from device medium. The format and meaning of command blocks are outside of the scope of SBP-3 and are command set- or device-dependent.

3.1.2.3 device server: A component of a logical unit responsible to execute tasks initiated by command blocks that specify data transfer or other device operations.

3.1.2.4 global node ID: A 16-bit address that may be used as the *destination_ID* in an asynchronous primary packet so long as the packet passes through at least one bridge portal en route to its destination. The value of the most significant 10 bits, the bus ID, ranges between zero and $3FE_{16}$, inclusive. Within a net, a bus ID in this range uniquely identifies a single bus. The least significant six bits of a global node ID, the virtual ID, uniquely identify a node on a particular bus.

3.1.2.5 initiator: A node that originates device service or management requests and signals these requests to a target for processing.

3.1.2.6 isochronous channel: A relationship, identified by a channel number, between a node that is the talker and zero or more nodes that are listeners. One isochronous subaction, identified by the channel number, may be sent by the talker during each isochronous period. Channel numbers are allocated cooperatively through isochronous resource management facilities.

3.1.2.7 isochronous period: An operating mode of Serial Bus that occurs, on average, every 125 µs. During an isochronous period, the bus is available to isochronous talkers only. Cooperative allocation of isochronous bandwidth guarantees a bounded worst-case latency for isochronous data.

3.1.2.8 local node ID: A 16-bit address usable as the *destination_ID* in an asynchronous primary packet so long as both the sender and the recipient(s) are on the same bus. The local node ID is the concatenation of $3FF_{16}$ and the node's local ID.

3.1.2.9 logical unit: The part of the unit architecture that is an instance of a device model, *e.g.*, disk, CD-ROM or printer. Targets implement one or more logical units; the device type of the logical units may differ.

3.1.2.10 login: The process by which an initiator obtains access to a logical unit fetch agent. The fetch agent and its control and status registers provide a mechanism for the initiator to signal operation request blocks to the logical unit.

3.1.2.11 login ID: A value assigned by the target during a login or create task set process. The login ID establishes a relationship between an initiator and a task set. ~~Either kind of~~ A login ID is used to identify subsequent requests from an initiator; in some cases the login ID is not present in the operation request block and its value is implicit.

3.1.2.12 listener: A node that receives an isochronous subaction from a channel. There may be zero, one, or more listeners for any given channel.

3.1.2.13 node: An addressable device attached to Serial Bus.

3.1.2.14 node ID: The 16-bit node identifier defined by IEEE 1394 that is composed of a bus ID portion and a physical ID portion. The physical ID is uniquely assigned as a consequence of Serial Bus initialization.

3.1.2.15 node space: The total Serial Bus address space available to each node. Addresses within node space are 48 bits and are based at zero. Node space includes memory space, private space, register space and units space. See either IEEE Std 1212-2001 or IEEE 1394 for more information on address spaces.

3.1.2.16 octlet: Eight bytes, or 64 bits, of data.

3.1.2.17 operation request block: A data structure fetched from system memory by a target in order to execute the command encapsulated within it.

3.1.2.18 quadlet: Four bytes, or 32 bits, of data.

3.1.2.19 receive: When any form of this verb is used in the context of Serial Bus primary packets, it indicates that the packet is made available to the transaction or application layers, *i.e.*, layers above the link layer. Neither a packet repeated by the PHY nor a packet examined by the link is "received" by the node unless the preceding is also true.

3.1.2.20 register: A term used to describe quadlet aligned addresses that may be read or written by Serial Bus transactions. In the context of this standard, the use of the term register does not imply a specific hardware implementation. For example, in the case of split transactions that permit sufficient time between the request and response subactions, the behavior of the register may be emulated by a processor.

3.1.2.21 register space: A 2048 byte portion of node space with a base address of FFFF F000 0000_{16}. Core registers defined by IEEE Std 1212-2001 are located within register space as are Serial Bus-dependent registers defined by IEEE 1394.

3.1.2.22 request subaction: A packet transmitted by a node (the requester) that communicates a transaction code and optional data to another node (the responder) or nodes.

3.1.2.23 response subaction: A packet transmitted by a node (the responder) that communicates a response code and optional data to another node (the requester). A response subaction may consist of either an acknowledge packet or a response packet.

3.1.2.24 split transaction: A transaction that consists of a request subaction followed by a separate response subaction. Subactions are considered separate if ownership of the bus is relinquished between the two.

3.1.2.25 status block: A data structure which may be written to system memory by a target when an operation request block has been completed.

3.1.2.26 store: When any form of this verb is used in the context of data transferred by the target to the system memory of either an initiator or other device, it indicates both the use of Serial Bus write request subactions, quadlet or block, to place the data in system memory and the corresponding response subactions that complete the writes.

3.1.2.27 system memory: The portions of any node's memory that are directly addressable by a Serial Bus address and which accepts, at a minimum, quadlet read and write access. Computers are the most common example of nodes that might make system memory addressable from Serial Bus, but any node, including those usually thought of as peripheral devices, may have system memory.

3.1.2.28 talker: A node that transmits an isochronous packet for a channel during an isochronous period. There shall be no more than one talker for any given channel.

3.1.2.29 target: A unit that receives device service or management requests from an initiator. In the case of device service requests, the commands are directed to one of the target's logical units to be executed. Management requests are serviced by the target. A CSR Architecture unit is synonymous with a target.

3.1.2.30 task: A task is an organizing concept that represents the work to be done by a logical unit to carry out a command encapsulated by an operation request block. In order to perform a task, a logical unit maintains context information for the task, which includes (but is not limited to) the command, parameters such as data transfer addresses and lengths, completion status and ordering relationships to other tasks. A task has a lifetime, which commences when the task is entered into the logical unit's task set, proceeds through a period of execution by the logical unit and finishes either when completion status is stored at the initiator or when completion may be deduced from other information. While a task is active, it makes use of logical unit, target and initiator resources.

3.1.2.31 task set: A group of tasks available for execution by a logical unit of a target. This standard specifies some dependencies between individual tasks within the task set but there may be others not specified by this standard.

3.1.2.32 task set ID: A synonym for login ID when the login ID has been returned by the target in response to a create task set request.

3.1.2.33 transaction: A Serial Bus request subaction and the corresponding response subaction. The request subaction transmits a transaction code (such as quadlet read, block write or lock); some request subactions include data as well as transaction codes. The response subaction is null for transactions with

broadcast destination addresses or broadcast transaction codes; otherwise it returns completion status and possibly data.

3.1.2.34 unit: A component of a Serial Bus node that provides processing, memory, I/O or some other functionality. Once the node is initialized, the unit provides a CSR interface that is typically accessed by device driver software at an initiator. A node may have multiple units, which normally operate independently of each other. Within this standard, a unit is equivalent to a target.

3.1.2.35 unit architecture: The specification of the interface to and the services provided by a unit implemented within a Serial Bus node. This standard is a unit architecture for SBP-3 targets.

3.1.2.36 unit attention: A state that a logical unit maintains while it has unsolicited status information to report to one or more logged-in initiators. A unit attention condition shall be created as described elsewhere in this standard or in the applicable command set- and device-dependent documents. A unit attention condition shall persist for a logged-in initiator until a) unsolicited status that reports the unit attention condition is successfully stored at the initiator or b) the initiator's login becomes invalid or is released. Logical units may queue unit attention conditions; after the first unit attention condition is cleared, another unit attention condition may exist.

3.1.2.37 units space: A portion of node space with a base address of FFFF F000 0800_{16}. This places units space adjacent to and above register space. The CSRs and other facilities defined by unit architectures are expected to lie within this space.

3.1.2.38 working set: The part of a task set that has been fetched from the initiator by the target and is available to the target in its local storage.

3.1.3 Abbreviations

The following are abbreviations that are used in this standard:

CIP	Common isochronous packet format
CSR	Control and status register
CRC	Cyclical redundancy checksum
~~DVCR~~	~~Digital video cassette recorder~~
EUI-64	Extended Unique Identifier, 64-bits
iMPR	input master plug register
iPCR	input plug control register
LBA	Logical block address
LUN	Logical unit number
~~MPEG~~	~~Motion picture experts group~~
oMPR	output master plug register
oPCR	output plug control register
ORB	Operation request block
SAM-2	SCSI Architecture Model 2 [B14]
SBP-3	Serial Bus Protocol 3 (this standard itself)
SPC-2	SCSI Primary Commands 2 [B12]

3.2 Notation

~~The following conventions should be understood by the reader in order to comprehend this standard.~~

3.2.1 Numeric values

Decimal and hexadecimal ~~and, occasionally, binary numbers~~ are used within this standard. By editorial convention, decimal numbers are most frequently used to represent quantities or counts. Addresses are uniformly represented by hexadecimal numbers. Hexadecimal numbers are also used when the value represented has an underlying structure that is more apparent in a hexadecimal format than in a decimal format. ~~Binary numbers are used infrequently and generally limited to the representation of bit patterns within a field.~~

Decimal numbers are represented by Arabic numerals without subscripts or by their English names. Hexadecimal numbers are represented by digits from the character set 0 – 9 and A – F followed by the subscript 16. ~~Binary numbers are represented by digits from the character set 0 and 1 followed by the subscript 2.~~ When the subscript is unnecessary to disambiguate the base of the number it may be omitted. For the sake of legibility~~, binary and~~ hexadecimal numbers are separated into groups of four digits separated by spaces.

As an example, 42 and $2A_{16}$ ~~and 0010 1010$_2$ all~~ both represent the same numeric value.

3.2.2 Bit, byte and quadlet ordering

SBP-3 is defined to use the facilities of Serial Bus, IEEE 1394, and therefore uses the ordering conventions of Serial Bus in the representation of data structures. In order to promote interoperability with memory buses that may have different ordering conventions, this standard defines the order and significance of bits within bytes, bytes within quadlets and quadlets within octlets in terms of their relative position and not their physically addressed position.

Within a byte, the most significant bit, *msb*, is that which is transmitted first and the least significant bit, *lsb*, is that which is transmitted last on Serial Bus, as illustrated below. The significance of the interior bits uniformly decreases in progression from *msb* to *lsb*.

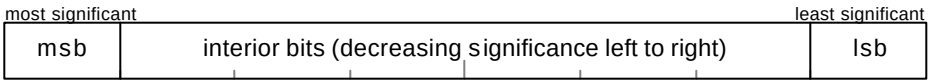

Figure 1 – Bit ordering within a byte

Within a quadlet, the most significant byte is that which is transmitted first and the least significant byte is that which is transmitted last on Serial Bus, as shown below.

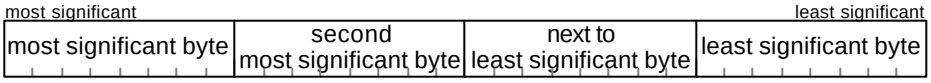

Figure 2 – Byte ordering within a quadlet

Within an octlet, which is frequently used to contain 64-bit Serial Bus addresses, the most significant quadlet is that which is transmitted first and the least significant quadlet is that which is transmitted last on Serial Bus, as the figure below indicates.

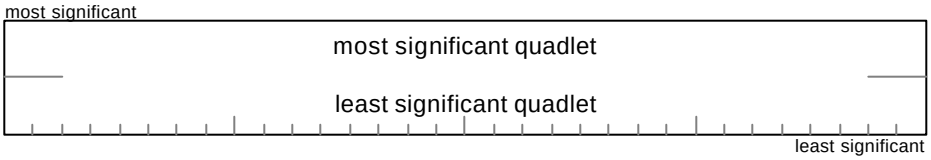

Figure 3 – Quadlet ordering within an octlet

When block transfers take place that are not quadlet aligned or not an integral number of quadlets, no assumptions can be made about the ordering (significance within a quadlet) of bytes at the unaligned beginning or fractional quadlet end of such a block transfer, unless an application has knowledge (outside of the scope of this standard) of the ordering conventions of the other bus.

3.2.3 Register specifications

This standard defines the format and function of control and status registers, CSRs. Some of these registers are read-only, some are both readable and writable and some generate special side effects subsequent to a write.

In order to define CSRs, their bit fields, their initial values and the effects of read, write or other transactions, the format illustrated by Figure 4 is used.

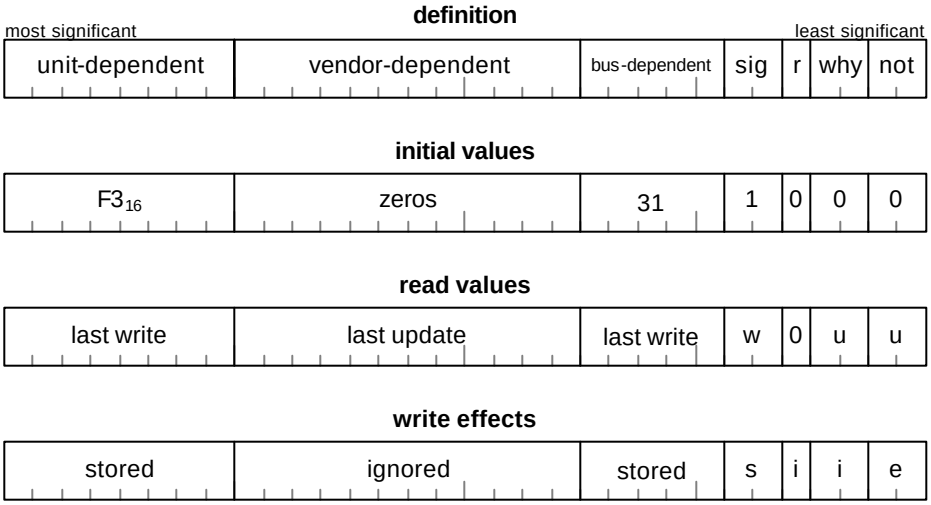

Figure 4 – CSR specification example

The register definition contains the names of register fields. The names are intended to be descriptive, but the fields are defined in the text; their function should not be inferred solely from their names. However, the following field names have defined meanings.

Name	Abbreviation	Definition
bus-dependent		The meaning of the field is defined by the bus standard, in this case IEEE 1394
reserved	r	The field is reserved for future standardization (see 3.1.1)
unit-dependent		The meaning of the field shall be defined by the organization responsible for the unit architecture
vendor-dependent		The meaning of the field shall be defined by the node's vendor

CSRs shall assume initial values upon the restoration of power (a power reset) or upon a write to the node's RESET_START register (a command reset). If the power reset values differ from the command reset values, they are separately and explicitly defined. Initial values for register fields may be described as numeric constants or with one of the terms defined for the register definition. Values for register fields subsequent to a reset may be described in the same terms or as defined below.

Name	Abbreviation	Definition
unchanged	x	The field retains whatever value it had just prior to the power reset, bus reset or command reset.

In addition to numeric values for constant fields, the read values returned in response to a quadlet read transaction may be specified by the terms below.

Name	Abbreviation	Definition
last write	w	The value of the field shall be either the initial value or, if a write or lock transaction addressed to the register has successfully completed, the value most recently stored in the field.[1]
last update	u	The value of the field shall be that most recently updated by the node hardware or software. An updated field value may be the result of a write effect to the same register address, a different register address or some other change of condition within the node.

The effects of data written to the register are specified by the terms below.

Name	Abbreviation	Definition
effect	e	The value of the data written to the field may have an effect on the node's state, but the effect ~~may~~ might not be immediately visible by a read of the same register. The effect may be visible in another register or ~~may~~ might not be visible at all.
ignored	i	The value of the data written to the field shall be ignored; it shall have no effect on the node's state.
stored	s	The value of the data written to the field shall be immediately visible by a read of the same register; it may also have other effects on the node's state.

[1] For clarity, read values for a field in a register that accepts lock transactions may be described as *last successful lock* rather than *last write*. However, the abbreviation in both cases remains *w*. Similar liberties may be taken with the use of *conditionally stored* in place of *stored* when the action occurs as the result of a lock transaction, but the corresponding one-letter abbreviation, *s*, is also unchanged.

Reserved fields within a register shall be explicitly described with respect to initial values, read values and write effects. Initial values and read values shall be zero while write effects shall be ignored. CSRs that are not implemented, either because they are optional or they fall within a reserved address space, shall abide by these same conventions if a successful completion response is returned for a read, write or lock request.

3.2.4 State machines

All state machines in this standard are defined in the style illustrated by Figure 5. The conditions and actions of the state machine transitions are formally defined by C code, as specified by ISO/IEC 9899:1990 [B15].

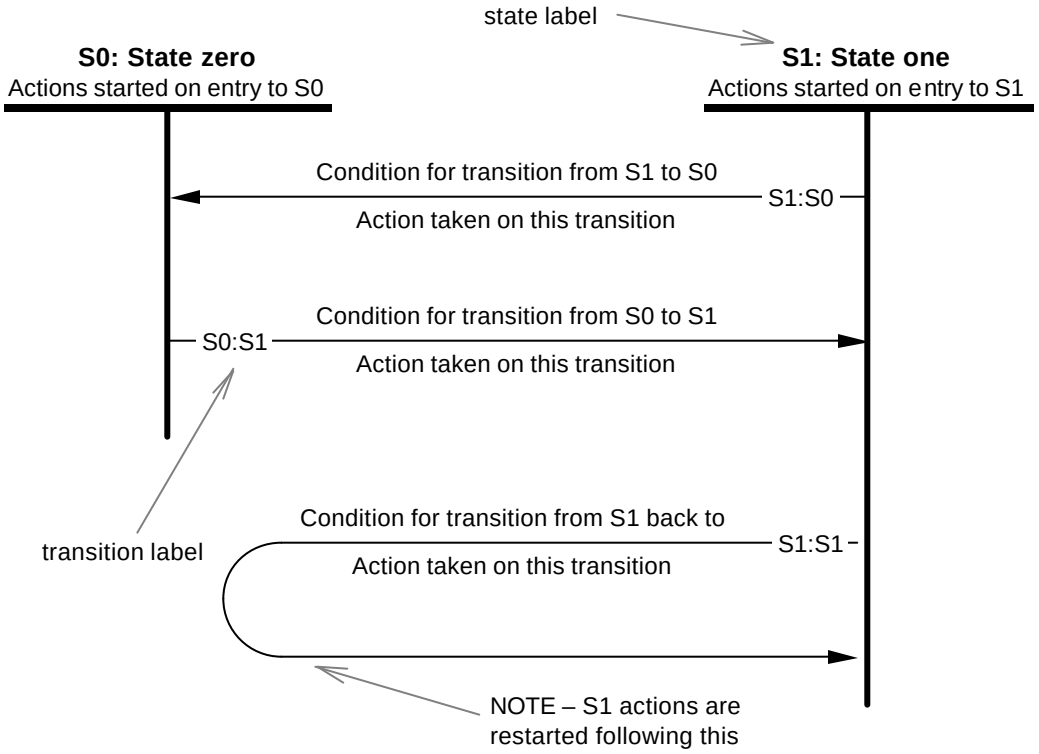

Figure 5 – State machine example

The state machines in this standard make three assumptions:

- Time elapses only within a discrete state;
- State transitions are conceptually instantaneous; the only actions taken during the transition are the setting of flags or variables and the sending of signals; and
- Each time a state is entered (or reentered from itself), the actions of that state are performed.

Multiple transitions may connect two states. In this case, the transitions are uniquely labeled by appending a character to the transition label, *e.g.*, S0:S1a and S0:S1b.

4 Model (informative)

4.1 Model overview

This clause is informative and describes typical components and operation of the SBP-3 model. It is intended to enhance the usefulness of the other, normative parts of this standard. In addition to the information in this clause, users of this standard should also be familiar with the CSR Architecture and Serial Bus standards.

Serial Bus Protocol 3 (SBP-3) is a transport protocol defined for IEEE 1394, High Performance Serial Bus. It defines facilities for requests (commands) originated by Serial Bus devices (initiators) to be communicated to other Serial Bus devices (targets) as well as the facilities required for the transfer of data or status associated with the commands. An SBP-3 device may assume the roles of initiator or target, either simultaneously or in succession. Commands and status may be transferred between the initiator and the target; data moves between the target and another device, which may be either the initiator or some other device.

4.2 Unit architecture

In CSR Architecture and Serial Bus terminology, targets implemented to this standard are units. A Serial Bus node that implements one or more targets has a configuration ROM unit directory for each that identifies the presence and capabilities of the target.

Each unit directory in configuration ROM permits initiators to detect the presence of a target during Serial Bus configuration, whether part of system initialization or subsequent to a Serial Bus reset. The node's 64-bit identifier, EUI-64, in combination with identifying characteristics of the unit directories themselves permits detected targets to be uniquely recognized despite changes in physical IDs that may occur as the result of Serial Bus resets.

4.3 Logical units

A logical unit is part of the unit architecture and is an instance of a device model, *e.g.*, disk, CD-ROM or printer. A logical unit consists of one or more device servers responsible to execute control or data transfer commands, one or more task sets that hold commands available for execution by the device servers and a logical unit number that is unique within the domain of the target.

Targets implement at least one logical unit, addressable as logical unit number (LUN) zero. Additional logical units may be implemented, addressable by their logical unit numbers. The logical units may implement different device models; for example, a single unit architecture might contain both a CD-ROM logical unit and an associated medium-changer logical unit. The logical units are visible to the initiator, either as described by configuration ROM or as discoverable by command set-dependent requests directed to the target.

NOTE – The structure of configuration ROM entries in the unit and logical unit directories permits considerable latitude to implementers in the description of targets. For example, a device which implements multiple functions or instances of a function may be described either by multiple unit directories, each with a single logical unit, or by one unit directory that includes multiple logical units—or any combination in between. Consult section 7, "Configuration ROM," for details.

4.4 Requests and responses

Target actions, such as a disk read that transfers data from device medium to system memory, are specified by means of requests created by the initiator and signaled to the target. The request is contained within a

data structure called an operation request block (ORB). The eventual completion status of a request is usually indicated by means of a status block stored by the target at an address provided by the initiator.

This standard defines several different formats for request blocks, whose principal uses are:

- to acquire or release target resources or to manage task sets (management requests—which include login requests); or
- to transport commands (command block requests).

Login and other management requests are directed to agents that can service only a single request at a time; there is no way to group these requests into a linked list. The ORBs for command block requests provide a field that contains the address of another ORB or a null pointer. This permits these requests to be in a linked list, as illustrated below.

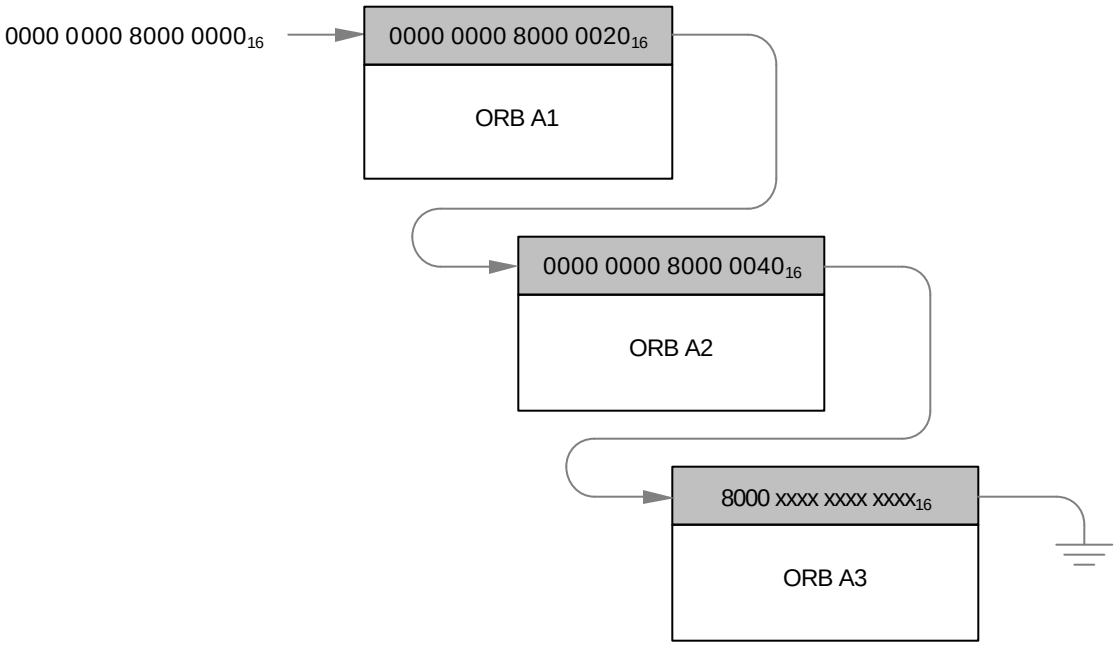

Figure 6 – Linked list of ORBs

Requests in a linked list are serviced by a fetch agent, which reads the requests from initiator memory when the initiator signals the availability of requests. The target may read ahead in the linked list; consequently the device server may reorder the execution of requests to improve performance.

When a request is completed successfully, the target usually stores a status block but if the request completes in error a status block is always stored.

4.5 Data buffers

The ORBs described in the preceding clause contain the device command and, for those commands which transfer data, the address of the data buffer for the command. The data buffer may be a single, contiguous buffer that is addressed directly by the ORB or it may be a collection of possibly disjoint segments that are addressed indirectly through a page table. The figures below illustrate both cases.

As an example, consider a command intended to transfer image data to a printer. If we assume that the image data is 3088_{16} bytes long, that the buffer starts at an address of 23 6174_{16} and that page boundaries

in the underlying system memory occur at 4096 byte intervals, the relationship between the ORB and the data buffer is as shown by Figure 7.

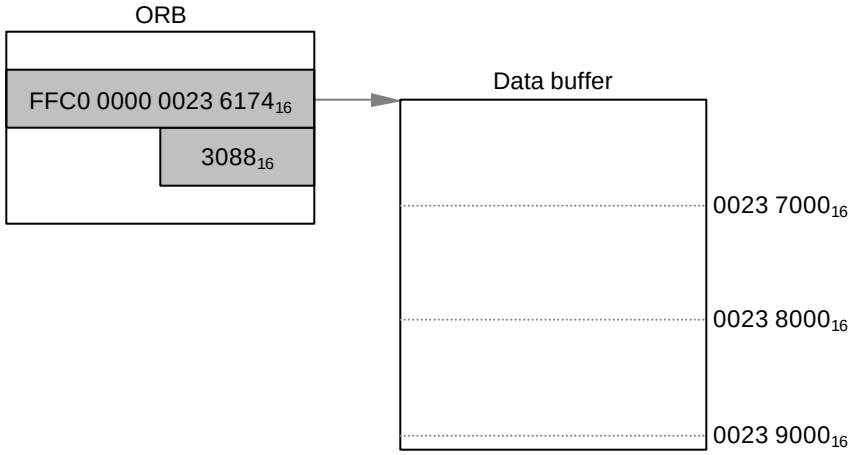

Figure 7 – Directly addressed data buffer

In the preceding example, two fields in the ORB specify the 64-bit address of the data buffer and its length, in bytes. The data buffer is shown with a node ID of FFC0$_{16}$, which is node zero on the local bus. The printer uses block read transactions to fetch data from the buffer before printing; the maximum size of the data payload for each request is controlled by a field in the ORB. The dotted lines within the data buffer indicate page boundaries. Although the data buffer is contiguous, the printer is not permitted to cross a page boundary in any one block read request.

When the data buffer consists of disjoint segments, it is necessary to indirectly address the data buffer through a page table, as shown in Figure 8. This figure could be an illustration of data read from a disk into various pages of an initiator's file system cache. In the example, assume that 2960$_{16}$ bytes of data are to be read from disk.

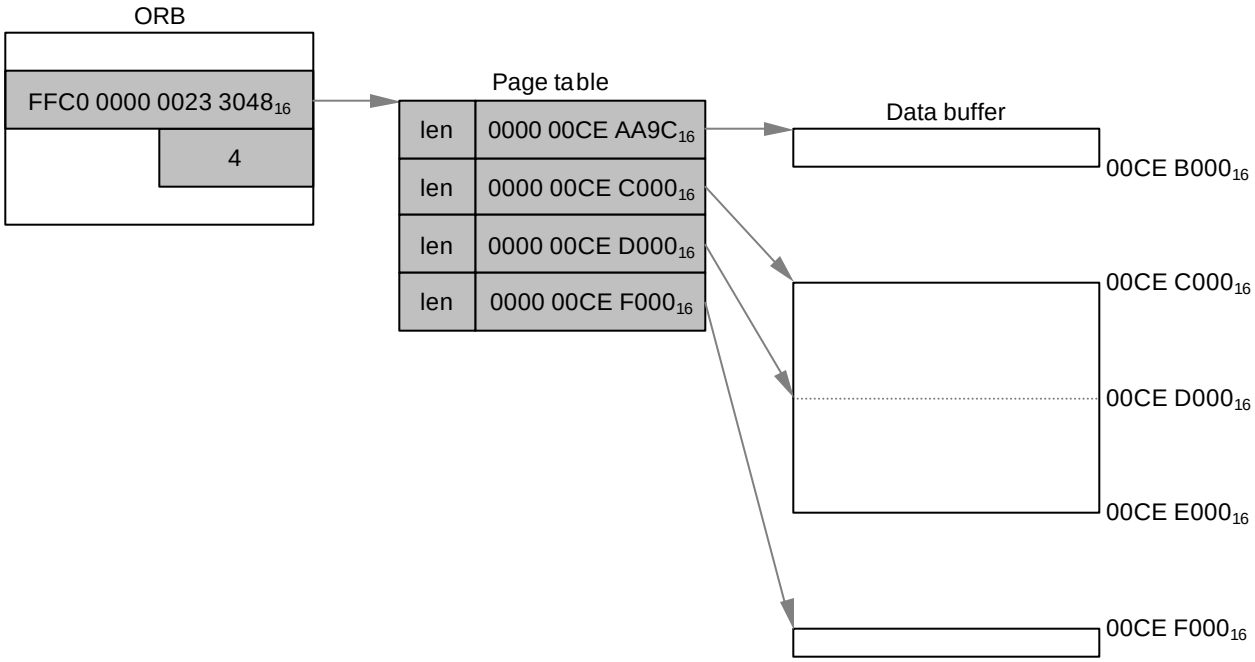

Figure 8 – Indirectly addressed data buffer (*via* page table)

The fields in the ORB that directly addressed a data buffer in the first example now point to a page table. Note that the ORB field that contains the data length when direct addressing is employed instead contains the number of elements in the page table—in this case, four. Each of the four page table elements points to the start of a segment of the data buffer. Each page table element also contains the length of the segment. The first segment ends on a page boundary, all other segments start on page boundaries (the middle segments also end on page boundaries) and the last segment may end on any boundary. In this example, the segment lengths are 0564_{16}, 1000_{16}, 1000_{16} and $03FC_{16}$, respectively.

When a page table is used, both the page table and the data buffer it describes usually reside in the same node. The node ID of the page table, $FFC0_{16}$, is not repeated in the page table elements. The space that would have otherwise been occupied by the node ID instead is used to contain the length of each segment.

Another variant of page table format is permitted, called an unrestricted page table (or a scatter/gather list). In an unrestricted page table, data buffer segments may start on any boundary and may have arbitrary lengths: there is no underlying page size.

4.6 Target agents

A target agent is a facility that receives signals from the initiator that indicate the availability of requests. There are two types of target agent, one that can execute a single request at a time and the other that can manage queues (linked lists) of requests, as illustrated by Figure 6. In the first case, the initiator signals the request to the agent by means of a Serial Bus block write request with the address of the request. In the other case, the initiator appends new requests to an active list, then rings a doorbell which causes the target agent to fetch the requests from system memory as target resources permit their execution.

Target agents that manage linked lists of requests utilize context maintained at both the initiator and target to fetch requests from memory. Once fetched, the request is locally available to the target for execution. The context consists of three elements:

- a linked list of ORBs at the initiator;
- a current ORB address at the target; and
- a doorbell at the target.

This standard defines procedures for both the initiator and the target that permits the addition of new requests to a linked list of ORBs while the target is actively fetching or executing previously queued requests. The procedures avoid the possibility of race conditions between the producer (initiator) and consumer (target) of the ORBs.

There are two types of target agents:

- management; and
- command block.

Management agents accept a variety of requests, such as login, create task set, task management, reconnect and logout. Before making other requests, an initiator first completes a login *via* the management agent. During the lifetime of a login, task management requests are directed to either the primary task set (created by the login) or to separately created secondary task sets associated with the login. Ultimately, management agents accept logout requests; these indicate the initiator's intent to release target resources previously acquired by a login or create task set request. Management agents service a single request at a time and do not support linked lists.

A successful login or create task set request returns the address of a command block agent that services requests which are organized in linked lists. Individual linked lists are managed by ~~a~~ separate command block agents.

4.7 Ordered and unordered execution

Targets may implement either an ordered or unordered model of task execution. The ordered model is usually appropriate for devices where the context of a command affects its execution, *i.e.*, the outcome of one command affects the subsequent command. A common example of a device with such command dependencies is a tape drive. The unordered model is usually appropriate for direct-access devices for which no positional or other context information is inherited from one command to the next.

The ordered model specifies both that tasks are executed in order and that completion status is returned in the same order. A consequence of ordering is that completion status for one task implicitly indicates successful completion status for all tasks that preceded it in the ordered list.

The unordered model permits the target to reorder active tasks without restriction. The actual execution sequence of tasks from any task set may bear no relationship to the order in which they were fetched. Unrestricted reordering leaves the responsibility for the assurance of data integrity with the initiator. If the integrity of data on the device medium could be compromised by unrestricted reordering involving a set of active tasks, $\{T_0, T_1, T_2, \ldots T_N\}$ and a new task T', the initiator shall wait until $\{T_0, T_1, T_2, \ldots T_N\}$ have completed before appending T' to an active request list.

NOTE – In multitasking operating system environments, independent execution threads may generate tasks that have ordering constraints within each thread but not with respect to other threads. If this is the case, an initiator may manage the constraints of each thread yet still keep the target substantially busy. This avoids the undesirable latencies that occur if the target is allowed to become idle before new ORBs are signaled.

4.8 Bridge-awareness

Targets designed to operate with initiators or data buffers on remote buses (*i.e.*, not the local bus but buses accessible *via* one or more intervening bridges) are described as “bridge-aware”. In general terms, this means that their designs embody an understanding of the requirements of draft standard IEEE P1394.1. More specifically, the salient features of bridge-aware targets are:

- The ability to distinguish between local node IDs, whose scope is restricted to the local bus, and global node IDs that reference a remote node (or indirectly reference a local node). The most significant ten bits of a node ID differentiate local and global node IDs;
- Separate split transaction time-out values for requests addressed to local nodes and those addressed to remote nodes. The remote time-out value is significantly longer than the local bus split time-out;
- The ability to generate commands addressed to remote bridges. These commands are identified both by the data payload of the packet and its destination CSR; their intended recipient (or recipients) are identified by the packet’s destination address and the value of the *snarf* field in the packet header of block write requests;
- Comprehension of new response packet header fields, such as *proxy_ID* and *ext_rcode*, and new response codes;
- Implementation of the MESSAGE_REQUEST, MESSAGE_RESPONSE and QUARANTINE registers;
- Particular behaviors in response to bus reset, notably self-quarantine of remote subactions and the possible invalidation of any global node IDs cached by the device;
- Recognition of commands originated by bridge portals and intended for bridge-aware nodes.

Most of the requirements above are best understood by reference to draft standard IEEE P1394.1 itself. However, there are other changes in Serial Bus Protocol necessitated by some of these new behaviors. In particular, *a)* initiators and targets require a stable method to identify nodes that contain data buffers and *b)* initiators and targets may no longer use bus reset as a mutual synchronization point since they do not observe bus reset on the other's bus.

The obvious candidate for stable reference to a node is its 64-bit unique ID, EUI-64. Unfortunately, legacy SBP-2 data structures are restricted to a 16-bit field to identify a node. The solution is to differentiate between two types of information that may be contained within the 16-bit field. One type is the local node ID documented by SBP-2. The second is a *node handle*, an arbitrary value assigned by the target to represent a particular EUI-64. Because draft standard IEEE P1394.1 restricts the most significant ten bits of a local node ID to all ones, this standard is free to define a node handle to be any 16-bit value whose most significant ten bits are other than all ones.

NOTE – Although a node handle and a global node ID are similar in that their most significant ten bits are not all ones, they are not the same thing. A global node ID, when used in the *destination_ID* field of a Serial Bus request subaction, causes the subaction to be routed by bridges to the intended recipient. A node handle should never be used in *destination_ID*; its value might coincidentally be equal to a valid global node ID—but one that corresponds to a different EUI-64 than the target had previously associated with the node handle.

Before an initiator may use a node handle to refer to a particular node, it asks the target for a node handle that corresponds to the node's EUI-64. Once the target has returned a node handle to the initiator, the node handle may be used in any address pointer to reference the node identified by the EUI-64.

When a target encounters a node handle in any address pointer field, it decodes the reference into a global node ID which may be used to address the desired node. The target is responsible to maintain a valid correlation between a node handle and its associated EUI-64 and global node ID. Device discovery methods (as described by draft standard IEEE P1394.1) are used to discover (or rediscover) the global node ID that

matches the EUI-64 associated with the node handle. If the desired node is no longer connected to the net, the target terminates affected tasks.

The other protocol change necessitated by bridges concerns bus reset. In general, bus resets in a network of interconnected buses are a local event not propagated by bridges.[2] If a bus reset occurs on the target's bus and the initiator is on a remote bus, the event will pass unobserved by the initiator. As a consequence, a protocol behavior useful in legacy SBP-2 becomes detrimental when bridges are present: a bridge-aware target cannot afford to abort its task sets in response to bus reset. The remote initiator is unable to detect such an event; even if the target attempts to notify the initiator, it is uncertain whether work at the target can make forward progress in the face of subsequent bus resets.

The use of node handles is essential to surmount the bus reset problem. When a target operating in bridge-aware mode observes a bus reset, it does not abort its task sets. Instead it revalidates the global node IDs in use and insures that they continue to correspond to nodes originally specified by EUI-64. The initiator need not know about this action by the target, since the initiator continues to use stable node handles to identify nodes.

NOTE – Although this new protocol feature was designed as a response to bridges, it may be very useful in the context of the local bus. A node handle may refer to a local node; a target operating in bridge-aware mode is capable of determining the corresponding local node ID after bus reset. Without task set aborts forced by bus reset, target operations may be significantly more efficient.

4.9 Streams

Streams are objects that are based upon the isochronous capabilities of Serial Bus. A stream consists of all of the target and logical unit functions and resources that are necessary to transfer isochronous data from one or more Serial Bus channels to the device's medium (the target is a listener) or to transfer data isochronously from the device's medium to one or more Serial Bus channels (the target is a talker). The direction, listener or talker, of any stream is independent of any other stream. Within each stream all of the data flows in the same direction.

Streams require Serial Bus resources as well as target resources. These include the aggregate bandwidth necessary for the stream, the channel numbers utilized by the stream and the isochronous connections that characterize the stream. An application (usually, but not necessarily, co-located with the initiator) allocates all necessary resources before activating a logical unit isochronous stream.

A stream of isochronous data appears on Serial Bus as ~~subactions~~ a subaction with a transaction code (*tcode*) of A_{16} during an isochronous period. This in turn is represented by an ordered byte stream of data on the device medium. The presentation of this data is controlled by command block ORBs that request data transfer to or from the medium.

Figure 9 illustrates the relationship between the different stream components during playback (the logical unit is assumed to have direct-access capabilities).

[2] Net topology changes anywhere within the net eventually cause a bus reset to occur on each bus within the net, but these bus resets are not synchronized with each other and do not carry any useful information except notification that net topology has changed.

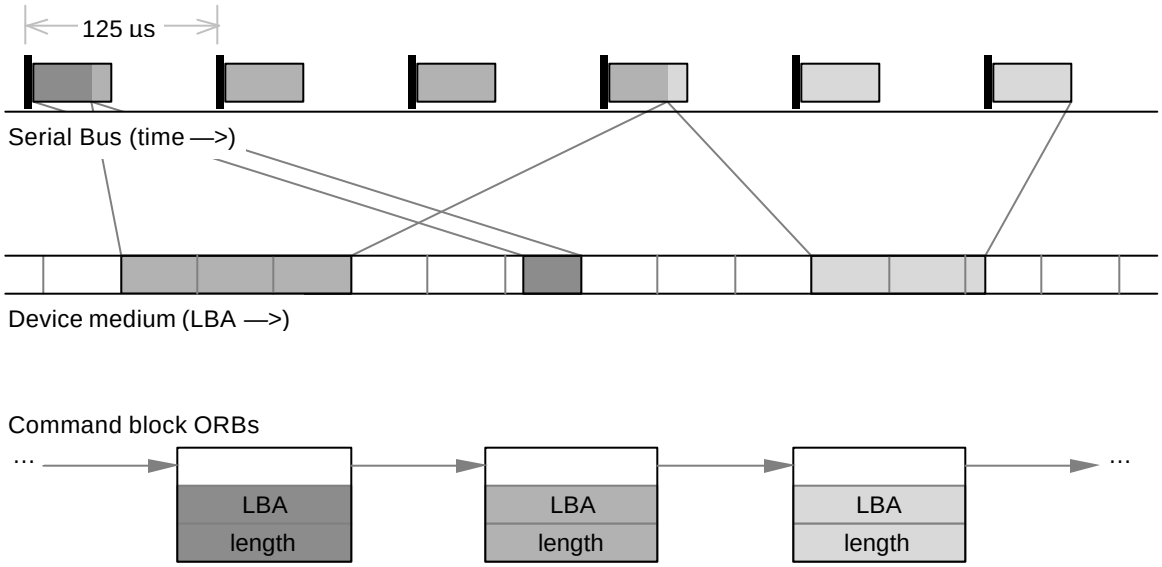

Figure 9 – Components of an isochronous stream (direct-access logical unit)

The figure shows that the order of presentation of bytes in a stream is determined by the order of command block ORBs—but that this order is independent of the location of the data on device medium. In this example, the size of the isochronous packet transmitted each cycle is determined by information previously recorded on the medium. The example shows a stream with only one channel (one packet per isochronous period) and a fixed packet size, but streams may consist of more than one channel and the packet size may be different each isochronous period.

Streams differ fundamentally from asynchronous data transfers. First, streams do not require any address context for the transfer of data to or from system memory: a channel number and the time-ordered location of the data within the stream identify the data. Second, the stream's flow of isochronous data may be controlled and synchronized to time or other time-dependent events.

In order to fully exploit these differences, a logical unit's command set should be cognizant of Serial Bus isochronous behavior. Such a command set may enable a logical unit to perform frame-precise synchronization or to manage multi-channel streams, to give just two examples. However, it is possible to adapt existing command sets (ones designed without awareness of Serial Bus isochronous facilities) to use isochronous data transfer methods.

In order to leverage existing command sets, simplifying assumptions are required. The devices are limited to single-channel streams, either input or output but not both the same time. The essential information that characterizes ~~this~~ data flow on a single isochronous channel is a) bandwidth, b) channel number and c) transmission speed. IEC 61883-1 [B5] defines plug control registers (PCRs) that provide this information. Output plug control registers (oPCRs) specify speed, channel number and maximum data payload per isochronous subaction (bandwidth) while input control registers (iPCRs) specify only channel number. The plug control registers also enable connection management and a simple on or off scheme to control the flow of isochronous data.

The information available in a plug control register is combined with data transfer length information in a command block ORB when the *isochronous* bit in the ORB is set to one; this causes the logical unit to use isochronous subactions to transfer data to or from the device medium in accordance with the parameters in the plug control register.

One example of a simple, single data stream device that might benefit from isochronous data transfer would be an isochronous DVD player. Contemporary DVD players use a multi-media command set (MMC [B4] [B13]) that assumes a transport protocol that provides confirmed, asynchronous data transfer. This is workable when the rendering device (typically a computer) and the DVD player are the only devices that share the transport medium: ample bandwidth is available. But if the DVD player is an SBP device and is in competition with many other devices for Serial Bus bandwidth, the best-effort nature of asynchronous data transfer on Serial Bus might result in late delivery of data to the rendering device. A practical solution is to leverage the existing command set, unchanged, and use the isochronous facilities of SBP-3 to implement an isochronous DVD player. If isochronous bandwidth is reserved in advance, the rendering device should receive all of the data transmitted by the DVD player in time for display.

The steps an initiator (which is also co-located with the rendering application) would take to control an isochronous DVD player are as follows:

a) Read the DVD player's plug control registers to determine the maximum speed and maximum data payload supported by the DVD player and attempt to allocate the required bandwidth;

b) If the bandwidth allocation is successful, attempt to allocate a channel number. If this fails, release the previously allocated bandwidth and try again later;

c) Otherwise, program the DVD player's output plug control register (oPCR) with the speed, channel number and maximum data payload information and at the same time establish a point-to-point connection in accordance with the procedures specified by IEC 61883-1;

d) Program Serial Bus adapter hardware (co-located with the initiator) to receive isochronous data on the specified channel;

e) Signal command block ORBs to the DVD player to request that it transmit data from its device medium as a sequence of isochronous subactions;

f) When complete, program the DVD player’s oPCR to relinquish the point-to-point connection in accordance with the procedures specified by IEC 61883-1 and then release the previously allocated bandwidth and channel number.

An initiator would utilize a similar procedure to control isochronous data transfer for logical unit that is a listener, for example an isochronous scanner.

5 Data structures

5.1 Data structure types and components

There are three classes of data structures defined by this standard:

- operation request blocks (ORBs);
- page tables;
- status blocks.

These data structures are allocated and initialized by an initiator in system memory at Serial Bus nodes. ORBs and status blocks shall be allocated at the initiator's node. Unless the logical unit supports node selectors (see 5.3.4), page tables shall be allocated at the same node as the data buffer to which they refer.

All data structures defined by this standard shall be aligned on quadlet boundaries. These alignment requirements permit 64-bit address pointers that reference these data structures to conform to the format specified below.

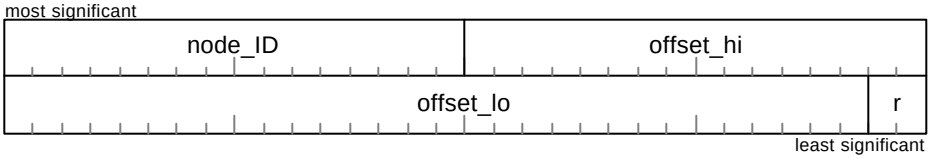

Figure 10 – Address pointer

The *node_ID* field shall identify the Serial Bus node for which the address pointer is valid, as defined by IEEE 1394 or this standard. In many cases, additional constraints on the location of data structures render the information in *node_ID* redundant. In these cases, *node_ID* is considered a reserved field or is explicitly redefined for other uses. Except when *node_ID* is redefined or reserved, it shall contain either a local node ID, as specified by IEEE 1394, or a node handle supplied by a target, as specified by this standard.

The *offset_hi* and the *offset_lo* fields shall together specify the most significant 46 bits of the Serial Bus offset and shall be combined with two low-order bits of zero to derive the 48-bit Serial Bus offset. Although the two least significant bits of a 48-bit address pointer are reserved (and therefore zeroed by the initiator), the target shall not assume that they are zero.

The size of a data structure addressed by a pointer that conforms to Figure 10 is either explicitly specified by an associated length field or implicitly known from context. Whichever the case, a target shall not initiate any Serial Bus request subactions (read, write or lock) that reference system memory outside of the range determined by an address pointer and length supplied by an initiator.

ORBs shall be allocated at the initiator's node. Some types of ORBs contain an address pointer which permits them to be organized as a linked list. Since the node ID is known for all ORBs in such a list, the address pointer format is redefined to reuse the *node_ID* field. An address pointer that references an ORB shall conform to the format below.

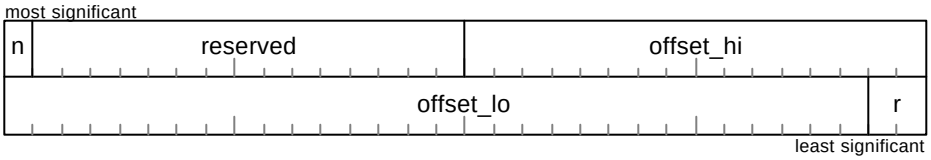

Figure 11 – ORB pointer

The *null* bit (abbreviated as *n* in the figure above) indicates a null pointer when it is one. In this case the target shall ignore the *offset_hi* and the *offset_lo* fields.

5.2 Operation request blocks (ORBs)

5.2.1 Generic ORB

All initiator requests for target actions are expressed within ORBs, which may either be fetched by the target *via* Serial Bus read transactions or directly signaled to the target by Serial Bus write transactions. ORB formats vary according to use and may be viewed in hierarchical relationship to each other, as illustrated below.

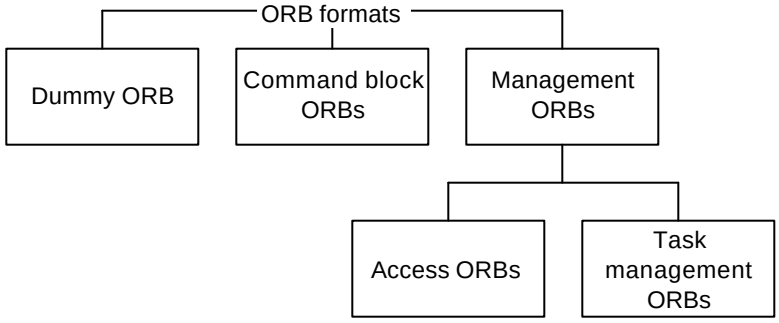

Figure 12 – ORB family tree

The formats of the ORBs are described in the clauses that follow. This clause specifies fields that are common to all ORBs, illustrated in the figure below.

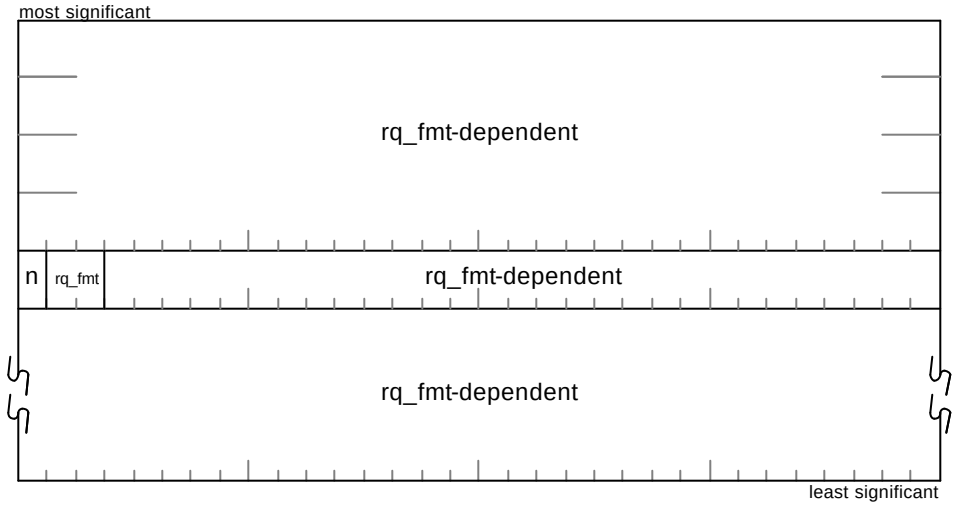

Figure 13 – ORB format

The *notify* bit (abbreviated as *n* in the figure above) advises the target whether or not completion notification is required. When *notify* is zero, the target may elect to suppress completion notification except when there is an error, in which case the value of *notify* is ignored and a status block shall be stored. If *notify* is one, the target shall always store a status block in initiator memory. When the target stores a status block, it shall store it at the *status_FIFO* address specified in the ORB or, if not specified in the ORB, at the address supplied in the login or create task set request.

The *rq_fmt* field specifies ORB format, as defined by the table below.

Value	ORB format
0	Format specified by this standard
1	Format specified by this standard
2	Vendor-dependent
3	Dummy (NOP) request format

The format of an ORB is uniquely determined by a combination of *rq_fmt*, the command set implemented by the target and the target agent to which the ORB is signaled. This standard specifies those parts of the ORB that are invariant across target command sets and device types.

5.2.2 Dummy ORB

Dummy ORBs may be used as placeholders within linked lists of requests. An example is the use of a dummy ORB in the initialization of a logical unit fetch agent (see 9.2.2). The initiator shall allocate system memory large enough to contain the ORB size specified by the logical unit. The format of a dummy ORB is illustrated below.

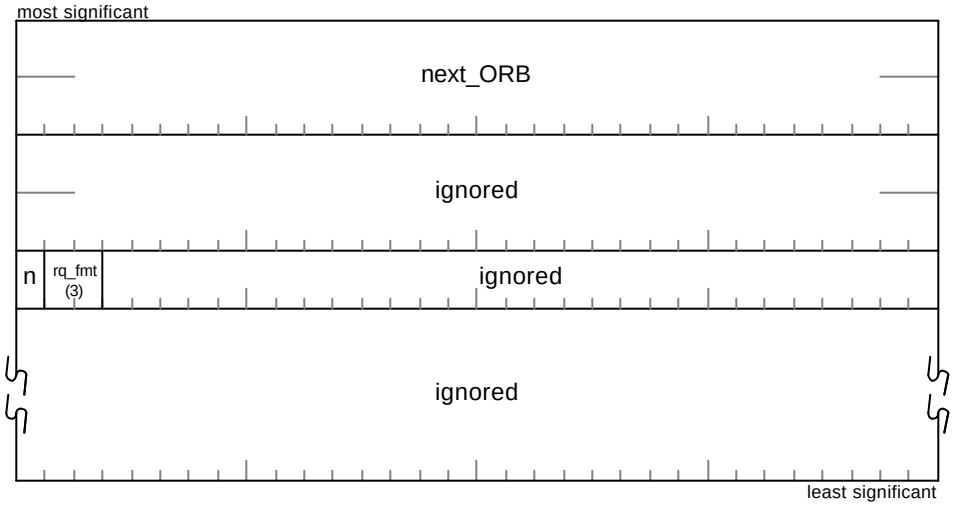

Figure 14 – Dummy ORB

The *next_ORB* field shall contain a null pointer or the address of an ORB and shall conform to the address pointer format illustrated by Figure 11.

The *notify* bit is as previously defined for all ORB formats.

The *rq_fmt* field is as previously defined for all ORB formats and shall be three.

An *rq_fmt* value of three is also used to indicate an ABORT TASK request to a target. See 10.5.1 for details of ORB processing by the target and for permissible completion status values.

5.2.3 Command block ORBs

Command block ORBs are used to encapsulate data transfer or device control commands for transport to the target. A target's command set and device type determine the length of these ORBs, which shall be fixed for a particular command set and device type. A target reports this size in its configuration ROM (see 7.8.10).

NOTE – Although device designers may select arbitrary ORB lengths, system considerations may favor some ORB sizes over others, *e.g.*, 32 bytes. An ORB size of 32 bytes limits the command set-dependent information in a command block ORB to twelve bytes. This is adequate for many command descriptor blocks defined in command sets such as SCSI, but device designers should not hesitate to utilize larger ORBs if 16-byte or larger commands are required. Operating systems designers should take care not to preclude the use of arbitrarily large ORBs.

Command block ORBs may have either one or two buffer descriptors. The format of a command block ORB with a single buffer descriptor is illustrated by the figure below.

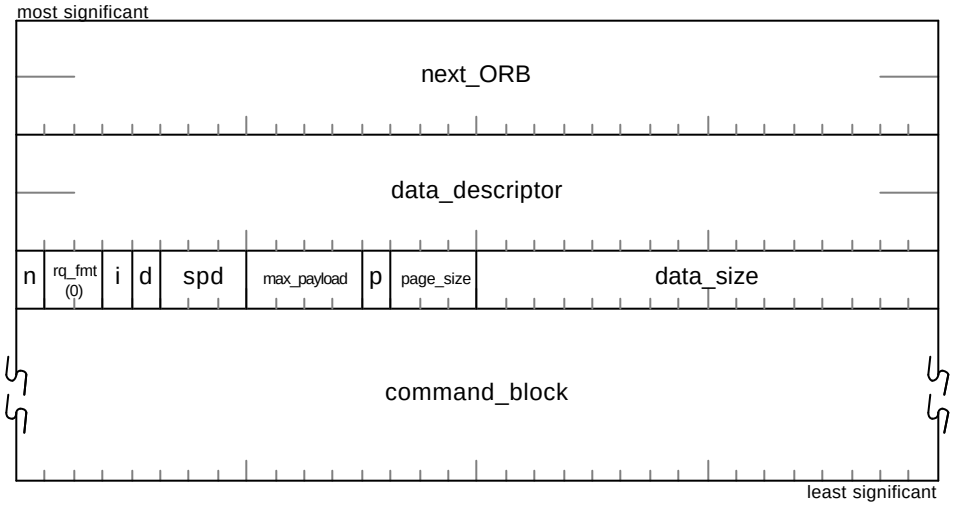

Figure 15 – Command block ORB (single buffer descriptor)

The *next_ORB* field shall contain a null pointer or the address of a dummy ORB or a command block ORB and shall conform to the address pointer format illustrated by Figure 11.

The value of the *data_descriptor* field is valid only when the *isochronous* bit is zero and *data_size* is nonzero, in which case it shall contain either the address of the data buffer or the address of a page table that describes the memory segments that make up the data buffer, dependent upon the value of *page_table_present* bit. The format of the *data_descriptor* field, when it directly addresses a data buffer, shall be a 64-bit Serial Bus address or, when it addresses a page table, shall be as specified by Figure 10. When *data_descriptor* specifies the address of a page table, the format of the page table shall conform to that described in 5.3.

The *notify* bit and *rq_fmt* field are as previously defined for all ORB formats. The *rq_fmt* field shall be zero for an ORB which contains a single buffer descriptor.

The value of the *isochronous* bit (abbreviated as *i* in the figure above) is valid only when *data_size* is nonzero, in which case this it specifies the transfer method used for data associated with the ORB. When this bit is zero, Serial Bus read or write transactions shall be used to move data to or from the buffer described by the *data_descriptor*, *spd*, *max_payload*, *page_table_present*, *page_size* and *data size* fields. Otherwise, when *isochronous* is one, data transfer shall be effected by Serial Bus isochronous streams, *i.e.*, packets with a transaction code of A_{16}.

NOTE – Command set-dependent methods may be used to specify isochronous data transfer even if the *isochronous* bit is zero. See Annex D for an example.

The value of the *direction* bit (abbreviated as *d* in the figure above) is valid only when *data_size* is nonzero, in which case it specifies direction of data transfer. The meaning of the *direction* bit shall be interpreted in conjunction with the *isochronous* bit. If both the *isochronous* and the *direction* bits are zero, the target shall use Serial Bus read transactions to fetch data from system memory. When the *isochronous* bit is zero and the *direction* bit is one, the target shall use Serial Bus write transactions to store data in system memory. Otherwise, when the *isochronous* bit is one, a *direction* bit of zero specifies that the target shall receive isochronous data while a *direction* bit of one specifies that it shall transmit isochronous data.

The value of the *spd* field is valid only when the *isochronous* bit is zero and *data_size* is nonzero, in which case it specifies the speed that the target shall use for data transfer transactions addressed to the data buffer or page table, as encoded by Table 1.

Table 1 – Data transfer speeds

Value	Speed
0	S100
1	S200
2	S400
3	S800
4	S1600
5	S3200
6 – 7	Reserved for future standardization

The value of the *max_payload* field is valid only when the *isochronous* bit is zero and *data_size* is nonzero, in which case the maximum data transfer length is specified as $2^{max\_payload + 2}$ bytes, which is the largest data transfer length that may be requested by the target in a single Serial Bus read or write transaction addressed to the data buffer. The *max_payload* field shall specify a length less than or equal to the maximum asynchronous data payload specified by IEEE 1394 for the data transfer rate indicated by *spd*.

The value of the *page_table_present* bit (abbreviated as *p* in the figure above) is valid only when the *isochronous* bit is zero and *data_size* is nonzero, in which case it shall be zero if *data_descriptor* directly addresses the data buffer, else one when *data_descriptor* addresses a page table.

The value of the *page_size* field is valid only when the *isochronous* bit is zero and *data_size* is nonzero, in which case it shall specify the underlying page size of the data buffer memory (see 9.4 for an explanation of target responsibilities with respect to page boundaries). A *page_size* value of zero indicates that the underlying page size is not specified. Otherwise the page size is $2^{page\_size + 8}$ bytes. The page size applies to the data buffer whether or not a page table is present. When a page table is used to describe the data buffer, the *page_size* field also specifies the page table format. A *page_size* value of zero indicates an unrestricted page table (also known as a scatter/gather list) while a nonzero *page_size* indicates a normalized page table.

If the *isochronous* bit is zero and *page_table_present* is zero, the *data_size* field shall contain the size, in bytes, of the system memory addressed by the *data_descriptor* field. When the *isochronous* bit is zero and *page_table_present* is one, *data_size* shall contain the number of elements in the page table addressed by *data_descriptor*. Otherwise, the *isochronous* bit is one and *data_size* shall specify the maximum count, in bytes, of isochronous data to be transferred.

The *command_block* field contains information not specified by this standard.

When *rq_fmt* equals one, the ORB contains two buffer descriptors, as illustrated by Figure 16.

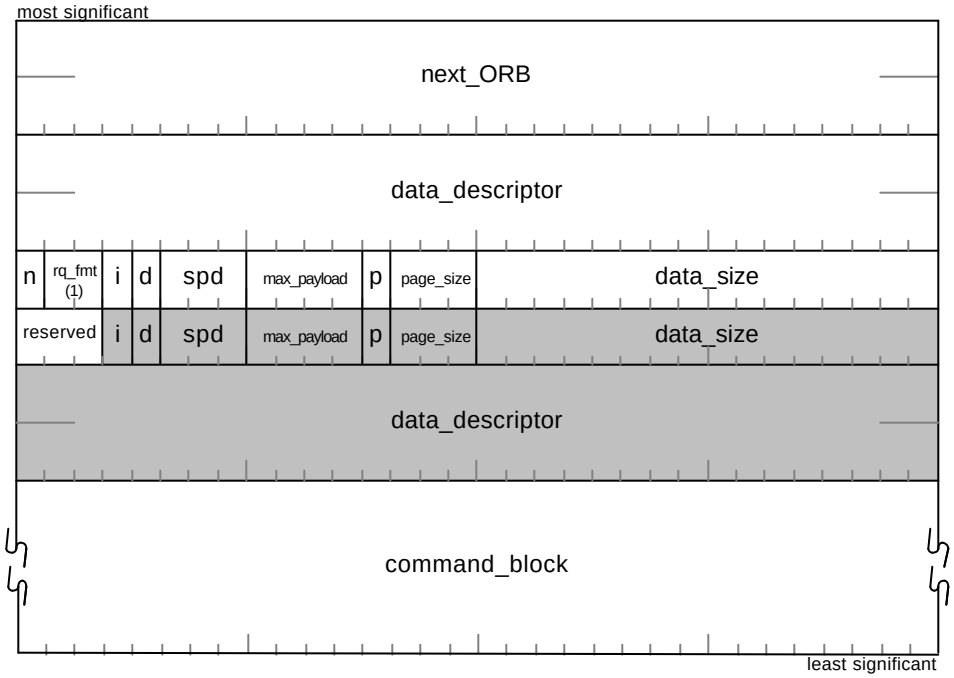

Figure 16 – Command block ORB (dual buffer descriptor)

Each buffer associated with an ORB is described by a set of bits and fields: *data_descriptor*, *isochronous*, *direction*, *spd*, *max_payload*, *page_table_present*, *page_size* and *data_size*. A command block ORB whose *rq_fmt* is zero describes a single buffer or isochronous stream (referred to as *buffer*[0]) via the fields specified by Figure 15. When *rq_fmt* is one, the ORB includes two sets of these fields, capable of describing *buffer*[0] and *buffer*[1]. The fields that describe *buffer*[0] are in the same location as in a single buffer descriptor command block ORB; the additional fields (shown shaded in Figure 16) describe *buffer*[1]. The meaning of the individual buffer descriptor fields remains the same whether the field pertains to *buffer*[0] or *buffer*[1].

All of a buffer's characteristics are independent of the other buffer. Buffers need not reside in the same node nor be subject to the same speed or maximum payload characteristics. One buffer may be described by a page table and the other not. The matrix below illustrates how it is possible in all except two cases to determine buffer use from the information contained in the ORB. The two cases that require additional information (shown shaded in gray) occur when two buffers are described and the *direction* bit for both has the same value.

			***data_size*[1]**		
			zero	nonzero	
				***direction*[1]**	
				0	1
***data_size*[0]** zero			No buffers	*buffer*[1] outbound	*buffer*[1] inbound
***data_size*[0]** nonzero	***direction*[0]**	0	*buffer*[0] outbound	Both buffers outbound; Consult command set for details	*buffer*[0] outbound *buffer*[1] inbound
		1	*buffer*[0] inbound	*buffer*[0] inbound *buffer*[1] outbound	Both buffers inbound; Consult command set for details

5.2.4 Management ORBs

5.2.4.1 Generic management ORB

Management ORBs are 32-byte data structures that encapsulate several types of management request:

- access requests (which include login and logout requests); and
- task management requests.

Unlike the command block ORBs (which are implicitly associated with a particular task set by virtue of the fetch agent to which they are addressed), management ORBs explicitly declare either the logical unit or the task set for which they are intended.

Management ORBs have the general format illustrated below. Note that since they lack a *next_ORB* field, they cannot be linked together to form a list.

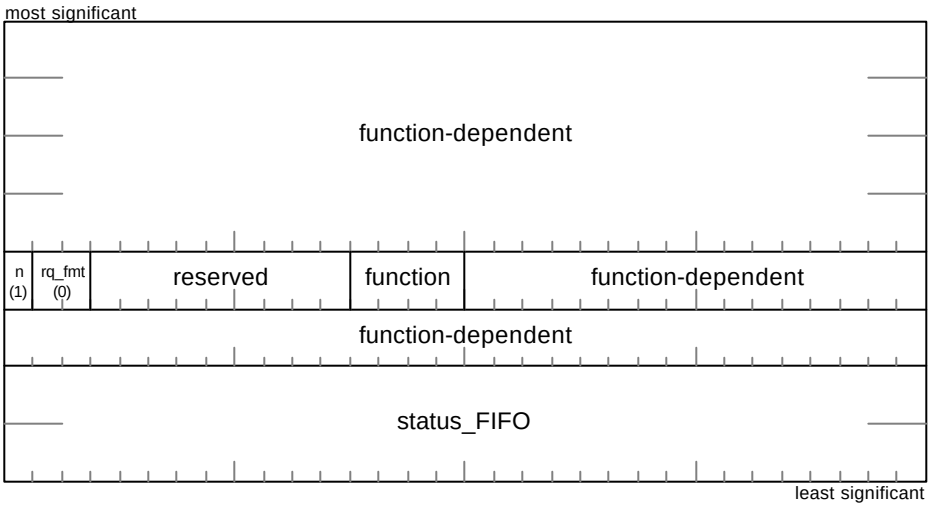

Figure 17 – Management ORB

The *notify* bit and *rq_fmt* field are as previously defined for all ORB formats. The *rq_fmt* field shall be zero and the *notify* bit shall be one.

The *function* field specifies the management function requested, as defined by Table 2. Target support for some management functions is mandatory. When a target receives a management request that specifies an optional, not supported function, it shall respond with an *sbp_status* of function rejected.

Table 2 – Management request functions

Value	Mandatory	Management function
0	Yes	LOGIN
1	Yes	QUERY LOGINS
2		CREATE TASK SET
3	Yes	RECONNECT
4		SET PASSWORD (see Annex C)
5		NODE HANDLE
6		Reserved for future standardization
7	Yes	LOGOUT
$8 – A_{16}$		Reserved for future standardization
B_{16}		ABORT TASK
C_{16}	Yes	ABORT TASK SET
D_{16}		Reserved for future standardization
E_{16}	Yes	LOGICAL UNIT RESET
F_{16}	Yes	TARGET RESET

The *status_FIFO* field shall contain an address allocated for the return of status information generated by the management request. The *status_FIFO* field shall conform to the format for address pointers specified by Figure 10 and shall address the same node as the initiator; consequently the *node_ID* field of this address pointer is reserved.

NOTE – The *status_FIFO* address explicitly specified within a management ORB may differ from the status FIFO address implicitly associated with command block requests. The address for command block requests is established by a LOGIN or CREATE TASK SET request and is not altered by other management requests.

5.2.4.2 Login ORB

Before any requests that address logical unit fetch agent CSRs or require a *login_ID* can be made of a target, the initiator shall first complete a login procedure that uses the ORB format shown below.

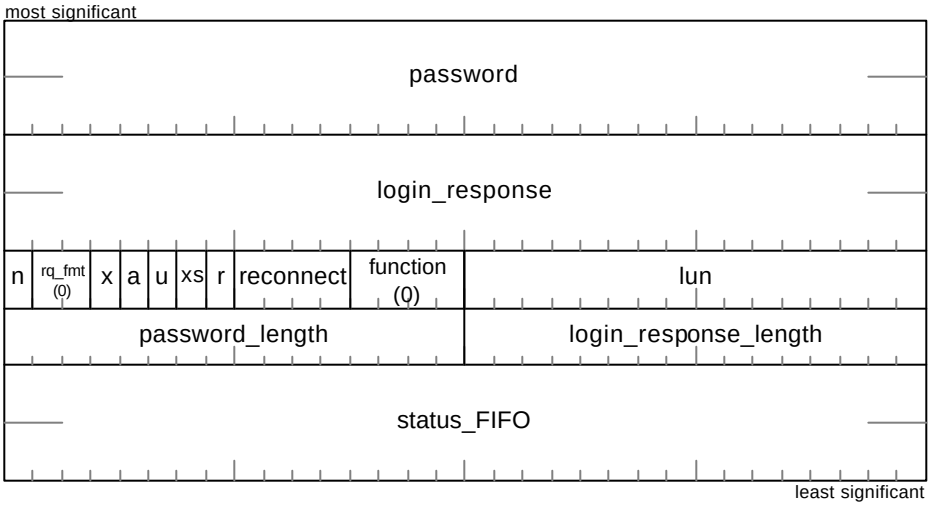

Figure 18 – Login ORB

The *password* and *password_length* fields may contain optional information used to validate the login request. If *password_length* is zero, the *password* field may contain immediate data. When *password_length* is nonzero, the *password* field shall conform to the format for address pointers specified by Figure 10 and shall contain the address of a buffer in the same node as the initiator; consequently the *node_ID* field of this address pointer is reserved. The buffer shall be accessible to a Serial Bus block read request with a data transfer length less than or equal to *password_length*. The format and usage of password data, whether immediate or indirectly addressed, are specified by Annex C.

The *login_response* and *login_response_length* fields specify the address and size of a buffer allocated for the return of the login response. The *login_response* field shall conform to the format for address pointers specified by Figure 10 and shall address the same node as the initiator; consequently the *node_ID* field of this address pointer is reserved. The buffer shall be accessible to a Serial Bus block write request with a data transfer length less than or equal to *login_response_length*. If the *aware* bit is zero, the initiator shall set *login_response_length* to a value of at least 12, otherwise to at least 16; the target may ignore this field if it stores no more than 12 bytes of login response data.

The *notify* bit and the *rq_fmt* field are as previously defined for management ORB formats.

The *exclusive* bit (abbreviated as *x* in the figure above) shall specify target behavior with respect to concurrent login to a logical unit. When *exclusive* is zero, the target, subject to its own implementation capabilities, may permit more than one initiator to login to a logical unit. If *exclusive* is one the target shall permit only one login to a logical unit at a time; see 8.3 for a description of target behavior.

The *aware* bit (abbreviated as *a* in the figure above) permits the initiator to request bridge-aware behavior (as specified by 8.4 and 10.6) from the target for operations associated with this login. When the *aware* bit is zero, target behavior with respect to bus reset and node IDs shall be compatible with SBP-2 ~~(see [B1])~~. Otherwise, the target is requested to behave in a bridge-aware manner as specified by this standard and draft standard IEEE P1394.1.

The *extended_status* bit (abbreviated as *xs* in the figure above) shall specify the status block format accepted by the initiator for this login. When *extended_status* is zero, only the basic status block format may be stored at the *status_FIFO*. Otherwise, either the basic or extended status formats may be stored (see 5.4 for details).

The *reconnect* field shall specify the desired reconnect time-out as $2^{reconnect}$ seconds. The default reconnect time-out, when *reconnect* is zero, is one second. The target ~~may~~ might not be able to support the requested value; see *reconnect_hold* in the login response data below.

The *update* bit (abbreviated as *u* in the figure above), when zero, specifies that the initiator is not logged in to the logical unit identified by *lun*. Otherwise, an existing login is to be updated with the information in the login request.

When the *update* bit is zero, the *lun* field specifies the logical unit number (LUN) to which the request is addressed. Otherwise, the *lun* field shall contain a *login ID* value obtained as the result of a successful login.

The *status_FIFO* field is as previously defined for management ORB formats and shall contain an address allocated for the return of status for the LOGIN request, status for all subsequent requests signaled to the *command_block_agent* allocated for this login and any unsolicited status generated by the logical unit.

If the login fails the contents of the response buffer are unspecified. Otherwise, upon successful completion of a login, the target shall store a minimum of 12 bytes of login response data and may store up to the entire 16 bytes illustrated below; the amount of data stored shall be an integral number of quadlets. Truncated login response data shall be interpreted as if the omitted fields had been stored as zeros.

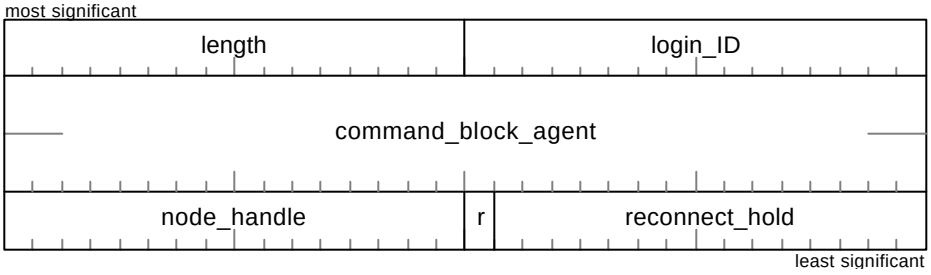

Figure 19 – Login response

The *length* field shall contain the length, in bytes, of the login response data.

The initiator shall use the *login_ID* value returned by the target to identify all subsequent requests directed to the target's management agent that pertain to this login.

The *command_block_agent* field specifies the base address of the agent's CSRs, which are defined in 6.6. This field shall conform to the format for address pointers specified by Figure 10.

When the *aware* bit in the login request is zero, the contents of the *node_handle* field are unspecified. Otherwise the node handle field shall contain a node handle that the initiator shall use as the most significant 16 bits of any address pointer (whose *node_ID* field is not reserved) that references initiator memory in any ORB which is signaled to the target in the context of this login. ~~This requirement applies equally to all address pointers with the exception of those contained in the login request itself.~~

The *reconnect_hold* field shall specify the time, in seconds less one, that the target will hold resources for a previously logged-in initiator subsequent to a bus reset or net update. The value of *reconnect_hold* shall not be greater than $2^{reconnect}$ -1, where *reconnect* is obtained from the login request. If an initiator fails to complete a successful reconnect request within *reconnect_hold* + 1 seconds after a bus reset or net update, the target will perform a logout and release all resources held by that initiator (see 8.3).

5.2.4.3 Query logins ORB

An initiator may determine the EUI-64 and node ID, local or global, of all currently logged-in initiators by means of a query logins request, whose format is illustrated below.

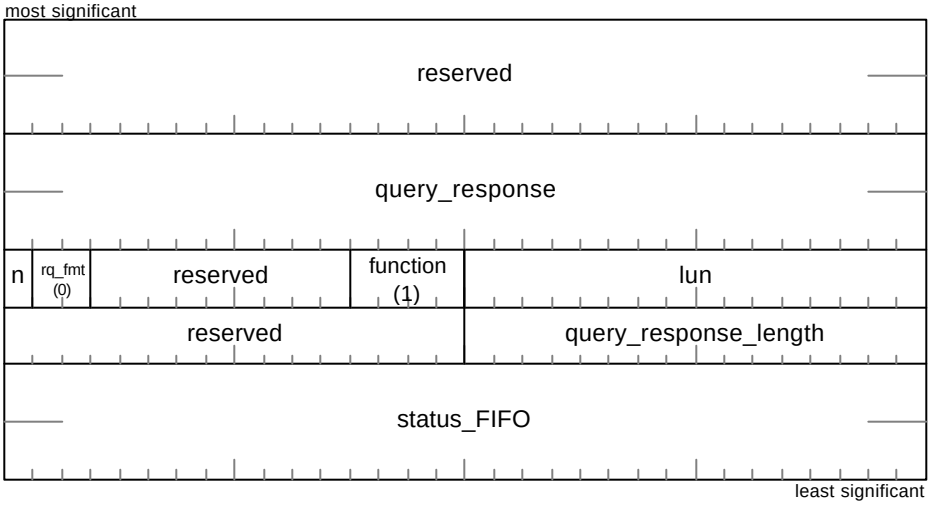

Figure 20 – Query logins ORB

The *query_response* and *query_response_length* fields specify the address and size of a buffer for the return of the query results. The *query_response* field shall conform to the format for address pointers specified by Figure 10 and shall address the same node as the initiator; consequently the *node_ID* field of this address pointer is reserved. The buffer shall be accessible to a Serial Bus block write request with a data transfer length less than or equal to *query_response_length*.

The *notify* bit, *rq_fmt* and *status_FIFO* fields are as previously defined for management ORB formats.

The *lun* field specifies the logical unit number (LUN) to which the request is addressed.

The query response data returned shall have the following format.

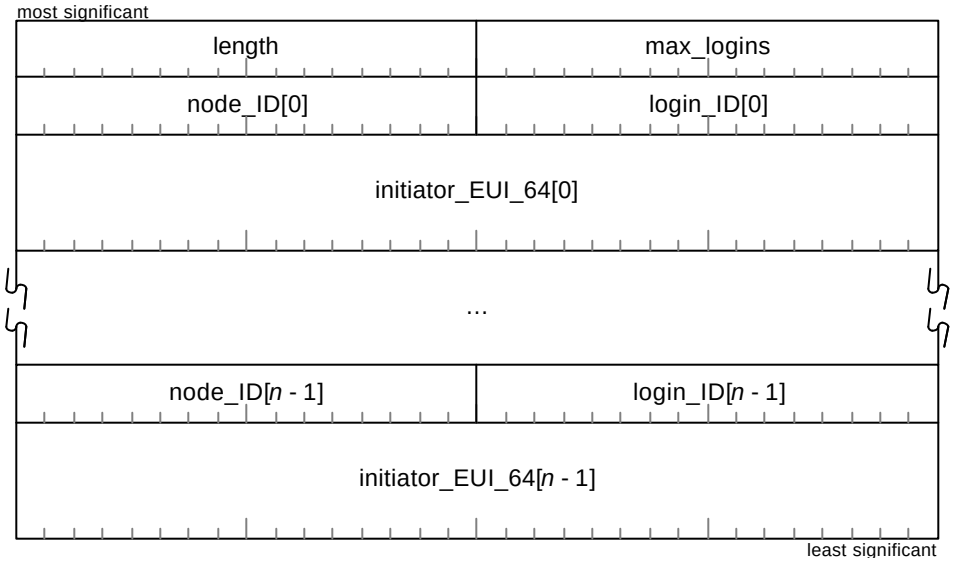

Figure 21 – Query logins response format

The *length* field shall contain the length, in bytes, of the query response data. The value of the *length* field shall be equal to 4+ 12 * *n*, where *n* is the number of logged-in initiators. If *query_response_length* in the query logins request is too small for the transfer of all the query response data, the *length* field shall not be adjusted to reflect the truncation.

The *max_logins* field shall contain the maximum concurrent logins that may be accepted by the logical unit.

The remainder of the query response is a variable-length array of 12-byte entries, one for each logged-in initiator, each of which contains a *node_ID*, *login_ID* and *initiator_EUI_64* field.

The *node_ID* field of an entry shall contain the node ID of a logged-in initiator. If a Serial Bus reset or net update (the former in the case of local node IDs and the latter in the case of global node IDs) has occurred since the login was established and the initiator has not reconnected the login, the *node_ID* field shall have a value of $FFFF_{16}$.

NOTE – A *node_ID* value of $FFFF_{16}$ may be observed only in the reconnect interval that exists for *reconnect_hold* + 1 seconds after a Serial Bus reset or net update because after this time the target performs an automatic logout for any initiator that has not reconnected.

If the *node_ID* field has a value of $FFFF_{16}$, the *login_ID* field shall contain the time remaining, in seconds less one, until the initiator is automatically logged-out by the target. Otherwise, the *login_ID* field of an entry shall contain the login ID provided to the initiator as a result of its successful login.

The *initiator_EUI_64* field of an entry shall contain the EUI-64 obtained by the target from the initiator's configuration ROM at the time the login was validated.

5.2.4.4 Create task set ORB

An initiator already logged-in to a target logical unit may request the creation of an additional task set (other than the primary task set associated with the login) by means of an ORB with the format shown below.

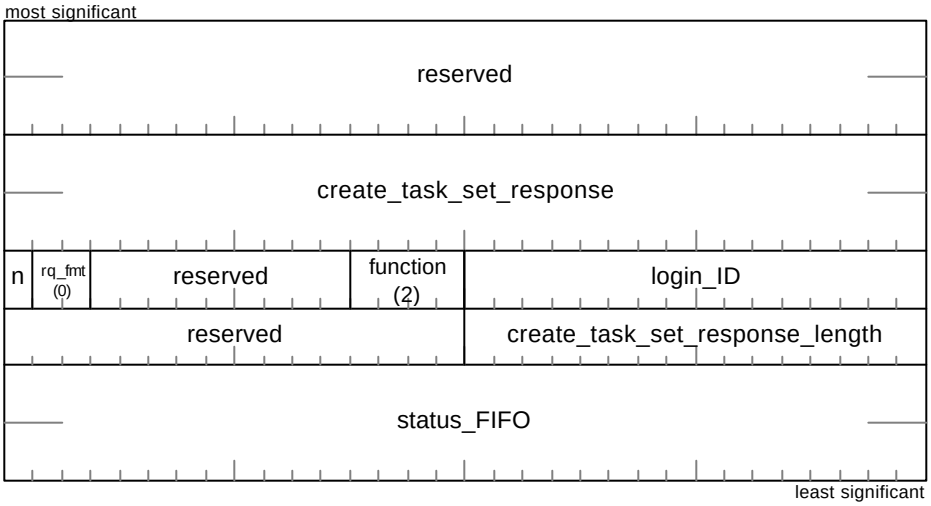

Figure 22 – Create task set ORB

The *create_task_set_response* and *create_task_set_response_length* fields specify the address and size of a buffer allocated for the return of the create task set response. The *create_task_set_response* field shall conform to the format for address pointers specified by Figure 10. The buffer shall be in the same node as the initiator and shall be accessible to a Serial Bus block write transaction with a data transfer length less than or equal to *create_task_set_response_length*. The initiator shall set *create_task_set_response_length* to a value of at least 12; the target may ignore this field.

The *notify* bit and the *rq_fmt* field are as previously defined for management ORB formats.

The *login_ID* field shall contain a login ID value obtained as the result of a successful login.

The *status_FIFO* field is as previously defined for management ORB formats and shall contain an address allocated for the return of status for the CREATE TASK SET request, status for all subsequent requests signaled to the *command_block_agent* allocated for this task set.

If the create task set request fails the contents of the response buffer are unspecified. Otherwise, upon successful completion of a create task set request, the response is returned in the format illustrated below.

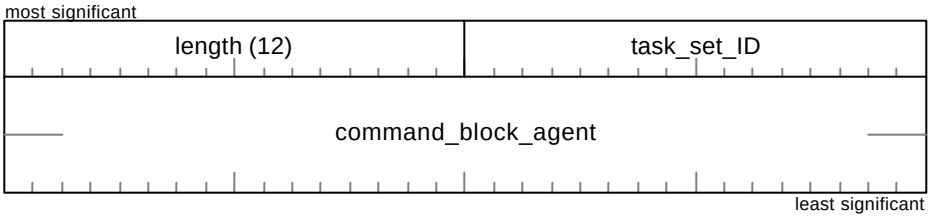

Figure 23 – Create task set response

The *length* field shall contain the length, in bytes, of the create task set response data and shall be equal to 12.

The *task_set_ID* identifies a task set for which target resources have been allocated. The value of *task_set_ID* shall differ from the *login_ID* with which it is associated and shall be unique within the context

of the target. The initiator shall use this value to identify all subsequent requests directed to the target's management agent that pertain to this task set.

The *command_block_agent* field specifies the base address of the agent's CSRs, which are defined in 6.6. This field shall conform to the format for address pointers specified by Figure 10.

5.2.4.5 Reconnect ORB

After a Serial Bus reset or net update an initiator shall reestablish access for a previously valid login before it signals new requests to the target for that login. This is accomplished by means of a reconnect request, with the format shown below.

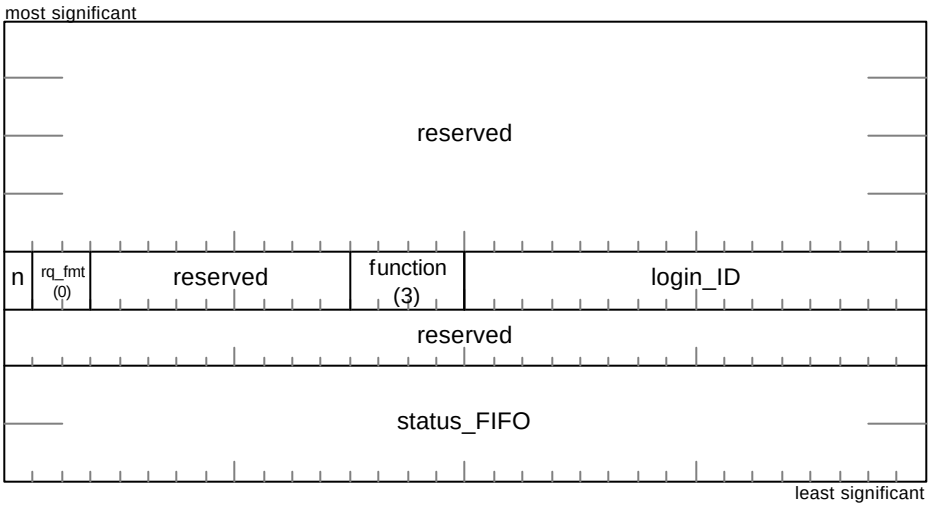

Figure 24 – Reconnect ORB

The *notify* bit and the *rq_fmt* field are as previously defined for management ORB formats.

The *login_ID* field shall contain a login ID value obtained as the result of a successful login. The target shall verify that the EUI-64 of the initiator requesting the login reestablishment matches the EUI-64 previously saved by the target for the *login_ID*.

The *status_FIFO* field is as previously defined for management ORB formats and shall contain an address allocated for the return of status for the RECONNECT request, only. The contents of this field shall not update the status FIFO address established by the successful login that returned *login_ID*.

Upon successful reestablishment of the login, the initiator may signal requests to the target agent at the same CSR addresses returned in the original login response data. The initiator shall also use the *login_ID* value to identify all requests directed to the target's management agent that pertain to the reestablished login.

Any secondary task sets associated with the same *login_ID* value specified in the reconnect ORB are once again available for use. The task set IDs of the secondary task sets remain the same.

5.2.4.6 Node handle ORB

When an initiator establishes a login in bridge-aware mode, it shall obtain node handles to use in all address pointers signaled to the target for the duration of the login. A node handle for a particular node, identified by its EUI-64, may be obtained by a node handle request that uses the ORB format shown below.

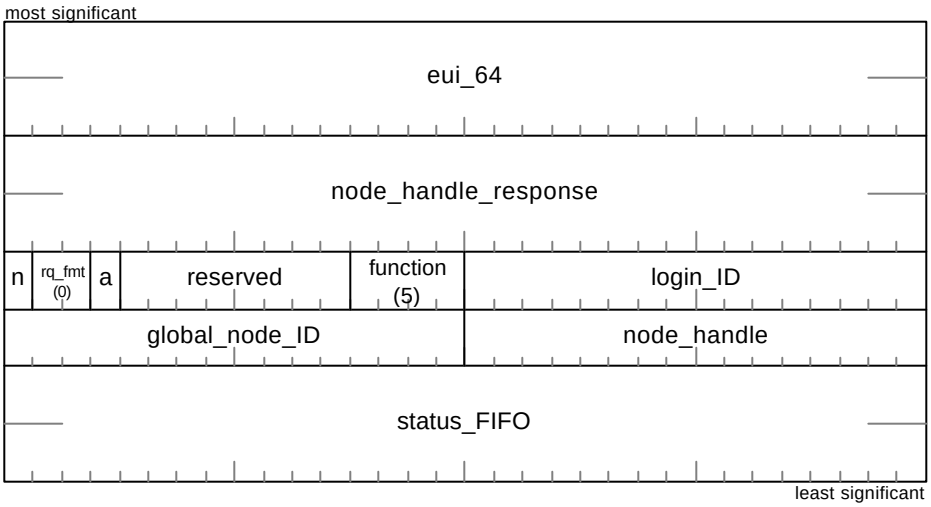

Figure 25 – Node handle ORB

The *notify* bit and the *rq_fmt* field are as previously defined for management ORB formats.

The value of the *allocate* bit (abbreviated as *a* in the figure above) determines the operation to be performed. If the *allocate* bit is one, the target is requested to provide (or update) a node handle that corresponds to the EUI-64 supplied and return it in the buffer specified by *node_handle_response*. Otherwise, the target is requested to release a particular node handle or all node handles (except the initiator's own node handle) associated with the login identified by *login_ID*.

When the *allocate* bit is one, the *eui_64* field shall contain the unique identifier of the node for which the node handle is requested and shall not be equal to the initiator's EUI-64. Otherwise, when the *allocate* bit is zero the value of *eui_64* is unspecified.

When the *allocate* bit is one, the *node_handle_response* field shall specify the address of a quadlet buffer allocated for the return of the node handle. The *node_handle_response* field shall conform to the format for address pointers specified by Figure 11 and shall address the same node as the initiator; consequently the *node_ID* field of this address pointer is reserved. The buffer shall be accessible to a Serial Bus quadlet write request. Otherwise, when the *allocate* bit is zero the value of *node_handle_response* is unspecified.

The *login_ID* field shall contain a login ID value obtained as the result of a successful login.

When the *allocate* bit is one, *global_node_ID* shall contain a global node ID that identifies the node whose EUI-64 matches that specified by the *eui_64* field.

The *node_handle* field is meaningful only if the *allocate* bit is zero. When *node_handle* equals $FFFF_{16}$, the target is requested to release all node handles associated with the login identified by *login_ID* (except the initiator's own node handle). For other nonzero values of *node_handle*, the target is requested to release the node handle specified.

The *status_FIFO* field is as previously defined for management ORB formats and shall contain an address allocated for the return of status for the NODE HANDLE request, only. The contents of this field shall not update the status FIFO address established by the successful login that returned *login_ID*.

If the node handle request fails, the contents of the *node_handle_response* buffer are unspecified. Otherwise, upon successful completion of a request to provide a node handle, the target shall store the quadlet illustrated below.

Figure 26 – Node handle response

The *node_handle* field shall contain a node handle whose value is assigned by the target. The most significant ten bits of the node handle shall not be all ones. The initiator may use *node_handle* in place of a local node ID in any address pointer signaled to the target in the context of the login identified by *login_ID*. For the duration of the login, the node identified by *node_handle* shall be the one specified by the *eui_64* field in the node handle request.

5.2.4.7 Logout ORB

In order to relinquish its access privileges for a logical unit or a secondary task set, an initiator shall perform a logout with the ORB format shown below.

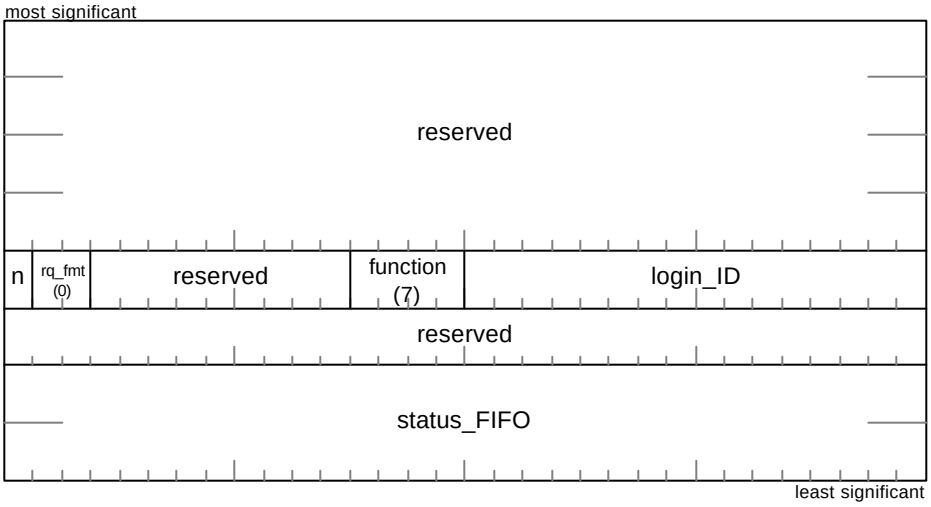

Figure 27 – Logout ORB

The *notify* bit, *rq_fmt* and *status_FIFO* fields are as previously defined for management ORB formats.

The *login_ID* field shall contain a login ID value obtained as the result of a successful login or create task set request.

5.2.4.8 Task management ORB

The task management ORB is used to control task sets. This ORB shall have the format defined below.

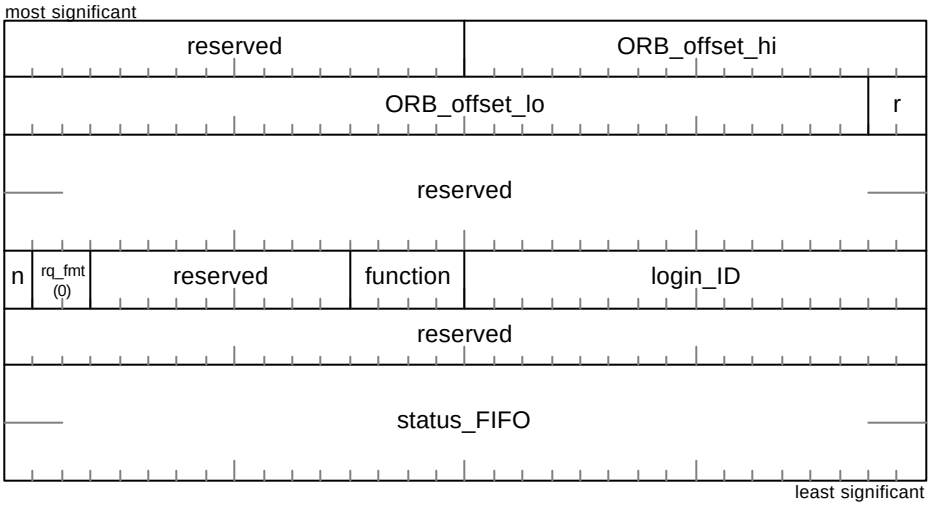

Figure 28 – Task management ORB

The *ORB_offset_hi* and *ORB_offset_lo* fields together form the *ORB_offset* field, which identifies the task to which the management function applies. *ORB_offset* is derived by taking the least significant 48 bits of the Serial Bus address of the ORB and discarding the least significant two bits. The *ORB_offset* field is ignored unless the *function* field is ABORT TASK. All tasks are uniquely identified by the Serial Bus address of the ORB that initiated the task.

The *notify* bit, *rq_fmt* and *status_FIFO* fields are as previously defined for management ORB formats.

The *function* field shall contain a value of ABORT TASK, ABORT TASK SET, LOGICAL UNIT RESET or TARGET RESET, as defined by Table 2.

The *login_ID* shall be set to the value returned in login response data or to the value of *task_set_ID* returned in create task set response data. In either case, *login_ID* identifies the task set or sets to which the task management request is directed. In the case of TARGET RESET, which does not pertain to any one task set, *login_ID* shall be set to a value obtained as the result of any successful login completed by the initiator.

5.3 Page tables

5.3.1 Overview

The data buffer specified by a command block ORB is described by the *data_descriptor*, *page_table_present*, *page_size* and *data_size* fields. The data buffer is a logically contiguous area in system memory. As previously described, when *page_table_present* is zero, the data buffer is also contiguous within Serial Bus address space and no more than 65,535 bytes in length. In this case, *data_descriptor* contains the 64-bit address of the data buffer and *data_size* specifies its length, in bytes.

When the data buffer cannot be directly addressed (either because it is discontiguous or too large), it is necessary to describe it *via* a page table. A page table is a variable-length array of elements, each of which describes a segment that is contiguous within Serial Bus address space. Page table elements are eight bytes long and shall be octlet aligned within system memory.

The presence of a page table is indicated by the value of *page_table_present* in the ORB. When *page_table_present* is one, the *data_descriptor* field in the ORB shall contain the address of the page table and the *data_size* field shall contain the number of elements in the page table.

Page tables may have one of two formats: an unrestricted page table or a normalized page table. The page table format is determined by *page_size*. When *page_size* is zero there are no underlying page boundaries to restrict the size or alignment of data buffer segments; this is the unrestricted format. Otherwise the size and alignment of data buffer segments is determined by the nonzero *page_size*; this is the normalized format.

The *spd* and *max_payload* fields of the ORB shall describe data transfer capabilities for the page table and may also pertain to the data buffer. The data buffer may be entirely co-located in the same node as the page table, it may be entirely within a different node or it may be distributed among two or more nodes—one of which may be the node that contains the page table. Portions of the data buffer not in the node that contains the page table shall be described by node selector entries (see 5.3.4) embedded within the page table. Whether the data buffer is contained within a single node or distributed, system memory addressed by a target request subaction that accesses the data buffer shall be entirely contained within a data buffer segment described by a single page table element.

5.3.2 Unrestricted page tables

An unrestricted page table shall be contiguous within Serial Bus address space and shall be accessible to block read requests with a *data_length* less than or equal to the smaller of *data_size* * 8 bytes or 2^{max_rec+1} bytes. The format of elements in an unrestricted page table is shown by Figure 29.

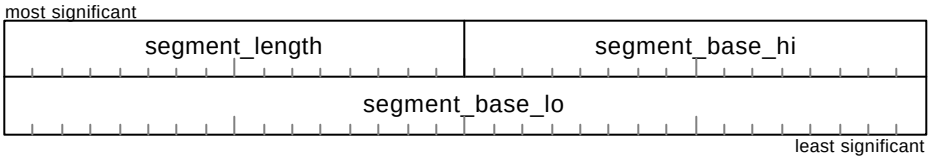

Figure 29 – Page table element (unrestricted page table)

The *segment_length* field shall contain the length, in bytes, of the portion of the data buffer (segment) described by the page table element. The value of *segment_length* shall be nonzero.

NOTE – A zero value in the same position as the *segment_length* field differentiates a node selector from a page table entry (see 5.3.4).

The *segment_base_hi* and *segment_base_lo* fields together shall specify the base address of the segment within the node's 48-bit system memory address range.

The 64-bit system memory address used to address the data is formed by the concatenation of the 16-bit *node_ID* field from the previous node selector or, if there is no previous node selector in the page table, the *node_ID* field from the *data_descriptor* field in the ORB, *segment_base_hi* and *segment_base_lo*.

5.3.3 Normalized page tables

A normalized page table shall be contiguous within Serial Bus address space and shall be accessible to Serial Bus block read requests with a *data_length* less than or equal to the smallest of *data_size* * 8 bytes, 2^{max_rec+1} bytes or 2^{page_size+8} bytes if the data requested does not cross Serial Bus address boundaries that occur every 2^{page_size+8} bytes relative to zero.

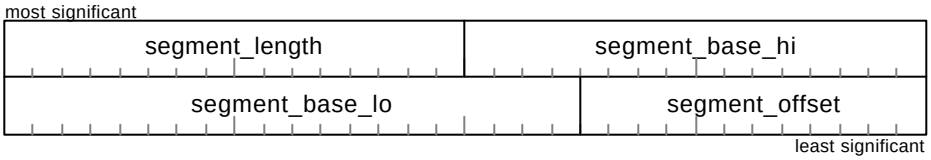

Figure 30 – Page table element (when *page_size* equals four)

NOTE – In the figure above, the field widths of *segment_base_lo* and *segment_offset*, 20 and 12 bits, respectively, are chosen only for the purposes of illustration. The field widths of *segment_base_lo* and *segment_offset* vary according to *page_size*. The field width, in bits, of *segment_offset* shall be *page_size* + *8*. In the example shown above, the page size is assumed to be 4096 bytes.

The *segment_length* field shall contain the length, in bytes, of the portion of the data buffer (segment) described by the page table element. The value of *segment_length* shall be nonzero and less than or equal to $2^{page\_size + 8}$.

NOTE – A zero value in the same position as the *segment_length* field differentiates a node selector from a page table entry (see 5.3.4).

The *segment_base_hi* and *segment_base_lo* fields together shall specify the base address of the segment within the node's 48-bit system memory address range.

The *segment_offset* field shall contain the starting address for data transfer within the segment.

The 64-bit system memory address used to address the data is formed by the concatenation of the 16-bit *node_ID* field from the previous node selector or, if there is no previous node selector in the page table, the *node_ID* field from the *data_descriptor* field in the ORB, *segment_base_hi*, *segment_base_lo* and *segment_offset*.

In all page table elements, the sum of *segment_length* and *segment_offset* shall be less than or equal to $2^{page\_size + 8}$.

In addition to the preceding requirements, the values of *segment_length* and *segment_offset* are constrained by their position within the page table. These additional restrictions are summarized below.

Element Position	**Total page table elements**		
	1	**2**	***n* (where *n* >= 3)**
First	No additional restrictions	*segment_length* = $2^{page\_size + 8}$ - *segment_offset*	
Middle	—		*segment_offset* = 0 *segment_length* = $2^{page\_size + 8}$
Last	—	*segment_offset* = 0	

5.3.4 Node selectors

A node selector is an 8-byte entry in a page table that identifies the node referenced by subsequent page table entries. A node selector applies to all subsequent page table entries until another node selector or the end of the page table is encountered. Node selectors permit a data buffer to be located in a different node than the page table; they also permit a data buffer to be distributed among more than one node. The format of a node selector is shown by Figure 31.

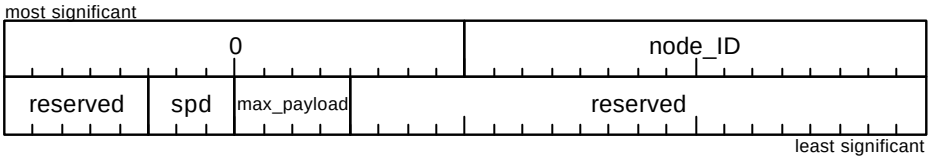

Figure 31 – Node selector

The most significant 16 bits of a node selector shall be zero.

The *node_ID* field shall identify the Serial Bus node to which subsequent page table entries pertain; it shall contain either a local node ID, as specified by IEEE 1394, or a node handle supplied by a target, as specified by this standard.

The *spd* and *max_payload* fields specify the speed and maximum data payload that shall be used by the target in request subactions addressed to the node identified by *node_ID*. The encoding of these fields is the same as the identically named fields in the command block ORB (see 5.2.3).

Target support for node selectors is optional and is indicated by the Unit_Characteristics entry in configuration ROM (see 7.8.10).

5.4 Status block

5.4.1 Status block formats

A target may store status at an initiator *status_FIFO* address when a request completes (successfully or in error) or because of an unsolicited event (device status change or interim status for an ORB). The *status_FIFO* address is obtained either explicitly from the ORB to which the status pertains or implicitly from the fetch agent context. Whenever the target has status to report and is enabled to do so, it shall store either a basic status block, in the format specified by Figure 32, or an extended status block, in the format specified by Figure 33.

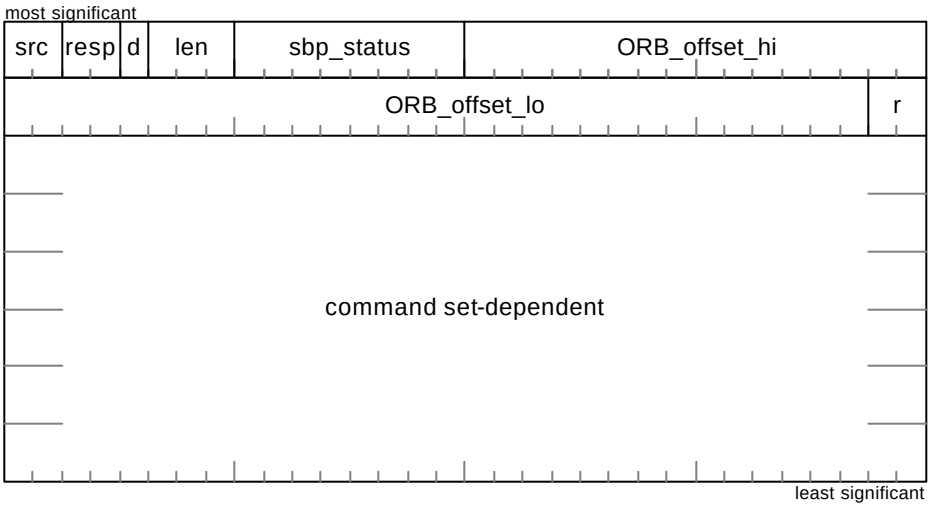

Figure 32 – Basic status block format

When the basic status format is used, the target shall store a minimum of eight bytes of status information and may store up to the entire 32 bytes defined above; the amount of data stored shall be an integral

number of quadlets. A truncated basic status block shall be interpreted as if the omitted fields had been stored as zeros. The target shall use a single Serial Bus block write transaction to store the status block at the *status_FIFO* address.

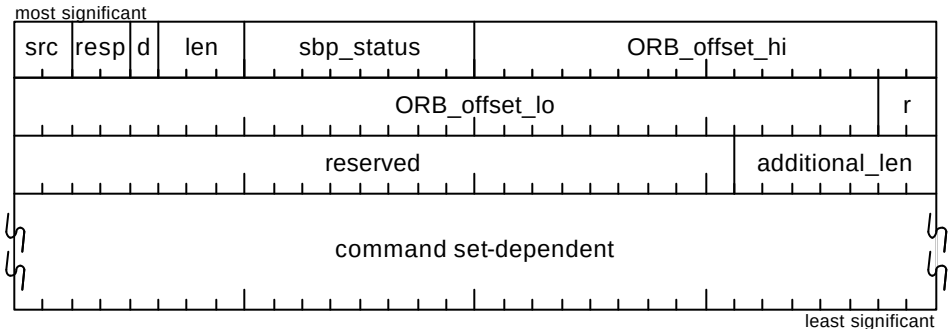

Figure 33 – Extended status block format

When the extended status format is used, the target shall store a minimum of 40 bytes of status information and may store up to 512 bytes; the amount of data stored shall be an integral number of quadlets. The target shall use a single Serial Bus block write transaction to store the status block at the *status_FIFO* address.

The *src* field indicates the origin of the status block, as specified by the table below.

Value	Description
0	The status block pertains to the ORB identified by *ORB_offset_hi* and *ORB_offset_lo*; at the time the ORB was most recently fetched by the target the *next_ORB* field did not contain a null pointer.
1	The status block pertains to the ORB identified by *ORB_offset_hi* and *ORB_offset_lo*; either the *next_ORB* field is absent or at the time the ORB was most recently fetched by the target the *next_ORB* field was null.
2	The status block is unsolicited and contains device status information; the contents of the *ORB_offset_hi* and *ORB_offset_lo* fields shall be ignored.
3	The status block contains interim request status that pertains to the ORB identified by *ORB_offset_hi* and *ORB_offset_lo*. No information is provided as to the value of the *next_ORB* field at the time the ORB was most recently fetched by the target.

When *src* is zero or one, the status is final; a target shall store final status no more than once for the corresponding ORB. When *src* is two, the status is unsolicited; interlock between the initiator and target is managed independently (see 5.4.3). Otherwise, when *src* is three, the status pertains to a particular ORB but is interim; a target shall store interim status no more than once for the corresponding ORB.

The *resp* field shall contain a response status defined in the table below.

Value	Name	Description
0	REQUEST COMPLETE	The request completed without transport protocol error (Either *sbp_status* or command set-dependent status information may indicate the success or failure of the request)
1	TRANSPORT FAILURE	The target detected a nonrecoverable transport failure that prevented the completion of the request
2	ILLEGAL REQUEST	There is an unsupported field or bit value within the first 20 bytes of a single buffer descriptor ORB or within the first 32 bytes of a dual buffer descriptor ORB
3	VENDOR DEPENDENT	The meaning of *sbp_status* shall be specified by the vendor

The *dead* bit (abbreviated as *d* in the figure above) shall indicate whether or not the logical unit fetch agent ~~transitioned to~~ entered the dead state upon storing the status block. When *dead* is zero, the reported status has not affected the state of the fetch agent. If the *dead* bit is set to one, the fetch agent ~~transitioned to~~ entered the dead state as a consequence of the error condition reported by the status block.

The *len* field shall specify the status block format, basic or extended, and, in the case of the basic status block, the quantity of valid status block information stored at the *status_FIFO* address. A *len* value of zero indicates the extended status block format; consult the *additional_len* field to determine the size of the status block. Otherwise, the size of the basic status block is encoded as *len* + 1 quadlets.

The *sbp_status* field provides additional information that qualifies the response status in *resp*. The meanings assigned to *sbp_*status vary according to the value of *src* and *resp* and are described below.

When *src* is zero, one or three, the *ORB_offset_hi* and *ORB_offset_lo* fields together uniquely identify the ORB to which the status block pertains. Otherwise, if *src* is two, the *ORB_offset_hi* and *ORB_offset_lo* fields are ignored.

For the basic status block format, the remainder of the status block after the first two quadlets, up to an overall maximum of 32 bytes, is command set-dependent.

When the extended status block format is used, the *additional_len* field shall specify the number of quadlets of command set-dependent information that follow. The value of *additional_len* shall be between seven and 125, inclusive. The maximum size of an extended status block is 512 bytes.

5.4.2 Request status

Upon completion of a request, if the *notify* bit in the ORB is one or if there is exception status to report, the target shall store ~~all or part of the~~ a status block in either of the formats shown in Figure 32 and Figure 33. For management ORBs (which explicitly provide the *status_FIFO* address as part of the ORB), the target shall store the status block at the address specified. Otherwise (for command block ORBs) the target shall store the status block at the *status_FIFO* determined by the fetch agent to which the ORB was signaled. In the case of command block ORBs the initiator provides the *status_FIFO* address as part of the login or create task set request.

When *resp* is equal to zero, REQUEST COMPLETE, the possible values for *sbp_status* are specified by the table below. Any value not enumerated is reserved for future standardization.

Value	Description
0	No additional information to report
1	Request type not supported
2	Speed not supported
3	Page size not supported
4	Access denied
5	Logical unit not supported
6	Maximum payload too small
7	Reserved for future standardization
8	Resources unavailable
9	Function rejected
10	Login ID invalid
11	Dummy ORB completed
12	Request aborted
13	Unknown EUI-64
14	Node handle invalid
FF_{16}	Unspecified error

If a logical unit implements a command set that utilizes both single and dual buffer descriptor command ORBs, its fetch agent shall be able to parse both ORB formats in order to deliver the *command_block* to the device server. The device server shall accept or reject the command and return a status block with command set-dependent status information that indicates success or failure. In other words, the logical unit fetch agent shall not reject an otherwise well-formed ORB whose *rq_fmt* value is zero or one by returning an *sbp_status* of one, request type not supported.

If a Serial Bus error occurs in the transport (*resp* is equal to one, TRANSPORT FAILURE), the *sbp_status* field either shall have a value of FF_{16}, unspecified error, or else the field shall be redefined as illustrated below. This format provides for the return of additional information about the transport failure.

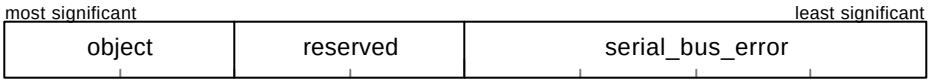

Figure 34 – TRANSPORT FAILURE format for *sbp_status*

The *object* field shall specify which component of an SBP-3 request, the ORB, the data buffer or the page table, was referenced by the target when the error occurred. The value of *object* shall be as defined by the following table.

Value	Referenced object
0	Operation request block (ORB)
1	Data buffer
2	Page table
3	Unable to specify

The *serial_bus_error* field shall contain the error response for the failed request, as encoded by the table below.

Value	Serial Bus error	Comment
0	Missing acknowledge	
1	Reserved; not to be used	
2	Time-out error	An *ack_pending* was received for the request but no response subaction was completed within the time-out limit
3	Reserved; not to be used	
4 – 6	Busy retry limit exceeded	The value reflects the last acknowledge, *ack_busy_X*, *ack_busy_A* or *ack_busy_B*
7 – A_{16}	Reserved for future standardization	
B_{16}	Tardy retry limit exceeded	An *ack_tardy* was received for the request and the vendor-dependent retry limit (which may be based upon either time or number of occurrences) for tardy responses has been exceeded
C_{16}	Conflict error	A resource conflict was detected by the addressed node
D_{16}	Data error	The data field failed the CRC check or the observed length of the payload did not match the *data_length* field
E_{16}	Type error	A field in the request was set to an unsupported value or an invalid transaction was attempted (*e.g.*, a write to a read-only address)
F_{16}	Address error	The *destination_offset* field specified an inaccessible address in the addressed node

In the cases of conflict error and data error, these are errors that the target may retry up to an implementation-dependent limit before reporting TRANSPORT FAILURE.

No additional information is provided in *sbp_status* when *resp* equals two, ILLEGAL REQUEST. In this case, *sbp_status* shall be set to FF_{16}. An SBP-3 response code of ILLEGAL REQUEST shall not be used to indicate unsupported fields or bit values in the command set-dependent portion of the ORB. This response code shall be used only to indicate an error in the first 20 bytes of ~~the~~ a single buffer descriptor ORB or the first 32 bytes of a dual buffer descriptor ORB.

The *ORB_offset_hi* and *ORB_offset_lo* fields together form an *ORB_offset* field that uniquely identifies the ORB to which the status block pertains. The target shall form *ORB_offset* from the least significant 48 bits of the Serial Bus address used to fetch the ORB; the least significant two bits shall be discarded.

5.4.3 Unsolicited device status

When a change in device status occurs that affects a logical unit, the target may store ~~the~~ a status block in either of the formats shown in Figure 32 and Figure 33 at the *status_FIFO* address provided by the initiator as part of a login request (see 5.2.4.2). If a target stores unsolicited status for any initiator logged-in to a logical unit it shall attempt to store status for all initiators logged-in to the same logical unit.

The *src* field shall be one to indicate unsolicited device status.

The *resp* field shall have a value of REQUEST COMPLETE or VENDOR DEPENDENT.

The *dead* bit and the *len* field are as previously defined for the status block.

If *resp* is equal to REQUEST COMPLETE, *sbp_status* shall be zero. Otherwise the content and meaning of *sbp_status* shall be specified by the vendor.

The contents of the *ORB_offset_hi* and *ORB_offset_lo* fields are unspecified and shall be ignored by the initiator.

5.4.4 Interim request status

Prior to the completion of a request, the target may store all or part of the status block shown in Figure 32 at the *status_FIFO* determined by the fetch agent to which the ORB was signaled; the initiator provides the *status_FIFO* address as part of the login or create task set request.

The *src* field shall have a value of three to indicate interim request status.

The *resp* field shall have a value of REQUEST COMPLETE or VENDOR DEPENDENT.

The *dead* bit and the *len* field are as previously defined for the status block.

If *resp* is equal to REQUEST COMPLETE, *sbp_status* shall be zero. Otherwise the content and meaning of *sbp_status* shall be specified by the vendor.

The *ORB_offset_hi* and *ORB_offset_lo* fields together form an *ORB_offset* field that uniquely identifies the ORB to which the status block pertains. The target shall form *ORB_offset* from the least significant 48 bits of the Serial Bus address used to fetch the ORB; the least significant two bits shall be discarded.

6 Control and status registers

6.1 Control and status registers overview

The control and status registers (CSRs) implemented by a target shall conform to the requirements defined by this standard and its normative references. The CSRs are arranged in three principal categories:

- core registers specified by IEEE Std 1212-2001;
- bus-dependent registers specified by IEEE 1394; and
- unit architecture registers specified by this standard.

Unless otherwise specified, all registers shall support quadlet read and quadlet write transactions. The registers defined in 6.4 through 6.6 shall ignore broadcast write requests. Under certain circumstances, these registers may also ignore other subactions, in which case ~~either~~ *ack_complete* (or *ack_pending* subsequently followed by *resp_complete*) should be returned—even though the request subaction has had no effect on the state of the register or target.

6.2 Core registers

The CSR Architecture standardizes the locations and functions of core registers. The addresses of these registers are specified in terms of offsets, in bytes, within register space, where the base address of register space is FFFF F000 0000_{16} relative to node space. IEEE 1394 should be consulted for detailed descriptions of these core registers; the table below summarizes which core registers are mandatory for targets.

Offset	Register name	Description
0	STATE_CLEAR	State and control information
4	STATE_SET	Sets STATE_CLEAR bits
8	NODE_IDS	Contains the 16-bit *node_ID* value used to address the node
$0C_{16}$	RESET_START	Resets the node's state
$18_{16} – 1C_{16}$	SPLIT_TIMEOUT	Time limit for local bus split transactions

The CSR Architecture and IEEE 1394 define the effects of a write to the RESET_START register. In addition to those requirements, a write to RESET_START should cause all of a node's SBP-3 units to reset in the same fashion as a power reset.

NOTE – Because of the potential for malicious interference in target operations by an unauthorized node, it is recommended that a write to RESET_START have no effect upon a target unless either a) there are no logged-in initiators or b) the *source_ID* of the write matches ~~that~~ the node ID of one of the currently logged-in initiators.

Bridge-awareness is optional for targets. If a target is bridge-aware, it shall implement additional core registers, as summarized by the table below.

Offset	Register name	Description
$80_{16} – BC_{16}$	MESSAGE_REQUEST	Well-known addresses for the exchange of messages between nodes
$C0_{16} – FC_{16}$	MESSAGE_RESPONSE	

6.3 Serial Bus-dependent registers

The CSR Architecture reserves a portion of register space for bus-dependent uses. Serial Bus defines registers within this address space, whose addresses are specified in terms of offsets, in bytes, within register space, where the base address of register space is FFFF F000 0000_{16} relative to node space. IEEE 1394 should be consulted for detailed descriptions of these core registers; the table below summarizes which Serial Bus-dependent registers are mandatory for targets.

Offset	Register name	Description
210_{16}	BUSY_TIMEOUT	Controls transaction layer retry protocols

Isochronous capabilities are optional for targets. If a target supports isochronous operations, it shall be cycle master capable and isochronous resource manager capable as well as isochronous capable. These capabilities require that additional Serial Bus-dependent registers shall be implemented, as summarized by the table below.

Offset	Register name	Description
200_{16}	CYCLE_TIME	24.576 MHz clock required for isochronous operation
204_{16}	BUS_TIME	System time in seconds
$21C_{16}$	BUS_MANAGER_ID	Contains the *node_ID* of the bus manager, if one is present
220_{16}	BANDWIDTH_AVAILABLE	Well-known location for Serial Bus isochronous bandwidth allocation
$224_{16} – 228_{16}$	CHANNELS_AVAILABLE	Well-known location for Serial Bus channel allocation
234_{16}	BROADCAST_CHANNEL	Channel number for asynchronous stream broadcast.

Bridge-awareness is optional for targets. If a target is bridge-aware, it shall implement additional Serial Bus-dependent registers, as summarized by the table below.

Offset	Register name	Description
214_{16}	QUARANTINE	Permits bridge-aware nodes to manage their quarantine periods (see draft standard IEEE P1394.1)

6.4 BUSY_TIMEOUT register

The BUSY_TIMEOUT register makes visible transaction layer variables that control target retry behavior for busied subactions. For targets compliant with this standard, the register has different initial values than specified by IEEE 1394 and does not change them in response to write requests. The format of the register is illustrated by Figure 35.

Responses from the BUSY_TIMEOUT register to write requests are not the same as responses from a read-only register. Read-only registers reject write requests with a response of type error; in the absence of busy conditions or other errors, write requests to the BUSY_TIMEOUT register shall receive a successful completion response but the write values shall be ignored.

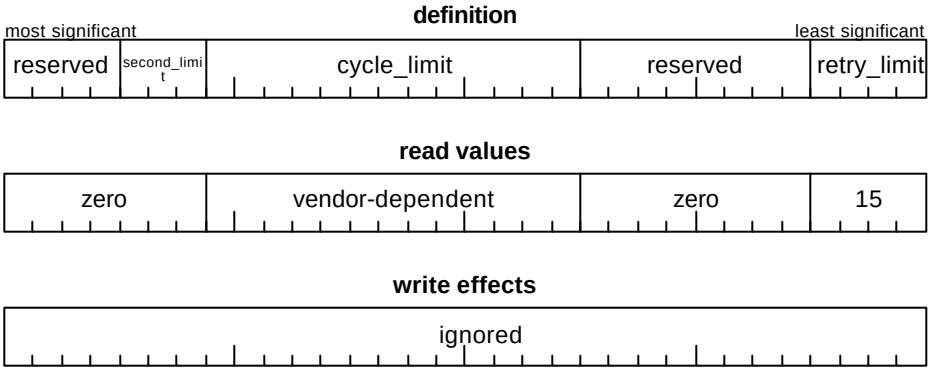

Figure 35 – BUSY_TIMEOUT format

The *second_limit* and *cycle_limit* fields specify the request subaction retry behavior utilized by the transaction layer when dual-phase retry protocol is used. A zero value in these fields indicates dual-phase retry protocol is not supported. Together, the *second_limit* and *cycle_limit* fields define a time limit for request subaction retry attempts. The format of these fields and the units used are identical to the *second_count* and *cycle_count* fields in the CYCLE_TIME register. When dual-phase retry protocol is active for a request subaction initiated by the target, the target shall not retransmit the subaction after this time limit has elapsed. Time counts from the receipt of the first busy acknowledgement for the request subaction. Although the read value of the *cycle_limit* field is vendor-dependent, it shall be either zero or 800.

The *retry_limit* field specifies the request subaction retry behavior utilized by the transaction layer when single-phase retry protocol is used. The target shall retransmit the request subaction *retry_limit* times until either no acknowledgment is observed or the receipt of a terminal acknowledgment (any acknowledgment, including *ack_pending*, other than *ack_busy_X*, *ack_busy_A* or *ack_busy_B*).

Although the BUSY_TIMEOUT register exposes transaction layer variables that control target retry behavior, it does not constitute a complete specification of retry behavior. Targets compliant with this standard shall behave as follows:

- A target should implement dual-phase retry protocol for inbound subactions;
- A target should always attempt to use dual-phase retry protocol for outbound subactions; the retry code, *rt*, of the oldest subaction addressed to a particular *destination_ID* should be set to *retry_1*. See IEEE Std 1394a-2000 for more information on the “oldest” subaction;
- A target shall retry response subactions until either no acknowledgment is observed, a terminal acknowledgment is received or the time limit specified by the target’s SPLIT_TIMEOUT register is exhausted. Time counts from the transmission of *ack_pending* for the request subaction that causes the response subaction;
- When single-phase retry protocol is in use, a target shall retry request subactions until either no acknowledgment is observed, a terminal acknowledgment is received or 15 retransmissions have been attempted. When dual-phase retry protocol is in use, a target shall retry request subactions until either no acknowledgment is observed, a terminal acknowledgment is received or 100 ms have elapsed since the first busy acknowledgement was received; and
- When single-phase retry protocol is in use, a target shall attempt no more than one retransmission of a busied subaction in the interval between the end of an isochronous period and the following cycle synchronization event. In the case of response subactions, this requirement is in addition to the IEEE

Std 1394a-2000 provision that retransmission of a busied response subaction shall not be attempted until the subsequent fairness interval.

Initiators compliant with this standard should implement the retry behavior described above.

6.5 MANAGEMENT_AGENT register

The MANAGEMENT_AGENT register permits the initiator to signal the address of a management ORB to the target. This register shall support 8-byte block read and block write requests whose *destination_offset* is equal to the address of the MANAGEMENT_AGENT register and shall reject quadlet write requests and all other block read and block write requests. The format of this register is illustrated below.

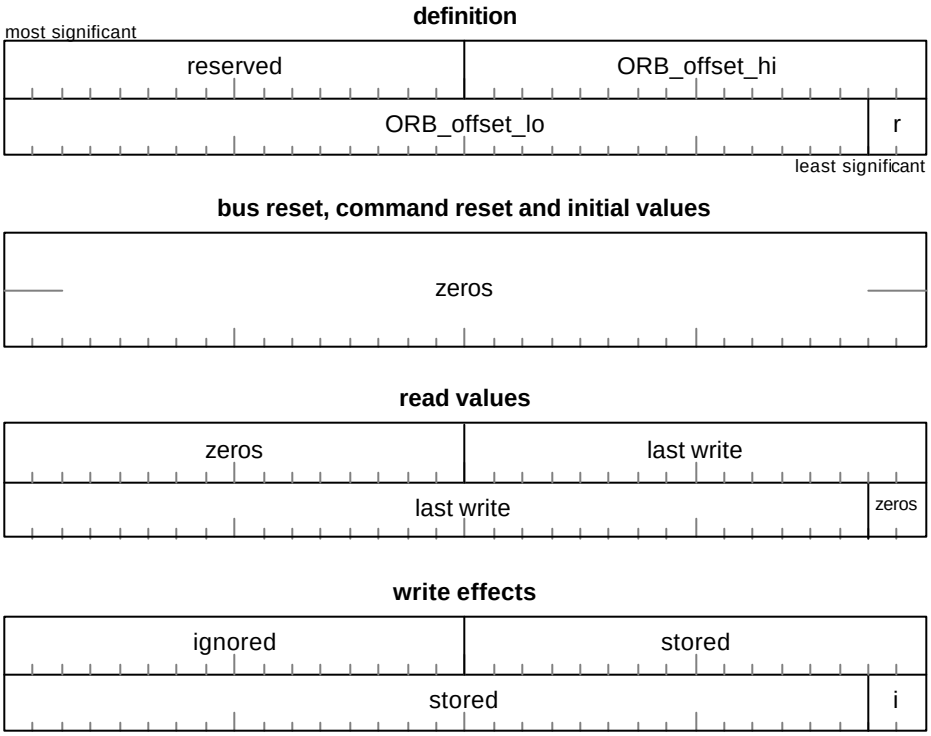

Figure 36 – MANAGEMENT_AGENT format

The *ORB_offset_hi* and *ORB_offset_lo* fields together form an *ORB_offset* field from which a Serial Bus address is derived when the management ORB is fetched. The Serial Bus address shall be formed from the concatenation of the 16-bit node ID of the initiator (available to the target as the *source_ID* field of the block write request that updated the register), the *ORB_offset* field and two least significant bits of zero.

An initiator may signal a request by means of an 8-byte block write transaction that specifies the address of the request. If the management agent is busy with another request, the block write shall be rejected with a response of *resp_conflict_error*. If the write transaction is successful, the management agent shall fetch the request specified by *ORB_offset* and execute it. Unsuccessful write transactions shall not affect the execution of any requests in progress.

Because IEEE 1394 reserves a portion of units space for bus-dependent use, the MANAGEMENT_AGENT register shall be located at address FFFF F001 0000_{16} or above within the node's 48-bit address range. The

address of the management agent is specified by the *csr_offset* field in the Management_Agent entry in configuration ROM (see 7.8.9).

6.6 Command block registers

6.6.1 Command block registers summary

Unlike the management agent, which services a single request at a time, the command block agents manage linked lists from which they fetch requests. For this reason they are referred to as fetch agents. Each fetch agent has a set of control and status registers that lie within the target's units space; the fetch agent CSRs shall be located at or above address FFFF F001 0000_{16} within the node's 48-bit address range.

Although the location of each fetch agent's CSRs is not fixed, the relative relationship of the registers is fixed with respect to each other, as defined by the table below. Implementation of the HEARTBEAT_MONITOR register is optional, but required if the logical unit is bridge-aware. Implementation of the FAST_START register is optional.

Relative offset	Name	Description
00_{16}	AGENT_STATE	Reports fetch agent state
04_{16}	AGENT_RESET	Resets fetch agent
08_{16}	ORB_POINTER	Address of most recently fetched ORB
10_{16}	DOORBELL	Signals fetch agent to refetch an address pointer
14_{16}	UNSOLICITED_STATUS_ENABLE	Acknowledges the initiator's receipt of unsolicited status
18_{16}	HEARTBEAT_MONITOR	Maintains bridge-aware login during target idle periods
$1C_{16} – 3C_{16}$		Reserved for future standardization
vendor-dependent	FAST_START	Signals a reset or suspended fetch agent to start a task; equivalent to a write to the DOORBELL register if the fetch agent is active

The base address of a fetch agent's CSRs is obtained from the *command_block_agent* field in the response returned by the target as part of a successful login or create task set request. The HEARTBEAT_MONITOR register shall not exist in the set of fetch agent CSRs created in response to a create task set request.

A target shall ignore or reject Serial Bus request subactions addressed to any of a fetch agent's CSRs unless the *source_ID* matches the node ID of the initiator associated with the login (the preferred action is to reject such subactions with a response of type error). See 9.2.6 for more information.

6.6.2 AGENT_STATE register

The AGENT_STATE register is a read-only register that provides information about the current condition of the fetch agent. The definition is given by Figure 37.

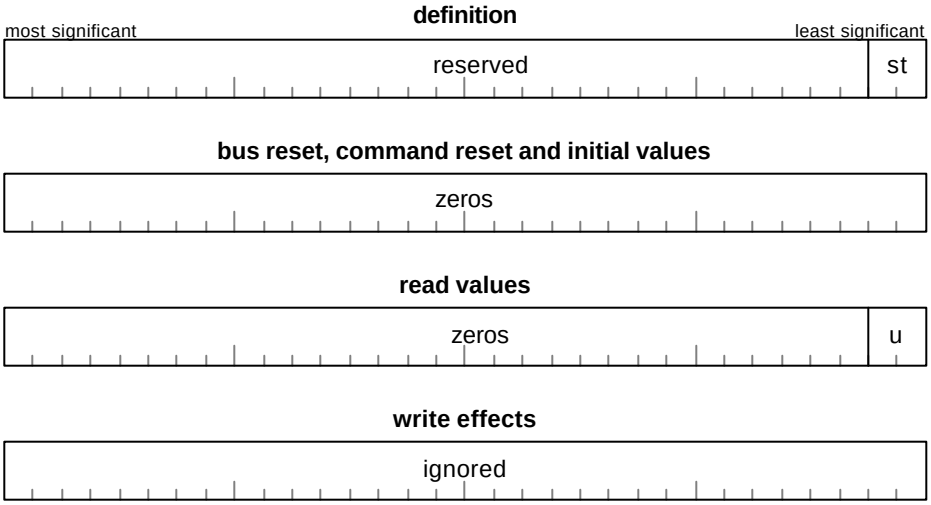

Figure 37 – AGENT_STATE format

The *st* field shall contain the current operational state of the fetch agent, as encoded by the values in the table below.

Value	Fetch agent state
0	RESET
1	ACTIVE
2	SUSPENDED
3	DEAD

6.6.3 AGENT_RESET register

The AGENT_RESET register permits an initiator to reset the operational state of a logical unit fetch agent. The definition of this write-only register is given by Figure 38.

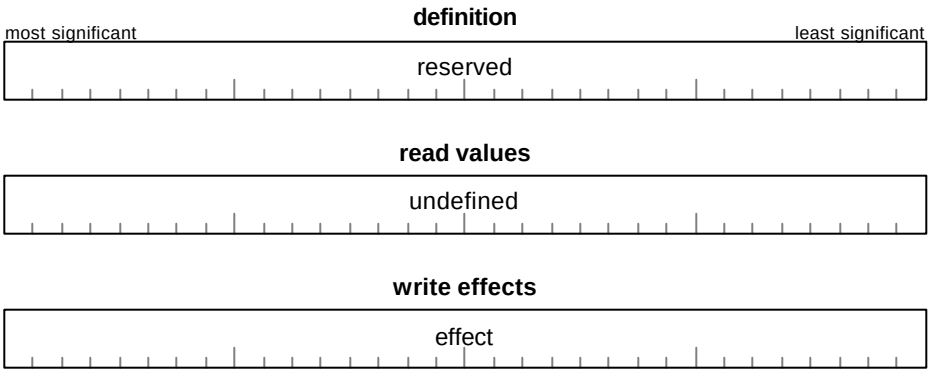

Figure 38 – AGENT_RESET format

A quadlet write of any value to this register shall cause all the fetch agent's CSRs to be reset to their initial values, after which the fetch agent shall ~~transition to~~ enter the reset state.

6.6.4 ORB_POINTER register

The ORB_POINTER register contains the address of an ORB in system memory. This register shall support 8-byte block read and block write requests whose *destination_offset* is equal to the address of the ORB_POINTER register and shall reject quadlet write requests and all other block read and block write requests. The definition is given by Figure 39.

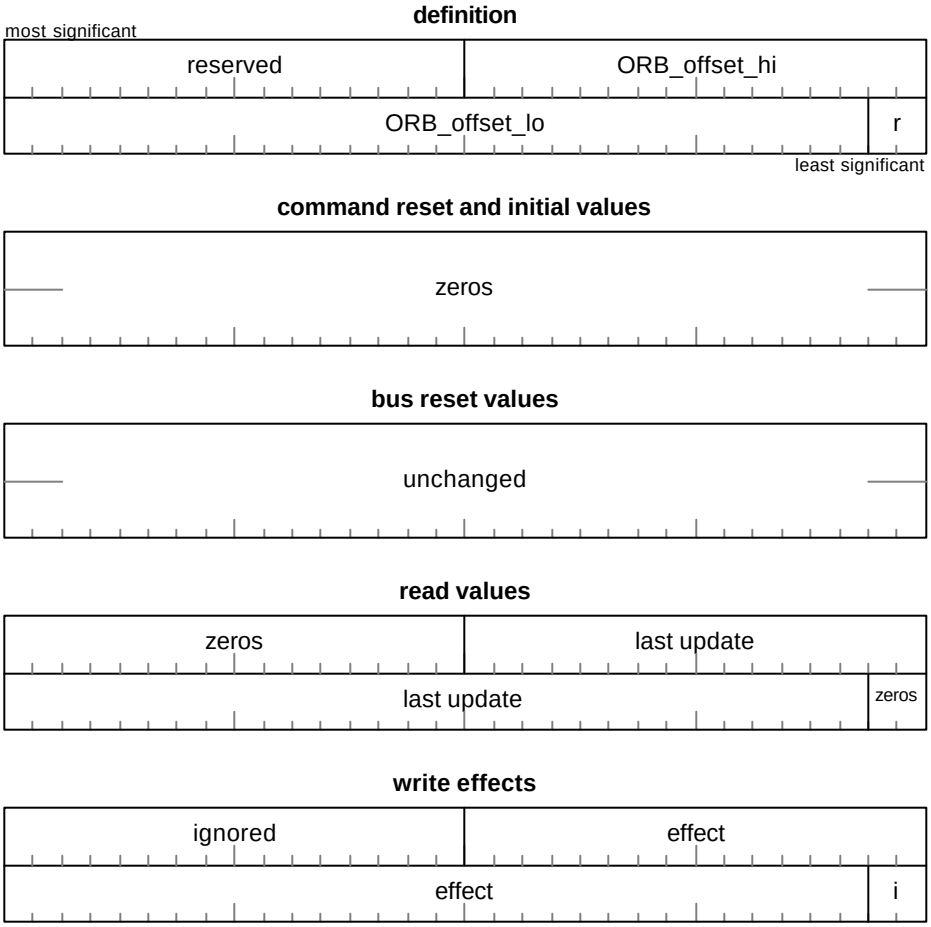

Figure 39 – ORB_POINTER format

The *ORB_offset_hi* and *ORB_offset_lo* fields together form an *ORB_offset* field. The Serial Bus address used to fetch the referenced ORB shall be formed from the concatenation of the 16-bit node ID of the initiator (available to the target as a result of a login or reconnect), the *ORB_offset* field and two least significant bits of zero.

The effects of a write transaction to the ORB_POINTER register are dependent upon the value of *st* in the AGENT_STATE register. If the target fetch agent is in the DEAD state, writes to the ORB_POINTER register shall be ignored. If the fetch agent is in the RESET or SUSPENDED state, a write to this register shall cause the *ORB_offset* to be stored and the agent to ~~transition to~~ enter the ACTIVE state. If the fetch agent is in the ACTIVE state, a write to the ORB_POINTER register may cause unpredictable target behavior.

6.6.5 DOORBELL register

The DOORBELL register provides a means by which the initiator signals the target that a linked list of requests has been updated. The definition of this write-only register is given by Figure 40.

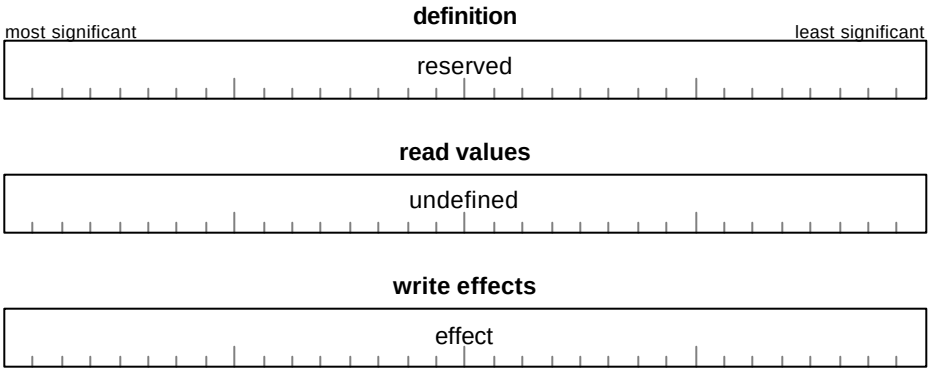

Figure 40 – DOORBELL format

A quadlet write of any value to this register shall cause the fetch agent's *doorbell* variable to be set to one.

6.6.6 UNSOLICITED_STATUS_ENABLE register

The UNSOLICITED_STATUS_ENABLE register provides a means by which the initiator may grant the logical unit permission to store an unsolicited status block. The definition of this write-only register is given by Figure 41.

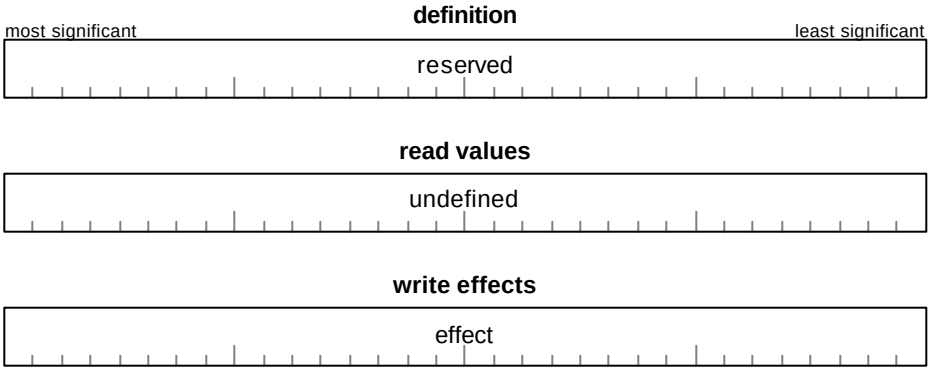

Figure 41 – UNSOLICITED_STATUS_ENABLE format

A quadlet write of any value to this register shall cause the fetch agent's *unsolicited status enabled* variable to be set to one. A successful login or create task set request shall zero the *unsolicited status enabled* variable. As described in 9.7, any time a logical unit stores an unsolicited status block it shall zero the *unsolicited status enabled* variable for the associated task set. Before the logical unit may store a subsequent unsolicited status block for the same task set, it is necessary for the initiator to write to the UNSOLICITED_STATUS_ENABLE register.

6.6.7 HEARTBEAT_MONITOR register

The HEARTBEAT_MONITOR register provides a means by which the initiator signals the logical unit to maintain the bridge-aware login associated with the register even if the logical unit is idle (*i.e.*, its task set is empty). The definition of this write-only register is given by Figure 42.

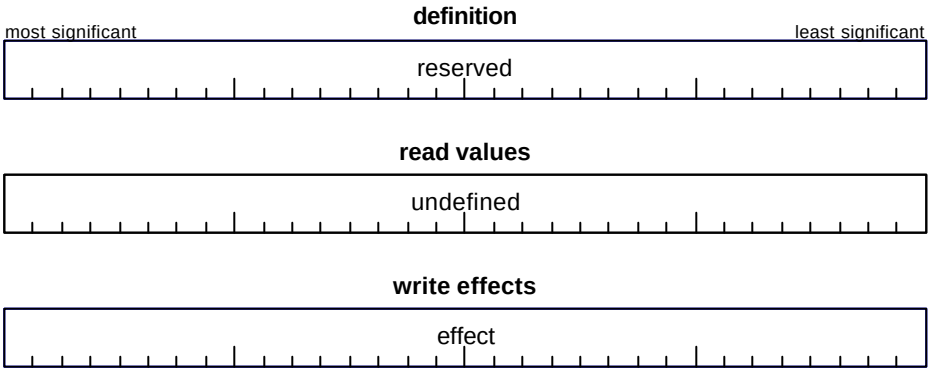

Figure 42 – HEARTBEAT_MONITOR format

Unless the login associated with this register is awaiting reconnection (see 8.6), a quadlet write of any value to this register shall cause the *heartbeat_timeout* variable in the login descriptor associated with the register to be set to the *reconnect_hold* time obtained from the same login descriptor (see 8.2). Otherwise, the target shall reject the request with a type error response.

6.6.8 FAST_START register

The FAST_START register permits an initiator to signal a new task to an idle fetch agent by means of a single block write request addressed to the register. This write-only register shall support block write requests whose *destination_offset* is equal to the address of the FAST_START register and whose *data_length* is a multiple of four and less than or equal to the vendor-dependent size of the register (see 7.8.12) but shall reject all other requests. The format of this register is illustrated below.

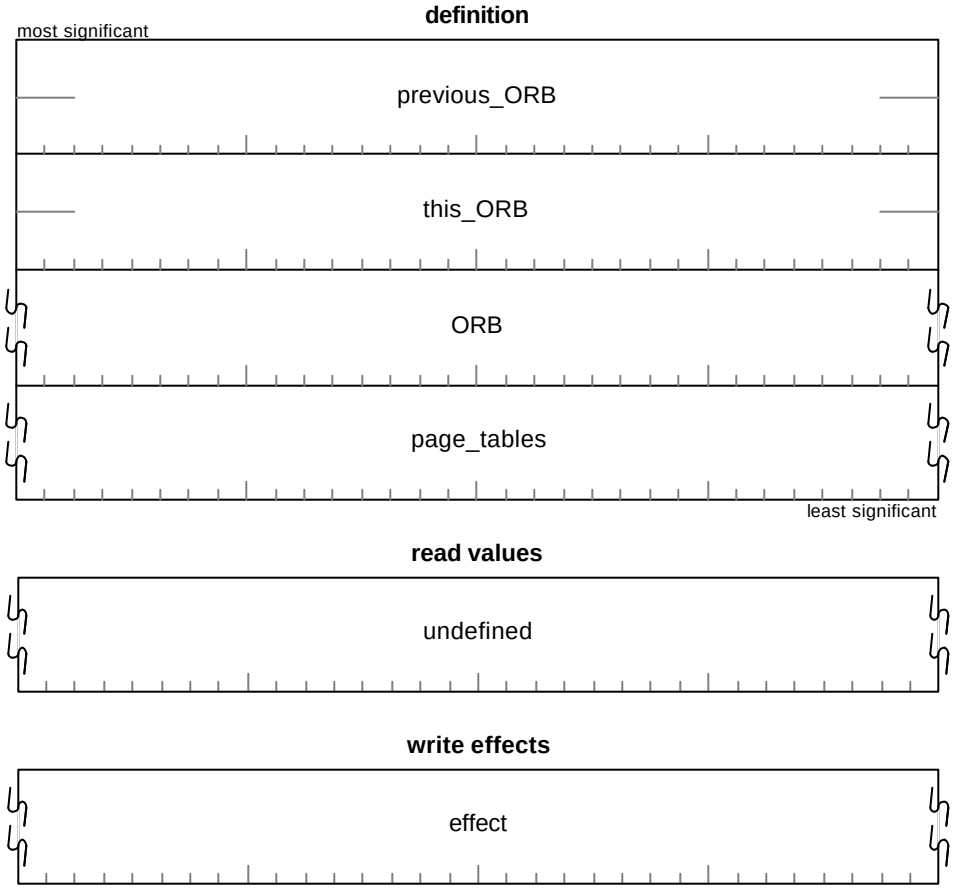

Figure 43 – FAST_START format

The *previous_ORB* field shall conform to the ORB pointer format illustrated by Figure 11 and shall either be a null pointer or reference an ORB in initiator memory whose *next*_ORB field is equal to the *this*_ORB field in the block write request addressed to the FAST_START register. When *previous_ORB* is not a null pointer, the ORB's Serial Bus address shall be formed from the concatenation of the 16-bit node ID of the initiator (available to the target as a result of login or reconnect) and the least significant 48 bits of the *previous_ORB* field.

The *this_ORB* field shall conform to the ORB pointer format illustrated by Figure 11 and shall contain the address of an ORB in initiator memory whose contents are identical to the *ORB* field in the block write request addressed to the FAST_START register. The ORB's Serial Bus address shall be formed from the concatenation of the 16-bit node ID of the initiator (available to the target as a result of a login or reconnect) and the least significant 48 bits of the *this_ORB* field.

The *ORB* field shall contain an ORB whose format conforms to those specified by 5.2. An initiator shall not address a block write request to the FAST_START register whose *data_length*, in bytes, is less than sixteen plus the size of the *ORB* field.[3] The target shall reject a block write request addressed to the FAST_START register if its *data_length*, in bytes, is less than sixteen plus the size of the *ORB* field.

The *page_tables* field, if present, shall immediately follow the *ORB* field. If the format of the *ORB* field includes nonzero *page_table_present* bits, the *page_tables* field may contain zero or more page table

[3] The size of the ORBs used by a target is fixed by the Unit_Characteristics configuration ROM entry.

entries. The target shall derive the number of immediately available page table entries from the *data_length* of the block write request addressed to the FAST_START. The number of page table entries is limited by the maximum size of the FAST_START register. The *page_tables* field may be a subset of the page tables specified by the ORB referenced by *this_ORB*, but no partial page table entries shall be present (see 5.3). The order and content of the entries shall be determined as follows:

- If *page_table_present*[0] is nonzero, the order and content of the first *n* page table entries in the *page_tables* field shall be identical to those contained within *page_table*[0], where *n* is the smaller of the number of entries in *page_table*[0] or the total number of entries in the *page_tables* field;
- If *page_table_present*[0] is zero and *page_table_present*[1] is nonzero, the order and content of the first *n* page table entries in the *page_tables* field shall be identical to those contained within *page_table*[1], where *n* is the smaller of the number of entries in *page_table*[1] or the total number of entries in the *page_tables* field; or
- If both *page_table_present*[0] and *page_table_present*[1] are nonzero, the *page*_tables field shall not contain any entries from *page_table*[1] unless *page_table*[0] is present in its entirety in the *page_tables* field. In this case, the order and content of the first page table entries in the *page_tables* field shall be identical to those contained within *page_table*[0] and may be followed by *n* entries whose order and content shall be identical to those contained within *page_table*[1], where *n* is the smaller of the number of entries in *page_table*[1] or the total number of entries in the *page_tables* field less the number of entries in *page_table*[0];

The effects of a write transaction to the FAST_START register are dependent upon the value of its *previous_ORB* field and the value of *st* in the associated AGENT_STATE register. If the fetch agent is in the DEAD state, writes to the FAST_START register shall be ignored. If the fetch agent is in the ACTIVE state, a write to the FAST_START register shall be interpreted as if it were a quadlet write request addressed to the fetch agent's DOORBELL register (the data payload shall be ignored). Otherwise, when the fetch agent is in the RESET or SUSPENDED state, the value of the *previous_ORB* field determines the effect of a write to this register. If *previous_ORB* contains a null pointer, *this_ORB* shall be stored in the associated ORB_POINTER register, the *ORB* and *page_tables* fields shall be stored in the target's working set and the agent shall ~~transition to~~ enter the ACTIVE state. When *previous_ORB* is not null, the target shall perform these actions if and only if *previous_ORB* is equal to the fetch agent's ORB_POINTER register. See 9.2.6 for a precise definition of fetch agent state transitions that involve the FAST_START register.

7 Configuration ROM

7.1 Configuration ROM hierarchy

All nodes that implement SBP-3 targets shall implement general format configuration ROM in accordance with IEEE Std 1212-2001, IEEE 1394 and this standard. General format configuration ROM is a self-descriptive structure as illustrated below. The bus information block and root directory are at fixed locations; all other directories and leaves are addressed by entries in their parent directory.

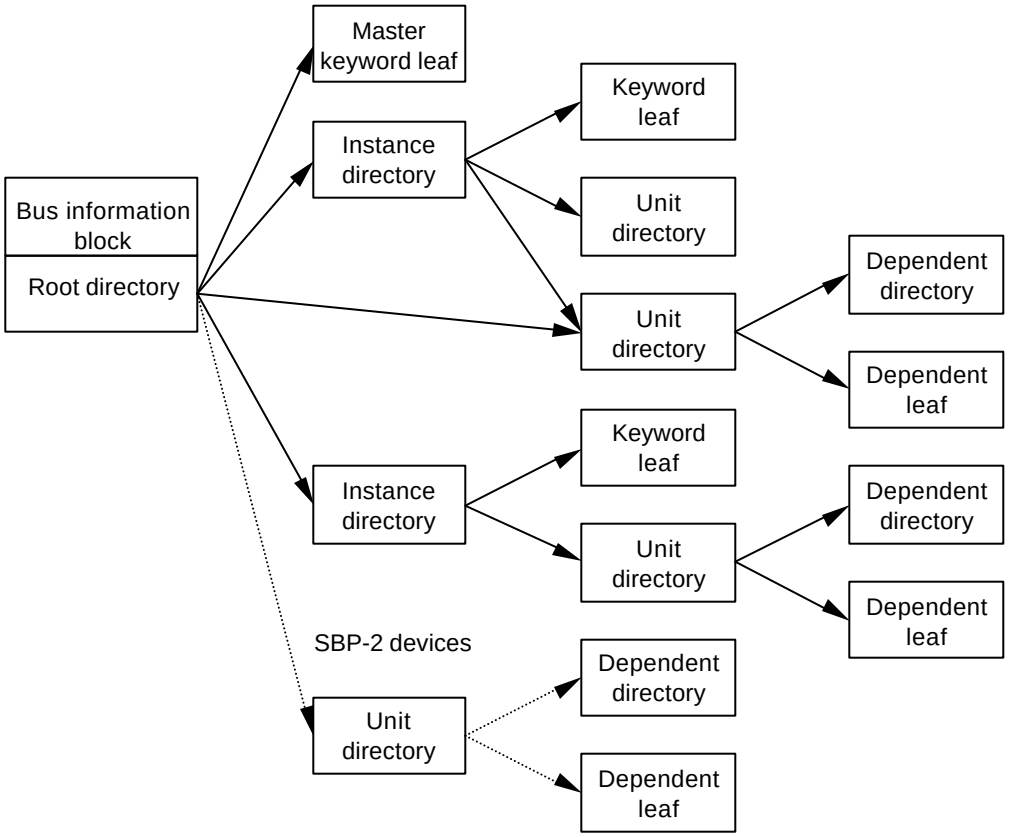

Figure 44 – Configuration ROM hierarchy

The figure above shows the potential of the general ROM format to accommodate a diversity of directory and leaf entries in a tree structure. In practice a target need implement only a portion of the entries shown above.

Two of the data structures illustrated, instance directories and keyword leaves, are first defined by IEEE Std 1212-2001. Instance directories provide high-level information about particular instantiations of functions while unit directories identify the software interface (i.e., device driver) used to access each separately controllable function. The instance directories are intended to present “thumbnails” of the devices by means of subsidiary keyword leaves—descriptions, such as “DISK” or “PRINTER” that have cognitive resonance for the human user. The unit directories are complementary to the instance directories; there may be more than one unit directory for a particular device instance, each of which specifies a different software interface for the same device.

NOTE – One example of a device that supports more than one unit architecture is a disk capable of both AV/C operations and SBP-3. See Annex F for examples of configuration ROM.

SBP-2 unit directories are always direct offspring of the root directory; they may be accessible *via* instance directories but are not required to be (as shown in the shaded area). Devices manufactured since the development of SBP-3 should not use this earlier style; each unit directory should be the child of an instance directory. Even when this is the case, configuration ROM compliant with this revised standard may address unit directories directly from the root (in addition to accessibility from intermediate instance directories), but this style is discouraged except as necessary to accommodate legacy device discovery software.

7.2 Power reset initialization

During the initialization process that follows a power reset a target ~~may~~ might not be able to respond to Serial Bus request subactions addressed to parts of configuration ROM. When the target has insufficient information to make more than the first quadlet of configuration ROM accessible, it shall return a data value of zero in the response to any read request addressed to FFFF F000 0400_{16} or acknowledge the request subaction with *ack_tardy*, as specified by IEEE 1394. Until the initialization process completes, responses to requests addressed to other parts of configuration ROM are unspecified.

Targets shall complete initialization within five seconds of a power reset. Once power reset initialization completes, the target shall make all mandatory configuration ROM entries available. The target should not initiate a Serial Bus reset solely as a consequence of the completion of power reset initialization.

Optional configuration ROM information, such as textual descriptor leaves that identify the target vendor and model, ~~may~~ might not be available when power reset initialization completes. The target may add this information to configuration ROM as it becomes available and may initiate a Serial Bus reset to alert other nodes to the changed configuration ROM. The target should initiate a Serial Bus reset if there is no expectation that other nodes would otherwise become aware of changed configuration ROM. Prior to the bus reset, the *generation* field in the bus information block should be changed; see IEEE Std 1394a-2000 for details.

7.3 Bus information block

All targets shall implement a bus information block at a base address of FFFF F000 0404_{16}. For convenience of reference, the format of the bus information block defined by IEEE 1394 and draft standard IEEE P1394.1 is reproduced below. The current version of the referenced standards and supplements shall be consulted for the most recent information.

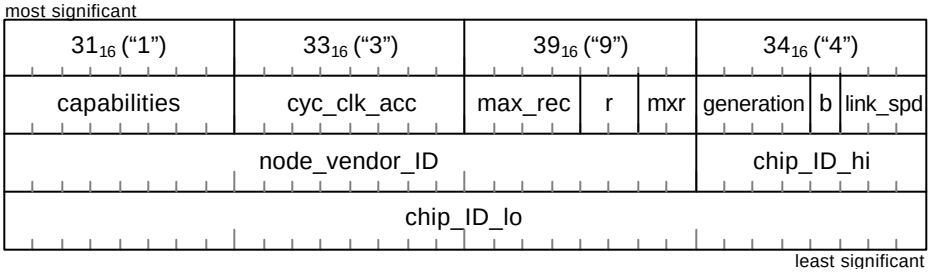

Figure 45 – Bus information block format

The first quadlet contains the string “1394” in ASCII characters.

The *capabilities* field is a collection of bits, illustrated by Figure 46.

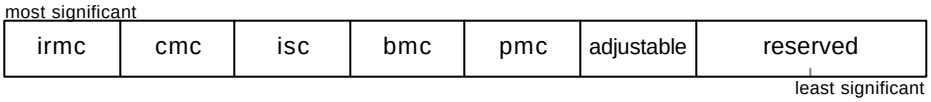

Figure 46 – Bus information block *capabilities* field

The *irmc* bit shall be one if the node is isochronous resource manager-capable; otherwise, this bit shall be zero.

The *cmc* bit shall be one if the node is cycle master-capable; otherwise, this bit shall be zero.

The *isc* bit shall be one if the node supports isochronous operations; otherwise, this bit shall be zero.

The *bmc* bit shall be one if the node is bus manager-capable; otherwise, this bit shall be zero.

The *pmc* bit shall be one if the node is power manager-capable; otherwise, this bit shall be zero. A node that reports a value of one for *pmc* shall also set *bmc* to one.

The *adjustable* bit shall be one of the node's cycle offset is adjustable, as specified by draft standard IEEE P1394.1; otherwise, this bit shall be zero.

The *cyc_clk_acc* field specifies the node's cycle master clock accuracy in parts per million. If the *cmc* bit is one, the value in this field shall be between zero and 100. If the *cmc* bit is zero, this field shall be all ones.

The *max_rec* field defines the maximum data payload size that the target supports. The data payload size applies to block write requests addressed to the target, asynchronous stream packets received by the target and block read responses transmitted by the target. The maximum data payload is equal to $2^{max\_rec + 1}$ bytes. The *max_rec* field does not place any limits on the maximum payload size of block write requests or block read responses that the target may transmit or receive, respectively. If *max_ROM* is nonzero, *max_rec* shall be greater than or equal to $2^{max\_ROM + 1} + 1$.

The *max_ROM* field (abbreviated as *mxr* in the figure above) shall specify the size and alignment of read requests supported by configuration ROM, whether within the address range FFFF F000 0400_{16} through FFFF F000 $07FF_{16}$ inclusive or another portion of the target's address space, as specified by IEEE Std 1394a-2000.

The *generation* field indicates changes in configuration ROM; see IEEE Std 1394a-2000 for details.

The *bridge_aware* bit (abbreviated as *b* in the figure above), shall be one if the target complies with the requirements of draft standard IEEE P1394.1 for a bridge-aware device; otherwise, this bit shall be zero.

The *lnk_spd* field shall report the maximum speed capability of the target's link layer, encoded as specified by Table 1.

The *node_vendor_ID* field shall be uniquely assigned by the IEEE RAC, as specified by IEEE Std 1212-2001. Unique identifiers for a company or organization may be obtained from:

Institute of Electrical and Electronic Engineers, Inc.
Registration Authority Committee
445 Hoes Lane
Piscataway, NJ 08855-1331

Application for a unique identifier (also known as a *company_ID*) may also be made *via* the Internet at http://standards.ieee.org/regauth/oui/forms/.

The *chip_ID_hi* and *chip_ID_lo* fields are concatenated to form a 40-bit chip ID value. The vendor or organization specified by *node_vendor_ID* shall administer the chip ID values. When appended to the *node_vendor_ID* value, these shall form a unique 64-bit value called the EUI-64 (Extended Unique Identifier, 64 bits). The EUI-64 is also referred to as the node unique ID. Because physical IDs on Serial Bus may change after a bus reset, this unique identifier is the only reliable method of node identification.

7.4 Root directory

7.4.1 Root directory (general)

Configuration ROM for targets shall contain a root directory. The root directory immediately follows the bus information block and has a base address of FFFF F000 0414_{16}. The root directory shall contain one each of Vendor_ID, Node_Capabilities and Keyword_Leaf entries.

The root directory should contain an Instance_Directory entry that specifies the location of an instance directory in the format specified by this standard.

The root directory may also contain Unit_Directory entries that specify the location of unit directories whose format is specified by this standard.

7.4.2 Vendor_ID entry

The Vendor_ID entry is an immediate entry in the root directory that provides the company ID of the vendor that manufactured the module. Figure 47 shows the format of this entry.

Figure 47 – Vendor_ID entry format

03_{16} is the concatenation of *key_type* and *key_value* for the Vendor_ID entry.

The IEEE RAC uniquely assigns the *vendor_ID* to each module vendor, as specified by IEEE Std 1212-2001. There is no requirement that the values of *vendor_ID* and *node_vendor_ID* be equal.

NOTE – A recommended convention to provide vendor identification in displayable form is to immediately follow the Vendor_ID entry with a textual descriptor leaf entry. This associates an ASCII string with the module vendor. See IEEE Std 1212-2001 for the specification of textual descriptor leaves; examples are given in Annex F.

7.4.3 Node_Capabilities entry

The Node_Capabilities entry is an immediate entry in the root directory that describes node capabilities. Figure 48 shows the format of this entry.

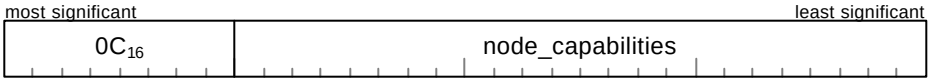

Figure 48 – Node_Capabilities entry format

$0C_{16}$ is the concatenation of *key_type* and *key_value* for the Node_Capabilities entry.

The *node_capabilities* field contains subfields, specified by IEEE 1394. Targets shall implement the SPLIT_TIMEOUT register, the 64-bit fixed addressing scheme, the STATE_CLEAR.*lost* bit and the STATE_CLEAR.*dreq* bit and indicate this by setting the *spt*, *64*, *fix*, *lst* and *drq* bits to one. If no other *node_capabilities* bits are one this results in a value of 00 $83C0_{16}$.

7.4.4 Keyword_Leaf entry

The Keyword_Leaf entry is a directory entry in the root directory or an instance directory that describes the location of a keyword leaf within configuration ROM. Figure 49 shows the format of this entry.

Figure 49 – Keyword_Leaf entry format

99_{16} is the concatenation of *key_type* and *key_value* for the Keyword_Leaf entry.

The *indirect_offset* field specifies the number of quadlets from the address of the Keyword_Leaf entry to the address of the keyword leaf within configuration ROM.

Configuration ROM for targets shall contain a Keyword_Leaf entry in the root directory that describes the location of the master keyword leaf, which shall contain the union of all keywords present in all the keyword leaves in the target's configuration ROM. The master keyword leaf shall contain the keyword "SBP" and may contain other keywords.

NOTE – A device that implements only one keyword leaf may reuse the leaf as the master keyword leaf by referencing it from both the root directory and an instance directory.

7.4.5 Instance_Directory entry

The Instance_Directory entry is a directory entry in the root directory or an instance directory that describes the location of an instance directory within configuration ROM. Figure 50 shows the format of this entry.

Figure 50 – Instance_Directory entry format

$D8_{16}$ is the concatenation of *key_type* and *key_value* for the Instance_Directory entry.

The *indirect_offset* field specifies the number of quadlets from the address of the Instance_Directory entry to the address of the instance directory within configuration ROM.

7.4.6 Unit_Directory entry

The Unit_Directory entry is a directory entry in the root directory or an instance directory that describes the location of a unit directory within configuration ROM. There may be more than one unit directory; each unit directory shall be located by a separate Unit_Directory entry. Figure 51 shows the format of this entry.

Figure 51 – Unit_Directory entry format

$D1_{16}$ is the concatenation of *key_type* and *key_value* for the Unit_Directory entry.

The *indirect_offset* field specifies the number of quadlets from the address of the Unit_Directory entry to the address of the unit directory within configuration ROM.

NOTE – IEEE Std 1212-2001 ~~[B5]~~ recommends that a unit directory be referenced from an instance directory and not be directly dependent from the root directory. The presence of a Unit_Directory entry in the root directory is strongly discouraged unless essential for compatibility with legacy (*i.e.*, SBP-2) initiators. Even in cases where a Unit_Directory entry is placed in the root, the unit directory should be accessible *via* an instance directory.

7.5 Instance directory

Configuration ROM for targets should contain at least one instance directory, in the format specified by IEEE Std 1212-2001, which contains a Unit_Directory entry that specifies the location of a unit directory in the format specified by this standard. Such an instance directory shall contain a Keyword_Leaf entry that specifies the location of a keyword leaf that contains at least the keyword “SBP”.

NOTE – Instance directories are described in more detail in IEEE Std 1212-2001 but may be summarized as follows. An instance directory characterizes the functions of a particular device instantiation (a physical instance) within a node. A single physical instance of a device function, *e.g.*, a printer, may be controllable by different software protocols, which are represented by one or more unit directories dependent from the instance directory. Instance directories may also provide a hierarchical structure that relates different functional components of a device. For example, a DVD-ROM changer and player controllable as a single unit might also be controlled separately as a changer unit and a player unit. The control method chosen might depend on the software capabilities of the initiator; the flexibility of configuration ROM instance directories permits initiators to discover the appropriate units. The DVD-ROM would be described by an instance directory that referenced a dependent unit directory for the combined changer and player functions—but the same instance directory would reference two dependent instance directories, one for the changer and the other for the player. These dependent instance directories would reference the unit directories for the separate functions.

7.6 Unit directory

Configuration ROM for targets shall contain at least one unit directory in the format specified by this standard. The unit directory shall contain Specifier_ID and Version entries, as specified by IEEE Std 1212-2001, and Management_Agent and Unit_Characteristics entries, as specified by this standard.

Targets shall implement at least one logical unit: logical unit zero. Additional logical units may be implemented. A logical unit is described by entries in the unit directory or by entries in a logical unit directory dependent upon the unit directory or by entries taken in combination from both places. The

properties of logical units are established by Command_Set_Spec_ID, Command_Set, Command_Set_Revision and Fast_Start entries; an instance of a specific logical unit is established by a Logical_Unit_Number entry.

7.7 Logical unit directory

The logical unit directory provides one of two methods by which a logical unit implemented by the target may be described (the other is a Logical_Unit_Number entry in the unit directory, described in 7.8.15).

A logical unit directory shall contain exactly one Logical_Unit_Number entry. It may contain additional entries permitted by 7.8. Some of these entries may be inherited from the parent unit directory but if an entry is present in both directories the entry in the logical unit directory shall take precedence.

7.8 Directory entries

7.8.1 Directory entries summary

This standard defines configuration ROM entries that may appear in a unit directory or logical unit directories dependent upon the unit directory or both, as specified by the table below.

Directory entry	Unit directory	Logical unit directory	Inherited
Specifier_ID	Required	Prohibited	
Version	Required	Prohibited	
Revision	Optional	Prohibited	
Command_Set_Spec_ID	Optional [4]	Optional [4]	Yes
Command_Set	Optional [4]	Optional [4]	Yes
Command_Set_Revision	Optional	Optional	Yes
Firmware_Revision	Optional	Optional	No
Management_Agent	Required	Prohibited	
Unit_Characteristics	Required	Prohibited	
Reconnect_Timeout	Optional	Prohibited	
Fast_Start	Optional	Optional	Yes
Plug_Control_Register	Optional	Optional	Yes
Logical_Unit_Directory	Optional	Prohibited	
Logical_Unit_Number	Optional [5]	Required	
Unit_Unique_ID	Optional	Prohibited	

For entries that may appear in a logical unit directory, the rightmost column in the table specifies whether or not the value of the entry is implicitly inherited from the parent unit directory if it is not present in the logical unit directory. In addition to the directory entries described above, unit directories and any of their dependent directories may contain entries permitted by IEEE Std 1212-2001.

[4] These entries ~~may~~ shall not be omitted altogether but shall be present in one of the combinations described below the table.

[5] A target shall have at least one Logical_Unit_Number entry, whether in a unit directory or a logical unit directory.

The command set of each of a target's logical units shall be identified by either explicit or inherited values of Command_Set_Spec_ID and Command_Set entries. If the unit directory contains one or more Logical_Unit_Number entries, both entries shall be present in the unit directory. If the unit directory contains one or more Logical_Unit_Directory entries, the logical units defined in each directory may inherit their command set from the Command_Set_Spec_ID and Command_Set entries in the parent unit directory or these entries may be present in the logical unit directory. If either of these entries is omitted from a logical unit directory, it shall be present in the parent unit directory.

7.8.2 Specifier_ID entry

The Specifier_ID entry is an immediate entry in the unit directory that specifies the organization responsible for the architectural definition of the target. Figure 52 shows the format of this entry.

Figure 52 – Specifier_ID entry format

12_{16} is the concatenation of *key_type* and *key_value* for the Specifier_ID entry.

00 $609E_{16}$ is the company ID obtained by INCITS from the IEEE RAC. The value indicates that the INCITS Secretariat and its Technical Committee T10 are responsible for the maintenance of this standard.

7.8.3 Version entry

The Version entry is an immediate entry in the unit directory that, in combination with the company ID obtained from the Specifier_ID entry, specifies the software interface of the target. Figure 53 shows the format of this entry.

Figure 53 – Version entry format

13_{16} is the concatenation of *key_type* and *key_value* for the Version entry.

01 0483_{16} indicates that the target conforms to ANSI NCITS 325-1998, Serial Bus Protocol 2 (SBP-2) and may additionally conform to this standard.

7.8.4 Revision entry

The Revision entry is an immediate entry in the unit directory that, in combination with the company ID obtained from the Specifier_ID entry and the version obtained from the Version entry, specifies the software interface of the target. Figure 54 shows the format of this entry.

Figure 54 – Revision entry format

21_{16} is the concatenation of *key_type* and *key_value* for the Revision entry.

The *revision* field shall be zero or one. A value of zero indicates conformance with ANSI NCITS 325-1998 while a value of one indicates conformance with this standard. If the Revision entry is omitted from the unit directory, the value of *revision* is implicitly zero.

7.8.5 Command_Set_Spec_ID entry

The Command_Set_Spec_ID entry is an immediate entry that, when present in either the unit directory or a dependent logical unit directory, specifies the organization responsible for the command set definition for the logical units. Figure 55 shows the format of this entry.

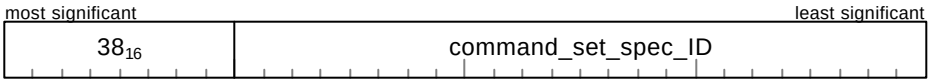

Figure 55 – Command_Set_Spec_ID entry format

38_{16} is the concatenation of *key_type* and *key_value* for the Command_Set_Spec_ID entry.

The *command_set_spec_ID* is an organizationally unique identifier obtained from the IEEE RAC. The organization to which this 24-bit identifier has been granted is responsible for the definition of the command set implemented by the target.

7.8.6 Command_Set entry

The Command_Set entry is an immediate entry that, when present in either the unit directory or a dependent logical unit directory, in combination with the *command_set_spec_ID* specifies the command set implemented by the logical unit. Figure 56 shows the format of this entry.

Figure 56 – Command_Set entry format

39_{16} is the concatenation of *key_type* and *key_value* for the Command_Set entry.

The value of *command_set* shall be specified by the owner of *command_set_spec_ID*.

7.8.7 Command_Set_Revision entry

The Command_Set_Revision entry is an immediate entry that, when present in either the unit directory or a dependent logical unit directory, specifies the revision level of the command set implemented by the logical unit. Figure 57 shows the format of this entry.

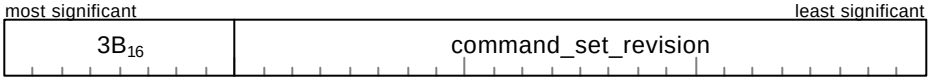

Figure 57 – Command_Set_Revision entry format

$3B_{16}$ is the concatenation of *key_type* and *key_value* for the Command_Set_Revision entry.

The value of *command_set_revision* shall be specified by the owner of *command_set_spec_ID*.

7.8.8 Firmware_Revision entry

The Firmware_Revision entry is an immediate entry that, when present in the unit directory or a logical unit directory, specifies the firmware revision level implemented by the target or logical unit, respectively. Figure 58 shows the format of this entry.

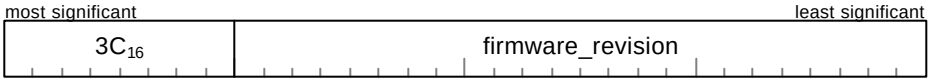

Figure 58 – Firmware_Revision entry format

$3C_{16}$ is the concatenation of *key_type* and *key_value* for the Firmware_Revision entry.

The value of *firmware_revision* shall be specified by the organization granted the 24-bit identifier obtained from the Vendor_ID entry in the unit directory or, if there is no such entry, from the Vendor_ID entry in the root directory.

NOTE – It is meaningful for different vendor IDs to be specified in the root directory and the unit directory. In the case of a product that incorporates devices native to another transport (*e.g.*, parallel SCSI), the vendor ID in the unit directory might identify the original equipment manufacturer of the interface chip and firmware that adapts SBP to the other transport, while the vendor ID in the root directory might identify the product integrator.

7.8.9 Management_Agent entry

The Management_Agent entry is an immediate entry in the unit directory that specifies the base address of the target's MANAGEMENT_AGENT register. Figure 59 shows the format of this entry.

Figure 59 – Management_Agent entry format

54_{16} is the concatenation of *key_type* and *key_value* for the Management_Agent entry.

The *csr_offset* field shall contain the offset, in quadlets, from the base address of register space, FFFF F000 0000_{16}, to the base address of the MANAGEMENT_AGENT register for the target. All target CSRs shall be located at or above address FFFF F001 0000_{16}; therefore the value of *csr_offset* shall not be less than 4000_{16}.

NOTE – If a device implements additional control and status registers that are dependent upon the device class, it is recommended that these registers be placed at one of two locations within the device's address space. If the additional registers pertain to a logical unit, the recommended locations are at offset ~~20_{16}~~ 40_{16} and above following the base address of the logical unit's command block agent registers. Additional registers that are associated with the device, and not a particular logical unit, may be located immediately after the MANAGEMENT_AGENT register. If this convention is followed, there is no necessity for additional configuration ROM entries to describe the location of device-dependent registers.

7.8.10 Unit_Characteristics entry

The Unit_Characteristics entry is an immediate entry in the unit directory which specifies characteristics of the target implementation. Figure 60 shows the format of this entry.

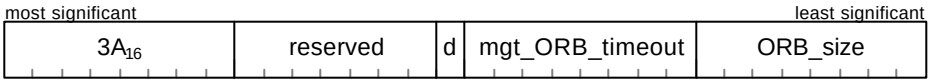

Figure 60 – Unit_Characteristics entry format

$3A_{16}$ is the concatenation of *key_type* and *key_value* for the Unit_Characteristics entry.

When the *distributed_data* bit (abbreviated as *d* in the figure above) is one, the target supports node selectors within page tables (see 5.3.4). This permits the initiator to distribute the data buffer among one or more nodes independent of each other or the node that contains the page table. Otherwise, when *distributed_data* is zero, there is no target support for node selectors and the page table and data buffer shall be located within the same node.

The *mgt_ORB_timeout* field shall specify, in units of 500 milliseconds, the maximum time permitted for a target to store a status block in response to a management ORB. After this time, a target shall not store a status block for the management ORB and an initiator shall indicate that the management ORB has timed out. For the initiator, the time-out commences when either an *ack_complete* or *resp_complete* subaction is received in response to the block write of the management ORB address to the MANAGEMENT_AGENT register. The target starts the time-out period when the *ack_complete* or *resp_complete* subaction is transmitted.

NOTE – An initiator that attempts retry of an expired management ORB should either a) wait at least a split time-out period after the management ORB time-out or b) signal the management ORB from a different address in system memory.

The *ORB_size* field shall specify, in quadlets, the largest read request size used by any of the target's logical unit fetch agents to obtain ORBs from initiator memory. The initiator shall allocate, on a quadlet aligned boundary, at least this much memory for each ORB signaled to the target. The *ORB_size* field does not apply to management ORBs (see 5.2.4), whose size is fixed at 32 bytes.

7.8.11 Reconnect_Timeout entry

The Reconnect_Timeout entry is an optional entry in the unit directory that describes the maximum reconnect timeout supported by the target. Figure 61 shows the format of this entry.

Figure 61 – Reconnect_Timeout entry format

$3D_{16}$ is the concatenation of *key_type* and *key_value* for the Reconnect_Timeout entry.

The *max_reconnect_hold* field specifies the maximum value of *reconnect_hold* that the target may return in login response data (see 5.2.4.2). If this entry is not present in configuration ROM either the target does not include *reconnect_hold* in login response data or the value returned is always zero.

7.8.12 Fast_Start entry

The Fast_Start entry is an optional entry in either the unit directory or a dependent logical unit directory that, if present (or inherited), identifies logical unit implementation of the FAST_START register specified by 6.6.8. Figure 62 shows the format of this entry.

Figure 62 – Fast_Start entry format

$3E_{16}$ is the concatenation of *key_type* and *key_value* for the Fast_Start entry.

The *max_payload* field shall specify the maximum *data_length* value that may be used in a block write request addressed to the FAST_START register. A zero value indicates that the maximum payload is constrained by the *max_rec* field in the target's bus information block. Otherwise, a nonzero value represents the maximum payload size, in quadlets, in which case *max_payload* shall be less than or equal to $2^{max\_rec + 1} / 4$.

The *FAST_START_offset* field shall specify the location of the FAST_START register relative to the base address of its associated fetch agent's CSRs, as obtained from the response returned by the target as part of a successful login or create task set request. The offset shall be specified in quadlets and shall have a minimum value of sixteen.

7.8.13 Plug_Control_Register entry

The Plug_Control_Register entry is an optional entry in either the unit directory or a dependent logical unit directory that, if present (or inherited), associates a plug control register with a logical unit. There may be more than one Plug_Control_Register entry within a unit directory or a logical unit directory. Figure 63 shows the format of this entry.

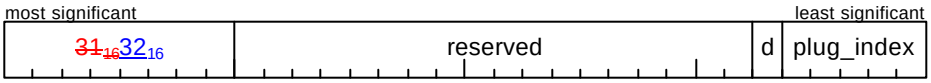

Figure 63 – Plug_Control_Register entry format

~~31_{16}~~ 32_{16} is the concatenation of *key_type* and *key_value* for the Plug_Control_Register entry.

The *direction* bit (abbreviated as *d* in the figure above) shall be zero when *plug_index* refers to an input plug control register (iPCR) and one when *plug_index* refers to an output plug control register (oPCR).

The *plug_index* field shall, in combination with the *direction* bit, identify a plug control register. The address of the plug control register, within node space, is obtained from the formula FFFF F000 0980_{16} - (128 * *direction*) + (4 * (*plug_index* +1)). Plug control registers are specified by IEC 61883-1 and draft standard IEEE P1394.1.

NOTE – When the *isochronous* bit in a command block ORB (see 5.2.3) is one, if (for the logical unit to which the ORB is signaled) there is one and only one Plug_Control_Register entry whose *direction* bit matches the *direction* bit in the ORB, that Plug_Control_Register entry identifies the plug control register from which isochronous stream information is obtained. In the case of an oPCR, this includes channel number, speed and maximum data payload; an iPCR specifies only the channel number.

7.8.14 Logical_Unit_Directory entry

The Logical_Unit_Directory entry is an optional directory entry in the unit directory that describes the location of the logical unit directory within configuration ROM. Figure 64 shows the format of this entry.

Figure 64 – Logical_Unit_Directory entry format

$D4_{16}$ is the concatenation of *key_type* and *key_value* for the Logical_Unit_Directory entry.

The *indirect_offset* field specifies the number of quadlets from the address of the Logical_Unit_Directory entry to the address of the logical unit directory within configuration ROM.

7.8.15 Logical_Unit_Number entry

The Logical_Unit_Number entry is an immediate entry that, when present in either the unit directory or a dependent logical unit directory, specifies the characteristics, peripheral device type and logical unit number of a logical unit implemented by the target. Figure 65 shows the format of this entry.

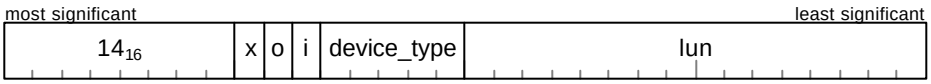

Figure 65 – Logical_Unit_Number entry format

14_{16} is the concatenation of *key_type* and *key_value* for the Logical_Unit_Number entry.

The *extended_status* bit (abbreviated as *x* in the figure above) specifies whether or not the logical unit supports extended status format (see 5.4). If the *extended_status* bit is zero, only the basic status format is supported. Otherwise, the logical unit supports both the basic and extended status formats; the logical unit shall not store status in the extended format unless enabled by the initiator during login (see 5.2.4.2).

The *ordered* bit (abbreviated as *o* in the figure above) specifies the manner in which the logical unit executes tasks signaled to the primary command block agent. If the logical unit executes and reports completion status without any ordering constraints, the *ordered* bit shall be zero. Otherwise, if the logical unit both executes all tasks in order and reports their completion status in the same order, the *ordered* bit shall be one.

The *isochronous* bit (abbreviated as *i* in the figure above) specifies whether or not the logical unit supports isochronous operations. If the *isochronous* bit is one, the *irmc*, *cmc* and *isc* bits in the bus information block shall also be one, as described in 7.3.

The *device_type* field indicates the peripheral device type implemented by the logical unit. This field shall contain a value specified by the table below.

Value	Peripheral device type
$0 – 1E_{16}$	The meaning of *device_type* is command set-dependent
$1F_{16}$	Unknown device type; command set-dependent means are necessary to determine the peripheral device type

The *lun* field shall identify the logical unit to which the information in the Logical_Unit_Number entry applies. The value of the *lun* field shall be unique within the scope of the unit directory and all dependent logical unit directories.

7.8.16 Unit_Unique_ID entry

The Unit_Unique_ID entry is an optional leaf entry in the unit directory that describes the location of the unit unique ID leaf within configuration ROM. If a vendor implements a device with multiple Serial Bus access paths, *i.e.*, multiple links to Serial Bus each of which receives a distinct *node_ID* as the result of Serial Bus initialization or bus enumeration, the Unit_Unique_ID entry shall be implemented. Figure 66 shows the format of this entry.

Figure 66 – Unit_Unique_ID entry format

$8D_{16}$ is the concatenation of *key_type* and *key_value* for the Unit_Unique_ID entry.

The *indirect_offset* field specifies the number of quadlets from the address of the Unit_Unique_ID entry to the address of the unit unique ID leaf within configuration ROM.

7.9 Unit unique ID leaf

Although the node unique ID (EUI-64) present in the bus information block is sufficient to uniquely identify nodes attached to Serial Bus, it is insufficient to identify a target when a vendor implements a device with multiple Serial Bus node connections. In this case initiator software requires information by which a particular target may be uniquely identified, regardless of the Serial Bus access path used. The figure below shows the format of the unit unique ID leaf.

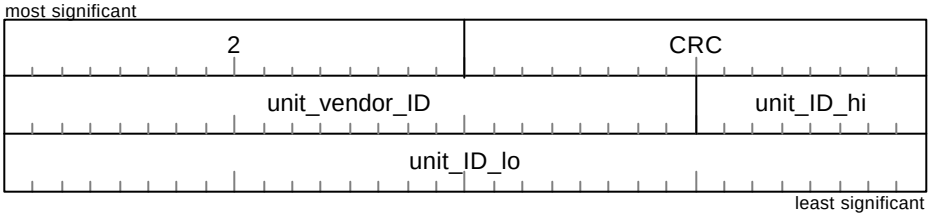

Figure 67 – Unit unique ID leaf format

The first quadlet of the unit unique leaf shall contain the number of following quadlets in the leaf and a CRC calculated for those quadlets, as specified by IEEE Std 1212-2001.

The *unit_vendor_ID* value shall be uniquely assigned by the IEEE RAC, as specified by IEEE Std 1212-2001.

The *unit_ID_hi* and *unit_ID_lo* fields are concatenated to form a 40-bit unit ID value. The vendor specified by *unit_vendor_ID* shall administer the unit ID values. When appended to the *unit_vendor_ID* value, these shall form a unique 64-bit value referred to as the unit unique ID.

As a consequence of the implementation of multiple Serial Bus nodes for the same unit, there is configuration ROM accessible for each node. Parts of these configuration ROMs shall differ from each other, *e.g.*, the node unique ID in the bus information block, but the unit unique ID shall be the same regardless of which node is used to access the information.

8 Access

8.1 Access overview

Before an initiator may signal commands to a logical unit or task management requests to a target, access privileges shall first be granted by the target. The criteria for the grant of access may include resource availability or other target requirements. This section specifies the target facilities that support access control and the methods by which an initiator requests access to a logical unit and eventually relinquishes access when it is no longer required.

When an initiator establishes bridge-aware access, it may require additional target resources to manage data transfer operations. This section specifies the methods an initiator uses to obtain or release these resources.

8.2 Access protocols

Targets shall implement a logical unit reservation protocol which may be used to guarantee single initiator access to the logical unit and to preserve that initiator's access rights across a Serial Bus reset. Targets may optionally implement the extensions to the logical unit reservation protocol specified by Annex C, which support both passwords and persistent reservations. Neither of these mechanisms preclude additional, command set-dependent methods that control access to a target's logical units.

In order to support the logical unit reservation protocol, a target shall implement resources to manage one or more logins from initiators. These resources are described below and are used in the specification of target actions in response to login requests signaled by an initiator to the target's management agent:

- The target implements a set of one or more *login_descriptors* that are used to hold context for logins. The context of a login stored in a *login_descriptor* consists of the *lun*, the *login_owner_ID*, the *login_owner_EUI_64*, the *status_FIFO* address, an *exclusive* variable, a *bridge_aware* variable, a *heartbeat_timeout* variable, the base addresses of the fetch agent CSRs, the *login_ID* to be used by the initiator to identify the login, the *initiator_node_handle* assigned by the target for a bridge-aware login and the *reconnect_hold* period guaranteed by the target—these last four are returned to the initiator in the login response data.
- The *login_owner_ID* is the 16-bit node ID, either local or global, of the current owner of a login. The target shall use the *login_owner_ID* to qualify the *source_ID* of all write requests addressed to the *login_descriptor* fetch agent CSRs. Upon a power reset, the *login_owner_ID* for all *login_descriptors* shall be initialized to all ones. At all other times, the value of *bridge_aware* determines which events cause *login_owner_ID* to be set to all ones. If *bridge_aware* is false, a Serial Bus reset causes *login_owner_ID* to be set to all ones; if *bridge_aware* is true and *login_owner_ID* contains a global node ID, a net update causes *login_owner_ID* to be set to all ones.
- The *login_owner_EUI_64* is the unique 64-bit identifier of the current owner of a login. Upon a power reset, the *login_owner_EUI_64* for all *login_descriptors* shall be initialized to all ones. After a Serial Bus reset, for those *login_*descriptors whose *bridge_aware* variable is false, the *login_owner_EUI_64* persists for *reconnect_hold* + 1 seconds and shall then be set to all ones unless the login has been reestablished. After a net update, for those *login_*descriptors whose *bridge_aware* variable is true, the *login_owner_EUI_64* persists for *reconnect_hold* + 1 seconds (measured from the end of quarantine) and shall then be set to all ones unless the login has been reestablished.

A *login_descriptor* is free if both its *login_owner_ID* and *login_owner_EUI_64* are all ones. The resources of this *login_descriptor* may be allocated to any initiator that successfully completes a login request. If a *login_descriptor*'s *login_owner_ID* is all ones but its *login_owner_EUI_64* contains a valid EUI-64, the

login_descriptor is reserved—the initiator identified by *login_owner_EUI_64* may reestablish the login. Active *login_descriptors* are those whose *login_owner_ID* and *login_owner_EUI_64* are both valid; the initiator that owns the login may signal requests to the fetch agent associated with the *login_descriptor*.

8.3 Access requests

~~The clauses that follow describe the use of the login and create task set ORBs defined in 5.1.3.1 and 5.1.3.3.~~

8.3.1 Login

Before an initiator may signal any requests to a target that either require a *login_ID* or address fetch agent CSRs, it shall first perform a login. The login request, whose format is specified in 5.2.4.2, shall be signaled to the target's MANAGEMENT_AGENT register by means of an 8-byte block write ~~transaction~~ request that specifies the Serial Bus address of the login request ORB. The address of the management agent shall be obtained from the target's configuration ROM.

Unless modified by a subsequent reconnect request, the speed at which the block write request to the MANAGEMENT_AGENT register is received shall determine the speed used by the target for all subsequent requests to read the initiator's configuration ROM, fetch ORBs from initiator memory or store status at the initiator's *status_FIFO*. Command block ORBs separately specify the speed for requests addressed to the data buffer or page table.

The login ORB shall specify the *lun* of the logical unit for which the initiator desires access.

The target shall perform the following steps, in the order specified, to validate a login request:

a) All targets that conform to this standard distinguish between local and global node IDs whether or not they implement bridge-aware capabilities. If the *source_ID* from the write request used to signal the login ORB to the target's MANAGEMENT_AGENT register contains a global node ID but the target does not implement bridge-aware capabilities, the target shall respond with a type error;

NOTE – Because bridges interdict read requests addressed to global node IDs if they are originated by nodes that are not bridge-aware, a target that is not bridge-aware is unable to fetch the login ORB and determine the address of the *status_FIFO*. Thus, the return of a type error response is the only method available to notify the initiator that the login failed.

b) In cases where *source_ID* is local, the *aware* bit is one in the login ORB and the target does not implement bridge-aware capabilities, the target shall reject the login request with an *sbp_status* of function rejected;

c) If *source_ID* contains a global node ID and the target implements bridge-aware capabilities, the target shall examine the *aware* bit in the login ORB and, if zero, shall reject the login with an *sbp_status* of function rejected. When the login specifies a global node ID and the *aware* bit is one, the target shall use a TIMEOUT request, as defined by draft standard IEEE P1394.1, to obtain the EUI-64 of the initiator and its remote timeout information;

d) Otherwise *source_ID* is local and the target shall read the initiator's unique ID, EUI-64, from the bus information block by means of two quadlet read transactions. The *source_ID* from the write transaction used to signal the login ORB to the target's MANAGEMENT_AGENT register shall be used as the *destination_ID* in the quadlet read transactions;

e) If the *update* bit in the login ORB is zero, the target shall determine whether or not the initiator already owns a login by comparing the EUI-64 just obtained against the *login_owner_EUI_64* for all *login_descriptors*. If the initiator is currently logged-in to the same logical unit, the login request shall be rejected with an *sbp_status* of access denied. Otherwise, when the *update* bit is one, the target

shall verify that the initiator owns the login identified by *login_ID* and, if not, shall reject the login request with an *sbp_status* of invalid login ID;

f) If the *exclusive* bit is set in the login ORB and there are any active *login_descriptors* for the logical unit (other than one whose *login_owner_EUI_64* matches the EUI-64 of the initiator requesting the login), the target shall reject the login request with an *sbp_status* of access denied;

g) If an active *login_descriptor* with the *exclusive* attribute exists for the lun specified in the login ORB (other than one whose *login_owner_EUI_64* matches the EUI-64 of the initiator requesting the login), the target shall reject the login request with an *sbp_status* of access denied; else

h) If the *update* bit in the login ORB is zero, the target shall determine if a free *login_descriptor* is available and, if none are available, reject the login request with an *sbp_status* of resources unavailable.

If the *update* bit is zero, once the above conditions have been met and a *login_descriptor* allocated, the initiator's *source_ID* is stored in *login_owner_ID*, the initiator's EUI-64 is stored in *login_owner_EUI_64*, the *lun* and *status_FIFO* fields from the login ORB are stored in the *login_descriptor*, the *bridge_aware* and *exclusive* variables in the *login_descriptor* are set to the values of the *aware* and *exclusive* bits, respectively, from the login ORB and the address of the fetch agent and the *reconnect_hold* value chosen by the target are stored in the *login_descriptor*. If the *bridge_aware* variable is true, the target allocates a node handle to the initiator (the process is essentially the same as described by 8.4.2) and stores it in *initiator_node_handle*. Lastly the target assigns a unique *login_ID* to this login and stores it in the *login_descriptor*.

When the *update* bit is one, the login request permits the initiator to change parameters associated with the login. If the login request meets all of the validation requirements described above, the target shall logout the initiator (see 8.7) without returning completion status and then shall process the login in accordance with the requirements of this clause, 8.3.1.

NOTE – The requirement in 6.5 that the target process one management ORB at a time insures that the target performs these two steps such that no other initiator is afforded an opportunity to login between the time that target resources have been released and the time the update login request completes.

If the target is able to satisfy the login request, it shall return a login response as specified in 5.2.4.2. When the *update* bit in the login ORB is one, the information returned in the login response may differ from that previously associated with the login.

8.3.2 Create task set

A secondary task set may be created for an initiator only after completion of the login process just described. The initiator shall supply a *login_ID* previously obtained as the result of a successful login.

The target shall perform the following to validate a create task set request:

– The target shall validate the *login_ID* supplied in the create task set ORB by comparing the *destination_ID* in the read requests used to fetch the ORB with the *source_ID* retained when *login_ID* was assigned to the initiator. If the node IDs do not match, the *login_ID* is invalid.

  If the *login_ID* is valid, the target shall determine if a free *task_set_descriptor* is available and, if none are available, reject the create task set request with an *sbp_status* of resources unavailable.

Once the above conditions have been met and a *task_set_descriptor* allocated, the *task_set_descriptor* is associated with the appropriate *login_descriptor* and the address of the new fetch agent is stored in the *task_set_descriptor*. Lastly the target assigns a unique *task_set_ID* to this task set ~~and~~, stores it in the *task_set_descriptor* and returns a create task set response as specified in 5.2.4.4.

8.4 Node handles

8.4.1 Node handles (general)

When a bridge-aware login is established, the target returns a node handle in the response data; the initiator uses the node handle in buffer descriptors in ORBs or page tables that refer to initiator system memory. If necessary to address a node other than the initiator itself in buffer descriptors, the initiator shall first obtain a node handle from the target by means of a node handle request. When an initiator no longer requires one or more node handles, it should release them. All node handles for a login are automatically released upon logout.

Bridge-aware targets shall support, at a minimum, the allocation of a node handle that occurs upon successful completion of a login whose *aware* bit is one. Support for the node handle request is optional; without it, bridge-aware target operations are possible only if all data buffers are located in the initiator.

The function of an individual node handle ORB is encoded by its *allocate* bit. When *allocate* is one, the target is requested to allocate a node handle; otherwise it is requested to release a particular node handle or all node handles (except the initiator's own node handle).

Before processing any node handle request, the target shall verify that the *login_descriptor* identified by *login_ID* in the node handle request is active; if not, the target shall reject the node handle request with an *sbp_status* of login ID invalid.

Node handles shall be unique within the context of a login. An initiator shall not use a node handle obtained for one login in the context of another login, but if necessary shall obtain an additional node handle for the other login even though the node handle references the same node.

8.4.2 Node handle allocation

The target shall perform the following steps to process a node handle allocation request:

- If the least significant six bits of *global_node_ID* are all ones, the target shall reject the node handle request with an *sbp_status* of unknown EUI-64;

- In cases where a node handle exists within the login for the node identified by *eui_64*, the target shall update the *global_node_ID* in the *node_handle_descriptor* with the new value provided in the node handle request, mark the node handle descriptor to indicate that the correlation between *eui_64* and *global_node_ID* is unverified and then skip the remaining steps described below;

- Otherwise, the target shall determine if the resources to create a node handle are available and, if not, reject the node handle request with an *sbp_status* of resources unavailable; else
- Once these verification steps have succeeded, the target shall store the *eui_64* and *global_node_ID* from the node handle request in the *node_handle_descriptor* for the allocated node handle. The node handle descriptor shall also indicate that the correlation between *eui_64* and *global_node_ID* is unverified.

After the target has associated the global or local node ID and the EUI-64 in the node handle request with a node handle, the target shall store the assigned node handle in the *node_handle_response* buffer provided by the initiator.

Until the node handle is released, the target shall maintain information in a *node_handle_descriptor* that includes, at a minimum, the *node_handle*, the *login_ID* with which the node handle is associated, the *eui_64* of the node, the current global or local node ID for the node and, for global node IDs, the remote time-out for

the node. The remote time-out value need not be available when the node handle request completes; the target may defer determination of remote time-out (see 8.4.5).

8.4.3 Node handle release

When a target fetches a node handle request whose *allocate* bit is zero, it shall release one or more node handles associated with the login identified by *login_ID*. If *node_handle* equals $FFFF_{16}$, all node handles associated with the login (except the initiator's own node handle) shall be released. If *node_handle* identifies the node handle returned by a login response, the target shall reject the node handle request with an *sbp_status* of invalid node handle. Otherwise, the node handle identified by *node_handle* shall be released.

If *node_handle* does not match any node handle associated with the login identified by *login_ID*, the target shall reject the node handle request with an *sbp_status* of invalid node handle.

8.4.4 Node handle update after bus reset

Upon a Serial Bus reset, all of a target's active *node_handle_descriptors* whose *node_ID* field contains a local node ID shall be updated with the local node ID currently valid for the associated EUI-64. The target shall obtain this information from its own analysis of self-ID packets observed subsequent to bus reset and the previous bus topology map. The bus topology map correlates EUI-64 with physical ID for nodes on the local bus. Once a correlation between EUI-64 and physical ID is provisionally established by self-ID packet analysis, the target shall confirm the node's EUI-64 by means of two quadlet read transactions addressed to the bus information block. If the expected EUI-64 is not confirmed, the target shall read EUI-64 information from other local nodes until the desired EUI-64 is discovered or all local nodes have been examined.

If a local node for which a node handle is allocated is disconnected, the *node_ID* field in its node handle descriptor shall be set to $FFFF_{16}$; this causes an error if the node handle is present in a command block ORB subsequently signaled to the target.

NOTE – The initiator, which is also aware of the disconnection of the local node, should update the target's node handle information with a node handle request if the node is subsequently reconnected to the local bus.

8.4.5 Node handle validation after net update

Net update begins when a target's QUARANTINE.*orphan* bit changes from zero to one, at which time all of its active *node_handle_descriptors* whose *node_ID* field contains a global node ID shall be marked to indicate that the correlation between *eui_64* and *global_node_ID* is unverified ~~invalidated~~. So long as this correlation remains unverified, the node handle is invalid. Before the target executes an ORB that contains or references (*via* a page table) an invalid node handle, it shall ~~reestablish~~ verify the correlation between the EUI-64 and global node ID associated with the node handle and obtain the remote time-out value for the bus to which the node is connected.

Node handles assigned to initiators as a consequence of a bridge-aware login are revalidated as part of the reconnection process (see 8.6).

Node handles allocated by node handle requests shall be revalidated as follows:

- The target may use one of two methods to ~~revalidate~~ verify the correlation of an EUI-64 with a global node ID. If the target has already determined remote time-out for the bus ID specified by the global node ID, it may use either a TIMEOUT request addressed to the global node ID or it may use two quadlet read requests to obtain the node's EUI-64 from configuration ROM. If the remote time-out is not known for the bus ID, the target shall use a TIMEOUT request to determine both the EUI-64 and remote time-out;

- If there is no successful response to either the TIMEOUT request or attempted read of configuration ROM or if the EUI-64 obtained does not match the EUI-64 expected, the target shall set the *node_ID* field in the node handle descriptor to $FFFF_{16}$; this causes an error if the node handle is present in a command block ORB subsequently signaled to the logical unit fetch agent; else

- Otherwise, if the EUI-64 obtained matches that associated with the node handle ~~prior to net update~~, the target shall store the remote time-out value in the *node_handle_descriptor* and mark the node handle valid.

Although a target may revalidate all its node handles when its QUARANTINE.*orphan* bit changes from one to zero, it should elect a "lazy" scheme and defer revalidation until an invalid node handle is encountered in a command block ORB signaled to a target logical unit. If an ORB signaled to a logical unit fetch agent contains a node handle whose global node ID in the *node_handle_descriptor* is equal to $FFFF_{16}$, the task set shall be aborted and an *sbp_status* of unknown EUI-64 shall be reported for the task that caused the abort.

8.5 Heartbeat

A "heartbeat" is a periodic signal from an initiator to a target logical unit; its purpose is to maintain an active, bridge-aware login for the initiator. Absent such a signal, the target, after a time-out period, commences actions that eventually logout the initiator and free target resources.

Whenever a task set owned by an initiator identified by a global node ID becomes empty—which is the case immediately after a successful login or reconnect as well as when a task set is aborted or its work completes—the target logical unit starts a timer, *heartbeat_timeout*. The timer is initially set to the *reconnect_hold* time reported to the initiator for the login; it counts down to zero. While the task set remains empty, the timer requires periodic refresh by a signal from the initiator. If the task set is empty, a request subaction whose *source_ID* matches the node ID of the initiator associated with the login and which is addressed to the HEARTBEAT_MONITOR register shall cause the target logical unit to reinitialize the value of *heartbeat_timeout* to *reconnect_hold* seconds.

When an ORB is signaled to a logical unit fetch agent whose associated task set is empty, the *heartbeat_timeout* timer shall be stopped. Once the task set again becomes empty, the timer shall be initialized to *reconnect_hold* seconds and the heartbeat time-out process shall resume.

If the *heartbeat_timeout* timer decrements to zero while the logical unit task set is empty, the target shall commence a reconnect hold period for the login as described in 8.6. Once a target commences a reconnect hold period, it shall reject, with a response of type error, Serial Bus request subactions addressed to any of the fetch agent CSRs associated with the login.

8.6 Reconnection

Upon a Serial Bus reset, the target shall abort all task sets for all command block agents created by or associated with login requests whose *aware* bit was zero. The target shall also determine whether or not net update is active; all net update conditions are signaled by bus reset but not all bus resets signal net update. If net update is not active, task sets created as the result of login requests whose *aware* bit was one shall not be aborted. Otherwise, when net update is active, the target shall abort all task sets for all command block agents.

There is a special case that applies to logins that were created as the result of login requests whose *aware* bit was one and for which the initiator is connected to the target's local bus. This condition may exist whether or not bridge portals are connected to the local bus. If a local initiator is the owner of a bridge-aware login and is disconnected from the bus, all its task sets shall be aborted.

For each login whose task set was empty at the expiration of a heartbeat timer or whose task set was aborted by bus reset, disconnection of a local initiator or net update, the target shall retain, for at least *reconnect_hold* + 1 seconds subsequent to the trigger event—either a bus reset, disconnection of a local initiator, net update or expiration of a heartbeat timer—sufficient information to permit an initiator to reconnect its login (and, implicitly, any associated secondary task sets). After this time, but within *reconnect_hold* + 2 seconds, the target shall perform an implicit logout for each expired login ID or task set ID that has not been successfully reconnected to its original initiator. The *reconnect_hold* parameter is communicated from the target to the initiator as part of the login response data. If the trigger event is a bus reset or disconnection of a local initiator, the time-out commences when the target observes the first subaction gap subsequent to a bus reset. If a bus reset occurs before the time-out expires, the timer is zeroed then restarted upon detection of a subaction gap. If the trigger event is net update, the time-out commences when the target's QUARANTINE.*orphan* bit changes from one to zero. If QUARANTINE.*orphan* changes from zero to one before the time-out expires, the timer is zeroed then restarted upon the next transition of QUARANTINE.*orphan* from one to zero. Otherwise, if the trigger event is the lack of a heartbeat, the time-out commences upon the heartbeat timer's expiration. In the last two cases, if a net update commences before the *reconnect_hold* period elapses, the timer is zeroed, restarted upon the next transition of QUARANTINE.*orphan* from one to zero and thereafter the reconnect hold time-out is managed as described for a net update trigger event.

After a task set is aborted by bus reset or net update, an initiator shall not signal requests for an otherwise valid login until it first performs a reconnect. The reconnect request, whose format is specified in 5.2.4.5, shall be signaled to the target's MANAGEMENT_AGENT register by means of an 8-byte block write transaction that specifies the Serial Bus address of the reconnect ORB. The address of the management agent is that previously obtained by the initiator from the target's configuration ROM.

The speed at which the block write request to the MANAGEMENT_AGENT register is received shall determine the speed used by the target for subsequent requests to read the initiator's configuration ROM, fetch ORBs from initiator memory or store status at the initiator's *status_FIFO*. This replaces the speed most recently obtained from the prior login or reconnect request.

The target shall perform the following to validate a reconnect request:

- If the *source_ID* from the write transaction used to signal the login ORB to the target's MANAGEMENT_AGENT register contains a global node ID, the target shall use a TIMEOUT request, as defined by draft standard IEEE P1394.1, to verify the EUI-64 of the reconnecting initiator and to obtain its remote timeout information;

- Otherwise, the target shall read the initiator's unique ID, EUI-64, from the bus information block by means of two quadlet read transactions. The *source_ID* from the write transaction used to signal the reconnect ORB to the target's MANAGEMENT_AGENT register shall be used as the *destination_ID* in the quadlet read transactions;

  The target shall determine whether or not the *login_ID* is valid by comparing the just obtained EUI-64 against the *login_owner_EUI_64* for the *login_descriptor* identified by *login_ID*;

If the *login_ID* is valid, the target shall update *login_owner_ID* in the referenced *login_descriptor* (and in all task set descriptors associated with the same initiator) with the initiator's *source_ID*.

Upon successful completion of a reconnect request, all fetch agents associated with *login_ID* shall be in the reset state. No login response data is stored for a reconnect request; the completion status is indicated by the status block stored at the *status_FIFO* address.

8.7 Logout

When an initiator no longer requires access to a target's resources, it shall signal a logout request to the management agent. The login or task set resources to be released shall be identified by *login_ID* in the logout ORB. A target shall reject a logout request if *login_ID* does not match that of any active *login_descriptor* or if the *source_ID* of the write request used to signal the logout ORB to the MANAGEMENT_AGENT register is not equal to the *source_ID* of the matching *login_descriptor*. A logout whose *login_ID* was obtained as the result of a login request implicitly causes the release of all node handles associated with *login_ID* and the logout of all secondary task sets associated with the *login_ID*. Any tasks active at the time of the logout request shall be aborted in the same fashion as if the task set had been aborted. Upon successful completion of a logout request, all resources allocated to the initiator are free once again and may be used by the target to satisfy subsequent login or create task set requests.

9 Command execution

9.1 Command execution overview

This section describes the procedures used by an initiator to request command execution by a target logical unit. As described in the model, requests are specified by data structures in system memory that are subsequently fetched by the logical unit. While a logical unit executes a request, it is responsible for any data transfer associated with the request. Once a request completes, successfully or in error, a status block may be stored in system memory by the logical unit. The data structures are defined in section 5; the initiator procedures for the use of these request and status blocks are described in the clauses that follow

9.2 Requests and request lists

9.2.1 Requests and request lists (general)

Management requests (which include login and logout requests) are signaled to the target agent by means of a Serial Bus block write request that specifies the address of the management ORB. The management agent becomes busy while executing a request and refuses subsequent Serial Bus requests with *ack_conflict_error* or *resp_conflict_error* until the current transaction is completed. The management agent does not require any initialization procedures.

The target's logical unit command block agents are called fetch agents since they manage linked lists of requests in system memory and are responsible to fetch the ORBs. For command block ORBs, the initiator produces requests and the logical unit consumes them. These processes are asynchronous and independent of each other. Logical unit efficiency is improved if the logical unit can be kept occupied with an ample working set of requests. To this end, the initiator is permitted to arrange ORBs in linked lists and to dynamically append new requests to the lists while the logical unit remains active.

Each command block ORB contains an address pointer, *next_ORB*, which shall either be a null pointer or point to another ORB. A linked list of ORBs, previously illustrated by Figure 6, implicitly orders the ORBs—the fact that the ORBs are in order permits the logical unit to execute them in order (or not) according to its device-dependent characteristics.

The logical unit is responsible to fetch ORBs from system memory, as described in detail in 9.3. The remainder of this clause describes what the initiator does to:

- initialize a logical unit fetch agent;
- dynamically append new requests to an active list and notify a fetch agent of the new requests;
- notify a fetch agent of a single new request; and
- use the FAST_START register to notify an idle fetch agent of a single new requests.

Within this clause there is also a description of how the logical unit parses an ORB and its associated page tables.

9.2.2 Fetch agent initialization (informative)

After successful completion of a login procedure and the return of the base address of the fetch agent CSRs, the initiator may initialize the fetch agent as follows:

a) The initiator allocates space for a dummy ORB and initializes it *per* the format described in 5.2.2. Although only the *next_ORB* field, *notify* bit and the *rq_fmt* field are significant within a dummy ORB,

the initiator allocates at least the minimum ORB size specified by the target's configuration ROM. The initiator sets the *next_ORB* field to the null pointer value;

b) The initiator resets the fetch agent by a quadlet write to the fetch agent's AGENT_RESET register;

c) The initiator writes the address of the dummy ORB to the fetch agent's ORB_POINTER register by means of an 8-byte block write request. In the example in Figure 68, this is the value 0000 0000 8004 $00C0_{16}$. This causes the fetch agent to ~~transition to~~ enter the ACTIVE state.

The figure below illustrates the result of these actions:

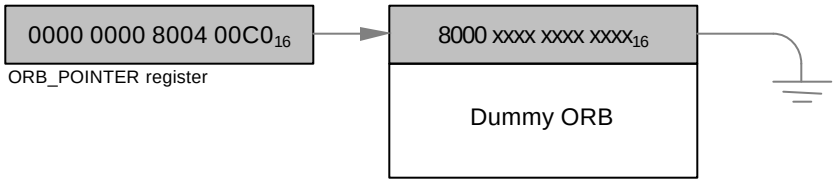

Figure 68 – Fetch agent initialization with a dummy ORB

When the fetch agent ~~transitions to~~ enters the active state as a result of the write to the ORB_POINTER register, it uses the value to fetch the dummy ORB (as target resources permit). The dummy ORB, by definition, completes immediately and the fetch agent stores a status block for the request. However, the null pointer in the *next_ORB* field of the dummy ORB causes the fetch agent to ~~transition to~~ enter the suspended state. The ORB_POINTER register still points to the dummy ORB and the initiator may subsequently append additional requests, as described in 9.2.3.

NOTE – Initialization does not require that the first command block signaled to a fetch agent be a dummy ORB nor that the list contain only one ORB. A linked list with more than one command may be used both to initialize the fetch engine and to execute the commands.

9.2.3 Dynamic appends to request lists (informative)

Once a logical unit fetch agent has been initialized and made active as described above, it is possible for the initiator to append new requests to the linked list while the fetch agent remains active. Assume that the initiator intends to add three new requests previously illustrated by Figure 6.

An initiator may append new requests to an active request list as follows:

a) The initiator constructs a linked list of ORBs in system memory, as illustrated in the example. The *next_ORB* field of the last ORB contains a null pointer. The *next_ORB* fields of all other ORBs contain a valid pointer to a subsequent ORB;

b) The initiator updates the *next_ORB* field of what had been the last ORB, in this example the dummy ORB in Figure 68, with the address of the first request in the new request list, in this example 0000 0000 8000 0000_{16}; and

c) Lastly, the initiator transmits a quadlet write request, with any data value, to the fetch agent's DOORBELL register.

The final step informs the fetch agent that address pointers in the request list have been updated by the initiator. If the fetch agent had not encountered a null pointer, the activation of the doorbell is redundant. However, if the fetch agent is already suspended at the time *next_ORB* is updated, the activation of the doorbell is essential to reactivate the fetch agent. In this latter case, it is necessary for the fetch agent to

refetch all or part of the ORB referenced by the ORB_POINTER register from system memory in order to ascertain if a previously null pointer contains a valid address of an ORB.

9.2.4 Fetch agent use by the BIOS (informative)

The BIOS, or any similar initiator application that executes in a single-threaded environment, has little need of the logical unit fetch agent's capabilities to manage multiple outstanding requests. The BIOS may use a simpler procedure than that described in 9.2.3 to signal requests to the logical unit. Subsequent to initialization of the fetch agent by means of a write to the AGENT_RESET register, the BIOS may signal one request at a time to the logical unit as follows:

a) The BIOS allocates space for the request in an ORB and initializes it according to the ORB format. The *next_ORB* field contains a null pointer;

b) The BIOS signals the request to the fetch agent by writing the address of the ORB to the ORB_POINTER register in an 8-byte block write transaction. This causes the fetch agent to ~~transition to~~ enter the ACTIVE state and to execute the request; and

c) Subsequent to the return of a status block to the *status_FIFO* address specified when the login was performed, the BIOS may signal additional requests by repeating this procedure.

The performance improvements yielded by the above procedure (which are accomplished by the elimination of a read transaction to fetch an ORB pointer) are minor; the principal benefit to the BIOS is code simplification.

9.2.5 Use of the FAST_START register (informative)

An initiator may signal new tasks to the fetch agent by a block write request addressed to the fetch agent's FAST_START register (see 6.6.8). The block write request contains pointers both to the ORB to be commenced and to the previous ORB (*i.e.*, the ORB whose *next_ORB* field references the ORB to be commenced), a copy of the ORB itself and, optionally, page table data associated with the ORB. Significant overhead reductions may result from the use of the FAST_START register, since the logical unit need not fetch ~~the~~ either the address of the ORB or the ORB itself. In cases where the block write request contains the entire page table, the logical unit need not fetch the page table; even if the entire page table is not written to the FAST_START register (it may be too large), the logical unit may significantly reduce startup latency by fetching the remaining page table entries concurrently with task execution.

Although an initiator may achieve optimal performance improvement by writing to the FAST_START register when the fetch agent is in either the RESET or SUSPENDED state, the register may also be used when the fetch agent is active. In this case, the fetch agent ignores the data payload of the block write request and behaves as if a quadlet write had been addressed to the fetch agent's DOORBELL register. There are several ways by which an initiator may securely know that a fetch agent is in the RESET or SUSPENDED state. If the initiator has not written to either of the fetch agent's ORB_POINTER or FAST_START registers since the most recently completed login or reconnect operation or the most recent write to the fetch agent's AGENT_RESET register, the fetch agent is in the RESET state. Similarly, if the initiator has not written to either of the fetch agent's ORB_POINTER or FAST_START registers since the logical unit last stored a status block with a *src* field equal to one (see 5.4.2), the fetch agent is in the SUSPENDED state.

NOTE – An initiator may use the above methods to deduce fetch agent state whether or not the logical unit implements the FAST_START register. If the register is not supported, startup latency for an idle fetch agent may be reduced by writing the address of an ORB directly to the ORB_POINTER register instead of a write to the DOORBELL register.

There are two variants to the use of the FAST_START register, one suitable for single-threaded initiators and the other suitable for multi-threaded (possibly multiprocessor) initiators. If the initiator implementation

guarantees that no more than one block write request to the FAST_START register is attempted while the fetch agent is idle, it may set the *previous_ORB* field to a null pointer; this causes the idle fetch agent to unconditionally update the ORB_POINTER register with the value of *this_ORB*. Multi-threaded initiators ~~may~~ might not be able to satisfy this constraint for ordered writes to the FAST_START register, in which case the method outlined below may be used:

a) Construct the ORB (with a null *next_ORB* field) and associated data structures in system memory. The address of the ORB is designated *this_ORB*;

b) In an effectively atomic operation (*i.e.*, one protected within a critical section), obtain the current tail pointer to the linked list of active ORBs, save it as *previous_ORB* and replace the tail pointer with *this_ORB*;

c) Store *this_ORB* in the *next_ORB* field of the ORB referenced by *previous_ORB*;

d) Initiate a block write request to the fetch agent's FAST_START register; its data payload should include *previous_ORB*, *this_ORB*, a copy of the ORB and, optionally, page table information.

The presence of a non-null *previous_ORB* field permits the fetch agent to ignore FAST_START write requests that arrive out of order.

Either a single ORB or a linked list of ORBs may be signaled in a single block write request to the FAST_START register, dependent upon the value of the *next_ORB* field in the ORB contained within the block write.[6] Once a successful completion response is received for the block write request, the initiator may append to the linked list of ORBs by the methods described in 9.2.3.

9.2.6 Fetch agent parse of ORB and page tables (informative)

Once a fetch agent has obtained an ORB from an initiator, whether read from the initiator's system memory or written to the logical unit's FAST_START register, it is necessary to parse the ORB into its header quadlets (specified by this standard) and its *command_block* (specified by the logical unit's command set) and also to decode whatever page tables (none, one or two) are associated with the ORB.

The number of header quadlets are determined by the *rq_fmt* field, as specified by the table below:

Value	ORB format	Header quadlets
0	Command block (single buffer descriptor)	5
1	Command block (dual buffer descriptor)	8
2	Vendor-dependent	
3	Dummy (NOP)	5

The size of the *command_block* field is determined by the Command_Set_Spec_ID, Command_Set and Command_Set_Revision entries for the logical unit served by the fetch agent.

If the ORB was read from initiator system memory, the buffer descriptor fields (specified in 5.2.3) determine the size and location of any page tables associated with the ORB. Otherwise the ORB was written to the FAST_START register and page table data, if any, immediately follows the ORB in the data payload written to the register. The buffer descriptor fields in the ORB determine whether a page table describes *buffer*[0],

[6] When more than one ORB is signaled by a write to the FAST_START register, the algorithm described for multi-threaded operations is modified to update the linked list tail pointer with the address of the final ORB in the list to be appended rather than with the value of *this_ORB*.

buffer[1] or both, but the rules below determine how to parse page table data included in the data written to the FAST_START register:

- If *buffer*[0] is described by a page table, the *buffer*[0] page table immediately follows the ORB, up to a maximum of *data_size*[0] bytes; the page table may be omitted or truncated.
- If *buffer*[0] is not described by a page table but *buffer*[1] is described by a page table, the *buffer*[1] page table immediately follows the ORB, up to a maximum of *data_size*[1] bytes; the page table may be omitted or truncated.
- If both *buffer*[0] and *buffer*[1] are described by page tables, page table data for *buffer*[1] may be in the FAST_START data only if the entire page table for *buffer*[0] immediately follows the ORB. In this case, the *buffer*[1] page table immediately follows the *buffer*[0] page table, up to a maximum of *data_size*[1] bytes; the page table for *buffer*[1] may be omitted or truncated.

9.3 Fetch agent state machine

The operations of a logical unit fetch agent are specified by the figure below. The state of a fetch agent is visible in the context displayed by the AGENT_STATE and ORB_POINTER registers described in 6.6. The state machine diagram and accompanying text explicitly specify the conditions for transition from one state to another and the actions taken within states.

The target shall qualify all writes to fetch agent CSRs by the *source_ID* of their currently logged-in initiator. A write to a fetch agent CSR by any other Serial Bus node shall be rejected by the target by one of the following methods:

- an acknowledgment of *ack_type_error*;
- an acknowledgment of *ack_complete* (although the write is ignored); or
- an acknowledgment of *ack_pending*. When the target subsequently responds, the response code shall be *resp_type_error*.

The recommended logical unit action is to indicate a type error, either by an acknowledgment of *ack_type_error* or an acknowledgment of *ack_pending* followed by *resp_type_error*. A ~~logical unit~~ target that reports a nonzero value in its configuration ROM Revision entry shall indicate a type error.

During a reconnect hold period for a particular set of fetch agent CSRs, the *source_ID* of the previously logged-in initiator is unknown; a write to those fetch agent CSRs by any node shall be rejected as specified above.

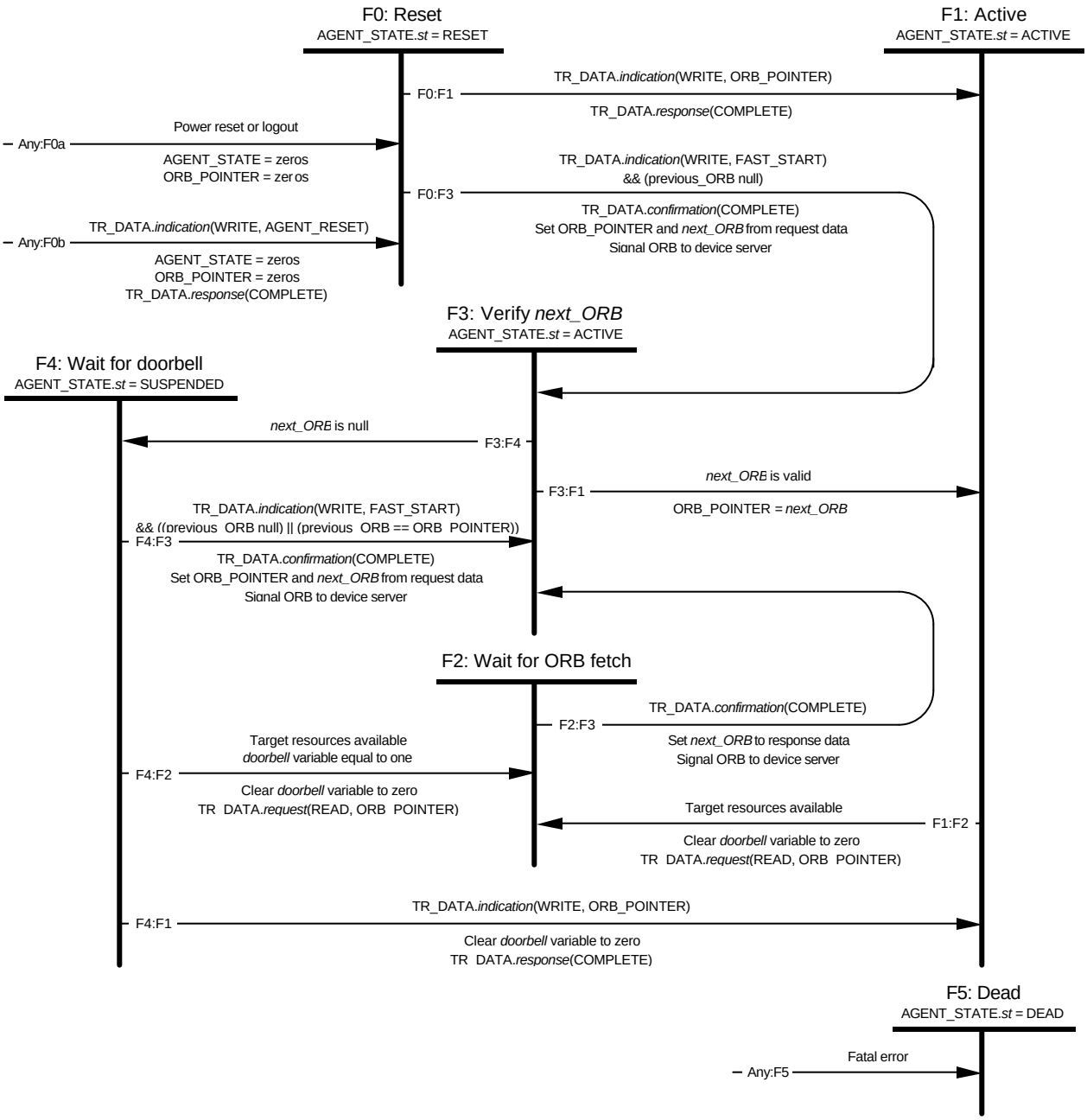

Figure 69 – Fetch agent state machine

Transition Any:F0a. A power reset shall cause the fetch agent to ~~transition to~~ enter the RESET state from any other state. The AGENT_STATE and ORB_POINTER registers (that control and make visible the operations of the fetch agent) shall be reset to zeros.

Transition Any:F0b. A quadlet write request by the initiator to the AGENT_RESET register shall cause the fetch agent to ~~transition to~~ enter state F0 from any other state. The fetch agent shall zero the AGENT_STATE and ORB_POINTER registers before the transition to state F0. Transaction labels for outstanding request subactions shall not be reused until either the corresponding response subaction completes or a split time-out (local or remote, as appropriate), expires; in the first case, the response data shall be discarded.

State F0: Reset. Upon entry to this state, the *st* field in the AGENT_STATE register shall be set to RESET. The fetch agent is inactive and available to be initialized by an initiator.

Transition F0:F1. An 8-byte block write of a valid *ORB_offset* to the ORB_POINTER register shall update the register and cause the fetch agent to ~~transition to~~ enter state F1. The target shall confirm the block write request with a response subaction of COMPLETE.

Transition F0:F3. A block write to the FAST_START register may affect the state of the fetch agent. If the *previous_ORB* field does not contain a null pointer, the fetch agent state shall not change and the target shall confirm the block write request with a response subaction of COMPLETE. Otherwise the fetch agent shall update the ORB_POINTER register with the value of *this_ORB*, shall update the *next_ORB* variable with the contents of the *next_ORB* field from the ORB contained within the block write request and shall cause the fetch agent to ~~transition to~~ enter state F3. The fetch agent shall also make the ORB and any page table data available to the device server for execution. Once these actions are complete, the target shall confirm the block write request with a response subaction of COMPLETE.

NOTE – If the *previous_ORB* field contains a null pointer, a target may interpret a block write addressed to its FAST_START register as if the *this_ORB* field had been written to its ORB_POINTER register. Although it is less efficient to elect transition F0:F1, it is functionally equivalent to transition F0:F3.

NOTE – When the fetch agent is in the RESET state, it is not necessary to write to the DOORBELL register upon either transition F0:F1 or F0:F3.

State F1: Active. Upon entry to state F1, the *st* field in the AGENT_STATE register shall be set to ACTIVE. In this state, the fetch agent may use the address information in the ORB_POINTER register to fetch ORBs from the initiator as resources permit.

Transition F1:F2. The availability of target resources is an implementation-dependent decision. Typically, the resources might be space in device memory to hold an image of the ORB while the command is scheduled for execution and subsequently completed. In any case, the fetch agent clears the *doorbell* variable to zero and then issues a block read request to obtain the ORB from system memory.

State F2: Wait for ORB fetch. The fetch agent is suspended and awaiting a read response for a block read directed to the address contained in the ORB_POINTER register.

Transition F2:F3. Subsequent to a block read request, issued as described above, the fetch agent may accept a block read response that contains either the *next_ORB* data or an entire ORB intended for execution by the device server. If a read response is received whose *source_ID*, *destination_ID* and *tl* fields match the *destination_ID*, *source_ID* and *tl* fields, respectively, of the read request, the fetch agent shall copy the *next_ORB* field from the response data to the *next_ORB* variable before making the transition to state F3. When the response data contains an entire ORB not yet in the device server's working set, the fetch agent shall make the ORB available to the device server for execution.

State F3: Verify *next_ORB*. Upon entry to state F3, the *st* field in the AGENT_STATE register shall be set to ACTIVE.[7] The *next_ORB* variable contains information about a subsequent ORB that may be linked in order after the one just fetched. As described in 5.2, the *next_ORB* pointer encodes the address of the next ORB. The actions of this state determine whether or not the *next_ORB* pointer is null.

Transition F3:F1. If the *next_ORB* variable does not indicate a null pointer the fetch agent shall update the ORB_POINTER register with the value of *next_ORB*.

[7] Although this action is redundant in the case of transition F2:F3, it is necessary for transitions F0:F3 and F4:F3.

Transition F3:F4. The fetch agent shall ~~transition to~~ enter a suspended state, F4, if *next_ORB* contains a null pointer. A null pointer is defined in 5.2 and exists if the most significant bit of the variable is one.

State F4: Wait for doorbell. Upon entry to state F4, the *st* field in the AGENT_STATE register shall be set to SUSPENDED. The fetch agent is suspended; the ORB_POINTER register contains the address of the ORB whose *next_ORB* field was null at the time state F4 was entered.

Transition F4:F1. If an indication of a write to the ORB_POINTER register is received, the fetch agent shall clear the *doorbell* variable to zero and confirm the write transaction with a response subaction of COMPLETE. After the confirmation, the fetch agent shall ~~transition to~~ enter state F1.

Transition F4:F2. Whenever the *doorbell* variable is equal to one, the fetch agent shall clear the *doorbell* variable to zero, issue a read request to obtain a fresh copy of the *next_ORB* field from the ORB whose address is contained in the ORB_POINTER register and then ~~transition to~~ enter state F2. The *doorbell* variable is set to one as the result of a quadlet write request of any value to the DOORBELL register, whether the write request is received in this or any other state.

The fetch agent may issue either an 8-byte block read request (to fetch just the *next_ORB* field) or it may reread the entire ORB. The initiator shall insure that system memory occupied by the ORB remains accessible, as described in 9.5.

Transition F4:F3. A block write to the FAST_START register may affect the state of the fetch agent. If the *previous_ORB* field does not contain a null pointer and is not equal to the ORB_POINTER register, the fetch agent state shall not change and the target shall confirm the block write request with a response subaction of COMPLETE. Otherwise the fetch agent shall update the ORB_POINTER register with the value of *this_ORB*, shall update the *next_ORB* variable with the contents of the *next_ORB* field from the ORB contained within the block write request and shall cause the fetch agent to ~~transition to~~ enter state F3. The fetch agent shall also make the ORB and any page table data available to the device server for execution. Once these actions are complete, the target shall confirm the block write request with a response subaction of COMPLETE.

NOTE – If the *previous_ORB* field either contains a null pointer or is equal to the ORB_POINTER register, a target may interpret a block write addressed to its FAST_START register as if the *this_ORB* field had been written to its ORB_POINTER register. Although it is less efficient to elect transition F4:F1, it is functionally equivalent to transition F4:F3.

Transition Any:F5. Upon the detection of any fatal error, the fetch agent shall ~~transition to~~ enter state F5. Examples of fatal errors include, but are not limited to:

- the failure of the addressed node to acknowledge a read request;
- the failure of the addressed node to respond to a read request (local or remote split time-out);
- a busy condition at the addressed node that ~~exceeds~~ persists beyond the target's busy retry limit;
- a data CRC error in a response subaction.

Some of these errors may be recoverable if retried by the target.

The fetch agent may also be instructed to ~~transition to~~ enter the dead state as a result of an error in command execution detected by the device server.

State F5: Dead. The dead state ~~is a unique state that~~ preserves fetch agent information in the AGENT_STATE and ORB_POINTER registers. Writes to any fetch agent register except AGENT_RESET shall have no effect while in state F5.

9.4 Asynchronous data transfer

The asynchronous transfer of data associated with a command is the responsibility of the target. When an ORB specifies one or more data buffers (*i.e.*, *data_size* is nonzero) and the *isochronous* bit is zero, the target shall use Serial Bus read transactions to fetch data from system memory and Serial Bus write transactions to store data in system memory.

The total transfer length may be larger than the maximum data payload that can be accommodated in a single transaction. The target is responsible to manage the size and number of read or write transactions to transfer all the requested data. The target may choose any appropriate size for these data transfer transactions, subject to constraints specified by the ORB.

The target shall observe alignment requirements specified by the *page_size* field. A *page_size* value of zero indicates that there are no alignment requirements. Nonzero *page_size* values specify that the target shall observe alignment boundaries that occur every $2^{page\_size + 8}$ bytes in the data buffer or optional page table; no single Serial Bus block read or block write transaction shall cross such a boundary[8].

The target shall issue data transfer requests with a speed equal to that specified by the controlling *spd* field, whether in the ORB or in a node selector in an associated page table. The target shall not issue block read or write requests with a data payload length greater than that specified by the controlling *max_payload* field, whether in the ORB or in a node selector in an associated page table.

Within the above speed, size and alignment constraints, the target is free to issue the data transfer requests in any order and to retry failed data transfer requests according to vendor-dependent algorithms.

9.5 Isochronous data transfer

The isochronous transfer of data associated with a command block ORB is only partly the responsibility of the logical unit. Depending upon the logical unit's role, either talker or listener, successful data transfer also depends upon the other endpoint; either the listeners' correct receipt or the talker's correct transmission of isochronous subactions are essential. When an ORB specifies a data stream (*i.e.*, *data_size* is nonzero) and the *isochronous* bit is one, the target shall transmit or receive Serial Bus isochronous subactions according to whether the *direction* bit in the ORB is one or zero, respectively.

If the logical unit is a talker, it shall transmit the data specified by the command in a sequence of isochronous subactions, zero or one per isochronous period; the data shall be transmitted on Serial Bus in the identical order it was obtained from the device medium or other source. The total transfer length may be larger than the maximum data payload that can be accommodated in a single isochronous subaction. The logical unit shall limit the size of each isochronous subaction to less than or equal to the maximum permitted; this information may be obtained either from the logical unit's output plug control register (oPCR; see 11.2) or by command set-dependent means. The channel number and the speed at which the isochronous subaction is to be transmitted shall be obtained from the same source as the maximum data payload.

If the logical unit is a listener, it shall receive the data specified by the command from a sequence of isochronous subactions, zero or one per isochronous period; the data shall be stored on the device medium or other destination in the identical order it was received from Serial Bus. The channel number from which to receive the information may be identified either by the logical unit's iPCR (see 11.3) or by command set-dependent means. The logical unit shall allocate buffers, each one of which shall be sufficiently large to receive the maximum data payload that may arrive in any isochronous period. Unless the maximum data payload is determined by command set-dependent means, the logical unit shall allocate buffers whose size

[8] Page boundaries exist relative to offset zero in system memory, not relative to the starting offset of the data buffer or page table itself.

is equal to the maximum isochronous subaction size permitted by IEEE 1394 for the data rate capability specified in the target's input master plug register (iMPR).

Because isochronous subactions are unacknowledged, some kinds of data transfer errors are undetectable by the logical unit. Absent a higher-level protocol, a talker cannot determine if a transmitted isochronous subaction is correctly received by the intended listeners nor can a listener determine, in all circumstances, if it has failed to receive isochronous subactions transmitted by the talker. Certain data transfer errors can be detected; these include the following:

- Failure to observe a cycle start subaction. Either the talker or the listener may detect this condition. Recovery strategies are implementation-dependent but might include a) ignoring the missed cycle start or b) terminating the task and aborting the task set.
- Data underrun at the talker. A malformed isochronous subaction results, because the talker is unable to obtain data from the device medium or other source in time for transmission on Serial Bus. Recovery strategies are implementation-dependent but might include a) terminating the task and aborting the task set, b) padding the remainder of the isochronous subaction with constant fill data and generating a valid CRC or c) truncating the isochronous subaction or generating an invalid CRC or both. If a) is neither appropriate nor optimal, the choice between b) and c) is strongly dependent on the data format and the effect of its receipt on the listeners.
- Data overrun at the listener. A loss of data occurs, because the listener resources are busy and the listener is unable to correctly receive all of the data from the isochronous subaction. Recovery strategies are implementation-dependent but might include a) terminating the task and aborting the task set, b) if the isochronous subaction's header was correctly receive and the *data_length* of the payload is known, filling some or all of the isochronous subaction with data or c) discarding the entire isochronous subaction. If a) is neither appropriate nor optimal, the choice between b) and c) is strongly dependent on the data format.
- Data length or CRC error detected by the listener. As in the case of data overrun, a loss of data occurs because the listener is unable to correctly receive all of the data from the isochronous subaction. The same recovery strategies are applicable and the same considerations about data format apply.

Logical units that implement command sets designed with knowledge of Serial Bus isochronous behavior should mandate specific recovery strategies for each of the error cases above. Logical units that implement command sets designed without such knowledge should consider the recommendations made in 11.4.

9.6 Interim and completion status

Prior to the completion of an ORB, the target may store an interim status block, but no more than once for a particular ORB. Upon completion of the ORB, the target shall examine the *notify* bit in the ORB to determine whether or not to store a completion status block. If *notify* is zero, the target may store a status block. ~~Otherwise, if *notify* is one or~~ But if the ORB completed with an error condition or if *notify* is one, the target shall store a status block. The target shall store no more than one completion status block for a particular ORB. The address for the status block is specified by *status_FIFO*, supplied by the initiator as part of the login or create task set request. The status block, previously described in 5.4, contains sufficient information to indicate successful command completion or, in the case of a faulted command, to permit the initiator to select the appropriate error handling strategy.

In all cases, the status FIFO allocated by the initiator shall be accessible to a single Serial Bus block write transaction with any *data_length* that is a multiple of four and less than or equal to either 32 bytes (if the initiator has not enabled the logical unit to use extended status) or 512 bytes (if extended status is enabled). The target shall store the status block by means of a single block write and shall not attempt any retries if either:

a) no acknowledge packet is received immediately after the write request; or

b) subsequent to the receipt of an *ack_pending* immediately after the write request, no corresponding response packet is received within the local or remote split time-out limit, as appropriate.

Other errors, including the link layer busy conditions, *ack_data_error*, *resp_conflict_error* and *resp_data_error*, may be retried up to a vendor-dependent limit. If no retry is attempted or if the retry limit is exhausted without success, the fetch agent shall ~~transition to~~ enter the DEAD state.

The return of completion status indicates to the initiator that the task commenced by the ORB is no longer part of the task set. The status block also specifies whether or not the system memory allocated to the ORB may be released. If the *src* field has a value of zero, the initiator may reuse or deallocate the system memory occupied by an ORB. When *src* has a value of one, the system memory shall not be reused or deallocated until the target stores completion status for some subsequent ORB in the linked list.

NOTE – For targets that support the ordered model of task execution, the return of completion status for an ORB implicitly indicates that all preceding ORBs in the linked list have completed successfully, are no longer part of the task set and that the initiator may reuse or deallocate their system memory.

An initiator shall report completion status in the same order as the status blocks are written to its *status_FIFO*. This is an obvious requirement in cases where the target implements the ordered model of task execution, but it may also be necessary in the case of unordered execution if a higher-level protocol based upon SBP-3 (for example, draft standard IEEE P1394.3 [B8]) has additional ordering requirements.

9.7 Unsolicited status

In addition to status associated with a particular ORB, described in the preceding section, a logical unit may store unsolicited status at the address specified by *status_FIFO*. A status block that contains unsolicited status shall be identified by setting the *src* field to a value of two (see 5.4.3).

A logical unit may store unsolicited status at any time that its *unsolicited status enabled* variable is one. Upon successful completion of the Serial Bus block write transaction used to store the status block, the logical unit shall zero its *unsolicited status enabled* variable. The initiator may set the logical unit's *unsolicited status enabled* variable to one by writing any data value to the corresponding UNSOLICITED_STATUS_ENABLE register.

NOTE – One use for unsolicited status is to report progress of lengthy operations such as a disk format or tape rewind. Device implementers that use unsolicited status for this purpose should pick an appropriate interval for the reports. Frequent unsolicited status transfers reduce available Serial Bus bandwidth and may increase processing overhead at the initiator without any perceivable benefits.

The action taken by a logical unit when unsolicited status is generated but cannot be stored because the *unsolicited status enable* variable is zero depends upon the nature of the status. If the status is for a unit attention condition, the logical unit shall retain the information with the intent to store it as soon as the *unsolicited status enable* variable is set to one. The unit attention condition shall persist until the corresponding status block is stored at the initiator's *status_FIFO*. The definition of unit attention conditions is beyond the scope of SBP-3 and is usually the province of the command-set standard for the logical unit. Other status information, that does not constitute a unit attention, may be discarded by the logical unit, queued for future delivery or replace an existing, pending status of the same type. Which of these behaviors is appropriate is determined by the command set standard.

10 Task management

10.1 Task management overview

The preceding section describes the procedures used by the initiator to signal the target that tasks are to be executed and the procedures by which the target performs data transfer or device control for the tasks and ultimately signals their completion back to the initiator. Section 9 gives no consideration to the larger perspective of how these tasks interact with each other and how the initiator may manage the tasks.

This section defines how individual tasks are collected together as task sets and how both tasks and task sets may be managed by the initiator.

10.2 Task sets

A task set is a collection of tasks, each of which has an associated command in an ORB, that is available to the logical unit for execution. The interactions among these tasks and the ordering relationships, if any, are governed by the task management model implemented by the logical unit.

A task enters the task set when it is linked into an active request list. The extent of a task set includes all the uncompleted ORBs linked into a request list in system memory, not solely the ORBs already fetched by the logical unit (the working set).

Historically, there has been one task set associated with each logical unit of a device. Although this one-to-one association between the primary task set (the task set created by a login request) and a logical unit is retained, the concept is extended by SBP-3 to permit multiple task sets per initiator per logical unit. There may be zero or more secondary task sets associated with a logical unit. Each secondary task set is separate and distinct from the primary task set and from other secondary task sets: there are no interactions between tasks that belong to different task sets.

10.3 Basic task management model

Logical units shall support, at a minimum, a basic task management model. Logical units may implement other task management models if each model supports all of the features of the basic model. Within the basic model, the following rules apply:

- All tasks within a task set share the same execution characteristics: either they are all reorderable or else they are all ordered;
- The reorderable or ordered execution characteristics of a task set are implicit in the logical unit implementation and are not subject to control by the initiator; ~~Configuration ROM shall specify whether the logical unit may reorder task execution or not;~~
- For secondary task sets, the logical unit shall execute all tasks in order and report their completion status in the same order. For primary task sets, configuration ROM shall specify whether the logical unit may reorder task execution or not;
- All tasks within a task set are uniquely identified by the Serial Bus address of the ORB that initiated the task. This address shall be unique for the life of the task;
- The abort task, abort task set and target reset task management functions, described later in this section, shall be implemented;

An element of choice in the implementation of a task set under the basic model is whether or not the logical unit may reorder task execution. An unordered model is usually appropriate for direct-access devices for

which no positional or other context information is inherited from one command to the next. An ordered model may be more appropriate for devices, such as sequential storage, where the outcome of one command affects the next. The same ordering considerations apply to secondary task sets, within which the data is time-ordered by its very nature.

The unordered model is characterized by unrestricted reordering of the active tasks. The logical unit may reorder the actual execution sequence of any tasks in a task set in any manner, including the concurrent execution of multiple tasks.[9] Unrestricted reordering places the responsibility for the assurance of data integrity on the initiator. If the integrity of data on the device medium could be compromised by unrestricted reordering involving a set of active tasks, {T_0, T_1, T_2, … T_N} and a new task T´, the initiator shall wait until {T_0, T_1, T_2, … T_N} have completed before appending T´ to an active request list.

The ordered model requires both that tasks shall be executed in order and that completion status shall be returned in order. Because Serial Bus transactions may complete out of order, the target shall single-thread the return of completion status. Once the target has transmitted a Serial Bus block write request to store completion status in system memory, it shall not attempt to store completion status for any other task in the task set until *ack_complete* or *resp_complete* is received.

10.4 Error conditions

Upon an error condition or fault detected during the execution of any task within a task set, the entire task set shall be cleared as follows:

a) The target shall halt the operation of the fetch agent associated with the task set by making a transition to the DEAD state;

b) For all recently completed tasks, the target shall wait until the completion status of each command has been successfully stored in system memory or until the implementation-dependent retry algorithms have been exhausted in the attempt to store completion status; and

c) Finally, the target shall return error completion status for the faulted task. The *dead* bit shall be one in the status block.

The return of error status for a faulted task is an indication to the initiator that the task set has been cleared and that any remaining active tasks in the request list have been aborted.

10.5 Task management requests

~~The clauses that follow describe the use of the *rq_fmt* field in ORBs and the task management ORBs defined in 5.1.3.7.~~

10.5.1 Abort task

Abort task is a task management function that permits an initiator to abort a specified task without otherwise affecting the task set or its fetch agent. A modification to the *rq_fmt* field of the ORB to be aborted is the basic method; in addition, targets may also recognize task management ORBs to abort tasks. All targets shall support abort task.

Because the task to be aborted ~~may~~ might not have been fetched by the target when the initiator wishes to abort the task, the following procedure shall be used to abort the task:

[9] Other transport protocols based upon SBP may require that targets be capable of concurrent execution of some number of tasks. Draft standard IEEE P1394.3 is an example.

a) The *rq_fmt* field shall be set to a value of three in the ORB for the task to be aborted. This field and the *next_ORB* field are the only two portions of an ORB that may be modified by the initiator once the ORB is linked into an active request list;

b) The initiator may construct a management ORB in system memory for the abort task function. The initiator shall set the appropriate values in the *rq_fmt*, *login_ID* and *ORB_offset* fields of the ORB, as described in 5.2.4.8. The *function* field shall be set to ABORT TASK; *ORB_offset* shall contain the Serial Bus address of the ORB for the task to be aborted. The initiator then signals the abort task management ORB to the management agent.

Mandatory support for abort task requires the target to recognize an *rq_fmt* value of three in an ORB and take the actions described below.

– If the ORB to be aborted has already been fetched by the target, the task may be completed by the target without recognition of the abort task request; otherwise

– When the ORB is first fetched, the target shall recognize the *rq_fmt* field value of three and shall not execute the command. That target shall store completion status for the aborted ORB; the request status shall be REQUEST COMPLETE and the *sbp_status* field shall indicate dummy ORB completed.

A second method to abort tasks may be available by means of task management ORBs with a *function* of ABORT TASK. If the *login_ID* specified in the ORB was returned in login response data, target support for this method of abort task is optional. Otherwise the *login_ID* was returned in a *create_task_set_response* (in which case it is a *task_set_ID*) and the target shall support this method to abort tasks, as specified below. Targets that implement this method shall store a completion status of REQUEST COMPLETE for the abort task request in the status buffer specified by the ORB.

If the task to be aborted, identified by *ORB_offset*, is not recognized by the target as part of its working set, one of two conditions may exist: either the ORB has not been fetched or completion status has already been stored. In either case the target is not required to take any immediate action. In the first case, when the ORB is ultimately fetched, the *rq_fmt* field has a value of three and the target shall not execute the command. The target shall store completion status for the aborted ORB; the request status shall be REQUEST COMPLETE and the *sbp_status* field shall indicate dummy ORB completed. In the second case, no action whatsoever need be taken by the target.

If the task to be aborted is recognized by the target as part of its working set, the target should attempt to abort the task according to the steps below. Note that timing conditions may exist that prevent targets from aborting the specified task. In particular, if the target has already issued a write request to store completion status for the task to be aborted, the target shall take no other action in response to the abort task request Otherwise, if the target undertakes to abort the task it shall perform the following actions in response to a task management ORB with the ABORT TASK *function*:

a) The target should not issue additional data transfer requests for the task;

b) The target shall wait for response subactions to pending data transfer requests;

c) Only if none of the target medium, data buffer or status FIFO have been modified as the result of partial execution of the task, the target shall store completion status of REQUEST COMPLETE with an *sbp_status* field that indicates dummy ORB completed;

d) Otherwise, if task execution has commenced and any one of the target medium, data buffer or status FIFO has been modified, then the target shall store completion status of REQUEST COMPLETE with an *sbp_status* field that indicates request aborted.

Regardless of which abort task methods are supported by the target, the initiator shall not reuse the system memory occupied by the ORB, data buffer or page table of the task to be aborted until completion status is returned for that ORB.

10.5.2 Abort task set

Abort task set is a task management function that permits an initiator to abort all of its tasks within a task set. All targets shall support abort task set.

To abort a task set, the initiator shall construct a management ORB in system memory for the abort task set function. The initiator shall set the appropriate values in the *rq_fmt* and *login_ID* fields of the ORB, as described in 5.2.4.8. The *function* field shall be set to ABORT TASK SET.

The initiator shall signal the abort task set ORB to the management agent.

Upon receipt of an abort task set request, the target shall perform the following actions:

a) The target shall halt the operation of the fetch agent associated with the *login_ID* by making a transition to the DEAD state;

b) The target shall not issue data transfer requests for any task in the task set whose *login_ID* is equal to that specified in the abort task set request;

c) The target shall wait for response subactions to pending data transfer requests for any task in the task set whose *login_ID* is equal to that specified in the abort task set request;

d) For all tasks for which command execution is complete and whose *login_ID* is equal to that specified in the abort task set request, the target shall wait until the completion status of each command has been successfully stored in system memory or until implementation-dependent retry algorithms have been exhausted in the attempt to store completion status; and

e) When all of the above events have completed, the target shall store completion status for the abort task set request in the status buffer provided. The completion status shall indicate REQUEST COMPLETE.

The initiator shall not reuse the system memory occupied by any of the ORBs, data buffers or page tables of the tasks to be aborted until completion status is returned for the abort task set request.

10.5.3 Logical unit reset

Logical unit reset is a task management function that causes a logical unit to perform the actions described below and to create unit attention conditions for all initiators logged-in to the logical unit. Support for logical unit reset is a target option.

To reset a logical unit, the initiator shall construct a management ORB in system memory for the logical unit reset function. The initiator shall set the appropriate values in the *rq_fmt* and *login_ID* fields of the ORB, as described in 5.2.4.8. The *function* field shall be set to LOGICAL UNIT RESET.

The initiator shall signal the logical unit reset ORB to the management agent.

Upon receipt of a logical unit reset request, the logical unit shall perform the following actions:

a) The target shall halt the operation of all of the logical unit's fetch agents by making transitions to the DEAD state;

b) The target shall not issue data transfer requests for any task in any of the logical unit's task sets;

c) The target shall wait for response subactions to pending data transfer requests for any task in any of the logical unit's task sets;

d) For all of the logical unit's tasks for which command execution is complete, the target shall wait until the completion status of each command has been successfully stored in system memory or until implementation-dependent retry algorithms have been exhausted in the attempt to store completion status;

e) The target shall create (and should attempt to signal *via* unsolicited status) a unit attention condition for all initiators logged-in to the logical unit other than the initiator, identified by *login_ID*, that signaled the logical unit reset request; and

f) When all of the above events have completed, the target shall store completion status for the logical unit reset request in the status buffer provided. The completion status shall indicate REQUEST COMPLETE.

The initiator shall not reuse the system memory occupied by any of the affected ORBs, data buffers or page tables of the tasks until completion status is returned for the target reset request.

10.5.4 Target reset

Target reset is a task management function that causes a target to perform the actions described below and to create unit attention conditions for all logged-in initiators. All targets shall support target reset.

To reset a target, the initiator shall construct a management ORB in system memory for the target reset function. The initiator shall set the appropriate values in the *rq_fmt* and *login_ID* fields of the ORB, as described in 5.2.4.8. The *function* field shall be set to TARGET RESET.

The initiator shall signal the target reset ORB to the management agent.

Upon receipt of a target reset request, the target shall perform the following actions:

a) The target shall halt the operation of all fetch agents for all logical units by making transitions to the DEAD state;

b) The target shall not issue data transfer requests for any task in any task set;

c) The target shall wait for response subactions to pending data transfer requests for any task in any task set;

d) For all tasks for which command execution is complete, the target shall wait until the completion status of each command has been successfully stored in system memory or until implementation-dependent retry algorithms have been exhausted in the attempt to store completion status;

e) The target shall create (and should attempt to signal *via* unsolicited status) a unit attention condition for all logged-in initiators other than the initiator, identified by *login_ID*, that signaled the target reset request; and

f) When all of the above events have completed, the target shall store completion status for the target reset request in the status buffer provided. The completion status shall indicate REQUEST COMPLETE.

The initiator shall not reuse the system memory occupied by any of the affected ORBs, data buffers or page tables of the tasks until completion status is returned for the target reset request.

10.6 Task management event matrix

Common events that affect the state of logical unit fetch agents and their associated task sets are summarized below. Refer to the governing clauses in sections 8 and 9 as well as this section for detailed information.

Event	AGENT_STATE.*st*		Task sets	
	~~Primary~~	~~Secondary~~	~~Primary~~	~~Secondary~~
Power reset	RESET		Clear all task sets	
Command reset (write to RESET_START)	RESET		Clear all task sets	
Bus reset	RESET (logins that are not bridge-aware)	—	Clear all task sets that are not bridge-aware	—
Net update	RESET (logins that are bridge-aware)	—	Clear all bridge-aware task sets	—
Reconnect hold expires	—		Logout the initiator that has failed to successfully reconnect	
Login	—		—	
Create task set	—		—	
Reconnect	—		—	
Logout	RESET		Abort initiator's task set	
Fetch agent reset (write to AGENT_RESET)	RESET		Abort initiator's task set	
Faulted command (CHECK CONDITION)	DEAD		Abort faulted initiator's task set	
ABORT TASK	—		—	
ABORT TASK SET	DEAD		Abort specified task set	
~~CLEAR TASK SET~~	~~DEAD~~		~~Clear all task sets~~	
LOGICAL UNIT RESET	DEAD		Abort all the logical unit's task sets	
TARGET RESET	DEAD		Clear all task sets	
~~TERMINATE TASK~~	~~—~~		~~—~~	

When an event affects more than one task set, all of the associated fetch agents ~~transition to~~ enter the state indicated by the table. With respect to events supported by the target's management agent, *e.g.*, logout, there is an assumption of successful completion. In the case of a function rejected response or other indication of failure, the preceding table does not apply.

Bus resets affect logical unit fetch agents and task sets according to the kind of request (login or create task set) by which the initiator first acquired access privileges.

Immediately upon detection of a bus reset, all command block fetch agents for logins without the *bridge_aware* attribute ~~transition to~~ enter the reset state and their associated task sets are cleared without the return of completion status. The operations of command block fetch agents for logins with the *bridge_aware* attribute and whose initiator is identified by a local node ID are paused until the node IDs for any node handles that refer to nodes on the local bus are updated to reflect changes in physical ID caused by bus reset; once this is complete, fetch agent operations resume without clearing the task set.

Immediately upon detection of a net update, all command block fetch agents for logins whose initiator is identified by a global node ID ~~transition to~~ enter the reset state and their associated task sets are cleared without the return of completion status.

For *reconnect_hold* + 1 seconds subsequent to a bus reset, net update or missed heartbeat, targets save state information for initiators that were logged-in at the time of the event. For bus reset, the timer commences when the target observes the first subaction gap subsequent to a bus reset; if a bus reset occurs before the timer expires, the timer is reset. For net update, the time-out commences when the target's QUARANTINE.*orphan* bit changes from one to zero; if the *orphan* bit changes from zero to one before the time-out expires, the timer is zeroed and restarted when the *orphan* bit is once again zeroed. Otherwise, for missed heartbeat, the time-out commences upon the heartbeat timer's expiration; if net update commences before the *reconnect_hold* the time-out expires, the timer is zeroed, restarted upon the next transition of QUARANTINE.*orphan* from one to zero and thereafter the time-out is managed as described for net update. If an initiator successfully completes a reconnect request during this period, the actions described in 8.6 occur. For command block requests, the task set is empty and, once the fetch agent is initialized, the initiator may signal new ORBs to the ~~target~~ fetch agent.

Once *reconnect_hold* + 1 seconds have elapsed after a bus reset, net update or missed heartbeat, the target shall automatically perform a logout operation for all login IDs and task set IDs that have not been reconnected with their initiator. This returns all the affected fetch agents to the reset state and aborts all the affected task sets.

11 Isochronous operations

11.1 Isochronous operations overview

This section describes procedures that may be used by an initiator to request a logical unit to transfer data associated with a command by isochronous instead of asynchronous methods. Two fundamentally different methods are available to control logical unit isochronous operations:

- Command set-dependent. This provides the most flexible and fully featured control of isochronous operations, since the command set is designed with intimate knowledge of Serial Bus isochronous behavior. Command set-dependent methods are beyond the scope of this standard—although an example of the transport of one such command set is given in Annex D.
- Transport protocol-dependent. This method leverages existing command sets designed without knowledge of Serial Bus isochronous behavior. Because such command sets do not specify all the information necessary to transfer data isochronously, supplemental information available from the transport protocol or other external source is merged with the data transfer length obtained from each command. This approach makes simplifying assumptions and is not as flexible as command set-dependent methods.

The remainder of this section specifies how to use SBP-3 facilities in conjunction with the connection management methods described in IEC 61883-1 to control isochronous operations of devices whose command sets are unaware of the isochronous facilities of Serial Bus. The methods are applicable only to devices that transmit no more than one isochronous output stream and receive no more than one isochronous input stream at a time. Example devices might include DVD players or printers (talkers) and scanners (listeners).

Control of isochronous operations involves the following elements:

- the allocation of Serial Bus resources, such as channel numbers and bandwidth (isochronous resource management);
- the establishment or breaking of connections between the logical unit and the talker or the listeners (connection management);
- the transfer of isochronous data to or from the logical unit's medium (command block ORBs); and
- the starting, stopping and synchronization of isochronous data reception or transmission by the target from or to Serial Bus.

Since there are significant differences between isochronous talkers and listeners, the operational details are described separately in the clauses that follow.

11.2 Talker operations

An isochronous talker is permitted to transmit zero or one isochronous subactions per isochronous period on a specified channel; the maximum data payload of the subaction is constrained both by the transmission speed and by the bandwidth previously allocated for the talker. The maximum data payload permitted at different transmission speeds is specified by IEEE 1394 and, for convenience of reference, is summarized by the table below.

Table 3 – Maximum payload for isochronous subactions

Speed	Maximum data payload (bytes)
S100	1024
S200	2048
S400	4096
S800	8192
S1600	16384
S3200	32768

Since the logical unit's command set has no means to specify channel, speed or maximum data payload, the logical unit shall obtain this information from the output plug control register (oPCR) identified by the configuration ROM Plug_Control_Register entry (whose *direction* bit is one) associated with the logical unit. If there is more than one such entry with a *direction* bit equal to one, unpredictable behavior beyond the scope of this standard may result. The initiator programs the channel and speed information in the oPCR but the maximum data payload is implementation-dependent and provided by the logical unit. Whenever the logical unit is ready to accept data transfer commands, the *online* bit in the oPCR shall be one and the *payload* field shall report the logical unit's maximum data payload per isochronous subaction, in quadlets. The size reported by the *payload* field shall not exceed the maximum data payload permitted for the speed reported by the *data_rate_capability* field in the target's output master plug register (oMPR).

NOTE – A *payload* value of zero encodes a size of 1024 quadlets, per IEC 61883-1.

Before an initiator signals data transfer commands to a logical unit within ORBs whose *isochronous* bit is one, it shall allocate the necessary isochronous resources and program the logical unit's oPCR as specified below:

a) The initiator shall read the target's oMPR to determine the fastest transmission speed supported and shall read the logical unit's oPCR to determine the maximum data payload;

b) The initiator shall select a transmission speed[10] less than or equal to the fastest speed supported by the target and shall adjust the maximum data payload to the smaller of the value obtained from the logical unit's oPCR and the maximum permitted at the selected transmission speed. The initiator shall attempt to obtain the necessary bandwidth for the data payload from the isochronous resource manager's BANDWIDTH_AVAILABLE register. If the bandwidth is unavailable, the initiator shall not signal any command block ORBs to the logical unit with an *isochronous* bit equal to one;

c) Otherwise, the initiator shall attempt to allocate a channel from the isochronous resource manager's CHANNELS_AVAILABLE register. If no channel is available, the initiator shall release the bandwidth previously obtained and shall not signal any command block ORBs to the logical unit with an *isochronous* bit equal to one;

d) Once both bandwidth and channel have been allocated, the initiator shall program the logical unit's oPCR with channel number and speed and shall increment the point-to-point connection count in accordance with the procedures specified by IEC 61883-1. If the speed selected by the initiator limits the maximum data payload to a value smaller than that reported by the logical unit in the

[10] The choice of transmission speed is influenced both by Serial Bus topology between the target (talker) and one or more listeners and by the speed capabilities of the listeners.

oPCR *payload* field, the logical unit shall update the *payload* field with the maximum data payload permitted for the selected speed.

After the oPCR has been programmed with channel number and speed and the point-to-point connection count is nonzero, the logical unit is ready to accept command block ORBs whose *isochronous* bit is one. Because the data shall be transmitted isochronously, the ORB shall not specify the address of a buffer but shall specify a total *data_size* for the data to be transmitted by the command; the *direction* bit in the ORB shall be one (see 5.2.3). The logical unit shall transmit the requested data on Serial Bus as specified by 9.5. When all of the requested data, up to the limit of *data_size*, has been transmitted, the logical unit shall store completion status for the ORB at the initiator's *status_FIFO*.

When the initiator has concluded isochronous operations with the logical unit (either because logout is imminent or in anticipation of substantial logical unit idle time), the initiator shall follow the procedures specified by IEC 61883-1 to program the logical unit's oPCR and decrement the point-to-point connection count. The initiator shall also release the isochronous resources, bandwidth and channel number, previously allocated.

11.3 Listener operations

An isochronous listener expects to receive zero or one isochronous subactions per isochronous period on a specified channel; the maximum data payload of the subaction is constrained by the reception speed as specified by IEEE 1394 and summarized in Table 3.

Since the logical unit’s command set has no means to specify channel number, the logical unit shall obtain this information from the input plug control register (iPCR) identified by the configuration ROM Plug_Control_Register entry (whose *direction* bit is zero) associated with the logical unit. If there is more than one such entry with a *direction* bit equal to zero, unpredictable behavior beyond the scope of this standard may result. The initiator programs the channel number information in the iPCR. Whenever the logical unit is ready to accept data transfer commands, the *online* bit in the iPCR shall be one.

Before an initiator signals data transfer commands to a logical unit within ORBs whose *isochronous* bit is one, it shall either allocate the necessary isochronous resources or determine that they have been allocated and program the logical unit's iPCR as specified below:

a) The initiator shall read the target's input master plug register (iMPR) to determine the fastest reception speed supported and shall examine Serial Bus topology between the talker and the target (listener) to determine the fastest speed isochronous subactions may be transmitted from the talker to the target;

b) If the initiator controls the talker, it shall configure it to transmit isochronous subactions no faster than the speed determined in the preceding step. Otherwise, the initiator shall determine the speed at which the talker is transmitting or will transmit isochronous subactions; if it is greater than speed determined in the preceding step, the initiator shall not signal any command block ORBs to the logical unit with an isochronous bit equal to one;

c) If the initiator does not control the talker it shall skip this step. Otherwise, it shall determine the maximum data payload for the talker's isochronous subactions and shall attempt to obtain the necessary bandwidth for the data payload from the isochronous resource manager's BANDWIDTH_AVAILABLE register. If the bandwidth is unavailable, the initiator shall not signal any command block ORBs to the logical unit with an isochronous bit equal to one;

d) If the initiator does not control the talker it shall skip this step. Otherwise, it shall attempt to allocate a channel from the isochronous resource manager's CHANNELS_AVAILABLE register. If no channel is available, the initiator shall release the bandwidth previously obtained and shall not signal any command block ORBs to the logical unit with an isochronous bit equal to one;

e) Otherwise, the initiator shall attempt to allocate a channel from the isochronous resource manager's CHANNELS_AVAILABLE register. If no channel is available, the initiator shall release previously obtained bandwidth, if any, and shall not signal any command block ORBs to the logical unit with an isochronous bit equal to one;

f) Once both bandwidth and channel have been allocated (whether by the initiator or another device), the initiator shall program the logical unit's iPCR with channel number and shall increment the point-to-point connection count in accordance with the procedures specified by IEC 61883-1.

After the iPCR has been programmed with channel number and the point-to-point connection count is nonzero, the logical unit is ready to accept command block ORBs whose *isochronous* bit is one. Because the data shall be received isochronously, the ORB shall not specify the address of a buffer but shall specify a total *data_size* for the data to be received by the command; the *direction* bit in the ORB shall be zero (see 5.2.3). The logical unit shall receive the requested data from Serial Bus as specified by 9.5. When all of the requested data, up to the limit of *data_size*, has been received, the logical unit shall store completion status for the ORB at the initiator's *status_FIFO*.

When the initiator has concluded isochronous operations with the logical unit (either because logout is imminent or in anticipation of substantial logical unit idle time), the initiator shall follow the procedures specified by IEC 61883-1 to program the logical unit's iPCR and decrement the point-to-point connection count. If the initiator previously allocated isochronous resources, bandwidth and channel number, it shall release them.

11.4 Implementation recommendations (informative)

The following suggestions are intended as useful guidance for implementers. They ~~may~~ might not be the most appropriate choices for all command sets. For example, a device whose command set is cognizant of data streaming requirements may find it preferable to ignore certain isochronous errors rather than abort tasks and task sets.

- The *online* bits in the logical unit's plug control registers should be zero when no initiator is logged in;
- An initiator should perform an exclusive login before programming a logical unit's plug control registers;
- Logical units should ignore missed cycle start indications. Talkers may transmit the data during the next isochronous period; listeners that implement "loose isochronous" reception (as permitted by IEEE Std 1394a-2000) are likely to receive any isochronous subaction intended for them even if they fail to observe the cycle start subaction;
- A talker that detects data underrun during transmission or a listener that detects data overrun during reception of an isochronous subaction should abort the task and the task set to which it belongs;
- An initiator receiving data from a talking logical unit should, once completion status has been stored at the initiator's *status_FIFO*, verify that the quantity of data received is equal to the quantity expected. If there is a mismatch, the initiator should abort the task set and reissue the command;
- An initiator transmitting data to a listening logical unit should time the completion of the command in order to detect isochronous data transfer errors. This is because the listening logical unit cannot determine when all data has been transferred except when the ORB *data_size* field is reached or exceeded. An exception to this recommendation exists when the data format used by the logical unit's command set is self-descriptive and the logical unit is capable of parsing the received data to autonomously determine when data transfer is complete.

The preceding recommendations have been made from the point of view that when the logical unit is the talker, the initiator is the listener and vice versa—but this is not a requirement. The initiator may be neither talker nor listener and, if the logical unit is a listener, ~~may~~ is not necessarily be in control of the talker.

Annex A
(normative)

Minimum Serial Bus node capabilities

In addition to the minimum capabilities defined by IEEE 1394, this annex specifies other capabilities that an initiator or a target shall support in order to implement SBP-3.

Once a node that implements one or more initiators or targets completes its power reset initialization sequence, it shall acknowledge, and subsequently respond to, Serial Bus transaction request subactions within the time limits specified by IEEE 1394. A Serial Bus reset shall not alter a node's responsiveness to request subactions.

A.1 Initiator capabilities

With the exception of configuration ROM and control and status registers, an initiator shall be capable of responding to block read or write requests with a *data_length* less than or equal to 32 bytes.

An initiator shall also be capable of responding to block read requests with a *data_length* less than or equal to 4 * *ORB_size*, where *ORB_size* is obtained from the Unit_Characteristics entry in the target's configuration ROM.

For the largest value of *max_payload* specified in any command block ORB signaled to the target, the initiator shall be capable of responding to block read and write requests with a *data_length* less than or equal to $2^{max\_payload + 2}$ bytes.

The initiator shall report the largest of these possible *data_length* values by setting the value of the *max_rec* field in the bus information block in its configuration ROM to a value equal to or greater than ($\log_2$ *data_length*) - 1.

A.2 Target capabilities

A target shall be capable of responding to block read or write requests with a *data_length* equal to eight bytes if the *destination_offset* specifies either the MANAGEMENT_AGENT or the ORB_POINTER register.

A target shall be capable of initiating write requests and shall report this by setting the *drq* bit in the Node_Capabilities entry in configuration ROM to one. Consequently, the target shall implement the *dreq* bit in the STATE_CLEAR and STATE_SET registers. The value of STATE_CLEAR.*dreq* shall be unaffected by a Serial Bus reset. The target may automatically set *dreq* to zero (request initiation enabled) upon a power reset or a command reset.

A target shall be capable of initiating block write requests with a *data_length* of at least eight bytes and shall report this by setting the value of the *max_rec* field in the bus information block in configuration ROM to a value greater than or equal to two.

While initializing after a power reset, a target shall respond to quadlet read requests addressed to FFFF F000 0400_{16} with either a response data value of zero or acknowledge the request subaction with *ack_tardy*, as specified by IEEE Std 1394a-2000. This indicates that the remainder of configuration ROM, as well as other target CSRs, are not accessible.

A.3 Target security

As mandated by IEEE 1394, a target shall abide by the following restrictions:

- If a target's unique ID, EUI-64, is read from the configuration ROM bus information block by quadlet read requests, the value returned shall be the EUI-64 assigned by the manufacturer;
- The target shall not originate a request or response subaction with a *source_ID* field that is not equal to either a) the most significant 16 bits of the target's NODE_IDS register or b) the concatenation of $3FF_{16}$ and the physical ID assigned to the target's PHY during the self-identify process; and
- The target shall not receive a request or response subaction that specifies *destination_ID* unless that field is equal to either a) the concatenation of the most significant 10 bits of the target's NODE_IDS register and either the physical ID assigned to the target's PHY during the self-identify process or $3F_{16}$, or b) the concatenation of $3FF_{16}$ and either the physical ID assigned to the target's PHY during the self-identify process or $3F_{16}$.

Annex B
(normative)

SCSI command and status encapsulation

SBP-3 defines a protocol that permits initiators to control the operation of devices (disks, tapes, printers, *etc.*), but it does not specify the command sets used by the devices—only the mechanisms by which commands, data and status are transported. This annex specifies how SBP-3 may be used for devices that use the SCSI command sets. This encompasses encapsulation of SCSI command descriptor blocks (CDBs), a standard format for SCSI status and sense data and the necessary configuration ROM entries.

B.1 SCSI command descriptor block

SBP-3 provides for the transport of 6-, 10-, 12- and 16-byte SCSI CDBs within ~~a~~ single buffer descriptor or dual buffer descriptor command block ORBs~~, as~~ (a single buffer descriptor ORB is illustrated by Figure B-1). There is no fundamental limit on the size of a command block, although many targets implement a 32-byte ORB (illustrated by the shaded area). When CDBs encapsulated within an ORB do not occupy all of the command-dependent portion of the ORB, the least significant (unused) bytes of the ORB shall be zero.

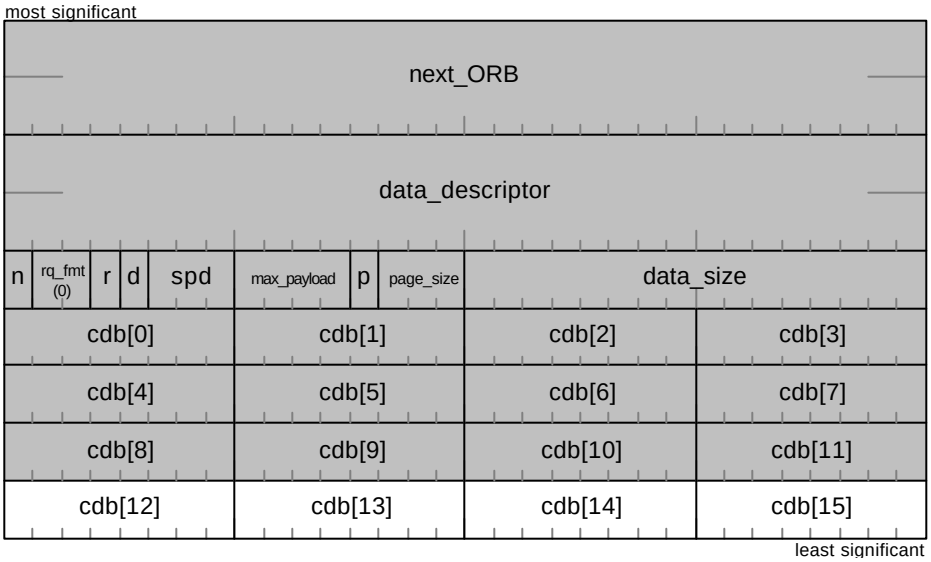

Figure B-1 – SCSI command block ORB

Parts of the control byte (the last byte of a SCSI command descriptor block) are constrained to values illustrated by Figure B-2.

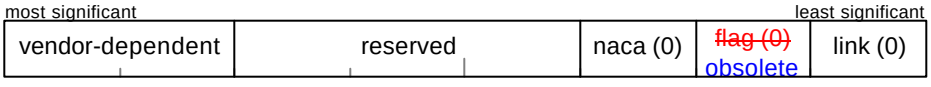

Figure B-2 – SCSI control byte

The *naca*, or normal ACA, bit shall be zero; SBP-3 does not support~~s~~ SAM-2 auto contingent allegiance.

The ~~*flag* and~~ *link* bit~~s~~ shall be zero; SBP-3 does not support linked commands.

B.2 SCSI status and sense data

Upon completion of a command, if the *notify* bit in the ORB is one or if there is exception status to report, the target shall signal the initiator by storing ~~all or part of the~~ a status block in either of the formats shown by Figure B-3 and Figure B-4 at the *status_FIFO* address provided by the initiator as part of the login request.

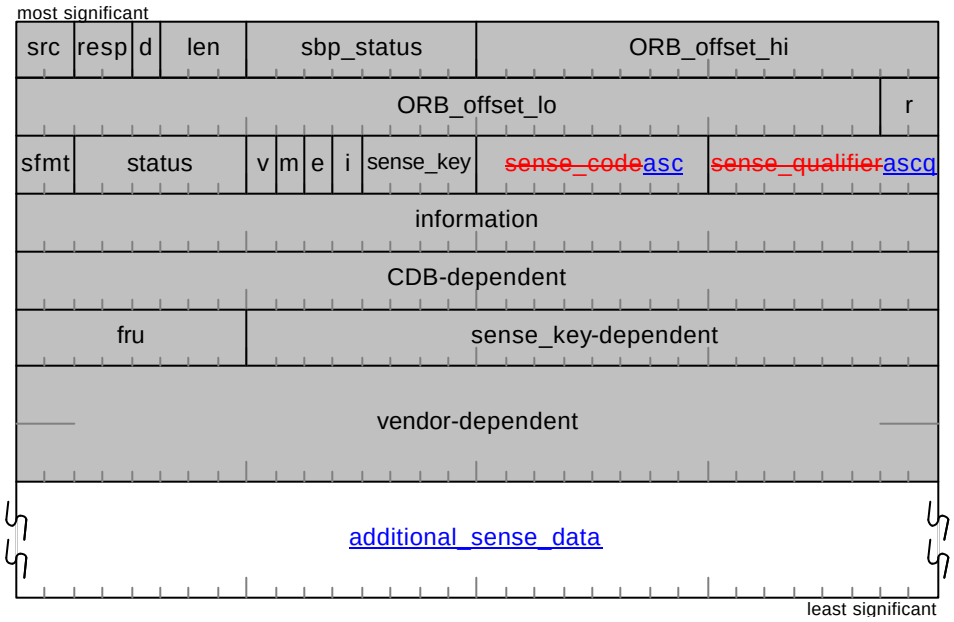

Figure B-3 – Status block ~~format~~ for fixed format SCSI sense data

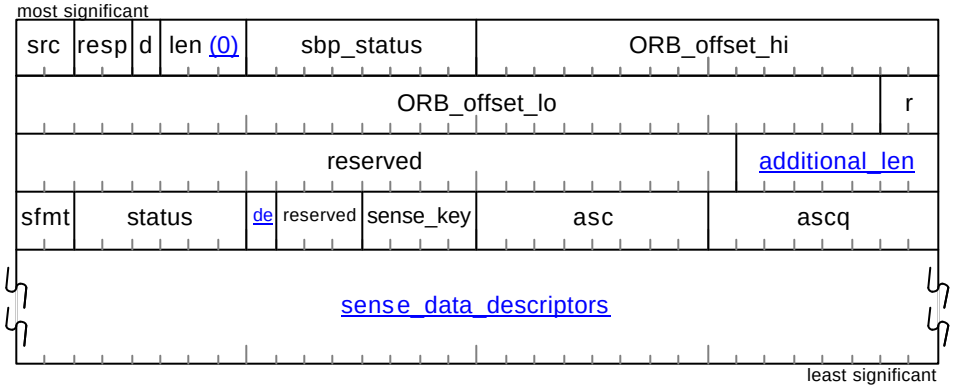

Figure B-4 – Status block for descriptor format SCSI sense data

When a command completes with GOOD status, only the first two quadlets of the status block shall be stored at the *status_FIFO* address; the *len* field shall be one. Otherwise, both SCSI status and sense data shall be stored in a status block that conforms to one of the formats illustrated above.

The *src*, *resp*, *len*, *sbp_status*, *ORB_offset_hi*, *ORB_offset_lo* and *additional_len* (present only in an extended status block) fields, as well as the *dead* bit (abbreviated as *d* in the figure above), are as previously described in 5.4.

When *len* is nonzero, SBP-3 permits the return of a status block between two and eight quadlets in length. ~~When~~ If ~~a truncated~~ status block smaller than eight quadlets is stored, the omitted quadlets shall be interpreted as if zero values were stored. Otherwise, when *len* is zero, the size of the extended status block is described by *additional_len* (see Figure B-4).

The *sfmt* field shall specify the format of the status block and shall additionally indicate whether the error condition associated with *sense_key* is current or deferred. The table below defines permissible values for *sfmt*.

Value	Description
0	Current error; fixed format status block defined by this standard
1	Deferred error; fixed format status block defined by this standard
2	~~Reserved for future standardization~~ Descriptor format status block defined by this standard
3	Vendor-dependent status block format

The *status* field shall contain SCSI status information as defined by SAM-2, with the exceptions noted in the table below.

Value	Description
0	GOOD
2	CHECK CONDITION
4	CONDITION MET
8	BUSY
10_{16}	Not supported by SBP-3
14_{16}	Not supported by SBP-3
18_{16}	RESERVATION CONFLICT
22_{16}	~~COMMAND TERMINATED~~ Obsolete
28_{16}	Not supported by SBP-3
30_{16}	Not supported by SBP-3
All other values	Reserved for future standardization

The *valid* bit (abbreviated as *v* in Figure B-3~~the figure above~~) shall specify the content of the *information* field. When the *valid* bit is zero, the contents of the information field are not specified. When the *sfmt* field has a value of zero or one and the *valid* bit is one, the contents of the information field shall be as defined by SPC-2 or the relevant command set standard.

The meanings of the *mark*, *eom* and *illegal_length_indicator* bits (abbreviated as *m*, *e* and *i*, respectively, in Figure B-3~~the figure above~~) are defined by SPC-2 or the relevant the command set standard. These bits correspond to the filemark, EOM and ILI bits defined by SPC-2 for sense data.

When the *deferred error* bit (abbreviated as *de* in Figure B-4) is one, the sense data describes a deferred error. Otherwise the sense data describes a current error.

The *sense_key*, ~~*sense_code*~~ *asc* and ~~*sense_qualifier*~~ *ascq* fields shall contain command completion information defined by SPC-2 or the relevant the command set standard. These fields correspond to the sense key, additional sense code and additional sense code qualifier fields defined by SPC-2 for sense data.

The contents of the *information* field are unspecified if either the *valid* bit is zero or the *sfmt* field has a value of three. For *sfmt* values of one or two, the contents of the *information* field are device-type or command dependent and, if the *valid* bit is one, are defined within SPC-2 or the appropriate standard for the command. Characteristic uses of the *information* field are for:

- the unsigned logical block address associated with *sense_key* and the command; or
- the least significant 32-bits of the unsigned logical block address associated with *sense_key* and the command; or
- the residue of the requested data transfer length minus the actual data transfer length, in either bytes or blocks as determined by the command. Negative values are indicated in two's complement notation.

The contents of the *CDB*-dependent field (which corresponds to the SPC-2 sense data command-specific information field) are device-type or command dependent and are defined within SPC-2 or the appropriate standard for the command.

~~Nonzero values in~~ The *fru* field corresponds to the field replaceable unit code field defined by SPC-2 for sense data ~~may be used to identify a device-dependent, field replaceable mechanism or unit that has failed. A value of zero in this field shall indicate that no specific mechanism or unit has been identified to have failed or that the data is unavailable. When *fru* is nonzero, the format of the information is not specified by this standard.~~

When *sfmt* has a value of zero or one, the contents of the *sense_key*-dependent field (which corresponds to the SPC-2 sense data sense key-specific field) are defined by SPC-2 or the relevant command set standard. In this case the most significant bit of the *sense_key*-dependent field is the SKSV bit defined by SPC-2. When *sfmt* is equal to three, the contents of *sense_key*-dependent are unspecified.

The *additional_sense_data* field, when present, shall contain additional sense bytes defined by SPC-2 or the relevant command set standard. The presence of *additional_sense_data* requires an extended status block; the quantity of *additional_sense_data* may be derived from *additional_len*.

NOTE – When SCSI sense data is contained within an extended status block, the sense data is offset by one quadlet because of the presence of the additional_len field. This is not shown in Figure B-3, but an example is given by Figure B-4.

The sense_data_descriptors field, when present, shall contain one or more sense data descriptors as defined by SPC-3 [B16] for descriptor format sense data.

B.3 Configuration ROM

SCSI targets shall implement configuration ROM in accordance with section 7 and this annex. At least one logical unit, logical unit zero, shall be implemented; additional logical units may be implemented. A logical unit is described by entries in a unit directory or by entries in a logical unit directory dependent upon the unit directory or by entries taken in combination from both places.

B.3.1 Command_Set_Spec_ID entry

The Command_Set_Spec_ID entry is an immediate entry in either a unit or logical unit directory that specifies the organization responsible for the command set definition for the target. The format of this entry is specified by 7.8.5.

SCSI targets shall have a *command_set_spec_ID* value of $00\ 609E_{16}$, which indicates that INCITS is responsible for the command set definition.

B.3.2 Command_Set entry

The Command_Set entry is an immediate entry in either a unit or logical unit directory that, in combination with the *command_set_spec_ID*, specifies the command set implemented by the target. The format of this entry is specified by 7.8.6.

SCSI targets shall have a *command_set* value of $01\ 04D8_{16}$, which indicates that the target's command set is specified by SCSI Primary Commands 2 (SPC-2) and related command set standards—as determined by the target's peripheral device types. In addition, this *command_set* value specifies that the target conforms to all requirements of this annex.

B.3.3 Logical_Unit_Number entry

The Logical_Unit_Number entry is an immediate entry in either a unit or logical unit directory that specifies the peripheral device type and logical unit number of a logical unit implemented by the SCSI target. The format of this entry is specified by 7.8.15.

The *device_type* field indicates the peripheral device type implemented by the logical unit. This field shall contain a value specified by the table below.

Value	Peripheral device type
$0 – 1E_{16}$	The value of *device_type* shall have the same meaning as the peripheral device type field returned in INQUIRY data as specified by SPC-2
$1F_{16}$	Unknown device type

Annex C
(normative)

Security extensions

SBP-3 specifies an access protocol, in section 8, that by itself makes no provisions for security. This annex defines extensions to SBP-3 that may be implemented by targets to provide some measure of security. Targets that implement these security extensions shall conform to all provisions of this annex.

Conformance to this annex does not preclude additional, command set-dependent security facilities.

C.1 Passwords

A target shall implement two passwords:

- The master password, which shall be unchangeable and equal to the target serial number. The target serial number should be in a humanly readable form affixed to the target. The master password shall not be readable *via* the target's Serial Bus interface except by a logged-in initiator; and
- The current password, which shall accommodate 28 bytes of password data and shall be alterable only by the set password function (see clause C.3).

All password values shall be unchanged by power reset, bus reset or command reset.

The value of the master password shall be obtainable by command set-dependent means.

A target may be manufactured with a current password of all zeros, with the expectation that the user assign a nonzero current password as part of target initialization. If a target is manufactured with a nonzero current password, the target shall be shipped with the current password in a humanly readable form.

C.2 Login

The description of the login protocol below reproduces that specified by section 8 and adds validation of cumulative login attempts and the *password* field from the login request. The target shall implement an internal counter, *login_attempts*, which shall be zeroed upon a power reset or upon a successful login or logout request. The target shall perform the following, in the order specified, to validate a login request:

a) If the *source_ID* from the write request used to signal the login ORB to the target's MANAGEMENT_AGENT register contains a global node ID but the target does not implement bridge-aware capabilities, the target shall respond with a type error;

b) If *login_attempts* is equal to three, the target shall reject the login request with an *sbp_status* of access denied and shall not increment *login_attempts*;

c) In cases where *source_ID* is local, the *aware* bit is set in the login ORB and the target does not implement bridge-aware capabilities, the target shall reject the login request with an *sbp_status* of function rejected but *login_attempts* shall not be incremented;

d) If *source_ID* contains a global node ID and the target implements bridge-aware capabilities, the target shall examine the *aware* bit in the login ORB and, if zero, shall reject the login with an *sbp_status* of function rejected but *login_attempts* shall not be incremented. When the login specifies a global node ID and the *aware* bit is one, the target shall use a TIMEOUT request, as defined by draft standard IEEE P1394.1, to obtain the EUI-64 of the initiator and its remote timeout information;

e) Otherwise *source_ID* is local and the target shall read the initiator's unique ID, EUI-64, from the bus information block by means of two quadlet read transactions. The *source_ID* from the write transaction used to signal the login ORB to the target's MANAGEMENT_AGENT register shall be used as the *destination_ID* in the quadlet read transactions;

f) If the *update* bit in the login ORB is zero, the target shall determine whether or not the initiator already owns a login by comparing the EUI-64 just obtained against the *login_owner_EUI_64* for all *login_descriptors*. If the initiator is currently logged-in to the same logical unit, the login request shall be rejected with an *sbp_status* of access denied but *login_attempts* shall not be incremented. Otherwise, when the *update* bit is one, the target shall verify that the initiator owns the login identified by *login_ID* and, if not, shall reject the login request with an *sbp_status* of invalid login ID but *login_attempts* shall not be incremented;

g) The target shall validate the password provided by the login request. If *password_length* is zero, the password is eight bytes of immediate data present in the *password* field. Otherwise *password_length* specifies the size of the password addressed by *password*. If *password_length* is greater than 28 the target shall increment *login_attempts* and reject the login request with an *sbp_status* of access denied. When *password_length* is valid, the password provided is extended to 28 bytes by the addition of least significant bytes of zeros; the result is compared with the target's passwords. If the password provided matches neither the target's current nor its master password, the *login_attempts* count shall be incremented and the login request shall be rejected with an *sbp_status* of access denied;

h) If the *exclusive* bit is set in the login ORB, the target shall reject the login request (with an *sbp_status* of access denied) if there are any active *login_descriptors* for the logical unit (other than one whose *login_owner_EUI_64* matches the EUI-64 of the initiator requesting the login) but shall not increment *login_attempts*;

i) If an active *login_descriptor* with the *exclusive* attribute exists for the *lun* specified in the login ORB (other than one whose *login_owner_EUI_64* matches the EUI-64 of the initiator requesting the login), the target shall reject the login request (with an *sbp_status* of access denied) but shall not increment *login_attempts*; else

j) If the *update* bit in the login ORB is zero, the target shall determine if a free *login_descriptor* is available and, if none are available, reject the login request with an *sbp_status* of resources unavailable. Otherwise, the target shall determine whether or not the initiator already owns a login by comparing the EUI-64 obtained by either a TIMEOUT request or configuration ROM read against *login_owner_EUI_64* for all *login_descriptors*. If the initiator is not logged-in to the logical unit identified by *lun*, the target shall reject the login request with an *sbp_status* of invalid login ID.

If the *update* bit is zero, once the above conditions have been met and a *login_descriptor* allocated, the initiator's *source_ID* is stored in *login_owner_ID*, the initiator's EUI-64 is stored in *login_owner_EUI_64*, the *lun* and *status_FIFO* fields from the login ORB are stored in the *login_descriptor*, the *bridge_aware* and *exclusive* variables in the *login_descriptor* are set to the values of the *aware* and *exclusive* bits, respectively, from the login ORB and the address of the fetch agent and the *reconnect_hold* value chosen by the target are stored in the *login_descriptor*. If the *bridge_aware* variable is true, the target allocates a node handle to the initiator (the process is essentially the same as described by 8.4.2). Lastly the target assigns a unique *login_ID* to this login and stores it in the *login_descriptor*.

When the *update* bit is one, the login request permits the initiator to change parameters associated with the login. If the login request meets all of the validation requirements described above, the target shall logout the initiator (see 8.7) without returning completion status and then shall process the login in accordance with the requirements of this clause. The target shall perform these two steps such that no other initiator is afforded an opportunity to login between the time that target resources have been released and the time the update login request completes.

If the target is able to satisfy the login request, it shall zero *login_attempts* and return a login response as specified in 5.2.4.2. When the *update* bit in the login ORB is one, the information returned in the login response may differ from that previously associated with the login.

C.3 Set password

In order to change a target's current password, an initiator may use a management ORB with the format shown below.

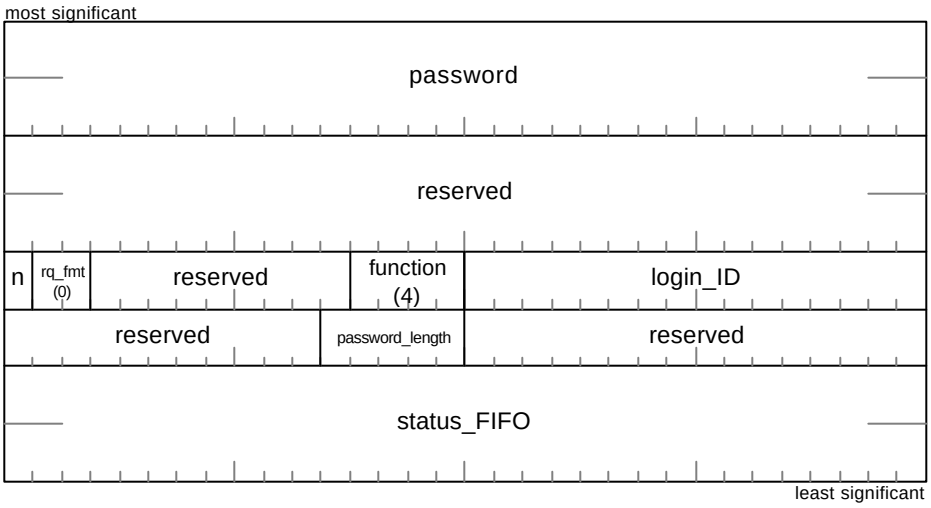

Figure C-1 – Set password ORB

The *password* and *password_length* fields contain the new value for the current password. If *password_length* is zero, the *password* field contains immediate data. When *password_length* is nonzero, the *password* field shall conform to the format for address pointers specified by Figure 10 and shall contain the address of a buffer. The maximum value of *password_length* shall be 28. The buffer shall be in the same node as the initiator and shall be accessible to a Serial Bus block read request with a data transfer length less than or equal to *password_length*.

The *notify* bit and the *rq_fmt* and *function* fields are as previously defined for management ORB formats.

The *login_ID* field shall contain a login ID value obtained as the result of a successful login.

The *status_FIFO* field shall contain an address allocated for the return of status for the SET PASSWORD request, only. The contents of this field shall not update the status FIFO address established by the successful login that returned *login_ID*.

If *login_ID* specifies a valid current login for the initiator that signaled the SET PASSWORD request to the target's MANAGEMENT_AGENT register, the target shall update the current password to the new value specified by the set password request. The target shall not return completion status for the request unless either the request is rejected or the new password has been successfully stored such that it will not be affected by any subsequent power reset, bus reset or command reset.

Annex D
(normative)

AV/C Encapsulation

Devices that use the AV/C Digital Interface Command Set use IEC 61883-1 ~~(1998-02)~~ Function Control Protocol (FCP) for the encapsulation of command and status. This annex specifies how SBP-3 may be utilized as the transport layer in place of FCP.

AV/C devices are consumer electronic devices such as camcorders, VCRs, stereo tuners and televisions—this is not an exhaustive list. Their commands, status and operations are standardized by the 1394 Trade Association Specification for AV/C Digital Interface Command Set General Specification~~, Version 4.0, July 23, 2001~~ [B1] and related 1394 Trade Association specifications. These devices present or accept most of their data over Serial Bus isochronous channels; asynchronous requests are used for control information.

D.1 Logical unit, unit and subunit models

SBP-3 describes a hierarchical device structure composed of units with subordinate logical units. AV/C has a hierarchical structure of units with subordinate subunits.

An AV/C unit is described by configuration ROM entries in a unit directory and is implemented as a single logical unit. AV/C subunits are not visible as SBP-3 objects; except for task sets that result from create task set operations, a single task set is maintained for commands for all the subunits of an AV/C unit.

D.2 AV/C command sequence

An AV/C command sequence consists of a command frame delivered to the AV/C device and one or two response frames returned to the controller. Return of the final response frame is indicated when ORB completion status is stored at the *status_FIFO* associated with the login; an interim response may be signaled by interim status stored at the same *status_FIFO*. Both command and response frames are variable-length data structures between four and 512 bytes in length, which shall be described by a command block ORB with dual buffer descriptors, as illustrated below.

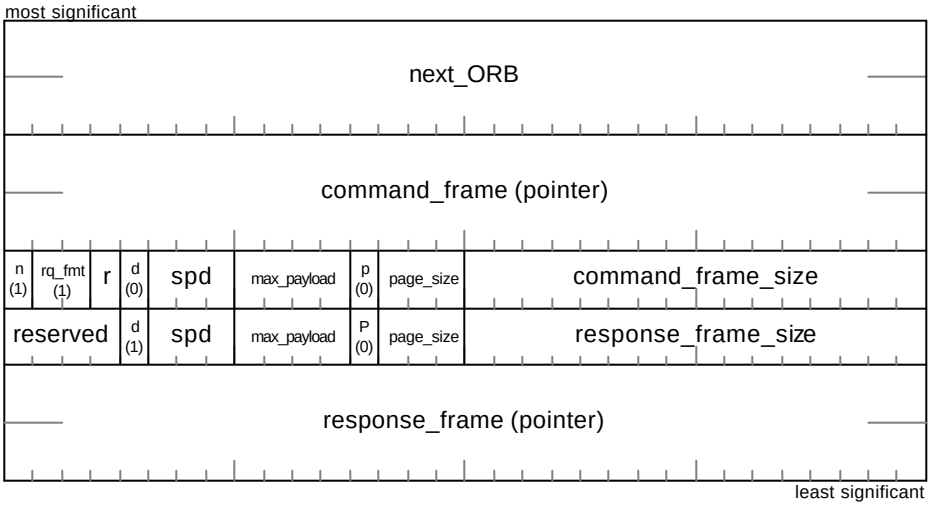

Figure D-1 – AV/C command sequence ORB

The *next_ORB*, *rq_fmt*, *spd*, *max_payload*, and *page_size* fields and the *notify*, *direction* and *page_table_present* bits (abbreviated as *n*, *d*, and *p*, respectively, in the figure above) are as specified in 5.2.3.

The first buffer descriptor in the ORB, *command_frame*, references a buffer that contains an AV/C command frame in the format specified by IEC 61883-1. The *direction* bit associated with this buffer descriptor shall be zero. Page tables are not used for AV/C command frames; the *page_table_present* bit shall be zero and *command_frame_size* shall be less than or equal to 512.

The second buffer descriptor in the ORB, *response_frame*, references a buffer into which the AV/C device may store a response frame in the format specified by IEC 61883-1. The *direction* bit associated with this buffer descriptor shall be one. Page tables are not used for AV/C response frames; the *page_table_present* bit shall be zero and *response_frame_size* shall be a multiple of eight and less than or equal to 1024.

AV/C commands shall return a final response frame and may return an interim response frame. Both response frames are stored in the buffer described by the *response_frame* field. Final response frames shall be stored at relative offset zero within the buffer. Interim response frames shall be stored at relative offset *response_frame_size* / 2 within the same buffer. If *response_frame _size* / 2 is less than the size of either the final or interim response frame, the target shall not store an interim response. If no interim response is stored, the entire buffer may be used for the final response.

NOTE – The maximum response frame size permitted by AV/C is 512 bytes. An initiator that provides a response frame buffer of 1024 bytes guarantees that there is sufficient space for both an interim and a final response frame.

Because the entire AV/C command sequence consists of the command and response frames, there is no command-dependent information in the ORB.

D.3 AV/C status

Upon completion of an AV/C command and after the final response frame has been stored, the target shall signal the initiator by storing the status block shown by Figure D-2 at the *status_FIFO* address provided by the initiator as part of the login request. Prior to the return of a final response, the target may also signal interim status in the same format to the same *status_FIFO*.

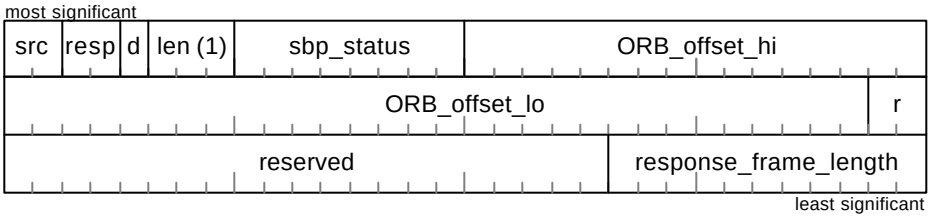

Figure D-2 – Status block format AV/C command sequence

The *src*, *resp*, *len*, *sbp_status*, *ORB_offset_hi* and *ORB_offset_lo* fields, as well as the *dead* bit (abbreviated as *d* in the figure above), are as previously described in 5.4; the *len* field shall be one.

AV/C devices use the *src* field as specified by the table below.

Value	Description
0	Final response frame
1	
2	Not used by AV/C devices
3	Interim response frame

The *response_frame_length* field shall specify the size of the response frame stored by the AV/C command in the buffer provided by the command sequence ORB.

NOTE – The offset of the response frame within the buffer is determined by *src*.

D.4 Configuration ROM

The sample bus information block and root directory for basic targets illustrated in Annex F are equally applicable to targets that use the AV/C command set and are not repeated below. Figure D-3 shows an example of a unit directory for an AV/C device.

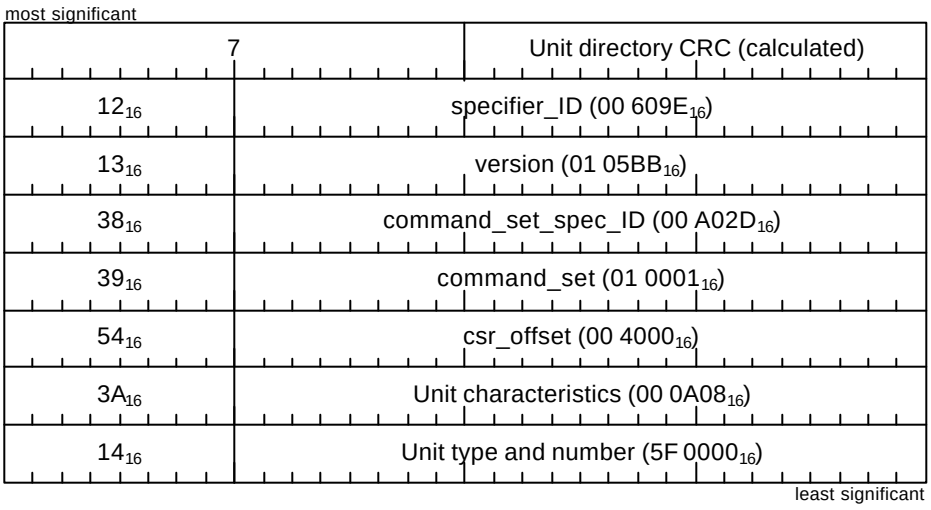

Figure D-3 – AV/C unit directory

The Command_Set_Spec_ID and Command_Set_Version entries, with a *key* field of 38_{16} and 39_{16}, respectively, specify that the target uses the AV/C command set defined by the 1394 Trade Association.

The Management_Agent entry in the unit directory, with a *key* field of 54_{16}, has a *csr_offset* value of 00 4000_{16} that indicates that the MANAGEMENT_AGENT register has a base address of FFFF F001 0000_{16} within the node's memory space.

The Unit_Characteristics entry in the unit directory, with a *key* field of $3A_{16}$, has an immediate value of 00 $0A08_{16}$. This indicates a target is expected to complete a login within five seconds and fetches 32-byte ORBs.

The Logical_Unit_Number entry in the unit directory, with a *key* field of 14_{16}, has an immediate value of 5F 0000_{16}; this identifies logical unit zero and indicates that AV/C commands are used to query the number and type of subunits associated with the unit. It also indicates a target that implements the basic task management model and executes commands in the order queued.

D.5 Operations

Control of an AV/C device implemented with SBP-3 follows the steps described in sections 8 and 9. Subsequent to a successful login, the controller may create a queue of one or more ORBs that describe AV/C command frames and signal these to the target. As the commands complete, the response frames are returned in the buffers provided by the controller. The target stores status to signal the controller that a response frame has been returned.

Annex E
(normative)

Isochronous data interchange format

Isochronous data stored on the medium may be kept in a form similar to the format of isochronous packets on Serial Bus, but the *tcode* field present in Serial Bus packets is reused to identify the type of recorded isochronous data. Three different packet formats may be present in recorded isochronous data, encoded by *tcode* as shown below.

Value	Name	Description
8	CYCLE MARK	Marks the time of a cycle start event
A_{16}	DATA	Isochronous data packet
E_{16}	NULL	Null (or filler) packet
All other values	—	Reserved for future standardization

The values used to indicate CYCLE MARK and DATA are identical to the *tcode* values defined for Serial Bus cycle start packets and isochronous data packets, respectively.

E.1 Cycle marks

Whenever a cycle start packet is observed on Serial Bus for an enabled isochronous stream, a CYCLE MARK packet shall be recorded on the medium. The CYCLE MARK packet is a single quadlet that stores the time transported by the cycle start packet, as shown by the figure below.

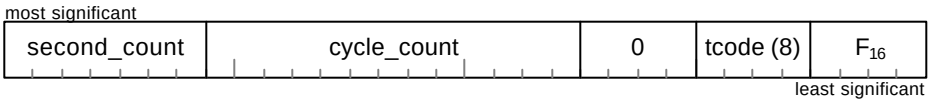

Figure E-1 – CYCLE MARK format

The *second_count* and the *cycle_count* fields shall contain the values of the corresponding fields from the most recently observed cycle start packet. No more than one CYCLE MARK packet shall be recorded for a single cycle start packet.

NOTE – The time information in the CYCLE MARK packet is not necessary for a logical unit to recreate an isochronous stream during playback, but it may be useful to applications that search for known time and cycle boundary locations in recorded isochronous data.

The *tcode* field shall be equal to eight.

When a logical unit is a listener and detects a missed isochronous period, it shall synthesize and record a CYCLE MARK packet on the medium. The *second_count* and *cycle_count* values shall be taken from the target's free-running cycle timer.

E.2 Isochronous data packets

The format of an isochronous data packet recorded on the medium is illustrated below. The header and data CRC fields observed as part of Serial Bus isochronous packets are not recorded on the medium. Recorded isochronous data packets shall be stored on quadlet boundaries on the medium and shall contain an integral number of quadlets.

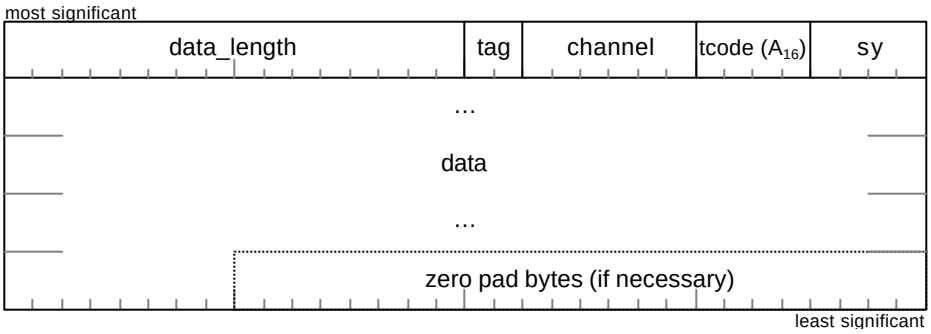

Figure E-2 – Format for recorded isochronous data

The *data_length* field shall contain the length, in bytes, of the *data* field for the packet. Zero is a permissible value for *data_length*; in this case, the packet shall consist of only the header and shall be a single quadlet in length.

The *tag* field shall specify the format of the *data* field, encoded as indicated by the following table.

Value	Data field format
0	Data format not specified by this standard
1	Common isochronous packet (CIP) format (as specified by IEC 61883-1 ~~(1998-02)~~)
2 – 3	Reserved for future standardization

The *channel* field shall identify the channel number for the packet. The *channel* field recorded on the medium may have been transformed by a mapping from the channel observed on Serial Bus. Upon playback, the *channel* field may be transformed by a similar mapping.

The *tcode* field shall be equal to A_{16}.

The *sy*, or synchronization code, field is an application-dependent field, the details of whose use are beyond the scope of this standard.

NOTE – A synchronization point may be defined as a boundary between video or audio frames, or any other point in the isochronous stream specified by the application.

The *data* field shall contain *data_length* bytes of information and shall be padded with trailing zero bytes, as necessary, to occupy an integral number of quadlets on the medium.

Dependent upon the value of *tag*, the logical unit may require additional knowledge of isochronous data formats. When *tag* is zero the data payload of the isochronous packet is unformatted and requires no transformations upon either recording or playback. When *tag* is one, the format of the data payload shall conform to the common isochronous packet (CIP) format standardized by IEC 61883-1 ~~(1998-02)~~.

E.3 Null packets

When the *tcode* field has a value of E_{16}, the data that is stored on the medium shall be ignored during playback. The format of a null packet is shown below.

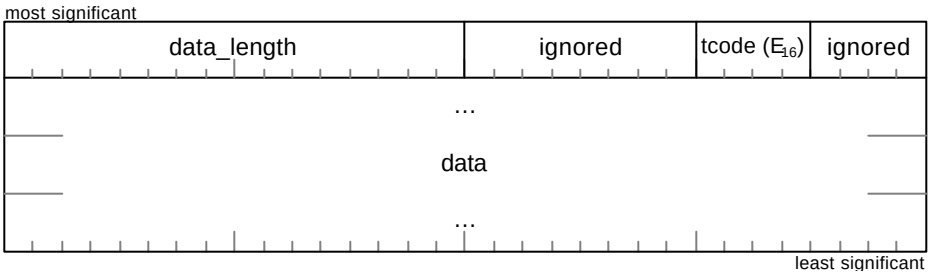

Figure E-3 – NULL packet format

The *data_length* field shall contain the length, in bytes, of the *data* field for the null packet. The number of quadlets occupied by the *data* field is (*data_length* + 3) modulus 4. Zero is a permissible value for *data_length*; in this case, the null packet shall consist of only the header and shall be a single quadlet in length.

The *tcode* field shall be equal to E_{16}.

The values of quadlets within the *data* field are unspecified for NULL packets.

NOTE – NULL packets serve no particular purpose for logical units, but they may be useful to some applications, such as nonlinear editing. Excessive quantities or sizes of NULL packets may cause some target implementations to experience underflow or other errors in the playback of isochronous data.

Annex F
(informative)

Sample configuration ROM

Configuration ROM is located at a base address of FFFF F000 0400_{16} within a node's memory space. The requirements for general format configuration ROM for targets are specified in section 7. This annex contains illustrations of typical configuration ROM for a simple targets.

F.1 Basic target

Figure F-1 below shows the bus information block, root directory and instance directory for a basic SBP-3 target. The unit directory, which implements a single logical unit is shown separately in Figure F-2.

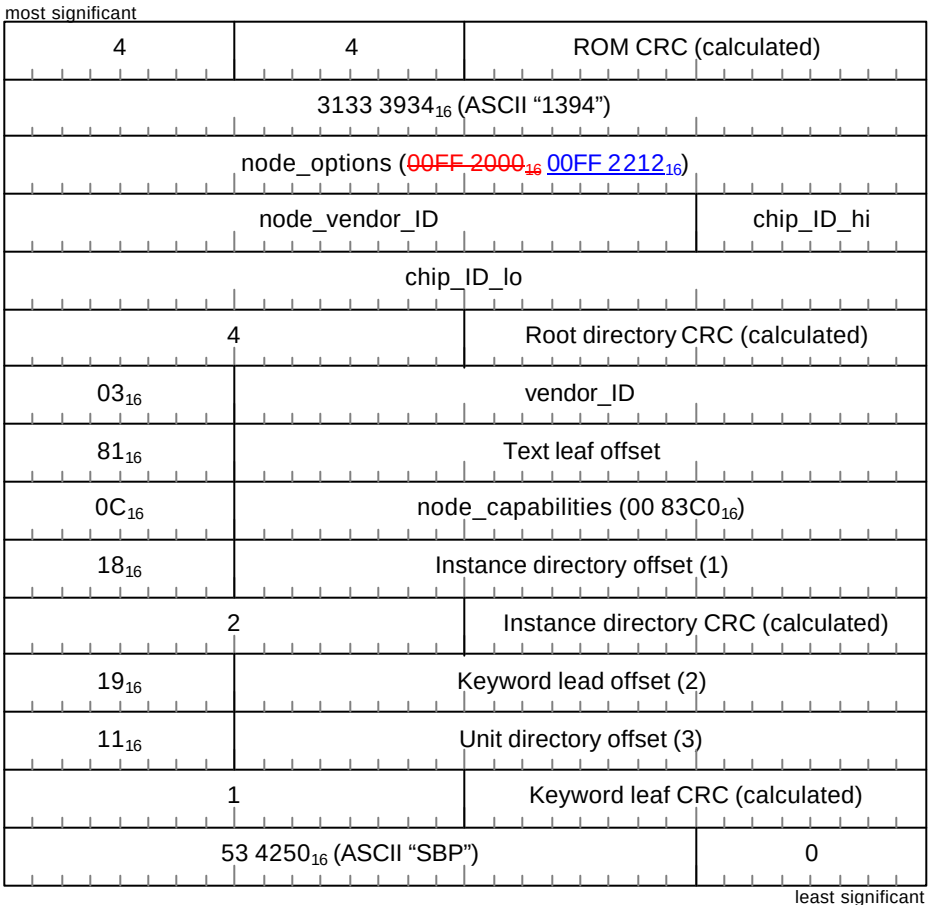

Figure F-1 – Bus information block, root and instance directories

The ROM CRC in the first quadlet is calculated on the four quadlets of the bus information block that follow.

F.1.1 Root directory

The *node_options* field represents a collection of bits and fields specified in 7.4.3. The value shown, ~~00FF 2000$_{16}$~~ 00FF 2212$_{16}$, represents basic characteristics of a device that is not isochronous capable. This value is composed of a *cyc_clk_acc* field with a value of FF$_{16}$, a *max_rec* value of two, a *max_ROM* value of two, a *generation* value of one and a *link_spd* value of two. The *max_rec* field ~~encodes~~ specifies that the target supports a maximum payload of eight bytes in block write requests ~~addressed to the target~~. The *max_ROM* field specifies that the target supports a *data_length* of up to 1024 bytes in block read requests addressed to configuration ROM. The *generation* field specifies that the target's configuration ROM never changes while its link is continuously active. The *link_spd* field specifies that the target's link supports S400 operations.

The Node_Capabilities entry in the root directory, with a *key* field of 0C$_{16}$, has a value where the *spt*, *64*, *fix*, *lst* and *drq* bits are all one. This is a minimum requirement for targets.

The Vendor_ID entry in the root directory, with a *key* field of 03$_{16}$, is immediately followed by a textual descriptor leaf entry, with a *key* field of 81$_{16}$, whose *indirect_offset* value points to a leaf that contains an ASCII string that identifies the vendor. Although this textual descriptor leaf is not shown, if it were placed immediately after the fifteen quadlets illustrated the value of *indirect_offset* would be eight. See F.2.2 for an example of a vendor identification text leaf.

The Instance_Directory entry in the root directory, with a *key* field of 18$_{16}$, has an *indirect_offset* value of one that points to the instance directory that immediately follows the root directory.

F.1.2 Instance directory

The Keyword entry in the instance directory, with a *key* field of 19$_{16}$, has an *indirect_offset* value of two that points to the keyword leaf that immediately follows the instance directory.

The Unit_Directory entry in the instance directory, with a *key* field of 11$_{16}$, has an *indirect_offset* value of three that points to the unit directory that is assumed to immediately follow the keyword leaf (see F.1.3).

The keyword leaf that immediately follows the instance directory contains a single keyword, "SBP". In an actual device, additional keywords such as "DISK" or "PRINTER" probably would be included in the keyword leaf.

F.1.3 Unit directory

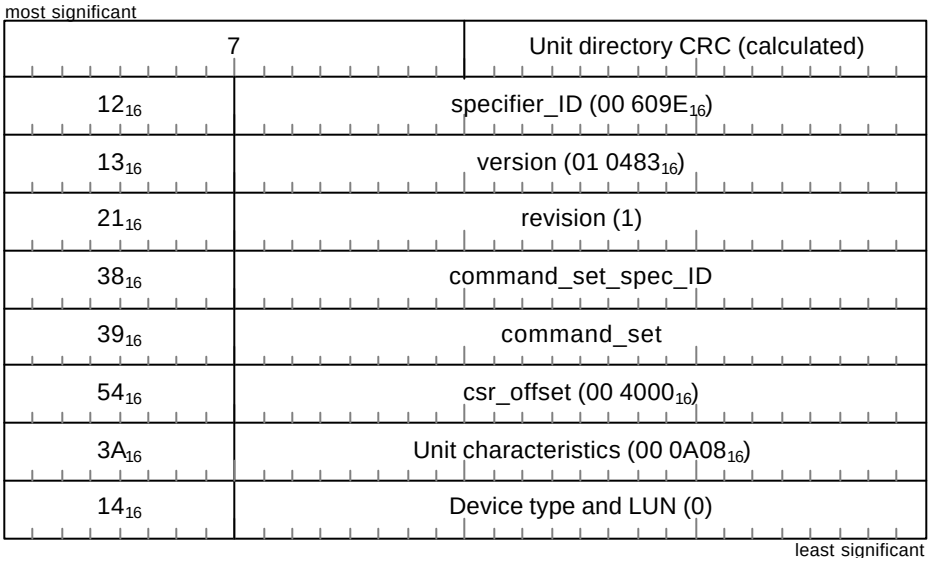

Figure F-2 – Basic unit directory

The Specifier_ID, Version and Revision entries, with *key* fields of 12_{16}, 13_{16} and 21_{16}, respectively, indicate that the target conforms to this standard.

The Command_Set_Spec_ID and Command_Set entries, with a *key* field of 38_{16} and 39_{16}, respectively, are expected to define the command set used by the target.

The Management_Agent entry in the unit directory, with a *key* field of 54_{16}, has a *csr_offset* value of 00 4000_{16} that indicates that the management agent CSR has a base address of FFFF F001 0000_{16} within the node's memory space.

The Unit_Characteristics entry in the unit directory, with a *key* field of $3A_{16}$, has an immediate value of 00 $0A08_{16}$. This indicates a target that is expected to complete task management requests (including login) within five seconds and fetches 32-byte ORBs.

The Logical_Unit_Number entry in the unit directory, with a *key* field of 14_{16}, has an immediate value of zero that indicates a device that may reorder tasks without restriction, does not support isochronous operations and has a logical unit number of zero.

F.2 SCSI command set target

The sample bus information block and root directory for basic targets are equally applicable to targets that use SCSI command sets and are not repeated below. Figure F-3 shows an example of a unit directory and textual descriptor leaves for a SCSI direct-access device.

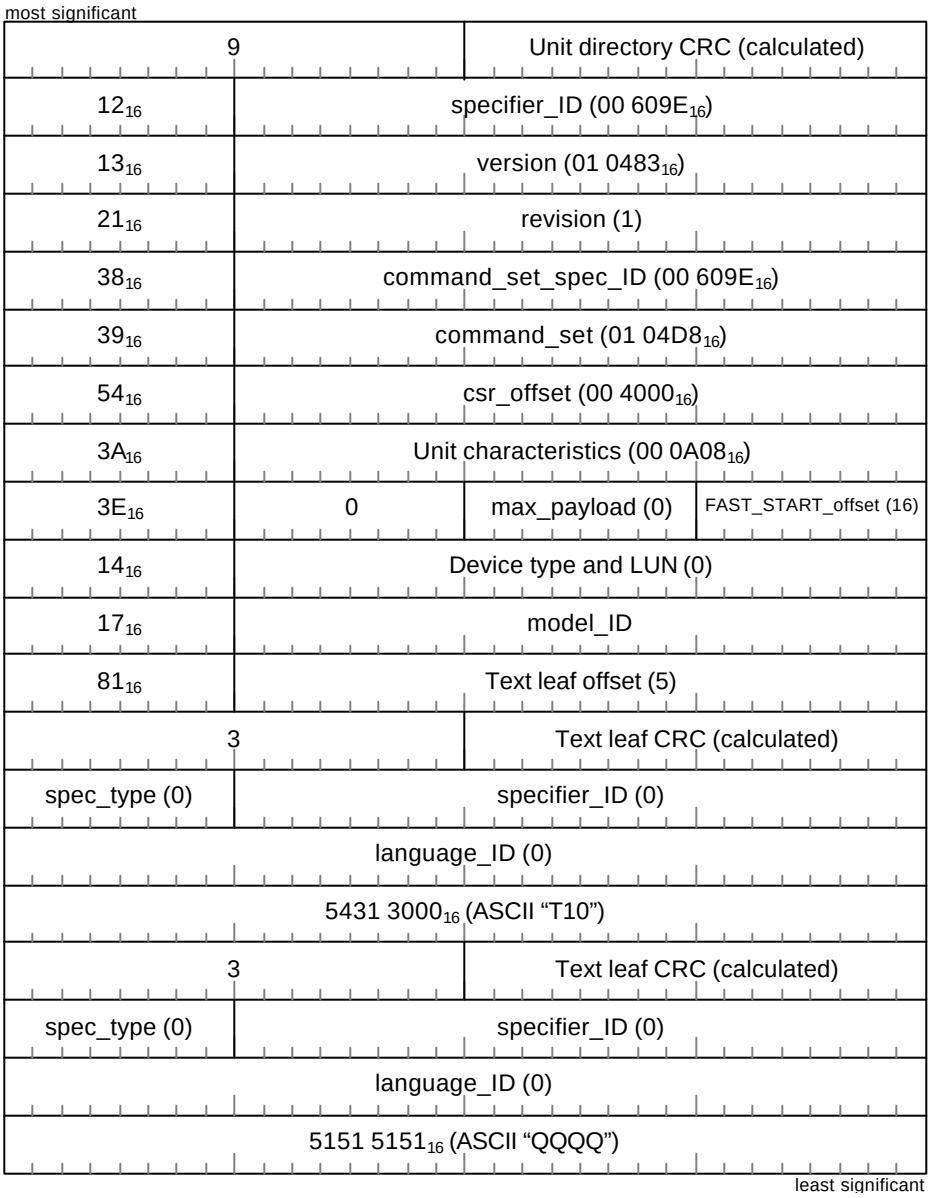

Figure F-3 – SCSI configuration ROM

F.2.1 Unit directory

The Specifier_ID, Version and Revision entries, with *key* fields of 12_{16}, 13_{16} and 21_{16}, respectively, indicate that the target conforms to this standard.

The Command_Set_Spec_ID and Command_Set_Version entries, with *key* fields of 38_{16} and 39_{16}, respectively, specify that the target's logical unit uses one of the SCSI command sets.

The Management_Agent entry in the unit directory, with a *key* field of 54_{16}, has a *csr_offset* value of 00 4000_{16} that indicates that the MANAGEMENT_AGENT register has a base address of FFFF F001 0000_{16} within the node's memory space.

The Unit_Characteristics entry in the unit directory, with a *key* field of $3A_{16}$, has an immediate value of 00 $0A08_{16}$. This indicates a target that is expected to complete task management requests (including login) within five seconds and fetches 32-byte ORBs.

The Fast_Start entry in the unit directory, with a *key* field of $3E_{16}$, describes the size and location of the FAST_START register. The *max_payload* field value of zero specifies the maximum size of a block write request addressed to the register as $2^{max\_rec + 1}$ bytes, where *max_rec* is obtained from the target's configuration ROM bus information block. The *FAST_START_offset* field value of 16 specifies the location of the register as *command_block_agent* + 40_{16}, where *command_block_agent* is obtained from a login response.

The Logical_Unit_Number entry in the unit directory, with a *key* field of 14_{16}, has an immediate value of zero; this indicates a direct-access device that may reorder tasks without restriction, does not support isochronous operations and whose logical unit number is zero.

The Model_ID entry in the unit directory, with a *key* field of 17_{16}, has an immediate value whose meaning is specified by the module vendor. Immediately following the Model_ID entry is a textual descriptor leaf entry, with a *key* field of 81_{16}, whose *indirect_offset* value of 5 points to a leaf that contains the ASCII string "QQQQ".

F.2.2 Textual descriptor leaves

Textual descriptor leaf entries, specified by IEEE Std 1212-2001, permit text strings to be associated with the immediately preceding configuration ROM entry. In this example, two textual descriptor leaves are shown to illustrate a product made by the T10 company with a model identification of QQQQ.

The first textual descriptor leaf is associated with the Vendor_ID entry in the root directory (not shown). The second leaf is associated with the Model_ID entry in the unit directory. The text strings are analogous to the vendor and product identification fields reported in INQUIRY data.

Because textual descriptor leaves are useful to device discovery and management software (in order to display meaningful messages for a user), implementers are encouraged to include textual descriptor leaves for at least the vendor ID and model ID.

Annex G
(informative)

Serial Bus transaction error recovery

Inherent in the nature of Serial Bus as a split-transaction bus are transaction errors that can leave the requester and responder with different or ambiguous information. One instance occurs when an acknowledge packet transmitted after receipt of a request or response packet is corrupted and not observed by the sender of the primary packet; the same ambiguity exists after either a local or remote split-transaction time-out.

When an acknowledge packet is missed by the sender of the primary packet, the sender does not know which of the following applies:

- The primary packet was correctly received by the destination node (and resultant side-effects in that node may have occurred); or
- The primary packet had a CRC or other error, has not been correctly received by the destination node and no state changes have occurred.

When a split time-out occurs after a request subaction addressed to a node on the local bus, the sender knows that the packet was correctly received by the destination node but does not know whether or not resultant side-effects have taken place. If a remote split time-out occurs, the sender does not know whether or not the packet was correctly received by the destination and therefore does not know whether resultant side-effects have taken place.

NOTE – IEEE 1394 contains important information about split time-out errors, the SPLIT_TIMEOUT register and different error recovery procedures for the requester and responder. Similar information may be found in draft standard IEEE P1394.1 that pertains to remote time-out for request subactions addressed to a global node ID.

If an acknowledge is missing after transmission of a response packet, IEEE 1394 prohibits retransmission of the response packet. Even in the case of a missing acknowledgement following a request packet, it ~~may~~ might not be advisable to retry (because of side effects associated with certain SBP-3 transactions).

A few of the more common error scenarios and the recommended error recover for each are described below.

G.1 MANAGEMENT_AGENT write request

When a management ORB is signaled to a target by means of an 8byte block write to the target's MANAGEMENT_AGENT register and no acknowledgement is received, the initiator does not know whether or not the ORB will be fetched by the target.

Error recovery is straightforward if the initiator waits a minimum of *mgt_ORB_timeout* for the return of a status block before any attempt is made to retry the management ORB. By waiting the specified time the initiator avoids the possibility of multiple status blocks for the same ORB address.

G.2 ORB_POINTER or FAST_START write request

A consequence of a write to either the ORB_POINTER or FAST_START register when the fetch agent is in the RESET or SUSPENDED state is that, if successful, the fetch agent ~~transitions to~~ enters the ACTIVE state. If no acknowledgement is received by the initiator after a write to either the ORB_POINTER or

FAST_START register when the fetch agent is in either of these states, the initiator should not retry the write. The recommended method for error recovery is a write to the AGENT_RESET register.

NOTE – An exception to this recommendation exists if no acknowledgment is received after a write to the FAST_START register whose *previous_ORB* field is non-null. Because the target fetch agent compares the *previous_ORB* field to the ORB_POINTER register, the write may be retried.

G.3 Data buffer, ORB or page table read request

If the target transmits a block read request and receives no acknowledgement, the read request may be retried immediately but care should be taken to not reuse the same transaction label as the failed request. For nodes on the local bus, the target should wait a minimum of a SPLIT_TIMEOUT period before the transaction label is reused in any subsequent request subaction otherwise it should wait a minimum of the remote time-out period currently in effect for the remote node.

G.4 Status FIFO write request

When the target detects a missing acknowledgement after a write to an initiator's status FIFO, it should take no error recovery actions. Any target resources allocated to the ORB should be released by the target. The initiator is expected to discover the error by means of a higher-level mechanism, such as a command time-out and to initiate appropriate error recovery. The nature of the error recovery undertaken by the initiator likely depends whether or not the target processes ORBs and returns their status in order, but in any event is beyond the scope of this description.

Annex H
(informative)

SCSI Architecture Model conformance

This annex provides information useful to systems implementers: it relates the facilities provided by SBP-3 to the terminology used by the SCSI Architecture Model.

H.1 Object definitions

The SCSI Architecture Model defines objects within the SCSI domain. The equivalency of SBP-3 objects is enumerated below in those cases where the correlation ~~may~~ might not be clear and in those cases where SBP-3 restricts the scope of an object.

Initiator port identifier: The *login_ID* returned by a SCSI target port in response to a successful login is the initiator port identifier for command block requests.

Logical unit number: SBP-3 restricts the scope of the logical unit number to 2^{16}. The *lun* field in management ORBs is one doublet. Its format may be either non-hierarchical or hierarchical (see SAM-2), as indicated by the HISUP bit in INQUIRY data returned by LUN 0.

NOTE – Neither the *login_ID* nor *lun* fields are present in SCSI command block ORBs, since the value of both is implicit in the CSR addresses of the logical unit fetch agent to which the ORBs are signaled.

Tag: The Serial Bus address of an ORB is the tag by which the task is identified. This requires that SCSI initiator port memory allocated to a request not be released or reused while the task is active within a task set. The scope of an SBP-3 tag is that of a Serial Bus address, 2^{64}, and equal to the scope defined by SAM-2.

Target port identifier: SCSI target ports are identified by means of a unit unique ID, or EUI-64, found in the SCSI target port's unit directory in configuration ROM. When a unit unique ID leaf is not present in configuration ROM, the value of the unit unique ID is construed to be equal to the node unique ID. The scope of a target port identifier, 2^{64}, is identical to the scope of a target port identifier specified by SAM-2.

Task set: An SBP-3 task set consists of the linked list of command block ORBs that are managed by a single fetch agent. There is no provision in SBP-3 for untagged tasks; an SBP-3 task set always consists of zero or more tagged tasks.

SBP-3 delimits the extent of a task set in a way that is compatible with SAM-2 but that differs from the definition given in SAM-2 of "…a group of tasks *within* a ~~target~~ logical unit…" (emphasis added). By way of contrast with SAM-2, an SBP-3 task enters a task set when it is linked into an active request list. The extent of an SBP-3 task set includes all the uncompleted ORBs linked into a request list in SCSI initiator port memory, not solely the requests already fetched by the SCSI target port.

Untagged task: SBP-3 does not define untagged tasks.

H.2 Status

SCSI status is reported in the *status* field, which is part of the status block defined in B.2.

H.3 Command delivery services

SAM-2 requires that four protocol services be defined to support the Execute Command remote procedure call. The SBP-3 facilities used to provide these services are described below.

H.3.1 Send SCSI Command

The formal arguments of the Send SCSI Command service are:

I_T_L_*x* nexus,
CDB,
[Task Attribute],
[Data-in buffer size],
[Data-out buffer],
[Data-out buffer size],
[Autosense request],
[Command reference number]

The I_T_L_*x* nexus argument is composed of the address of the ORB and the logical unit for which it is intended. The logical unit is implicit in the fetch agent to which the ORB is signaled. The request is signaled to a logical unit fetch agent by the methods specified by section 9.

The CDB argument is encapsulated within an ORB and fetched by the ~~target~~ logical unit from SCSI initiator port memory.

The task attribute argument, either SIMPLE or ORDERED, is implicit in the ~~target~~ logical unit implementation. The Logical_Unit_Number entry in either a unit or logical unit directory indicates which task attribute is implemented. When the *ordered* bit in this entry is zero, all tasks have an implicit attribute of SIMPLE. Otherwise, if *ordered* is set to one, the task attribute is ORDERED for all tasks.

The data-in buffer size, data-out buffer and data-out buffer size arguments, if present, are specified by the *data_descriptor* and *data_size* fields in the ORB.

The SBP-3 status block provides for the return of autosense data; the SCIS initiator port may be expected to reformat the information as SCSI sense data before it is presented to the application client.

SBP-3 does not directly support command reference numbers (CRNs), but an application client and logical unit may use the order inherent in queues of SCSI command block ORBs to reproduce CRN functionality.

H.3.2 SCSI Command Received

The formal arguments of the SCSI Command Received service are:

I_T_L_*x* nexus,
[Task Attribute],
CDB,
[Autosense request]
[Command reference number]

The I_T_L_*x* nexus argument is composed of the address of the ORB and the logical unit for which it is intended. The logical unit is implicit in the fetch agent to which the ORB is signaled. A Serial Bus write

indication for the AGENT_RESET register signals the logical unit fetch agent that there may be a new SCSI command. The details of this indication are specified in section 9.

The task attribute argument is implicit in the ~~target~~ logical unit implementation and may be determined by an examination of the *ordered* bit in the Logical_Unit_Number entry in configuration ROM.

The CDB argument is encapsulated within the ORB and fetched by the ~~target~~ logical unit from SCSI initiator port memory.

The *status_FIFO* address, provided by the SCSI initiator port as part of the login procedure, is the address for the return of autosense data.

SBP-3 does not directly support command reference numbers (CRNs), but an application client and logical unit may use the order inherent in queues of SCSI command block ORBs to reproduce CRN functionality.

H.3.3 Send Command Complete

The formal arguments of the Send Command Complete service are:

I_T_L_*x* nexus,
[Sense data],
Status,
Service response

The I_T_L_*x* nexus argument is derived from the address at which the status information is stored. The ORB specified the address for the status block, either implicitly by means of a fixed offset from the address of the ORB or explicitly by means of the *status_FIFO* field. In either case, the SCSI initiator port ensures that the address at which status is stored is sufficient to uniquely correlate the status with the I_T_L_*x* nexus.

SCSI sense data may be returned in the status block upon completion of a SCSI command.

The service response argument is encoded within the status block by *resp* and *sbp_status* as summarized below.

Table H-1 – SAM-2 Service responses

Service response	*resp*	*sbp_status*	Description
Task complete	0	0	The task has ended with a completion status indicated by *status*
Linked command complete	—	—	Not supported by SBP-3
Linked command complete (with flag)	—	—	Not supported by SBP-3
Function complete	0	0	Used by task management functions
Service delivery or SCSI target device failure	1	Various (as defined by SBP-3)	The command has completed because of a Serial Bus service failure or a SCSI target device malfunction
Function rejected	0	9	Used by task management functions

H.3.4 Command Complete Received

The formal arguments of the Command Complete Received service are:

I_T_L_*x* nexus,
[Data-in buffer],
[Sense data],
Status,
Service response

The I_T_L_*x* nexus argument is derived from the address to which the status information was stored. The ORB specified the address for the status block, either implicitly by means of a fixed offset from the address of the ORB or explicitly by means of the *status_FIFO* field. In either case, the SCSI initiator port ensures that the address at which status is stored is sufficient to uniquely correlate the status with the I_T_L_*x* nexus.

SCSI sense data may be returned in the status block upon completion of a SCSI command.

The service response argument is encoded within the status block by *resp*, as described by Table H-1.

H.4 Data transfer services

SAM-2 requires that four protocol services be defined to support data transfer necessary for the Execute Command remote procedure call. The SBP-3 facilities used to provide these services are described below.

H.4.1 Send Data-in

The formal arguments of the Send Data-in service are:

I_T_L_*x* nexus,
Device server buffer,
Application client buffer offset,
Request byte count

The I_T_L_*x* nexus argument is the Serial Bus address of the ORB for the active task. It is expected that ~~target~~ logical unit implementations reference a copy of the ORB maintained in the device's local memory, although nothing precludes a fetch of the information from the SCSI initiator port memory occupied by the ORB.

The device service buffer argument is vendor-dependent. The data available in the device server buffer is formed into Serial Bus write transactions as described below.

The application client buffer offset is a value maintained by the device server to correlate medium locations with locations in the application client buffer. The base of the application client buffer is specified by the *data_descriptor* field supplied by the SCSI initiator port.

The request byte count is determined by the device server.

The ~~target~~ logical unit uses one or more Serial Bus quadlet or block write requests to store the requested data into the application client buffer. For the sake of efficiency, it is expected that the ~~target~~ logical unit uses the largest block write requests permitted by the *max_payload* field in the ORB and transmit these requests at the speed mandated by the *spd* field in the ORB.

H.4.2 Data-in Delivered

The formal arguments of the Data-in Delivered service are:

I_T_L_*x* nexus

Upon completion of each of the quadlet or block write requests initiated as a result of Send Data-in, the ~~target~~ logical unit receives Serial Bus write response confirmations. It is the ~~target's~~ logical unit's responsibility to correlate the Serial Bus addresses (for which write responses are received) with the I_T_L_*x* nexus in order to provide the Data-in Delivered confirmation.

H.4.3 Receive Data-out

The formal arguments of the Receive Data-out service are:

I_T_L_*x* nexus,
Application client buffer offset,
Request byte count,
Device server buffer

The I_T_L_*x* nexus argument is the Serial Bus address of the ORB for the active task. It is expected that ~~target~~ logical unit implementations reference a copy of the ORB maintained in the device's local memory, although nothing precludes a fetch of the information from the SCSI initiator port memory occupied by the ORB.

The application client buffer offset is a value maintained by the device server to correlate medium locations with locations in the application client buffer. The base of the application client buffer is specified by the *data_descriptor* field supplied by the SCSI initiator port.

The request byte count is determined by the device server.

The device service buffer argument is vendor-dependent. The data obtained from Serial Bus read response subactions is moved to the device server buffer as described below.

The ~~target~~ logical unit uses one or more Serial Bus quadlet or block read requests to fetch the requested data from the application client buffer. For the sake of efficiency, it is expected that the ~~target~~ logical unit use the largest block read requests permitted by the *max_payload* field in the ORB and transmit these requests at the speed mandated by the *spd* field in the ORB.

H.4.4 Data-out Received

The formal arguments of the Data-out Received service are:

I_T_L_*x* nexus

Upon completion of each of the quadlet or block read requests initiated as a result of Receive Data-out, the ~~target~~ logical unit receives Serial Bus read response confirmations and their accompanying data. It is the ~~target's~~ logical unit's responsibility transfer the data to the device server buffer and to correlate the Serial Bus addresses (for which read response subactions are received) with the I_T_L_*x* nexus in order to provide the Data-out Received confirmation.

H.5 Contingent allegiance

SBP-3 SCSI target ports implement ~~SCSI-2~~ SAM-2 contingent allegiance and do not support CDBs whose *naca* bit in the control byte is one. The contingent allegiance condition exists within a task set when a logical unit stores a status block for a command where *status* is set to CHECK CONDITION ~~or COMMAND TERMINATED~~. Since SBP-3 SCSI target ports implement autosense *via* the return of the status block, the contingent allegiance condition is automatically cleared.

At the time a contingent allegiance condition is created, the logical unit:

a) immediately halts the operations of the fetch agent for the faulted SCSI initiator port;

b) aborts the task set in the same fashion as if an ABORT TASK SET task management function had been signaled to the SCSI target port;

c) clears the contingent allegiance condition.

Because the ~~faulted initiator's~~ fetch agent utilized by the faulted SCSI initiator port has been halted, it is necessary for the SCSI initiator port to reset and reinitialize the fetch agent before any commands may be signaled to the ~~target~~ logical unit.

H.6 Asynchronous event reporting

SBP-3 does not support asynchronous event reporting as defined by SAM-2.

H.7 Autosense

SBP-3 supports autosense through the return of a status block to the address specified by the login parameter *status_FIFO*. The status block is always stored in the event of an exception condition, *e.g.*, CHECK CONDITION.

H.8 Hard reset

A write to the RESET_START register (see 6.2) or a task management function of TARGET RESET causes the SCSI target port to execute a hard reset, as defined by SAM-2.

H.9 Task set type

SBP-3 ~~targets~~ logical units implement one task set per SCSI initiator port ~~per logical unit~~. For ~~targets~~ logical units that implement the SCSI control mode page (page code A_{16}), the task set type (TST) field is unchangeable and reports a value of one.

H.10 Task management functions

SBP-3 SCSI target ports implement the basic task management model, as specified by SAM-2. SBP-3 support for task management functions is described in the table below.

Function	Support	Comments
ABORT TASK	Required	
ABORT TASK SET	Required	May also be performed directly through the AGENT_RESET register
CLEAR ACA	Not supported	SBP-3 supports ~~only SCSI-2~~ SAM-2 contingent allegiance
CLEAR TASK SET	Not supported	ABORT TASK SET is equivalent
TARGET RESET	Required	May also be performed directly through the RESET_START register
LOGICAL UNIT RESET	Optional[11]	Functions as TARGET RESET but scope is limited to a single logical unit
~~TERMINATE TASK~~	~~Not supported~~	
WAKEUP	Not supported	

SAM-2 additionally requires protocol services to support the task management functions, enumerated below. In all of the definitions for the task management function protocol services, the following apply:

- The nexus is as specified by SAM-2 and modified by the SBP-3 object definitions;
- The function identifier is one of the task management functions both defined by SAM-2 and supported by SBP-3; and
- The service response is one of Function complete, Function rejected or Service delivery or SCSI target port failure. These service responses are encoded by the *resp* and *sbp_status* fields in the status block stored by the SCSI target port upon completion of a request. See Table H-1 for the numeric values that encode the service responses.

H.10.1 Send Task Management Request

The formal arguments of the Send Task Management Request service are:

Nexus,
Function identifier

Subsequent to the creation of a task management ORB in SCSI initiator port memory, the SCSI initiator port signals the request to the SCSI target port management agent by the methods described in SBP-3.

H.10.2 Task Management Request Received

The formal arguments of the Task Management Request Received service are:

Nexus,
Function identifier

[11] Although support for logical unit reset is optional for SBP-3 targets, it is required of targets that implement one or more logical units compliant with SAM-2.

When a Serial Bus write indication is received for the SCSI target port's MANAGEMENT_AGENT register, the SCSI target port may fetch the request from SCSI initiator port memory.

H.10.3 Task Management Function Executed

The formal arguments of the Task Management Function Executed service are:

Nexus,
Service response

The SCSI target port signals the completion of the task management function by storing an 8-byte status block at the address specified by *status_FIFO* in the ORB.

H.10.4 Received Task Management Function Executed

The formal arguments of the Received Task Management Function Executed service are:

Nexus,
Service response

When the SCSI initiator port receives a Serial Bus write indication for data addressed to the *status_FIFO* address, it may examine the status block to determine the service response from *resp*.

Annex I
(informative)

Common isochronous packet (CIP) format

Data packets recorded in the isochronous data interchange format (described in Annex E) may also conform to a CIP format that divides the data payload into two parts: the CIP header and the application-dependent data that follows. Figure I-1 illustrates the organization of the common isochronous packet format.

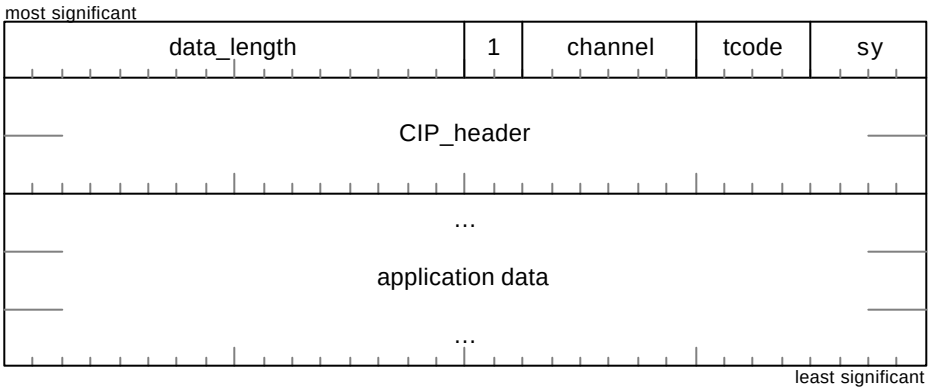

Figure I-1 – Common isochronous packet (CIP) format

The CIP header is a variable number of quadlets (although only two are shown in the preceding figure). The most significant bit of each quadlet of the CIP header is called the *eoh* bit. For an *n* quadlet CIP header, *eoh* is zero for quadlets zero through *n* - 2, inclusive, and *eoh* is one for quadlet *n* - 1. The next most significant bit of each quadlet is called the *form* bit. Together, the *eoh* and *form* bits specify the format of the CIP header quadlet. CIP header formats are defined for *form* values of zero; *form* values of one are reserved for future standardization.

The only CIP header format currently defined is a two-quadlet header shown below.

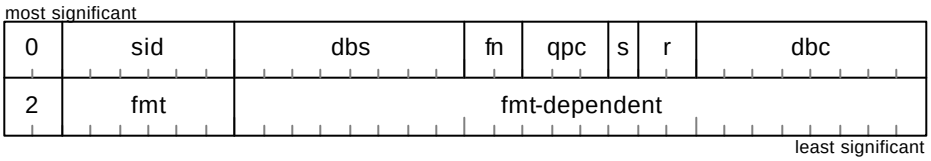

Figure I-2 – Two-quadlet CIP header format

The *sid*, or source ID, field identifies the node whose plug control registers control the source (talker) for the isochronous data.

The *dbs*, or data block size, field contains the size of each of the application-dependent data blocks that follow the CIP header. A *dbs* value of zero encodes a size of 256 quadlets; for all other values of *dbs* the number of quadlets is the value of *dbs* itself. Individual data blocks are entirely contained within a single Serial Bus isochronous packet (which may encapsulate more than one data block).

The *fn*, or fraction number, field contains the number of data blocks that form a higher level, application-dependent object—the isochronous source packet. The number of data blocks that form an isochronous source packet is specified as 2^{fn}; when there is a one-to-one correspondence between isochronous source packets and data blocks *fn* is zero.

The *qpc*, or quadlet padding count, field contains the number of pad quadlets appended to an isochronous source packet before it is divided into data blocks. The quadlet padding count is less than the data block size and has a value that results in equal sizes for the data blocks. If *fn* is zero, *qpc* are also zero. When *qpc* is nonzero the last data block includes the pad quadlets, which are recorded when the target is a listener and transmitted when the target is a talker.

The *sph*, or source packet header, bit (abbreviated as *s* in Figure I-2) is one if the isochronous source packet begins with a header quadlet of the format shown below; otherwise, it is zero.

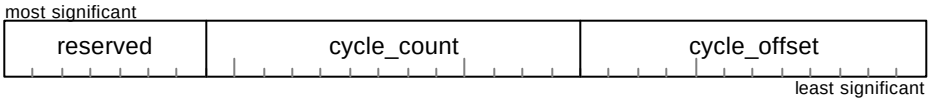

Figure I-3 – Source packet header format

The source packet header contains a time stamp encoded in the same fashion as the least significant 25 bits of the CYCLE TIME register.

The *dbc*, or data block continuity counter, field contains the sequence number of the isochronous source packet and the sequence number of the of the data block within the isochronous source packet. The least significant *fn* bits of *dbc* hold the sequence number of the data block while the most significant 8 - *fn* bits hold the sequence number of the isochronous source packet itself. The data block continuity counter labels the first data block that follows the CIP header; the continuity counter of additional data blocks after the first increases monotonically from the value of *dbc*.

NOTE – The data block that immediately follows the CIP header is not necessarily the first data block of the isochronous source packet. The location of the starting data block of an isochronous source packet can be determined from the values of *dbc* and *fn*. Relative to the first data block after the CIP header (counting from zero), the ordinal of this data block is given by (2^{fn} - (*dbc* modulus 2^{fn})) modulus 2^{fn}. If a source packet header is present (as indicated by the *sph* bit), it is the first quadlet of this data block.

The *fmt* field specifies the formats of both the *fmt*-dependent field within the same quadlet of the CIP header and the application-dependent data contained within the common isochronous packets. An *fmt* value of $3F_{16}$ indicates that no application-dependent data follows the CIP header and that the *dbs*, *fn*, *qpc* fields, the *sph* bit and the *dbc* field in the CIP header are all ignored. Other values of *fmt* encode the application-dependent format of the isochronous data, *e.g.*, DVCR or MPEG. The details of most application-dependent formats are not relevant to targets and are beyond the scope of this standard. However, the value of *fmt* specifies the format of the *fmt*-dependent field within same quadlet of the CIP header; this field is meaningful to targets when it contains a time stamp, since the time stamp is transformed during playback. The table below summarizes the recommended meanings of *fmt* for targets.

NOTE – IEC 61883-1 ~~(1998-02)~~ does not require that the most significant bit of *fmt* govern the presence or absence of time stamps within the CIP header. This standard recommends that future extensions to IEC 61883-1 ~~(1998-02)~~ conform to the table below.

Value	Description
$0 – 1F_{16}$	Application data is present; the *fmt*-dependent field contains a time stamp defined by *syt* below
$20_{16} – 3E_{16}$	Application data is present; the contents of the *fmt*-dependent field are unspecified
$3F_{16}$	No application data is present

When *fmt* is in the range zero to $1F_{16}$, inclusive, the second quadlet of the CIP header has the format illustrated below.

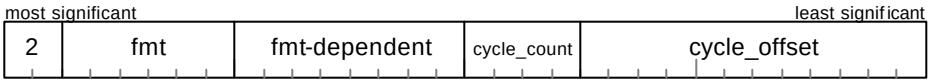

Figure I-4 – Synchronization time (*syt*) format

The two fields, *cycle_count* and *cycle_offset*, are collectively referred to as the *syt*, or synchronization time, field. When *syt* has a value of $FFFF_{16}$, no synchronization time information is present. Otherwise, the *syt* field represents a time stamp encoded in the same fashion as the least significant 16 bits of the CYCLE_TIME register. Just as in the case of the CYCLE_TIME register, the value of *cycle_offset* is constrained to be in the range zero to 3071 inclusive; Values of *syt* for which *cycle_offset* is greater than 3071 are invalid.

Annex J
(informative)

Bibliography

[B1] 1394 Trade Association, AV/C Digital Interface Command Set General Specification 4.1, December 11, 2001

[B2] ANSI NCITS 325-1998, Serial Bus Protocol 2 (SBP-2)

[B3] ANSI NCITS 330-2000, Reduced Block Commands

[B4] ANSI NCITS 333-2000, SCSI Multimedia Commands 2 (MMC-2)

[B5] IEC 61883-1 (1998-02), Consumer audio/video equipment—Digital interface—Part 1: General

[B6] IEEE Std 1212-2001, Standard for a Control and Status Registers (CSR) Architecture for microcomputer buses

[B7] IEEE P1394.1, Draft Standard for High Performance Serial Bus Bridges

NOTE – If, at the time the ANSI Editor prepares this standard for publication, draft standard IEEE P1394.1 has been approved by the IEEE-SA Standards Board, the Editor is requested to change this citation to reference the approved standard and to change all occurrences of the phrase “draft standard IEEE P1394.1” to “IEEE Std 1394.1-200X”. Otherwise the citation should reference, by version and date, the most recent draft of IEEE P1394.1.

[B8] IEEE P1394.3, Draft Standard for a High Performance Serial Bus Peer-to-Peer Data Transport Protocol (PPDT)

NOTE – If, at the time the ANSI Editor prepares this standard for publication, draft standard IEEE P1394.3 has been approved by the IEEE-SA Standards Board, the Editor is requested to change this citation to reference the approved standard and to change all occurrences of the phrase “draft standard IEEE P1394.3” to “IEEE Std 1394.3-2003”. Otherwise the citation should reference, by version and date, the most recent draft of IEEE P1394.3.

[B9] IEEE Std 1394-1995, Standard for a High Performance Serial Bus

[B10] IEEE Std 1394a-2000, Standard for a High Performance Serial Bus—Amendment 1

[B11] IEEE Std 1394b-2002, Standard for a High Performance Serial Bus—Amendment 2

[B12] INCITS 351-2001, SCSI Primary Commands 2 (SPC-2)

[B13] INCITS 360-2002, SCSI Multimedia Commands 3 (MMC-3)

[B14] INCITS 366-2003, SCSI Architecture Model 2 (SAM-2)

[B15] ISO/IEC 9899:1990, Programming Languages—C

~~[B15] T10 Project 1157D, SCSI Architecture Model 2 (SAM-2)~~

[B16] T10 Project 1416D, SCSI Primary Commands 3 (SPC-3)