Working Draft

T10
Project 1467D

Revision 1e
October 1, 2001

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

This is a draft proposed American National Standard under development by T10, a Technical Committee of the National Committee for Information Technology Standardization (NCITS). As such, this is not a completed standard and has not been approved. The Technical Committee may modify this document as a result of comments received during public review and its approval as a standard.

Permission is granted to members of NCITS, its technical committees and their associated task groups to reproduce this document for the purposes of NCITS standardization activities without further permission, provided this notice is included. All other rights are reserved. Any commercial or for-profit replication or republication is prohibited.

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 NCITS xxx-200x

Printed October 1, 2001

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@IX.NetCom.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:	NCITS 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
NCITS Secretariat:	NCITS Secretariat 1250 I Street NW, Suite 200 Washington, DC 20005 USA (202) 737-8888 (202) 638-4922 FAX

ANSI®
NCITS 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.

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

Copyright © 2001 by American National Standards Institute
All rights reserved.

Printed in the United States of America

Contents

Page

Foreword ... vii

Revision history ... ix

1 Scope and purpose ... 1
  1.1 Scope ... 1
  1.2 Purpose ... 1

2 Normative references ... 3
  2.1 Approved references ... 3
  2.2 References under development ... 3

3 Definitions and notation ... 5
  3.1 Definitions ... 5
    3.1.1 Conformance ... 5
    3.1.2 Glossary ... 5
    3.1.3 Abbreviations ... 8
  3.2 Notation ... 9
    3.2.1 Numeric values ... 9
    3.2.2 Bit, byte and quadlet ordering ... 9
    3.2.3 Register specifications ... 10
    3.2.4 State machines ... 12

4 Model (informative) ... 13
  4.1 Unit architecture ... 13
  4.2 Logical units ... 13
  4.3 Requests and responses ... 13
  4.4 Data buffers ... 14
  4.5 Target agents ... 16
  4.6 Ordered and unordered execution ... 17
  4.7 Bridge-awareness ... 17
  4.8 Streams ... 19
    4.8.1 Stream task set ... 21
    4.8.2 Stream controller ... 21
    4.8.3 Error reporting ... 21

5 Data structures ... 23
  5.1 Operation request blocks (ORBs) ... 24
    5.1.1 Dummy ORB ... 25
    5.1.2 Command block ORBs ... 25
    5.1.3 Stream control ORB ... 30
    5.1.4 Management ORBs ... 35
  5.2 Page tables ... 46
    5.2.1 Unrestricted page tables ... 47
    5.2.2 Normalized page tables ... 47
    5.2.3 Node selectors ... 48
  5.3 Status block ... 49
    5.3.1 Request status ... 50
    5.3.2 Unsolicited device status ... 52
    5.3.3 Unsolicited isochronous error report ... 53

6 Control and status registers ... 55
  6.1 Core registers ... 55

6.2 Serial Bus-dependent registers....55
6.3 MANAGEMENT_AGENT register....56
6.4 Command block and stream control agent registers....57
6.4.1 AGENT_STATE register....57
6.4.2 AGENT_RESET register....58
6.4.3 ORB_POINTER register....59
6.4.4 DOORBELL register....60
6.4.5 UNSOLICITED_STATUS_ENABLE register....60
6.4.6 FAST_START register....61

7 Configuration ROM....63
7.1 Power reset initialization....64
7.2 Bus information block....64
7.3 Root directory....66
7.3.1 Vendor_ID entry....66
7.3.2 Node_Capabilities entry....66
7.3.3 Unit_Directory entry....67
7.4 Unit directory....67
7.5 Logical unit directory....67
7.6 Directory entries....67
7.6.1 Specifier_ID entry....68
7.6.2 Version entry....68
7.6.3 Revision entry....69
7.6.4 Command_Set_Spec_ID entry....69
7.6.5 Command_Set entry....69
7.6.6 Command_Set_Revision entry....70
7.6.7 Firmware_Revision entry....70
7.6.8 Management_Agent entry....70
7.6.9 Unit_Characteristics entry....71
7.6.10 Reconnect_Timeout entry....71
7.6.11 Fast_Start entry....72
7.6.12 Logical_Unit_Directory entry....72
7.6.13 Logical_Unit_Number entry....73
7.6.14 Unit_Unique_ID entry....73
7.7 Unit unique ID leaf....74

8 Access....75
8.1 Access protocols....75
8.2 Access requests....75
8.2.1 Login....75
8.2.2 Create stream....76
8.3 Reconnection....77
8.4 Logout....78

9 Command execution....79
9.1 Requests and request lists....79
9.1.1 Fetch agent initialization (informative)....79
9.1.2 Dynamic appends to request lists (informative)....80
9.1.3 Fetch agent use by the BIOS (informative)....80
9.1.4 Use of the FAST_START register (informative)....81
9.1.5 Fetch agent state machine....82
9.2 Data transfer (normal command block ORBs)....86
9.3 Completion status....86
9.4 Unsolicited status....87

10 Task management 89
10.1 Task sets 89
10.2 Basic task management model....89
10.3 Error conditions 90
10.4 Task management requests....90
10.4.1 Abort task....90
10.4.2 Abort task set....92
10.4.3 Logical unit reset 92
10.4.4 Target reset 93
10.5 Task management event matrix....93

11 Isochronous data interchange format 97
11.1 Cycle marks....97
11.2 Isochronous data packets 97
11.3 Null packets....98

12 Isochronous operations 101
12.1 Stream command block requests 101
12.2 Stream control (optional)....102
12.2.1 Channel masks....102
12.2.2 Stream control 103
12.2.3 Isochronous data transformation....103
12.3 Error logs....104

Tables

Table 1 – Data transfer speeds....27
Table 2 – Management request functions....36
Table F-1 – SAM-2 Service responses 127

Figures

Figure 1 – Bit ordering within a byte 9
Figure 2 – Byte ordering within a quadlet 9
Figure 3 – Quadlet ordering within an octlet 10
Figure 4 – CSR specification example 10
Figure 5 – State machine example 12
Figure 6 – Linked list of ORBs....14
Figure 7 – Directly addressed data buffer....15
Figure 8 – Indirectly addressed data buffer (*via* page table)....15
Figure 9 – Components of an isochronous stream (direct-access target)....19
Figure 10 – Stream engine block diagram....20
Figure 11 – Address pointer....23
Figure 12 – ORB pointer....23
Figure 13 – ORB family tree 24
Figure 14 – ORB format 24
Figure 15 – Dummy ORB 25
Figure 16 – Normal command block ORB (single data descriptor) 26
Figure 17 – Normal command block ORB (dual data descriptor)....28
Figure 18 – Stream command block ORB....29
Figure 19 – Stream control ORB 31
Figure 20 – Channel mask 33
Figure 21 – Channel configuration map entry....34
Figure 22 – Management ORB....36
Figure 23 – Login ORB 37
Figure 24 – Login response....38

Figure 25 – Query logins ORB....39
Figure 26 – Query logins response format....39
Figure 27 – Create stream ORB....40
Figure 28 – Create stream response....42
Figure 29 – Reconnect ORB....43
Figure 30 – Node handle ORB....44
Figure 31 – Node handle response....45
Figure 32 – Logout ORB....45
Figure 33 – Task management ORB....46
Figure 34 – Page table element (unrestricted page table)....47
Figure 35 – Page table element (when *page_size* equals four)....48
Figure 36 – Node selector....49
Figure 37 – Status block format....49
Figure 38 – TRANSPORT FAILURE format for *sbp_status*....51
Figure 39 – Unsolicited status format for isochronous errors....53
Figure 40 – MANAGEMENT_AGENT format....56
Figure 41 – AGENT_STATE format....58
Figure 42 – AGENT_RESET format....58
Figure 43 – ORB_POINTER format....59
Figure 44 – DOORBELL format....60
Figure 45 – UNSOLICITED_STATUS_ENABLE format....60
Figure 46 – FAST_START format....61
Figure 47 – Configuration ROM hierarchy....63
Figure 48 – Bus information block format....64
Figure 49 – Bus information block *capabilities* field....65
Figure 50 – Vendor_ID entry format....66
Figure 51 – Node_Capabilities entry format....66
Figure 52 – Unit_Directory entry format....67
Figure 53 – Specifier_ID entry format....68
Figure 54 – Version entry format....69
Figure 55 – Revision entry format....69
Figure 56 – Command_Set_Spec_ID entry format....69
Figure 57 – Command_Set entry format....70
Figure 58 – Command_Set_Revision entry format....70
Figure 59 – Firmware_Revision entry format....70
Figure 60 – Management_Agent entry format....71
Figure 61 – Unit_Characteristics entry format....71
Figure 62 – Reconnect_Timeout entry format....72
Figure 63 – Fast_Start entry format....72
Figure 64 – Logical_Unit_Directory entry format....72
Figure 65 – Logical_Unit_Number entry format....73
Figure 66 – Unit_Unique_ID entry format....73
Figure 67 – Unit unique ID leaf format....74
Figure 68 – Fetch agent initialization with a dummy ORB....80
Figure 69 – Fetch agent state machine....83
Figure 70 – CYCLE MARK format....97
Figure 71 – Format for recorded isochronous data....98
Figure 72 – NULL packet format....99
Figure B-1 – SCSI command block ORB....109
Figure B-2 – SCSI control byte....109
Figure B-3 – Status block format for SCSI sense data....110
Figure C-1 – Set password ORB....114
Figure D-1 – Bus information block, root and instance directories....117
Figure D-2 – Basic unit directory....118
Figure D-3 – SCSI configuration ROM....120
Figure G-1 – Common isochronous packet (CIP) format....133

Figure G-2 – Two-quadlet CIP header format..133
Figure G-3 – Source packet header format...134
Figure G-4 – Synchronization time (*syt*) format..135
Figure H-1 – AV/C command frame ORB..138
Figure H-2 – Status block format for AV/C response frame...............................138
Figure H-3 – AV/C unit directory 139

Annexes

Annex A (normative) Minimum Serial Bus node capabilities..............................107

Annex B (normative) SCSI command and status encapsulation.......................109

Annex C (normative) Security extensions...113

Annex D (informative) Sample configuration ROM...117

Annex E (informative) Serial Bus transaction error recovery............................123

Annex F (informative) SCSI Architecture Model conformance 125

Annex G (informative) Common isochronous packet (CIP) format 133

Annex H (informative) AV/C Encapsulation 137

Annex I (informative) Bibliography..141

Foreword (This foreword is not part of American National Standard NCITS 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 nine annexes in this standard. Annexes A, B and C are normative and part of this standard. Annexes D through I, 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 NCITS 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 National Committee for Information Technology Standardization (NCITS). Committee approval of this standard does not necessarily imply that all committee members voted for approval. At the time it approved this standard, NCITS had the following members:

James D. Converse, Chair
Donald C. Loughry, Vice-chair
Joanne M. Flanagan, Secretary

Organization Represented	*Name of Representative*
American Nuclear Society	Geraldine C. Main
AMP, Inc.	Edward Kelly
Apple Computer	Karen Higginbottom
Association of the Institute for Certification of Professionals	Kenneth Zemrowski
AT&T/NCR	Thomas W. Kern
Boeing Company	Catherine Howells
Bull HN Information Systems, Inc.	William George
Compaq Computer Corporation	James Barnes
Digital Equipment Corporation	Delbert Shoemaker
Eastman Kodak	James D. Converse
GUIDE International	Frank Kirshenbaum
Hewlett-Packard	Donald C. Loughry
Hitachi America, Ltd.	John Neumann
Hughes Aircraft Company	Harold L. Zebrack
IBM Corporation	Joel Urman
National Communication Systems	Dennis Bodson
National Institute of Standards and Technology	Robert E. Roundtree
Northern Telecom, Inc.	Mel Woinsky
Neville & Associates	Carlton Neville
Recognition Technology Users Association	Herbert P. Schantz
Share, Inc.	Gary Ainsworth
Sony Corporation	Michael Deese
Storage Technology Corporation	Joseph S. Zajaczkowski
Sun Microsystems	Scott Jameson
3M Company	Eddie T. Morioka
Unisys Corporation	John L. Hill
US Department of Defense	William C. Rinehuls
US Department of Energy	Alton Cox
US General Services Administration	Douglas Arai

Wintergreen Information Services .... Joun Wheeler
Xerox Corporation.. Dwight McBain

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
T. Bradshaw
C. Brill
Z. Daggett
R. Elliott
M. Evans
L. Fong
B. Galloway
E. Gardner
R. Griswold
R. Haagens
E. Haske
N. Hastad
G. Houlder
P. Johansson
S. Jones
T. Kasebayashi
T. Kulesza
E. Lew
B. Mable
W. McFerrin
P. McLean
K. Moe
C. Monia
D. Moore
J. Neer
T. Nelson
R. Nixon
E. Oetting
D. Peterson
D. Piper
B. Raudebaugh
R. Roberts
G. Robinson
C. Simpson
R. Snively
R. Stockford
C. Tashbook
M. Taylor
W. Terrell
D. Wagner
N. Wanamaker
D. Woelz

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 D.

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 directories. 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.

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 as amended by IEEE Std 1394a-2000. The transport protocol, Serial Bus Protocol 3 or SBP-3, requires implementations to conform to the requirements of the aforementioned standard as well as to draft standard IEEE P1212, Control and Status Register (CSR) Architecture for microcomputer buses, 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), 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 Std 1394-1995 as amended by IEEE Std 1394a-2000, 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;
– 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 draft standard IEEE P1394b; 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

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.

2.1 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 1394-1995, Standard for a High Performance Serial Bus

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

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.

2.2 References under development

At the time of publication, the following referenced standards were still under development.

IEEE P1212, Draft Standard for a Control and Status Registers (CSR) Architecture for microcomputer buses

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

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

T10 Project 1236D, SCSI Primary Commands 2 (SPC-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 ORB 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 node space: The 256 terabytes of Serial Bus address space that may be 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 draft standard IEEE P1212 or IEEE 1394 for more information on address spaces.

3.1.2.5 register space: A two kilobyte portion of node space with a base address of FFFF F000 0000_{16}. Core registers defined by draft standard IEEE P1212 are located within register space as are Serial Bus-dependent registers defined by IEEE 1394.

3.1.2.6 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.

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

3.1.2.8 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 packet, 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.

3.1.2.9 isochronous period: An operating mode of Serial Bus that occurs, on average, every 125 microseconds. 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.

3.1.2.10 kilobyte: A quantity of data equal to 2^{10} bytes.

3.1.2.11 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.

3.1.2.12 login: The process by which an initiator obtains access to a set of target fetch agents. The target fetch agents and their control and status registers provide a mechanism for an initiator to signal ORBs to the target.

3.1.2.13 login ID: A value assigned by the target during a login or create stream process. The login ID establishes a relationship between an initiator and a task set. In the case of a create stream request, the login ID (or, synonymously, the stream ID) establishes a relationship between an initiator and an isochronous stream. Either kind of 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.

3.1.2.14 node: An addressable device attached to Serial Bus.

3.1.2.15 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.

3.1.2.16 normal command block: A command block whose ORB specifies a data buffer address for the transfer of data to or from the device medium. Asynchronous Serial Bus transfers are used to effect the data transfer.

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

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

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

3.1.2.20 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.

3.1.2.21 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.

3.1.2.22 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 subaction(s), quadlet or block, to place the data in system memory and the corresponding response subaction(s) that complete the write(s).

3.1.2.27 stream: An object that represents a target's functions and resources necessary to transfer isochronous data from one or more Serial Bus channels to the device medium (the target is a listener) or to transfer isochronous data from the device medium to one or more Serial Bus channels (the target is a talker).

3.1.2.28 stream command block: A command block whose ORB does not specify a data buffer address. Data transfer to or from the device medium may require coordination with stream control ORBs that mediate data transfer from or to Serial Bus by means of isochronous transactions.

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

3.1.2.30 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.

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

3.1.2.32 target: A unit architecture 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.

3.1.2.33 task: A task is an organizing concept that represents the work to be done by a target to carry out a command encapsulated by an ORB. In order to perform a task, a target 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 target's task set, proceeds through a period of

execution by the target 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 both target resources and initiator resources.

3.1.2.34 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.

3.1.2.35 terabyte: A quantity of data equal to 2^{40} bytes.

3.1.2.36 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.

3.1.2.37 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.

3.1.2.38 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.

3.1.2.39 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.

3.1.2.40 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
LUN	Logical unit number
MPEG	Motion picture experts group
ORB	Operation request block
SAM-2	SCSI Architecture Model 2
SBP-3	Serial Bus Protocol 3 (this standard itself)
SPC-2	SCSI Primary Commands 2

3.2 Notation

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

3.2.1 Numeric values

Decimal, 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, $2A_{16}$ and $0010\ 1010_2$ all 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.

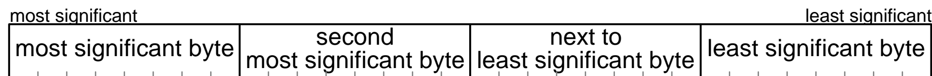

Figure 2 – Byte ordering within a quadlet

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.

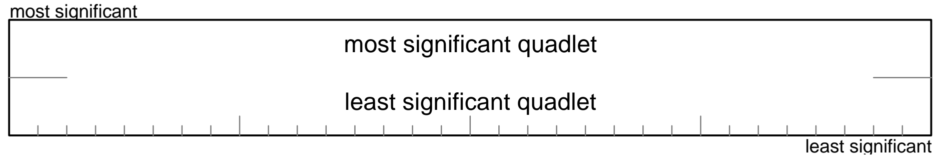

Figure 3 – Quadlet ordering within an octlet

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.

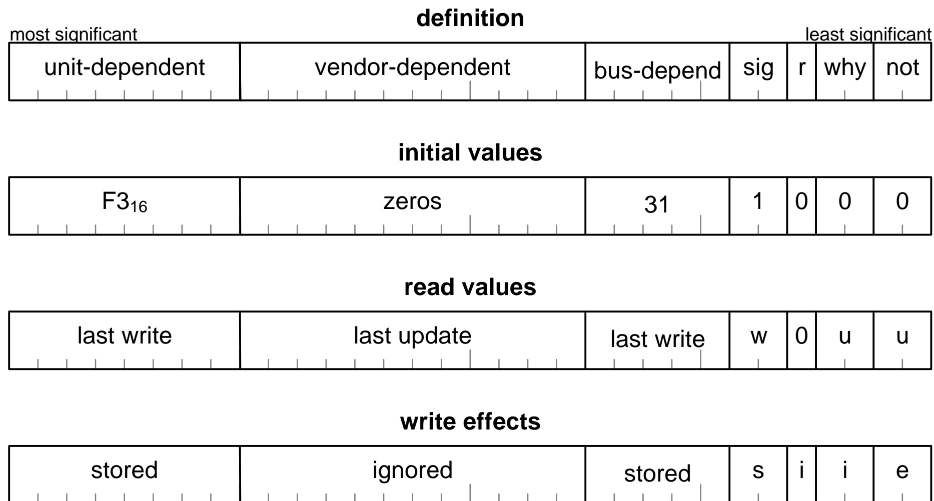

Figure 4 – CSR specification example

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	bus-depend	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 definitions)
unit-dependent	unit-depend	The meaning of the field shall be defined by the organization responsible for the unit architecture
Vendor-dependent	vendor-depend or v	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 not be immediately visible by a read of the same register. The effect may be visible in another register or may 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.

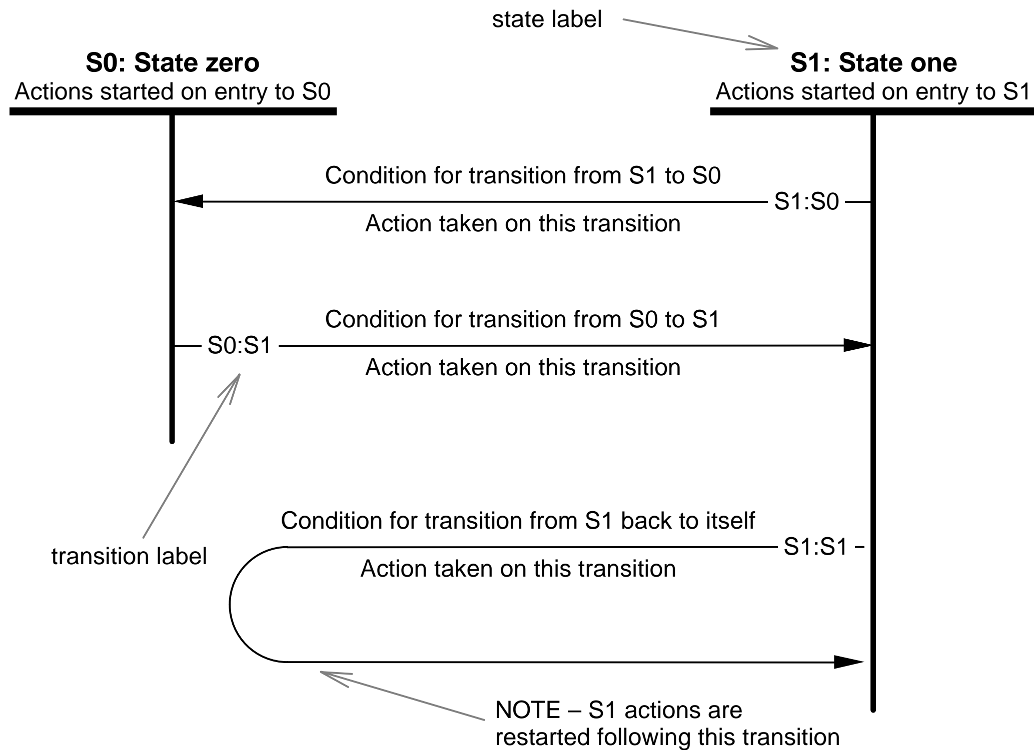

Figure 5 – State machine example

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)

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, Standard for a 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.

4.1 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 unit directory for each in configuration ROM 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.

4.2 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 server(s) responsible to execute control or data transfer commands, zero or more stream controllers, one or more task sets that hold commands available for execution by the device server(s) or stream controller(s) 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, which may be 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 unit(s) 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 target(s). 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.

4.3 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 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 (normal and stream command block requests).

– to control the flow of isochronous data (stream control 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 the other requests, normal command block, stream command block and stream control, 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.

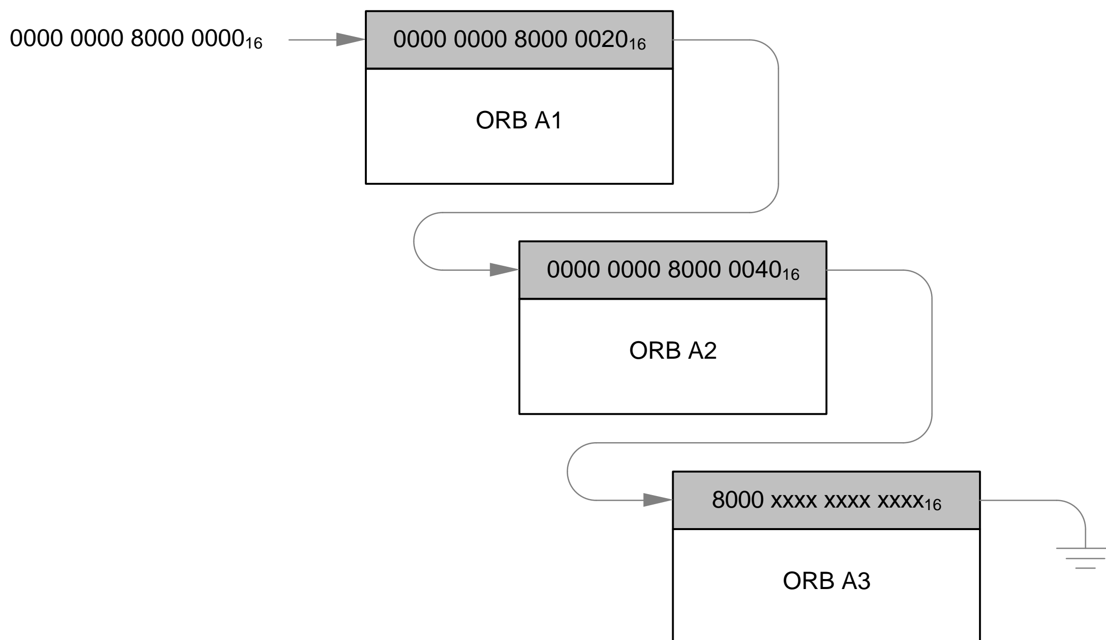

Figure 6 – Linked list of ORBs

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

When the request is completed, either in success or failure, the target stores a status block at an address specified by the initiator.

4.4 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 and the buffer starts at an address of 23 6174_{16}, the relationship between the ORB and the data buffer might appear as in Figure 7.

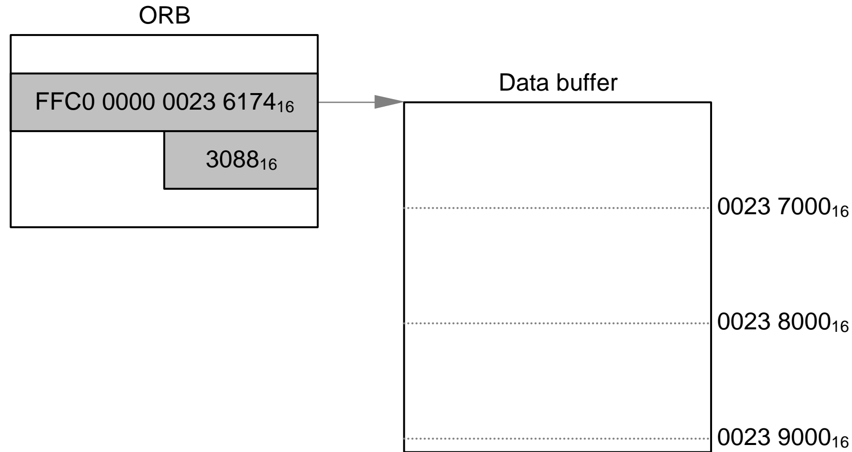

Figure 7 – Directly addressed data buffer

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.

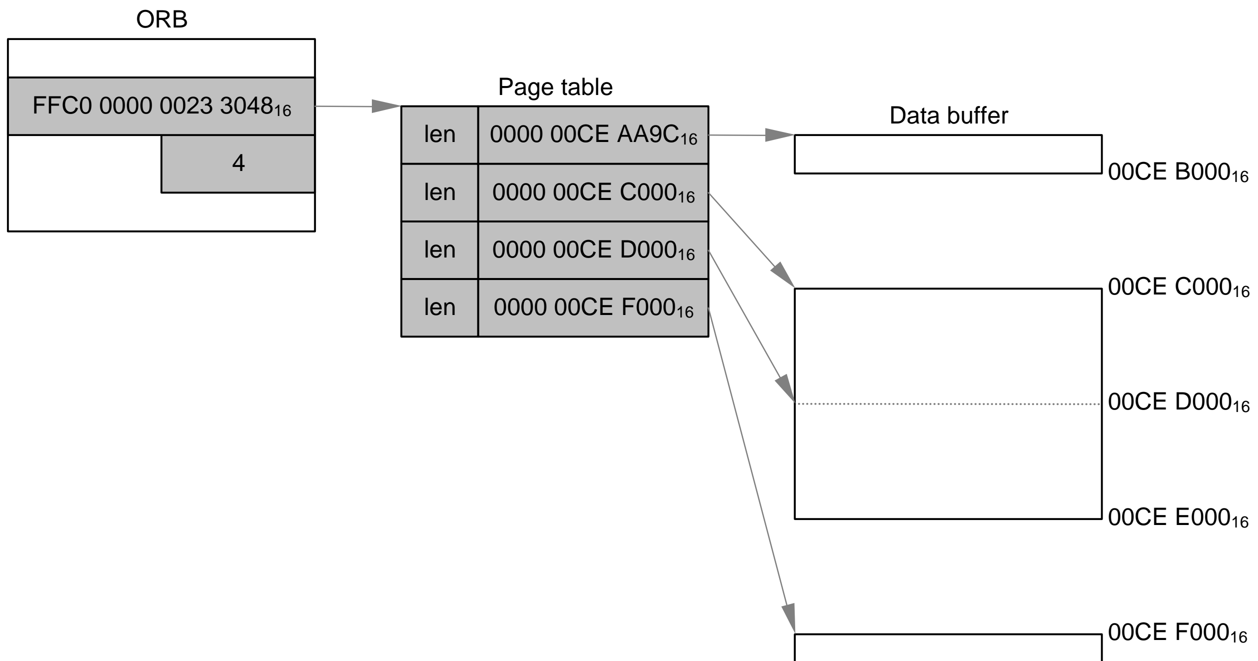

Figure 8 – Indirectly addressed data buffer (*via* page table)

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 (and the middle segments also end on page boundaries) while 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 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.

4.5 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, 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 three types of target agents:

– management;

– command block (for either normal or stream commands); and

– stream control.

Management agents accept a variety of requests: login, create stream, task management and logout. Before making other requests, an initiator first completes a login *via* the management agent. Once this is done, the management agent will accept create stream request and task management requests. The latter are directed to either a normal (asynchronous) task set or to a task set associated with an isochronous stream. Ultimately, management agents accept logout requests; these indicate the initiator‘s intent to release target resources previously acquired by a login or create stream request. Management agents service a single request at a time and do not support linked lists.

A successful login or create stream request returns the address of a normal or stream command block agent, respectively. Both types of command block agents service requests which are organized in linked lists. Individual linked list(s) are managed by a separate command block agent(s).

A create stream request may return the address of a stream control agent. This agent meters the flow of isochronous data to or from Serial Bus. Unless the target provides other facilities to meter this flow (*e.g.*, plug control registers as specified by IEC 61883-1 (1998-02)), each stream requires both a stream command block agent and a stream control agent to coordinate operations. The time-critical nature of some isochronous operations requires that stream control agents support linked lists of requests, just as command block agents do.

4.6 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.

4.7 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 particular 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);
- 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 by the value of the *snarf* field in the packet header of block write requests;
- Comprehension of new primary packet header fields, such as *proxy_ID* and *ext_rcode*, and new response codes;
- Implementation of the NET_GENERATION register;

- 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 direct 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 will 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 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 correlates with a different EUI-64 than is 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. The initiator may provide a global node ID as a hint to the target, but the target is responsible to discover the global node ID that corresponds to the EUI-64 supplied by the initiator. Once the target has validated a relationship between the EUI-64 and a global node ID, the target returns a node handle to the initiator. Thereafter the node handle may be used in any address pointer to address 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. A target that encounters a remote addressing error when a global node ID is used does not immediately terminate the associated task. Instead, device discovery methods (such as those described by IEEE P1394.1) are used to rediscover the global node ID that matches the EUI-64 cached for the node handle. Only if the desired node is no longer connected to the net does the target terminate 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 ill-equipped to detect such an event; even if the target attempts to notify the initiator, it is moot whether the initiator can successfully reinitiate work at the target 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.

[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.

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.

4.8 Streams

Streams are objects that are based upon the isochronous capabilities of Serial Bus. A stream consists of all of the target 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 isochronous data 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 channels flow 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 allocates all necessary resources before activating a target isochronous stream.

A stream of isochronous data appears on Serial Bus as packet(s) during each 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 stream command block ORBs that request data transfer to or from the medium and, optionally, stream control ORBs that control its flow on Serial Bus. The stream control ORB is not required if the target has other facilities to control the flow of isochronous data from or to Serial Bus.

The relationship between the different stream components during playback (for a target with direct-access capabilities) is given by Figure 9.

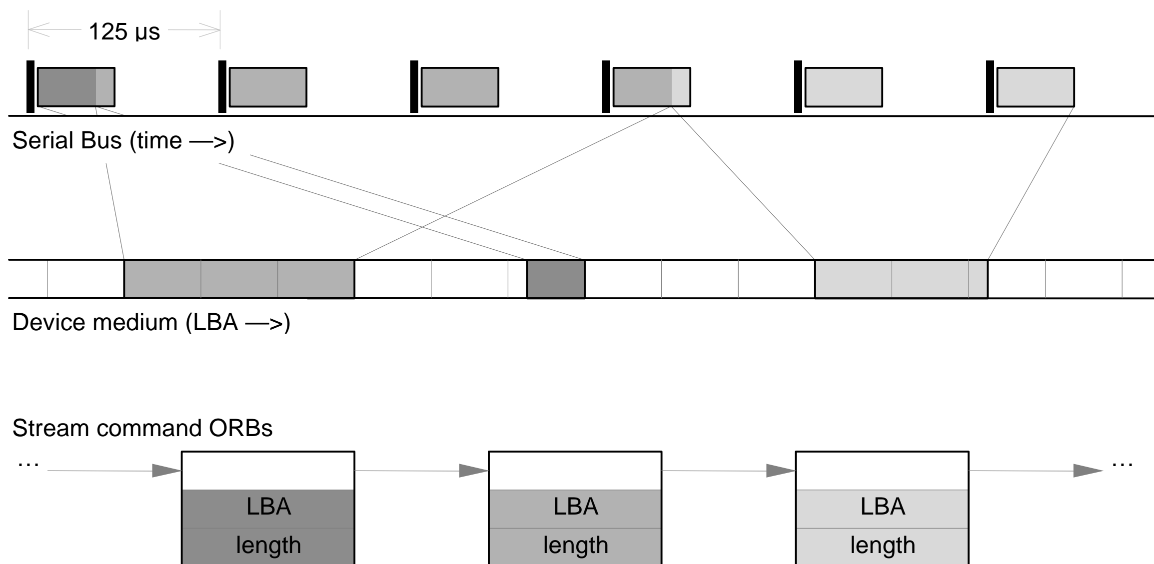

Figure 9 – Components of an isochronous stream (direct-access target)

The figure shows that the order of presentation of bytes in a stream is determined by the order of stream command 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 the data transfers described by normal command blocks in two important respects. First, streams do not require any address context for the transfer of data to or from system memory: stream data is identified by a channel number and the time-ordered location of the data within the stream. Second, is the stream's flow of isochronous data may be controlled and synchronized to time or other time-dependent events.

Because of these differences from normal operations, two functional components are required within the target to fully control a stream: a stream task set and a stream controller. The functions of a stream controller may be provided by a stream control agent or by methods outside the scope of this standard. Figure 10 illustrates the way in which these target components cooperate to form a functional unit, the stream engine.

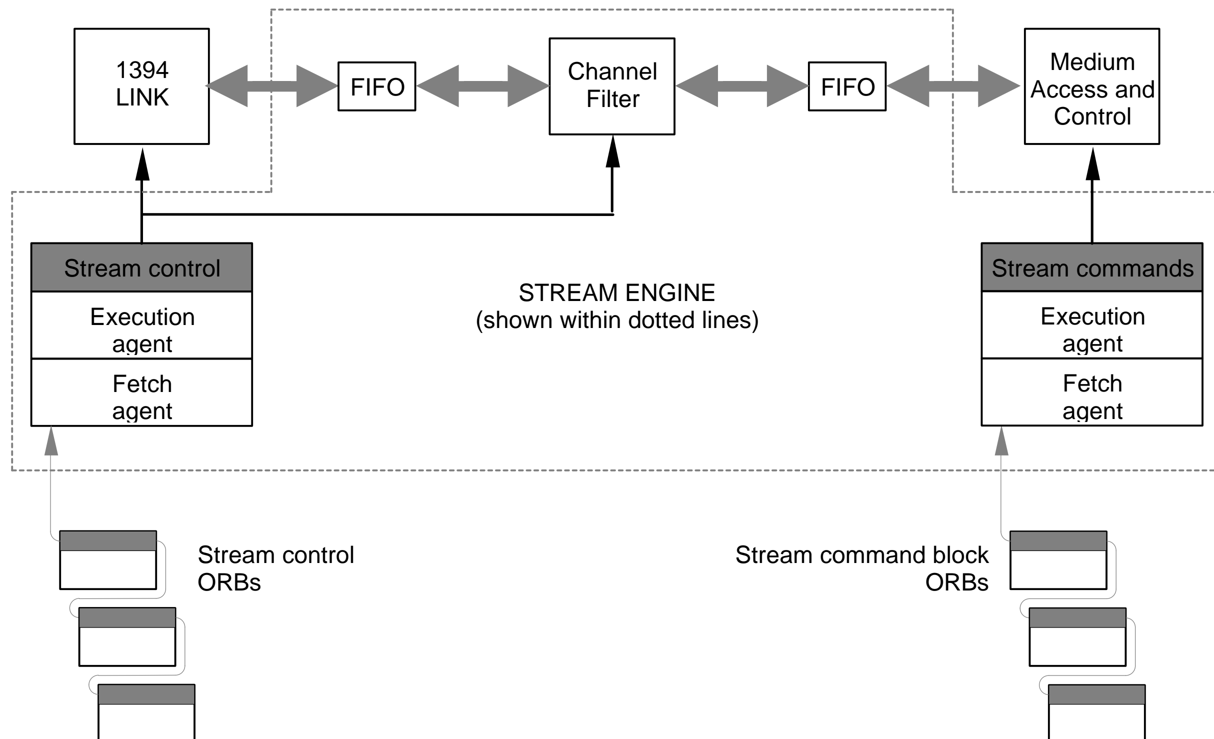

Figure 10 – Stream engine block diagram

The stream engine components that are shown within the dotted lines represent target resources necessary for each instance of an isochronous stream. The two execution agents each have separate queues of ORBs that direct their operation. The stream command block ORBs specify the length and location of isochronous data as it is transferred to or from the device medium while the stream control ORBs synchronize and meter the isochronous data on Serial Bus as the target listens or talks. These two components do not communicate directly with each other—they coordinate their operations by using a common isochronous data format for the information in the FIFOs that separate them. Although the details of the FIFOs (or buffers) are implementation-dependent, this model assumes that the byte ordering of the isochronous data is preserved.

NOTE – When a target does not implement a stream control agent, the left-hand side of the figure (which shows stream control ORBs and their fetch agent) is not present. The stream control functions and methods used by such a target are beyond the scope of this standard.

4.8.1 Stream task set

A stream task set consist of a set of commands which request data transfer to or from a device's medium. It differs from a normal (asynchronous) task set in two important respects:

- commands do not reference data buffers in system memory; and
- tasks are executed sequentially.

Data buffers are not needed (and stream command block ORBs provide no fields to describe them) because data transferred to or from the device medium is associated with one or more Serial Bus isochronous channels. When isochronous data is read from the device medium it is made available, in order, to the stream controller described below. When isochronous data is written to the device medium it is obtained, in order, from the stream controller. In neither case is a data buffer in system memory necessary.

Isochronous data is essentially time-ordered. As a consequence, the isochronous data transferred to or from the device medium needs to be presented in correct order. Therefore no reordering of stream commands is done within the task set associated with an isochronous stream and the failure of any one task results in all subsequent tasks being aborted.

Not only are tasks in a stream task set executed in order but also their completion status is reported in the same order.

4.8.2 Stream controller

By means that are implementation dependent, an ordered data pipe is assumed to exist between a target's stream task set and the associated stream controller. The function of the stream controller is to mediate the flow of isochronous data between this data pipe and Serial Bus.

The format of data transported through the data pipe is similar to that of Serial Bus isochronous packets. The data is identified by time stamps (cycle times) and channel numbers and the payload is described in terms of its length, in bytes.

The stream controller:

- filters isochronous data according to channel numbers;
- transforms time stamps and channel numbers in the isochronous data; and
- synchronizes the flow of the isochronous data with external, time-dependent events.

Any of these operations may take place whether the stream controller is a listener or a talker. Stream control ORBs that specify these operations are independent of the stream command block ORBs in the stream task set. The stream task set and the stream controller communicate with each other through the data pipe.

Stream controller actions may be queued by the target. This permits time-critical operations to be specified in advance and avoids latency problems that could arise if the stream controller could accept no more than one request at a time. Within the queue of requests to the stream controller, each is executed in order as the preceding stream control ORB completes.

4.8.3 Error reporting

In addition to the data transfer errors that may be encountered by any of the stream command block ORBs, errors may occur within the isochronous stream itself as it is transferred to or from Serial Bus. These errors might include, but are not limited to:

- a missing isochronous packet or cycle start indication;
- an isochronous packet with a header CRC error;
- when the target is a talker, an underflow in the availability of data from the stream command block ORBs that causes no data to be transmitted for one or more channels during an isochronous period; or
- when the target is a listener, an overflow in which isochronous data from Serial Bus must be discarded because of an internal buffer overflow or a lack of stream command block ORB(s) to transfer the data to the medium.

In any of these cases, the initiator may wish to ignore all errors, receive reports of all errors but continue the isochronous stream or receive a report of the first error and halt isochronous operations. The initiator selects the error reporting option, which determines whether or not the target may store an unsolicited status block at the time an error occurs.

5 Data structures

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

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

These data structures may be 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; 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 within ORBs to conform to the format specified below.

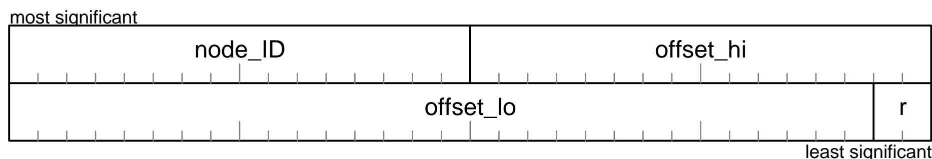

Figure 11 – Address pointer

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.

The size of a data structure or buffer addressed by a pointer that conforms to Figure 11 is either explicitly specified by an associated length field or implicitly known from context. Whichever the case, the target shall not initiate any Serial Bus request subactions (read, write or lock) that reference system memory outside of the range determined by the address pointer and length.

ORBs shall be allocated at the initiator's node. Some types of ORBs contain a forward address pointer and may 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.

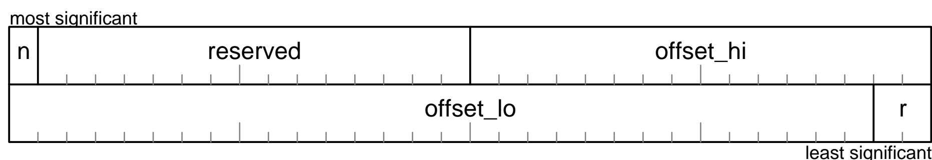

Figure 12 – ORB pointer

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.

5.1 Operation request blocks (ORBs)

All initiator requests for target actions are expressed within ORBs fetched by the target *via* Serial Bus read transaction(s). ORB formats vary according to use and may be viewed in hierarchical relationship to each other, as illustrated below.

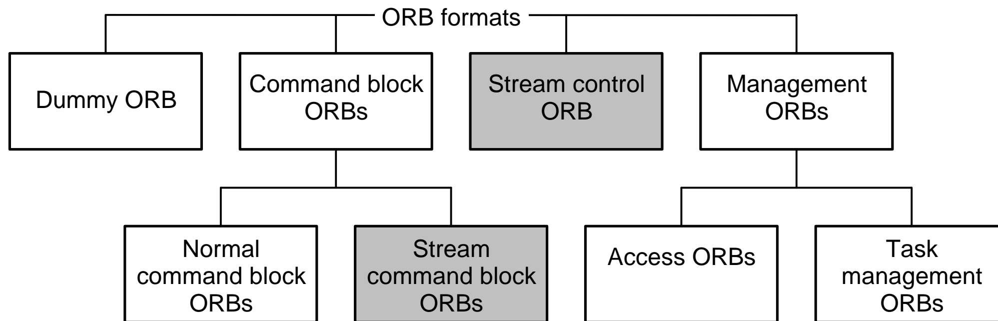

Figure 13 – ORB family tree

In the preceding figure, the ORBs that pertain to isochronous operations are shown shaded in gray. 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.

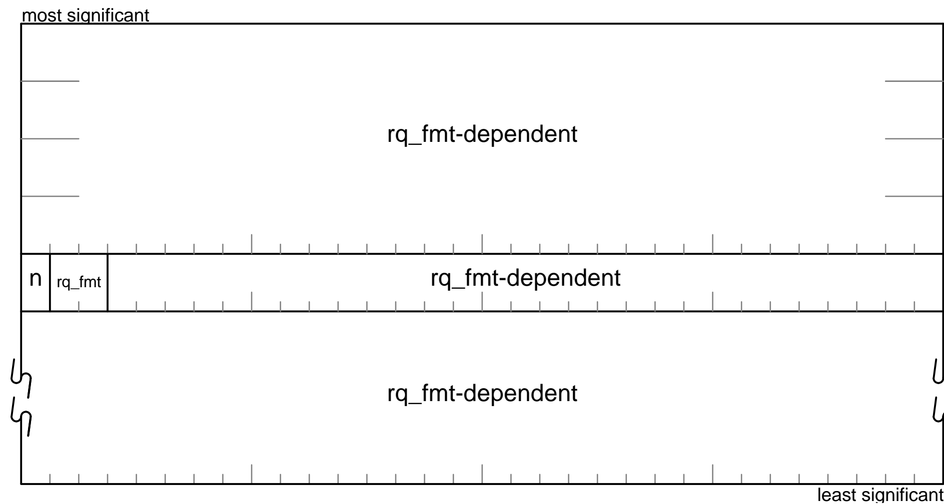

Figure 14 – ORB format

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 stream 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.

5.1.1 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 target fetch agent (see 9.1.1). The initiator shall allocate at least the maximum ORB fetch size implemented by the target. The format of a dummy ORB is illustrated below.

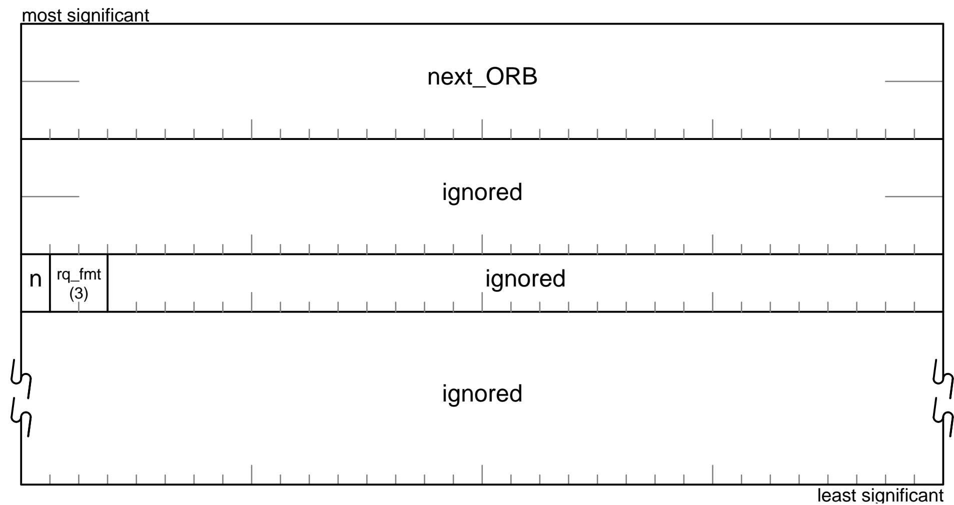

Figure 15 – Dummy ORB

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 12.

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 10.4.1 for details of ORB processing by the target and for permissible completion status values.

5.1.2 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 7.6.9).

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 normal 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.

There are two kinds of command block ORB, one for normal (asynchronous) operations and one for stream (isochronous) operations; both have the same size. Normal command block ORBs permit the specification of a data buffer in system memory, from which or to which data is transferred by the target.

Stream command block ORBs do not specify a data buffer in system memory. Isochronous operations involve a stream of data without system memory address context. Data bytes within a stream have relative ordering with respect to each other; there is no explicit system memory address that is the source or the sink for the stream. An isochronous stream is coupled to a stream controller that can start, stop or pause the isochronous stream on Serial Bus. For this reason only a stream length governs the data transfer.

5.1.2.1 Normal command block ORBs

The format of a normal command block ORB with a single data descriptor is illustrated by the figure below.

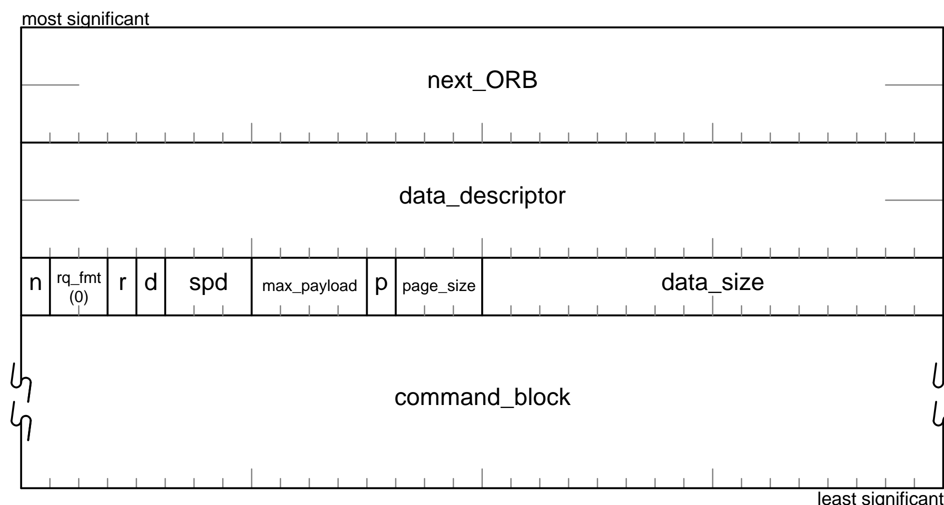

Figure 16 – Normal command block ORB (single data descriptor)

The *next_ORB* field shall contain a null pointer or the address of a dummy ORB or a normal command block ORB and shall conform to the address pointer format illustrated by Figure 12.

The value of the *data_descriptor* field is valid only when *data_size* is nonzero, in which case this field 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 11. When *data_descriptor* specifies the address of a page table, the format of the page table shall conform to that described in 5.2.

The *notify* bit and *rq_fm*t 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 *direction* bit (abbreviated as *d* in the figure above) specifies direction of data transfer for the buffer. If the *direction* bit is zero, the target shall use Serial Bus read transactions to fetch data destined for the device medium. Otherwise, when the *direction* bit is one, the target shall use Serial Bus write transactions to store data obtained from the device medium.

The *spd* field 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 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 maximum data transfer length less than or equal to the length permissible at the data transfer rate specified by *spd*.

The *page_table_present* bit (abbreviated as *p* in the figure above) shall be zero if *data_descriptor* directly addresses the data buffer. When *data_descriptor* addresses a page table, this bit shall be one.

The *page_size* field shall specify the underlying page size of the data buffer memory. 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.

When *page_table_present* is one, the *page_size* field also specifies the format of the data structure that describes the data buffer. A *page_size* value of zero implies the unrestricted page table format (also known as a scatter/gather list). Otherwise, a nonzero *page_size* indicates a normalized page table.

If *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. Otherwise *data_size* shall contain the number of elements in the page table addressed by *data_descriptor*.

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

Normal command block ORBs may have either one or two data descriptors, as determined by the value of the *rq_fmt* field. When *rq_fmt* equals one, the ORB contains two data descriptors, as illustrated by Figure 17.

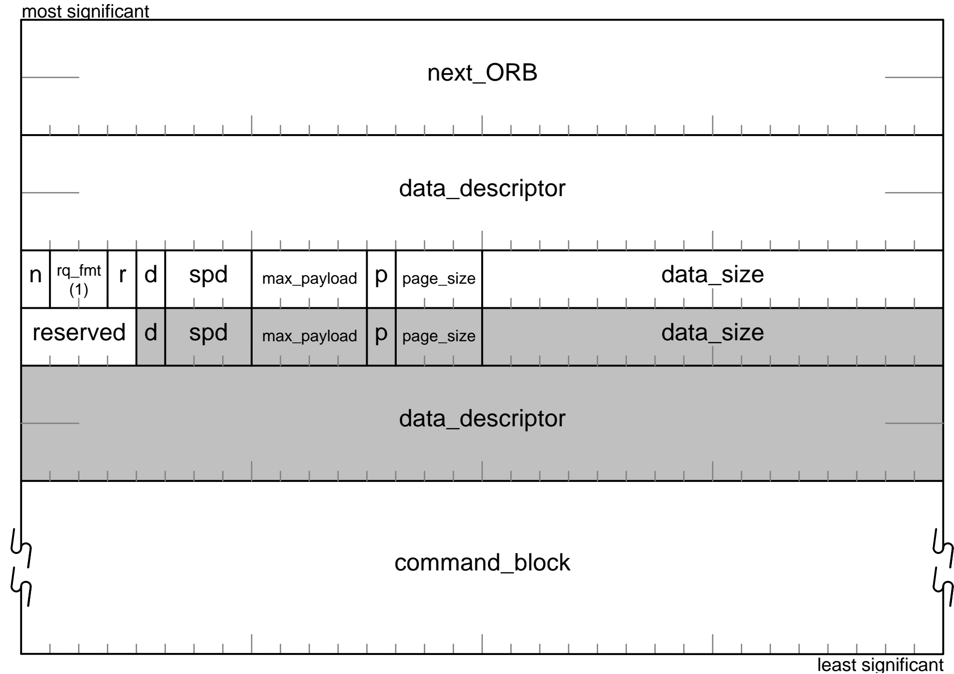

Figure 17 – Normal command block ORB (dual data descriptor)

Each buffer associated with an ORB is described by a set of bits and fields: *data_descriptor*, *direction*, *spd*, *max_payload*, *page_table_present*, *page_size* and *data_size*. A normal command block ORB whose *rq_fmt* is zero describes a single buffer (referred to as *buffer*[0]) with fields located as specified by Figure 16. 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 a normal command block ORB; the additional fields (shown shaded in Figure 17) 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, if present. 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. In particular, unless otherwise specified by the command set used by the target, there is no requirement that a buffer used for inbound data (from the perspective of the node that contains the buffer) be described in a fixed order with respect to a buffer used for outbound data. 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	Buffer usage specified by command set	*buffer*[0] outbound *buffer*[1] inbound
***data_size*[0]** nonzero	***direction*[0]** 1	*buffer*[0] inbound	*buffer*[0] inbound *buffer*[1] outbound	Buffer usage specified by command set

5.1.2.2 Stream command block ORB

A stream command block ORB is a structure that has the format illustrated below.

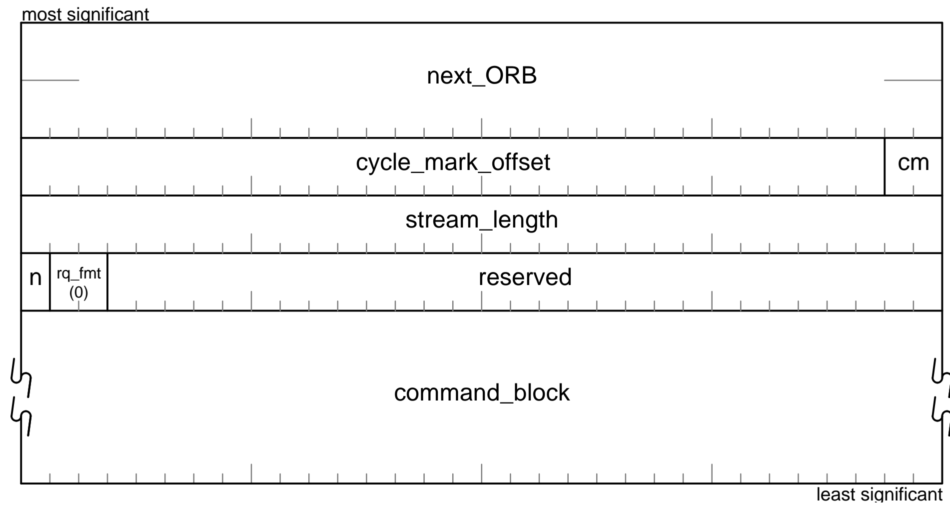

Figure 18 – Stream command block ORB

The *next_ORB* field shall contain a null pointer or the address of a dummy ORB or a stream command block ORB and shall conform to the address pointer format illustrated by Figure 12.

The *cm* field (together with the *cycle_mark_offset* field) specifies the location of the first quadlet of isochronous data (stream offset) as encoded by the table below.

Value	***cycle_mark_offset***	Stream offset
0	Undefined	Zero
1	Undefined	*cycle_mark_offset*
2	Location of first CYCLE MARK	Zero
3	Location of first CYCLE MARK	*cycle_mark_offset*

The stream offset derived from the combination of *cm* and *cycle_mark_offset* specifies the location of the first quadlet of the isochronous data as an offset, in quadlets, relative to the starting medium location indicated by the *command_block*. For a block device, the stream offset, expressed in bytes, shall be less than the block size of the device.

NOTE – The command transported by the stream command block ORB specifies a starting location on the medium and an associated transfer length. Particularly in the case of block devices, the relevant isochronous data may be a subset of the data length and may commence at a nonzero offset relative to the natural block boundaries of the medium—hence the necessity for the additional values, *stream_length* and stream offset, to completely characterize the request.

The *cycle_mark_offset* field, when *cm* has a value of two or three, specifies the location of the first CYCLE MARK packet as an offset, in quadlets, relative to the starting medium location indicated by the *command_block*. When *cm* has a value of one, *cycle_mark_offset* specifies the stream offset instead. In either case, the value of *cycle_mark_offset*, converted to bytes, shall be less than *stream_length*.

NOTE – The *cycle_mark_offset* field may be useful to reestablish synchronization within the recorded isochronous data if a prior stream command block terminated in error.

The *stream_length* field specifies the length of data, in bytes, that is to be transferred to or from the device medium.

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

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

5.1.3 Stream control ORB

Stream control ORBs direct the action of a logical unit stream controller. The stream controller is configured at the time of a create stream request as either a talker or a listener. When listening, the stream controller accepts isochronous data from Serial Bus in accordance with stream control ORBs, transforms the isochronous stream and then records the data on the medium as specified by stream command block ORBs. When talking, this process is reversed and an isochronous data stream obtained from the medium is filtered and transformed by the stream controller before isochronous packets are transmitted on Serial Bus.

The format of the stream control ORB is illustrated below.

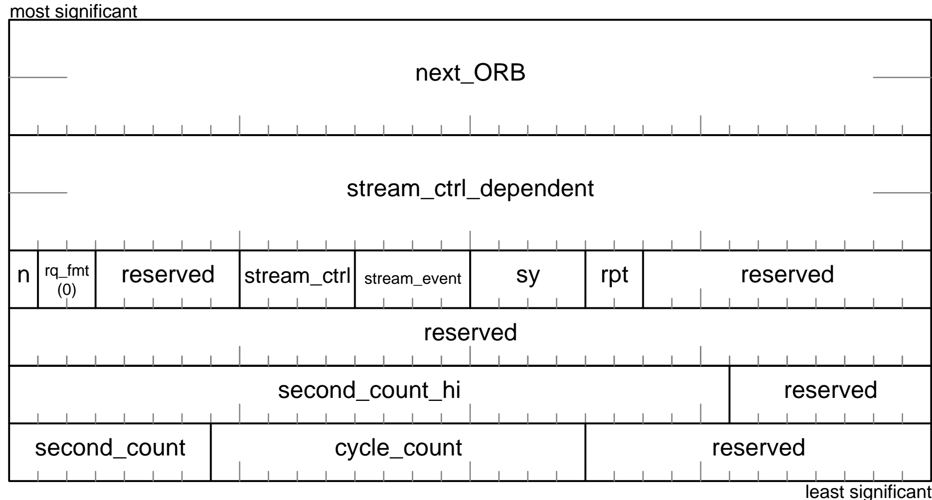

Figure 19 – Stream control ORB

The *next_ORB* field shall contain a null pointer or the address of a dummy ORB or a stream control ORB and shall conform to the address pointer format illustrated by Figure 12.

The usage of the *stream_ctrl_dependent* field varies according to the value of *stream_ctrl* and is described in more detail for each stream control function.

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

The *stream_ctrl* field shall specify a stream control function for the stream, as encoded below.

Value	Stream control function
0	Reserved; not to be used
1	START
2	STOP
3	PAUSE
4	UPDATE CHANNEL MASK
5	CONFIGURE CHANNELS
6	SET ERROR MODE
7	QUERY STREAM STATUS
8 – F_{16}	Reserved for future standardization

Individual stream control functions are described in 5.1.3.1 through 5.1.3.7 below.

The *stream_event* field is valid only if the *stream_ctrl* field specifies a value of START, STOP, PAUSE or UPDATE CHANNEL MASK. When one of these control functions is specified, the *stream_event* field specifies the time at which the action is to take place, as encoded below.

Value	Stream event code
0	IMMEDIATE
1	CYCLE MATCH
2	SY MATCH
3	FIRST DATA
4 – F_{16}	Reserved for future standardization

A value of IMMEDIATE instructs the logical unit's stream controller to perform the specified action as soon as possible, within the capabilities of the target implementation.

A value of CYCLE MATCH instructs the logical unit's stream controller to perform the specified action at the cycle time specified by *second_count_hi*, *second_count* and *cycle_count*.

A value of SY MATCH instructs the logical unit's stream controller to perform the specified action on the isochronous period for which the *sy* field of an isochronous packet for any enabled channel matches the *sy* field in the stream control ORB. A *stream_event* value of SY MATCH is valid only if the logical unit's stream controller is configured as a listener.

A value of FIRST DATA instructs the logical unit's stream controller to perform the specified action when isochronous data is observed for any enabled isochronous channel. A *stream_event* value of FIRST DATA is valid only if the logical unit's stream controller is configured as a listener.

NOTE – A *stream_event* field value of FIRST DATA may have effects similar to IMMEDIATE, in that it is possible for isochronous data to be recorded immediately. The difference between the two stream events is apparent if no isochronous packets for any of the enabled channels are present when the stream control ORB is executed. If IMMEDIATE is specified, CYCLE MARK packets are recorded as each cycle start is observed. If FIRST DATA is specified, no packets are recorded until the first isochronous packet for an enabled channel is observed. When this event occurs, a CYCLE MARK packet with the most recent cycle start data is recorded followed by a DATA packet for the enabled channel.

The *sy* field is valid only if the logical unit's stream controller is configured as a listener and the *stream_event* field specifies SY MATCH. See the preceding description of *stream_event*.

The *rpt* field is described in 5.1.3.6 below.

The *second_count_hi*, *second_count* and *cycle_count* fields are valid only if the *stream_ctrl* field specifies START, STOP, PAUSE or UPDATE CHANNEL MASK and the *stream_event* field specifies CYCLE MATCH. Together, these fields specify a cycle time for comparison with the target's cycle clock. An equal comparison occurs if the *second_count_hi* field matches the field of the same name in the target's BUS_TIME register and if both the *second_count* and *cycle_count* fields match their corresponding fields in the target's CYCLE_TIME register.

5.1.3.1 START stream control function

The START control function instructs the logical unit's stream controller to commence (or resume) talking or listening on Serial Bus. The time at which the action is to occur shall be specified by the *stream_event* field in conjunction with other stream control ORB fields.

5.1.3.2 STOP stream control function

The STOP control function instructs the logical unit's stream controller to terminate the isochronous stream and to flush the stream buffers. The time at which the action is to occur shall be specified by the *stream_event* field in conjunction with other stream control ORB fields.

If the target had been listening, any isochronous data already received from Serial Bus shall be made available to the stream commands previously queued at the stream command block agent. Unless prevented by other errors, the device server shall transfer all flushed isochronous data to the medium in accordance with the queued stream commands. Completion status for the stream command that completes the transfer of isochronous data shall be stored at the initiator's *status_FIFO* and shall reflect the length of the data transfer. Unexecuted stream commands that remain in the task set shall be aborted. If the target had been talking, the stream command task set shall be aborted and any untransmitted isochronous data obtained from the stream commands shall be discarded. The state of the stream command fetch agent is unaffected by the STOP control function.

5.1.3.3 PAUSE stream control function

The PAUSE control function instructs the logical unit's stream controller to suspend the transfer of isochronous data with the expectation that isochronous data transfer will resume. If the target is a talker, the stream controller shall pause on the requested stream event and shall not send any isochronous packets for the stream while paused. Subject to target implementation limitations, data from stream commands previously queued at the stream command block agent may continue to accumulate at the target while the data stream is paused. If the target is a listener, the target shall pause on the requested stream event and shall discard any isochronous packets for the stream while paused. The target may flush any isochronous data already received from Serial Bus in order to make it available to any stream commands previously queued at the stream command block agent.

5.1.3.4 UPDATE CHANNEL MASK stream control function

The UPDATE CHANNEL MASK control function instructs the logical unit's stream controller to change the set of enabled channels. The enabled channels shall be specified by the *channel_mask* field. The time at which the action is to occur shall be specified by the *stream_event* field in conjunction with other stream control ORB fields. When *stream_ctrl* specifies a value of UPDATE CHANNEL MASK, the *stream_ctrl_dependent* field shall contain a 64-bit channel mask, as shown below.

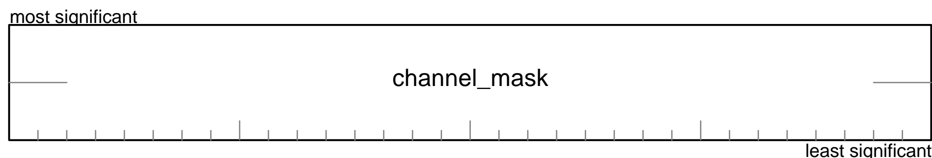

Figure 20 – Channel mask

A one in the bit position that corresponds to one of the numbered Serial Bus isochronous channels, zero to 63, indicates that the channel is to be enabled. Channel zero is represented by the most significant bit while the least significant bit represents channel 63. When a channel is enabled for listening, isochronous packets observed for that channel are transferred to the device medium under control of stream command block ORBs for the stream. Conversely, when a channel is enabled for talking, an isochronous stream is obtained from the medium as directed by stream command block ORBs and isochronous packets are transmitted on Serial Bus for the enabled channel. The channel number specified is the channel number prior to any transformation that is a result of values in the logical unit's stream controller channel map.

5.1.3.5 CONFIGURE CHANNELS stream control function

The CONFIGURE CHANNELS control function instructs the logical unit's stream controller to update the 64-entry channel map maintained internally for the isochronous stream. When listening, channel numbers observed in Serial Bus isochronous packets are replaced with numbers specified by the channel map before the isochronous data is recorded on the medium. When talking, channel numbers encountered in recorded isochronous data are replaced with numbers specified by the channel map before the isochronous packets are transmitted on Serial Bus.

NOTE – It is possible for a mapping of two or more source channels into a single destination channel to be meaningful. For example, isochronous data recorded at different times from different channels may be concatenated on the medium and subsequently replayed as a single channel.

When *stream_ctrl* specifies a value of CONFIGURE CHANNELS, the *stream_ctrl_dependent* field shall contain the address of 64-entry channel configuration map. In this case the *stream_ctrl_dependent* 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 channel configuration map consists of 64 quadlet entries. The channel configuration map is indexed by the source channel. When the target is a listener, the source channel is that observed in the isochronous packet header on Serial Bus. When the target is a talker, the source channel is that previously recorded on the medium. The format of the channel configuration map entries is illustrated below.

Figure 21 – Channel configuration map entry

The *spd* field is valid only if the target is configured as a talker. In this case *spd* determines the speed at which the isochronous packets shall be transmitted, as encoded by Table 1.

The *dest_channel* field shall specify the channel number transformation for either listening or talking. When the target is a listener, the observed channel number in the isochronous packet header shall be replaced with *dest_channel* as the data is recorded on the medium. When the target is a talker, the channel number obtained from the medium shall be replaced with *dest_channel* at the time the isochronous packet is transmitted.

The *sid* field is valid only if the target is configured as a talker. If the recorded isochronous data conforms to the CIP format described in Annex G, the *sid* field in the previously recorded CIP header shall be replaced with the *sid* value specified by the channel map when the packet is transmitted.

5.1.3.6 SET ERROR MODE stream control function

The SET ERROR MODE control function instructs the logical unit's stream controller to configure its error handling mode as specified by the *rpt* field.

The *rpt* field specifies an operational mode for the logical unit's stream controller, as described in the table below. The *rpt* field is valid only if the *stream_ctrl* field specifies SET ERROR MODE.

Value	Error handling mode
0	Report errors and halt stream
1	Report errors and continue stream
2	Ignore all errors
3	Reserved for future standardization

Different sorts of errors may be detected when the logical unit's stream controller is configured as a talker or a listener. If an error occurs, the stream controller shall take one of three actions, as specified by the value of *rpt*:

- Report the error by writing unsolicited status to the initiator and then halting isochronous data transfers by performing the equivalent of a STOP control function with a *stream_event* value of IMMEDIATE;
- Report the error by writing unsolicited status to the initiator but continue isochronous data transfers; or
- Ignore the error and continue isochronous data transfers.

A more detailed description of isochronous errors and how they are handled is provided in 12.3.

5.1.3.7 QUERY STREAM STATUS stream control function

The QUERY STREAM STATUS control function instructs the logical unit's stream controller to return status information that indicates whether or not the stream controller is ready to accept a START control function.

NOTE – Assume that a target is to be instructed to listen to isochronous data and transfer the stream to device medium. If the starting medium location is at a nonzero offset relative to a block boundary, some implementations may require time to read previously recorded data from the medium before being ready to commence recording the new isochronous data. Subsequent to queuing a stream command ORB at the stream command block agent, the QUERY STREAM STATUS control function may be used to determine if the target is ready to accept a START control function.

5.1.4 Management ORBs

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 normal command block, stream command block and stream control ORBs (which are all implicitly associated with a particular task set or stream by virtue of the fetch agent to which they are addressed), most management ORBs explicitly declare the task set or stream 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.

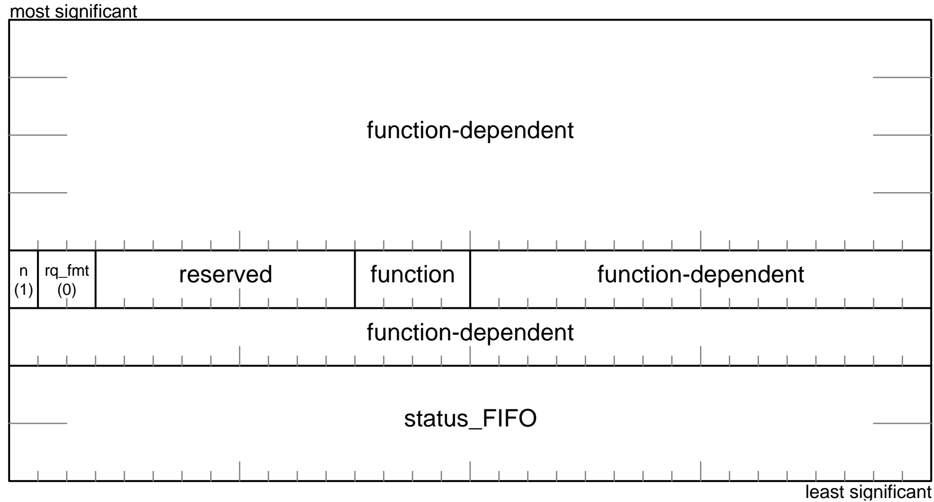

Figure 22 – Management ORB

The *notify* bit and *rq_fm*t 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.

Table 2 – Management request functions

Value	Mandatory	Management function
0	Yes	LOGIN
1	Yes	QUERY LOGINS
2		CREATE STREAM
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 11 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 (normal or stream) or stream control requests. The address in those cases is established by a LOGIN or CREATE STREAM request and is not altered by other management requests.

5.1.4.1 Login ORB

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

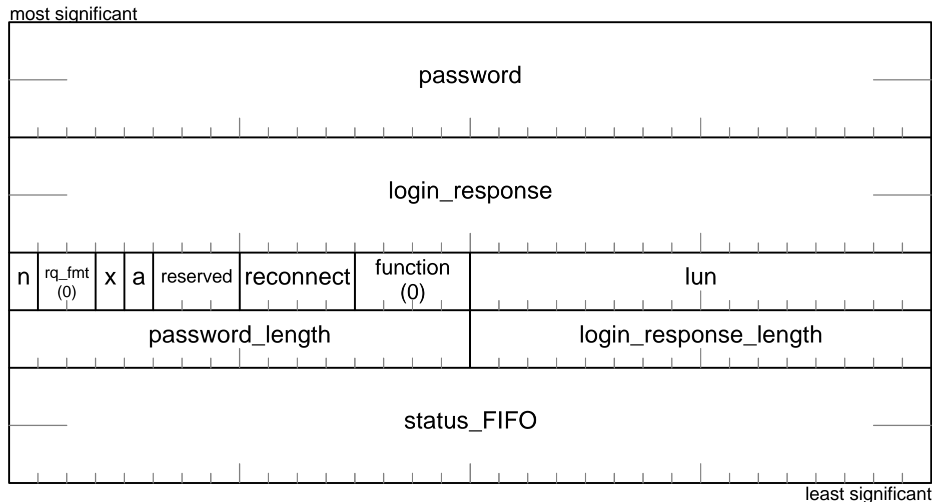

Figure 23 – Login ORB

When the *aware* field is either zero or one, 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 11 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 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 block write request with a data transfer length less than or equal to *login_response_length*. The initiator shall set *login_response_length* to a value of at least 12; 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 8.2 for a description of target behavior.

The *aware* bit (abbreviated as *a* in the figure above) permits the initiator to request target “bridge-aware” behavior for this login. When the *aware* bit is zero, target behavior shall be compatible with SBP-2 (see

[B1]). Otherwise, the target is requested to behave in a “bridge-aware” manner compatible with this standard and draft standard IEEE P1394.1.

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 not be able to support the requested value; see *reconnect_hold* in the login response data below.

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

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 so long as the amount of data stored is an integral number of quadlets. Truncated login response data shall be interpreted as if the omitted fields had been stored as zeros.

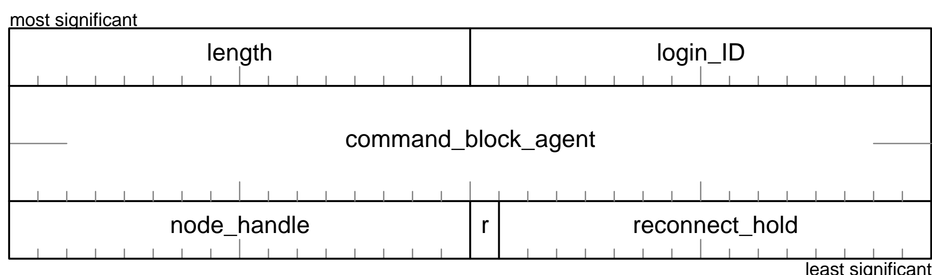

Figure 24 – Login response

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 6.4. This field shall conform to the format for address pointers specified by Figure 11.

When the *aware* field 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 that references initiator memory in an 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. 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, the target will perform a logout and release all resources held by that initiator (see 8.3).

5.1.4.2 Query logins ORB

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

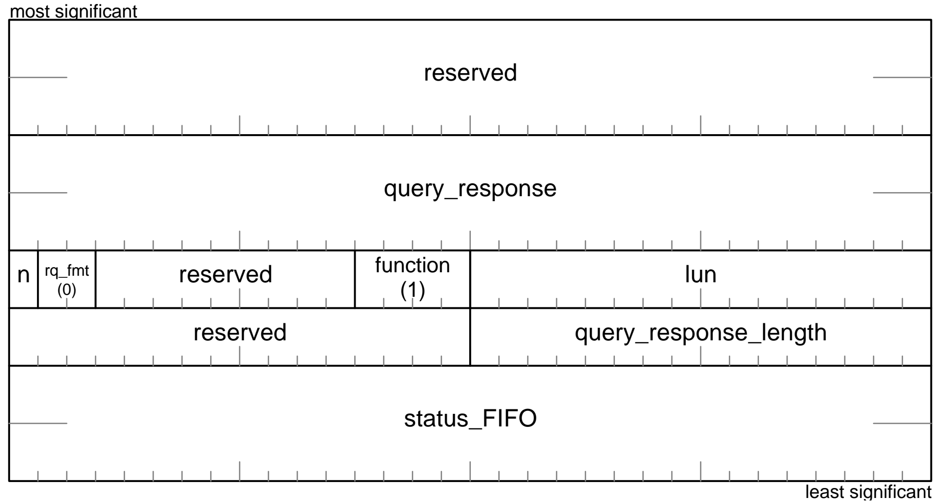

Figure 25 – Query logins ORB

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 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 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.

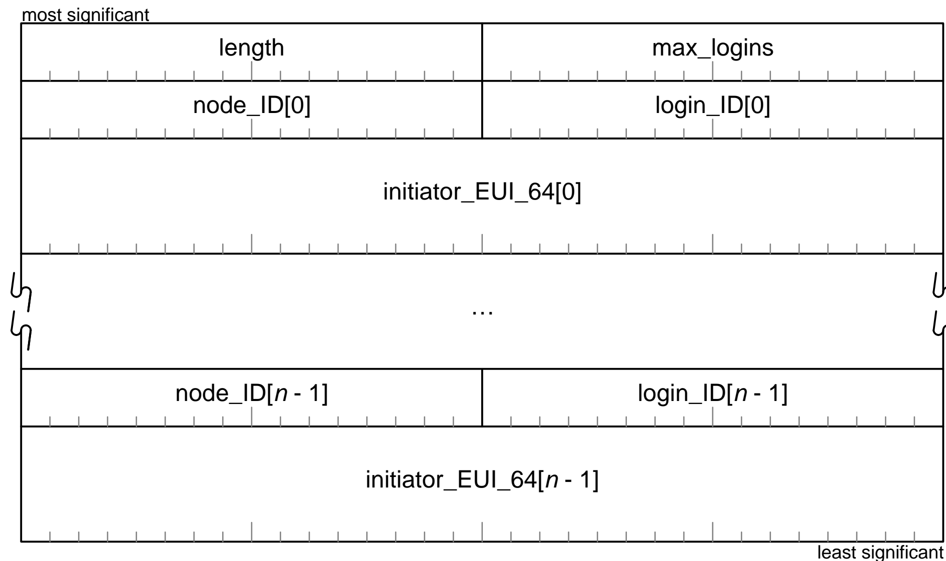

Figure 26 – Query logins response format

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 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 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.

5.1.4.3 Create stream ORB

Before any stream requests are made of a target, the initiator shall first complete a create stream procedure that uses the ORB format shown below.

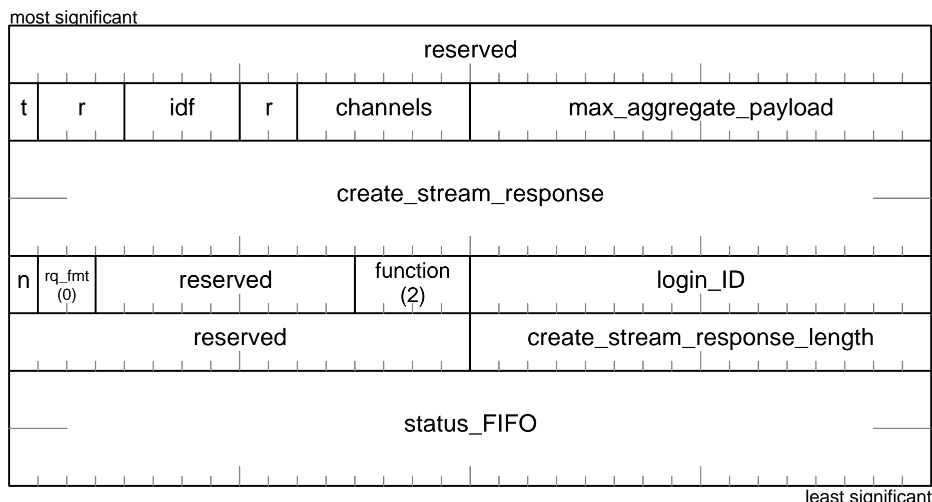

Figure 27 – Create stream ORB

The *talker* bit (abbreviated as *t* in the figure above) shall specify the type of isochronous stream requested. If the target resources are to be configured for listening, *talker* shall be zero.

The *idf* field specifies the format of recorded isochronous data. Valid values for *idf* are shown in the table below.

Value	Data format	Description	Reference
0	Unspecified	The data format is determined by the command set or device type and is beyond the scope of this standard	
1	—	Reserved for future standardization	
2	Isochronous data interchange (without cycle mark indices)	The data is structured by cycle marks, which are recorded on the medium.	Section 11 (also 12.2.3)
~~3~~	~~Isochronous data interchange (with cycle mark indices)~~	~~The data is structured by cycle marks which are recorded on the medium; a cycle mark index is present every 512 bytes.~~	~~Section 11 (also 11.3 and 12.2.3)~~
4 – F_{16}	—	Reserved for future standardization	

The *channels* field specifies the maximum number of isochronous channels that are to be simultaneously transmitted or received.

The *max_aggregate_payload* field is the aggregate maximum isochronous payload that the target is requested to support for the stream. That is, the sum of all the *data_length* fields of Serial Bus isochronous packets transmitted or received for all of the stream's isochronous channels shall not exceed *max_aggregate_payload* in a single isochronous period. The target is not required to enforce this limit.

The *create_stream_response* and *create_stream_response_length* fields specify the address and size of a buffer allocated for the return of the create stream response. The *create_stream_response* field shall conform to the format for address pointers specified by Figure 11. 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_stream_response_length*. The initiator shall set *create_stream_response_length* to a value of at least 24; 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 STREAM request, status for all subsequent requests signaled to either the *stream_command_block_agent* or *stream_control_agent* allocated for this login and any unsolicited isochronous error report(s) generated by the logical unit for this stream.

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

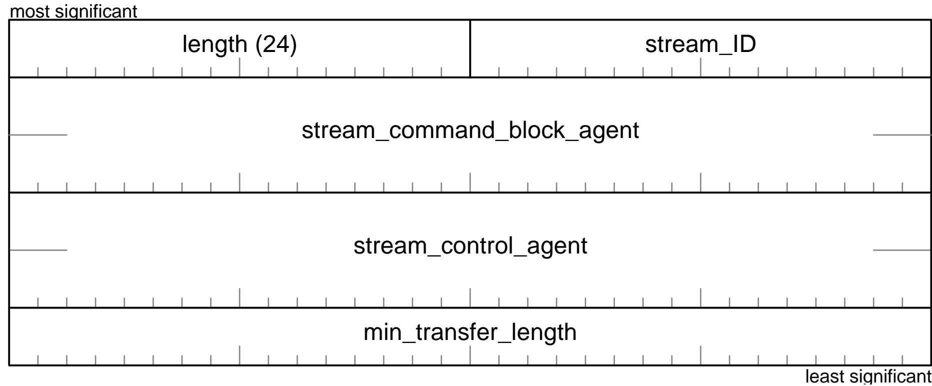

Figure 28 – Create stream response

The *length* field shall contain the length, in bytes, of the create stream response data and shall be equal to 24.

The *stream_ID* identifies an isochronous stream for which target resources have been allocated. The initiator shall use this value to identify all subsequent requests directed to the target's management agent that pertain to this stream.

The *stream_command_block_agent* and *stream_control_agent* fields specify the base address of the agent's CSRs, which are defined in 6.4. Both fields shall conform to the format for address pointers specified by Figure 11. The *node_ID* portion of either field shall have a value equal to the most significant 16 bits of the target's NODE_IDS register. If the target does not implement a stream control agent, the contents of *stream_control_agent* are unspecified and shall be ignored by the initiator.

The *min_transfer_length* field advises the initiator of the minimum *stream_length* value desired by the target in stream command block ORBs in order to sustain the isochronous data transfer rate requested by the login. If the initiator presents any stream command block ORBs whose *stream_length* value is less than this minimum, the target may experience underflow or overflow in isochronous data while talking or listening at the requested rate. Even if this minimum is respected it is still possible for underflow or overflow to occur.

5.1.4.4 Reconnect ORB

After a Serial Bus reset 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.

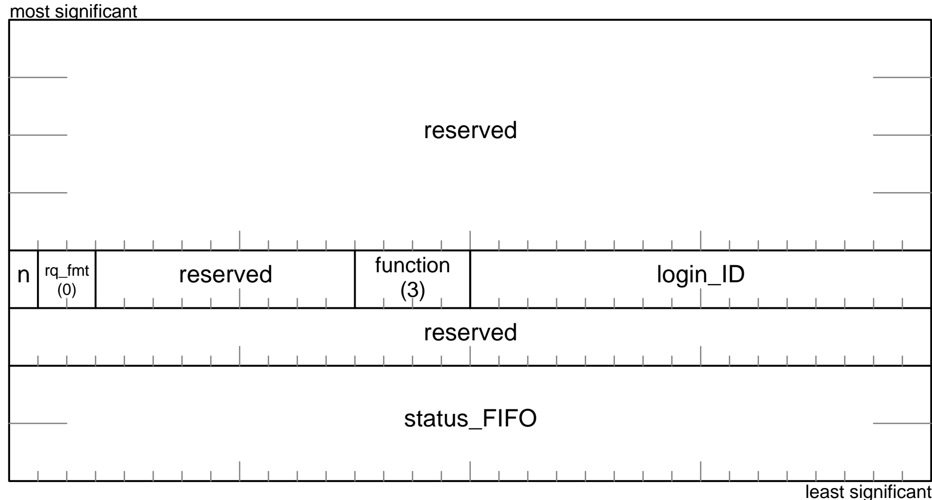

Figure 29 – Reconnect ORB

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 isochronous streams established with the same *login_ID* value specified in the reconnect ORB are also reestablished. The login IDs of the isochronous streams remain the same.

5.1.4.5 Node handle ORB

When an initiator establishes a login in bridge-aware mode, it shall obtain node handle(s) 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.

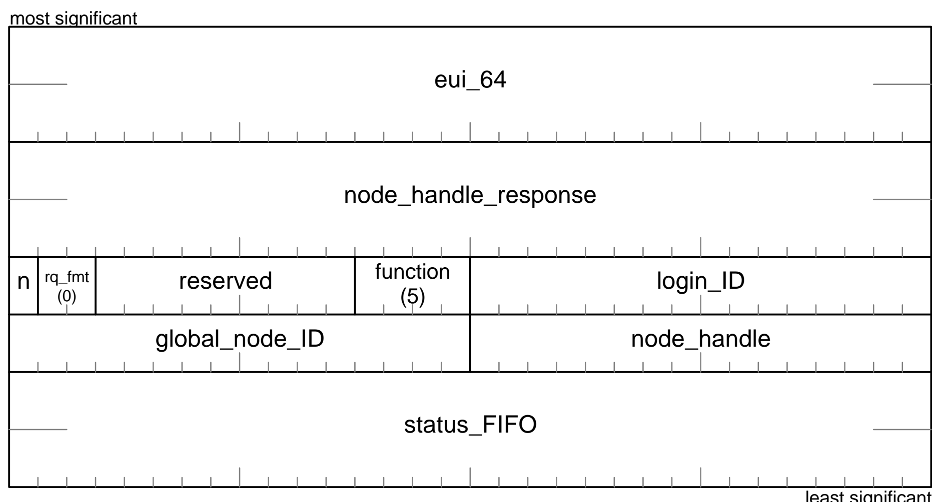

Figure 30 – Node handle ORB

When the *node_handle* field is zero, 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 *node_handle* is nonzero the value of *eui_64* is unspecified.

When the *node_handle* field is zero, 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. The buffer shall be accessible to a Serial Bus quadlet write request. Otherwise, when *node_handle* is nonzero the value of *node_handle_response* is unspecified.

The value of the *node_handle* field determines the operation to be performed. If *node_handle* is zero, the target is requested to provide a node handle that corresponds to the EUI-64 supplied and return it in the buffer specified by *node_handle_response*. When *node_handle* equals $FFFF_{16}$, the target is requested to release all node handles associated with the login identified by *login_ID*. For other nonzero values of *node_handle*, the target is requested to release the node handle specified.

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.

In cases where the node handle ORB is used to request a node handle, the initiator may set the value of the *global_node_ID* field to provide a hint to the target. When the value is all ones, no hint is given. Otherwise, *global_node_ID* contains a global node ID that may identify a node whose EUI-64 matches that specified by the *eui_64* field. In this case, the target, in place of an EUI-64 discovery search, may use a TIMEOUT message (as specified by draft standard IEEE P1394.1) to validate whether or not *global_node_ID* correlates with the specified EUI-64.

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.

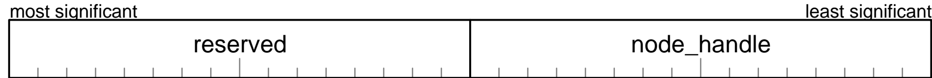

Figure 31 – Node handle response

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 ones. The initiator may use *node_handle* in place of a local node ID 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.

5.1.4.6 Logout ORB

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

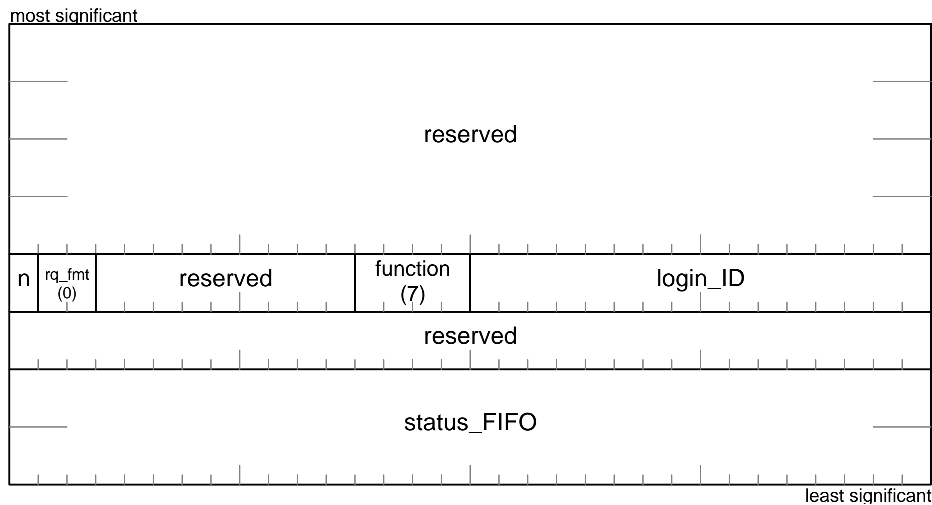

Figure 32 – Logout ORB

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 stream request.

5.1.4.7 Task management ORB

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

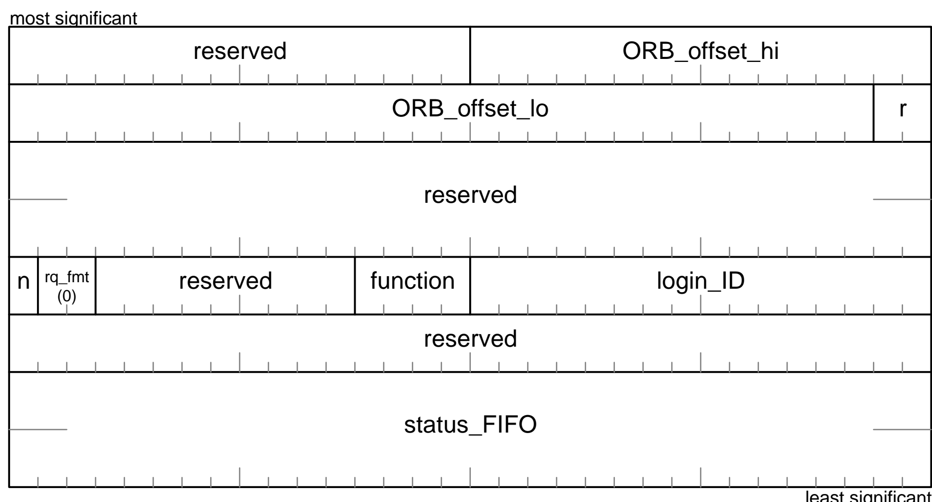

Figure 33 – Task management ORB

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 *stream_ID* returned in create stream response data. In either case, *login_ID* identifies the task set(s) 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.

5.2 Page tables

The data buffer specified by a normal 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.

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 5.2.3) 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.

5.2.1 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 *data_size* * 8 bytes. The format of elements in an unrestricted page table is shown by Figure 34.

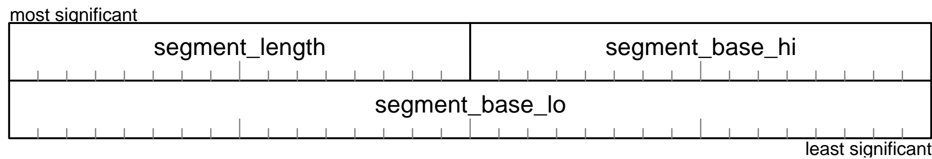

Figure 34 – Page table element (unrestricted page table)

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 5.2.3).

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*.

5.2.2 Normalized page tables

A normalized page table shall be contiguous within Serial Bus address space and shall be accessible to Serial Bus block read transactions with a *data_length* less than or equal to the smaller of *data_size* * 8 bytes or $2^{page_size + 8}$ bytes so long as they do not cross Serial Bus address boundaries that occur every $2^{page_size + 8}$ bytes.

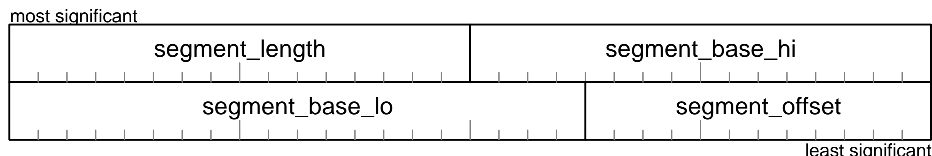

Figure 35 – Page table element (when *page_size* equals four)

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 size 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 5.2.3).

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	

5.2.3 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 36.

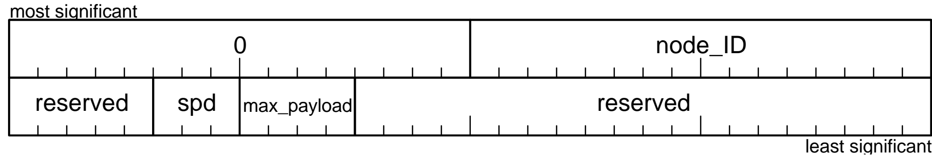

Figure 36 – Node selector

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 normal command block ORB (see 5.1.2.1).

Target support for node selectors is optional and is indicated by the Unit_Characteristics entry in configuration ROM (see 7.6.9).

5.3 Status block

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 isochronous error report). 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 all or part of the status block shown below.

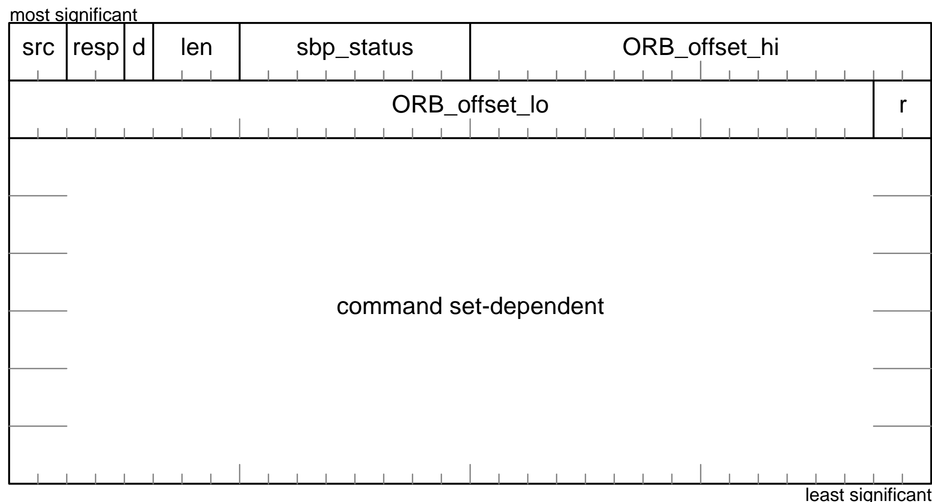

Figure 37 – Status block format

The target shall store a minimum of eight bytes of status information and may store up to the entire 32 bytes defined above so long as the amount of data stored is an integral number of quadlets. A truncated 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 and shall store a status block no more than once for the corresponding ORB.

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

Value	Description
0	The status block pertains to an 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 an 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 is unsolicited and contains isochronous error report information; the contents of the *ORB_offset_hi* and *ORB_offset_lo* fields are redefined to contain the information.

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 the 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 target fetch agent transitioned to 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 the dead state as a consequence of the error condition reported by the status block.

The *len* field shall specify the quantity of valid status block information stored at the *status_FIFO* address. The minimum value of *len* is one. The size of the 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 or one, the *ORB_offset_hi* and *ORB_offset_lo* fields together uniquely identify the ORB to which the status block pertains. For other values of *src*, the *ORB_offset_hi* and *ORB_offset_lo* fields are either ignored or redefined.

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

5.3.1 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 status block shown in Figure 37. 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 normal 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 request while for stream command block and stream control ORBs it is provided in the create stream 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 not recognized
11	Dummy ORB completed
12	Request aborted
FF_{16}	Unspecified error

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 38 – 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 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.

5.3.2 Unsolicited device status

When a change in device status occurs that affects a logical unit, the target may store the status block shown in Figure 37 at the *status_FIFO* address provided by the initiator as part of a login request (see 5.1.4.1). 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.

5.3.3 Unsolicited isochronous error report

Upon detection of an isochronous error while talking or listening, if error reporting has been enabled the target shall store the status block shown below at the *status_FIFO* address provided by the initiator as part of the create stream request (see 5.1.4.3).

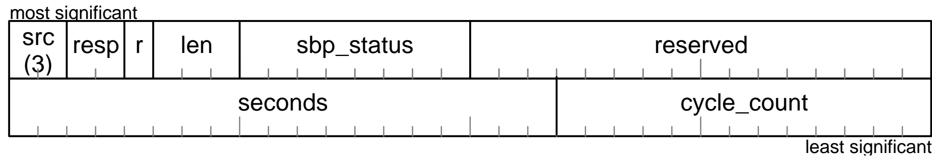

Figure 39 – Unsolicited status format for isochronous errors

The *src* field shall be three to indicate unsolicited status that describes an isochronous error.

The *resp* field shall contain a response status of TRANSPORT FAILURE.

The *len* field shall be equal to one.

The *sbp_status* field shall specify the nature of the isochronous error, as encoded by the table below.

Value	Isochronous error description
0	Reserved (not available for future standardization)
1	Missing CYCLE START packet
2	Data CRC error in received isochronous packet
3	Data length error in received isochronous packet
4	Internal underflow with the result that recorded isochronous data was not transmitted on Serial Bus
5	Internal overflow with the result that isochronous data observed on Serial Bus was not recorded on the medium
6	Format error in recorded isochronous data with the result that data was not transmitted on Serial Bus
7	Data payload in recorded isochronous data too large for transmission on Serial Bus at the requested speed
8 — FE_{16}	Reserved for future standardization
FF_{16}	Unspecified error

The *seconds* field shall contain the least significant 19 bits of the BUS_TIME register at the time of the isochronous stream error.

The *cycle_count* field shall contain the cycle count, between zero and 7999, at the time of the error. The cycle count shall be obtained from the target's free-running cycle timer and shall not be latched from the last observed CYCLE START packet.

6 Control and status registers

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 required by draft standard IEEE P1212;
- bus-dependent registers required by IEEE 1394; and
- unit architecture registers required or permitted by this standard.

Unless otherwise specified, all registers shall support quadlet read and quadlet write transactions. The registers defined in 6.3 and 6.4 shall ignore broadcast write requests.

6.1 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 split transactions

The CSR architecture and IEEE 1394 broadly 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 of one of the currently logged-in initiators.

6.2 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 isochronous channel allocation
234_{16}	BROADCAST_CHANNEL	Channel number for asynchronous stream broadcast.

6.3 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.

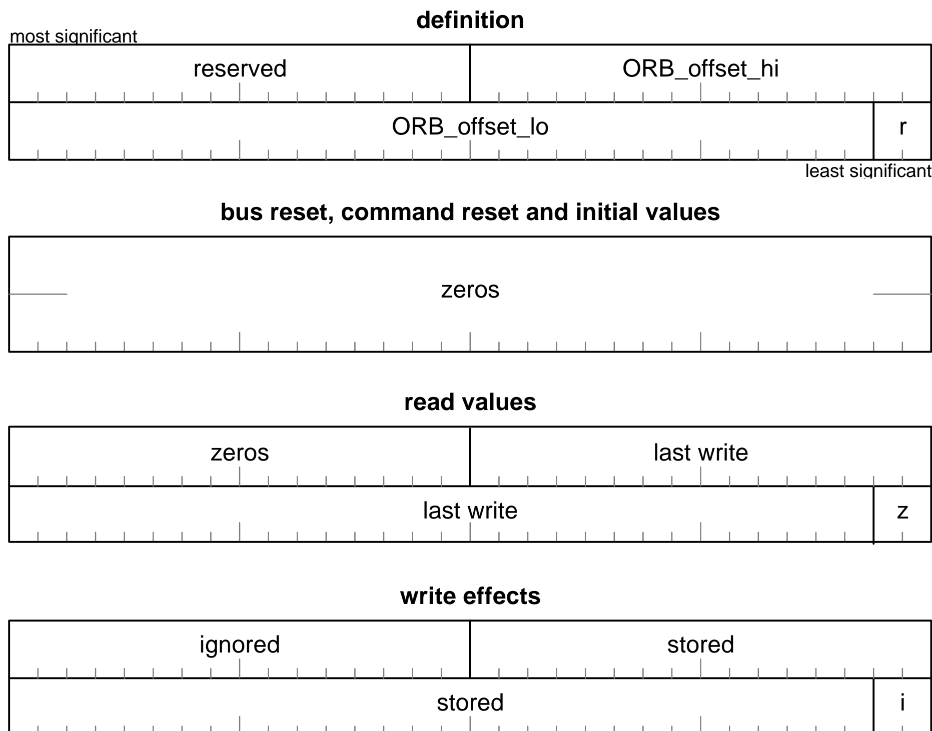

Figure 40 – MANAGEMENT_AGENT format

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 request(s) 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 7.6.8).

6.4 Command block and stream control agent registers

Unlike the management agent, which services a single request at a time, the command block and stream control agents manage linked lists from which they fetch requests. For this reason they are referred to as fetch agents. Each target 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. Target 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 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} – $3C_{16}$		Reserved for future standardization
vendor-dependent	FAST_START	Signals a reset or suspended fetch agent to start a task

The base address of a fetch agent's CSRs is obtained from the appropriate field, *command_block_agent*, *stream_command_block_agent* or *stream_control_agent,* in the response returned by the target as part of a successful login or create stream 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 logged-in to that initiator.

6.4.1 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 41.

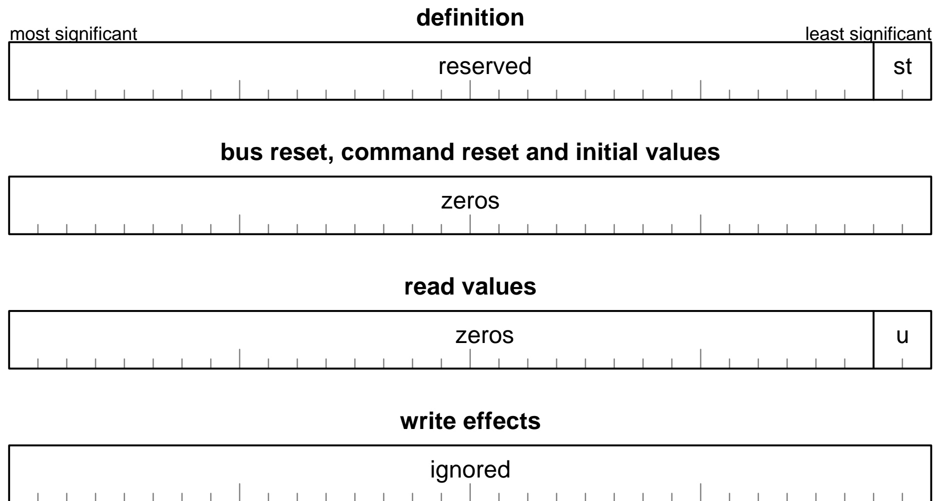

Figure 41 – AGENT_STATE format

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

6.4.2 AGENT_RESET register

The AGENT_RESET register permits an initiator to reset the operational state of a target fetch agent. The definition of this write-only register is given by Figure 42.

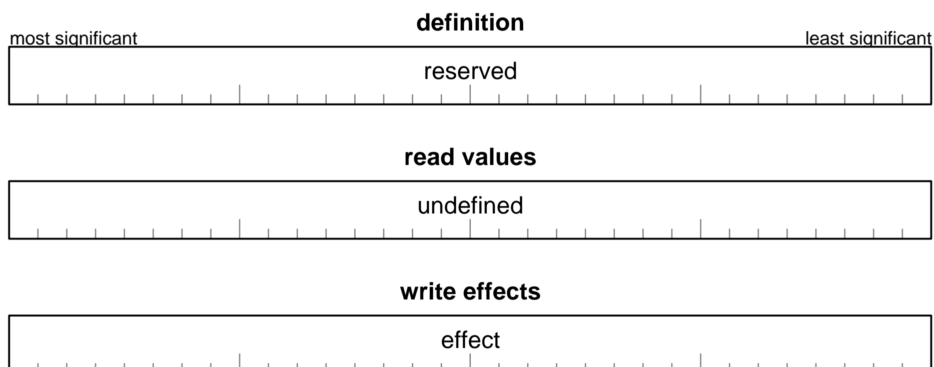

Figure 42 – AGENT_RESET format

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 the reset state.

6.4.3 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 43.

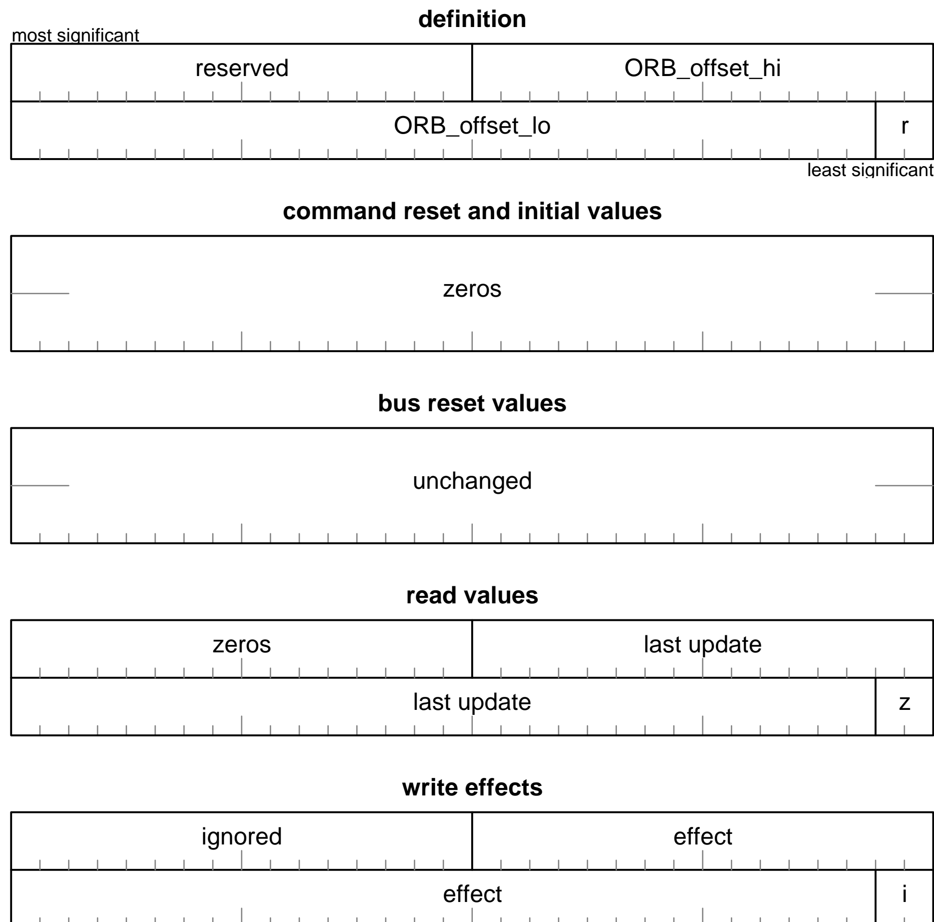

Figure 43 – ORB_POINTER format

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 agent is in the DEAD state, writes to the ORB_POINTER register shall be ignored. If the target 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 the ACTIVE state. If the target agent is in the ACTIVE state, a write to the ORB_POINTER register may cause unpredictable target behavior.

6.4.4 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 44.

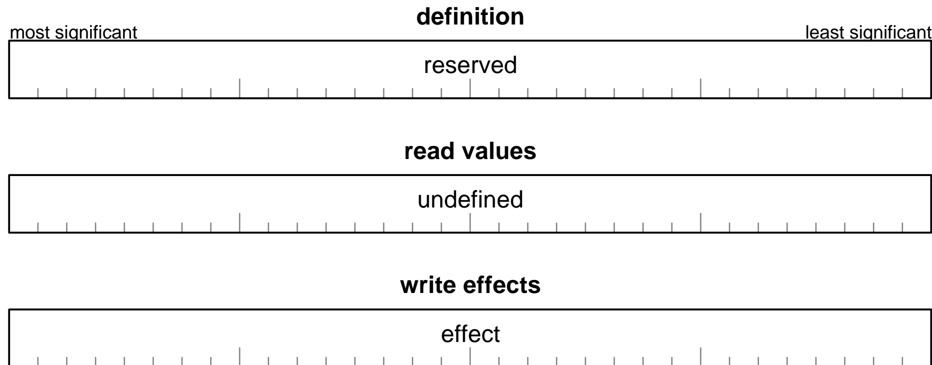

Figure 44 – DOORBELL format

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

6.4.5 UNSOLICITED_STATUS_ENABLE register

The UNSOLICITED_STATUS_ENABLE register provides a means by which the initiator may grant the target permission to store an unsolicited status block. The definition of this write-only register is given by Figure 45.

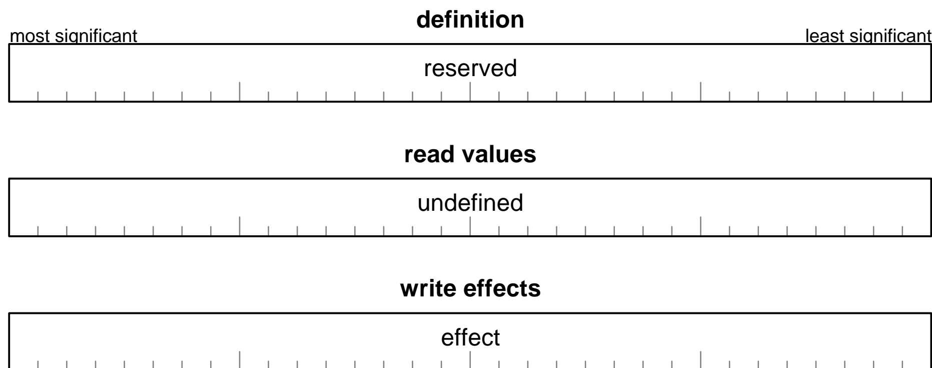

Figure 45 – UNSOLICITED_STATUS_ENABLE format

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 stream request shall zero the *unsolicited status enabled* variable. As described in 9.4, any time a target stores an unsolicited status block it shall zero the *unsolicited status enabled* variable for that login. Before the target may store a subsequent unsolicited status block it is necessary for the initiator to write to the UNSOLICITED_STATUS_ENABLE register.

6.4.6 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 7.6.11) but shall reject all other requests. The format of this register is illustrated below.

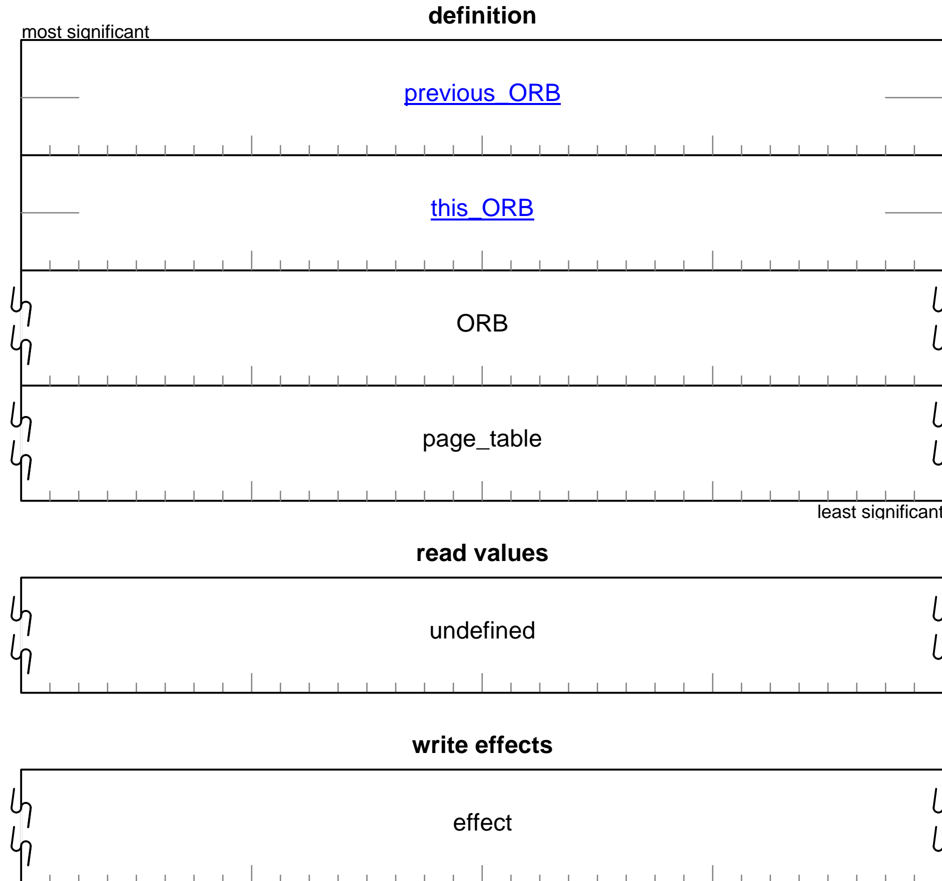

Figure 46 – FAST_START format

The *previous_ORB* field shall conform to the ORB pointer format illustrated by Figure 12 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 ~~*ORB_offset_hi* and *ORB_offset_lo* fields together form an *ORB_offset* field which shall reference~~ *this_ORB* field shall conform to the ORB pointer format illustrated by Figure 12 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 ~~corresponding~~ 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), the ~~*ORB_offset*~~ *this_ORB* field ~~and two least significant bits of zero~~.

The *ORB* field shall contain an ORB whose format conforms to those specified by 5.1. The length of the ORB is variable and shall be determined by the combination of *rq_fmt* and the type of fetch agent (normal or stream) associated with the FAST_START register. An initiator shall not address a block write request to the FAST_START register whose *data_length*, in bytes, is less than ~~eight~~ 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 ~~eight~~ sixteen plus the size of the *ORB* field.

The *page_table* field, if present, shall immediately follow the *ORB* field. If the format of the *ORB* field includes a nonzero *page_table_present* bit, the *page_table* field shall contain zero or more page table entries whose order and content shall be identical to those contained within the page table referenced by the *data_descriptor* field in the ORB referenced by ~~*ORB_offset*~~*this_ORB*. The *page_table* field may be a subset of the page table referenced by the ORB, but no partial page table entries shall be present (see 5.2). 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 *data_length* of block write requests addressed to the FAST_START register shall be a multiple of four.~~

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 ~~target~~ fetch agent is in the DEAD state, writes to the FAST_START register shall be ignored. If the ~~target~~ 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) ~~may cause unpredictable target behavior~~. Otherwise, when the ~~target~~ 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 ~~shall cause the *ORB_offset* to~~. If *previous_ORB* contains a null pointer, *this_ORB* shall be stored in the associated ORB_POINTER register, the *ORB* and *page_table* fields shall be stored in the target's working set and the agent shall transition to 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 9.1.5 for a precise definition of fetch agent state transitions that involve the FAST_START register.

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

7 Configuration ROM

All nodes that implement SBP-3 targets shall implement general format configuration ROM in accordance with draft standard IEEE P1212, 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.

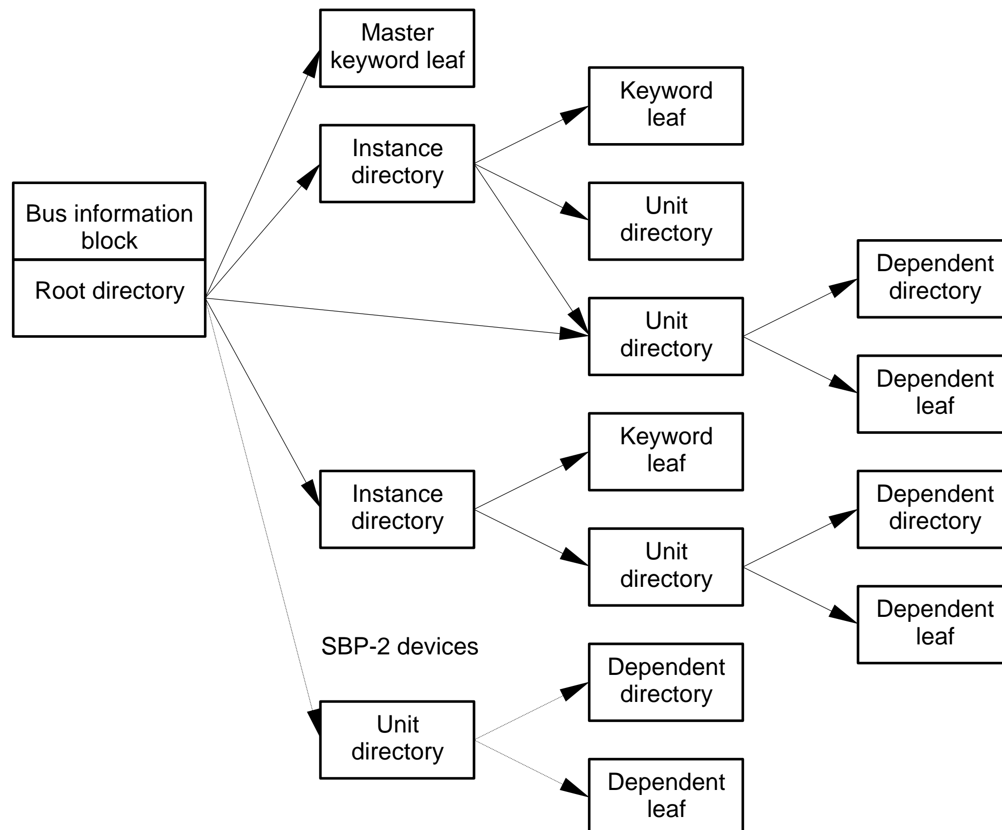

Figure 47 – Configuration ROM hierarchy

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 new with respect to SBP-2. 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 D 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.

7.1 Power reset initialization

During the initialization process that follows a power reset a target may 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 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 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.

7.2 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 standard or its supplements shall be consulted for the most recent information.

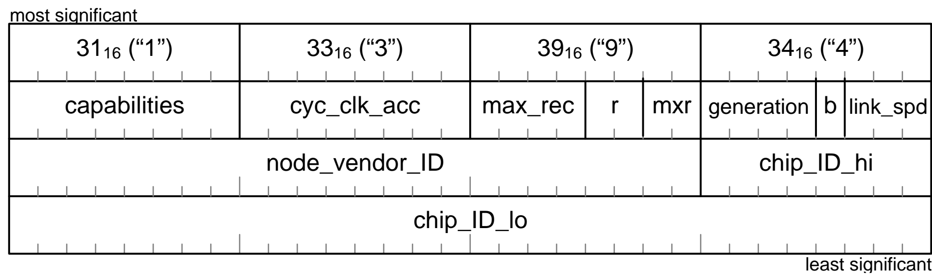

Figure 48 – Bus information block format

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

The *capabilities* field is a collection of bits, illustrated by Figure 49.

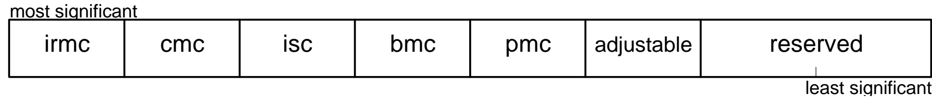

Figure 49 – Bus information block *capabilities* field

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* (abbreviated as *mxr* in the figure above) field 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 draft standard IEEE P1212. 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.

7.3 Root directory

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 Vendor_ID and Node_Capabilities entries.

The root directory shall also contain at least one Unit_Directory entry that specifies the location of a unit directory whose format is specified by this standard.

7.3.1 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 50 shows the format of this entry.

Figure 50 – 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 draft standard IEEE P1212. 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 draft standard IEEE P1212 for the specification of textual descriptor leaves; examples are given in Annex D.

7.3.2 Node_Capabilities entry

The Node_Capabilities entry is an immediate entry in the root directory that describes node capabilities. Figure 51 shows the format of this entry.

Figure 51 – 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 $83C0_{16}$.

7.3.3 Unit_Directory entry

The Unit_Directory entry is a directory entry in the root 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 52 shows the format of this entry.

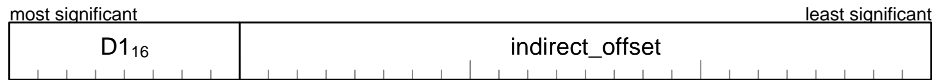

Figure 52 – Unit_Directory entry format

$D1_{16}$ 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.

7.4 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 draft standard IEEE P1212, 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.

7.5 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, already described in 7.6.13).

At a minimum, a logical unit directory shall contain exactly one Logical_Unit_Number entry. It may contain additional entries permitted by 7.6. 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.

7.6 Directory entries

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[3]	Yes

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

Command_Set	Optional[3]	Optional[3]	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
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 draft standard IEEE P1212.

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.

7.6.1 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 53 shows the format of this entry.

Figure 53 – 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 NCITS from the IEEE RAC. The value indicates that the NCITS Secretariat and its Technical Committee T10 are responsible for the maintenance of this standard.

7.6.2 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 54 shows the format of this entry.

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

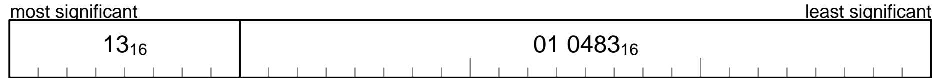

Figure 54 – Version entry format

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.

7.6.3 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 55 – 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.

7.6.4 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 unit(s). Figure 56 shows the format of this entry.

Figure 56 – 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.

7.6.5 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(s). Figure 57 shows the format of this entry.

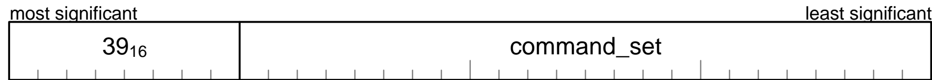

Figure 57 – Command_Set entry format

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*.

7.6.6 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(s). Figure 58 shows the format of this entry.

Figure 58 – 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*.

7.6.7 Firmware_Revision entry

The Firmware_Revision entry is an immediate entry that, when present in the unit directory or logical unit directory, specifies the firmware revision level implemented by the target or logical unit, respectively. Figure 59 shows the format of this entry.

Figure 59 – 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 bridged product that incorporates device(s) native to another transport (*e.g.*, parallel SCSI), the vendor ID in the unit directory might identify the original equipment manufacturer of the bridge chip and firmware while the vendor ID in the root directory might identify the product integrator.

7.6.8 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 60 shows the format of this entry.

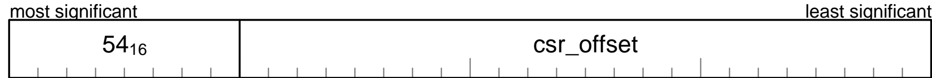

Figure 60 – Management_Agent entry format

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 register(s) pertain to a logical unit, the recommended locations are at offset 20_{16} and above following the base address of the logical unit's command block agent registers. Additional register(s) 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.

7.6.9 Unit_Characteristics entry

The Unit_Characteristics entry is an immediate entry in the unit directory which specifies characteristics of the target implementation. Figure 61 shows the format of this entry.

Figure 61 – 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 5.2.3). 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 an initiator shall allow for a target to store a status block in response to a management ORB. The time-out commences when the initiator receives either *ack_complete* or *resp_complete* from the target in response to the block write of the management ORB address to the MANAGEMENT_AGENT register.

The *ORB_size* field shall specify, in quadlets, the fetch size used by the target 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.

7.6.10 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 62 shows the format of this entry.

Figure 62 – 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 5.1.3.1). 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.

7.6.11 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 6.4.6. Figure 63 shows the format of this entry.

Figure 63 – 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 stream request. The offset shall be specified in quadlets and shall have a minimum value of sixteen.

7.6.12 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.

7.6.13 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.

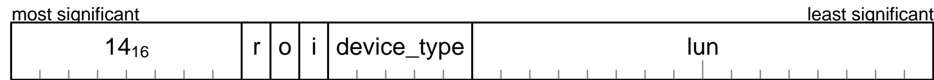

Figure 65 – Logical_Unit_Number entry format

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

The *ordered* bit (abbreviated as *o* in the figure above) specifies the manner in which the logical unit executes tasks signaled to the normal 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 target supports isochronous operations. When *isochronous* is one, create stream requests and stream command block requests shall be supported; stream control requests may be supported. If the *isochronous* bit is one, the *irmc*, *cmc* and *isc* bits in the bus information block shall also be one, as described in 7.2.

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.

7.6.14 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.

7.7 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.

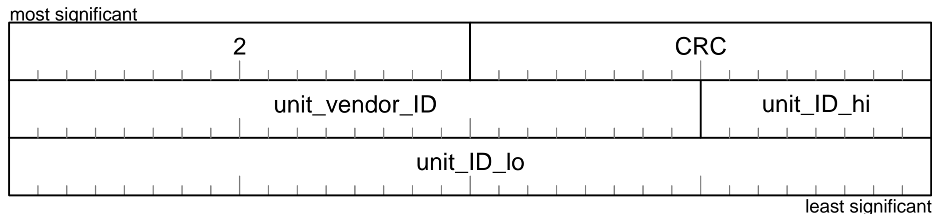

Figure 67 – Unit unique ID leaf format

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 draft standard IEEE P1212.

The *unit_vendor_ID* value shall be uniquely assigned by the IEEE RAC, as specified by draft standard IEEE P1212.

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

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.

8.1 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.

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, the base addresses of the fetch agent CSRs, the *login_ID* to be used by the initiator to identify the login and the *reconnect_hold* period guaranteed by the target —these last three are returned to the initiator in the *login_response* data.
- The *login_owner_ID* is the 16-bit node ID of the current owner of a login. Upon either a Serial Bus reset or a power reset, the *login_owner_ID* for all *login_descriptors* shall be reset to all ones. The target shall use the *login_owner_ID* to qualify all write requests addressed to the *login_descriptor* fetch agent CSRs.
- 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 reset to all ones. Upon a Serial Bus reset, the *login_owner_EUI_64* persists for *reconnect_hold* + 1 seconds and shall then be reset to all ones unless it has been reestablished.

A *login_descriptor* is considered 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* holds 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(s) associated with the *login_descriptor*.

8.2 Access requests

The clauses that follow describe the use of the login and create stream ORBs defined in 5.1.4.1 and 5.1.4.3.

8.2.1 Login

Before an initiator may signal any requests to a target that require a *login_ID* or address fetch agent CSRs, it shall first perform a login. The login request, whose format is specified in 5.1.4.1, 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 login request. The address of the management agent shall be obtained from 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 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 any order) to validate a login request:

- 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;

  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;
- If the *exclusive* bit is set in the login ORB and there are any active *login_descriptors* for the logical unit, the target shall reject the login request with an *sbp_status* of access denied;
- If an active *login_descriptor* with the *exclusive* attribute exists for the *lun* specified in the login ORB, the target shall reject the login request with an *sbp_status* of access denied; else
- 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.

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 *exclusive* variable in the *login_descriptor* is set to the value of the *exclusive* bit 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*. Lastly the target assigns a unique *login_ID* to this login and stores it in the *login_descriptor*.

If the target is able to satisfy the login request, it shall return a login response as specified in 5.1.4.1. A critical component of a login response returned to the initiator is the base address of the target agent that the initiator shall use to signal any subsequent requests to the target for the indicated *login_ID*.

8.2.2 Create stream

An isochronous stream 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 as well as other information in the create stream request that characterizes the isochronous operations to be performed.

The information consists of four items:

- whether the target is to function as a talker or a listener;
- the isochronous data format;
- the maximum number of channels that may be simultaneously enabled; and
- the aggregate maximum isochronous payload for all channels to be transferred between Serial Bus and the device medium in a single isochronous period.

The aggregate maximum isochronous payload is the worst-case amount of data the target may have to transfer to or from Serial Bus and from or to the medium in an isochronous period. Implementation-dependent constraints may limit the performance of the target, which requires this information in order to determine if the login may be accepted. Upon playback (when the target is a talker), the aggregate maximum isochronous payload shall reflect the total of all channels recorded on the medium—not just the aggregation of payload(s) for the channels to be transmitted on Serial Bus. This is essential since the target reads all of the data from the medium even though the channel mask may select a small subset for playback.

These parameters—listener *vs.* talker, isochronous data format, maximum channels and aggregate maximum isochronous payload—may be used by the target to determine if sufficient resources are available to create the stream and, if so, the manner in which they are to be configured.

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

– The target shall validate the *login_ID* supplied in the create stream ORB by comparing the *destination_ID* in the read request(s) 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 *stream_descriptor* is available and, if none are available, reject the create stream request with an *sbp_status* of resources unavailable.

Once the above conditions have been met and a *stream_descriptor* allocated, the *stream_descriptor* is associated with the appropriate *login_descriptor* and the addresses of the fetch agent(s) are stored in the *stream_descriptor*. Lastly the target assigns a unique *stream_ID* to this stream and stores it in the *stream_descriptor*.

NOTE – The *stream_descriptor* may also hold other information from the create stream request, such as listener *vs.* talker, isochronous data format, number of channels or aggregate maximum payload as dictated by the target implementation.

In addition to the addresses of the stream command block and stream control fetch agents, the target shall also specify in the *create_stream_response* data the minimum transfer length that the initiator should specify in the *stream_length* field of any stream command block request signaled to the target.

8.3 Reconnection

Upon a Serial Bus reset, the target shall abort all task sets for all command block agents created as the result of login request(s). Task sets associated with isochronous streams shall not be aborted. Both the stream command block and stream control requests fetched prior to the bus reset shall continue to be executed by the target but the return of status shall be deferred until a successful reconnection occurs. Stream command block and stream control requests shall not be fetched until the login associated with the stream is successfully reconnected.

For each login, the target shall retain, for at least *reconnect_hold* + 1 seconds subsequent to a bus reset, sufficient information to permit an initiator to reconnect its login (and, implicitly, any associated streams). After this time, but within *reconnect_hold* + 2 seconds, the target shall perform an implicit logout for each login ID or stream 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.

NOTE – The rationale for a reconnect hold period of at least one second is to permit initiators to reallocate isochronous channels and bandwidth and to reestablish isochronous connections. 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.

After a Serial Bus reset, an initiator should not signal requests for an otherwise valid login until it first performs a reconnect. The reconnect request, whose format is specified in 5.1.4.3, 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 all 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:

– 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 stream descriptors associated with the same initiator) with the initiator's *source_ID*.

Fetch agents for stream command block and stream control requests for the reconnected initiator may resume; status for completed ORBs that had not been stored in the initiator's *status_FIFO* (because the initiator's *source_ID* had been invalidated by the bus reset) may also be stored.

Upon successful completion of a reconnect request, the fetch agent associated with *login_ID* shall be in the reset state; the state of the fetch agent(s), if any, for the streams dependent upon *login_ID* is not affected by the reconnect request. 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.

8.4 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 stream 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 logout of all streams associated with the *login_ID*. Any tasks or stream control ORBs 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 stream requests.

9 Command execution

This section describes the procedures used by an initiator to request command execution by a target. As described in the model, requests are specified by data structures in system memory that are subsequently fetched by the target. While a target 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 is stored in system memory by the target. 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

9.1 Requests and request lists

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 other target agents, command block and stream control, are characterized as fetch agents since they manage linked lists of requests in system memory and are responsible to fetch the ORBs. For normal command block, stream command block and stream control ORBs, the initiator produces requests and the target consumes them. These processes are asynchronous and independent of each other. Target efficiency is improved if the target 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 target remains active.

Each normal command block, stream command block and stream control 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 target to execute them in order (or not) according to its device-dependent characteristics.

The target is responsible to fetch ORBs from system memory, as described in more detail in 9.1.4. The remainder of this clause describes what the initiator shall do to:

- initialize a target fetch agent;
- dynamically append new requests to an active list and notify a target fetch agent of the new requests; and
- notify a target fetch agent of a single new request.

9.1.1 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 5.1.1. 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 target 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 the ACTIVE state.

The figure below illustrates the result of these actions:

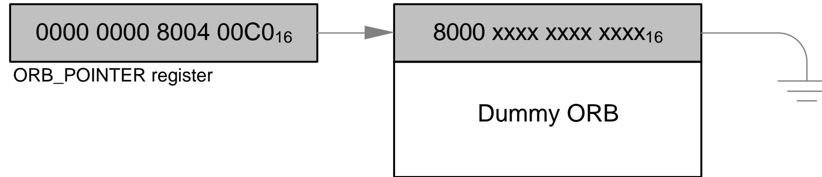

Figure 68 – Fetch agent initialization with a dummy ORB

When the fetch agent transitions to 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 target 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 the suspended state. The ORB_POINTER register still points to the dummy ORB and the initiator may subsequently append additional requests, as described in 9.1.2.

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.

9.1.2 Dynamic appends to request lists (informative)

Once a target 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 target that address pointers in the request list have been updated by the initiator. If the target fetch agent had not encountered a null pointer, the activation of the doorbell is redundant. However, if the target 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 target 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.

9.1.3 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 target fetch agent's capabilities to manage multiple outstanding requests. The BIOS may use

a simpler procedure than that described in 9.1.2 to signal requests to the target. Subsequent to initialization of the target fetch agent by means of a write to the AGENT_RESET register, the BIOS may signal one request at a time to the target 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 target agent by writing the address of the ORB to the ORB_POINTER register in an 8-byte block write transaction. This causes the target agent to transition to 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) are minor; the principal benefit to the BIOS is code simplification.

9.1.4 Use of the FAST_START register (informative)

An initiator ~~aware that a fetch agent is in either the RESET or SUSPENDED state~~ may signal new task(s) to the fetch agent by a block write request addressed to the fetch agent's FAST_START register (see 6.4.6). The block write request contains ~~the address of~~ 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 target 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 target 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 target 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 target 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 target last stored a status block with a *src* field equal to one (see 5.3.1), 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 target 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 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 9.1.2.

9.1.5 Fetch agent state machine

The operations of a target 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 6.4. 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 the 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 target 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*.

[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*.

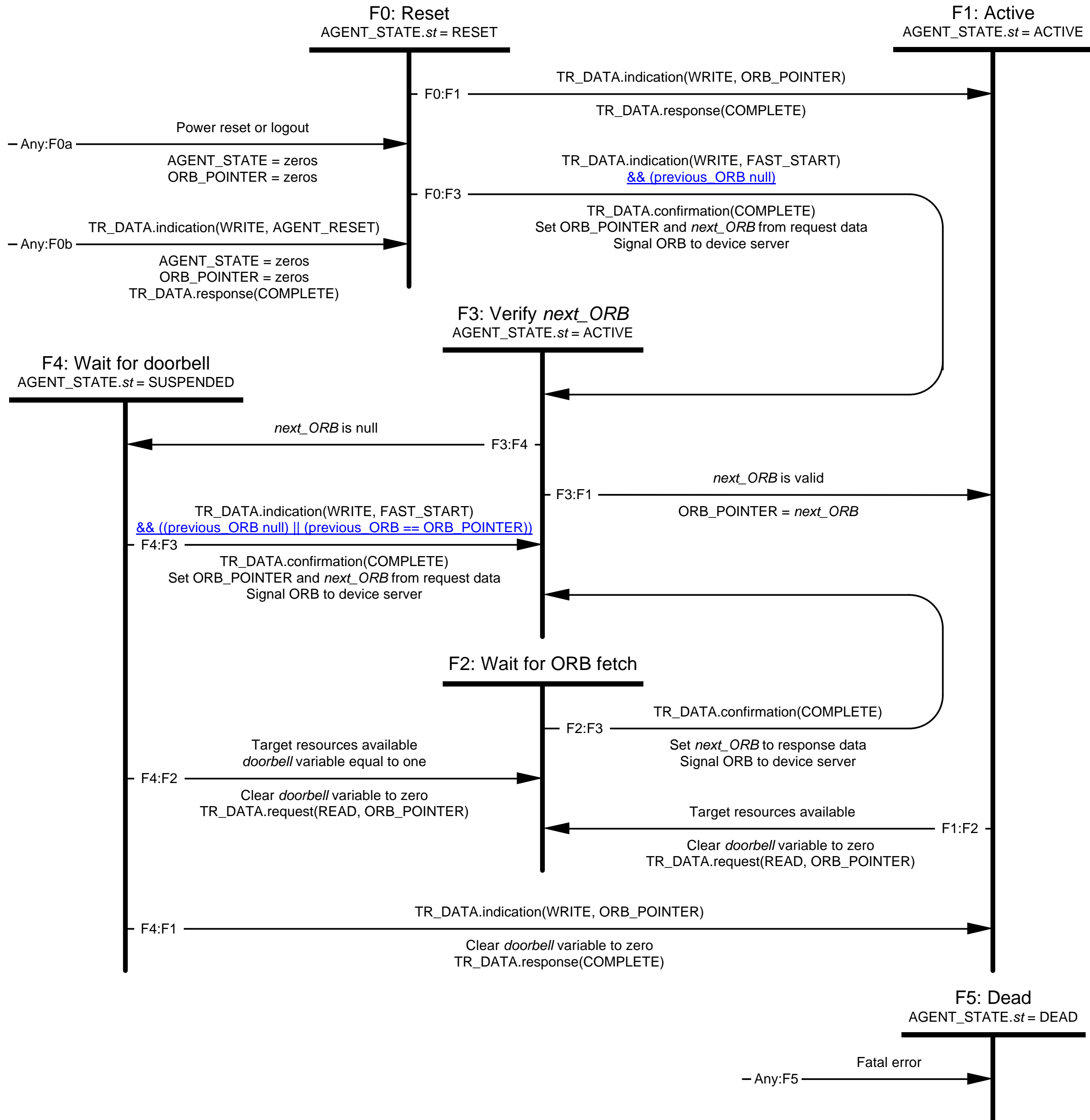

Figure 69 – Fetch agent state machine

Transition Any:F0a. A power reset shall cause the fetch agent to transition to 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 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 label(s) for outstanding request subaction(s) shall not be reused until either the corresponding response subaction completes or a split time-out expires; in the former 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 state F1. The target shall confirm the block write request with a response subaction of COMPLETE.

Transition F0:F3. A block write ~~of a valid *ORB_offset*, ORB and optional page table data~~ 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 ~~*ORB_offset*~~*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 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 ~~first eight bytes~~ *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 5.1, 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*.

Transition F3:F4. The fetch agent shall transition to a suspended state, F4, if *next_ORB* contains a null pointer. A null pointer is defined in 5.1 and exists if the most significant bit of the variable is one.

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

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 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 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 9.3.

Transition F4:F3. A block write ~~of a valid *ORB_offset*, ORB and optional page table data~~ 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 ~~*ORB_offset*~~ *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 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 ~~first eight bytes~~ *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 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 (split time-out);
– a busy condition at the addressed node that exceeds 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 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.

9.2 Data transfer (normal command block ORBs)

The transfer of data associated with a command is the responsibility of the target. 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_table_present* bit and the *page_size* field. If *page_table_present* is one, the target shall observe alignment boundaries that occur every $2^{page_size + 8}$ bytes; no single Serial Bus block read or block write transaction shall cross such a boundary. When *page_table_present* is zero, a *page_size* value of zero indicates that there are no alignment requirements. Nonzero *page_size* values specify alignment boundaries in the same fashion as when a page table is present.

The target shall issue data transfer requests with a speed equal to that specified by the *spd* field in the ORB. The target shall not issue block read or write requests with a data payload length greater than that specified by the *max_payload* field in the ORB.

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.

9.3 Completion status

Upon completion of an ORB, the target shall examine the *notify* bit in the ORB to determine whether or not to store a status block. If *notify* is zero, the target may store a status block. Otherwise, if *notify* is one or if the ORB completed with an error condition, the target shall store a status block. The target shall not store intermediate status for an ORB but shall store a status block only upon the completion, successful or in error, of an ORB. The address for the status block is specified by *status_FIFO*, supplied by the initiator as part of the login or create stream request. The status block, previously described in 5.2.3, 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 32 bytes. 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 split time-out limit.

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 target fetch agent shall transition to 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 block(s) 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, IEEE P1394.3) has additional ordering requirements.

9.4 Unsolicited status

In addition to status associated with a particular ORB, described in the preceding section, a fetch agent 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 or three, according to whether the unsolicited information is device status or the report of an isochronous error, respectively (see 5.2.3).

A fetch agent 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 fetch agent shall zero its *unsolicited status enabled* variable. The initiator may set the fetch agent'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 target 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 target 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 target. Other status information, that does not constitute a unit attention, may be discarded by the target, 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

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.

10.1 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 target for execution. The interactions among these tasks and the ordering relationships, if any, are governed by the task management model implemented by the target.

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 target (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 a normal 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 stream task sets per logical unit. Each time target resources are allocated for isochronous operations (by means of a create stream request), a task set, identified by a stream ID, is created that is associated with a logical unit; there may be zero or more stream task sets associated with a logical unit. Each stream task set is separate and distinct from the normal task set and from other stream task sets: there are no interactions between tasks that belong to different task sets.

NOTE – A successful login, which establishes a normal task set, is a prerequisite to the creation of a stream task set—even if the initiator never signals any commands to the normal task set.

10.2 Basic task management model

Targets shall support, at a minimum, a basic task management model. Targets may implement other task management models so long as they support 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 target implementation and are not subject to control by the initiator. Configuration ROM shall specify whether the target may reorder task execution or not;
- For stream task sets, the target shall execute all tasks in order and report their completion status in the same order. For normal task sets, configuration ROM shall specify whether the target 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 target 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 stream 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 target may reorder the actual execution sequence of any tasks in a task set in any manner, including the concurrent execution of multiple tasks.[8] 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, $\{T_0, T_1, T_2, \ldots T_N\}$ 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.

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.

10.3 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.

10.4 Task management requests

The clauses that follow describe the use of the *rq_fmt* field in ORBs and the task management ORBs defined in 5.1.4.7.

10.4.1 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 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:

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;

[8] 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.

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 5.1.4.7. 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 task(s) 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 a *login_response*, target support for this method of abort task is optional. Otherwise the *login_ID* was returned in a *create_stream_response* (in which case it is a *stream_ID*) and the target shall support this method to abort task(s), 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) So long as 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.

10.4.2 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 5.1.4.7. 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 the 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.

10.4.3 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 5.1.4.7. 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 create 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
d) 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.

10.4.4 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 5.1.4.7. 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 create a unit attention condition for all logged-in initiators other than the initiator, identified by *login_ID*, that signaled the target reset request; and

d) 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.

10.5 Task management event matrix

Common events that affect the state of target fetch agents and their associated task set(s) 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* Normal	AGENT_STATE.*st* Stream	Task set(s) Normal	Task set(s) Stream
Power reset	RESET		Clear all task sets	
Command reset (write to RESET_START)	RESET		Clear all task sets	
Bus reset (immediate)	RESET	—	Clear all task sets	—
Bus reset (after one second)	—		Logout any initiator that has failed to successfully reconnect	
Login	—		—	
Create stream	—		—	
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 initiator's 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 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 target fetch agents and task sets according to the kind of request, login or create stream, by which the initiator first acquired access privileges. A login request allocates normal command block resources while a create stream request allocates stream command block and stream control resources.

Immediately upon detection of a bus reset, all normal command block fetch agents transition to the reset state and their associated task sets are cleared without the return of completion status. Stream command block and stream control fetch agents do not fetch any additional ORBs subsequent to a bus reset but otherwise preserve their state. The task sets associated with these stream agents continue execution, but status for completed commands is held by the target and not stored to the initiator's *status_FIFO*.

For *reconnect_hold* + 1 seconds subsequent to a bus reset, targets save state information for initiators that were logged-in at the time of the 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. If an initiator successfully completes a reconnect request during this period, the actions described in 8.3 occur. For normal command block requests, the task set is empty and, once the fetch agent is initialized, the initiator may signal new ORBs to the target. For both stream command block and stream control agents, fetching operations resume from the same point as before the bus reset. Any completion status held by the target during this one second period may also be stored to the initiator's *status_FIFO* after the successful reconnection.

Once *reconnect_hold* + 1 seconds have elapsed after a bus reset, the target shall automatically perform a logout operation for all login IDs and stream IDs that have not been reconnected with their initiator. This returns all the affected fetch agents to the reset state and aborts any associated stream task sets.

11 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.

11.1 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.

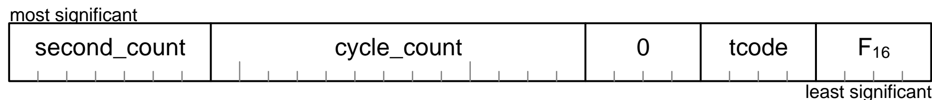

Figure 70 – CYCLE MARK format

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 target 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 target 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.

11.2 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.

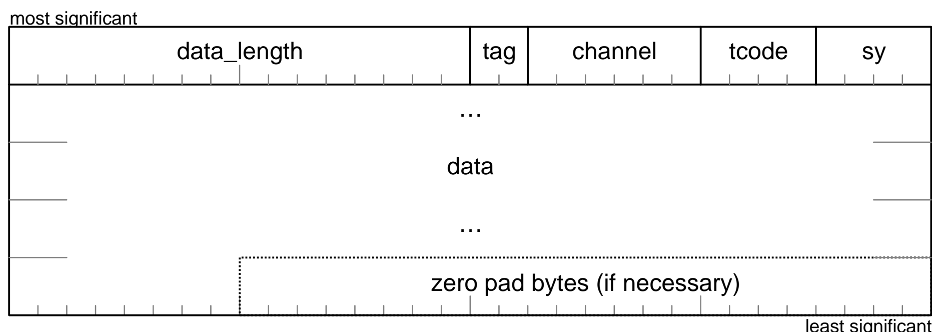

Figure 71 – Format for recorded isochronous data

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 isochronous channel number for the packet. The *channel* field recorded on the medium may have been transformed by the mapping from source channel (observed on Serial Bus) to *dest_channel* specified by a stream control ORB with a CONFIGURE CHANNELS control function (see 5.1.3). Upon playback, the *channel* field shall be transformed by the mapping to *dest_channel* specified by a stream control ORB with a CONFIGURE CHANNELS control function.

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 the application may consider appropriate.

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 target 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).

11.3 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.

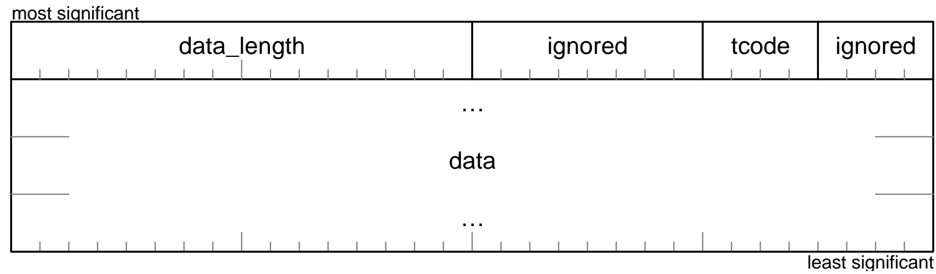

Figure 72 – NULL packet format

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 targets, 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.

12 Isochronous operations

For each active channel on Serial Bus, isochronous data consists of zero or one isochronous packet transmitted by a talker in an isochronous period and received by zero or more listeners in the same isochronous period. This section describes how an initiator may control isochronous data transfers when a target is either the talker or a listener.

From the perspective of Serial Bus, control of isochronous streams involves the following elements:

- the allocation of target resources (create stream requests);
- the establishment or breaking of connections between the target the talker and the listener(s) (connection management);
- the transfer of isochronous data to or from the target’s medium (stream command block ORBs);
- the starting, stopping and synchronization of isochronous data reception or transmission by the target from or to Serial Bus (stream control ORBs); and
- the allocation of Serial Bus resources, such as channel numbers or bandwidth.

The first item, create stream, has already been defined as part of the access procedures in section 8. The second, connection management, may not affect target implementations and therefore is outside of the scope of this document. The last, channel and bandwidth allocation, is specified by IEEE 1394. This section describes the remaining procedures of isochronous operations as they affect targets.

12.1 Stream command block requests

With the exception of the lack of *data_descriptor* and *data_size* fields, stream command block requests are similar to and operate in a like fashion as normal command block requests that transfer data to or from the medium.

The target agent that is responsible to fetch stream command block ORBs operates in the same way as the command block agent(s). The initiator may build linked lists of stream command block ORBs, signal them to the target and dynamically append new stream command block ORBs while the target is active. See section 9 for a more detailed description of command execution; the information is equally applicable to normal command block requests and stream command block ORBs.

Stream command block requests shall be completed in order. The target shall insure that data is transferred in the same order as specified by the linked list of stream command block ORBs. Each isochronous stream active at a logical unit shall have its own task set that is disjoint from all other task sets for the logical unit. The task set for an isochronous stream is instantiated when a successful create stream request completes. The details of a task set for isochronous streams are explained at length in 10.1.

A stream command block ORB that transfers data from the device medium shall return completion status to the initiator when *stream_length* bytes of data have been successfully transferred or an error condition occurs. The return of completion status to the initiator does not provide any information as to how many of the *stream_length* bytes of data have been transmitted on Serial Bus.

A stream command block ORB that transfers data to the device medium shall return completion status to the initiator when *stream_length* bytes of data have been successfully transferred or an error condition occurs.

NOTE – A bus reset can prevent the target from fetching new stream command block requests from the initiator for up to one second—until the initiator successfully completes a reconnection. In order to provide for the uninterrupted flow of isochronous data when a bus reset occurs, the initiator should insure that sum of

stream_length for all the uncompleted stream command block ORBs represents a minimum of slightly more than one second's isochronous data at the configured data rate(s).

12.2 Stream control (optional)

The stream controller may be implemented within the target to mediate the transfer of data between Serial Bus and the medium; the stream command block ORBs contain instructions that govern this transfer. The stream controller's responsibilities include:

- the synchronization, starting and stopping of either the reception or transmission of isochronous data from or to Serial Bus. The synchronization may occur at a specified time or may occur in response to a specified data pattern in observed isochronous data;
- the selective enablement of specified isochronous channels. When listening, the stream controller selects which channels to receive and provide to the stream command block ORBs. When talking, the stream controller selects which channels to transmit from the data provided by stream command block ORBs; and
- the transformation of Serial Bus header and common isochronous packet (CIP) header information between the representation of the isochronous data on Serial Bus and on the device medium. This may include the channel number and time-stamp information and the generation or elimination of null packets, as appropriate.

The target agent that is responsible to fetch stream control ORBs supports a linked list of requests that the initiator may build in system memory. This capability is essential for stream control ORBs, since synchronization boundaries may occur with as little separation as 125 µs. See section 9 for a more detailed description; the information is as applicable to stream control ORBs as it is to normal command block requests and stream command block ORBs.

The clauses below describe the procedures that an initiator shall use to govern the actions of the stream controller. In addition, the stream controller shall transform isochronous header information independent of any stream control ORBs issued by the initiator.

12.2.1 Channel masks

For each stream, targets are configured to receive or transmit a set of zero or more isochronous channels, which may be enabled or disabled independently of each other. The selection of an active set of channels is performed by the stream controller according to a channel mask maintained for each stream.

The channel mask may be updated by a stream control ORB with a UPDATE CHANNEL MASK control function, as described in 5.1.3. If the target is a talker, the channel mask may be updated only when the stream controller is stopped or paused. At all other times a target that is a talker shall reject any attempts to modify the channel mask and shall leave it unchanged.

When the target is a listener, the channel mask refers to channel numbers as they are observed on Serial Bus. The channel mask is applied to select isochronous packets from Serial Bus before any transformations are performed and before the data is recorded on the medium.

When the target is a talker, the channel mask refers to channel numbers as they are recorded on the medium. The channel mask is applied to the stream of isochronous data presented to the stream controller before any transformations are performed and before isochronous packets are transmitted on Serial Bus.

12.2.2 Stream control

The execution of the stream control ORB functions START, STOP, PAUSE and UPDATE CHANNEL MASK can be synchronized with external events, as described in 5.1.3. This section describes the effect of synchronization on the transfer of data between the stream controller and Serial Bus.

When the target is a talker, data flows from the device medium (under the control of stream command block ORBs) to the stream controller. If the stream controller is in a state where isochronous data transmission is enabled (typically as the result of a stream control ORB with a START control function), isochronous data is selected according to the stream's channel mask, is then transformed as described in 12.2.3 and transmitted on Serial Bus. If the stream controller is in a state where isochronous data transmissions are suspended (as the result of a stream control ORB with a PAUSE control function), isochronous data transfer from the medium may continue.

When the target is a listener, data flows from Serial Bus to the stream controller. If the stream controller is in a state where isochronous data reception is enabled (typically as the result of a stream control ORB with a START control function), isochronous data is selected according to the stream's channel mask, is then transformed as described in 12.2.3 and is transferred to the device medium under the control of stream command block ORBs.

12.2.3 Isochronous data transformation

The stream controller is responsible to filter and transform isochronous data as it is recorded on or played from the device medium. Filtering is performed according to the channel number(s) of the isochronous packet(s) (either received from Serial Bus or read from the device medium) before any transformations are applied. Once a set of enabled channel(s) is selected, the stream controller shall:

- replace the channel number found in the source isochronous data with the remapped channel number for the destination isochronous data. This may be the identity map;
- upon playback, if the isochronous data conforms to the common isochronous packet (CIP) format, replace the *sid* (or source ID) field in the CIP header with the source ID specified by a CONFIGURE CHANNELS request;
- when recording, transform all instances of time stamps in the CIP or source packet headers to reflect the delta, or time difference, between the time stamp and the Serial Bus cycle time at which the data is observed; and
- upon playback, add the Serial Bus cycle time at which the data is transmitted to all instances of time stamps in the CIP or source packet headers, in order to recreate an absolute time stamp.

The first two transformations are straightforward and require no additional explanation. The transformations to be applied to time stamps are more complex and are explained below.

The common isochronous packet (CIP) format, as standardized by IEC 61883-1 (1998-02), may contain isochronous time stamp information that is absolute rather than relative. That is, the time stamps contained within header data are a fixed offset ahead of, in the most significant bits, the cycle times contained in the cycle start packet that indicates the cycle in which the packets are transmitted. Time stamps are found in two places in packets that conform to CIP format:

- the *syt* field of the second quadlet of a two-quadlet CIP header if the *fmt* field in that quadlet has a value between zero and $1F_{16}$, inclusive; and
- the *cycle_count* field of the isochronous source packet header, when present.

See 11.3 for the exact specifications of these fields and the circumstances under which they are present in isochronous data.

In order to permit the subsequent recreation of absolute time stamps upon playback, the stream controller shall calculate and store the delta, or time difference, between the time stamp in the observed data and the CYCLE_TIME register when the data is observed.

For the *syt* field, no transformation shall be performed if the observed value is $FFFF_{16}$. Otherwise, the delta time shall be obtained by applying the following formula (shown in C code notation):

$$syt_{stored} = syt_{observed} - (CYCLE_TIME \& 0x0000F000);$$

Unsigned arithmetic is assumed; the results shall be truncated to fit within the 16-bit *syt* field. Upon playback, if *syt* is other than $FFFF_{16}$, a new *syt* shall be calculated using the value of the CYCLE_TIME register for the isochronous period in which the packet is transmitted:

$$syt_{transmitted} = syt_{stored} + (CYCLE_TIME \& 0x0000F000);$$

The result, now an absolute time in terms of current cycle time, shall be truncated to fit within the 16-bit *syt* field and transmitted in the outbound isochronous packet.

For the *cycle_count* component of source packet headers, the delta time shall be obtained by applying the following formula:

$$cycle_count_{stored} = (8000 + cycle_count_{observed} - CYCLE_TIME.cycle_count) \% 8000;$$

Upon playback, the addition of a delta time to the *cycle_count* shall be performed modulus 8000, as shown below:

$$cycle_count_{transmitted} = (cycle_count_{stored} + CYCLE_TIME.cycle_count) \% 8000;$$

The result, now an absolute time in terms of current cycle time, shall be transmitted in the outbound isochronous packet.

12.3 Error logs

When an isochronous stream is active, a stream controller may detect error conditions on Serial Bus or internally. Typical errors include but are not limited to:

- a missing isochronous packet or cycle start indication;
- an isochronous packet with a data CRC error;
- when the target is a talker, an underflow in the availability of data from the stream command block ORBs that causes no data to be transmitted for one or more channels during an isochronous period; or
- when the target is a listener, an overflow in which isochronous data from Serial Bus must be discarded because of an internal buffer overflow or a lack of stream command block ORB(s) to transfer the data to the medium.

The stream controller reports these and other errors by storing unsolicited status at the *status_FIFO* address specified at the time the stream was created. The *rpt* field in a stream control ORB with a SET ERROR MODE control function places the stream controller in one of three error reporting modes:

- report the first error and stop execution of the stream control ORBs;
- report all errors but continue execution of the stream control ORBs; or
- ignore all errors and continue execution of the stream control ORBs.

See 5.1.3 for details of the SET ERROR MODE control function.

Upon detection of an isochronous error while talking or listening, if error reporting has been enabled the target shall store the status block defined in 5.3.3 at the *status_FIFO* address provided by the initiator as part of the create stream request (see 5.1.4.3).

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 initiator(s) or target(s) 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 normal 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 ~~of~~ 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 initiator(s) 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 set(s). 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 normal command block ORB, as illustrated by Figure B-1. There is no fundamental limit on the size of a normal 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.

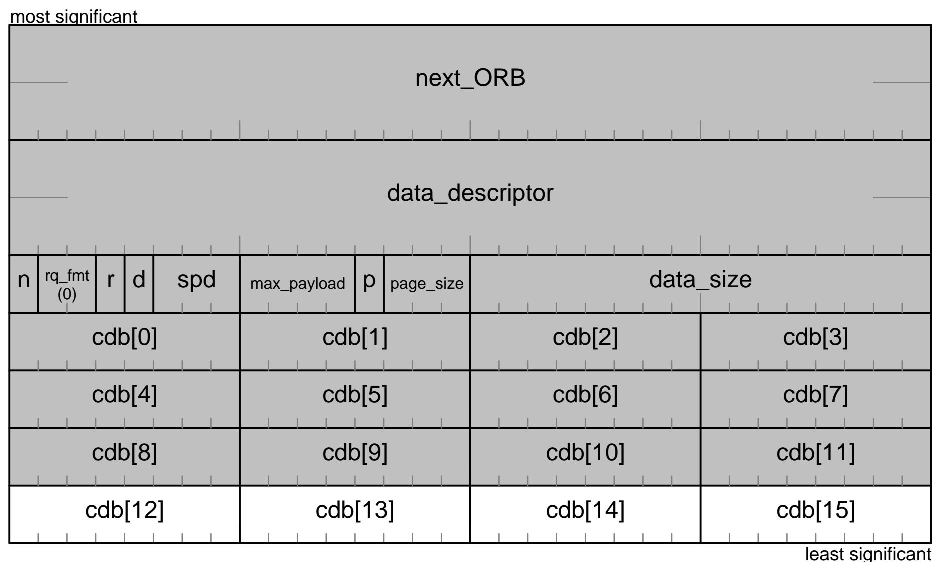

Figure B-1 – SCSI command block ORB

A SCSI CDB may also be transported within a stream command block ORB (see 5.1.2.2).

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 supports SCSI-2 contingent allegiance.

The *flag* and *link* bits 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 status block shown by Figure B-3 at the *status_FIFO* address provided by the initiator as part of the login request.

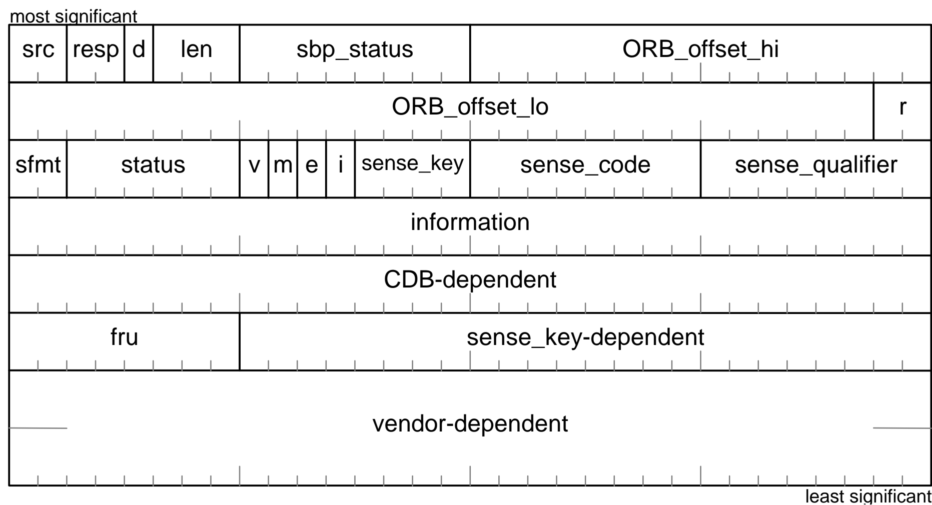

Figure B-3 – Status block format for SCSI sense data

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 the format illustrated above.

SBP-3 permits the return of a status block between two and eight quadlets in length. When a truncated status block is stored, the omitted quadlets shall be interpreted as if zero values were stored.

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 5.3.

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; status block format defined by this standard
1	Deferred error; status block format defined by this standard
2	Reserved for future standardization
3	Status block format vendor-dependent

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
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 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 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.

The *sense_key*, *sense_code* and *sense_qualifier* 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 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 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 are defined by SPC-2 or the relevant the 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.

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 7.6.3.

SCSI targets shall have a *command_set_spec_ID* value of $00\,609E_{16}$, which indicates that NCITS 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 7.6.5.

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 standard(s)—as determined by the target's peripheral device type(s). 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 7.6.13.

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 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 any order) to validate a login request:

- The target shall reject the login request if *login_attempts* is equal to three;
- 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;

  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;
- 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 fails to match either the target's current or master password, the *login_attempts* shall be incremented and the login request shall be rejected with an *sbp_status* of access denied;

- 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 but shall not increment *login_attempts*;
- If an active *login_descriptor* with the *exclusive* attribute exists for the *lun* specified in the login ORB, the target shall reject the login request (with an *sbp_status* of access denied) but shall not increment *login_attempts*; else
- 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.

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 *exclusive* variable in the *login_descriptor* is set to the value of the *exclusive* bit from the login ORB and the address of the fetch agent is stored in the *login_descriptor*. Lastly the target assigns a unique *login_ID* to this login and stores it in the *login_descriptor*.

If the target is able to satisfy the login request, it shall zero *login_attempts* and return a login response as specified in 5.1.4.1.

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.

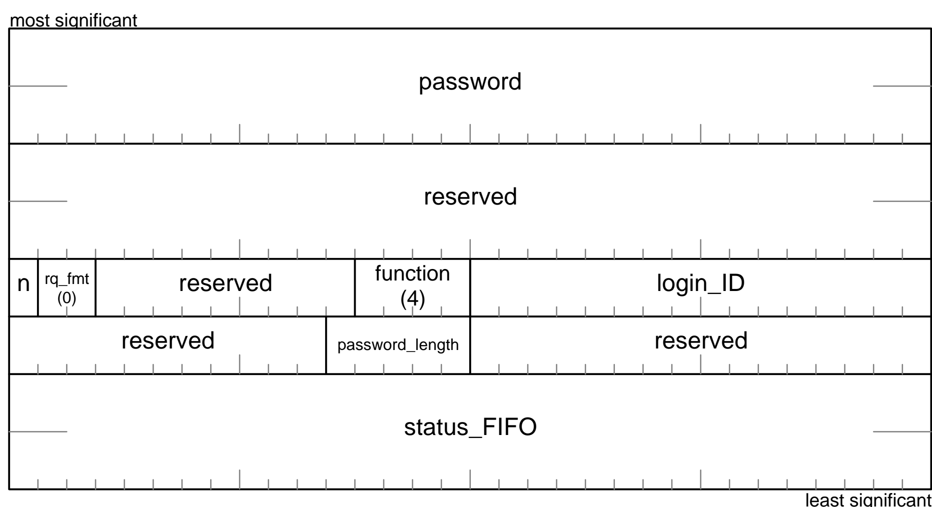

Figure C-1 – Set password ORB

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 11 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
(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.

D.1 Basic target

Figure D-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 D-2.

most significant

4	4	ROM CRC (calculated)	
3133 3934_{16} (ASCII "1394")			
node_options (00FF 2000_{16})			
node_vendor_ID			chip_ID_hi
chip_ID_lo			
4		Root directory CRC (calculated)	
03_{16}	~~module_~~vendor_ID		
81_{16}	Text leaf offset		
$0C_{16}$	node_capabilities (00 $83C0_{16}$)		
18_{16}	Instance directory offset (1)		
2		Instance directory CRC (calculated)	
19_{16}	Keyword lead offset (2)		
11_{16}	Unit directory offset (3)		
1		Keyword leaf CRC (calculated)	
53 4250_{16} (ASCII "SBP")			0

least significant

Figure D-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.

D.1.1 Root directory

The *node_options* field represents a collection of bits and fields specified in 7.3.2. The value shown, 00FF 2000_{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} and a *max_rec* value of two. The *max_rec* field encodes a maximum payload of eight bytes in block write requests addressed to the target.

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 D.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.

D.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 keyword leaf.

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 D.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.

D.1.3 Unit directory

most significant

7	Unit directory CRC (calculated)
12_{16}	specifier_ID (00 $609E_{16}$)
13_{16}	version (01 $05BB_{16}$)
38_{16}	command_set_spec_ID
39_{16}	command_set
54_{16}	csr_offset (00 4000_{16})
$3A_{16}$	Unit characteristics (00 $0A08_{16}$)
14_{16}	Device type and LUN (0)

least significant

Figure D-2 – Basic unit directory

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.

D.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 set(s) and are not repeated below. Figure D-3 shows an example of a unit directory and textual descriptor leaves for a SCSI direct-access device.

most significant

9		Unit directory CRC (calculated)
12_{16}	specifier_ID (00 $609E_{16}$)	
13_{16}	version (01 $05BB_{16}$)	
38_{16}	command_set_spec_ID (00 $609E_{16}$)	
39_{16}	command_set (01 $04D8_{16}$)	
54_{16}	csr_offset (00 4000_{16})	
$3A_{16}$	Unit characteristics (00 $0A08_{16}$)	
14_{16}	Device type and LUN (0)	
17_{16}	model_ID	
81_{16}	Text leaf offset (5)	
3		Text leaf CRC (calculated)
spec_type (0)	specifier_ID (0)	
language_ID (0)		
5431 3000_{16} (ASCII “T10”)		
3		Text leaf CRC (calculated)
spec_type (0)	specifier_ID (0)	
language_ID (0)		
5151 5151_{16} (ASCII “QQQQ”)		

least significant

Figure D-3 – SCSI configuration ROM

D.2.1 Unit directory

The Command_Set_Spec_ID and Command_Set_Version entries, with *key* fields of 38_{16} and 39_{16}, respectively, specify that the target 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 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".

D.2.2 Textual descriptor leaves

Textual descriptor leaf entries, specified by draft standard IEEE P1212, 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 two instances shown.

Annex E
(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 a 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, 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.

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.

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 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.

E.1 MANAGEMENT_AGENT write request

When a management ORB is signaled to a target by means of an 8-byte 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.

E.2 ORB_POINTER or FAST_START write request

A consequence of a write to either the ORB_POINTER or FAST_START register ~~is valid only~~ when the ~~target~~ fetch agent is in the RESET or SUSPENDED state~~. A consequence of the write~~ is that, if successful, the ~~target~~ fetch agent transitions to 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. An exception is a write to the FAST_START register with a non-null *previous_ORB* field; because the target compares the *previous_ORB* field to the ORB_POINTER register, the write may be retried.

E.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. The target should wait a minimum of a SPLIT_TIMEOUT period before the transaction label is reused in any subsequent request subaction.

E.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 F
(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.

F.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 not be clear and in those cases where SBP-3 restricts the scope of an object

Initiator identifier: The *login_ID* returned by a target in response to a successful login is the initiator identifier for normal 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.

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 target 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 initiator 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 identifier: Targets are identified by means of a unit unique ID, or EUI-64, found in the target's 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 identifier, 2^{64}, is identical to the scope of a target identifier specified by SAM-2.

Task set: An SBP-3 task set consists of the linked list of normal command block ORBs that are managed by a single target 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…" (emphasis added). By way of contrast with SAM-2, an SBP-3 task enters the 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 initiator memory, not solely the requests already fetched by the target.

Untagged task: SBP-3 does not define untagged tasks.

F.2 Status

SCSI status is reported in the *status* field, which is part of the status block defined in B.2.

F.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.

F.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]

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 target fetch agent to which the ORB is signaled. The request is signaled to a target fetch agent by the methods specified by section 9.

The CDB argument is encapsulated within an ORB and fetched by the target from initiator memory.

The task attribute argument, either SIMPLE or ORDERED, is implicit in the target 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, the 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 initiator may be expected to reformat the information as SCSI sense data before it is presented to the application client.

F.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]

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 target fetch agent to which the ORB is signaled. A Serial Bus write indication for the AGENT_RESET register signals the target 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 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 from initiator memory.

The *status_FIFO* address, provided by the initiator as part of the login procedure, is the address for the return of autosense data.

F.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 initiator 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 F-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 target failure	1	Various (as defined by SBP-3)	The command has completed because of a Serial Bus service failure or a target malfunction
Function rejected	0	9	Used by task management functions

F.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 initiator 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 F-1.

F.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.

F.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 implementations reference a copy of the ORB maintained in the device's local memory, although nothing precludes a fetch of the information from the initiator 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 initiator.

The request byte count is determined by the device server.

The target 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 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.

F.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 receives Serial Bus write response confirmations. It is the target'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.

F.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 implementations reference a copy of the ORB maintained in the device's local memory, although nothing precludes a fetch of the information from the initiator 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 initiator.

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 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 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.

F.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 receives Serial Bus read response confirmations and their accompanying data. It is the target'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.

F.5 Contingent allegiance

SBP-3 targets implement SCSI-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 targets 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 initiator;
b) aborts the task set in the same fashion as if an ABORT TASK SET task management function had been signaled to the target;
c) clears the contingent allegiance condition.

Because the faulted initiator's fetch agent has been halted, it is necessary for the initiator to reset and reinitialize the fetch agent before any commands may be signaled to the target.

F.6 Asynchronous event reporting

SBP-3 does not support asynchronous event reporting as defined by SAM-2.

F.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.

F.8 Hard reset

A write to the RESET_START register (see 6.1) or a task management function of TARGET RESET causes the target to execute a hard reset, as defined by SAM-2.

F.9 Task set type

SBP-3 targets implement one task set per initiator per logical unit. For targets 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.

F.10 Task management functions

SBP-3 targets 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 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	Functions as TARGET RESET but scope is limited to a single logical unit
TERMINATE TASK	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 target failure. These service responses are encoded by the *resp* and *sbp_status* fields in the status block stored by the target upon completion of a request. See Table F-1 for the numeric values that encode the service responses.

F.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 initiator memory, the initiator signals the request to the target management agent by the methods described in SBP-3.

F.10.2 Task Management Request Received

The formal arguments of the Task Management Request Received service are:

Nexus,
Function identifier

When a Serial Bus write indication is received for the target's MANAGEMENT_AGENT register, the target may fetch the request from initiator memory.

F.10.3 Task Management Function Executed

The formal arguments of the Task Management Function Executed service are:

Nexus,
Service response

The target 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.

F.10.4 Received Function-Executed

The formal arguments of the Received Function-Executed service are:

Nexus,
Service response

When the initiator 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 G
(informative)

Common isochronous packet (CIP) format

Isochronous data packets (already described in 11.2) 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 G-1 illustrates the organization of the common isochronous packet format.

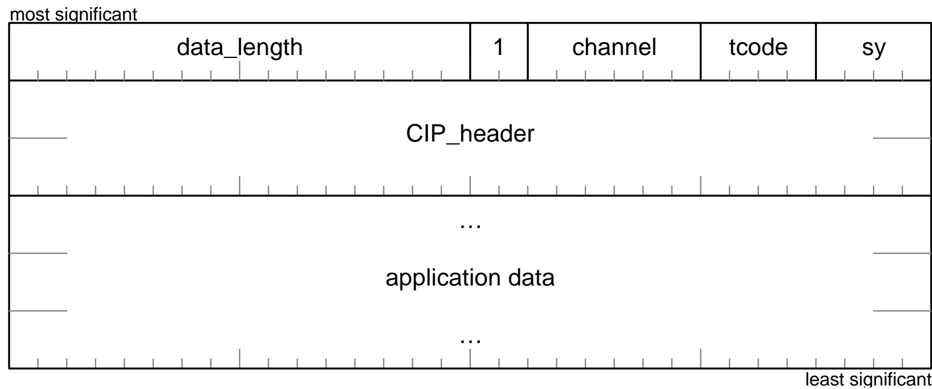

Figure G-1 – Common isochronous packet (CIP) format

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.

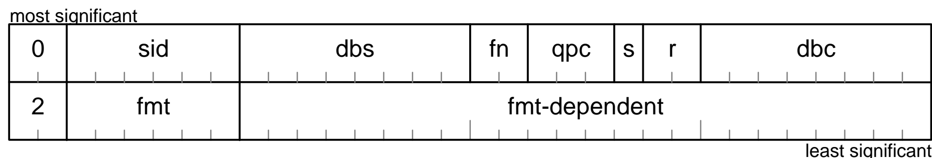

Figure G-2 – Two-quadlet CIP header format

The *sid*, or source ID, field identifies the node whose plug control register(s) 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 block(s) 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 G-2) is one if the isochronous source packet begins with a header quadlet of the format shown below; otherwise, it is zero.

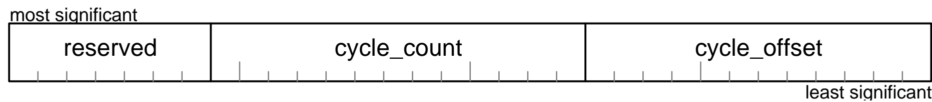

Figure G-3 – Source packet header format

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.

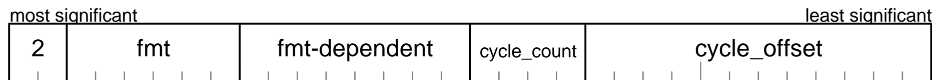

Figure G-4 – Synchronization time (*syt*) format

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 and the *syt* field value is preserved upon playback. 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. When *syt* contains valid *cycle_count* and *cycle_offset* fields, the target transforms these values upon playback, as described in 12.2.3.

Annex H
(informative)

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 illustrates how SBP-3 could 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, Version 2.0, April 22, 1997. These devices present or accept most of their data over Serial Bus isochronous channels; asynchronous requests are used for control information.

AV/C devices differ from the isochronous model presented in 4.7 in two important ways: *a)* there is a single command queue and *b)* isochronous stream control is accomplished by plug control registers (PCRs) and commands contained within the single queue. These differences are illustrative, not only for AV/C devices but also for other, future devices which may use a similar model.

This annex is particular but the intent is general: by way of an AV/C example, it provides guidance for the use of SBP-3 for any device that follows an equivalent, single-queue model.

H.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; a single queue is maintained for commands for all the subunits of an AV/C unit.

H.2 AV/C command frame

An AV/C command frame is a variable-length data structure between four and 512 bytes in length. A normal command block ORB can accommodate 12 bytes of command frame, as illustrated below.

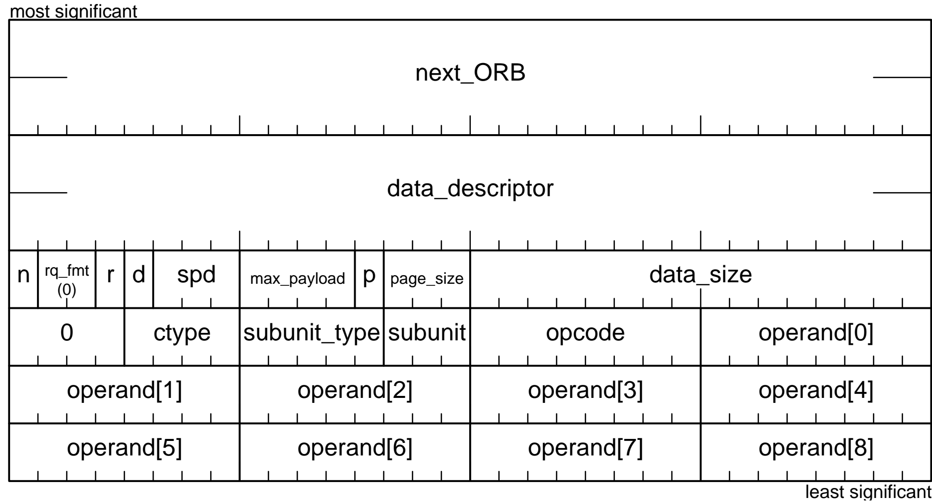

Figure H-1 – AV/C command frame ORB

The *next_ORB*, *data_descriptor*, *rq_fmt*, *spd*, *max_payload*, *page_size* and *data_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 5.1.2.1. The AV/C commands described in the Version 2.0 1394 Trade Association specification do not reference a data buffer; consequently the *data_size* field is always zero in an AV/C command frame ORB.

The *ctype*, *subunit_type*, *subunit*, *opcode* and *operand* fields are specified by AV/C.

H.3 AV/C response frame

An AV/C response frame is a variable-length data structure between four and 512 bytes in length. The status block could hold up to 24 bytes of response frame, but since this exceeds the capacity of the AV/C command ORB the size of the status block is restricted to 20 bytes, as illustrated below.

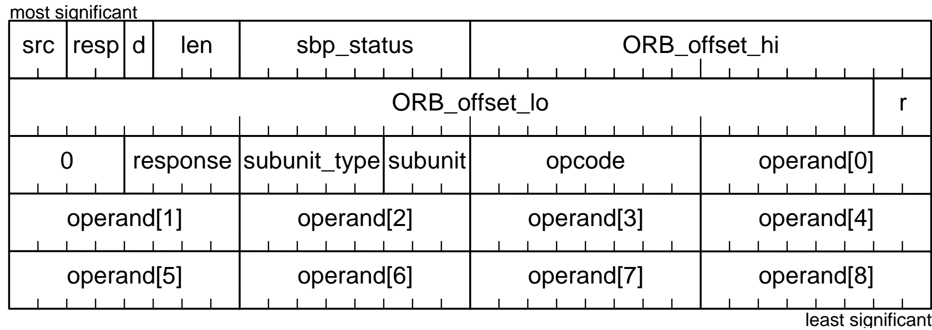

Figure H-2 – Status block format for AV/C response frame

The *src*, *resp*, *sbp_status* fields and the *dead* bit (abbreviated as *d* in the figure above) are as specified in 5.2.3.

The *len* field indicates the size of the status block and has a value of four.

The *ORB_offset_hi* and *ORB_offset_lo* fields together identify the AV/C command frame to which the AV/C response frame pertains. This provides positive command and response matching—an enhanced capability in comparison to FCP.

The *response*, *subunit_type*, *subunit*, *opcode* and *operand* fields are specified by AV/C.

H.4 Configuration ROM

The sample bus information block and root directory for basic targets illustrated in Annex D are equally applicable to targets that use the AV/C command set and are not repeated below. Figure H-3 shows an example of a unit directory and textual descriptor leaves for an AV/C device.

most significant

7	Unit directory CRC (calculated)
12_{16}	specifier_ID (00 $609E_{16}$)
13_{16}	version (01 $05BB_{16}$)
38_{16}	command_set_spec_ID (00 $A02D_{16}$)
39_{16}	command_set (01 0001_{16})
54_{16}	csr_offset (00 4000_{16})
$3A_{16}$	Unit characteristics (00 $0A08_{16}$)
14_{16}	Unit type and number (5F 0000_{16})

least significant

Figure H-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 Version 2.0 1394 Trade Association Specification.

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, executes commands in the order queued and does not support (the dual-queue model of) isochronous operations.

H.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 hold AV/C

command frames and signal these to the target. As the commands complete, the response frames are returned in status blocks addressed to the *status_FIFO* specified by the controller at login.

SBP-3 offers a number of advantages over FCP for the delivery of AV/C command and response frames:

- The target may pace command processing to suit its own requirements and resource availability. Consequently the controller should not expect busy responses and does not have to provide error recovery for busy conditions;
- The correlation between a particular AV/C command frame and its response frame is explicit and unambiguous—even in the case of complex AV/C devices that may be able to queue or process more than one command at a time;
- SBP-3 permits the controller to manage the commands in progress at the target and be certain of the completion status of pending commands if one or more commands are terminated prematurely; and
- The AV/C command frame ORB has descriptor fields for a data buffer that permit the target to use a "pull" model for asynchronous data transfer to or from the controller. The target paces data delivery between itself and the controller; FCP has no such model.

As AV/C command sets are developed for newer and more sophisticated devices, these SBP-3 features may become increasingly important to systems implementers.

Annex I
(informative)

Bibliography

[B1] ANSI NCITS 325-1998, Serial Bus Protocol 2 (SBP-2)

[B2] ANSI NCITS 330-2000, Reduced Block Commands

[B3] ANSI NCITS 333-2000, SCSI Multimedia Commands 2 (MMC-2)

[B4] IEC 61883-1 (1998-02), Consumer audio/video equipment—Digital interface—Part 1: General

[B5] IEEE P1212, Draft Standard for a Control and Status Registers (CSR) Architecture for microcomputer buses

[B6] IEEE P1394.1, Draft Standard for High Performance Serial Bus Bridges

[B7] IEEE P1394.3, Draft Standard for a High Performance Serial Bus Peer-to-Peer Data Transport Protocol (PPDT)

[B8] IEEE Std 1394-1995, Standard for a High Performance Serial Bus

[B9] IEEE Std 1394a-2000, Standard for a High Performance Serial Bus—Amendment 1

[B10] ISO/IEC 9899:1990, Programming Languages—C

[B11] T10 Project 1157D, SCSI Architecture Model 2 (SAM-2)

[B12] T10 Project 1236D, SCSI Primary Commands 2 (SPC-2)

[B13] T10 Project 1363D, SCSI Multimedia Commands 3 (MMC-3)