IEEE Std 1394.1™-2004

IEEE Standards

1394.1™

IEEE Standard for High Performance Serial Bus Bridges

IEEE Computer Society

Sponsored by the
Microprocessor and Microcomputer Standards Committee

IEEE

3 Park Avenue, New York, NY 10016-5997, USA

1 July 2005

Print: SH95311
PDF: SS95311

IEEE Std 1394.1™-2004

Recognized as an

American National Standard (ANSI)

IEEE Standard for High Performance Serial Bus Bridges

Sponsor

Microprocessor and Microcomputer Standards Committee
of the
IEEE Computer Society

Approved 12 April 2005

Amerincan National Standards Institute

Approved 8 December 2004

IEEE-SA Standards Board

Abstract: The model, definition, and behaviors of High Performance Serial Bus bridges, which are devices that can be used to interconnect two separately enumerable buses, are specified.

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

Print: ISBN 0-7381-4647-1 SH95311
PDF: ISBN 0-7381-4648-X SS95311

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

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

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

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

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

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

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

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

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

Introduction

This introduction is not a part of IEEE Std 1394.1-2004, IEEE Standard for High Performance Serial Bus Bridges.

January 15 through 16, 1996, in Dallas, Texas, Gerald Marazas convened a study group authorized by the IEEE Microprocessor Standards Committee (MSC). The study group was chartered to investigate the industry desire for and the feasibility of enhancements to recently approved IEEE Std 1394™-1995; enhancements that would enable the geographic scope of a single Serial Bus to be extended to an interconnected net of multiple buses. Although the informative overview in IEEE Std 1394-1995 describes the use of bus bridges to connect up to 1023 Serial Buses into a single net, the standard is scant on normative details as to how bus bridges might operate.

The study group continued meeting through May 1996, during which time it considered presentations on possible designs for Serial Bus bridges and discussed the Scope and Purpose of a Project Authorization Request (PAR) to be submitted to the IEEE Standards Association. The basic objectives were agreed at the second study group meeting; drafting the PAR was delegated to a small group that completed its work and obtained study group ratification of the PAR in time to submit it to the IEEE-SA for consideration at its June meeting. The IEEE-SA authorized the project, IEEE P1394.1, High Performance Serial Bus Bridges, on June 20, 1996.

The IEEE P1394.1 working group held its first official meeting in July 1996, in San Jose, CA. At this meeting, Richard Scheel was elected Chair of the working group. Scheel shepherded the working group's deliberations until July 2000, when the focus of his work activity at Sony shifted away from IEEE 1394. During Scheel's tenure as Chair, the working group engaged in vigorous debate on topics central to Serial Bus bridges:

- Self-organizing behavior of bridge portals when net topology changes, which necessitates updates to each portal's routing information.
- Distribution and synchronization of CYCLE_TIME.*cycle_offset* from a single cycle master (dubbed the net cycle master) to all buses within the net.
- Knowledge of application-dependent isochronous data formats, such as those specified by the IEC 61883 family of standards, so that bridge portals can modify timestamps embedded in the data.
- Relatively stable 16-bit node IDs (dubbed global node IDs) used to reference remote nodes.
- Detection and elimination of routing loops introduced into the net topology by user actions.
- Connection management for isochronous streams.
- Minimum capabilities for "bridge-aware" devices that communicate with other, remote devices.
- Limited support for legacy devices that are not "bridge-aware."
- Congestion management strategies, *i.e.*, the persistence of bridge portals in attempts to forward request and response subactions to the next bridge portal.
- Assignment of unique bus IDs to distinct Serial Buses within the net.
- Device discovery protocols, both for local devices on the same bus and for remote devices connected to other parts of the net.
- Net management messages for inter-portal communication and, in some cases, communication between "bridge-aware" devices and bridge portals.

In April 2000, at the 1394 Trade Association meeting in Brussels, Dr. Judi Romijn of the Technische Universiteit Eindhoven in the Netherlands made a presentation on the use of formal methods in the discovery of a flaw in the IEEE 1394 PHY state machines that govern bus configuration. This was a fortuitous circumstance, since key members of the IEEE P1394.1 working group were present and engaged her in conversation about the possible application of formal methods to IEEE P1394.1. There was mutual interest, and since then Judi Romijn has been an active and invaluable contributor—particularly with respect to formal proof that the self-organizing behavior of bridge portals (net update) terminates in a consistent state.

Later that summer, at the July 2000 meeting, the working group elected Peter Johansson as Chair. One of the first orders of business was to take stock of the draft standard and determine what, if any, incomplete areas existed that required substantial "invention" before the draft could proceed to Sponsor Ballot. Consensus resulted that although significant effort remained to document the working group's agreements, none of the unresolved issues were major. The working group anticipated a schedule that would yield a functionally complete and reviewed draft standard by the end of 2000, with Sponsor Ballot to follow in the first quarter of 2001. Work proceeded apace on the draft and by the end of March 2001, the penultimate draft before Sponsor Ballot was published for final review by the working group. A modest number of changes were made in the final draft, published in June, and Sponsor Ballot commenced the same month.

After collation of more than 500 ballot comments (the substance and volume of which revealed as overly optimistic the earlier working group consensus with respect to "no major issues") and formation of the Ballot Response Committee (BRC), the Editor spent several months resolving the less controversial editorial and technical comments before publishing an interim draft in December 2001. The BRC held its first meeting in San Jose at the beginning of December 2001, and continued to meet the subsequent year until agreement in principle on the resolution of all comments was achieved in October 2002.

Because of a change in management support, as well as commitment to other projects, the Editor was unable to publish a candidate draft for Recirculation Ballot until November 2003. The BRC met in early December for final review and corrections to the draft before a Recirculation Ballot was initiated in March 2004.

Notice to users

Errata

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

Interpretations

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

Patent notice

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying all patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention. A patent holder has filed a statement of assurance that it will grant a license under these rights without compensation or under reasonable rates and nondiscriminatory, reasonable terms and conditions to all applicants desiring to obtain such licenses. The IEEE makes no representation as to the reasonableness of rates and/or terms and conditions of the license agreements offered by patent holders. Further information may be obtained from the IEEE Standards Department.

Participants

The following is a list of active participants in the IEEE P1394.1 working group (those who attended three or more meetings from inception to the time of publication):

Peter Johansson, *Chair and Editor*
David Hunter, *Secretary*

Jean-Paul Accarie
Masa Akahane
Subrata Bannerjee
Steven Bard
Philippe Boucachard
Mohamed Braneci
Richard Churchill
Beth Cooper
Chris Dorsey
Firooz Farhoomand
Steve Finch
Laurent Frouin
John Fuller
Eric Hannah
Jerry Hauck
Dieter Haupt
Hisaki Hiraiwa
Daisuke Hiraoka
Du Hung Hou
David James
Mark Knecht
David LaFollette
Yvon Legallais
Jun-ichi Matsuda
Daniel Meirsman
Neil Morrow
Atsushi Nakamura
Yoshikatsu Niwa
Fritz Nordby
Takayuki Nyu
Ozay Oktay
Tomoki Saito
Takashi Sato
Tetsuya Sato
Bradley Saunders
Yoshi Sawada
Richard Scheel
Hisato Shima
Michael Smith
David Smith
Carlton Sparrell
Gilles Straub
Thomas Thaler
Kazonobu Toguchi
Satoru Toguchi
Masatoshi Ueno
Colin Whitby-Strevens
Calto Wong
David Wooten
Patrick Yu
Dave Zalatimo
Frank Zhao

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

Jean-Paul Accarie
Stelios Akalestos
Eric Anderson
Larry Arnett
Terry Arnold
Mohamed Braneci
Keith Chow
Elizabeth Cooper
Guru Dutt Dhingra
Georg Dickmann
Sourav Dutta
Roger D. Edwards
Firooz Farhoomand
Michael Fischer
Gordon Force Sr
Laurent Frouin
John Fuller
Straub Gilles
Dieter Haupt
Neil Horman
David Hunter
David James
Peter Johansson
Mark Knecht
Robert Mortonson
Chuck Rice
Gary Robinson
Bivabasu Sarkar
Takashi Sato
Bradley Saunders
Thomas Schaal
Richard Scheel
Akihiro Shimura
Michael Teener
Thomas Thaler
David Thompson
Kazunobu Toguchi
Robert Tripi
Colin Whitby-Strevens
David Wooten
Paul Work
Don Wright
Patrick Yu
Oren Yuen
Janusz Zalewski

The following people served on the ballot response committee:

Peter Johansson, *Chair*

Mohamed Braneci
Georg Dickmann
John Fuller
David Hunter
Judi Romijn
Takashi Sato
Gilles Straub
Eldad Teeni
Thomas Thaler
Kazunobu Toguchi
Colin Whitby-Strevens
David Wooten

This standard was approved by the IEEE-SA Standards Board on 8 December 2004 with the following membership:

Don Wright, *Chair*
Steve M. Mills, *Vice Chair*
Judith Gorman, *Secretary*

Chuck Adams
Stephen Berger
Mark D. Bowman
Joseph A. Bruder
Bob Davis
Roberto de Marca Boisson
Julian Forster*
Arnold M. Greenspan
Mark S. Halpin
Raymond Hapeman
Richard J. Holleman
Richard H. Hulett
Lowell G. Johnson
Joseph L. Koepfinger*
Hermann Koch
Thomas J. McGean
Daleep C. Mohla
Paul Nikolich
T. W. Olsen
Ronald C. Petersen
Gary S. Robinson
Frank Stone
Malcolm V. Thaden
Doug Topping
Joe D. Watson

*Member Emeritus

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

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

Michelle Turner
IEEE Standards Project Editor

Contents

IEEE Standard for High Performance Serial Bus Bridges

1. Overview

1.1 Scope

This is a full-use standard whose scope is to extend the already defined asynchronous and isochronous services of High Performance Serial Bus beyond the local bus by means of a device, the bridge, which consists of two nodes, each connected to a separate bus and both interconnected by implementation-dependent means.

The project is intended to standardize the model, definition, and behaviors of High Performance Serial Bus bridges, which are devices that may be used to interconnect two separately enumerable buses. This project extends IEEE Std 1394™-1995[1], as amended by IEEE Std 1394a™-2000, and IEEE Std 1394b™-2002, and is based upon those documents as well as upon IEEE Std 1212™-2001, Control and Status Registers (CSR) Architecture for microcomputer buses.

NOTE—The set of IEEE 1394 standards specifies the interfaces, functions, and operations necessary to ensure interoperability between conforming implementations. These standards are functional descriptions. An implementation may employ any design whose behavior is compliant with the pertinent standards.[2]

1.2 Purpose

IEEE Std 1394-1995, as amended by IEEE Std 1394a-2000 and IEEE Std 1394b-2002, is a cost-effective desktop interconnect for both computer peripherals and consumer electronics. However, the use of High Performance Serial Bus in other environments, *e.g.*, an interconnect to carry high-speed digital video data between rooms of a house, is hampered by the incomplete architectural and protocol specifications for bridges in the existing standards. This project proposes to adequately specify bridge requirements in order to enable a larger consumer and computer market for High Performance Serial Bus products.

[1]Information on references can be found in Clause 2.

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

2. Normative references

The standards named in this section contain provisions that, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision; parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid international standards.

IEC 61883-1 (2003-01), Consumer audio/video equipment—Digital interface—Part 1: General.[2]

IEEE Std 1212™-2001, IEEE Standard for Control and Status Registers (CSR) Architecture for microcomputer buses.[3, 4]

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

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

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

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

[2] IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch/). IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA. (http://www.ansi.org/)

[3] The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[4] IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

3. Definitions and notation

For the purposes of this standard, the following definitions, terms and notational conventions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B2][5], should be consulted for terms not defined in this clause.

3.1 Conformance

The following keywords are used to differentiate between different levels of requirements and optionality in this standard:

3.1.1 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.

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

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

3.1.4 reserved: A keyword used to describe objects—bits, bytes, quadlets, octlets and fields—or the code values assigned to these objects; the object or the code value is set aside for future standardization by the IEEE. A reserved object shall be zeroed by its originator or, upon development of a future IEEE 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 whose code values are defined by this standard shall inspect its value and reject reserved code values.

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

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

3.2 Technical

The following are terms that are used within this standard:

3.2.1 adjustable cycle master: A cycle master that recognizes cycle master adjustment packets and, according to the instruction received, lengthens or shortens the interval to the next cycle start event by a single cycle timer tick.

3.2.2 allocation map: A data structure that specifies for each of the 1023 possible bus IDs whether it is in use (VALID), not in use (CLEAN) or in a transitional state from in use to not in use (DIRTY). An allocation map is similar to a bridge portal’s route map except that it contains less information: the route map states FORWARD and VALID are collapsed into a single state, VALID.

3.2.3 alpha portal: A portal on a bus that snoops and forwards transaction subactions addressed to the prime portal. Except during net update, there is only one alpha portal on a bus. On the bus to which the prime portal is connected, the alpha portal and the prime portal are one and the same.

3.2.4 bridge: A device capable of connecting two buses to form a net. A bridge implements two portals, forwards asynchronous subactions and might forward isochronous subactions, both as determined by the route information it maintains. A distributed algorithm executed by bridge portals initializes asynchronous routing information; applications initialize, on a stream-by-stream basis, isochronous routing information.

3.2.5 bridge-aware: A device capable of originating and timing transaction requests addressed to remote nodes. Bridge-aware devices also interact with bridges by means of messages or other signals.

[5]The numbers in brackets correspond to those of the bibliography in Annex I.

3.2.6 bridge-bound: A description applied to a request, response, or stream subaction received by a bridge portal with the intent to forward the subaction to its co-portal. The subaction is bridge-bound in that its next (intermediate) destination is the internal fabric of the bridge that connects the two portals. A portal that snoops a bridge-bound subaction is called an entry portal.

3.2.7 bus: A group of Serial Bus nodes interconnected within the same arbitration domain and mutually addressable by packets with a *destination_bus_ID* field of $3FF_{16}$.

3.2.8 bus ID: A 10-bit identifier that, unless net update is in progress, is unique for each bus within a Serial Bus net. The bus ID assigned to a particular bus is not visible in the most significant ten bits of the NODE_IDS register, which always exhibits a *bus_ID* value of $3FF_{16}$, the local bus. Bridge portals maintain the assigned bus ID internally and use it to transform local node IDs into global node IDs and vice-versa.

3.2.9 bus-bound: A description applied to a request, response, or stream subaction received by a bridge portal from its co-portal with the intent to transmit the subaction on the portal's local bus. A portal that transmits a bus-bound subaction is called an exit portal.

3.2.10 channel: A relationship between a group of nodes: zero or one talker and zero or more listeners. A number between zero and 63, inclusive, identifies the group. Channel numbers are allocated cooperatively through isochronous resource management facilities.

3.2.11 clan: A group of affiliated bridge portals; all of the members of a clan exhibit allegiance to the same prime portal, which is identified by its EUI-64. When two or more nets are joined, it is possible for bridge portals on a bus to belong to different clans—but this is a temporary condition that is resolved by net update procedures.

3.2.12 clan allocation map: An allocation map derived from the route maps of bridge portals that belong to the same clan (*see:* **net allocation map**).

3.2.13 coordinator: A bridge portal selected after each bus reset; it manages net update, which might be required as a consequence of local or remote topology changes in the net. The coordinator is the bridge portal with the largest physical ID; it retains this role until a subsequent bus reset. The coordinator is also responsible to maintain virtual ID mappings and distribute them to all bridge portals on the bus.

3.2.14 co-portal: From the perspective of a particular bridge portal, the other portal. A portal is connected to its co-portal by the bridge's internal fabric.

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

3.2.16 cycle master: On a particular bus, the node that generates the periodic cycle start packet 8000 times a second. In a net of interconnected buses there is a cycle master for each bus.

3.2.17 downstream: An adjective used to describe the relationship of a node to a particular location within the net. When used in the context of cycle time synchronization with respect to the net cycle master, a downstream cycle master or a downstream portal has more bridge portals between itself and the net cycle master than the portal to which it is compared. Alternately, in the context of a particular isochronous stream, within a bridge the downstream portal is the one with more bridge portals between itself and the talker.

3.2.18 entry portal: A description applied to a bridge portal when it snoops and assumes responsibility for the disposition of a bridge-bound request, response or stream subaction. Commonplace actions are to forward the subaction to the co-portal (which acts as the exit portal) or to echo the subaction on the local bus, but an entry portal may also discard a subaction or create and transmit a response packet if errors are detected.

3.2.19 exit portal: A description applied to a bridge portal when it transmits bus-bound request, response or stream subactions forwarded by the entry portal. The same portal might be both the entry and exit portal for a subaction; this is the case when a transaction request or response is echoed to the local bus.

3.2.20 GASP: Global asynchronous stream packet, specified by IEEE Std 1394a-2000.

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

3.2.22 global subaction: Either a) an asynchronous subaction either of whose *source_ID* and *destination_ID* contains a global node ID or b) a GASP subaction whose *sy* value is greater than or equal to eight.

3.2.23 initial entry portal: The first bridge portal that snoops request, response or stream subaction and transforms the subaction's *source_ID*, if present, from a local node ID to a global node ID.

3.2.24 heterogeneous bridge: A bridge, one portal of which is an IEEE 1394 node but whose other portal implements a different transport protocol, *i.e.*, the portal is not an IEEE 1394 node. The operational details of heterogeneous bridges are beyond the scope of this standard.

3.2.25 isochronous bridge: A bridge that implements the facilities (buffers, CSRs, stream management messages, *etc*.) necessary to transfer isochronous subactions between its portals. These capabilities are in addition to the base requirements for a bridge to forward asynchronous subactions and to maintain cycle time synchronization.

3.2.26 isochronous period: A period that begins after a cycle start packet is sent and ends when a subaction gap is detected. During an isochronous period, only isochronous subactions may occur. An isochronous period nominally begins every 125 µs.

3.2.27 isochronous resource manager: A node that implements the BUS_MANAGER_ID, BANDWIDTH_AVAILABLE, CHANNELS_AVAILABLE and BROADCAST_CHANNEL registers (some of which permit the cooperative allocation of isochronous resources). Subsequent to each bus reset, one isochronous resource manager is selected from all nodes capable of this function. All bridge portals are required to be isochronous resource manager-capable.

3.2.28 isochronous subaction: Within the isochronous period, either a concatenated packet or a packet and the gap that preceded it.

3.2.29 listener: A node, or an application at a node, that receives a stream packet.

3.2.30 local ID: The least significant six bits of the NODE_IDS register; each node on a local bus is assigned a different local ID as a consequence of bus reset. Also known as physical ID.

3.2.31 local node: A Serial Bus node is local with respect to another node if they are both connected to the same bus. This is true whether the bus does not yet have a unique *bus_ID* and is addressable only as the local bus, $3FF_{16}$, or if the bus has been assigned a *bus_ID*.

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

3.2.33 local subaction: Any of a) an asynchronous subaction whose *source_ID* and *destination_ID* both contain a local node ID, b) an asynchronous stream subaction that is not GASP, c) a GASP subaction whose *sy* value is less than eight or d) an isochronous subaction.

3.2.34 maximum forward time: The maximum time a bridge is permitted to persist in its attempts to forward a snooped asynchronous subaction to the next recipient, either an intermediate bridge portal on the route to the node identified by *destination_ID* or the destination node itself. The period to be timed commences when the entry portal transmits *ack_pending* or *ack_complete* for a snooped subaction and concludes when the exit portal receives *ack_pending* or *ack_complete* for the retransmitted subaction.

3.2.35 mute bridge: A bridge whose routing functions are disabled; neither asynchronous nor isochronous subactions are forwarded from either co-portal to the other. Each of the mute bridge's portals continues to respond to request subactions addressed to its local node ID and may initiate local or remote request subactions.

3.2.36 net: A collection of buses interconnected by bridges. Each bus within the net is uniquely identified by its *bus_ID*.

3.2.37 net allocation map: An allocation map derived from all of a net's clan allocation maps (see also clan allocation map). When network topology is stable, there is only one clan allocation map, which is identical to the net allocation map.

3.2.38 net cycle master: The cycle master for one Serial Bus in the net selected to be the cycle time source for the entire net.

3.2.39 net update: A process triggered by the addition or removal of one or more buses (or bus IDs) from a net; net topology and all bridge portal route maps are updated to a consistent state.

3.2.40 node: A Serial Bus device that can be addressed independently of other nodes. A minimal node consists of only a PHY without an enabled link. If the link and other layers are present and enabled, they are considered part of the node.

3.2.41 node ID: A 16-bit number that uniquely identifies a node, either within the context of a particular bus (local node ID) or within a net of interconnected buses (global node ID). The ten most significant bits of node ID, the bus ID, identify the bus to which the node is connected. The six least significant bits of node ID identify the node on that bus; within a local node ID they are a physical ID but within a global node ID they are a virtual ID.

3.2.42 physical ID: Synonymous with local ID, a 6-bit address assigned to each node on a bus by the self-identification process that follows a bus reset (see IEEE 1394).

3.2.43 portal: A connection from a Serial Bus bridge to a bus. Each portal presents a set of Serial Bus CSRs, as defined by IEEE 1394 and this document, to the connected bus. There may be multiple PHY ports for each portal. Serial Bus bridges shall implement two portals.

3.2.44 prime portal: The singular portal within a net of interconnected buses that enumerates unique bus IDs and manages their distribution. When two or more nets are joined, one prime portal is selected from the incumbents. Since the location of the prime portal also determines the location of the net cycle master, the selection process is designed to minimize disruption to isochronous streams (which rely upon the stability of the net cycle master). Ties in the selection process between prime portal candidates are resolved by their EUI-64s.

3.2.45 quarantine: A period of time during which a bridge-aware node or bridge-portal shall not originate global subactions. Quarantine commences whenever the self-ID packets that occur subsequent to a bus reset indicate net topology changes; it ends upon a signal from the coordinator or the alpha portal. Multiple bus resets can occur during quarantine.

3.2.46 reallocation proxy: A bridge portal responsible to reallocate channel numbers and bandwidth after a bus reset. The reallocation proxy also keeps track of listeners on the local bus; if the count of listeners drops to zero, either as a result of explicit connection teardown or because of node removal, the reallocation proxy releases the pertinent channel number and bandwidth.

3.2.47 remote node: A Serial Bus node is remote with respect to another node if the nodes are connected to buses that have differing bus IDs or if one or more Serial Bus bridges lie on the path between the two nodes. Nodes connected to the same bus are considered remote with respect to each other if global node IDs are used in request and response subactions exchanged between them.

3.2.48 remote transaction-capable: A transaction-capable Serial Bus node that is additionally capable of initiating transaction requests directed to a remote node.

3.2.49 route map: A data structure, whose contents are different for each portal, that specifies, for each of the 1023 possible bus IDs, whether transaction subactions are forwarded by the portal. The four possible states for a bus ID are: in use but not

forwarded by the portal (VALID), in use and forwarded by the portal (FORWARD), not in use and therefore not forwarded by any portal in the same clan (CLEAN), or a transitional state (DIRTY).

3.2.50 snarf: Jargon that came into use in the Unix community in the 1960s. The following definition is obtained from the Jargon File 4.0.0: "To acquire, with little concern for legal forms or politesse (but not quite by stealing)." In the context of this standard, snarf refers to the interception of some asynchronous subactions while en route to their *destination_ID*. The snarfed packet might or might not be retransmitted by the intercepting bridge.

3.2.51 stream: Either asynchronous or isochronous data originated by a talker and received by zero or more listeners. An isochronous stream is uniquely identified by the talker's EUI-64 and an index locally unique at the talker. An isochronous stream's parameters include the payload, arbitration overhead and speed.

3.2.52 stream controller: A node that uses net management messages to set up or tear down routing for an isochronous stream between a talker and one or more listeners.

3.2.53 subordinate portal: A portal that is neither an alpha portal nor the prime portal; there may be multiple subordinate portals on a bus.

3.2.54 terminal acknowledgement: An acknowledge packet other than *ack_pending*, *ack_busy_X*, *ack_busy_A* or *ack_busy_B*. If received in connection with a request subaction, the acknowledgement completes the transaction; no response subaction will follow.

3.2.55 terminal exit portal: The last bridge portal that transmits (or would transmit) a transaction request or response subaction en route to its destination; the portal that normally transforms the subaction's *destination_ID* from a global node ID to a local node ID.

3.2.56 upstream: An adjective used to describe the relationship of a node to a particular location within the net. When used in the context of cycle time synchronization with respect to the net cycle master, an upstream cycle master or an upstream portal has fewer bridge portals between itself and the net cycle master than the portal to which it is compared. Alternately, in the context of a particular isochronous stream, within a bridge the upstream portal is the one with fewer bridge portals between itself and the talker.

3.2.57 virtual ID: A 6-bit address assigned by the coordinator to all of the nodes present on the portal's local bus. For any bus, all the bridge portals share the same mapping from 6-bit physical ID to virtual ID.

3.3 Document notation

3.3.1 Size notation

This document avoids the terms *word, half-word,* and *double-word,* which have widely different definitions depending on the word size of the processor. In their place, processor-independent terms established by previous IEEE bus standards are used. These terms are illustrated in Table 3-1.

Table 3-1 — Size notation examples

Size (in bits)	IEEE standard notation (used in this standard)	Pseudocode type
4	nibble	
8	byte	BYTE
16	doublet	DOUBLET
32	quadlet	QUADLET
64	octlet	OCTLET

Serial Bus uses big-endian ordering for byte addresses within a quadlet and quadlet addresses within an octlet. For 32-bit quadlet registers, byte 0 is always the most significant byte of the register. For a 64-bit quadlet-register pair, the first quadlet is always the most significant. The field on the left (most significant) is transmitted first; within a field the most significant (leftmost) bit is also transmitted first. This ordering convention is illustrated in Figure 3-1.

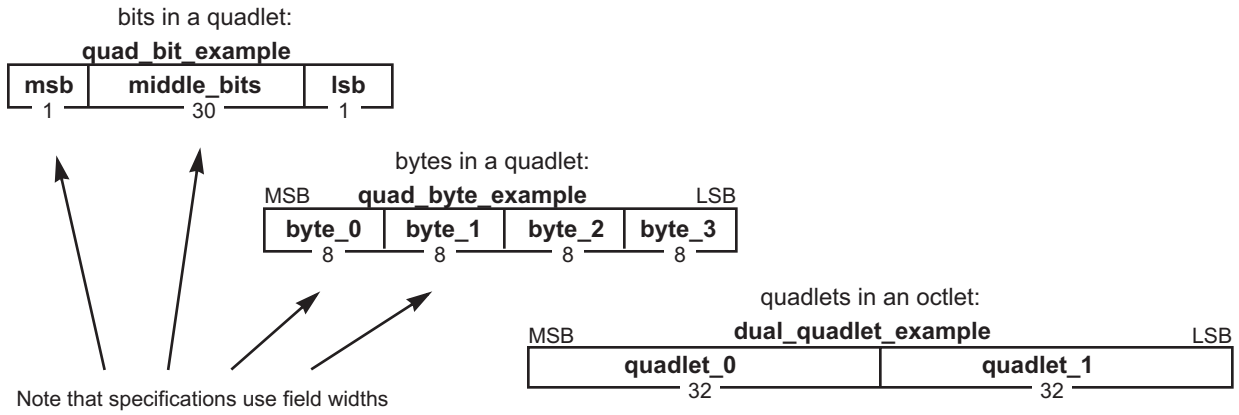


Figure 3-1 — Bit and byte ordering

Although Serial Bus addresses are defined as big-endian, their data values can also be processed by little-endian processors. To minimize the confusion between conflicting notations, the location and size of bit fields are usually specified by width, rather than their absolute positions, as is also illustrated in Figure 3-1.

When specific bit fields must be used, the CSR Architecture convention of consistent big-endian numbering is used. Hence, the most significant bit of a quadlet (“msb” in figure 1-2) will be labeled “quad_bit_example[0],” the most significant byte of a quadlet (“byte_0”) will be labeled “quad_byte_example[0:7],” and the most significant quadlet in an octlet (“quadlet_high”) will be labeled “dual_quadlet_example[0:31].”

The most significant bit shall be transmitted first for all fields and values defined by this standard, including the data values read or written to control and status registers (CSRs).

3.3.2 Numerical values

Decimal, hexadecimal, and binary numbers are used within this document. For clarity, the decimal numbers are generally used to represent counts, hexadecimal numbers are used to represent addresses, and binary numbers are used to describe bit patterns within binary fields.

Decimal numbers are represented in their standard 0, 1, 2, ... format. Hexadecimal numbers are represented by a string of one or more hexadecimal (0-9, A-F) digits followed by the subscript 16. Binary numbers are represented by a string of one or more binary (0 or 1) digits, followed by the subscript 2. Thus the decimal number “26” can also be represented as “$1A_{16}$” or “11010_2”. In C pseudocode examples, hexadecimal numbers have a “`0x`” prefix and binary numbers have a “`0b`” prefix, so the decimal number “26” would be represented by “`0x1A`” or “`0b11010`.”

3.3.3 Packet formats

Most Serial Bus packets consist of a sequence of quadlets. Packet formats are shown using the style given in Figure 3-2.

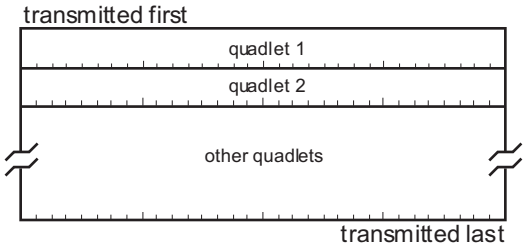


Figure 3-2 — Example packet format

Fields appear in packet formats with their correct position and widths. Bits in a packet are transmitted starting with the upper leftmost bit and finishing with the bottom rightmost bit. Given the rules in 3.3.1, this means that all fields defined in this standard are sent most significant bit first.

3.3.4 C pseudocode notation

This standard normatively describes the operations of a bridge by means of C language code and expository text. The C pseudocode is not a normative description of an implementation; it is a normative description of externally observable bridge behaviors. Different implementations are possible. In case of conflict between the C pseudocode and other text, precedence shall be given first to the C pseudocode and second to the expository text.

Although familiar to software engineers, C pseudocode operators are not necessarily obvious to all readers. The meanings of C pseudocode operators, arithmetic, relational logical and bitwise, both unary and binary, are summarized in Table 3-2.

Table 3-2 — C pseudocode operators summary

Operator	Description
+, -, * and /	Arithmetic operators for addition, subtraction, multiplication and integer division
%	Modulus; x % y produces the remainder when x is divided by y
>, >=, < and <=	Relational operators for greater than, greater than or equal, less than and less than or equal
== and !=	Relational operators for equal and not equal; the assignment operator, =, should not be confused with ==
++	Increment; i++ increments the value of the operand after it is used in the expression while ++i increments it before it is used in the expression
--	Decrement; post-decrement, i--, and pre-decrement, --i, are permitted
&&	Logical AND
||	Logical OR
!	Unary negation; converts a nonzero operand into 0 and a zero operand into 1
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR
<<	Left shift; x << 2 shifts the value of x left by two bit positions and fills the vacated positions with zero

Table 3-2 — C pseudocode operators summary *(continued)*

Operator	Description
>>	Right shift; vacated bit positions are filled with zero or one according to the data type of the operand but in this amendment are always filled with zero
~	One's complement (unary)

A common construction in C is conditional evaluation, in the form `(expr) ? expr1 : expr2`. This indicates that if the logical expression `expr` evaluates to a nonzero value then `expr1` is evaluated, otherwise `expr2` is evaluated. For example, `x = (q > 5) ? x + 1 : 14` first evaluates `q > 5`. If nonzero (TRUE), `x` is incremented; otherwise `x` is assigned the value 14.

The descriptions above are casual; if in doubt, the reader is encouraged to consult ISO/IEC 9899:1990 [B3].

The C pseudocode examples assume the data types listed in Table 3-3 are defined.

Table 3-3 — Additional C data types

Data type	Description
timer	A real number, in units of seconds, that autonomously increments at a defined rate
Boolean	A single bit, where 0 encodes FALSE and 1 encodes TRUE

All C pseudocode is to be interpreted as if it could be executed instantaneously, although time can elapse when conditional expressions are evaluated as part of iterated C pseudocode. Time elapses unconditionally only when the following function is called:

```
void wait_time(float time);// Wait for time, in seconds, to elapse
```

3.3.5 CSR, ROM, and field notation

This standard describes CSRs and fields within them. All CSRs are documented in the style used by IEEE Std 1212-2001. To distinguish register and field names from node states or descriptive text, the register name is always capitalized. For example, the notation STATE_CLEAR.*lost* is used to identify the *lost* bit within the STATE_CLEAR register.

All CSRs are one or more contiguous quadlets that are quadlet aligned. The address of a register is specified as the byte offset from the beginning of the register space and is always a multiple of four. When a range of register addresses is described, the ending address is the address of the register's last quadlet.

This standard describes configuration ROM entries and fields within these entries. To distinguish ROM entry and field names from node states or descriptive text, the first character of the entry name is always capitalized. Thus, the notation Bus_Info_Block.*cmc* is used to describe the *cmc* bit within the Bus_Info_Block entry.

Other data structures (such as packets, timers, and counters) are shown in lowercase and formatted in a fixed-pitch typeface.

3.3.6 Register specification format

This document defines the format and function of Serial Bus-specific CSRs. Registers can be read only, write only, or both readable and writable. The same distinctions may apply to any field within a register. A CSR specification includes the format (the sizes and names of bit field locations), the initial value of the register, the value returned when the register is read, and the effects when the register is written. An example register is illustrated in Figure 3-3.

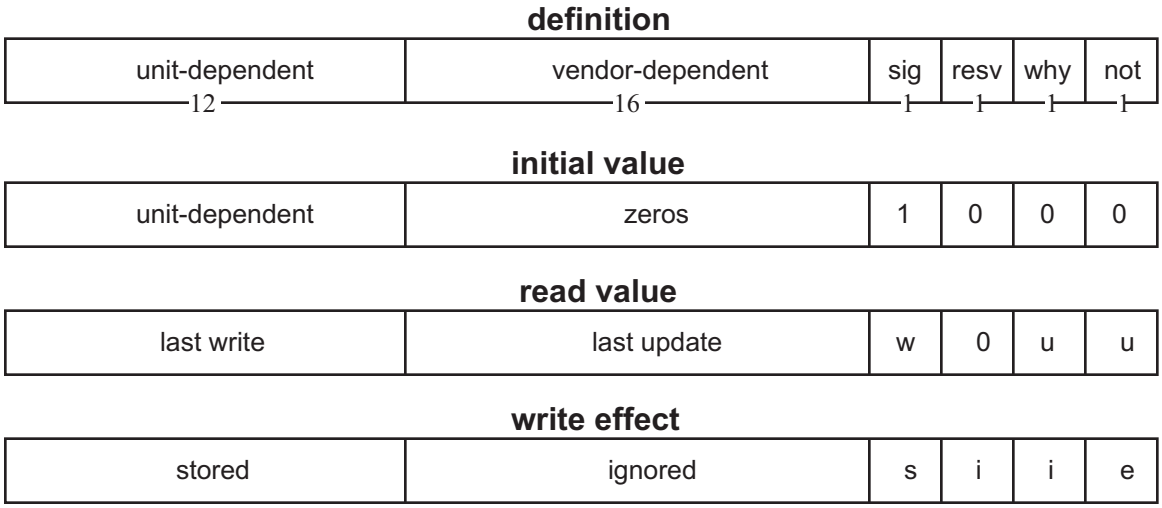


Figure 3-3 — CSR format specification (example)

The register definition lists the names of register fields. These names are descriptive, but the fields are defined in the text; their function should not be inferred solely from their names. However, the register definition fields in Figure 3-3 have the meanings specified by Table 3-4.

Table 3-4 — Register definition and initial values fields

Name	Abbreviation	Definition
unit-dependent	u	The meaning of this field shall be defined by the pertinent unit architecture.
vendor-dependent	v	When used to name a field, the meaning of the field shall be defined by the vendor. When used to specify an initial value, the value shall be determined by the vendor but might be subject to constraints imposed by this or other standards. Within a unit architecture, the unit-dependent fields may be defined to be vendor-dependent.

A node's CSRs shall be initialized when power is restored (power reset) and might be initialized when a bus reset occurs or a quadlet is written to the node's RESET_START register (command reset). If a CSRs bus reset or command reset values differ from its initial values, they shall be explicitly specified.

The read value fields in Figure 3-3 have the meanings specified by Table 3-5.

Table 3-5 — Read value fields

Name	Abbreviation	Definition
last write	w	The value of the data field shall be the value that was previously written to the same register address.
last update	u	The value of the data field shall be the last value that was updated by node hardware.

The write-effect fields in Figure 3-3 have the meanings specified by Table 3-6.

Table 3-6 — Write value fields

Name	Abbreviation	Definition
stored	s	The value of the written data field shall be immediately visible to reads of the same register.
ignored	i	The value of the written data field shall be ignored; it shall have no effect on the state of the node.
effect	e	The value of the written data field shall have an effect on the state of the node, but might not be immediately visible to reads of the same register.

3.3.7 Reserved CSR fields

Reserved fields within a CSR conform to the requirements of the conformance glossary in this standard (see 3.1). Within a CSR, such a field is labeled *reserved* (sometimes abbreviated as lowercase *r* or *resv*). Reserved fields behave as specified by Figure 3-4: they shall be zero and any attempt to write to them shall be ignored.

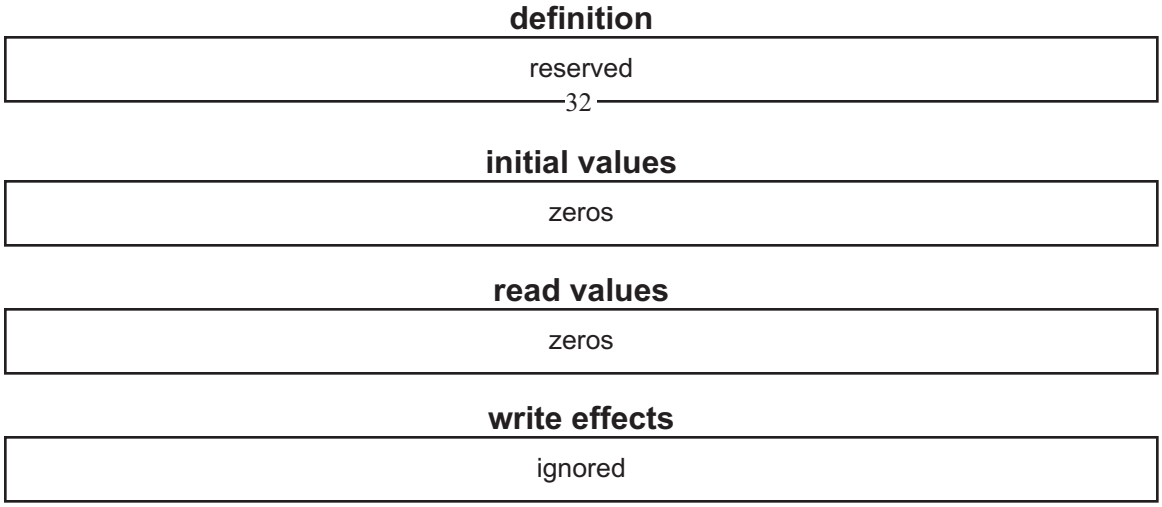


Figure 3-4 — Reserved CSR field behavior

This is straightforward as it applies to read and write requests. The same rules apply to lock requests, but the behaviors are less obvious; see IEEE Std 1394a-2000 for details.

4. Bridge model (informative)

A Serial Bus bridge consists of two ***bridge portals*** (each with its associated PHY and link), ***queues*** (FIFOs) for asynchronous and isochronous subactions (which collectively form an implementation-dependent ***fabric*** between the two portals), ***cycle timers***, ***route maps,*** and ***configuration ROM***. Figure 4-1 illustrates this model.

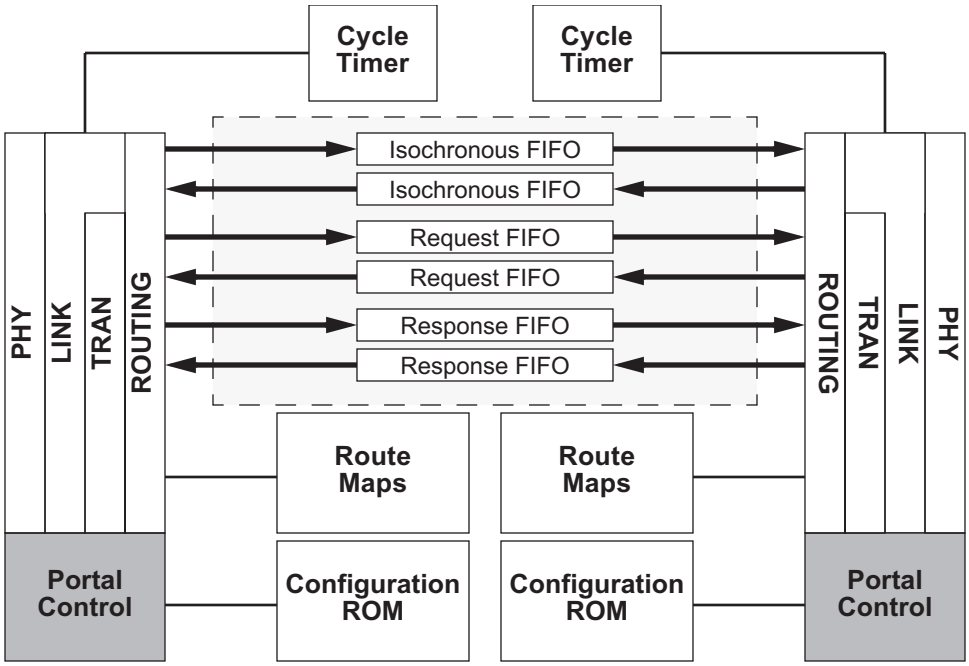


Figure 4-1 — Bridge model

Each bridge portal is a separate Serial Bus node, with a different EUI-64 and its own address space on the bus to which it is connected. A bridge portal responds to Serial Bus read, write, and lock requests from its connected bus as described in this standard. A bridge portal also monitors all Serial Bus subactions, asynchronous and isochronous, and uses route maps to determine which subactions, if any, are to be routed through the bridge's fabric to the co-portal.

The bridge portals are interconnected by means of a fabric that is capable of transferring any Serial Bus subaction from one portal to its co-portal. The fabric is conceptualized as a set of FIFOs (shown enclosed with dashed lines in Figure 4-1) that support bidirectional, non-blocking transfer of asynchronous request subactions, asynchronous response subactions, and isochronous subactions. Although this standard mandates some behavior for these queues, the details of the fabric implementation are not addressed by this standard. The fabric could be modest in geographical extent, as when both of the bridge portals and fabric are located within a single enclosure. Conversely, the fabric could be physically extensive, as could be the case if a bridge's portals were located in separate rooms. In both cases, the model remains the same.

Each portal has a cycle timer driven by a 24.576 MHz oscillator. The propagation of cycle time from one bridge portal to the other is implementation-dependent and beyond the scope of this standard.

The route maps contain information that permits the selective forwarding of both asynchronous and isochronous subactions through the fabric to the co-portal.

4.1 Global node IDs

A global node ID is a 16-bit object that consists of two parts, a 10-bit bus ID and a 6-bit virtual ID; the latter is analogous to the physical ID specified by IEEE 1394, but is not assigned by the self-ID process and does not necessarily change upon each bus reset. A global node ID may be present as a destination or source address in an asynchronous subaction—so long as at least one bridge portal is on the path between the source and the destination nodes.[6] Global node IDs supplement local node IDs, but they do not replace them.

The usage of global node IDs and the role that bridge portals play in the conversion from local to global node IDs and vice versa can be understood in the context of the simple net topology illustrated by Figure 4-2. Bridges are shown as circles with a line to divide each portal from the other, while bus IDs are shown above the lines that represent different arbitration domains (the buses themselves). The requester and responder nodes are squares with the designation Rq and Rsp, respectively. Other nodes that might be present on a bus are omitted from the diagram.

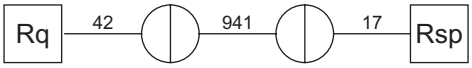


Figure 4-2 — Simple net topology (reference model)

When Rq originates a remote request subaction intended for Rsp, the *destination_ID* field routes the subaction toward bus 17. The initial entry portal (the bridge portal connected to bus 42) transforms the *source_ID* field from the local node ID transmitted by Rq to the corresponding global node ID. When the subaction is transmitted on bus 941, no transformation of either *source_ID* or *destination_ID* is necessary; both of these fields contain global node IDs. Once the request subaction arrives at the destination bus, the terminal exit portal (the bridge portal connected to bus 17) transforms the *destination_ID* field from a global node ID to a local node ID recognizable by Rsp. If Rsp responds, the global node ID present in the request subaction's *source_ID* field is usable as the destination for the response subaction—and the process just described operates in reverse as the response subaction is transmitted to the requester's bus. The table below summarizes the observable values for the subaction's *destination_ID* and *source_ID* fields on each of the three buses.

Subaction	Bus	*destination_ID*		*source_ID*	
		Upper 10 bits	**Lower 6 bits**	**Upper 10 bits**	**Lower 6 bits**
Request	42	17	Virtual ID	$3FF_{16}$	Physical ID
	941	17	Virtual ID	42	Virtual ID
	17	$3FF_{16}$	Physical ID	42	Virtual ID
Response	17	42	Virtual ID	$3FF_{16}$	Physical ID
	941	42	Virtual ID	17	Virtual ID
	42	$3FF_{16}$	Physical ID	17	Virtual ID

The features of global node IDs that make them useful in a bridged environment are as follows:

— At any point in time, a one-to-one correlation exists between a node's EUI-64 and its global node ID.
— A node's virtual ID does not change solely because its physical ID changes after a bus reset.
— On each bus, one and only one bridge portal (the coordinator) is responsible to synchronize virtual ID assignments to the local nodes so that all bridge portals have identical mappings.

[6]The commonplace situation is for one or more bridges to separate nodes on different buses, but a bridge portal may also intercept and transform subactions between nodes on the same bus. These "echo" subactions use global node IDs.

— Global node IDs do not vary with the path to the node. On a bus with multiple bridge portals, each portal maps the same global node ID to the same local node ID.
— Global node IDs do not change unless asynchronous routes within the net change or unless no unallocated virtual ID is available when a new node is inserted on a bus.
— Whenever one or more global node IDs within a net might have changed, this information is reliably signaled on each bus within the net.

As already mentioned, a global node ID is composed of two parts, a 10-bit bus ID and a 6-bit virtual ID. 6-bit virtual IDs are managed locally on each bus by the coordinator, while 10-bit bus IDs are managed centrally for the entire net by the prime portal (see 4.3). This insures that within the scope of each there are no duplicated bus IDs or virtual IDs.

When a previously unrecognized node is inserted on a local bus, it is necessary for the coordinator to assign it a virtual ID and to communicate the virtual ID to all the other bridge portals on the bus. The coordinator maintains information, for each of the 63 possible virtual IDs, that indicates whether it is unallocated and, if allocated, the EUI-64 of the node to which it corresponds. When a node is inserted and its EUI-64 does not match one of those associated with a virtual ID, the coordinator chooses an unallocated virtual ID and assigns it to the node.

When a node is removed from a local bus, it does not immediately lose its virtual ID. The coordinator and all the other bridge portals retain the correlation between its EUI-64 and assigned virtual ID. Should the node subsequently be reinserted, it usually retains the same virtual ID. A node loses its virtual ID assignment if the coordinator requires a virtual ID for a previously unrecognized node and none of the 63 possible virtual IDs are available. In this case, the coordinator releases the current bus ID, discards all virtual ID assignments, and reassigns virtual IDs only for those nodes currently connected to the bus.

NOTE—When the coordinator releases the current bus ID, net update is initiated (see 4.3). This causes the alpha portal to request a new bus ID from the prime portal. Although the coordinator may reassign virtual IDs as soon as the bus ID has been released, the nodes with new virtual IDs are not addressable by global subactions until a bus ID is received from the prime portal.

In many cases subsequent to a bus reset, the coherency of the virtual ID mapping shared by all bridge portals can be maintained without communication between the portals and without intervention by the coordinator. Since the only event that causes the coordinator to assign a virtual ID is the insertion of a previously unrecognized node, bridge portals can rely on a topological analysis of self-ID packets to independently reestablish EUI-64 identity of all previously allocated virtual IDs (see Annex G). Analysis of self-ID packets establishes a correlation between EUI-64 and physical ID for all currently connected nodes, which in turn can be used to derive a tuple of physical ID, virtual ID, and EUI-64 for all nodes connected just prior to bus reset. When a previously unrecognized node is discovered, bridge portals may continue to use existing virtual ID mappings but must await the coordinator's assignment of a virtual ID to the new node.

4.2 Remote time-out

Within the context of the reference model illustrated by Figure 4-2, it is clear that a split transaction originated by Rq and responded by Rsp cannot be guaranteed to complete in the period used by Rq to time split transactions on the local bus. After the initial entry portal on bus 42 transmits *ack_pending* for the request subaction, the subaction will spend some time in the bridge's queues before it is forwarded across bus 941. The time a bridge persists in its attempts to forward a subaction to the next intermediate bridge portal is implementation-dependent; it might be longer than the requester's local bus split time-out. Even if the maximum forward time for a particular bridge is shorter than the requester's split time-out, the sum of all such times might exceed the requester's local split time-out. Finally, there is the split time-out on the destination bus. When the terminal exit portal transmits the request to Rsp, that node will assume that it has a local bus split time-out period within which to respond. When the response subaction returns by the reverse path to Rq, similar transit times are added to the total time that Rq has been awaiting a response.

The naturally longer time necessary for remote transactions leads to the definition of a ***remote time-out*** period analogous to local bus split time-out. Remote time-out is the sum of all the maximum forward times for all the bridges on the path between requester and responder plus the value of SPLIT_TIMEOUT register on the destination bus. The maximum forward times must be accumulated both for the request subaction on its outbound journey and for the response subaction

on its return trip. In the case of the response subaction, this includes the time the terminal exit portal (which was the initial entry portal from the perspective of the request subaction) persists in its attempt to deliver the response to the original requester.

Remote time-out is path dependent; a net management message exists to permit a requester to determine remote time-out for any valid global node ID. This message is processed by all intervening bridge portals, both in the outbound and return direction, and provides a vehicle within which the various maximum times can be accumulated. Once remote time-out value is obtained for a particular global node ID, it remains valid so long as the global node ID remains valid. From the perspective of a particular requester, the remote time-out for all nodes on a remote bus is identical to the remote time-out for any node on that bus.

NOTE—With reference to Figure 4-2, the remote time-out value obtained by Rq for Rsp might not be equal to the remote time-out value obtained by Rsp for Rq, since the local bus split time-out periods might be different for bus 42 and bus 17.

4.3 Clan affinity and net update

When both portals of a bridge are powered, it constitutes a net formed of the two buses connected to each of its portals. Even if no other devices are connected to those buses, a simple net exists. The bridge's manufacturer may make arbitrary design choices, such as the assignment of a bus ID to each of the buses, without the necessity for coordination with another bridge. However, once two bridge portals are connected to the same bus, coordination is required. A new concept, ***clan affinity***, is used to describe how coordination between bridge portals is effected.

A net topology that could be found in a home network is illustrated by Figure 4-3. The figure assumes a star topology with the central bus, 42, located within a wiring closet; the clusters of equipment in other rooms (not shown) are connected through bridges whose cabling radiates from the wiring closet.

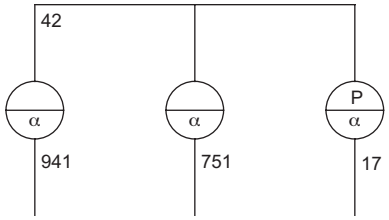


Figure 4-3 — Example home network topology

Under stable operating conditions, each of the bridge portals replicates the routing database for the net—but with slight variations that result from the portal's location within the net. The copies of the routing database are mutually consistent: no bus ID is duplicated within the net and valid routes to each bus ID exist from any point in the net. All of the bridge portals that share a consistent distributed database form a ***clan***. A clan is identified by the EUI-64 of a single portal within the clan, called the ***prime portal***. In Figure 4-3, the letter *P* designates the prime portal. A prime portal has three functions within a clan. The first is simply to provide the clan with an identity: the prime portal's EUI-64. This identity permits clans to be distinguished from each other when more than one clan is simultaneously connected to the same bus.[7] The second is to maintain a central registry of allocated bus IDs; whenever a bridge portal requires a new bus ID for its attached bus, it requests one from the prime portal. The central registry of bus IDs prevents duplicate assignment of bus IDs to geographically separate buses. The third is to identify the net cycle master (see 4.4).

Although any portal within a clan may assume the role of prime portal and perform its functions as capably as any other portal, some geographic locations within a clan are superior to others because the identity of the prime portal tends to be stable over time. For example, in Figure 4-3 the prime portal is shown on the central bus (assumed to be located within

[7]The coexistence of two or more clans on a single bus is a temporary state of affairs that is resolved by merging one clan into the other, but until this process completes, it is essential to distinguish between the different clans.

the wiring closet). Net topology changes could occur in the outlying rooms of the home network, caused by the insertion or removal of bridges, without the necessity to relocate the prime portal. A portal within the net may be configured to indicate a preference that it be selected as the prime portal.

Also shown in Figure 4-3 are ***alpha portals*****,** designated by the letter α; these are portals that are on the route from the local bus towards the prime portal. Alpha portals perform a useful function when it is necessary to route an inter-bridge message to the prime portal without knowledge of the prime portal's global ID. It suffices to send the message to the alpha portal on the bus of origin. The alpha portal transfers the message to its co-portal—which either is the prime portal or forwards the message to the next alpha portal.

The net topology illustrated by Figure 4-3 would change if a bridge portal were removed from or added to any bus in the net. The removal of a bridge portal is straightforward from the perspective of clan affinity. The clan loses a member and the remaining bridge portals are informed of the changes in the routing database; there is no change to the prime portal or to the clan affinity of the remaining portals. When a portal is inserted, the process is more complicated, since two clans must be merged into one. The clan that retains its prime portal is the ***survivor clan*** while those that lose their former prime portal and transfer their affinity to the survivor clan are called ***victim clans*****.** A hypothetical personal computer (PC), illustrated by Figure 4-4, is used to demonstrate the process that occurs upon the addition of a bridge to a bus that already includes one or more bridge portals.

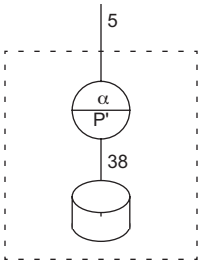


Figure 4-4 — Example PC with internal bridge

The PC is shown with a built-in bridge that separates an external connector from a private, internal bus. A mass storage device is connected to the internal bus; in this case, it is assumed to be advantageous to locate the system disk here so that it could have greater access to bus bandwidth than it would have if it competed with other devices external to the PC. So long as the PC remains disconnected from an external bus that contains other bridges, it forms a small net comprised of buses 5 and 38. The net has a prime portal, *P′* and an alpha portal. However, if a connection were made between bus 5 in Figure 4-4 and bus 17 in Figure 4-3, the topology illustrated by Figure 4-5 would result.

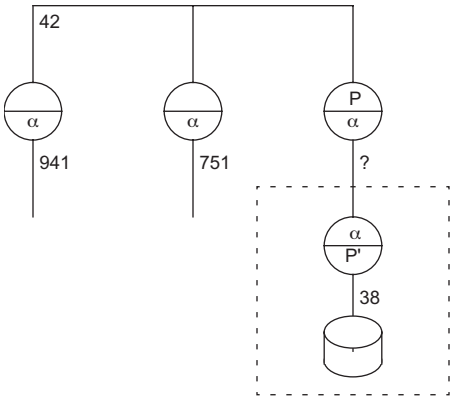


Figure 4-5 — Addition of a PC to the home network

When the connection is made between the PC and the home network, a bus reset is generated. After each bus reset, one bridge portal is selected to act as the ***coordinator*** to manage ***net update***, a process that updates net configuration if its topology has changed. The role of coordinator is retained until the next bus reset, at which time it might shift to a different bridge portal. In this example, the coordinator detects that two clans, *P* and *P′*, are present on the bus: net topology has changed. Since there were two bus IDs previously assigned to what is now a single bus, its bus ID is temporarily unresolved.[8] First, the coordinator determines which clan is to be the victim and which is to be the survivor. The algorithm employed is specified in Clause 10.. The coordinator combines the routing information from all clans into a new routing database and communicates this information to all the local bridge portals. The routing database in the example would include only bus IDs 17, 38, 42, 751, and 941—bus ID 5 is deleted because the just-reset bus retains its survivor clan bus ID assignment. At the same time that the routing database is transferred to each bridge portal, the coordinator informs it of its clan affinity. Within the victim clan, a portal's role as an alpha portal changes to that of a ***subordinate portal*** (any portal that is neither an alpha nor a prime portal) as it is adopted into the survivor clan. When these operations complete, the net is once again stable, as shown by Figure 4-6.

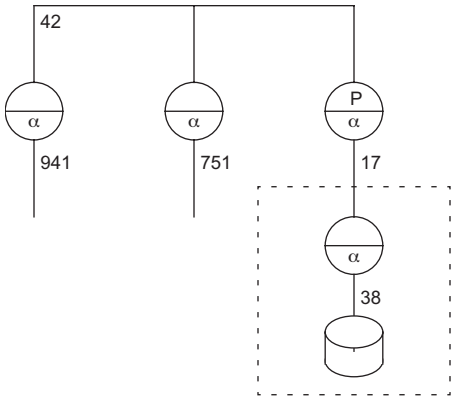


Figure 4-6 — Survivor clan with adopted PC

In Figure 4-6, the bridge within the PC has changed its clan affiliation and become part of the survivor clan, *P*.

4.4 Cycle time distribution and synchronization

Just as each bus has a single cycle master that provides uniform cycle time to that bus, a net requires a single cycle master to provide cycle offset for the entire net. This singular cycle master is named the ***net cycle master***; its location is determined by the location of the prime portal. The cycle master connected to the same bus as the prime portal is the net cycle master.[9]

Net cycle offset originates at the net cycle master and bridges distribute it throughout the net. On the bus that contains the prime portal, all portals obtain cycle time from the net cycle master and communicate its cycle offset to their co-portals, which in turn regulate cycle start events on their own buses. On all buses except the prime bus, the alpha portal receives cycle offset synchronization information from its co-portal and the subordinate portals receive cycle time information from the cycle master—and in turn pass its cycle offset to their co-portals.

For bridges to reliably transport isochronous streams, it is necessary for all cycle masters within the net to maintain frequency synchronization with each other. Without frequency synchronization, cycle time drift might grow large enough to cause isochronous data overrun or underrun. In practice, the simplest method to obtain frequency synchronization is to

[8]Any bus IDs duplicated in different clans (none are shown in this example) are discarded by the coordinator. Once net update completes on a particular bus, if the alpha portal discovers it lacks an assigned bus ID, a request is made to the prime portal for a new bus ID.

[9]A node other than the prime portal can be the net cycle master. For example, a gateway node with access to an accurate external time source, such as a network interface to a satellite or terrestrial cable system, could be the net cycle master.

maintain an effective cycle offset phase difference of zero (phase synchronization) between cycle timers on adjacent buses. Cycle offset phase synchronization exists when cycle start events are simultaneous within tolerances inherent to Serial Bus cycle time distribution and measurement.

The distribution of cycle offset synchronization from the net cycle master requires that any two adjacent buses be synchronized to the upstream cycle master, *i.e.*, the one with fewer intervening bridges between itself and the net cycle master. All alpha portals (except the prime portal) are downstream of the cycle master on their co-portal's bus (which is referred to as the ***upstream portal***) and therefore are responsible to synchronize cycle offset on their own bus to that on the upstream portal's bus. An alpha portal may regulate cycle offset on its own bus if either a) it is the cycle master or b) the cycle master is adjustable by means of "go fast" and "go slow" packets. These cycle master adjustment packets (see 6.3) permit the alpha portal to instruct a cycle master to delay or hasten the advent of the next cycle synchronization event by one tick of its 24.576 MHz cycle timer. Four items of data are sufficient for an alpha portal to determine whether a phase adjustment is required: simultaneous samples of CYCLE_TIME.*cycle_offset* from both of the bridge's portals, the propagation delay between the alpha portal and the downstream cycle master, and the propagation delay between the upstream portal and the upstream cycle master. Figure 4-7 illustrates the components of the feedback loop employed to maintain cycle offset phase synchronization between two buses in a net.

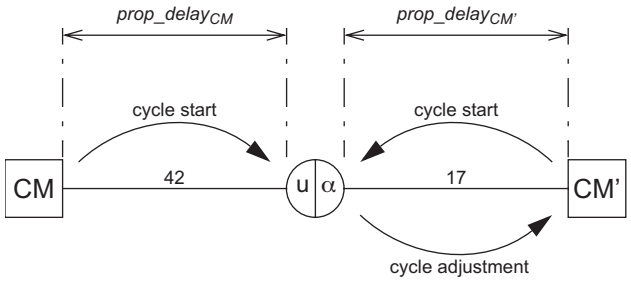


Figure 4-7 — Phase synchronization between two buses

The upstream and downstream (with respect to the net cycle master) cycle masters are marked *CM* and *CM'*; within the bridge the upstream portal is labeled *u* while the alpha portal is identified by α. The phase difference between the two buses can be calculated by the following formula:

$$(\text{CYCLE\_TIME}.cycle\_offset_{alpha} + prop\_delay_{CM'}) - (\text{CYCLE\_TIME}.cycle\_offset_{upstream} + prop\_delay_{CM})$$

Each bridge portal measures the propagation delay between itself and the cycle master on its bus by means of PHY ping packets (see IEEE Std 1394a-2000). The propagation delay is assumed to remain constant over time, within tolerances, unless the path between the portal and the cycle master alters. The propagation delay is half the round-trip delay measured by a ping packet. Both bridge portals have an independent 24.576 MHz cycle timer whose value can be sampled *via* the CYCLE_TIME register. Once each isochronous period, the phase of both cycle timers (visible as CYCLE_TIME.*cycle_offset*) is sampled and the result combined with the propagation delays according to the formula above. If the resultant phase difference is negative, the downstream cycle master is running slower than the upstream cycle master and the alpha portal instructs it to reduce the threshold value for the impending cycle start by one cycle timer tick. Otherwise, when a positive phase difference is observed, the downstream cycle master is too fast and the alpha portal instructs it to increase the threshold value by one cycle timer tick. Any adjustments made by the alpha portal to the cycle master's threshold value are in effect only for the next CYCLE_TIME.*cycle_count* increment, whereupon the threshold reverts to the nominal 3071 cycle timer ticks. An example of data that could be observed by a bridge, along with the calculated phase difference and the resultant "go fast" adjustment packet transmitted to the downstream cycle master, CM', is captured in the table that follows.

Observed data	CYCLE_TIME.*cycle_offset*$_{upstream}$	25 ticks
	prop_delay$_{CM}$	7 ticks
	CYCLE_TIME.*cycle_offset*$_{alpha}$	15 ticks
	prop_delay$_{CM'}$	5 ticks
	Phase difference	–12 ticks
Result	Alpha portal transmits a cycle master adjustment packet to temporarily decrease the CYCLE_TIME.*cycle_count* increment threshold by one cycle timer tick (”go fast”).	

NOTE—Although the preceding discussion assumes that the alpha portal and the downstream cycle master are distinct nodes, the principles are the same if the alpha portal is also the cycle master for its bus. In this case, the value of $prop\_delay_{CM'}$ is zero.

4.5 Universal time

The preceding subclause explained the necessity for a single source of cycle time synchronization within the net (called the net cycle master) and described how bridge portals distribute synchronized cycle offset. Bridge portals, however, do not distribute a constant time reference throughout the net. There is no directly accessible universal time pervasive across the net: BUS_TIME and CYCLE_TIME.*cycle_count* may vary (and probably do vary) from bus to bus even though CYCLE_TIME.*cycle_offset* is maintained in phase lock synchronization on all buses.

A common time reference is valuable for many applications, whether the need for coordinated action on separate buses is coarse-grained (*e.g.*, a video source, such as a set-top box, becomes active at close to the same time that a destination, such as a DVCR, seeks to record the program) or more exacting, as when editing or mixing is performed. Although a universal time reference is not directly maintained within the net, facilities are provided that permit applications to derive a common sense of time. All that is required is a common reference bus mutually agreed by the applications.

Derivation of universal time is based upon a net management message that permits the time difference, in cycles, to be measured between any two buses within the net. Consider the example topology illustrated by Figure 4-3. Assume that it is possible to simultaneously sample BUS_TIME and CYCLE_TIME.*cycle_count* on all the buses,[10] with the results summarized in the table below.

Bus ID	BUS_TIME	CYCLE_TIME	Cycles
17	3	0690 Cxxx$_{16}$	26 316
42	1	022B 2xxx$_{16}$	8690
751	5	0B56 1xxx$_{16}$	45 473
941	2	04F3 1xxx$_{16}$	19 889

The rightmost column represents the bus time in terms of cycles, that is, BUS_TIME seconds multiplied by 8000 plus CYCLE_TIME.*cycle_count*. From a simultaneous sample of bus time on all the buses, the time difference offset between any pair of buses can be derived, as shown in the table that follows. Note that the signed time offset in cycles is computed for ordered pairs of buses; at a given moment T_{source_bus}, the simultaneous time $T_{destination\_bus}$ is obtained by adding the time offset to T_{source_bus}.

[10] In fact, it is not possible to sample bus time simultaneously on different buses, but it is possible to obtain equivalent information by means of a net management message whose operation is described later in this subclause.

		Destination bus ID			
		17	**42**	**751**	**941**
Source Bus ID	**17**	0	–17 626	19 157	–6427
	42	17 626	0	36 783	11 199
	751	–19 157	–36 783	0	–25 584
	941	6427	–11 199	25 584	0

Equipped with knowledge of relative time offsets between arbitrary pairs of buses, applications located on buses remote with respect to each other may mutually agree upon a common time; all that is required is a shared point of reference. One such reference is the net cycle master itself. In order to use time on the prime bus, T_{prime_bus}, as the common time reference, applications need only convert their local time to the equivalent time on the prime bus and communicate it to the remote participant, which can then reconvert it to local time. Consider a set-top box located on bus 17 and a DVCR located on bus 941 in Figure 4-3. A user, *via* some unspecified interface such as a remote control, instructs the set-top box to transmit the program starting at a cycle count sometime in the future with respect to current local time on bus 17, T_{bus_17}. The user also wishes to program the DVCR on bus 941 to commence recording at the same time (this is to be accomplished indirectly, *via* the set-top box user interface). The set-top box converts T_{bus_17} to its equivalent time on the prime bus by subtracting 17 626 cycles and then communicates this T_{prime_bus} (along with channel number, bandwidth characteristics and other command information) to the DVCR, which then adds 11 199 cycles to obtain equivalent T_{bus_941} on its local bus.

This example shows how a common sense of time can be shared by applications on different buses even if there is no absolute sense of calendar date and time. All that is necessary is a common point of reference, relative to which “now” can be derived in terms of local bus time. The obvious candidate for shared reference is the prime bus—but there might instances for which another point of reference is superior. For example, if a network interface connected to a terrestrial cable system were located on bus 751 and accurate calendar date and time were available from the service provider, applications could share calendar date and time by determining the relative time offset, in cycles, between their local bus time and the time provided by the network interface. Determination of this relative time offset is a two-step process. First, the time difference between the local bus and bus 751 is obtained as described above. Second, the time difference, if any, between T_{bus_751} and the calendar date and time provided by the network interface is separately obtained and arithmetically combined with the relative time difference between the buses.

As already noted in a footnote (Footnote 10), it is not possible for devices (except bridges) to take simultaneous samples of bus time on different buses. In the examples given, a hypothetical simultaneous sample was used to derive the relative time offsets between any ordered pair of buses in the net. The same information can be obtained by other means: a net management request and its corresponding response. This method relies upon each bridge on the route between two buses to intercept the message, update the cumulative time difference between the two buses by adding the time difference measured between its own bridge portals, and forward it to the next bridge portal on the route. After the terminal bridge intercepts and processes the request message, it converts it to a response and transmits it directly to the originator without further processing by the intervening bridges.

Once the relative time difference between two buses has been measured, it does not change unless the value of BUS_TIME or CYCLE_TIME.*cycle_count* on either bus changes other than as a result of normal increment—in which case it is necessary to measure the relative time offset anew. Bridge portals do not change BUS_TIME or discontinuously alter CYCLE_TIME.*cycle_count* except during net update, which invalidates all cached relative time difference measurements. In addition, a bridge portal that observes a change, other than as a result of normal increment, in bus time or cycle time should initiate net update.

4.6 Stream connection management

Although bridges configure themselves with respect to asynchronous subaction routing and GASP transmitted on the default broadcast channel, isochronous stream communication between two nodes located on different buses requires explicit connection management. Messages are used to set up, tear down, and query the status of stream routing within the net.

This subclause introduces the reader to the basic principles of stream connection management; it uses the model net topology illustrated by Figure 4-8 as a reference. The drawing is simplified by the omission of most of the devices that would populate a real-world example—including the cycle masters and isochronous resource managers. Neither the prime portal nor the alpha portals are illustrated, since they are not relevant to the scenarios discussed.

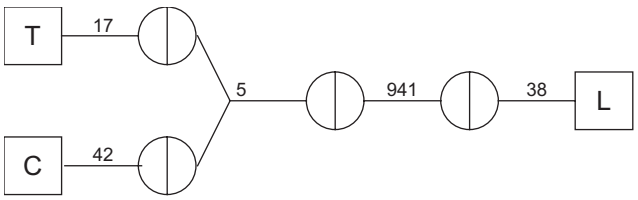


Figure 4-8 — Stream connection management (reference model)

Bridges are shown as circles with a line to divide each portal from the other. The bus IDs are shown above the lines that represent different arbitration domains. The controller, talker, and listeners are squares with the designation C, T, and L respectively. The following definitions are essential to understand the examples:

controller: A node that uses net management messages to set up or tear down routing for a stream between a talker and one or more listeners.

initial entry portal: The first bridge portal that snoops a subaction and transforms the subaction's *source_ID* from a local node ID to a global node ID.

listener: A node (other than a bridge portal) that is the recipient of a stream.

listening portal: A bridge portal that listens to a set of channel numbers so that their packets may be retransmitted by its co-portal.

reallocation proxy: A bridge portal responsible to reallocate channel numbers and bandwidth after a bus reset. The reallocation proxy also keeps track of listeners on the local bus; if the count of listeners ever drops to zero, either because of explicit connection teardown or node removal, the reallocation proxy releases the pertinent channel number and bandwidth.

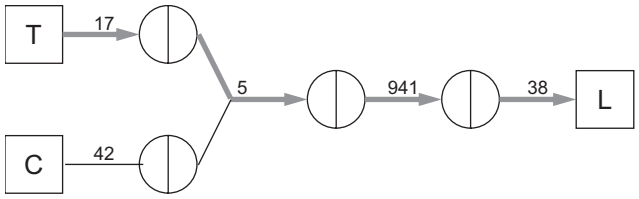
stream: Isochronous data originated by a talker and received by zero or more listeners. A stream is uniquely identified by the talker's EUI-64 and an index unique within the context of the talker.

talker: A node (other than a bridge portal) that is the source of a stream within the net.

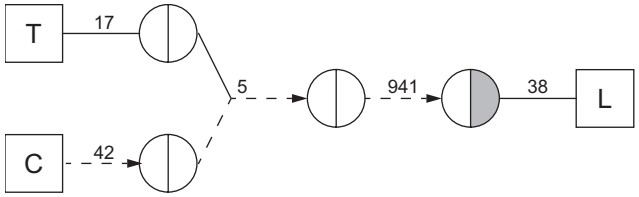
talking portal: A bridge portal that transmits stream packets for a set of channel numbers.

terminal exit portal: The last bridge portal that transmits (or could transmit) a subaction en route to its destination; the portal that normally transforms the subaction's *destination_ID* from a global node ID to a local node ID.

The first example is a simple connection setup that does not overlay an existing connection. The controller on bus 42 wishes to establish a path from the talker on bus 17 to the listener on bus 38 (shown by gray arrows in the figure below). Once the controller has received confirmation that the path has been created, it may instruct the talker to commence transmission of isochronous data.



The controller initiates the connection by transmitting a JOIN request as if it were intended for the listener. The JOIN request's *snarf* field[11] is one, which causes it to be intercepted by the terminal exit portal. The path of the JOIN request is indicated by dashed lines below; the portal that intercepts the JOIN request is shown shaded.



The JOIN request contains parameters sufficient to permit incremental set-up of the entire connection and eventual provision of completion notification to the originating controller. The parameters pertinent to this discussion are listed in the table below.

JOIN request parameters

Parameter	Description
talker_EUI_64	Unique ID from talker's bus information block.
talker_index *listener_index*	Values unique within the context of the talker or listener that identify the stream.
controller_ID *talker_ID* *listener_ID*	Each is used during different phases of the stream setup process to route the JOIN request or other requests towards the appropriate bridge portal.
max_payload *aggregate_payload*	In conjunction with speed, permits calculation of bandwidth units to be allocated from BANDWIDTH_AVAILABLE.
tspd *lspd*	The maximum speed at which stream packets are to be transmitted by the talker or are to be transmitted to the listener (by the terminal exit portal), respectively.

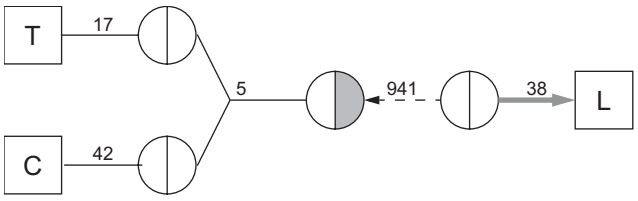
In addition to the parameters above, the composite of *talker_EUI_64* and *talker_index* form another parameter, *stream_ID*, which uniquely identifies a stream within a net of interconnected buses. A stream ID remains valid across bus resets and net topology changes; it persists until the stream, explicitly or automatically, is torn down.

The initial recipient of the JOIN request, a terminal exit portal, becomes the reallocation proxy on its local bus for the stream identified by *stream_ID* only if it is the talking portal for the stream. A reallocation proxy maintains a database for all active isochronous streams, keyed upon *stream_ID*. The database includes the information summarized by the following table.

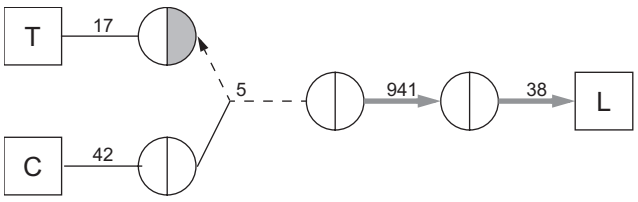
[11] A field in the block write packet header, newly defined by this standard. It may cause some or all of the bridge portals to intercept and process the request subaction en route to its destination. See 6.5 for details.

Stream context information maintained by a reallocation proxy	
Parameter	**Description**
stream_ID	Uniquely identifies a stream within a net of interconnected buses.
controller_ID	Provide error or other status information to the controller if the reallocation proxy takes action that affects the stream.
channel	The channel number allocated by the reallocation proxy from CHANNELS_AVAILABLE for the stream's use on the local bus
payload	If nonzero, the maximum payload permitted in a stream packet transmitted on the local bus. When zero, there is no payload limit.
speed	The speed at which stream packets are to be transmitted.
bandwidth	The number of bandwidth units allocated for the stream's use on the local bus. It is calculated from payload and speed but also includes an allowance for isochronous arbitration overhead.
listeners	A bit map that represents each node on the local bus for which a JOIN request has been successfully processed. The use of a bit map in place of a count permits the reallocation proxy to monitor the removal of listeners.

Upon receipt of a JOIN request, the reallocation proxy first reserves the necessary resources, channel number and bandwidth on its local bus. If successful in its attempted reservations, the reallocation proxy's local bus is configured for the stream (indicated by the gray arrow in the next figure below) and an iterative process is started that extends a stream path backward from the listener's bus to the talker's bus. This process uses a JOIN request sent to either a) the talking portal (or what will be the talking portal) for the stream on the immediately upstream bus or b) a portal selected to be a reallocation proxy on the talker's bus. This JOIN request is essentially the same as the initial JOIN request originated by the controller except that its *destination_ID* is the talker's (not the listener's) global node ID and the *snarf* field has a value of three, which causes it to be intercepted by all portals on its route. The path of this JOIN request, originated by the terminal exit portal, is shown by a dashed line below:



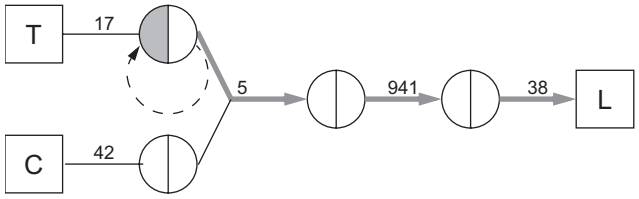
If the recipient of this JOIN request is not already a reallocation proxy for the stream identified by *stream_ID* it becomes one, as described above, and allocates the necessary resources from the isochronous resource manager on its bus. If the recipient is already the talking portal and reallocation proxy for the stream, it simply adds the new listener (the bridge portal identified by *source_ID* in the JOIN request's packet header) to its bit map of members on the local bus. In either case, if resources are successfully allocated, the process is iterated as shown below:



In this iteration, the JOIN request is once again addressed to the talker's *destination_ID*; the *snarf* field causes it to be intercepted by the next entry portal on the route back to the talker. Resources were successfully allocated on bus 941; the new JOIN will cause resources to be allocated on bus five and the bit map of local listeners to be updated.

Eventually the connection's path will be established on all intermediate buses and the terminal bus that contains the listener. At this point, the behavior required of a JOIN request changes. All of the previous JOIN requests have nominated a bridge portal as the reallocation proxy for a particular bus—and each of these bridge portals is also the talking portal that transmits the stream on the bus. However, now that the JOIN message has arrived at the talker's bus, there is no need for a talking portal: the stream's source is the talker itself. Yet, there is still a requirement for a reallocation proxy on the talker's bus, since the talker is not necessarily a controller and is not necessarily responsible to reallocate stream resources after bus reset.

In this case, the listening portal on the talker's bus is designated as a reallocation proxy. The last step in resource allocation is shown by the figure below. No Serial Bus transaction is involved—the listening portal is the recipient of the JOIN request from its co-portal, as is indicated by the dashed line. More than one bridge on the talker's bus could be listening to the same stream. In this case, all the listening portals are potential reallocation proxies; a cooperative connection management protocol is used to avoid duplicate allocation of resources.

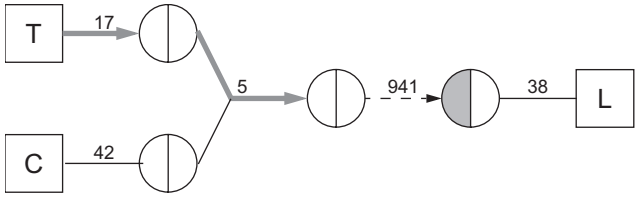


Once the final set of resources are allocated (on the talker's bus), it is necessary to communicate to the various listening portals on the route from talker to listener the channel number they should associate with a particular stream ID. This is accomplished by a LISTEN request propagated and modified by the listening portals. The parameters included within a LISTEN request are shown in the table below.

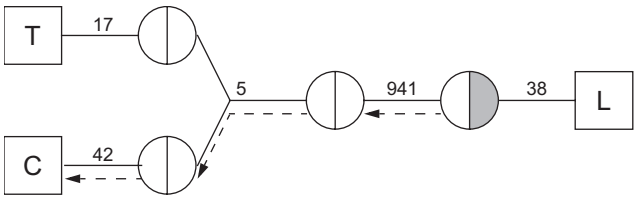
LISTEN request parameters	
Parameter	**Description**
stream_ID	Uniquely identifies a stream within a net of interconnected buses.
channel	The channel number on which the talking portal for the bus will transmit the stream.

The process starts after the listening portal on the talker's bus has either confirmed that resources are already allocated or else allocated resources in its role as reallocation proxy for the stream. The listening portal transmits a LISTEN request that identifies the channel number allocated and associates it with the stream. The request's *destination_ID* is set to the value of the listener's global node ID (saved from the JOIN request originated by the controller) and the *snarf* field is set to three so the packet will be intercepted by all listening portals on the way towards the listener.

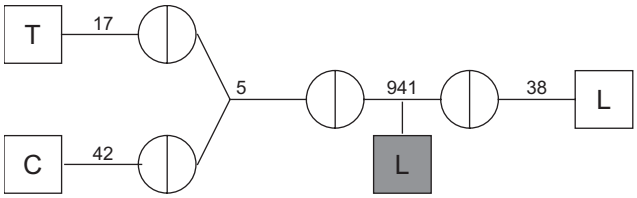
When a listening portal receives a LISTEN request, it configures itself to listen to the indicated *channel* and forward stream packets to its co-portal. The listening portal also forwards the LISTEN request to its co-portal, which modifies the request to identify the channel number used when the co-portal (talking portal) transmits the stream on its bus. The talking portal retransmits the LISTEN request, which is still addressed to the listener on the terminal bus. This process iterates for all intermediate buses until the terminal bridge intercepts the LISTEN request. The figure below shows the arrival of the request at the terminal bridge; at this point, all of the upstream buses are fully configured and able to transport the stream.



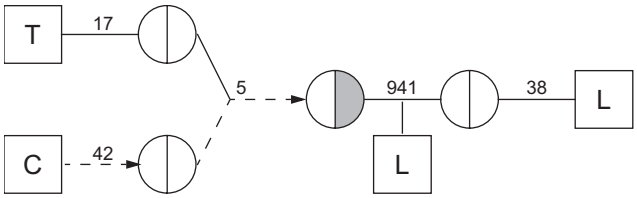
The terminal bridge has several jobs. First, it configures itself to listen on the indicated channel (as have all of the upstream listening portals). Next, it passes the request to its co-portal—but instead of further propagation of the request to another listening portal, the co-portal configures the listener to receive the stream on the allocated channel. Once this step is complete, the terminal exit portal transmits a STREAM STATUS message to the controller that originated the JOIN request. This confirmation contains the *stream_ID* supplied by the controller in the original request (the path of the confirmation message, transmitted by the portal that intercepted the original JOIN request from the controller, is shown by a dashed line).



What happens in the case that a new listener is to be added to an existing multicast group? The operations are similar, even though the route from the talker to the new listener might completely or partially overlay the path already in use. Consider the addition of a new listener (shown shaded).

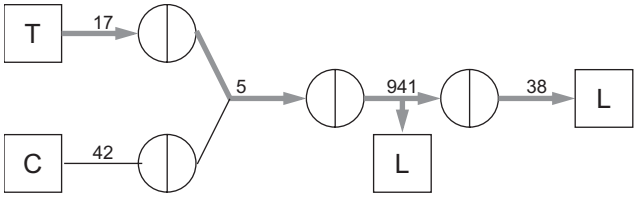


The controller transmits a JOIN request to the listener to be added; the *snarf* field is one to cause the terminal exit portal to intercept the request.

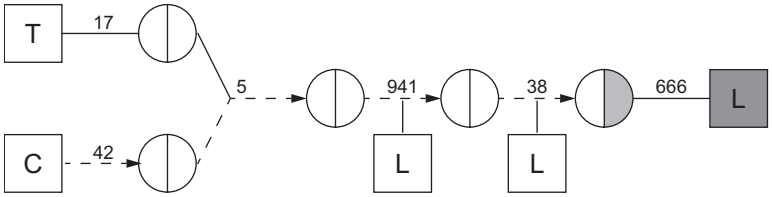


The sequence of JOIN, LISTEN, and STREAM STATUS messages generated by the bridge portals is exactly analogous to that already described, except that no new resources are allocated. The shaded bridge portal, in its role as reallocation proxy, updates its bit map of members to indicate the presence of the new listener but otherwise has no additional work to perform.

The gray arrows in the figure below show the resultant stream path once the operations have completed.



If the overlaid connection involved a new listener beyond the current extent of the stream's flow, the stream path is extended to accommodate the added listener. This is illustrated below.



The variations on this theme are endless (complex multicast routes with multiple branches could be created) but the basic operations of JOIN, LISTEN, and STREAM STATUS remain the same.

When a stream connection to a particular listener is no longer needed, it is usually torn down in an orderly fashion by a LEAVE message. Just as is the case for the JOIN message, the stream controller transmits a LEAVE message toward the listener. When it reaches the talking portal on the listener's bus, the connection with the listener is deleted and, if no other listeners remain on the bus, the stream's isochronous resources (channel and bandwidth) are released. Once the resources are released, the talking portal transmits the LEAVE message toward the talker; its receipt by intervening portals on the path between talker and listener causes resources to be released. After all the resources on the path have been released, the last portal process the LEAVE message transmits a STREAM STATUS message to the stream controller.

An isochronous connection can also be torn down in response to an unexpected disconnection anywhere along the path from talker to listener. For example, if the talker is disconnected, all of the listening portals on the talker's former bus detect this event and initiate stream teardown. If a listener is disconnected, its talking portal detects the event and initiates stream teardown for that listener, only. If the connection between an intermediate talking portal and an intermediate listening portal is severed, two actions take place: the talking portal initiates stream teardown for the disconnected listening portal, while the listening portal initiates stream teardown for all of its downstream listeners.

5. Bridge portal and bridge-aware node facilities

This clause describes the facilities that a bridge portal or a bridge-aware node shall support in order to interoperate with other bridge portals. These facilities are configuration ROM entries (which identify the presence of a bridge within a Serial Bus node) and control and status registers (CSRs—which can be used to control the operations of or obtain information from a bridge portal or bridge-aware node).

5.1 Configuration ROM

Bridge-aware nodes and bridge portals shall implement configuration ROM in the general format defined by IEEE 1394. Annex H contains a sample of valid configuration ROM for a bridge portal and illustrates the usage of the entries defined below.

5.1.1 Bus information block

A bridge-aware node's or bridge portal's configuration ROM shall contain a bus information block, as defined by this standard in Figure 5-1.

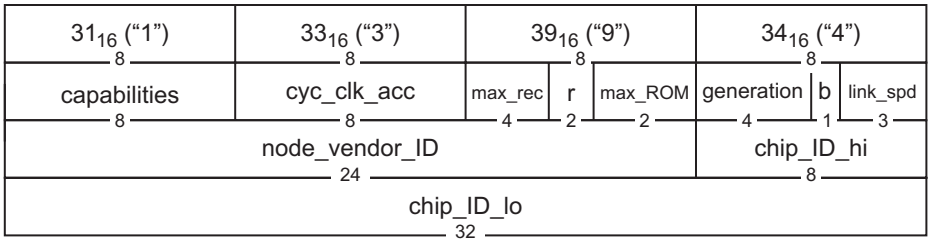


Figure 5-1 — Bus information block format

The definition and usage of all fields specified by IEEE 1394, with the exception of *max_rec*, are unchanged. Other fields are new while some have particular applicability to Serial Bus bridges, as defined below.

The ***capabilities*** field consists of individual bits shown in Figure 5-2.

Figure 5-2 — Bus information block *capabilities* field

A bridge portal shall report values of one for the *irmc*, *cmc,* and *adjustable* bits; bridge portals and bridge-aware nodes whose *irmc* bit is one shall implement the BANDWIDTH_AVAILABLE and CHANNELS_AVAILABLE registers with the lock (*compare_swap*) algorithms specified by IEEE Std 1394a-2000. The *isc* bit does not describe any bridge capabilities, only the node's isochronous capabilities on the local bus. If the node is a bridge portal, consult the Bridge_Capabilities entry in configuration ROM to determine whether the portal is capable of forwarding isochronous data to its co-portal. When the *adjustable* bit is one, the node implements support for the cycle master adjustment packets defined in 6.2 and interprets them as specified by 8.1.2. If the *adjustable* bit is one, the *cmc* bit shall also be one and the node shall implement support for the PHY ping packet specified by IEEE Std 1394a-2000. Bus manager and power manager capabilities are orthogonal to the other capabilities; the *bmc* and *pmc* bits shall be set according to the presence or absence of these optional capabilities.

NOTE—Nodes other than bridge portals may implement an adjustable cycle master and report this capability by means of the *adjustable* bit.

The ***max_rec*** field specifies the maximum payload that a bridge portal accepts in any of an asynchronous block write request, lock request, block read response, lock response, or asynchronous stream packet. This usage is an extension to that defined by IEEE 1394, which specifies only the size of an asynchronous block write request. When *max_rec* is zero the portal's maximum payload is not specified. Otherwise, the maximum payload is defined as 2^{max_rec+1} bytes and shall be the maximum asynchronous data payload permitted by the slower of each portal's fastest PHY port.

NOTE—The maximum payload for an isochronous packet has no relationship to the *max_rec* field; see the *max_isoch* field defined in 5.1.4.

The ***bridge_aware*** bit (abbreviated as *b* in the Figure 5-1), when set to one indicates the node complies with the requirements for a bridge-aware device specified in 9.2 and elsewhere within this standard. A bridge portal shall be bridge-aware and report a value of one for this bit.

The ***node_vendor_ID***, ***chip_ID_hi,*** and ***chip_ID_lo*** fields collectively form the bridge portal's EUI-64, whose value shall not be equal to the value of the co-portal's EUI-64.

5.1.2 Node_Capabilities entry

The Node_Capabilities entry is obsolete. Conforming devices may include the entry in their root directory, as specified by IEEE 1394, but this is not recommended.

5.1.3 Bus_Dependent_Info entry

The Bus_Dependent_Info entry is a directory entry in the root directory that specifies the location of the Bus_Dependent_Info directory within configuration ROM. Figure 5-3 shows the format of this entry.

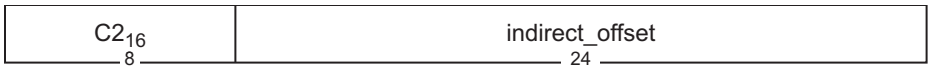


Figure 5-3 — Bus_Dependent_Info entry format

The entry is identified by the *key* field, which has a value of $C2_{16}$.

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

5.1.4 Bridge_Capabilities entry

The Bridge_Capabilities entry is an immediate entry in the Bus_Dependent_Info directory that identifies the presence of a bridge portal within the node and specifies its capabilities. Bridge portal configuration ROM shall contain a Bridge_Capabilities entry even at times when the node's self-ID packets report a *brdg* value of zero. Figure 5-4 shows the format of this entry.

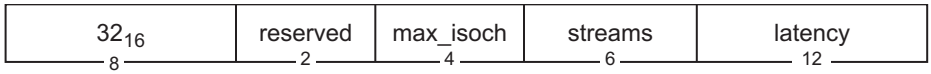


Figure 5-4 — Bridge_Capabilities entry format

The entry is identified by the *key* field, which has a value of 32_{16}.

The ***max_isoch*** field specifies the maximum isochronous subaction data payload the bridge is capable of transporting from the fastest PHY port of one portal to the fastest PHY port of the other. The bridge's internal fabric shall be capable of transporting isochronous subactions at least as large as those permitted by the slower of each portal's link speed. When *max_isoch* is zero, the portal's maximum isochronous payload is not specified. Otherwise, the maximum isochronous payload is defined as $2^{max\_isoch+1}$ bytes. The value of *max_isoch* shall be the same for both portals of a bridge.

The ***streams*** field specifies the maximum number of concurrently active isochronous streams supported by the bridge. A value of zero indicates that the bridge does not forward isochronous streams. The value of *streams* shall be the same for both portals of a bridge. With the exception of heterogeneous bridges, *streams* shall be nonzero.

NOTE—A bridge might not be able to support as many concurrently active isochronous streams as the maximum reported in the *streams* field if the aggregate bandwidth of the streams exceeds the bridge's resources.

The ***latency*** field specifies the constant delay that isochronous packets incur when they are transferred from a bridge's entry portal to a listener on the bus connected to the bridge's exit portal. If the bridge implements no capability to forward isochronous streams (*i.e.*, the *streams* field is zero) or if different isochronous streams can incur different constant delays, the value of *latency* shall be zero. Otherwise, the *latency* field shall specify the delay in units of 125 μs and shall have a minimum value of two. The value of *latency* may be different for each of a bridge's portals.

5.2 Control and status registers

The control and status registers (CSRs) implemented by a bridge portal or bridge-aware node shall conform to the requirements defined by this standard and its normative references. The CSRs belong to three groups:

— Core registers specified by IEEE Std 1212-2001
— Bus-dependent registers specified by IEEE 1394
— Registers specified by this standard

The addresses of all 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 shall be consulted for detailed descriptions of both core and Serial Bus-dependent registers; Table 5-1 lists those that are mandatory for bridge-aware nodes.

Table 5-1 — Bridge-aware node control and status registers

Offset	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 bridge portal locally.
C_{16}	RESET_START	Initiates a command reset; see individual register definitions for the effects (if any).
$18_{16} – 1C_{16}$	SPLIT_TIMEOUT	Time limit for split transactions on the connected bus.
$80_{16} – BC_{16}$	MESSAGE_REQUEST	Well-known addresses for receipt and protocol demultiplex of 64-byte messages in a standard format.
$C0_{16} – FC_{16}$	MESSAGE_RESPONSE	
214_{16}	QUARANTINE	Permits bridge-aware nodes (which include bridge portals) to manage their quarantine periods.

In addition to those registers required of bridge-aware nodes, bridge portals shall implement the registers listed in Table 5-2.

Table 5-2 — Bridge portal control and status registers

Offset	Name	Description
200_{16}	CYCLE_TIME	24.576 MHz cycle timer required for isochronous operation.
204_{16}	BUS_TIME	System time in seconds.
210_{16}	BUSY_TIMEOUT	Controls the bridge portal's retry protocols.
$21C_{16}$	BUS_MANAGER_ID	Contains the 16-bit *node_ID* of the bus manager on the connected bus, if one is present.
220_{16}	BANDWIDTH_AVAILABLE	Known location for Serial Bus bandwidth allocation.
$224_{16} – 228_{16}$	CHANNELS_AVAILABLE	Known location for Serial Bus channel allocation.
234_{16}	BROADCAST_CHANNEL	Identifies the channel number used for asynchronous broadcast.
$1C00_{16} – 1DFC_{16}$	VIRTUAL_ID_MAP	Correlates EUI-64s of nodes on the local bus with virtual IDs assigned by the coordinator.
$1E00_{16} – 1EFC_{16}$	ROUTE_MAP	Asynchronous routing information for each bridge portal.
$1F00_{16} – 1F04_{16}$	CLAN_EUI_64	Identifies the clan of which a bridge portal is a member.
$1F08_{16}$	CLAN_INFO	Clan information (bus ID, bridge hops to prime portal, *etc*.) used during net update.

The CYCLE_TIME, BUS_TIME, BUS_MANAGER_ID, BANDWIDTH_AVAILABLE, CHANNELS_AVAILABLE, and BROADCAST_CHANNEL registers are required because a bridge portal might need to function as an isochronous resource manager.

The QUARANTINE, VIRTUAL_ID_MAP, ROUTE_MAP, CLAN_EUI_64, and CLAN_INFO registers are particular to bridge functions and are specified by this standard. If *brdg* was zero in a bridge portal's most recent set of self-ID packets, it may respond to requests addressed to these registers even though their contents might not be meaningful.

5.2.1 QUARANTINE register

This register permits bridge-aware nodes and bridge-portals to manage their own quarantine periods. Both bridge-aware nodes and bridge portals shall implement the QUARANTINE register. The contents of the QUARANTINE register are updated in response to analysis of self-ID packets observed by the node and are affected by write requests.

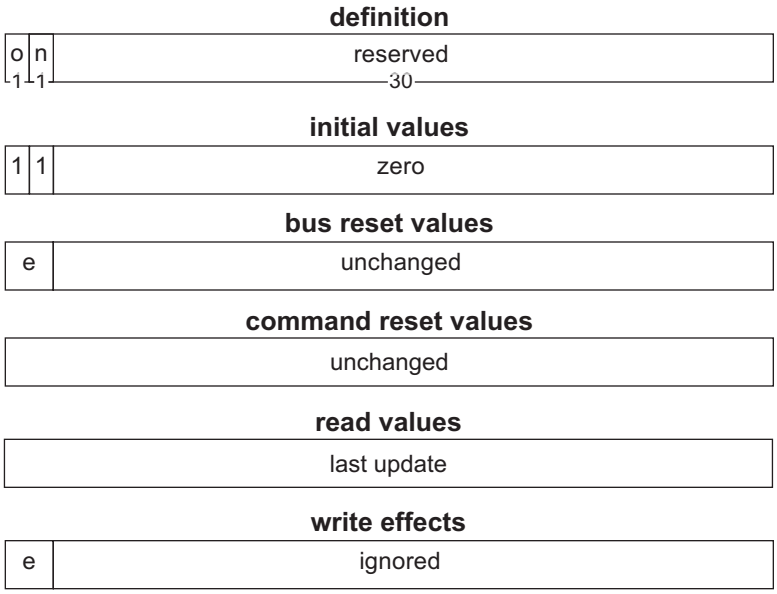


Figure 5-5 — QUARANTINE format

The *orphan* bit (abbreviated as *o* in Figure 5-5) indicates whether a global node ID is assigned to the node. A zero value indicates that the node is assigned a valid global node ID, while a value of one indicates uncertainty. When this bit is zero, the node may originate both global and local subactions. Otherwise, when *orphan* is one, the node shall not originate global subactions.

The *net_update* bit (abbreviated as *n* in Figure 5-5) indicates whether net update is in progress. Bridge-aware nodes that are not also bridge portals shall reserve this bit; its value shall be zero. Bridge portals shall implement the *net_update* bit with an initial value of one. When this bit is zero, the bridge portal may transmit both global and local subactions. Otherwise, when *net_update* is one, the bridge portal shall not transmit global subactions.

NOTE—"Originate" is a subset of "transmit." If a subaction's *source_ID* contains a local node ID, the subaction was originated by the node that transmitted it; however, if it contains a global node ID, the subaction was transmitted—but not originated—by a bridge portal.

A bridge-aware node or bridge portal shall set its *orphan* and *net_update* bits (when implemented) to one if, subsequent to a bus reset, any of the following conditions exist:

— One or more self-ID packets (including the bridge portal's own self-ID packet) are observed whose *brdg* field is equal to three.
— Topology analysis of the self-ID packets (see Annex G) reveals that one or more bridge portals, not present on the bus prior to the bus reset, have been connected to the bus.
— Topology analysis of the self-ID packets reveals that one or more bridge portals, present on the bus prior to the bus reset, have been removed from the bus.
— Topology analysis of the self-ID packets reveals that one or more nodes, present on the bus prior to the bus reset, has reported a *brdg* field whose value changed from zero to nonzero (or vice versa) relative to the value reported prior to the bus reset. This last condition indicates the virtual insertion or removal of a bridge portal.

Any of these conditions indicates that either the node does not have an assigned global node ID or that net update is underway. When the state of either the *orphan* or *net_update* bit changes from zero to one, the node has recognized that quarantine has begun. Once either bit has been set to one, it shall retain its value through subsequent bus resets until zeroed. When the *net_update* bit changes from one to zero, a bridge portal shall set the value of *brdg* to two.

NOTE—The values of the *net_update* and *orphan* bits might not be identical for all nodes on a bus. For example, when a bridge-aware node is connected to the bus, its *orphan* bit is one but all the other node's *orphan* bits remain zero. The already connected nodes do not perceive a net topology change but the newly connected node observes a bridge portal not present before the bus reset.

Write requests addressed to the QUARANTINE register affect the value of the *orphan* and *net_update* bits according to the value written. A zero written to either bit shall zero the bit while a one written to either bit shall not affect its value. Whenever zero is written to QUARANTINE.*orphan*, the bridge portal shall clear its *transmit_virtual_ID_map* flag (see 11.1) to FALSE. A bridge portal ignores writes to its QUARANTINE register during certain critical periods that can occur during net update; see 10.5 for details.

5.2.2 VIRTUAL_ID_MAP register

This 512-byte register correlates an EUI-64 with each of the 63 possible virtual IDs assignable to nodes on the local bus. It also identifies the alpha portal. The format of the register is specified by Figure 5-6.

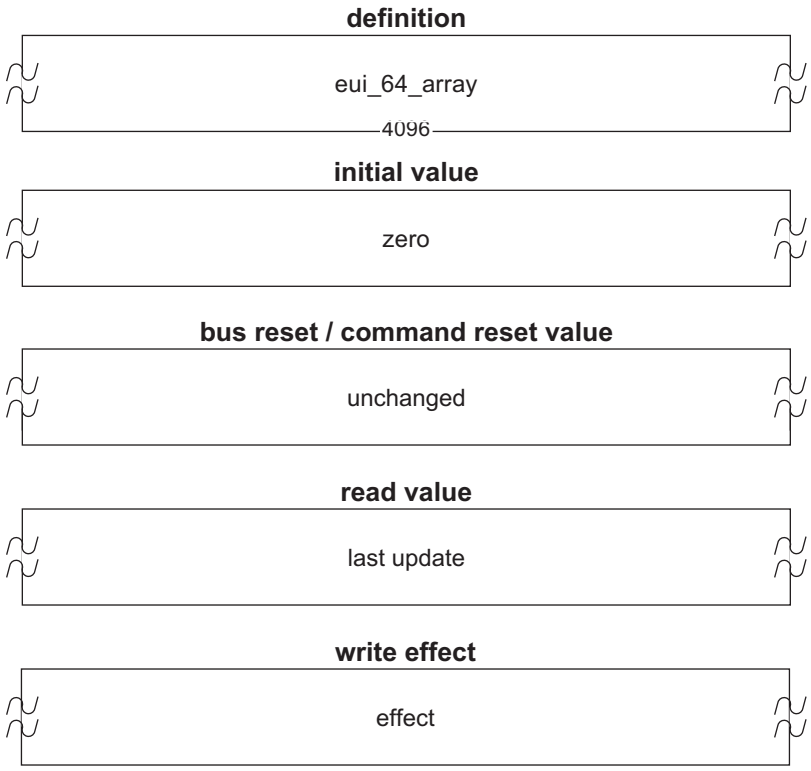


Figure 5-6 — VIRTUAL_ID_MAP format

The ***eui_64_array*** consists of 64 eight-byte entries, each of which contains an EUI-64. With the exception of the last entry, the value of VIRTUAL_ID_MAP.*eui_64_array*[*n*] shall specify the EUI-64 of the node assigned virtual ID *n*. An entry whose value is zero indicates an unassigned virtual ID. Since $3F_{16}$ is invalid as a virtual ID, its corresponding entry in the virtual ID map is used for a different purpose. VIRTUAL_ID_MAP.*eui_64_array*[63] shall contain the EUI-64 of the alpha portal; in no other case shall a nonzero EUI-64 be present more than once in the array.

Bridge portals shall reject write requests addressed to any portion of the VIRTUAL_ID_MAP register except block write requests originated by the coordinator with a *data_length* of 512 bytes addressed to the start of the register's address space. The coordinator maintains its own VIRTUAL_ID_MAP autonomously, without need of block write requests. A bridge portal that receives a 512-byte write addressed to its VIRTUAL_ID_MAP register shall set its *transmit_virtual_ID_map* flag (see 11.1) to TRUE.

5.2.3 ROUTE_MAP register

This is a 256-byte register that contains the bridge portal's asynchronous routing information for all 1023 possible buses within a net. The format of the register is specified by Figure 5-7.

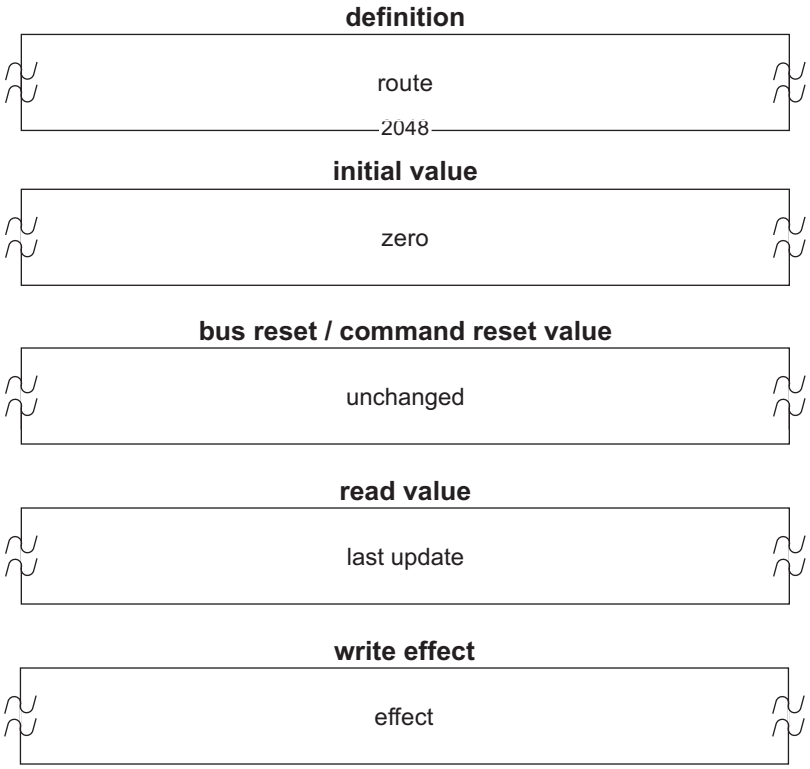


Figure 5-7 — ROUTE_MAP format

The ***route*** data is an array of 1024 two-bit entries, each of which represents the asynchronous subaction routing state for the corresponding bus ID. Although the local bus ID, $3FF_{16}$, is not routable, its space in the array is allocated for the sake of simplicity. For a particular *bus_ID*, reference is made to byte *n*, relative to zero, within the *route* data, where *n* is equal to *bus_ID* / 4. Figure 5-8 illustrates the arrangement of two-bit *route* entries within a byte.

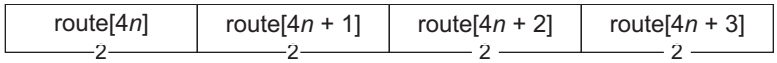


Figure 5-8 — ROUTE_MAP entries format (within one byte)

NOTE—Throughout this standard, the notation ROUTE_MAP[*bus_ID*] represents the *route* entry that corresponds to *bus_ID*.

Each *route* entry shall have a value specified by Table 5-3. The *route* entry for the unused local bus ID, $3FF_{16}$, shall be CLEAN.

Table 5-3 — ROUTE_MAP entries encoding

Value	Name	Comment
0	CLEAN	The bus ID is not in use within the clan; subactions addressed to this bus ID are not forwarded by any portal.
1	DIRTY	A valid route to the bus ID no longer exists; subactions addressed to this bus ID are not forwarded by any portal. This is a transient state that persists until net update completes.
2	VALID	The bus ID is in use within the clan but subactions addressed to this bus ID are not forwarded by this portal.
3	FORWARD	The bus ID is in use within the clan and subactions addressed to this bus ID are forwarded by this portal.

5.2.4 CLAN_EUI_64 register

The CLAN_EUI_64 register identifies the clan to which the bridge portal belongs. A clan is identified by the EUI-64 of one of the clan's bridge portals, designated the prime portal. The register is specified by Figure 5-9.

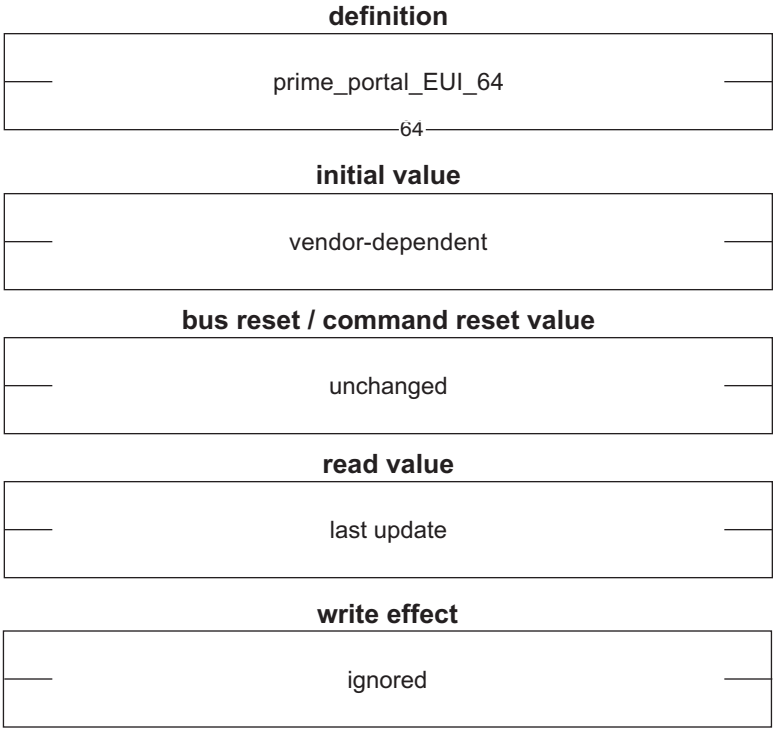


Figure 5-9—CLAN_EUI_64 format

The ***prime_portal_EUI_64*** field shall contain the same value as the *eui_64* field in the prime portal's bus information block. The initial value shall be equal for both portals of a bridge, shall be the EUI-64 of either the bridge portal or its co-portal, and shall be the larger byte-wise reversed EUI-64. The comparison of the two EUI-64 values shall be performed in

byte-wise reverse order, *i.e.*, for the purpose of the unsigned comparison, the least significant byte shall be treated as if it were most significant, the most significant byte shall be treated as if it were least significant, and the interior bytes shall be compared in reverse order of their normal significance.

5.2.5 CLAN_INFO register

The CLAN_INFO register contains state information for the bridge portal and its clan, as specified by Figure 5-10.

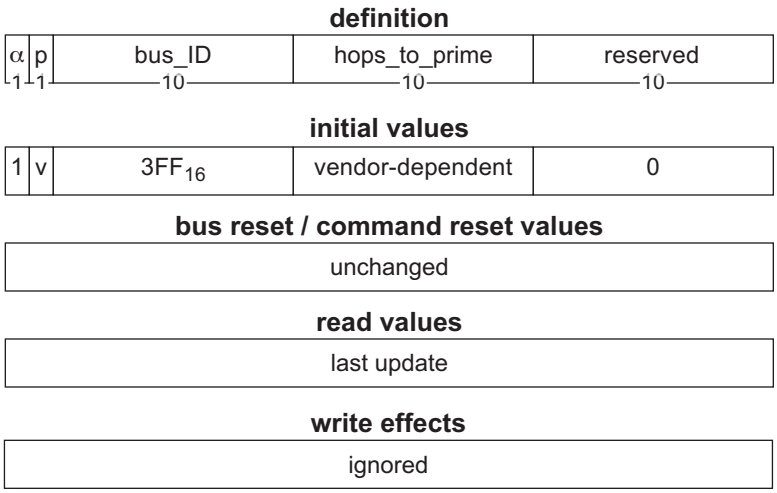


Figure 5-10—CLAN_INFO format

The ***alpha*** bit (abbreviated by the letter α in Figure 5-10) shall be zero if the bridge portal is a subordinate portal. If the portal is an alpha portal, the *alpha* bit shall be one. An alpha portal whose EUI-64 in its bus information block is equal to the value of its CLAN_EUI_64 register is a prime portal.

NOTE—The initial value of *alpha* is one since upon completion of power reset initialization a bridge consists of a prime portal and its alpha co-portal.

The ***preferred_clan*** bit (abbreviated by the letter *p* in Figure 5-10) indicates whether the clan shall be granted preference in the survivor clan selection algorithm (see 10.2). The value of the *preferred_clan* bit shall be identical for all of a clan's portals and shall be inherited from the prime portal. The prime portal's *preferred_clan* bit shall be zero unless it is field-configurable. Within a net, only one portal should be configured to set *preferred_clan* to one if and when it becomes the prime portal.

The ***bus_ID*** field shall be $3FF_{16}$ if the local bus connected to the portal has no bus ID assigned and otherwise shall specify the assigned bus ID.

The ***hops_to_prime*** field shall be zero if the bridge portal is the prime portal or on the same bus as the prime portal. Otherwise, the field shall contain the count of bridges that are on the routing path between the portal and the prime portal. For alpha portals other than the prime portal, the count shall include the bridge that contains the alpha portal. On a particular bus, the value of *hops_to_prime* shall be equal for all of a clan's portals. Although the initial value of *hops_to_prime* is vendor-dependent, it shall be either zero or one dependent upon whether the bridge portal is prime or not, respectively.

6. Packet formats

This standard defines packet formats for inter-bridge communications; it also modifies primary packets defined by IEEE 1394 in order to add supplemental information unique to the bridged environment.

6.1 Self-ID packet zero

Subsequent to a bus reset it is essential to determine whether bridge portals have been inserted or removed or otherwise to indicate a change in net topology and, if so, to rapidly select one portal to coordinate net update on the bus. When nodes require rapid and effectively simultaneous access to information after bus reset, the only reliable carriers are the self-ID packets; the first of which is modified to include information that distinguishes bridge portals from other Serial Bus nodes. The format of self-ID packet zero illustrated by Figure 6-1 replaces that specified by IEEE Std 1394a-2000.

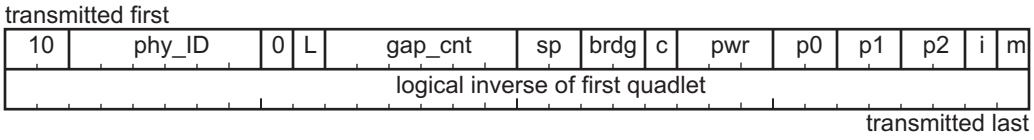


Figure 6-1 — Self-ID packet zero format

Two bits previously reserved by IEEE Std 1394a-2000 are redefined to become the bridge-capability field, ***brdg***. The value of the *brdg* field shall be ignored if the *L* bit in the self-ID packet is zero. Table 6-1 enumerates the values for this field, which is an addition to Table 4-1 in IEEE Std 1394a-2000.

Table 6-1 — Bridge-capability field in self-ID packet zero

Field	Derived from	Comment
brdg		Bridge capabilities of the node: 00_2 Not a bridge portal 01_2 Reserved for future standardization 10_2 Bridge portal (unchanged net topology) 11_2 Bridge portal (changed net topology)

The *brdg* field not only identifies bridge portals but also indicates whether net update is necessary (see 10.2 and 11.1 for more detailed information). Once *brdg* has been set to three, its value shall persist across bus reset; only when the QUARANTINE.*net_update* bit is zeroed shall *brdg* be set to two.

NOTE—The means by which a bridge portal controls the value of the *brdg* field are implementation-dependent, but the method specified by IEEE Std 1394b-2002, which permits control *via* the PHY registers, is strongly recommended.

6.2 Cycle master adjustment packet

A cycle master adjustment packet, illustrated by Figure 6-2, instructs the recipient to adjust the interval between successive cycle synchronization events. A cycle master adjustment packet shall be transmitted only during an isochronous period; it requires no bandwidth allocation.

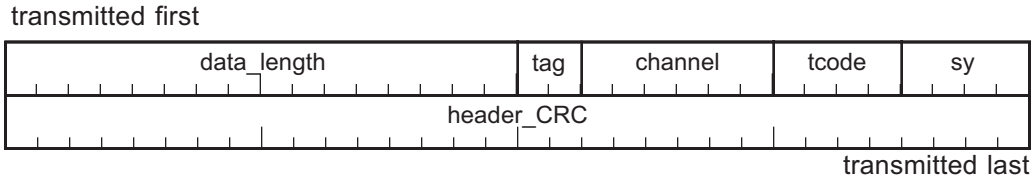


Figure 6-2 — Cycle master adjustment packet format

The ***data_length*** field shall be zero.

The ***tag*** field shall be three.

The ***channel*** field shall be 31 (the default broadcast channel).

The ***tcode*** field shall be A_{16} (a stream packet).

The ***sy*** field shall indicate the adjustment to be made to the cycle master's behavior at the time of the next cycle synchronization event. The adjustment is effected by a temporary override of the threshold value above which CYCLE_TIME.*cycle_offset* wraps around to zero and causes CYCLE_TIME.*cycle_count* to increment. A value of one specifies that the threshold shall be set to 3072, an *sy* value of two specifies that the threshold shall be set to 3071 while an *sy* value of three specifies that threshold shall be set to 3070. An *sy* value of zero is reserved; if the cycle master observes a cycle master adjustment packet that specifies a zero *sy* value of *sy* it shall ignore the packet and make no adjustment to the threshold value. Upon the next cycle synchronization event, the threshold value shall be restored to 3071 (see 8.1.2).

NOTE—The cycle master adjustment packet *sy* values of three and one, respectively, correspond to "go fast" and "go slow" commands sent to the cycle master; they hasten or delay, respectively, the next cycle synchronization event by one tick of the cycle master's 24.576 MHz cycle timer.

Nodes other than the cycle master may ignore cycle master adjustment packets.

Global asynchronous stream packets (GASP) also use the default broadcast channel. In order to reliably identify cycle master adjustment packets, recipients shall verify that all of the packet header fields have the values specified above. GASP are differentiated from cycle master adjustment packets in two important ways: a) the *data_length* field is greater than or equal to eight and b) the *sy* field is either zero or greater than or equal to four (see Table 6-4).

6.3 Response packet

When both the originator of a transaction request and the responder are connected to the local bus there is no possibility of confusion over the responder's identity: it is the node identified by the response packet *source_ID* field. However, once bridges lie on the path between requester and responder, matters are more complex: a request packet might encounter congestion or transmission errors en route to its destination, the routing information might no longer be valid, or the intended recipient might have been detached from the terminal bus or might be in a power conservation mode, unable to provide a timely response. In all of these cases, the response is generated by some bridge portal acting as a proxy for the node identified by *source_ID*.

In order to localize the source of errors for management purposes, fields are defined within the first three quadlets of the header for all response packets, as illustrated by Figure 6-3. The format of write response, quadlet read response, block read response, and lock response packets is identical within these quadlets, although the length of the header as well as the contents of the subsequent quadlets vary for each type of response.

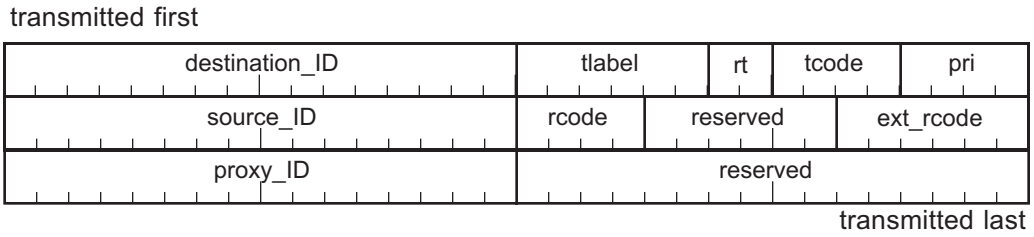


Figure 6-3 — Response header format (first three quadlets)

The meaning and usage of the *destination_ID*, *tlabel*, *rt*, *tcode*, *pri*, and *source_ID* fields shall be as specified by IEEE 1394.

If the node identified by *source_ID* originated the response packet, the meaning and usage of the *rcode* field are specified by IEEE 1394. Otherwise, if the bridge portal identified by *proxy_ID* originated the response packet, the meaning and usage of the *rcode* field are specified by this standard (see D.2).

NOTE—An initial entry portal that detects a transaction error may indicate this to the requester either by transmission of a pending acknowledgement and subsequent response packet or, if there is an acknowledge code equivalent to the intended *rcode*, by transmission of a terminal acknowledgement.

The *ext_rcode* field indicates the validity of the information in the *proxy_ID* field and can provide additional information that localizes the source of the error. When *ext_rcode* is zero, the contents of *proxy_ID* are unspecified. Otherwise, the *proxy_ID* field shall contain the global node ID of the bridge portal that originated the response. Table 6-2 specifies the meaning of nonzero values for *ext_rcode*; all values not shown are reserved for future standardization.

Table 6-2 — Extended response codes

Value	Name	Meaning
0		No extended response information available.
$1 - F_{16}$		The value of the terminal acknowledgment received in response to a request subaction.
10_{16}	*ext_legacy_quarantine*	Remote read or lock request invalid from a legacy source, *i.e.*, one that is not bridge-aware.
11_{16}	*ext_invalid_route*	Nonexistent destination; destination *bus_ID* not present in route maps.
12_{16}	*ext_invalid_global_ID*	Nonexistent destination; no such global node ID on terminal bus.
14_{16}	*ext_payload_too_big*	The asynchronous subaction is too large for a bridge or is too large to be transferred between bridge portals on an intermediate bus. In the case of block read transactions, a bridge portal can detect the error in either a request or response subaction.
15_{16}	*ext_congestion*	Bridge congestion; maximum request forwarding time exceeded.
16_{16}	*ext_data_error*	Malformed response packet received; unable to salvage contents of data payload.

Table 6-2 — Extended response codes *(continued)*

Value	Name	Meaning
17_{16}	*ext_no_virtual_ID*	The coordinator has not yet assigned a virtual ID to the requester; the request can be retried later.
$3F_{16}$	*ext_unspecified*	Unspecified error.

Extended response codes are meaningful only in combination with certain response codes, as specified by Table 6-3. With the exception of the *ext_unspecified* extended response code, which may be combined with any response code other than *resp_complete*, combinations not shown in the table are prohibited.

Table 6-3 — Valid response and extended response code combinations

Response code	Valid extended response codes	Comment
resp_complete	—	Extended response codes are meaningful only for transactions that complete in error.
resp_conflict_error	*ack_busy_X* *ack_busy_A* *ack_busy_B*	Detected by any exit portal when the maximum forward time is exceeded.
	ack_tardy *ack_conflict_error*	Only detected by the terminal exit portal.
	ext_congestion	Reported by an entry portal if there are no bridge resources to transfer the subaction to the co-portal.
resp_data_error	*ack_data_error*	Only detected by the terminal exit portal.
resp_type_error	*ack_type_error*	Only detected by the terminal exit portal.
	ext_quarantine_violation	Reported by the entry portal on the bus to which the legacy node is connected. Merge with no virtual ID.
	ext_payload_too_big	May be reported by any bridge portal on the route between requester and responder.
resp_address_error	*ack_address_error*	Detected by any exit portal.
	ext_invalid_global_ID	Detected by the terminal exit portal when no local ID is mapped to the destination virtual ID.

The *proxy_ID* field identifies the originator of the response packet when it is other than the node identified by *source_ID*. When a node to which a request was addressed transmits the corresponding response, both the *ext_rcode* and *proxy_ID* fields shall be zero. Otherwise *proxy_ID* shall contain the global node ID of the bridge portal that originated the response. In this case, *source_ID* shall be identical to the *destination_ID* field from the corresponding request packet.

6.4 Global asynchronous stream packets (GASP)

This standard defines additional values (and permanently reserves others) for the *sy* field in the GASP format specified by IEEE Std 1394a-2000. Table 6-4 enumerates the permitted values for *sy*; it includes the zero value previously standardized.

Table 6-4 — GASP *sy* values

sy	Description
0	Local bus GASP
1 – 3	Reserved; not to be standardized
4 – 7	Reserved for future standardization
8	Remote GASP
9 – 15	Reserved for future standardization

GASP whose *sy* value is less than or equal to seven shall not be forwarded across bridges. GASP whose *sy* value is eight or greater shall be forwarded by bridges, as specified by this standard.

6.5 Net management message interception

Net management messages (see 6.6) are used to request information about the net or to configure its operations. They can be originated either by bridge portals or bridge-aware devices. In the simplest case, a net management message is addressed directly to a bridge portal. However, some messages require processing by more than one portal, while others are intended for a portal whose global node ID is unknown to the originator of the message. In these last two cases, the relevant portals are always on the route from the originator of the message to a node whose global node ID is known to the message's sender. All that is needed is a method to identify messages to be intercepted by bridge portals and, once intercepted, to specify whether the message shall continue to be forwarded along the route determined by its *destination_ID* or shall generate a message in response from the intercepting portal.

In order to enable message interception by bridge portals, this standard defines a field, *snarf*, in the block write request packet header. The 16-bit *extended_tcode* field specified by IEEE 1394 is reduced to an 8-bit field and two of the available bits are allocated to the new field. This redefinition of *extended_tcode* is backwards compatible with IEEE 1394 since values eight through $FFFF_{16}$, inclusive, are reserved for future standardization. Figure 6-4 specifies the revised definition of a write request for data block.

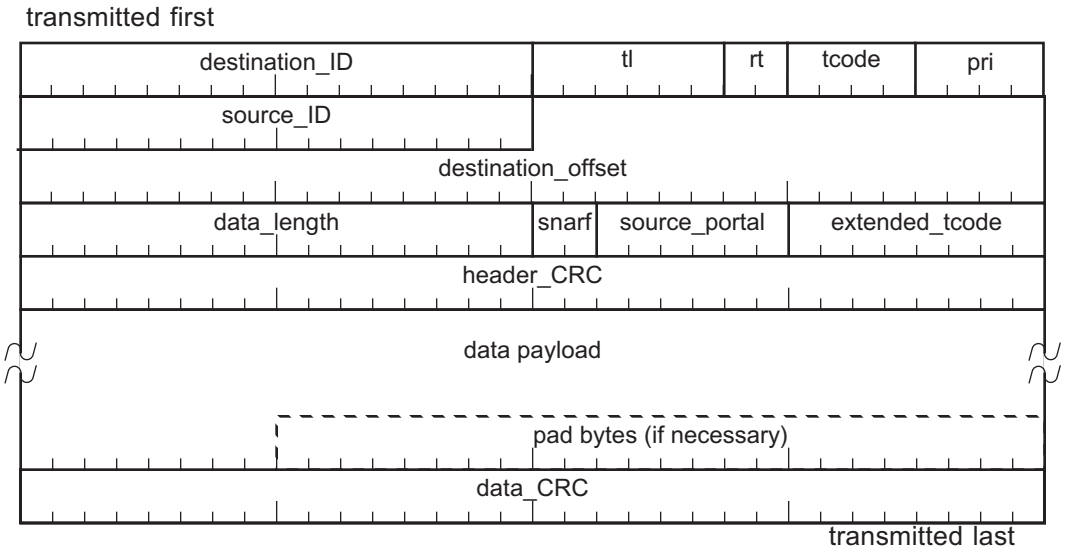


Figure 6-4 — Write request for data block packet format

The usage of the *destination_ID*, *tl*, *rt*, *tcode*, *pri*, *source_ID*, *destination_offset*, *data_length,* and *extended_tcode* fields (as well as both the header and data CRC) shall be as specified by IEEE 1394.

The ***snarf*** field shall be ignored if the write request *destination_ID* field contains a local node ID, otherwise it indicates which bridge portals, if any, intercept the subaction en route to its final destination, according to Table 6-5.

Table 6-5 — *snarf* field encoding

Value	Action
0	Not intercepted by any bridge portal; forwarded to *destination_ID*. This is the request subaction processing defined by IEEE 1394.
1	Intercepted by the terminal exit portal on the route determined by *destination_ID*.

Table 6-5 — *snarf* field encoding *(continued)*

Value	Action
2	Intercepted by the initial entry portal on the route determined by *destination_ID*.
3	Intercepted by all bridge portals on the route determined by *destination_ID*.

When an entry portal snoops a bridge-bound block write request subaction, the permissible acknowledgments depend upon the value of the *snarf* field. When *snarf* is zero, entry portals process subactions as described in Clause 7. and Annex D. All entry portals that snoop a bridge-bound block write request subaction whose *snarf* field is nonzero shall, unless busy, transmit a terminal acknowledgement or *ack_pending*. A pending acknowledgement shall not be followed by a response subaction; with one exception (see E.1.5), the originator of a block write request with a nonzero *snarf* field shall consider the transaction complete upon receipt of *ack_pending*. Once a portal has intercepted a request subaction, additional actions are determined by the message format (see 6.6 and E.1.5).

When a bridge portal transmits a write request subaction whose *snarf* field is nonzero, the ***source_portal*** field shall contain the physical ID of the portal. Otherwise, the contents of the field are undefined.

6.6 Net management messages

Net management messages shall be encapsulated within a block write request addressed to the MESSAGE_REQUEST or MESSAGE_RESPONSE registers; the format of the data payload shall conform to IEEE Std 1212-2001. For convenience of reference, the data payload format is reproduced in Figure 6-5.

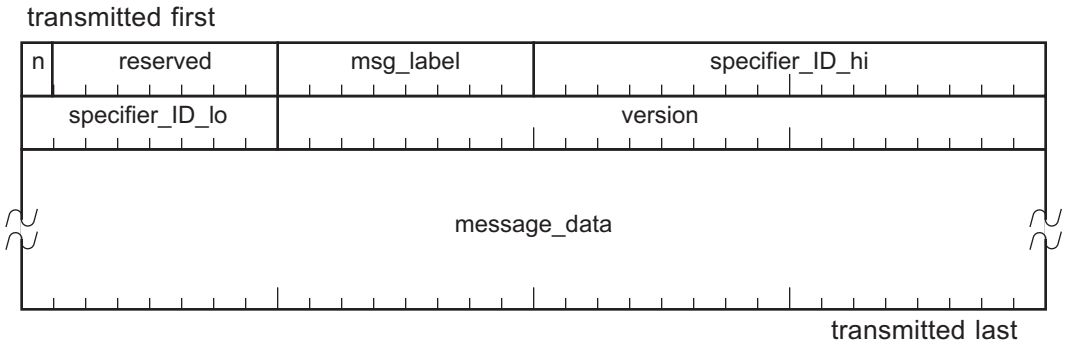


Figure 6-5 — Net management message encapsulation within block write data payload

The ***notify*** bit (abbreviated as *n* in Figure 6-5) shall be one in block write requests addressed to the MESSAGE_REQUEST register and zero in those addressed to the MESSAGE_RESPONSE register.

NOTE—The meaning of the *notify* bit depends upon its context. When a net management message is written to the MESSAGE_REQUEST register and *notify* is one, a response to the requester's MESSAGE_RESPONSE register is expected. When a net management response is received with a zero *notify* bit, the bridge portal has processed the net management request, whose success or failure is indicated by information in *message_data*.

The usage of the ***msg_label*** field shall be as specified by IEEE Std 1212-2001.

The value of ***specifier_ID*** (the 24-bit field formed by the concatenation of *specifier_ID_hi* and *specifier_ID_lo*) shall be $00\ A03F_{16}$, the Organizationally Unique Identifier (OUI) granted to the IEEE Microprocessor Standards Committee (MSC) by the IEEE Registration Authority. This value indicates that the MSC and its Working Groups are responsible for the maintenance of this standard. The value of the ***version*** field shall be $00\ 0200_{16}$. Their combined 48-bit value identifies this standard as the document that specifies the meaning of the *message_data* that follows.

The ***message_data*** field shall contain the net management message in the format illustrated by Figure 6-6.

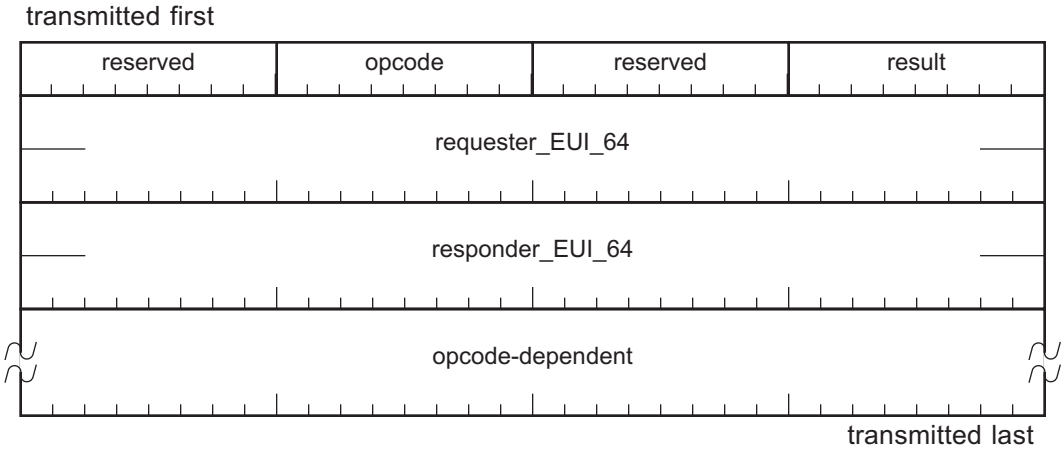


Figure 6-6 — Net management message format

The ***opcode*** field shall indicate the net management message type, as specified by the table below. All values not enumerated in the table are reserved for future standardization.

opcode	Name	Comment
1	TIMEOUT	Request remote time-out information for a global node ID.
2	TIME OFFSET	Request the difference in bus time between the local bus and a remote bus.
16	JOIN	Establish a stream connection between a talker and listener when controller, talker and listener are not on the same bus.
17	LEAVE	Delete a listener from an existing stream connection.
18	LISTEN	Originated by talking bridge portals, only. Used to communicate a stream's channel number to a listening portal on the local bus.
19	RENEW	Establish a new value for the remaining lifetime of a listener's stream connection.
20	TEARDOWN	Tear down all or part of a stream if the stream's path is severed.
21	STREAM STATUS	Report stream status to the requester in response to a JOIN, LEAVE or RENEW message.
80_{16}	MUTE	Instruct a bridge portal to disable forwarding of both asynchronous and stream subactions.
81_{16}	BUS ID	Request a bus ID assignment from the prime portal or return an assigned bus ID.
82_{16}	BUS ID ANNOUNCEMENT	Inform bridge portals of a newly assigned bus ID.
83_{16}	PANIC	Instruct bridge portals to shut down bridge functions and perform power reset initialization.

The contents of the ***result*** field shall be set to FF_{16} by the originator of a net management request. For a net management response, the *result* field shall indicate the completion status of the request, as specified by the table below. All values not enumerated in the table are reserved for future standardization.

result	Name	Comment
0	OK	The net management request completed successfully.
1	ERROR	The net management request did not complete successfully.
2	CONNECTION DELETED	
FF_{16}	PENDING	An intermediate value maintained in the net management message until its completion.

The ***requester_EUI_64*** field shall contain the requester's EUI-64, as obtained from its bus information block.

The contents of the ***responder_EUI_64*** field are unspecified for a net management request. For a net management response, the field shall contain the responder's EUI-64, as obtained from its bus information block. If a bridge portal generates a response on behalf of the node identified by the *source_ID* field, the portal shall have originally obtained the EUI-64 from the node's bus information block by means of two quadlet read transactions.

The size and format of the remainder of the net management message is dependent upon ***opcode***. In the subclauses that follow, the first five quadlets of the net management message format are not repeated, since they are the same for all values of *opcode*.

6.6.1 TIMEOUT message

A TIMEOUT request message can be used to obtain the remote time-out for the path between the requester and a remote bus or a remote node. The format of the *opcode_dependent* portion of the message is illustrated by Figure 6-7.

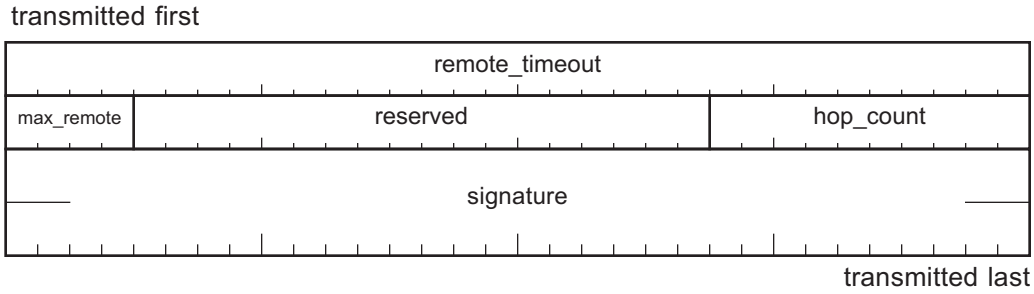


Figure 6-7 — TIMEOUT message format

The ***remote_timeout*** field, denominated in units of 125 µs, is used to accumulate the sum of a) the maximum time allotted each bridge on the route to *destination_ID* within which to forward a request subaction to the next bridge,[12] b) the maximum time allotted each bridge on the return route to the requester (*source_ID* in the original TIMEOUT request) within which to forward a response subaction to the next bridge,[13] and c) the value of the SPLIT_TIMEOUT register on the terminal bus. Each bridge on the route shall update the *remote_timeout* field dependent upon the *destination_offset* to which the TIMEOUT message is addressed. When the message is addressed to the MESSAGE_REQUEST register, the

[12] This value is vendor-dependent and may vary from bridge to bridge, although it is subject to the minimum and maximum constraints specified in 7.4.

[13] This value is vendor-dependent and, subject to the minimum and maximum constraints specified in 7.4, may vary both from bridge to bridge or from the same bridge's maximum forward time for request subactions.

bridge shall add the maximum forward time for request subactions (from the perspective of the exit portal) to *remote_timeout*. Otherwise, when the message is addressed to the MESSAGE_RESPONSE register, the bridge shall add the maximum forward time for response subactions (from the perspective of the exit portal) to *remote_timeout*. In addition, when the bridge contains the terminal exit portal it shall convert the value of the terminal exit portal's SPLIT_TIMEOUT register to units of 125 µs and add it to *remote_timeout*.

The ***max_remote*** field specifies the maximum asynchronous subaction data payload that may be transferred between the originator of the TIMEOUT message and the node addressed by *destination_ID*, as $2^{max\_remote+1}$ bytes. It shall be updated, by each bridge that intercepts the TIMEOUT message, to contain the minimum of a) the field's current value, b) the value that describes the maximum asynchronous subaction data payload that the bridge is capable of transferring between its portals, or c) the value that describes the maximum data payload that can be transmitted between the exit portal and either the next recipient of the TIMEOUT message or the node addressed by *destination_ID*. The last term is affected by entry and exit portal *max_rec*, PHY and link speeds, and also the minimum PHY port speed on the path between the exit portal and either the next entry portal or the node addressed by *destination_ID*. The terminal exit portal does not include the destination node's *max_rec* information in the calculation.

The ***hop_count*** field shall contain the number of bridges that lie on the path between the two endpoints. Each bridge shall increment *hop_count* by one.

The originator of the TIMEOUT request initializes the ***signature*** field; it can be used to correlate a TIMEOUT response with an outstanding request. Bridges that process a TIMEOUT message en route to its destination shall not modify the *signature* field.

The originator of the TIMEOUT request shall zero reserved fields and the *remote_timeout* and *hop_count* fields, and shall set the *max_remote* field to F_{16} before transmission.

A TIMEOUT request message shall be encapsulated as a block write request addressed to the remote node's MESSAGE_REQUEST register. The least significant six bits of *destination_ID* can be equal to $3F_{16}$; in this case, the TIMEOUT message is intended to determine remote time-out for the bus ID specified by the most significant ten bits of *destination_ID*. The *snarf* field in the packet header shall have a value of three (all bridges on the route shall intercept the message).

The terminal bridge that intercepts a TIMEOUT message shall transmit a response message encapsulated in a block write request addressed to the requester's MESSAGE_RESPONSE register; the *snarf* field in the packet header shall have a value of three (all bridges on the route shall intercept the message) and the *result* field shall be zero. If *destination_ID* does not contain a valid global node ID or the least significant six bits of the TIMEOUT request message *destination_ID* are equal to $3F_{16}$, the *responder_EUI_64* field in the response shall be zero. Otherwise, if the global node ID contained in *destination_ID* is valid, the *responder_EUI_64* field shall be set to the EUI-64 of the addressed node.

NOTE—If the *destination_ID* does not contain a valid global node ID, the accumulated time-out values are valid for other nodes on the bus.

6.6.2 TIME OFFSET message

The TIME OFFSET message is used to calculate the relative time difference, in cycles, between two buses in a net. The format of the *opcode_dependent* field in this message is illustrated by Figure 6-8.

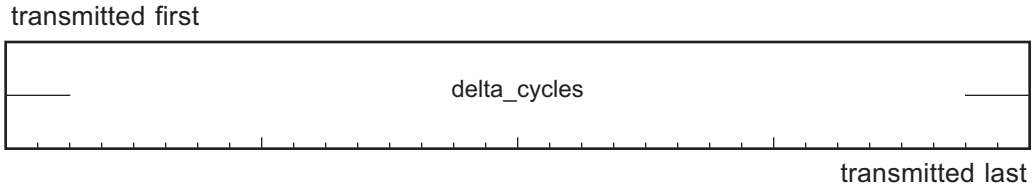


Figure 6-8 — TIME OFFSET message format

The ***delta_cycles*** field is a 64-bit signed integer in two's-complement notation that represents the time difference, in cycles, between the requester's bus and the bus identified by the most significant ten bits of *destination_ID* in the TIME OFFSET message. In order to update *delta_cycles*, local time on a particular bus is converted to an unsigned count of cycles: 8000 * BUS_TIME + CYCLE_TIME.*cycle_count*.

The originator of a TIME OFFSET request message shall zero the *delta_cycles* field before transmission.

A TIME OFFSET request message shall be encapsulated as a block write request whose *destination_offset* field addresses the MESSAGE_REQUEST register. The most significant ten bits of *destination_ID* shall identify the other bus and shall not be equal to $3FF_{16}$. The least significant six bits of *destination_ID* shall be equal to $3F_{16}$. The *snarf* field in the packet header shall have a value of three (all bridges on the route shall intercept the message).

Bridges that intercept a TIME OFFSET shall update *delta_cycles* to reflect the time difference between their portals (see 8.2 for details). As a consequence, *delta_cycles* accumulates the difference between the originator's bus and the terminal bus addressed by *destination_ID*. The terminal bridge, after updating *delta_cycles*, shall zero the *snarf* bit and transmit the TIME OFFSET response message to the requester's MESSAGE_RESPONSE register.

6.6.3 Stream management messages

The stream management messages (JOIN, LEAVE, LISTEN, RENEW, TEARDOWN, and STREAM STATUS) share a common message format, whose *opcode_dependent* portion is illustrated by Figure 6-9. Not all fields are relevant to all messages. The details of stream setup and teardown operations are beyond the scope of this clause, which specifies packet formats. See 4.6 for an overview of the process and 8.6 for normative information.

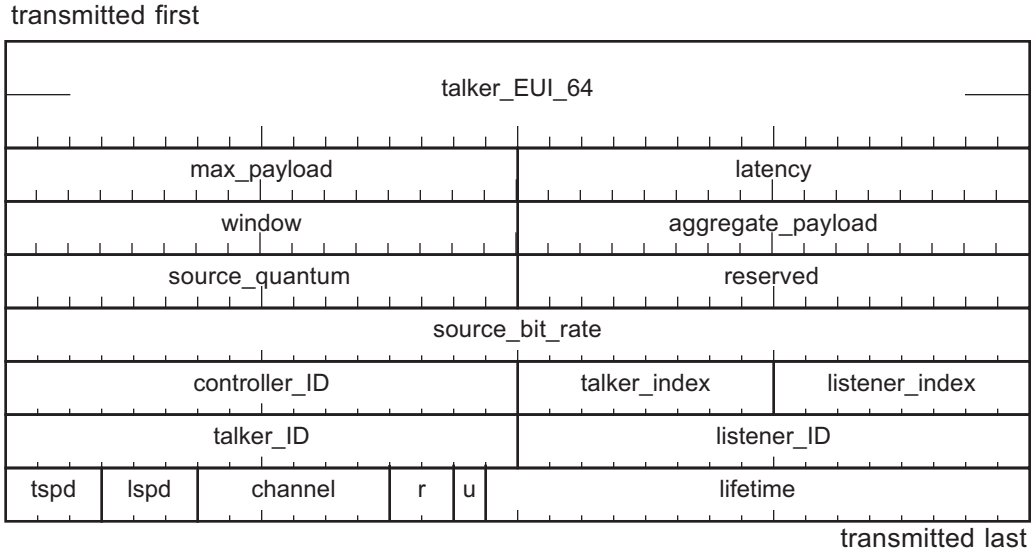


Figure 6-9 — Stream management message format

The ***talker_EUI_64*** field shall contain the talker's EUI-64, obtained from its bus information block. The stream to which the stream management message pertains is uniquely identified by the combination of *talker_EUI_64* and *talker_index*, referred to as the stream ID. A stream ID is globally unique and is persistent across bus resets and net topology changes.

The ***max_payload*** field shall specify the maximum number of data bytes that the talker may transmit in a single isochronous stream packet. The value of *max_payload*, in combination with other information such as the local bus topology, shall determine the bandwidth to be allocated from the BANDWIDTH_AVAILABLE register.

NOTE—The value of *max_payload* does not include the isochronous header, header CRC or data CRC that are part of an isochronous stream packet; it counts only those bytes that are part of the data payload.

The ***latency*** field shall contain the sum of the isochronous delays, in units of 125 μs, encountered by an isochronous stream originated by the talker and destined for the listener. This field is updated by each listening portal on the path between the talker and listener. A portal whose configuration ROM Bridge_Capabilities entry *latency* field (see 5.1.4) is nonzero shall increase accumulated latency by its value. Otherwise, the portal shall increase accumulated latency by the value determined when the stream was set up.

The ***window*** field specifies the size of a smoothing window, in units of 125 μs. The value of *window*, when combined with the maximum aggregate payload, could permit a bridge to determine if it is capable of transporting an isochronous stream whose maximum payload exceeds the bandwidth available to one or both of the bridge's portals during a single 125 μs interval. The implementation details of such a bridge are beyond the scope of this standard. When *window* is zero, no aggregate payload information is provided.

The meaning of the ***aggregate_payload*** field is unspecified when *window* is zero. Otherwise, the aggregate payload, in bytes, of the isochronous stream in any arbitrarily chosen smoothing window of *window* consecutive isochronous periods shall not exceed the value of *aggregate_payload*.

NOTE—A *window* value of either zero or one describes the same situation: a smoothing window of 125 μs. The only difference is that a *window* value of one obligates the values of *aggregate_payload* and *max_payload* to be identical.

The ***source_quantum*** field specifies the smallest indivisible data size, in bytes, generated by the source of the isochronous stream; it shall include any encoding format overhead but shall exclude the isochronous header, header CRC, or data CRC that are part of an isochronous stream packet. When *source_quantum* is zero, no information is provided.

The meaning of the ***source_bit_rate*** field is unspecified when *source_quantum* is zero. Otherwise, it shall specify the source's bit rate, in bits per second; it shall include any encoding format overhead but shall exclude the isochronous header, header CRC, or data CRC that are part of an isochronous stream packet.

NOTE—For example, the source quantum for an MPEG transport stream encoded per IEC 61883-4 is 192 bytes, comprised of a 188-byte transport stream packet (TSP) prefixed by a 4-byte source packet header. If the transport stream's bit rate were 19.4 Mb/s, the value of *source_bit_rate* would be 19.8 Mb/s in order to include the source packet header overhead.

The ***controller_ID***, ***talker_ID,*** and ***listener_ID*** fields shall specify the global node IDs of the controller, talker, and listener, respectively. The *controller_ID* field shall be initialized to $FFFF_{16}$ by the originator of the message and shall be updated to contain the global node ID of the controller by the first bridge portal that intercepts the message. The *source_ID* field in the packet header of the block write request that encapsulates a stream management message shall specify the node ID of the transmitter of the subaction.

The ***talker_index*** field shall uniquely identify a stream transmitted by the talker within the context of the talker itself. When *talker_index* is in the range zero to $1E_{16}$, inclusive, it identifies a talker output plug control register (oPCR) that mediates transmission of the stream. Other values of *talker_index* are reserved for future standardization.

The ***listener_index*** field shall uniquely identify a stream received by the listener within the context of the listener itself. When *listener_index* is in the range zero to $1E_{16}$, inclusive, it identifies a listener input plug control register (iPCR) that mediates reception of the stream. Other values of *listener_index* are reserved for future standardization.

NOTE—Output and input plug control registers are specified by IEC 61883-1.

The ***tspd*** and ***lspd*** fields encode the maximum speed at which the talker and the listener may transmit and receive, respectively, as specified by the table below.

Value	Speed
0	S100
1	S200
2	S400
3	S800
4	S1600
5	S3200
6 - 7	Reserved

The ***channel*** field shall specify the channel used for the stream. This field is meaningful only for the LISTEN message.

When the ***upstream*** bit (abbreviated as *u* in Figure 6-9) is zero, the stream management message is propagated away from the talker identified by *talker_ID* (downstream); otherwise, the message is propagated towards the talker (upstream). This field is meaningful only for the TEARDOWN message.

The ***lifetime*** field shall specify the time remaining, in seconds, before the reallocation proxy on the listener's bus shall tear down the stream connection for the listener.

6.6.3.1 JOIN message

A controller originates a JOIN request message to establish a stream connection between a talker and listener in cases where the controller, talker, and listener are not connected to the same bus. If the listener is remote with respect to the controller, the JOIN request message shall be encapsulated within a block write request addressed to the listener's MESSAGE_REQUEST register. Otherwise, it shall be addressed to the talker's MESSAGE_REQUEST register.

The controller shall initialize the *requester_EUI_64* field with its own EUI-64 and shall zero the *responder_EUI_64* field. It shall also initialize the *talker_EUI_64*, *max_payload*, *controller_ID*, *talker_index*, *listener_index*, *talker_ID*, *listener_ID*, *tspd*, *lspd,* and *lifetime* fields. If the information is available, the controller should provide either *source_quantum* and *source_bit_rate* (recommended) or *window* and *aggregate_payload*. The contents of the *latency* and *channel* fields are unspecified and shall be ignored.

The expected response to a JOIN request message is a STREAM STATUS response message transmitted to the controller when stream setup is complete or an error has been encountered.

6.6.3.2 LEAVE message

A controller originates a LEAVE request message to terminate a stream connection between a talker and listener in cases where the controller, talker, and listener are not connected to the same bus. If the listener is remote with respect to the controller, the LEAVE request message shall be encapsulated within a block write request addressed to the listener's MESSAGE_REQUEST register. Otherwise, it shall be addressed to the talker's MESSAGE_REQUEST register.

The controller shall initialize the *requester_EUI_64* field with its own EUI-64 and shall zero the *responder_EUI_64* field. It shall also initialize the *talker_EUI_64*, *controller_ID*, *talker_index*, *listener_index*, *talker_ID,* and *listener_ID* fields to their appropriate values. The contents of the *max_payload*, *latency*, *window*, *aggregate_payload*, *source_quantum*, *source_bit_rate*, *tspd*, *lspd*, *channel,* and *lifetime* fields are unspecified and shall be ignored.

The expected response to a LEAVE request message is a STREAM STATUS response message transmitted to the controller's MESSAGE_RESPONSE register when stream teardown is complete or an error has been encountered.

6.6.3.3 LISTEN message

The LISTEN message is used solely by a talking bridge portal to communicate the channel number and speed used for a stream on the talking portal's local bus to another portal on the local bus that intends to listen and forward the stream. A LISTEN message shall be encapsulated within a block write request addressed to the listener's MESSAGE_REQUEST register; the *snarf* field shall be three (intercepted by all bridges).

The listening portal on the talker's bus shall copy the *requester_EUI_64, talker_EUI_64*, *controller_ID*, *talker_index*, *listener_index*, *channel*, *talker_ID*, *listener_ID*, *lspd,* and *lifetime* fields from the JOIN message and shall initialize the *latency* field with the value obtained from the portal's configuration ROM Bridge_Capabilities entry before transferring the LISTEN message to its co-portal. The contents of the *max_payload*, *window*, *aggregate_payload*, *source_quantum*, *source_bit_rate*, *tspd,* and *channel* fields are unspecified and shall be ignored.

A talking portal that transmits a LISTEN message to another portal expects no response message. After the portal connected to the listener's bus receives and processes a LISTEN message, it sends a STREAM STATUS response to the controller.

6.6.3.4 RENEW message

A controller originates a RENEW request message to extend the remaining stream connection lifetime for a particular listener. If the listener is remote with respect to the controller, the RENEW request message shall be encapsulated within a block write request addressed to the listener's MESSAGE_REQUEST register. Otherwise, it shall be addressed to the talker's MESSAGE_REQUEST register.

The controller shall initialize the *requester_EUI_64* field with its own EUI-64 and shall zero the *responder_EUI_64* field. It shall also initialize the *talker_EUI_64*, *controller_ID*, *talker_index*, *listener_index*, *talker_ID*, *listener_ID,* and *lifetime* fields. The contents of the *max_payload*, *latency*, *window*, *aggregate_payload*, *source_quantum*, *source_bit_rate*, *tspd*, *lspd,* and *channel* fields are unspecified and shall be ignored.

The expected response to a RENEW request message is a STREAM STATUS response message transmitted to the controller's MESSAGE_RESPONSE register once the remaining stream connection lifetime has been updated or an error has been encountered.

6.6.3.5 TEARDOWN message

The TEARDOWN message is an inter-portal message that causes the teardown of all or part of a stream whose path has been severed by the disconnection of one or more nodes. Only the *requester_EUI_64*, *talker_EUI_64*, *talker_index,* and *talker_ID* fields and the *upstream* bit are meaningful. The contents of the other fields are unspecified and shall be ignored.

6.6.3.6 STREAM STATUS message

The STREAM STATUS response message is used to communicate to the controller the result of an earlier JOIN, LEAVE, or RENEW request. The message shall be encapsulated in a block write request addressed to the controller's MESSAGE_RESPONSE register. The *snarf* field shall be zero.

The only fields valid in the STREAM STATUS response message are *requester_EUI_64*, *talker_EUI_64*, *talker_index*, *latency,* and *result*.

NOTE—Receipt of a STREAM STATUS response message indicates only the status of the stream's route from the talker to the listener (both identified by an earlier JOIN, LEAVE, or RENEW request). Once a path exists, the means by which the talker or listener are commanded to start or stop stream transmission or reception are beyond the scope of this standard.

6.6.4 Net update messages

The MUTE, BUS ID, BUS ID ANNOUNCEMENT, and PANIC net management messages are used exclusively by bridge portals during and after net update. The UPDATE ROUTES message is also used exclusively by bridge portals during net update, but because its format differs from net management messages, it is described separately.

6.6.4.1 MUTE message

The MUTE message instructs the recipient to disable all asynchronous and stream subaction routing functions between itself and its co-portal and to update the route maps of both itself and its co-portal. A MUTE message does not contain an *opcode_dependent* field. A MUTE request shall be encapsulated as a block write request addressed to a portal's MESSAGE_REQUEST register. The *snarf* field in the packet header shall be zero.

6.6.4.2 BUS ID message

The BUS ID message is used either to request or receive a bus ID assignment from the prime portal (see 11.2); its usage depends on whether the message is addressed to the MESSAGE_REQUEST or MESSAGE_RESPONSE register, respectively. In both cases, the *snarf* field shall be zero. The format of the *opcode_dependent* field in a BUS ID message is illustrated by Figure 6-10.

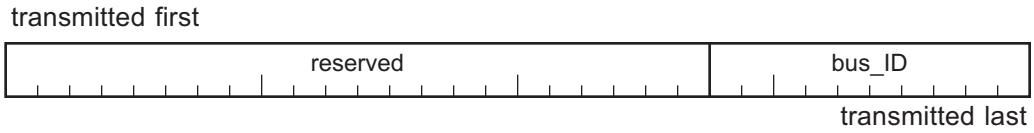


Figure 6-10 — BUS ID message format

The contents of the ***bus_ID*** field depend upon the *destination_offset* of the message. In a BUS ID request message, *bus_ID* shall contain the originating alpha portal's EUI-64 modulo 1023. Otherwise, in a BUS ID response message it shall contain the bus ID assigned by the prime portal; $3FF_{16}$ indicates no bus ID is available.

A BUS ID request message shall be encapsulated in a block write request addressed to the MESSAGE_REQUEST register of an alpha portal.

A BUS ID response message transmitted by the prime portal shall be encapsulated in a block write request addressed to the requester's MESSAGE_RESPONSE register.

6.6.4.3 BUS ID ANNOUNCEMENT message

The BUS ID ANNOUNCEMENT message informs bridge portals of a newly assigned bus ID so that they can adjust the corresponding entry in their route maps to FORWARD or VALID, as appropriate. The format of the *opcode_dependent* field in a BUS ID ANNOUNCEMENT message is illustrated by Figure 6-11.

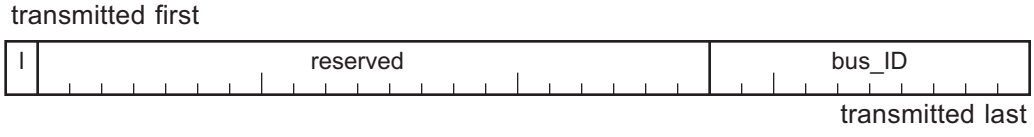


Figure 6-11 — BUS ID ANNOUNCEMENT message format

The ***local_bus*** bit (abbreviated as *l* in Figure 6-11), when one, indicates that the recipient's CLAN_INFO.*bus_ID* field shall be updated with the value of the *bus_ID* field in the message. Otherwise, when it is zero, the CLAN_INFO register shall not be modified. See 11.2 for more information.

The ***bus_ID*** field shall contain a bus ID assigned by the prime portal by means of a BUS ID response message and shall not be equal to $3FF_{16}$.

6.6.4.4 PANIC message

The PANIC message instructs the recipient to participate in the net panic process. Net panic is invoked if net update enters a pathological state that never completes by itself. Net panic is an extreme remedy that causes all bridge portals to cease functioning as bridge portals before performing the equivalent of power reset initialization and subsequently resuming bridge operations. The format of the *opcode_dependent* field in a PANIC message is illustrated by Figure 6-12.

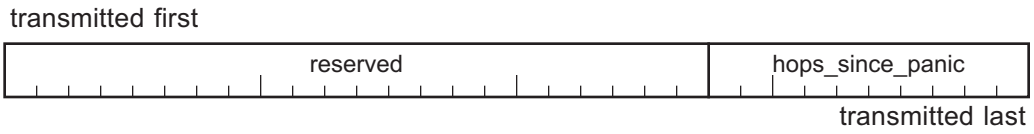


Figure 6-12 — PANIC message format

The ***hops_since_panic*** field counts the number of bridges traversed by the PANIC message since it was originated by the bridge portal that initiated net panic. The field shall be equal to the value of the *hops_since_panic* variable of the portal that transmits the message. See 10.6 for net panic details.

6.7 UPDATE ROUTES message

The coordinator uses the UPDATE ROUTES message to communicate updated clan allegiance or routing information to other portals on the bus during net update. The format of the message is illustrated by Figure 6-13.

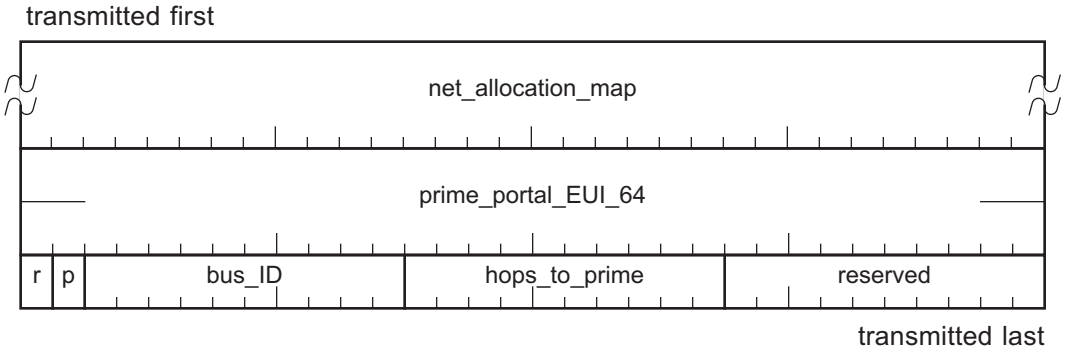


Figure 6-13 — UPDATE ROUTES message format

The ***net_allocation_map*** field has the same format as the ROUTE_MAP register described in 5.2.3. It consists of 1024 two-bit entries, each of which represents the state of the corresponding bus ID within the clan. An entry shall have a value of CLEAN, DIRTY, or VALID as specified by Table 5-3; the FORWARD value defined for the ROUTE_MAP register is not used.[14] Although the local bus ID, $3FF_{16}$, needs no entry because it is not routable, its space in the array is allocated for the sake of simplicity.

[14] The routing distinction between VALID and FORWARD, which is explicitly encoded within a portal's ROUTE_MAP register, is implicitly encoded by the context of the UPDATE ROUTES message. See Table 10-9 for details.

The ***prime_portal_EUI_64*** field indicates the clan allegiance and, in an UPDATE ROUTES message transmitted from the coordinator to another portal, shall be equal to the survivor clan's alpha portal CLAN_EUI_64 register.

In an UPDATE ROUTES message transmitted from the coordinator to another portal, the ***preferred_clan*** (abbreviated as *p* in Figure 6-13) bit and the ***bus_ID*** and ***hops_to_prime*** fields shall be equal to the same fields in the survivor clan's alpha portal CLAN_INFO register.

When an UPDATE ROUTES message is communicated from the coordinator to another bridge portal on the local bus, it shall form the data payload of a block write request addressed to a portal's ROUTE_MAP register. The *data_length* shall be 268; this completely overlaps both the CLAN_EUI_64 and CLAN_INFO registers adjacent to the ROUTE_MAP register. Otherwise, when the UPDATE ROUTES message is communicated from a bridge portal to its co-portal, it shall consist of the 268 bytes above delivered by implementation-dependent means.

7. Transaction routing and operations

This subclause specifies the behavior of bridge portals when they receive or transmit request and response subactions during normal operations.[15] All of these subactions are routed according to the value of *destination_ID* in their packet headers; Table 7-1 summarizes the eligible transaction codes (*tcode*).

Table 7-1 — Transaction codes routed by *destination_ID*

tcode	Description
0	Quadlet write request
1	Block write request
2	Write response
4	Quadlet read request
5	Block read request
6	Quadlet read response
7	Block read response
9	Lock request
B_{16}	Lock response

A remote transaction is always a split transaction; the progress of its request and response subactions is divided into three phases: origination of the subaction on the source bus, propagation of the subaction across zero or more intermediate buses, and delivery of the subaction on the destination bus. Bridge portal treatment of request and response subactions is fundamentally similar; differences between the two are noted in the subclauses that follow.

7.1 Source bus (initial entry portal)

A remote subaction from a bridge-aware node that is not also a bridge portal originates in the same fashion as any other local subaction on the transmitter's bus—the only difference is that the *destination_bus_ID* specifies a bus ID other than $3FF_{16}$. A remote subaction originated by a bridge portal, dependent upon its *destination_ID*, may be processed internally and never appear on the local bus; this is described in more detail later. When a subaction is transmitted on the local bus, it is observed by all bridge portals; no more than one bridge portal connected to the bus shall respond to a subaction whose *source_bus_ID* is equal to $3FF_{16}$ and whose *destination_bus_ID* is not equal to $3FF_{16}$, as specified by Table 7-2.

Table 7-2 — Bridge-bound routing from the source bus

destination_bus_ID	Routing state	Comment
Not equal to CLAN_INFO.*bus_ID*	CLEAN DIRTY	All portals except the alpha portal ignore the subaction. The alpha portal transmits *ack_complete* for responses and *ack_pending* (and a subsequent error response whose *ext_rcode* specifies an invalid bus ID) for requests.
	FORWARD	If a bridge-aware node originated the subaction, it is forwarded by this portal, which transmits *ack_pending* for requests or *ack_complete* for responses and forwards the subaction to its co-portal (after first transforming *source_ID* into a global node ID). Special processing when the subaction originator is not bridge-aware is described later in this subclause.
	VALID	The subaction is ignored by this portal, since another portal is expected to transmit an acknowledgement.

[15] The phrase "normal operations" does not apply to those times (*e.g.*, immediately after a bus reset or during net update) when bridge reconfiguration is in progress; consult the appropriate later clauses for descriptions of bridge behavior at these times.

Table 7-2 — Bridge-bound routing from the source bus *(continued)*

destination_bus_ID	Routing state	Comment
Equal to CLAN_INFO.*bus_ID*	—	All portals except the alpha portal shall ignore the subaction. Subactions addressed to the alpha portal itself are indicated to the application layer. Otherwise, the alpha portal transmits *ack_complete* for responses and *ack_pending* for requests and subsequently queues the subaction for retransmission on the local bus. In both cases, the alpha portal first transforms *source_ID* into a global node ID and *destination_ID* into a local node ID.

In Table 7-2, CLAN_INFO.*bus_ID* contains the bus ID of the portal's bus. The routing state is obtained from the initial entry portal's ROUTE_MAP register (indexed by *destination_bus_ID*) and is one of the four possible states enumerated by Table 5-3.

When the originator of a request subaction is not bridge-aware, the portal on the source bus shall forward the subaction only if it is a write request. If the subaction is eligible to be forwarded (or echoed onto the local bus) after global node ID transformation of the *source_ID*, the bridge portal shall synthesize a write response packet that indicates *resp_complete* and return it to the requester.[16] This is called a "posted" write, since the originator receives successful completion status before the write request is received at its destination. The original write request is forwarded (or echoed), as appropriate, but there is no guarantee it reaches its intended destination.

All other request subactions transmitted by a node that is not bridge-aware shall be rejected with *resp_type_error* and an extended response code of *ext_legacy_quarantine*. Response subactions transmitted by nodes unaware of bridges are forwarded according to the eligibility requirements of Table 7-2 and require no other special processing.

When a bridge portal originates a remote subaction, it shall inspect the *destination_ID*. If the portal's route map indicates FORWARD for the subaction's *destination_bus_ID*, the subaction shall be transferred to the co-portal (after its *source_ID* has been transformed to a global node ID), in which case the application within the bridge portal that originated the subaction shall receive an LK_DATA.*confirmation* of either *ack_pending* or *ack_complete*, as appropriate. Whether the subaction appears on the local bus is implementation-dependent. For subactions not transferred to the co-portal, if the subaction's *destination_bus_ID* is not equal to CLAN_INFO.*bus_ID*, the portal shall transmit the subaction on the local bus and another portal processes the subaction as described in Table 7-2. When the subaction's *destination_bus_ID* and CLAN_INFO.*bus_ID* are equal, the disposition of the subaction depends upon whether the portal is alpha or subordinate. Subordinate portals shall transmit the subaction unaltered on the local bus but an alpha portal shall first transform the *destination_ID* from a global to a local node ID before transmitting the subaction.

7.2 Intermediate buses

Packets in transit on intermediate buses are identified by the presence of a global node ID in both *source_ID* and *destination_ID*. The subaction's original *source_ID* was transformed to a global node ID by the initial entry portal that first snooped the packet from the source bus, while the global *destination_ID* supplied by the originator has not yet been transformed to a local node ID by the terminal exit portal connected to the destination bus.

A transient packet can be processed either as a bus-bound subaction (as a consequence of the receipt of the packet from the co-portal) or as a bridge-bound subaction (as a result of snooping on packets on the local bus). These two roles are described separately in the clauses that follow.

[16] In lieu of *ack_pending* and a subsequent response packet that indicates *resp_complete*, a bridge portal may return *ack_complete* in response to a write request from a node that is not bridge-aware.

7.2.1 Packet reception (intermediate entry portal)

The behavior of bridge portals that snoop packets in transit is fundamentally similar to that already described in 7.1, except that they are concerned solely with packets whose *source_bus_ID* is not equal to $3FF_{16}$ and whose *destination_bus_ID* is not equal to the bus ID assigned to the portal's bus by the prime portal. For packets that meet those requirements, Table 7-3 specifies the bridge-bound behavior of a portal on an intermediate bus.

Table 7-3 — Bridge-bound routing from an intermediate bus

destination_bus_ID	Routing state	Comment
Not equal to CLAN_INFO.*bus_ID*	CLEAN DIRTY	All portals except the alpha portal ignore the subaction. The alpha portal transmits *ack_complete* for responses and *ack_pending* (and a subsequent error response whose *ext_rcode* specifies an invalid bus ID) for requests.
	FORWARD	Transmit *ack_pending* for requests or *ack_complete* for responses and forward the subaction to the co-portal.
	VALID	The subaction is ignored by this portal, since another portal is expected to transmit an acknowledgement.
Equal to CLAN_INFO.*bus_ID*	—	Should not occur when a packet is snooped from an intermediate bus.

Just as was the case for snooped packets on their source bus, CLAN_INFO.*bus_ID* contains the bus ID of the portal's bus. The routing state is obtained from the intermediate entry portal's ROUTE_MAP register (indexed by *destination_bus_ID*) and is one of the four possible states enumerated by Table 5-3.

The preceding describes cases in which snooped packets are received without data error. When a subaction is received with data payload error, the intermediate entry portal shall transmit a busy acknowledgment.

7.2.2 Packet transmission (intermediate exit portal)

Packets forwarded from the co-portal have already been screened for a valid *destination_bus_ID* and require no further validation or transformation before retransmission on an intermediate bus: both the *source_ID* and *destination_ID* fields contain global node IDs. Since the bridge portal might not be aware of the bus topology between itself and the next bridge portal to snoop and forward the subaction, the transmission speed is usually the fastest speed common to all paths between all bridge portals on the bus. The bridge portal shall select the transmission speed based upon the subaction's *destination_bus_ID*.[17] For request subactions, if the *data_length* field is too large for transmission at the speed selected or, for all subactions, if the packet data payload is too large, the portal shall discard the packet and synthesize a response of (or, in the case of a response subaction, modify the response to) *resp_type_error* with an extended response code of *ext_payload_too_big*.

The expected acknowledgement is *ack_pending* for request subactions and *ack_complete* for response subactions. When either of these is received (or if no acknowledgement is received), the bridge portal shall discard the corresponding subaction, which is now entrusted to the next bridge portal on the route. Even in the case of a missing acknowledgment, the next bridge portal is assumed to have received the subaction, since only the acknowledgment might have been garbled.

If a busy acknowledgment is received for a subaction, the bridge portal shall queue the subaction for retransmission until the maximum forwarding time expires. A busy acknowledgment is terminal only after time runs out.

[17] Bridge portals should maintain a table that encodes, for all 1023 possible bus IDs, the maximum transmission speed for subactions in transit routed to a particular bus. Net update resets all of the table's entries to S100. Whenever a bridge portal subsequently intercepts a TIMEOUT message, it sets the entry that corresponds to the *source_bus_ID* of the TIMEOUT message to the maximum speed supported by the bus topology that separates the two portals (see 9.4 for details).

An error might occur in the attempt to forward the subaction to the next bridge portal, either because of traffic congestion or the receipt of a terminal acknowledgement from the receiving portal. If a nonrecoverable error occurs in the transmission of a response subaction, the portal discards it without further action. For request subactions that encounter nonrecoverable errors, however, the bridge portal shall synthesize a response packet in accordance with Table 7-4.

Table 7-4 — Synthesized responses on intermediate buses

Acknowledge code	Response code	Extended response code	Comment
ack_busy_X *ack_busy_A* *ack_busy_B*	*resp_conflict_error*	Copy the numeric value of the acknowledge code to the *ext_rcode* field in the response.	For busy acknowledgments, this is a nonrecoverable error only if insufficient time remains to queue the subaction for retransmission before the maximum forwarding time expires.
ack_address_error	*resp_address_error*		
ack_data_error	*resp_data_error*		The subaction was received as a malformed packet by the next portal (invalid data payload length or data CRC).

The bridge portal that detects the error shall set *proxy_ID* in the synthesized response to the value of its own global node ID. Note that the next bridge portal, by design, will never return certain acknowledge codes: address errors do not exist except in the destination node and the initial entry portal that first snooped the subaction screened it for type errors.

7.3 Destination bus (terminal exit portal)

The terminal exit portal, the one connected to the same bus as the node identified by *destination_ID*, is responsible to translate the global node ID in the *destination_ID* field into the corresponding local node ID for the addressed node before transmitting the subaction on its destination bus.[18] The behavior of an exit portal on the terminal bus is essentially similar to that already described in 7.2.2 for packet transmission on intermediate buses. If a busy acknowledgement is received, the terminal exit portal shall queue the subaction for retransmission until the maximum forward time is exhausted. The expected acknowledgement for a request subaction is either *ack_complete* or *ack_pending*. In both cases, the request subaction is discarded; however, in the first case, the terminal exit portal creates a response subaction that indicates *resp_complete* and starts it on its return journey. When a request subaction is acknowledged by *ack_pending*, the recipient of the request subaction is expected to create the response. Response subactions are discarded upon receipt of a terminal acknowledgement, whether it is *ack_complete* or not. If no acknowledgement is received, the terminal exit portal shall discard the subaction and, in the case of request subactions, shall not synthesize a response.

When an error occurs in the attempted delivery of a request subaction, either because the maximum time allotted to attempt delivery of the subaction expired or the destination node returned a terminal acknowledgement, the terminal exit portal synthesizes a response packet. Table 7-5 summarizes all of the possible responses; it is more fully populated than Table 7-4 because destination nodes are permitted a larger repertoire of acknowledgements than are intermediate bridge portals.

[18] If the terminal exit portal itself is addressed by *destination_ID*, no subactions occur on the local bus. In this case, *destination_ID* is translated to the portal's local node ID; from then on the subaction is processed as if it had been received from the bus. For request subactions, the acknowledgment provided by LK_DATA.*response* is processed according to Table 7-5 as if it also had been received from the bus.

Table 7-5 — Synthesized responses on destination buses

Acknowledge code	Response code	Extended response code	Comment
ack_complete	*resp_complete*	Zero	Normal response when the destination node successfully completes the request subaction as a unified transaction.
ack_busy_X *ack_busy_A* *ack_busy_B*	*resp_conflict_error*	Copy the numeric value of the acknowledge code to the *ext_rcode* field in the response.	This is a nonrecoverable error only if insufficient time remains to queue the subaction for retransmission before the maximum forwarding time expires.
ack_tardy *ack_conflict_error*			Although a subsequent retry might meet with success, the time it takes a node's link layer to become responsive (in the case of *ack_tardy*) or the time after which resources might be available (in the case of *ack_conflict_error*) is indeterminate.
ack_data_error	*resp_data_error*		Nonrecoverable errors that terminate the transaction.
ack_type_error	*resp_type_error*		
ack_address_error	*resp_address_error*		

The terminal exit portal shall insert its own global node ID in the response packet's *proxy_ID* field.

7.4 Maximum forward time

Whenever a bridge portal on an intermediate bus or the destination bus attempts to transmit a subaction to the next portal or destination node, respectively, it might encounter recoverable errors. These include busy conditions, data errors detected by the recipient, and lack of resources (congestion) at the recipient. A bridge portal that encounters a potentially recoverable error shall queue the subaction for retransmission until either it is correctly acknowledged by the recipient, a nonrecoverable error occurs, or the maximum forward time limit expires. The maximum forward time is vendor-dependent, may differ for request and response subactions, and may be different for each of a bridge's portals, subject to the constraints of Table 7-6.

Table 7-6 — Maximum forward time for asynchronous subactions

Subaction type	Minimum	Maximum
Request	100 ms	8 s
Response	300 ms	

For subactions without a *snarf* field in the packet header or whose *snarf* field is zero, a bridge shall start timing the maximum forward interval for a subaction when the entry portal transmits a pending acknowledgement after the subaction's receipt. The exit portal shall not transmit a pending asynchronous subaction after maximum forward time has expired.

Asynchronous stream subactions are subject to essentially the same requirements as acknowledged subactions; the difference is that there is no acknowledgment to use in as a reference point. For a received asynchronous stream packet, a bridge commences to time the maximum forward interval when the entry portal receives the subaction. The exit portal shall persist in attempts to transmit the pending asynchronous stream subaction so long as it is possible for arbitration to be granted before the expiration of the maximum forward time.

Block write request subactions whose *snarf* field is nonzero are subject to different rules. The maximum forward interval shall be equal to the portal's maximum response forward time, *i.e.*, a minimum of 300 ms. The exit portal shall time the maximum forward interval from the time the subaction is first placed in the portal's transmit queue and shall persist in attempts to transmit the pending subaction so long as it is possible for arbitration to be granted before the expiration of the maximum response forward time.

7.5 Congestion management

Congestion in a bridge occurs when an exit portal is unable to transmit asynchronous subactions as rapidly as they are received from its co-portal. Eventually, the bridge's internal resources are completely occupied by subactions awaiting transmission and either or both portals are unable to receive new asynchronous subactions. Because the root causes of congestion on intermediate buses are likely to be restricted to a subset of possible entry portals, an exit portal retry strategy that temporarily bypasses busied subactions could permit forward progress for other subactions routed through entry portals that are not congested.

Bridge portals shall implement inbound and outbound dual-phase retry protocol; bridge-aware nodes should implement inbound and outbound dual-phase retry protocol. Exit portals should use the following strategy for transmission from their separate request and response subaction queues:

— Within a fairness interval, an exit portal should attempt to transmit as many request subactions as permitted by its priority arbitration budget. If a request subaction receives a busy acknowledgment, the portal should attempt transmission of other request subactions—but set their *rt* field (retry code) to *retry_X*. The portal should continue to transmit request subactions until its queue is empty or its priority arbitration budget is exhausted, at which time it should switch to transmitting response subactions.
— An exit portal should attempt to transmit all of its response subactions until the queue is empty or a response subaction receives a busy acknowledgment, at which time it should switch to transmitting request subactions.

This exit portal transmit behavior is specified in more detail by Table 7-7:

Table 7-7 — Request and response subaction transmission by an exit portal (Sheet 1 of 2)

```
#include "global.h"
#include "csr.h"
#include "packets.h"

typedef struct qblk {         /* Data structure for request / response queues */
   struct qblk *next;
   PACKET *packet;
} QBLK;

PACKET *oldestRequest;
PACKET *oldestResponse;
BYTE priorityBudget;          /* Reset to value of PRIORITY_BUDGET upon reset gap */
QBLK *requestQueue;           /* Separate request ... */
QBLK *responseQueue;          /* ... and response queues for nonblocking behavior */

VOID exitPortalTransmit() {

   BYTE ack;
   QBLK *prevQBlk, *qBlk;
   PACKET *request, *response;

   while (TRUE) {                        /* Forever alternate between queues */
     prevQBlk = (QBLK *) &requestQueue;  /* Attempt transmission of requests (up to PRIORITY_BUDGET) */
      while ((qBlk = prevQBlk->next) != NULL && priorityBudget > 0) {
         request = qBlk->packet;
         if (oldestRequest == NULL) {
            oldestRequest = request;
            request->rt = RETRY_1;        /* Attempt outbound dual-phase retry */
         } else if (request != oldestRequest)
            request->rt = RETRY_X;        /* Restricted to outbound single-phase retry */
         ack = transmitPacket(request);
         priorityBudget--;                /* Consumed one priority arbitration */
         if (ack == ACK_BUSY_A)           /* Adjust retry code to match acknowledgment */
            request->rt = RETRY_A;
         else if (ack == ACK_BUSY_B)
            request->rt = RETRY_B;
         else if (ack == ACK_PENDING) {
```

Table 7-7 — Request and response subaction transmission by an exit portal (Sheet 2 of 2)

```
            if (request == oldestRequest)
               oldestRequest = NULL;      /* Enable promotion of next oldest request */
            prevQBlk->next = qBlk->next;  /* Done with this request */
         }
      }
      prevQBlk = (QBLK *) &responseQueue; /* Attempt transmission of all responses */
      while ((qBlk = prevQBlk->next) != NULL) {
         response = qBlk->packet;
         if (oldestResponse == NULL) {
            oldestResponse = response;
            response->rt = RETRY_1;       /* Attempt outbound dual-phase retry */
         } else if (response != oldestResponse)
            response->rt = RETRY_X;       /* Restricted to outbound single-phase retry */
         ack = transmitPacket(response);
         if (ack == ACK_BUSY_A) {         /* Adjust retry code to match acknowledgment */
            response->rt = RETRY_A;
            break;                        /* Switch to requests */
         } else if (ack == ACK_BUSY_B) {
            response->rt = RETRY_B;
            break;
         } else if (ack == ACK_COMPLETE) {
            if (response == oldestResponse)
               oldestResponse = NULL;     /* Enable promotion of next oldest response */
            prevQBlk->next = qBlk->next;  /* Done with this response */
         }
      }
   }

}
```

Enhanced implementations may improve upon the retry strategy described by Table 7-7. For example, if an exit portal can determine the next entry portal from a subaction's *destination_ID*, it might skip over subactions bound for congested entry portals and instead transmit local subactions and subactions bound for entry portals that are not congested. Also, dual-phase retry protocol is not limited to a single "oldest" request or response subaction; there may be multiple request and response subactions with a retry code of *retry_1* so long as there is no more than one such request and one such response outstanding for an individual local node. Bridge portals should also provide sufficient resources for estimated congestion, *e.g.,* support at least 64 dual-phase retry reservations (an average allowance of at least one reservation per node).

8. Stream operations and routing

This clause describes the normal operations of a bridge to route GASP on the default broadcast channel as well as to route isochronous stream data once an application has configured intervening bridges to support an end-to-end path between a talker and a listener.

All bridges shall support the distribution of a synchronized cycle time and the routing of GASP subactions throughout the Serial Bus net. Bridges that support isochronous streams shall be able to recognize explicitly routed stream subactions (identified by their channel number) at either portal and retransmit the stream subaction (with a remapped channel number) from the co-portal.

A bridge portal functions as a ***listening portal*** when it snoops its local bus[19] to receive stream subactions to be forwarded to its co-portal. A portal that transmits a stream subaction (received from its co-portal) on its bus acts as a ***talking portal***.

The subclauses that follow describe cycle time synchronization and the algorithms that govern operations for both modes of portal behavior.

8.1 Cycle timer synchronization

The bridges that interconnect buses into a net shall be responsible to maintain phase-locked synchronization between the net cycle master and all the other cycle masters in the net. Phase lock is achieved when the CYCLE_TIME.*cycle_offset* value is identical for two cycle masters separated by a single bridge. A bridge accomplishes phase lock by measuring the cycle offset difference between the cycle master on the upstream portal's bus and the cycle master on the alpha portal's bus and adjusts this difference to account for propagation delays between the portals and their respective cycle masters. When the difference (measured in ticks of a 24.576 MHz cycle timer) is nonzero, the alpha portal either adjusts its own cycle time (if it is the cycle master) or transmits a cycle master adjustment packet to the cycle master on its bus. Because of accumulated phase error in propagation delay between the alpha portal and its cycle master, the cycle offset difference between the two buses exhibits variations that center on zero.

Bridges shall be capable of cycle time phase synchronization, as specified by 8.1.1; each portal shall be adjustable cycle master-capable, as specified by 8.1.2.

8.1.1 Alpha portal regulation of the local cycle master

All alpha portals, except the prime portal itself,[20] shall adjust the cycle master on their local bus as necessary to maintain phase synchronization with the cycle master on the upstream portal's bus. Adjustment of the downstream cycle master is necessary when the phase difference between the adjacent buses, measured in ticks of a 24.576 MHz cycle timer, is nonzero. This subclause specifies how the phase difference between the adjacent buses is measured and the circumstances under which a phase difference causes the alpha portal to adjust the cycle master.

If no cycle master is active on the alpha portal's bus or the downstream cycle master is not adjustable, the alpha portal shall transmit a PHY configuration packet to cause itself to be selected as root and shall initiate an arbitrated (short) bus reset to cause selection of itself as the new, adjustable cycle master. These actions shall be taken as soon as possible and without regard for the recommendations of IEEE Std 1394a-2000 with respect to minimum intervals between successive bus resets. An alpha portal shall not attempt to transfer root and cycle master responsibilities to any node unless the portal is more capable than the current root (see IEEE Std 1394a-2000 for details of the hierarchy of root capabilities).

---

[19] A bridge portal also monitors stream subactions originated by other unit architectures implemented within the same node in order to detect stream subactions to forward to its co-portal.

[20] On the same bus as the net cycle master, a single node that is not the cycle master may adjust the interval between successive cycle synchronization events by transmitting cycle master adjustment packets. The means by which node and only one such node is selected are beyond the scope of this standard.

Once each isochronous period, the bridge shall determine the instantaneous phase difference, modulus 3072, between its upstream portal's CYCLE_TIME.*cycle_offset* and that of the alpha portal. First, the bridge shall subtract the upstream portal's cycle offset from that of the alpha portal and retain the remainder modulus 3072. The resultant value is an intermediate phase difference; it does not reflect the propagation delay of cycle start information from the cycle masters to their respective bridge portals.[21] Next, the bridge shall add the propagation delay from the downstream cycle master to the alpha portal and shall subtract the propagation delay from the upstream cycle master to the upstream portal. The adjusted phase difference shall determine whether the downstream cycle master shall be adjusted, as specified by Table 8-1. The units in which phase difference and propagation delay are measured are ticks of a 24.576 MHz cycle timer.

Table 8-1 — Downstream cycle master adjustment dependent upon phase difference

Phase difference	Action	*sy*
Nonzero and less than or equal to 1536	Set the cycle offset threshold value for the next cycle synchronization event to 3072 ("go slow").	1
Zero	Set the cycle offset threshold value for the next cycle synchronization event to 3071 (no cycle master adjustment necessary).	2
Greater than 1536	Set the cycle offset threshold value for the next cycle synchronization event to 3070 ("go fast").	3

If the alpha portal is also the downstream cycle master, there is no need to transmit a cycle master adjustment packet; it shall adjust its own threshold value in accordance with Table 8-1. Otherwise, if the phase difference is nonzero, the alpha portal shall transmit the appropriate cycle master adjustment packet, with the *sy* value indicated, as soon as possible within the current isochronous period. When the cycle offset threshold value should be 3071, transmission of a cycle master adjustment packet is optional. The alpha portal shall transmit the cycle master adjustment packet at the fastest speed permitted by the capabilities of the alpha portal, the downstream cycle master, and the PHYs of any intervening nodes. On any bus except the prime bus, only an alpha portal may transmit cycle master adjustment packets.

The algorithm for the determination of phase difference between two adjacent buses and the subsequent adjustment of the downstream cycle master is expressed in more detail in Table 8-2.

Table 8-2 — Bridge actions to synchronize cycle time phase for adjacent buses (Sheet 1 of 2)

```
#include "csr.h"
#include "global.h"

VOID adjustDownstreamCycleMaster(UNSIGNED alphaDelay, UNSIGNED upstreamDelay,
                                 UNSIGNED alphaOffset, UNSIGNED upstreamOffset) {

   STATIC INT adjustment = 0;    /* Correction factor for next cycle sync event */
   struct {                      /* Cycle master adjustment packet */
      DOUBLET dataLength;
      BYTE tag:2;
      BYTE channel:6;
      BYTE tcode:4;
      BYTE sy:4;
   } cycleMasterAdjustmentPacket;
   UNSIGNED phaseDifference;

   phaseDifference = (  6144        /* Slop for modulus 3072 arithmetic */
                      + (alphaOffset + alphaDelay) /* "True" time of cycle synch events */
                      - (upstreamOffset + upstreamDelay)) % 3072;
```

[21] The delay in cycle start information depends not only upon the propagation delay from the cycle master to the bridge portal, but also upon the length and transmission speed of the cycle start packet itself, since the cycle master's CYCLE_TIME register is sampled before the first bit of cycle time is inserted into the cycle start packet and a cycle slave updates its cycle time only after receipt of the entire cycle start packet.

Table 8-2 — Bridge actions to synchronize cycle time phase for adjacent buses (Sheet 2 of 2)

```
   if (phaseDifference <= 4 || phaseDifference >= 3068) {   /* Phases aligned within tolerances? */
      adjustment = 0;              /* Yes, reset "sticky" adjustment variable */
      return;                      /* Nothing more to do! */
   } else if (phaseDifference <= 1344)
      adjustment = 1;              /* Delay the next cycle sync by a tick */
   else if (phaseDifference > 1728)
      adjustment = -1;             /* Hasten the next cycle sync by a tick */
   else if (adjustment == 0)       /* No existing bias in adjustment direction? */
      adjustment = (phaseDifference <= 1536) ? 1 : -1;   /* Choose initial direction */
   if (stateSet.cmstr)            /* Are we the cycle master? */
      cycleOffsetThreshold = 3071 + adjustment; /* Yes, alter our own threshold */
   else {         /* No, create adjustment packet and transmit to cycle master */
      cycleMasterAdjustmentPacket.dataLength = 0;
      cycleMasterAdjustmentPacket.tag = 3;
      cycleMasterAdjustmentPacket.channel = DEFAULT_BROADCAST_CHANNEL;
      cycleMasterAdjustmentPacket.tcode = 0x0A;
      cycleMasterAdjustmentPacket.sy = 2 + adjustment;
      LK_ISO.request = &cycleMasterAdjustmentPacket; /* Send the adjustment packet isochronously */
   }

}
```

The procedure `adjustDownstreamCycleMaster()` executes on the bridge's alpha portal each time a LK_CYCLE.*indication* is generated locally. This insures that the phase difference across the bridge is sampled once and only once each isochronous period. The four input parameters to the procedure (`alphaDelay`, `upstreamDelay`, `alphaOffset`, and `upstreamOffset`) represent the observed values for the propagation delay from the cycle masters to the bridge portals and the simultaneous sample of CYCLE_TIME.*cycle_offset*, for both portals, taken shortly after the LK_CYCLE.*indication*. When the alpha portal is also the downstream cycle master, the adjustment to the global variable `cycleOffsetThreshold` directly affects the next cycle synchronization event at the alpha portal (see Table 8-3). Otherwise, a cycle master adjustment packet is constructed and signaled to the alpha portal's link for isochronous transmission to the cycle master. When the phase difference is approximately half an isochronous period, the static variable adjustment retains the direction of the previous cycle master adjustment to insure convergence towards a effectively zero phase difference.

NOTE—When the alpha portal is also the downstream cycle master, the value of `alphaDelay` is zero. Similarly, if the upstream portal is the cycle master on its bus then `upstreamDelay` is zero.

8.1.2 Cycle master adjustment

An active, adjustable cycle master that observes a cycle master adjustment packet shall temporarily override the threshold value at which CYCLE_TIME.*cycle_offset* wraps around to zero and causes CYCLE_TIME.*cycle_count* to increment. The *sy* field in the packet shall indicate the override value, as specified by 6.2. If no cycle master adjustment packet is observed by the cycle master during an isochronous period, CYCLE_TIME shall behave as specified by IEEE 1394: an increment to CYCLE_TIME.*cycle_offset* from the value of 3071 shall cause it to wrap around to zero and increment the cycle count.

Cycle master implementations that recognize and act upon cycle master adjustment packets shall be designed to insure that cycle synchronization events are not lost as a consequence of a temporary override of the threshold value.

The temporary override value for the threshold remains in effect until the next cycle synchronization event, as specified by Table 8-3. For nodes that implement cycle master adjustment capability, Table 8-3 supersedes Table 6-14 in IEEE Std 1394-1995.

Table 8-3 — Cycle time update actions for adjustable cycle masters

```
#include “csr.h”

BOOLEAN cycleSynchQueued;           /* Need cycle start if we’re cycle master */

VOID updateCycleTime() {            /* Executed 24,576,000 times per second */

   if (cycleTime.cycleOffset >= cycleOffsetThreshold) {
      cycleTime.cycleOffset = 0;    /* Wrap cycle offset back to zero */
      cycleOffsetThreshold = 3071;  /* Restore threshold to nominal value */
      if (cycleTime.cycleCount == 7999) {
         cycleTime.cycleCount = 0;  /* Wrap cycle count back to zero */
         busTime.secondsCount++;    /* Count another second */
         cycleTime.secondsCount = busTime.secondsCountLo;    /* Least significant bits */
      } else
         cycleTime.cycleCount++;
      LK_CYCLE.indication = TRUE;   /* Indicate cycle synch event to interested parties */
      if (stateSet.cmstr)           /* Are we the cycle master? */
         cycleSynchQueued = TRUE;   /* Yes, request a cycle start packet */
   } else
      cycleTime.cycleOffset++;

}
```

In Table 8-3, `cycleOffsetThreshold` is a global link variable that shall be updated as soon as possible but before the second cycle synch event after receipt of a cycle master adjustment packet. An alpha portal that is a cycle master may also directly update the variable (see Table 8-2). The power reset value of `cycleOffsetThreshold` shall be 3071.

NOTE—The normative requirements of this clause are behavioral; the intent is to permit the occurrence of the next cycle synchronization event to be advanced or retarded by at most one cycle timer tick. Alternative, compliant implementations may exist.

8.2 Net time

Although the procedures described above permit all cycle timers within a net to be synchronized to the net cycle master, they are not sufficient to permit nodes on different buses to share a common time. Common time is relative between any two buses; the TIME OFFSET message specified in 6.6.4.3 permits the time difference between arbitrary buses to be measured. The relative time difference, measured in isochronous cycles, between any two buses does not change so long as neither the value of BUS_TIME nor the combined value of CYCLE_TIME.*second_count* and CYCLE_TIME.*cycle_count* on either bus changes other than as a result of normal increment. A potential change in either of these circumstances is signaled by net update. When this occurs, nodes that require a shared knowledge of time may transmit TIME OFFSET requests.

A TIME OFFSET request’s *snarf* field shall be equal to three, which causes it to be intercepted and processed by all bridge portals on the route between the two buses whose relative time difference is to be measured. Each bridge updates a cumulative cycle count difference maintained in the message with the relative difference between its two portals. The direction of the adjustment, is determined by the relationship of the entry and exit portals, as shown by Table 8-4.

Table 8-4 — Entry portal actions on receipt of a TIME OFFSET request (Sheet 1 of 2)

```
#include “csr.h”
#include “global.h”
#include “packets.h”

VOID timeOffset(DOUBLET sourceID, DOUBLET destinationID,
                TIME_OFFSET_MSG *timeOffsetMsg, DOUBLET exitBusID,
                QUADLET entryBusTime, QUADLET exitBusTime,
                DOUBLET entryCycleCount, DOUBLET exitCycleCount) {
```

Table 8-4 — Entry portal actions on receipt of a TIME OFFSET request (Sheet 2 of 2)

```
   timeOffsetMsg->deltaCycles +=          (exitCycleCount - entryCycleCount)
                               + 8000 * (exitBusTime - entryBusTime);
   if (destinationID >> 6 != exitBusID)
      transmitMsg(destinationID, MESSAGE_REQUEST, SNARF_ALL, timeOffsetMsg);
   else
      transmitMsg(sourceID, MESSAGE_RESPONSE, NO_SNARF, timeOffsetMsg);

}
```

The procedure `timeOffset()` processes an intercepted TIME OFFSET message. The first three input parameters are the *source_ID* and *destination_ID* fields from the packet header and the subaction data payload (the TIME OFFSET message itself). The fourth parameter is the value of the exit portal's CLAN_INFO.*bus_ID* (the procedure is presented from the viewpoint of the bridge's entry portal with respect to the TIME OFFSET message). The remaining input parameters to the procedure, `entryBusTime`, `entryCycleCount`, `exitBusTime`, and `exitCycleCount`, represent the observed values of BUS_TIME.*seconds* and CYCLE_TIME.*cycle_count*, respectively, as seen on the entry portal's and exit portal's buses, respectively. It is essential that these values be sampled simultaneously.

Although the procedure shows the calculation of the relative time difference between the bridge's portals each time a TIME OFFSET message is intercepted, implementations may sample this difference less frequently.

8.3 GASP routing and operations

This subclause specifies the behavior of bridge portals when they receive or transmit global asynchronous stream packet (GASP) subactions during normal operations (*i.e.*, when QUARANTINE.*net_update* is zero). Bridge portals forward GASP if the *channel* field in the packet header is equal to 31 (the default broadcast channel) and the *sy* field in the packet header is greater than or equal to eight. GASP that meets these criteria is propagated away from the originator and throughout the entire net.

When a listening portal snoops a GASP subaction, it shall verify that the *channel* field is equal to 31 and that the *sy* field is greater than or equal to eight. If either condition is false, the GASP subaction shall be discarded. Otherwise, the listening portal shall inspect the *source_ID* field in the GASP header. If the most significant ten bits of *source_ID* are equal to $3FF_{16}$, the originator of the GASP subaction is connected to the listening portal's bus. The listening portal shall verify that a virtual ID is assigned to the originator of the subaction. If not, the GASP subaction shall be discarded. Otherwise, the listening portal shall replace the local node ID in *source_ID* with a global node ID that contains the listening portal's bus ID and the originating node's virtual ID and then shall forward the GASP subaction to the co-portal for transmission on the co-portal's bus.

If the most significant ten bits of *source_ID* (bus ID) are not equal to $3FF_{16}$ and the ROUTE_MAP entry for the bus ID is VALID, the listening portal shall forward the GASP subaction to its co-portal for transmission on the co-portal's bus. Otherwise, the GASP subaction shall be discarded.

NOTE—The requirement that the ROUTE_MAP entry be VALID prevents GASP subactions from being forwarded back towards their originator.

A listening portal shall not transmit an acknowledgment in response to a snooped GASP subaction. This is true whether the subaction is discarded by the listening portal or forwarded to the co-portal.

Independent of the routing operations described previously, any listening or talking portal that snoops or transmits a GASP subaction shall also provide a TR_DATA.*indication* for the subaction to unit architectures present in the portal node. In the case of an listening portal, this requirement shall apply whether the subaction is forwarded to the co-portal or not.

8.4 Listening portal operations (isochronous streams)

Listening portals snoop all stream subactions in order to inspect the *channel* field in the packet header. Listening portals also inspect stream subactions originated by unit architectures implemented in the portal node. Although the details are dependent upon the implementation, it is assumed that each portal has a bit mask that identifies which of the 64 channels are to be buffered and, after a constant isochronous delay across the bridge's internal fabric, subsequently retransmitted by the co-portal.

A listening portal shall forward a stream subaction by transferring it to the co-portal *via* the bridge's internal fabric, with the expectation that the co-portal subsequently retransmits the subaction during the appropriate isochronous period.

8.5 Talking portal operations (isochronous streams)

Talking portals shall maintain queues of isochronous subactions observed on the bridge's internal fabric that match the portal's criteria for retransmission. Isochronous stream subactions shall be retransmitted during a particular isochronous period whose cycle number is a fixed number of cycles later than the isochronous period during which the subactions were snooped by the listening portal.

Stream subactions transferred *via* the bridge's internal fabric shall be identified to the talking portal by implementation-dependent methods.

Before the subaction may be retransmitted, the talking portal shall transform the packet header according to information in the portal's internal data structures. The stream information specifies the *channel* number for the retransmitted stream packet header and the speed at which the stream subaction shall be transmitted. If the data payload of the stream subaction exceeds the maximum data payload specified by the most recent JOIN or RENEW message, the stream subaction shall be discarded.

8.6 Isochronous stream connection management

Before an isochronous stream may be transferred from a talker on one bus to zero or more listeners on other buses, it is necessary to explicitly establish routes across bridges and allocate resources for the stream. In the case when the talker and listeners are on the same bus but the controller is on a different bus, it is necessary to instruct a bridge portal on the talker's bus to allocate resources—even though the isochronous stream passes through no bridges. Isochronous stream connections are set up and torn down by the messages described in 6.6.3. Subclause 4.6 provides an overview of stream connection management; 8.6 normatively defines bridge behavior when a stream management message is intercepted or received by a bridge portal. In addition to the allocation or deallocation of resources, both within the bridge and on the local bus, the receipt of a stream management message triggers the transmission of another stream management message.

A controller that originates a stream management message shall initialize its *snarf* field and address the message to the appropriate *destination_ID* as specified by the table below:

Talker	Listener	*destination_ID*	*snarf*	Comments
Local	Local	—		Do not use stream management messages; controller, talker, and listener are all local.
	Remote	Listener	2	Initial entry portal converts *talker_ID* to a global node ID, changes *snarf* to 1 and transmits message to listener.
Remote	Local	Talker	3	Initial entry portal converts *listener_ID* to a global node ID and transmits message to talker.
	Remote	Listener	1	Both *talker_ID* and *listener_ID* contain global node IDs.

The expected response to a stream management message is a STREAM STATUS message. Since stream management messages might be processed by numerous bridge portals along routes whose hop count cannot be known *a priori* by the controller, it cannot calculate a deterministic time limit for the receipt of the STREAM STATUS message. However, the time limit should be at least twice the remote time-out from the controller to the listener plus the remote time-out from the controller to the talker.

Stream management messages are global subactions, which are discarded if they encounter net update; these messages may also be discarded if they encounter bridge congestion. In either case, a stream controller that originates a JOIN, LEAVE, or RENEW message might fail to receive the anticipated STREAM STATUS message. The controller should retransmit the original stream management message (after altering its *signature* field) if net update occurs or an implementation-dependent time limit elapses before the return of a STREAM STATUS message whose *signature* field matches that of the outstanding stream management message. This error recovery strategy is reliable because the behaviors of the stream management messages have been designed to be idempotent.

The C pseudocode that specifies the processing of stream management messages uses some common procedures; these are provided in Table 8-5.

Table 8-5 — Common stream management procedures (Sheet 1 of 2)

```
#include “csr.h”
#include “global.h”
#include “packets.h”

STREAM_INFO *allocateStreamDescriptor() {

   UNSIGNED i;

   for (i = 0; i < bridgeCapabilities.streams; i++)      /* Search for free stream descriptor */
      if (streamInfo[i].channel == UNASSIGNED_CHANNEL)
         return(&streamInfo[i]);
   return(NULL);                                         /* No stream descriptor available */

}

VOID deallocateStreamDescriptor(STREAM_INFO *streamInfo) {

   memset(streamInfo, 0, sizeof(streamInfo));
   streamInfo->channel = UNASSIGNED_CHANNEL;             /* Mark stream descriptor available */
   memset(streamInfo->controllerID, 0xFF,                /* Mark all controller IDs unknown */
          sizeof(streamInfo->controllerID));

}

STREAM_INFO *allocateResources(DOUBLET maxPayload, BYTE speed) {

   QUADLET bandwidth = 0;
   BYTE channel;
   DOUBLET packetSize;
   STREAM_INFO *streamInfo;

   if ((streamInfo = allocateStreamDescriptor()) == NULL)
      return(NULL);
   channel = allocateChannel(UNASSIGNED_CHANNEL);        /* Attempt channel allocation */
   if (channel == UNASSIGNED_CHANNEL)
      return(NULL);                                      /* No channel available */
   if (maxPayload != 0) {                                /* Allocate isochronous bandwidth? */
      packetSize = 3 + (maxPayload + 3) / 4;             /* Packet size, in quadlets */
      bandwidth = 512 + packetSize << (4 - speed);       /* Worst-case overhead assumption */
      if (!allocateBandwidth(bandwidth)) {
         deallocateChannel(channel);
         return(NULL);                                   /* Insufficient bandwidth available */
      }
   }
```

Table 8-5 — Common stream management procedures (Sheet 2 of 2)

```
   streamInfo->channel = channel;
   streamInfo->speed = speed;
   streamInfo->maxPayload = maxPayload;
   streamInfo->bandwidth = bandwidth;
   return(streamInfo);

}

VOID deallocateResources(STREAM_INFO *streamInfo) {

   deallocateBandwidth(streamInfo->bandwidth);          /* Release bandwidth ... */
   deallocateChannel(streamInfo->channel);              /* ... and channel */
   deallocateStreamDescriptor(streamInfo);              /* Return stream descriptor */

}

VOID listenRequest(STREAM_MSG *joinMsg, STREAM_INFO *streamInfo) {

   STREAM_MSG listenMsg;

   memset(&listenMsg, 0, sizeof(listenMsg));
   listenMsg.opcode = LISTEN;
   listenMsg.requesterEUI64 = joinMsg->requesterEUI64;
   listenMsg.talkerEUI64 = joinMsg->talkerEUI64;
   listenMsg.talkerIndex = joinMsg->talkerIndex;
   listenMsg.listenerIndex = joinMsg->listenerIndex;
   listenMsg.lspd = joinMsg->lspd;
   listenMsg.channel = streamInfo->channel;
   listenMsg.controllerID = joinMsg->controllerID;
   listenMsg.talkerID = joinMsg->talkerID;
   listenMsg.listenerID = joinMsg->listenerID;
   listenMsg.latency = bridgeCapabilities.latency;
   forwardMsg(listenMsg.listenerID, MESSAGE_REQUEST, SNARF_ALL, &listenMsg);

}

VOID streamStatusResponse(STREAM_MSG *streamMsg, BYTE result) {

   STREAM_MSG statusMsg;

   if (streamMsg->controllerID == 0xFFFF)    /* Controller's whereabouts unknown? */
      return;
   memset(&statusMsg, 0, sizeof(statusMsg));
   statusMsg.opcode = STREAM_STATUS;
   statusMsg.result = result;
   statusMsg.requesterEUI64 = streamMsg->requesterEUI64;
   statusMsg.responderEUI64 = ownEUI64;
   statusMsg.talkerEUI64 = streamMsg->talkerEUI64;
   statusMsg.talkerIndex = streamMsg->talkerIndex;
   statusMsg.latency = (streamMsg->opcode == LISTEN) ? streamMsg->latency : 0;
   transmitMsg(streamMsg->controllerID, MESSAGE_RESPONSE, NO_SNARF, &statusMsg);

}
```

8.6.1 JOIN request processing

The principal function of a JOIN request message is the allocation of stream resources (channel and bandwidth) and the retention of enough information about the stream on a bus so that the resources can be reallocated after bus reset or released if the connection is torn down. On any particular bus, the bridge portal responsible for the initial allocation of stream resources and their subsequent management is the reallocation proxy. A bridge portal can be the reallocation proxy for more than one stream on a bus, but for a particular stream there is typically one reallocation proxy on a bus. The talker's bus is the exception: each listening portal is potentially a reallocation proxy (connection management procedures

prevent duplicate reallocation of resources). The processing of a JOIN message by a bridge portal depends upon whether the portal is on the path between the talker and listener or is on the talker's bus. The table below summarizes these relationships; *talker_bus_ID* represents the most significant ten bits of the *talker_ID* field in the JOIN message.

Bus ID	ROUTE_MAP[*talker_bus_ID*]	Comment
talker_bus_ID not equal to CLAN_INFO.*bus_ID*	VALID	The portal might be on the path from the talker to the listener but is not a talking portal. Transmit the JOIN message toward the talker (it will be intercepted by the talking portal).
	FORWARD	The portal is the talking portal and reallocation proxy for the stream. If not already allocated, allocate isochronous resources. Next, forward the JOIN message to the co-portal for transmission toward the talker.
talker_bus_ID equal to CLAN_INFO.*bus_ID*	—	The portal might be a reallocation proxy for the stream. If not already allocated, allocate isochronous resources.[a] If the listener is connected to the local bus, return a STREAM STATUS message to the controller else transmit a LISTEN message toward the listener.

[a] When multiple listening or reallocation only portals are present on the talker's bus, they cooperate with each other so that isochronous resources are not redundantly allocated.

Resource allocation for the stream is complete when the JOIN request message is intercepted by a portal connected to the talker's bus and, if not already allocated, the portal succeeds in allocating resources for the stream. Unless the listening portal previously established a connection with the talker, if the talker uses IEC 61883-1 connection management protocol, the portal shall configure the talker's output plug control register. If the listener is connected to the local bus and uses IEC 61883-1 connection management protocol, the portal shall configure the listener's input plug control register. If the portal and the listener are connected to the same bus, stream setup is complete and the portal shall transmit a STREAM STATUS response message to the controller that originated the JOIN request message. Otherwise, the portal shall transmit a LISTEN message toward the listener; the intercepting portals on the route complete the stream set up (see 8.6.2).

The functional behavior of a bridge portal that intercepts a JOIN request message is expressed in detail by Table 8-6.

Table 8-6 — Bridge portal actions on receipt of a JOIN request (Sheet 1 of 4)

```
#include “csr.h”
#include “global.h”
#include “packets.h”

VOID join(DOUBLET sourceID, BYTE snarf, STREAM_MSG *joinMsg) {

   BOOLEAN listenerPCR, talkerPCR, resourcesAllocated = FALSE;
   BYTE listenerLocalID, listenerVirtualID, talkerLocalID, talkerVirtualID, speed;
  DOUBLET listenerBusID = joinMsg->listenerID >> 6, talkerBusID = joinMsg->talkerID >> 6, packetSize;
   IMPR_CSR iMPR;
   IPCR_CSR iPCR;
   OMPR_CSR oMPR;
   OPCR_CSR oPCR;
   STREAM_INFO *streamInfo;

   if (joinMsg->controllerID >> 6 == 0x3FF)  /* First snarfing portal updates controller ID */
      joinMsg->controllerID = sourceID;
   if (listenerBusID == 0x3FF) {             /* Initial entry portal updates local listener ID */
      listenerBusID = clanInfo.busID;
      joinMsg->listenerID = (listenerBusID << 6) | physicalToVirtual[joinMsg->listenerID & 0x3F];
   }
   if (talkerBusID == 0x3FF) {               /* Initial entry portal updates local talker ID */
      talkerBusID = clanInfo.busID;
      joinMsg->talkerID = (talkerBusID << 6) | physicalToVirtual[joinMsg->talkerID & 0x3F];
   }
```

Table 8-6 — Bridge portal actions on receipt of a JOIN request (Sheet 2 of 4)

```
if (snarf == SNARF_INITIAL_ENTRY) {       /* Forced intercept by initial entry portal? */
   transmitMsg(joinMsg->listenerID, MESSAGE_REQUEST, SNARF_TERMINAL_EXIT, joinMsg);
   return;                                /* Exit after redirection to terminal exit portal */
}
if (clanInfo.busID == listenerBusID) {    /* Check listener IEC 61883-1 capabilities */
   listenerVirtualID = joinMsg->listenerID & 0x3f;
   listenerLocalID = virtualToPhysical[listenerVirtualID];
   if (listenerPCR = (joinMsg->listenerIndex <= 30)) {
      readQuadlet(listenerLocalID, IMPR, &iMPR);
      if (joinMsg->listenerIndex >= iMPR.inputPlugs) {   /* Plug index valid? */
         streamStatusResponse(joinMsg, INVALID_PLUG);
         return;
      }
      joinMsg->lspd = MIN(joinMsg->lspd, iMPR.spd);   /* iMPR limits speed */
   }
}
if (clanInfo.busID == talkerBusID) {      /* Check talker IEC 61883-1 capabilities */
   talkerVirtualID = joinMsg->talkerID & 0x3f;
   talkerLocalID = virtualToPhysical[talkerVirtualID];
   if (talkerPCR = (joinMsg->talkerIndex <= 30)) {
      readQuadlet(talkerLocalID, OMPR, &oMPR);
      if (joinMsg->talkerIndex >= oMPR.outputPlugs) {    /* Plug index valid? */
         streamStatusResponse(joinMsg, INVALID_PLUG);
         return;
      }
      joinMsg->tspd = MIN(joinMsg->tspd, oMPR.spd);   /* oMPR limits speed */
   }
}
if (clanInfo.busID == talkerBusID) {      /* Are we on the talker's bus? */
   if (clanInfo.busID == listenerBusID) { /* Listener on the same bus? */
      speed = pathSpeed(talkerLocalID, listenerLocalID);
      speed = MIN(speed, MIN(joinMsg->tspd, joinMsg->lspd));
   } else {                               /* No, we'll actually be listening */
      speed = pathSpeed(talkerLocalID, ownLocalID);
      speed = MIN(speed, MIN(joinMsg->tspd, ownSpeed));
   }
   streamInfo = getStreamDescriptor(joinMsg->talkerEUI64, joinMsg->talkerIndex);
   if (streamInfo == NULL) {
      if (talkerPCR) {                    /* Using IEC 61883-1 connection management? */
         readQuadlet(talkerLocalID, OPCR + 4 * joinMsg->talkerIndex, &oPCR);
         if (oPCR.broadcastConnection == 0 && oPCR.pointToPointConnections == 0)
            if ((streamInfo = allocateResources(joinMsg->maxPayload, speed)) == NULL) {
               streamStatusResponse(joinMsg, RESOURCES_UNAVAILABLE);
               return;
            } else {
               resourcesAllocated = TRUE;
               streamInfo->talkerEUI64 = joinMsg->talkerEUI64;
               streamInfo->talkerIndex = joinMsg->talkerIndex;
               streamInfo->talkerID = joinMsg->talkerID;
               oPCR.pointToPointConnections++;
               oPCR.spd = speed;
               oPCR.channel = streamInfo->channel;
            }
         else if (oPCR.spd > speed || 4 * oPCR.payload > joinMsg->maxPayload) {
            streamStatusResponse(joinMsg, STREAM_CONFLICT);
            return;
         } else if ((streamInfo = allocateStreamDescriptor()) == NULL) {
            streamStatusResponse(joinMsg, RESOURCES_UNAVAILABLE);
            return;
         } else {
            streamInfo->talkerEUI64 = joinMsg->talkerEUI64;
            streamInfo->talkerIndex = joinMsg->talkerIndex;
            streamInfo->talkerID = joinMsg->talkerID;
            streamInfo->channel = oPCR.channel;
            streamInfo->speed = oPCR.spd;
```

Table 8-6 — Bridge portal actions on receipt of a JOIN request (Sheet 3 of 4)

```
            streamInfo->maxPayload = (oPCR.payload == 0) ? 1024 : 4 * oPCR.payload;
            oPCR.pointToPointConnections++;
            packetSize = 3 + oPCR.payload;   /* Packet size, in quadlets */
            if (oPCR.overhead == 0)
               streamInfo->bandwidth = 512 + packetSize << (4 - speed);
            else
               streamInfo->bandwidth = oPCR.overhead * 32 + packetSize << (4 - speed);
         }
         if (!programOutputPlug(talkerLocalID, joinMsg->talkerIndex, &oPCR)) {
            if (resourcesAllocated)
               deallocateResources(streamInfo);
            streamStatusResponse(joinMsg, STREAM_CONFLICT);
            return;
         }
      } else
         ;                                   /* Unspecified connection management */
   } else if (   streamInfo->speed > speed   /* Joining an existing stream */
              || streamInfo->maxPayload > joinMsg->maxPayload) {
     streamStatusResponse(joinMsg, STREAM_CONFLICT);  /* Existing speed or payload incompatible */
      return;
   }
   if (clanInfo.busID == listenerBusID) {    /* Listener on this bus? */
      if (streamInfo->listenerIndex[listenerVirtualID] == 0) { /* New listener? */
         streamInfo->listenerIndex[listenerVirtualID] = joinMsg->listenerIndex;
         streamInfo->controllerID[listenerVirtualID] = joinMsg->controllerID;
         streamInfo->lifetime[listenerVirtualID] = joinMsg->lifetime;
         streamInfo->localListeners |= (OCTLET) 1 << listenerVirtualID;
         if (streamInfo->portalRole == UNSPECIFIED)      /* First setup on this path? */
            streamInfo->portalRole = REALLOCATION_ONLY;  /* Yes, therefore we only reallocate */
         if (listenerPCR) {                  /* Need to program iPCR? */
            readQuadlet(listenerLocalID, IPCR + 4 * joinMsg->listenerIndex, &iPCR);
            if (iPCR.broadcastConnection == 0 && iPCR.pointToPointConnections == 0)
               iPCR.channel = streamInfo->channel;
            else if (iPCR.channel != streamInfo->channel) {
               if (resourcesAllocated)
                  deallocateResources(streamInfo);
               streamStatusResponse(joinMsg, STREAM_CONFLICT);
               return;
            }
            iPCR.pointToPointConnections++;
            programInputPlug(listenerLocalID, joinMsg->listenerIndex, &iPCR);
         } else
            ;                                /* Unspecified connection management */
      }
      streamStatusResponse(joinMsg, OK);
   } else {                                  /* Listener elsewhere---launch LISTEN message */
      streamInfo->portalRole = LISTENER;  /* Implies reallocation proxy on talker's bus */
      listenRequest(joinMsg, streamInfo);
   }
} else if (routeMap[talkerBusID] == VALID) {    /* Are we an innocent bystander? */
   transmitMsg(joinMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, joinMsg);
} else if (routeMap[talkerBusID] == FORWARD) {  /* Are we a talking portal? */
   if (bridgeCapabilities.maxIsoch != 0)
      if (joinMsg->maxPayload > 2 << bridgeCapabilities.maxIsoch) {
         streamStatusResponse(joinMsg, PAYLOAD_TOO_BIG);
         return;
      }
   if (clanInfo.busID == listenerBusID)         /* Listener on our bus? */
      speed = MIN(pathSpeed(ownLocalID, listenerLocalID), joinMsg->lspd);
   else                                         /* No, another portal will listen */
      speed = pathSpeed(ownLocalID, (BYTE) (sourceID & 0x3F));
   streamInfo = getStreamDescriptor(joinMsg->talkerEUI64, joinMsg->talkerIndex);
   if (streamInfo == NULL) {
      if ((streamInfo = allocateResources(joinMsg->maxPayload, speed)) == NULL) {
         streamStatusResponse(joinMsg, RESOURCES_UNAVAILABLE);
```

Table 8-6 — Bridge portal actions on receipt of a JOIN request (Sheet 4 of 4)

```
            return;
         } else {
            streamInfo->portalRole = TALKER;
            streamInfo->talkerEUI64 = joinMsg->talkerEUI64;
            streamInfo->talkerIndex = joinMsg->talkerIndex;
            streamInfo->talkerID = joinMsg->talkerID;
         }
      } else if (streamInfo->speed > speed) {
         streamStatusResponse(joinMsg, STREAM_CONFLICT);
         return;
      }
      forwardMsg(joinMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, joinMsg);
   } else                                          /* CLEAN or DIRTY indicates an error */
      streamStatusResponse(joinMsg, INVALID_TALKER);

}
```

8.6.2 LISTEN request processing

The principal function of a LISTEN request message is to communicate a channel number to a listening portal, but it also causes the final talking portal (the portal on the listener's bus) to program the listener to receive the stream and then return a STREAM STATUS response message to the controller that requested the stream connection setup with a JOIN request message. The processing of a LISTEN request message by a bridge portal depends upon whether the portal is a listening or talking portal on the path from the talker to the listener and, in the latter case, whether the portal is the final talking portal. The table below summarizes these relationships; *listener_bus_ID* represents the most significant ten bits of the *listener_ID* field in the LISTEN message.

ROUTE_MAP[*listener_bus_ID*]	Bus ID	Comment
VALID	—	The portal is a listening portal. Configure its hardware to listen for the stream data on the indicated channel, adjust the accumulated isochronous delay in the LISTEN message and forward the message to the co-portal for processing and transmission toward the listener.
FORWARD	*listener_bus_ID* not equal to CLAN_INFO.*bus_ID*	The portal is an intermediate talking portal. Configure its hardware to transmit the stream data on the channel previously allocated, then modify the LISTEN message to reflect the channel in use on this bus and transmit it towards the listener.
	listener_bus_ID equal to CLAN_INFO.*bus_ID*	The portal is the final talking portal. Configure its hardware to transmit the stream data on the channel previously allocated, program the listener to receive the stream and then transmit a STREAM STATUS to the controller that requested the stream setup.

The functional behavior of a bridge portal that intercepts a LISTEN request is expressed in detail by Table 8-7.

Table 8-7 — Bridge portal actions on receipt of a LISTEN request (Sheet 1 of 2)

```
#include "csr.h"
#include "global.h"
#include "packets.h"

VOID listen(STREAM_MSG *listenMsg) {

   BOOLEAN listenerPCR;
   BYTE listenerLocalID, listenerVirtualID;
   DOUBLET listenerBusID = listenMsg->listenerID >> 6, talkerBusID = listenMsg->talkerID >> 6;
   IMPR_CSR iMPR;
   IPCR_CSR iPCR;
```

Table 8-7 — Bridge portal actions on receipt of a LISTEN request (Sheet 2 of 2)

```
   STREAM_INFO *streamInfo;

   if (clanInfo.busID == listenerBusID) {    /* Check listener IEC 61883-1 capabilities */
      listenerVirtualID = listenMsg->listenerID & 0x3f;
      listenerLocalID = virtualToPhysical[listenerVirtualID];
      if (listenerPCR = (listenMsg->listenerIndex <= 30)) {
         readQuadlet(listenerLocalID, IMPR, &iMPR);
         if (listenMsg->listenerIndex >= iMPR.inputPlugs) {   /* Plug index valid? */
            streamStatusResponse(listenMsg, INVALID_PLUG);
            return;
         }
      }
   }
   if (routeMap[talkerBusID] == VALID) {                /* Are we a listening portal? */
      streamInfo = getStreamDescriptor(listenMsg->talkerEUI64, listenMsg->talkerIndex);
      if (streamInfo == NULL)
         streamInfo = allocateStreamDescriptor();
      streamInfo->portalRole = LISTENER;                /* Yes, initialize a stream descriptor */
      streamInfo->talkerEUI64 = listenMsg->talkerEUI64;
      streamInfo->talkerIndex = listenMsg->talkerIndex;
      streamInfo->talkerID = listenMsg->talkerID;
      streamInfo->channel = listenMsg->channel;
      listenMsg->latency += bridgeCapabilities.latency;
      forwardMsg(listenMsg->listenerID, MESSAGE_REQUEST, SNARF_ALL, listenMsg);
   } else if (routeMap[talkerBusID] == FORWARD) {       /* Are we a talking portal? */
      streamInfo = getStreamDescriptor(listenMsg->talkerEUI64, listenMsg->talkerIndex);
      if (streamInfo == NULL) {                         /* Missing stream descriptor? */
         streamStatusResponse(listenMsg, UNEXPECTED_ERROR);
         return;
      }
     streamInfo->coportalChannel = listenMsg->channel;  /* Remember the channel our co-portal uses */
      listenMsg->channel = streamInfo->channel;          /* Channel used on this bus */
     if (clanInfo.busID != listenerBusID) {             /* Are we an intermediate talking portal? */
         transmitMsg(listenMsg->listenerID, MESSAGE_REQUEST, SNARF_ALL, listenMsg);
     } else {                                          /* Almost done! We are the final talking portal */
         if (streamInfo->listenerIndex[listenerVirtualID] == 0) { /* Listener not yet setup? */
            streamInfo->listenerIndex[listenerVirtualID] = listenMsg->listenerIndex;
            streamInfo->controllerID[listenerVirtualID] = listenMsg->controllerID;
            streamInfo->lifetime[listenerVirtualID] = listenMsg->lifetime;
            streamInfo->localListeners |= (OCTLET) 1 << listenerVirtualID;
            if (listenerPCR) {                          /* Using IEC 61883-1? */
               readQuadlet(listenerVirtualID, IPCR + 4 * listenMsg->listenerIndex, &iPCR);
               if (iPCR.broadcastConnection == 0 && iPCR.pointToPointConnections == 0)
                  iPCR.channel = streamInfo->channel;
               else if (iPCR.channel != streamInfo->channel) {
                  streamStatusResponse(listenMsg, STREAM_CONFLICT);
                  return;
               }
               iPCR.pointToPointConnections++;
               programInputPlug(listenerVirtualID, listenMsg->listenerIndex, &iPCR);
            } else
               ;                                        /* “Other” connection management TBD */
         }
         streamStatusResponse(listenMsg, OK);
      }
   } else                                               /* CLEAN or DIRTY indicates an error */
      streamStatusResponse(listenMsg, INVALID_TALKER);

}
```

8.6.3 LEAVE request processing

The principal function of a LEAVE message is the deallocation of stream resources (channel and bandwidth) as well as the release of any internal resources used by the bridge. The processing of a LEAVE message by a bridge portal depends upon whether the portal is on the path between the talker and listener or is on the talker's bus. The table below summarizes these relationships; *talker_bus_ID* represents the most significant ten bits of the *talker_ID* field in the LEAVE message.

Bus ID	ROUTE_MAP[*talker_bus_ID*]	Comment
talker_bus_ID not equal to CLAN_INFO.*bus_ID*	VALID	The portal might be on the path from the talker to the listener but is not a reallocation proxy. If on the path from the talker to the listener, release internal resources. Transmit the LEAVE message toward the talker (it will be intercepted by the talking portal).
	FORWARD	The portal is the reallocation proxy for the stream. If there are no remaining listeners on the local bus (including listening portals), release the stream resources and forward the LEAVE message to the co-portal for transmission toward the talker. Otherwise, transmit a STREAM STATUS message to the controller.
talker_bus_ID equal to CLAN_INFO.*bus_ID*	—	The portal might be a reallocation proxy for the stream. If there are no remaining listeners on the local bus (including listening portals), release the stream resources. Transmit a STREAM STATUS message to the controller.

The functional behavior of a bridge portal that intercepts a LEAVE request is expressed in detail by Table 8-8.

Table 8-8 — Bridge portal actions on receipt of a LEAVE request (Sheet 1 of 3)

```
#include “csr.h”
#include “global.h”
#include “packets.h”

VOID leave(DOUBLET sourceID, BYTE snarf, STREAM_MSG *leaveMsg) {

   BOOLEAN listenerPCR, talkerPCR;
   BYTE listenerLocalID, listenerVirtualID, talkerLocalID, talkerVirtualID;
   DOUBLET listenerBusID = leaveMsg->listenerID >> 6, talkerBusID = leaveMsg->talkerID >> 6;
   IMPR_CSR iMPR;
   IPCR_CSR iPCR;
   OMPR_CSR oMPR;
   OPCR_CSR oPCR;
   STREAM_INFO *streamInfo;

   if (leaveMsg->controllerID >> 6 == 0x3FF) /* First snarfing portal updates controller ID */
      leaveMsg->controllerID = sourceID;
   if (listenerBusID == 0x3FF) {             /* Initial entry portal updates local listener ID */
      listenerBusID = clanInfo.busID;
      leaveMsg->listenerID = (listenerBusID << 6) | physicalToVirtual[leaveMsg->listenerID & 0x3F];
   }
   if (talkerBusID == 0x3FF) {               /* Initial entry portal updates local talker ID */
      talkerBusID = clanInfo.busID;
      leaveMsg->talkerID = (talkerBusID << 6) | physicalToVirtual[leaveMsg->talkerID & 0x3F];
   }
   if (snarf == SNARF_INITIAL_ENTRY) {       /* Forced intercept by initial entry portal? */
      transmitMsg(leaveMsg->listenerID, MESSAGE_REQUEST, SNARF_TERMINAL_EXIT, leaveMsg);
      return;                                /* Exit after redirection to terminal exit portal */
   }
   if (clanInfo.busID == listenerBusID) {    /* Check listener IEC 61883-1 capabilities */
      listenerVirtualID = leaveMsg->listenerID & 0x3f;
      listenerLocalID = virtualToPhysical[listenerVirtualID];
      if (listenerPCR = (leaveMsg->listenerIndex <= 30)) {
```

Table 8-8 — Bridge portal actions on receipt of a LEAVE request (Sheet 2 of 3)

```
         readQuadlet(listenerLocalID, IMPR, &iMPR);
         listenerPCR = (leaveMsg->listenerIndex < iMPR.inputPlugs);  /* Plug index valid? */
      }
   }
   if (clanInfo.busID == talkerBusID) {      /* Check talker IEC 61883-1 capabilities */
      talkerVirtualID = leaveMsg->talkerID & 0x3f;
      talkerLocalID = virtualToPhysical[talkerVirtualID];
      if (talkerPCR = (leaveMsg->talkerIndex <= 30)) {
         readQuadlet(talkerLocalID, OMPR, &oMPR);
         talkerPCR = (leaveMsg->talkerIndex < oMPR.outputPlugs);     /* Plug index valid? */
      }
   }
   if (clanInfo.busID == talkerBusID) {         /* Are we on the talker's bus? */
      streamInfo = getStreamDescriptor(leaveMsg->talkerEUI64, leaveMsg->talkerIndex);
      if (streamInfo == NULL) {                 /* Missing stream descriptor */
         streamStatusResponse(leaveMsg, INVALID_STREAM);
         return;
      }
      if (clanInfo.busID == listenerBusID) {    /* Listener on same bus? */
         if (listenerPCR) {                     /* Need to program iPCR? */
            readQuadlet(listenerLocalID, IPCR + 4 * leaveMsg->listenerIndex, &iPCR);
            if (iPCR.pointToPointConnections > 0) {
               iPCR.pointToPointConnections--;
               programInputPlug(listenerLocalID, leaveMsg->listenerIndex, &iPCR);
            }
         } else
            ;                                   /* "Other" connection management TBD */
         streamInfo->listenerIndex[listenerVirtualID] = 0;
         streamInfo->controllerID[listenerVirtualID] = 0xFFFF;
         streamInfo->lifetime[listenerVirtualID] = 0;
         streamInfo->localListeners &= ~((OCTLET) 1 << listenerVirtualID);
      } else
         streamInfo->localListeners &= ~((OCTLET) 1 << ownVirtualID);
      if (streamInfo->localListeners == 0) {    /* Any local listeners left? (that we know about) */
         if (talkerPCR) {                       /* Need to program oPCR? */
            readQuadlet(talkerLocalID, OPCR + 4 * leaveMsg->talkerIndex, &oPCR);
            if (oPCR.pointToPointConnections > 0) {
               oPCR.pointToPointConnections--;
               programOutputPlug(talkerLocalID, leaveMsg->talkerIndex, &oPCR);
            }
            if (oPCR.pointToPointConnections == 0)       /* No remaining listeners, period */
               deallocateResources(streamInfo);          /* Release all resources */
          else                                  /* Listeners (or listening portals) still present */
               deallocateStreamDescriptor(streamInfo);   /* Release internal resources, only */
            streamStatusResponse(leaveMsg, CONNECTION_DELETED);
         } else
            ;                                           /* "Other" connection management */
      } else                                            /* We've torn down as much as we can */
         streamStatusResponse(leaveMsg, CONNECTION_DELETED);
   } else if (routeMap[talkerBusID] == VALID) {    /* Are we (possibly) a listening portal? */
      streamInfo = getStreamDescriptor(leaveMsg->talkerEUI64, leaveMsg->talkerIndex);
      if (streamInfo != NULL)                       /* We're on the stream's path */
         deallocateStreamDescriptor(streamInfo);    /* Release internal resources, only */
      transmitMsg(leaveMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, leaveMsg);
   } else if (routeMap[talkerBusID] == FORWARD) {  /* Are we a talking portal? */
      streamInfo = getStreamDescriptor(leaveMsg->talkerEUI64, leaveMsg->talkerIndex);
      if (streamInfo == NULL) {                     /* Missing stream descriptor? */
         forwardMsg(leaveMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, leaveMsg);
         return;
      }
      if (clanInfo.busID == listenerBusID) {        /* Listener on our bus? */
         if (listenerPCR) {                         /* Need to program iPCR? */
            readQuadlet(listenerLocalID, IPCR + 4 * leaveMsg->listenerIndex, &iPCR);
            if (iPCR.pointToPointConnections > 0) {
               iPCR.pointToPointConnections--;
```

Table 8-8 — Bridge portal actions on receipt of a LEAVE request (Sheet 3 of 3)

```
                programInputPlug(listenerLocalID, leaveMsg->listenerIndex, &iPCR);
             }
          } else
             ;                                    /* “Other” connection management */
          streamInfo->listenerIndex[listenerVirtualID] = 0;
          streamInfo->controllerID[listenerVirtualID] = 0xFFFF;
          streamInfo->lifetime[listenerVirtualID] = 0;
          streamInfo->localListeners &= ~((OCTLET) 1 << listenerVirtualID);
       } else
          streamInfo->localListeners &= ~((OCTLET) 1 << physicalToVirtual[sourceID & 0x3F]);
       if (streamInfo->localListeners == 0) {    /* Any local listeners left? */
          deallocateResources(streamInfo);       /* No, release all resources */
          forwardMsg(leaveMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, leaveMsg);
       } else                                    /* Yes, we’ve torn down as much as we can */
          streamStatusResponse(leaveMsg, CONNECTION_DELETED);
    } else                                        /* CLEAN or DIRTY indicates an error */
       streamStatusResponse(leaveMsg, INVALID_TALKER);

}
```

8.6.4 RENEW request processing

The RENEW request message extends stream lifetime for a particular listener. If a listener’s stream lifetime decrements to zero, the talking portal on a listener’s bus automatically removes the listener from the stream by generating a LEAVE message as if it had been originated by the controller. Successful extension of a listener’s stream lifetime causes a STREAM STATUS response message to be transmitted to the requester. The functional behavior of a bridge portal that intercepts a RENEW request message is expressed by Table 8-9.

Table 8-9 — Bridge portal actions on receipt of a RENEW request (Sheet 1 of 2)

```
#include “csr.h”
#include “global.h”
#include “packets.h”

VOID renew(DOUBLET sourceID, BYTE snarf, STREAM_MSG *renewMsg) {

   BYTE listenerVirtualID;
   DOUBLET listenerBusID = renewMsg->listenerID >> 6, talkerBusID = renewMsg->talkerID >> 6;
   STREAM_INFO *streamInfo;

   if (renewMsg->controllerID >> 6 == 0x3FF) /* First snarfing portal updates controller ID */
      renewMsg->controllerID = sourceID;
   if (listenerBusID == 0x3FF) {             /* Initial entry portal updates local listener ID */
      listenerBusID = clanInfo.busID;
      renewMsg->listenerID = (listenerBusID << 6) | physicalToVirtual[renewMsg->listenerID & 0x3F];
   }
   if (talkerBusID == 0x3FF) {               /* Initial entry portal updates local talker ID */
      talkerBusID = clanInfo.busID;
      renewMsg->talkerID = (talkerBusID << 6) | physicalToVirtual[renewMsg->talkerID & 0x3F];
   }
   if (snarf == SNARF_INITIAL_ENTRY) {       /* Forced intercept by initial entry portal? */
      transmitMsg(renewMsg->listenerID, MESSAGE_REQUEST, SNARF_TERMINAL_EXIT, renewMsg);
      return;                                /* Exit after redirection to terminal exit portal */
   }
   listenerVirtualID = renewMsg->listenerID & 0x3f;
  if (routeMap[talkerBusID] == VALID)         /* Terminal exit portal---but not the talking portal */
      transmitMsg(renewMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, renewMsg);
  else if (routeMap[talkerBusID] == FORWARD) { /* Are we the talking portal on the listener’s bus? */
      streamInfo = getStreamDescriptor(renewMsg->talkerEUI64, renewMsg->talkerIndex);
```

Table 8-9 — Bridge portal actions on receipt of a RENEW request (Sheet 2 of 2)

```
      if (streamInfo == NULL)                    /* Invalid stream ID? */
         streamStatusResponse(renewMsg, INVALID_STREAM);
      else {                                     /* OK, extend STREAM lifetime */
         streamInfo->lifetime[listenerVirtualID] += renewMsg->lifetime;
         streamStatusResponse(renewMsg, OK);
      }
   } else                                        /* CLEAN or DIRTY indicates an error */
      streamStatusResponse(renewMsg, INVALID_TALKER);

}
```

NOTE—After net update, a stream controller should issue RENEW messages for all its active streams; the *lifetime* field should be zero. Although this does not change the stream's lifetime, it updates the controller's global node ID stored in the stream information kept by the talking portal on the listener's bus.

8.6.5 TEARDOWN request processing

When the disconnection of one or more nodes severs a stream's path, bridge portals connected to the bus from which the nodes were disconnected autonomously initiate teardown of part or all of the stream's path (see 8.6.6). The TEARDOWN message propagates teardown information away from the discontinuity that interrupted the stream's flow—either upstream (towards the talker) or downstream.

An upstream TEARDOWN message transferred from a former talking portal to its co-portal indicates that no downstream listeners—either endpoint listeners or listening portals—remain on the former talking portal's bus, i.e., there is no longer any reason to forward the stream from one portal to the other. Unless the recipient of the upstream TEARDOWN message is connected to the talker's bus, it shall transmit the message towards the talker. A talking portal that receives an upstream TEARDOWN message from a listening portal shall delete the portal from its list of local bus listeners; if the list is empty, the talking portal shall release isochronous resources, bandwidth and channel number, previously allocated for the stream and forward the TEARDOWN message to its co-portal. When a TEARDOWN message is received by a listening portal connected to the talker's bus, the portal shall decrement the talker's point-to-point connection counter (if the talker implements IEC 61883-1 plug control registers); if the point-to-point connection counter is zero, the portal shall release isochronous resources, bandwidth and channel number, previously allocated for the stream.

A downstream TEARDOWN message transmitted from a former talking portal to a listening portal on the same bus indicates that the stream's source is disconnected. The listening portal shall transfer the message to its co-portal. When a talking portal receives a downstream TEARDOWN message, it shall delete all local listeners—both endpoint listeners and listening portals. For endpoint listeners, the portal shall decrement each listener's point-to point connection counter (if implemented) and shall attempt to transmit a STREAM STATUS message to the controller that established the stream connection. For listening portals, the portal shall transmit the TEARDOWN message in order to propagate its effects to all listeners downstream from that portal. Once all local bus listeners are deleted, the portal shall release isochronous resources, bandwidth and channel number, previously allocated for the stream.

The functional behavior of a bridge portal that processes a TEARDOWN message is expressed in detail by Table 8-10.

Table 8-10 — Bridge portal actions on receipt of a TEARDOWN request (Sheet 1 of 3)

```
#include “csr.h”
#include “global.h”
#include “packets.h”

VOID teardown(DOUBLET sourceID, STREAM_MSG *teardownMsg) {

   BOOLEAN talkerPCR;
   BYTE listenerLocalID, listenerVirtualID, talkerLocalID, talkerVirtualID;
   DOUBLET talkerBusID = teardownMsg->talkerID >> 6;
   IMPR_CSR iMPR;
   IPCR_CSR iPCR;
   OMPR_CSR oMPR;
```

Table 8-10 — Bridge portal actions on receipt of a TEARDOWN request (Sheet 2 of 3)

```
OPCR_CSR oPCR;
STREAM_INFO *streamInfo;
STREAM_MSG statusMsg;

streamInfo = getStreamDescriptor(teardownMsg->talkerEUI64, teardownMsg->talkerIndex);
if (teardownMsg->upstream) {                /* Tear down inactive upstream portions */
   if (streamInfo == NULL) {                /* Unrecognized stream? */
      if (clanInfo.busID != talkerBusID) {  /* Yes, are we on an intermediate bus? */
         if (routeMap[talkerBusID] == FORWARD)
            forwardMsg(teardownMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, teardownMsg);
         else if (routeMap[talkerBusID] == VALID)
            transmitMsg(teardownMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, teardownMsg);
      }
      return;                               /* Nothing more to do */
   }
   if (streamInfo->portalRole == TALKER) {  /* Just lost a listening portal */
      listenerLocalID = sourceID & 0x3F;    /* Get listening portal's local ... */
      listenerVirtualID = physicalToVirtual[listenerLocalID];  /* ... and virtual IDs */
     streamInfo->listenerIndex[listenerVirtualID] = 0;  /* Remove listening portal from stream */
      streamInfo->controllerID[listenerVirtualID] = 0xFFFF;
      streamInfo->lifetime[listenerVirtualID] = 0;
      streamInfo->localListeners &= ~((OCTLET) 1 << listenerVirtualID);
      if (streamInfo->localListeners == 0) {
         deallocateResources(streamInfo);   /* Release bandwidth and channel number */
         forwardMsg(teardownMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, teardownMsg);
      }
   } else {                                         /* Listening portal no more ... */
      if (clanInfo.busID != talkerBusID) {          /* Intermediate listening portal? */
         deallocateStreamDescriptor(streamInfo);    /* No longer in use */
         transmitMsg(teardownMsg->talkerID, MESSAGE_REQUEST, SNARF_ALL, teardownMsg);
      } else {                                      /* We have arrived at the talker's bus! */
         talkerVirtualID = teardownMsg->talkerID & 0x3f;
         talkerLocalID = virtualToPhysical[talkerVirtualID];
         if (talkerPCR = (teardownMsg->talkerIndex <= 30)) {
            readQuadlet(talkerLocalID, OMPR, &oMPR);
            talkerPCR = (teardownMsg->talkerIndex < oMPR.outputPlugs);
         }
         if (talkerPCR) {
            readQuadlet(talkerLocalID, OPCR + 4 * teardownMsg->talkerIndex, &oPCR);
            if (oPCR.pointToPointConnections > 0) {
               oPCR.pointToPointConnections--;
               programOutputPlug(talkerLocalID, teardownMsg->talkerIndex, &oPCR);
               if (oPCR.pointToPointConnections == 0)    /* No remaining listeners, period */
                  deallocateResources(streamInfo);       /* Release all resources */
                else if (streamInfo->localListeners == 0) /* Our listeners (or listening portals)? */
                    deallocateStreamDescriptor(streamInfo);   /* Release internal resources, only */
               else
                  streamInfo->portalRole = REALLOCATION_ONLY;  /* No longer forward the stream */
               }
         } else                                         /* "Other" connection management */
            ;
      }
   }
} else {                                /* Tear down all listener connections downstream */
   if (streamInfo == NULL)                              /* Unrecognized stream? */
      return;                                           /* Yes, just quit */
   if (streamInfo->portalRole == LISTENER) {            /* Upstream flow cut off? */
      deallocateStreamDescriptor(streamInfo);           /* Release internal resources */
      forwardMsg(0xFFFF, MESSAGE_REQUEST, SNARF_ALL, teardownMsg);
     return;                                           /* Co-portal will continue the good work */
   }
   for (listenerVirtualID = 0; listenerVirtualID < 63; listenerVirtualID++)
      if ((streamInfo->localListeners & (OCTLET) 1 << listenerVirtualID) != 0) {
         listenerLocalID = virtualToPhysical[listenerVirtualID];
         if ((brdg[listenerLocalID] & BRIDGE) == BRIDGE)   /* Listening portal? */
```

Table 8-10 — Bridge portal actions on receipt of a TEARDOWN request (Sheet 3 of 3)

```
                transmitMsg(0xFFC0 || listenerLocalID, MESSAGE_REQUEST, SNARF_ALL, teardownMsg);
             else {
                if (streamInfo->controllerID[listenerVirtualID] != 0xFFFF) {
                 memset(&statusMsg, 0, sizeof(statusMsg));    /* STREAM STATUS to known controllers */
                   statusMsg.opcode = STREAM_STATUS;
                   statusMsg.result = CONNECTION_DELETED;
                   statusMsg.responderEUI64 = ownEUI64;
                   statusMsg.talkerEUI64 = streamInfo->talkerEUI64;
                   statusMsg.talkerIndex = streamInfo->talkerIndex;
                   transmitMsg(streamInfo->controllerID[listenerVirtualID], MESSAGE_RESPONSE,
                               NO_SNARF, &statusMsg);
                }
                if (streamInfo->listenerIndex[listenerVirtualID] <= 30) {
                   readQuadlet(listenerLocalID, IMPR, &iMPR);
                   if (streamInfo->listenerIndex[listenerVirtualID] < iMPR.inputPlugs) {
                 readQuadlet(listenerLocalID, IPCR + 4 * streamInfo->listenerIndex[listenerVirtualID],
                             &iPCR);
                    if (iPCR.pointToPointConnections > 0) {   /* Adjust listener's connection count */
                         iPCR.pointToPointConnections--;
                         programInputPlug(listenerLocalID,
                                          streamInfo->listenerIndex[listenerVirtualID], &iPCR);
                      }
                   }
                }
             }
          }
       deallocateResources(streamInfo);                          /* Release isochronous resources */
    }

}
```

8.6.6 Autonomous connection teardown

Although isochronous stream connections are usually terminated explicitly, by means of a LEAVE request, the following unexpected events might require bridge portals to delete all or part of a connection:

— The stream lifetime (for a particular listener) expires.
— The path from a talking portal to a listening portal is severed because one or both portals (or an intervening node) disconnects from the bus.
— The path from the talker to a listening portal is severed.
— The path from a talking portal to a listener is severed.

The talking portal connected to the listener's bus monitors the listener's stream lifetimes for expiration. When a stream's lifetime decrements to zero, the talking portal shall create a LEAVE request formatted as if originated by the controller that established the stream connection and process it as specified by 8.6.3.

All of the other events that trigger autonomous connection teardown involve the unexpected disconnection of one or more nodes from the local bus. The actions taken differ according to what type of portal observes the disconnection—a talking or listening portal or a portal that is a reallocation proxy, only.

Subsequent to a bus reset, a talking portal shall compare the virtual ID of each disconnected node to its list of local endpoint listeners and listening portals. If the disconnected node was a listening portal, the talking portal shall delete it from the list. Otherwise, if a valid global node ID is available for the controller that established the stream connection to the endpoint listener, the talking portal shall transmit a STREAM STATUS message with a *result* of CONNECTION DELETED to the controller and then delete the endpoint listener from its list. If the talking portal's list of local endpoint listeners and listening portals is empty, the portal shall create a TEARDOWN message marked for upstream transmission (*i.e.*, towards the talker) and forward it to its co-portal.

A listening portal or a reallocation-only proxy connected to the talker's bus first shall compare the virtual ID of each disconnected node to its list of local endpoint listeners. If the disconnected node was an endpoint listener and the global node ID of the controller that established the stream connection is valid, the portal shall transmit a STREAM STATUS message with a *result* of CONNECTION DELETED to the controller and then delete the endpoint listener from its list. If the endpoint talker is also connected to the bus and uses IEC-61883 connection management protocol, the portal shall decrement the talker's point-to-point connection counter. If the disconnected node was an endpoint talker or talking portal, the listening portal shall decrement all local endpoint listeners' IEC-61883 point-to-point connection counters (if implemented), transmit a STREAM STATUS message with a *result* of CONNECTION DELETED to each controller that established a stream connection (if the controller's global node ID is valid) and delete each node from the portal's list of local endpoint listeners. Once no endpoint listeners remain, the portal shall create a TEARDOWN message marked for downstream transmission (*i.e.*, away from the talker) and forward it to its co-portal.

Table 8-11 specifies the functional behavior of a bridge portal that discovers, subsequent to bus reset, that one or more nodes have been disconnected.

Table 8-11 — Bridge portal actions upon disconnection of a local node (Sheet 1 of 3)

```
#include "csr.h"
#include "global.h"
#include "packets.h"

VOID nodeUnplugged(BYTE unpluggedVirtualID) {

   BOOLEAN talkerPCR;
   BYTE listenerLocalID, listenerVirtualID, talkerLocalID, talkerVirtualID;
   DOUBLET talkerBusID;
   INT i;
   IMPR_CSR iMPR;
   IPCR_CSR iPCR;
   OMPR_CSR oMPR;
   OPCR_CSR oPCR;

   for (i = 0; i < bridgeCapabilities.streams; i++) { /* Search all active streams */
      if (streamInfo[i].channel == UNASSIGNED_CHANNEL)
         continue;                                   /* Skip inactive stream descriptors */
      talkerPCR = FALSE;                             /* Probable assumption */
      if (streamInfo[i].talkerID != 0xFFFF) {        /* Do we know which bus the talker is on? */
         talkerBusID = streamInfo[i].talkerID >> 6;
         if (clanInfo.busID == talkerBusID) {        /* Check talker IEC 61883-1 capabilities */
            talkerVirtualID = streamInfo[i].talkerID & 0x3f;
            talkerLocalID = virtualToPhysical[talkerVirtualID];
            if (talkerPCR = (streamInfo[i].talkerIndex <= 30)) {
               readQuadlet(talkerLocalID, OMPR, &oMPR);
               talkerPCR = (streamInfo[i].talkerIndex < oMPR.outputPlugs);
            }
         }
      }
      switch (streamInfo[i].portalRole) {
         case LISTENER:                              /* Any bus, forwards stream to co-portal */
        case REALLOCATION_ONLY:                     /* Talker's bus only---and the stream stops here! */
            if ((streamInfo[i].localListeners & (OCTLET) 1 << unpluggedVirtualID) != 0) {
               deleteLocalListener(unpluggedVirtualID, &streamInfo[i]);
             if (talkerPCR) {                      /* Decrement connection count if talker on this bus */
                  readQuadlet(talkerLocalID, OPCR + 4 * streamInfo[i].talkerIndex, &oPCR);
                  if (oPCR.pointToPointConnections > 0) {
                     oPCR.pointToPointConnections--;
                     programOutputPlug(talkerLocalID, streamInfo[i].talkerIndex, &oPCR);
                   if (   oPCR.pointToPointConnections == 0     /* No remaining listeners on bus? */
                    || (   streamInfo->localListeners == 0   /* None of "our" local listeners left? */
                       && streamInfo->portalRole == REALLOCATION_ONLY)) /* Not forwarding stream? */
                     deallocateStreamDescriptor(streamInfo);   /* OK, release internal resources */
                  }
               }
            } else if (   streamInfo[i].talkerID == 0xFFFF  /* Do we know the talker's identity? */
```

Table 8-11 — Bridge portal actions upon disconnection of a local node (Sheet 2 of 3)

```
                        || clanInfo.busID != talkerBusID)    /* If so, is this the talker's bus? */
               break;                                  /* Nope, continue with next stream descriptor */
            else if (streamInfo[i].talkerID & 0x3F == unpluggedVirtualID) {   /* Talker disconnected? */
                 for (listenerVirtualID = 0; listenerVirtualID < 63; listenerVirtualID++)
                    if ((streamInfo[i].localListeners & (OCTLET) 1 << listenerVirtualID) != 0) {
                       if (streamInfo[i].listenerIndex[listenerVirtualID] <= 30) {
                          listenerLocalID = virtualToPhysical[listenerVirtualID];
                          readQuadlet(listenerLocalID, IMPR, &iMPR);
                          if (streamInfo[i].listenerIndex[listenerVirtualID] < iMPR.inputPlugs) {
                             readQuadlet(listenerLocalID,
                                         IPCR + 4 * streamInfo[i].listenerIndex[listenerVirtualID],
                                         &iPCR);
                         if (iPCR.pointToPointConnections > 0) {   /* Adjust listener's connection count */
                                   iPCR.pointToPointConnections--;
                                   programInputPlug(listenerLocalID,
                                                 streamInfo[i].listenerIndex[listenerVirtualID], &iPCR);
                             }
                          }
                       }
                       deleteLocalListener(listenerVirtualID, &streamInfo[i]);
                    }
               if (streamInfo[i].portalRole == LISTENER)      /* Tear down all downstream connections */
                    teardownRequest(FALSE, &streamInfo[i]);
                 deallocateStreamDescriptor(&streamInfo[i]);
               }
               break;

            case TALKER:                                          /* Any bus---except talker's */
               if ((streamInfo[i].localListeners & (OCTLET) 1 << unpluggedVirtualID) != 0)
                  deleteLocalListener(unpluggedVirtualID, &streamInfo[i]);
               if (   streamInfo[i].localListeners == 0                /* Anyone still listening? */
                 && streamInfo[i].talkerID != 0xFFFF) {             /* Do we know where the talker is? */
                  teardownRequest(TRUE, &streamInfo[i]);                /* OK, tear down upstream */
                  deallocateStreamDescriptor(&streamInfo[i]);
               }
               break;
         }
      }

}

VOID deleteLocalListener(BYTE listenerVirtualID, STREAM_INFO *streamInfo) {

   STREAM_MSG statusMsg;

   if (streamInfo->controllerID[listenerVirtualID] != 0xFFFF) {   /* Controller global node ID OK? */
      memset(&statusMsg, 0, sizeof(statusMsg));
      statusMsg.opcode = STREAM_STATUS;
      statusMsg.result = CONNECTION_DELETED;
      statusMsg.responderEUI64 = ownEUI64;
      statusMsg.talkerEUI64 = streamInfo->talkerEUI64;
      statusMsg.talkerIndex = streamInfo->talkerIndex;
      transmitMsg(streamInfo->controllerID[listenerVirtualID], MESSAGE_RESPONSE, NO_SNARF,
                  &statusMsg);
   }
   streamInfo->listenerIndex[listenerVirtualID] = 0;              /* Remove listener from stream */
   streamInfo->controllerID[listenerVirtualID] = 0xFFFF;
   streamInfo->lifetime[listenerVirtualID] = 0;
   streamInfo->localListeners &= ~((OCTLET) 1 << listenerVirtualID);

}

VOID teardownRequest(BOOLEAN upstream, STREAM_INFO *streamInfo) {

   STREAM_MSG teardownMsg;
```

Table 8-11 — Bridge portal actions upon disconnection of a local node (Sheet 3 of 3)

```
   memset(&teardownMsg, 0, sizeof(teardownMsg));
   teardownMsg.opcode = TEARDOWN;
   teardownMsg.result = PENDING;
   teardownMsg.requesterEUI64 = ownEUI64;
   teardownMsg.talkerEUI64 = streamInfo->talkerEUI64;
   teardownMsg.talkerIndex = streamInfo->talkerIndex;
   teardownMsg.controllerID = 0xFFFF;
   teardownMsg.talkerID = streamInfo->talkerID;
   teardownMsg.listenerID = 0xFFFF;
   teardownMsg.upstream = upstream;
   if (upstream)
      forwardMsg(teardownMsg.talkerID, MESSAGE_RESPONSE, SNARF_ALL, &teardownMsg);
   else
      forwardMsg(0xFFFF, MESSAGE_RESPONSE, SNARF_ALL, &teardownMsg);

}
```

8.7 Common Isochronous Packet (CIP) format headers

The bridge is responsible to both filter and transform isochronous subactions with respect to channel numbers, as described in the preceding clauses. Isochronous subactions, if they utilize the Common Isochronous Packet (CIP) two-quadlet header format specified by IEC 61883-1 require additional transformations of source physical ID and optional time stamp data contained within their data payload. The transformations are explained below.

Isochronous subactions with a two-quadlet CIP header contain a field, *sid*, which shall be set to the physical ID of the talking portal.

Two-quadlet CIP headers can contain absolute time stamp information or indicate its presence elsewhere in the packet's data payload. Absolute time stamps may be found in one or more places in isochronous subactions that utilize the two-quadlet CIP header format:

— The *syt* field of the second quadlet of the CIP header if the *fmt* field in that quadlet has a value between zero and $1F_{16}$, inclusive.
— The *cycle_count* and *cycle_offset* fields of all of the source packet headers (SPH) within the isochronous subaction.

Both of these time stamps are specified as absolute values that specify a future cycle time. Since isochronous subactions experience a constant delay when routed through a bridge it is sufficient to transform the time stamps by the addition of this constant plus the difference in cycle times perceived by the portal and its co-portal. If the *latency* field in the Bridge_Capabilities entry in the listening portal's configuration ROM is nonzero, it represents the isochronous delay, in units of 125 μs cycles. Otherwise, the isochronous delay was determined by the bridge when the stream was set up. The difference in cycle time between the two sides of the bus, *delta*, is measured by implementation-dependent means but is defined by the following formula (shown in C pseudocode notation):

$$delta = \text{CYCLE\_TIME}.cycle\_count_{talking\_portal} - \text{CYCLE\_TIME}.cycle\_count_{listening\_portal};$$

When the *syt* field contains time-stamp information, the transformation shall be performed by applying the following formula:

$$syt_{transmitted} = (syt_{observed} + ((latency + delta) << 12)) \,\&\, 0x0000FFFF;$$

Because IEEE 1394 constrains *cycle_count* to the range zero to 7999, inclusive, the transformation of the *cycle_count* component of the source packet header differs. The addition of the constant isochronous delay to the *cycle_count* shall be performed modulus 8000, as shown in the following formula:

$$cycle\_count_{transmitted} = (cycle\_count_{observed} + latency + delta) \,\%\, 8000;$$

When the *sph* bit in the CIP header is one, there may be zero, one or multiple source packets (each with its own header) within a single isochronous subaction.

9. Operations in a bridged environment

Although it is possible to deduce, from other parts of this standard, the behaviors required of devices that operate in a bridged environment, this clause collects in one place all normative requirements for such devices, whether bridge-aware or legacy, exclusive of bridge portals themselves.

9.1 CSR architecture assumptions

In order for devices separated by a heterogeneous bridge to successfully communicate with each other, it is essential that each bus conform to certain assumptions about the format and meaning of the bus information block. Specifically, the location and usage of the *max_rec*, *max_ROM*, *node_vendor_ID*, *chip_ID_hi,* and *chip_ID_lo* fields[22] shall be identical to that specified by IEEE 1394.

9.2 Bridge-aware devices

A bridge-aware device is one that complies with the requirements of 9.2 of this standard. The salient features of a bridge-aware device are as follows:

— The ability to distinguish between local node IDs, whose scope is restricted to the local bus, and global node IDs that reference a remote node (or indirectly reference a local node). The most significant ten bits of a node ID differentiate local and a global node IDs.
— Different split transaction time-out values for requests addressed to local nodes and those addressed to remote nodes. The remote time-out value can be significantly greater than the local bus split time-out.
— The ability to generate commands addressed to bridge portals. The commands are identified by the data payload of the packet and its destination CSR; their intended recipient (or recipients) are identified by the packet's destination address and the value of the *snarf* field in the packet header of block write requests.
— Comprehension of new response packet header fields, such as *proxy_ID* and *ext_rcode*, and new response codes;
— Implementation of the QUARANTINE register (see 5.2.1).
— Implementation of the MESSAGE_REQUEST and MESSAGE_RESPONSE registers (see IEEE Std 1212-2001).
— Particular behaviors in response to bus reset, notably self-quarantine of global subactions and the possible invalidation of any global node IDs cached by the device.
— For bridge-aware devices that manage isochronous streams that pass through bridges (the device itself is not necessarily either a talker or listener), the ability to utilize stream connection management procedures defined by this standard.
— Recognition of commands originated by bridge portals and intended for bridge-aware nodes.

NOTE—In addition to the above requirements, a device intended to control remote devices should implement some method of device discovery that operates across bridges; one such method is described in Annex E.

Immediately subsequent to a bus reset, a bridge-aware device should defer all nonessential operations (for example, reads of the bus information block of newly inserted nodes) until after its QUARANTINE.*orphan* bit has been zeroed. This restricts bus traffic in the hope of permitting the coordinator to complete its work rapidly.

The subclauses below specify the required behaviors and characteristics of a bridge-aware device in greater detail.

9.2.1 Configuration ROM bus information block

The ***bridge_aware*** bit (abbreviated as *b* in Figure 5-2), shall be one to indicate the node complies with the requirements for a bridge-aware device, as specified in 9.2 and elsewhere within this standard.

[22] The last three fields are collectively labeled the *eui_64* field by IEEE Std 1212-2001.

9.2.2 Remote time-out

The remote time-out value enables the detection of remote transaction errors; it is the maximum time permitted for the receipt of a response subaction after the transmission of a request subaction. After this time, a requester shall terminate the request with a time-out error; the transaction's label, *tl*, is eligible for reuse (see IEEE 1394 for more information). For a requester, the time-out period commences when *ack_pending* is received in response to a request subaction.

The value of remote time-out between any two nodes is determined by means of a TIMEOUT message (see 6.6.1). A remote time-out value obtained for any node on a remote bus is valid for all nodes on the same bus. When a bridge-aware node times a request subaction addressed to a remote node, it shall use a remote time-out value no less than that returned by a response to a TIMEOUT message addressed to any node on the same bus. Once a remote time-out value is obtained by means of a TIMEOUT message, it remains valid until a quarantine period commences. When quarantine ends, a node's identity and remote time-out value can be reconfirmed by transmitting a TIMEOUT request to the node.

NOTE—If the transmitter of the TIMEOUT request message does not receive a TIMEOUT response message within an implementation-dependent time limit, another TIMEOUT request message, with a different *signature* value, should be transmitted.

9.2.3 Bus reset and quarantine

Upon detection of a bus reset, local subactions queued for transmission and pending local transactions shall be handled as specified by IEEE 1394. A local transaction is one initiated by a local request subaction. A pending transaction is one whose request subaction was initially acknowledged by *ack_pending* and for which a response subaction has yet to be transmitted. This includes all outstanding request subactions, whether they are held by the link or transaction layers or have already been indicated to an application. In addition, a bridge-aware node shall hold in suspense all global subactions and all pending global transactions until the value of QUARANTINE.*orphan* is determined. A global transaction is one initiated by a global request subaction.

The disposition of global subactions and transactions held in suspense is determined by an analysis of the self-ID packets observed subsequent to bus reset and the resultant value of QUARANTINE.*orphan* (see 5.2.1). If the *orphan* bit is zero, no quarantine condition exists—the bridge-aware node may resume normal processing of all global subactions and all pending global transactions.

Otherwise, when QUARANTINE.*orphan* is one, a quarantine condition exists; bridge-aware nodes shall not transmit global subactions and shall cancel pending global transactions. Quarantine persists across subsequent bus resets; once created, it shall end only when the *orphan* bit is zeroed. The coordinator signals the end of quarantine by a broadcast write request to the QUARANTINE register with a zero *orphan* bit. Although asynchronous broadcast is usually reliable, it is not confirmed and therefore possible for a bridge-aware node to fail to observe the end of the quarantine period. A bridge-aware node may interrogate the coordinator to determine whether a quarantine condition exists by reading the coordinator's QUARANTINE register. If the coordinator's *orphan* bit is zero, the bridge-aware node shall update its own QUARANTINE register with the contents of the coordinator's register and terminate the quarantine period as described below. A bridge-aware node shall not read the coordinator's QUARANTINE register until at least two seconds have elapsed since the most recent bus reset and thereafter should not read that register at intervals shorter than one second.

If there is no coordinator to signal the end of quarantine (no bridge portals are connected to the same bus as the bridge-aware node), QUARANTINE.*orphan* shall remain one.

When the *orphan* bit is zeroed, a bridge-aware node shall mark all cached global node IDs to indicate that each might not address the node identified by the EUI-64 previously associated with the global node ID. A bridge-aware node shall not transmit request subactions addressed to a global node ID that could affect the node's state[23] until the node's EUI-64 has been verified. The EUI-64 associated with a global node ID can be determined by transmitting a TIMEOUT message addressed to the node's MESSAGE_REQUEST register.

[23] Write and lock requests often affect the state of the recipient, but it is possible for a read request to alter the state. Read requests addressed to the contiguous kilobyte of configuration ROM, FFFF F000 0400_{16} through FFFF F000 $07FF_{16}$, inclusive, are known to leave the node's state unaltered.

If a bridge-aware node has no request subactions pending for a global node ID, it should defer verification of the remote node's EUI-64 until there is a need to transmit a remote request to the node. This might reduce traffic congestion through bridges that could otherwise occur if bridge-aware nodes attempt to revalidate EUI-64s for all global node IDs as soon as quarantine ends.

9.2.4 Serial Bus event indication (SB_EVENT.*indication*)

The following values (in addition to those already specified by IEEE 1394) are defined for the SB_EVENT.*indication* service primitive's BUS_EVENT parameter:

— QUARANTINE. Subsequent to a bus reset, the node's QUARANTINE register *net_update* (if implemented) and *orphan* bits have changed from zero to one.
— NET UPDATE COMPLETE. The node's QUARANTINE.*net_update* bit has changed from one to zero.
— GLOBAL ID ASSIGNED. The node's QUARANTINE.*orphan* bit has changed from one to zero.

9.2.5 Lock operations

Although bridges route remote lock requests and responses, no net-wide synchronization event is defined by this standard equivalent to bus reset on a local bus. As a consequence, if remote lock operations are used to manage shared resources, the application shall provide a method to release resources acquired by a node if that node is subsequently disconnected from the net or fails to release the resources when no longer in use.

9.3 Legacy devices

A legacy device is any Serial Bus device that does not comply with the requirements of 9.2. Legacy devices are ignorant of the distinction between node IDs usable on the local bus and global node IDs that reference remote nodes.

Bridge portals shall reject remote read and remote lock requests originated by a legacy device (see 7.1 for details) because such requests are unusable across bridges unless the originator is bridge-aware.

Bridge portals provide limited support for remote write requests originated by legacy devices by means of a "posted write" mechanism. A bridge portal shall provide successful response completion to the legacy device (either *ack_complete* or *resp_complete*) and shall forward the write request to its intended destination. When a response is ultimately returned to complete the transaction, the bridge portal discards it.

Bridge portals forward response subactions transmitted by legacy devices to their intended destination.

NOTE—Dependent upon the application protocol used, some legacy devices might be able to operate successfully in a bridged environment.

9.4 TIMEOUT message operations

The TIMEOUT message serves three purposes: a) it verifies the correlation between a global node ID and an EUI-64, b) it calculates the remote time-out value for requests addressed to the remote node and c) it determines the maximum data payload that may be transmitted to or received from the remote node. Because a TIMEOUT message's *snarf* field is set to three, all bridge portals—both entry and exit—process the message on its route from source toward destination and back to the source. The principal function of the entry portals is to use the *source_portal* field in the packet header to derive the maximum inter-portal transmission speed for subactions addressed to the bus identified by the TIMEOUT message's *source_bus_ID* and to use this information to update the TIMEOUT message's *max_remote* field. The principal function of the exit portals is to accumulate the maximum request or response forward time into the TIMEOUT message's

remote_timeout field. The terminal exit portal adds its local bus split time-out to *remote_timeout* and transmits the message to the originator's MESSAGE_RESPONSE register. The functional behavior of a bridge portal that intercepts a TIMEOUT request message is expressed in detail by Table 9-1.

Table 9-1 — Bridge portal actions on receipt of a TIMEOUT message

```
#include "csr.h"
#include "global.h"
#include "packets.h"

VOID timeout(PACKET *packet) {

   BOOLEAN coportal;                                  /* TRUE if message goes to co-portal next */
   BYTE pathPayload, speed;
   NODE_ID destination;
   TIMEOUT_MSG *timeoutMsg = (VOID *) &packet->data[2];     /* Allow for IEEE 1212 message header */

   if (routeMap[packet->destination.busID] == FORWARD) {    /* Are we an entry portal? */
      coportal = TRUE;                                      /* Send to co-portal when we finish */
      if (packet->source.busID == 0x3FF) {                  /* Are we the initial entry portal? */
         speed = pathSpeed(ownLocalID, packet->source.localID);   /* Yes, figure out path speed */
         packet->source.busID = clanInfo.busID;             /* Convert source ID to global ID */
         packet->source.localID = physicalToVirtual[packet->source.localID];
     } else {                                           /* No, update inter-portal speed information */
        speed = pathSpeed(ownLocalID, packet->sourcePortal);    /* path speed from previous portal */
        maxInterportalSpeed[packet->source.busID] = speed; /* Optimize future inter-portal speed */
      }
      if (speed < 4)
         pathPayload = 8 + speed;                           /* S100 through S800, inclusive */
      else
        pathPayload = 0xB;                                  /* S1600 and faster limited to 4096 bytes */
      timeoutMsg->maxRemote = MIN(timeoutMsg->maxRemote, pathPayload);
   } else {                                                 /* We are an exit portal */
     coportal = FALSE;                                     /* Transmit on local bus when we finish */
      if (packet->destinationOffset == MESSAGE_REQUEST) {      /* Request forward times? */
         timeoutMsg->remoteTimeout += maxRequestForwardTime;
         timeoutMsg->hopCount++;                            /* Count each bridge on the route */
        if (packet->destination.busID == clanInfo.busID) {    /* Are we the terminal exit portal? */
            timeoutMsg->remoteTimeout += 8000 * splitTimeout.seconds + splitTimeout.cycles;
           timeoutMsg->result = OK;                         /* Message got to destination bus OK */
         timeoutMsg->responderEUI64 = virtualIDMap[packet->destination.localID]; /* NOT our EUI-64 */
            destination = packet->destination;              /* Save copy of destination ID */
            packet->destination = packet->source;           /* Swap source and destination IDs */
            packet->source = destination;
            packet->proxy = ownGlobalID;                    /* Add our own signature */
            packet->destinationOffset = MESSAGE_RESPONSE;
           coportal = TRUE;                                 /* Send the message backs to its source */
        } else                                              /* Not terminal exit, keep forwarding */
         packet->sourcePortal = ownLocalID;             /* Enables inter-portal speed optimization */
      } else if (packet->destinationOffset == MESSAGE_RESPONSE) { /* Response forward times? */
         timeoutMsg->remoteTimeout += maxResponseForwardTime;
        if (packet->destination.busID == clanInfo.busID) {    /* Are we the terminal exit portal? */
          packet->destination.busID = 0x3FF;                /* Convert destination ID to local ID */
           packet->destination.localID = virtualToPhysical[packet->destination.localID];
         }
      }
   }
   queuePacket(TRUE, coportal);                             /* Transfer TIMEOUT message onwards */

}
```

9.5 Modifications to the BUS_TIME and CYCLE_TIME registers

The BUS_TIME register may be modified only by the coordinator or the bus manager or, if neither a bus manager nor any bridge portals are present on the bus, the isochronous resource manager. Unless made by the coordinator, all modifications of the BUS_TIME register shall be effected by broadcast write requests. If no bridge portals are connected to the bus, the bus manager or, in the absence of a bus manager, the isochronous resource manager, should broadcast the value of its BUS_TIME register whenever its least significant seven bits wrap to zero or when a node is connected to the bus.

NOTE—The paragraph above replaces the normative requirements of IEEE Std 1394-1995 with respect to initialization of the BUS_TIME register.

While QUARANTINE.*net_update* bit is zero, a coordinator shall not broadcast write requests to the BUS_TIME register. During net update, the coordinator shall broadcast the survivor clan's bus time to all nodes' BUS_TIME register. Before zeroing a newly connected node's QUARANTINE.*orphan* bit, the coordinator shall update the node's BUS_TIME register with the value of its own BUS_TIME register by means of a write request addressed to the node.

A coordinator, whose QUARANTINE.*net_update* bit is zero, that observes a change, other than as a result of normal increment, to its own BUS_TIME or CYCLE_TIME registers should set its *coportal_update_required* flag to TRUE, set its *brdg* variable to three, and initiate an arbitrated (short) bus reset in order to force net update. These actions should be taken as soon as possible after the unanticipated change and without regard for the recommendations of IEEE Std 1394a-2000 with respect to minimum intervals between successive bus resets.

9.6 Remote access to core and bus-dependent CSRs

In some cases, remote access to certain bus-dependent CSRs might be inadvisable because the information necessary to correctly alter or interpret the contents of the register might be unavailable to a node not connected to the same bus. For example, the BUS_MANAGER_ID, BANDWIDTH_AVAILABLE, and CHANNELS_AVAILABLE register are meaningful only at the isochronous resource manager, but a remote node does not have access to the self-ID packets that identify the isochronous resource manager. Unless the implementer undertakes a thorough analysis of the consequences of remote access, remote access to the CSRs identified by Table 9-2 is strongly discouraged.

Table 9-2 — Core and bus-dependent registers inadvisable for remote access

Offset	Name	Comment
0	STATE_CLEAR	The information in these registers is entirely bus-dependent.
4	STATE_SET	
8	NODE_IDS	The *bus_ID* field is fixed at $3FF_{16}$ by this standard; the *local_ID* field and remaining bus-dependent fields require bus-dependent information to be modified correctly.
$18_{16} - 1C_{16}$	SPLIT_TIMEOUT	Information necessary to establish a meaningful split time-out depends upon local bus conditions.
200_{16}	CYCLE_TIME	Bridge portals manage net cycle time distribution and synchronization.
204_{16}	BUS_TIME	
208_{16}	POWER_FAIL_IMMINENT	Power management is restricted to the local bus.
$20C_{16}$	POWER_SOURCE	
210_{16}	BUSY_TIMEOUT	Impractical to modify this register remotely because it is initialized by bus reset.
214_{16}	QUARANTINE	Local bus quarantine requires bus-dependent information and is autonomously managed by bridge-aware nodes and bridge portals.

Table 9-2 — Core and bus-dependent registers inadvisable for remote access *(continued)*

Offset	Name	Comment
218_{16}	PRIORITY_BUDGET	The sum of *pri_req* from all PRIORITY_BUDGET registers on the bus is constrained to be less than or equal to 63 minus the number of nodes on the bus; it is not feasible to monitor the node count remotely.
$21C_{16}$	BUS_MANAGER_ID	Valid only at the isochronous resource manager, whose identity cannot be determined by a remote node.
220_{16}	BANDWIDTH_AVAILABLE	
$224_{16} – 228_{16}$	CHANNELS_AVAILABLE	
234_{16}	BROADCAST_CHANNEL	
$1C00_{16} – 1DFC_{16}$	VIRTUAL_ID_MAP	The distributed algorithms implemented by bridge portals to manage this information do not function correctly remotely.
$1E00_{16} – 1EFC_{16}$	ROUTE_MAP	
$1F00_{16} – 1F04_{16}$	CLAN_EUI_64	
$1F08_{16}$	CLAN_INFO	

A terminal exit portal may prevent write or lock access to any of the above-mentioned CSRs based upon bus-dependent knowledge beyond the scope of this standard.

10. Net update

This clause specifies the cooperative behavior of bridge portals within a net of interconnected Serial Buses in response to a change in net topology. In most cases, non-bridge nodes can be inserted into or removed from a bus within the net without altering net topology, but when a bridge portal is inserted or removed, the net configuration and routing tables require update.

Wherever this clause specifies bus reset, an arbitrated (short) bus reset shall be initiated as soon as possible and without regard for the recommendations of IEEE Std 1394a-2000 with respect to minimum intervals between successive bus resets.

10.1 Power reset initialization

Subsequent to power reset and before either of a bridge's portals transmits a self-ID packet whose *L* bit and *brdg* field are both nonzero, both of a bridge's portals shall perform the following initialization:

a) Both portals shall establish communication with each other (*via* the implementation-dependent bridge fabric) and initialize a shared semaphore that controls the exchange of critical net management messages (a single semaphore shall govern all critical message exchanges—see 10.3).
b) Each portal's CSRs shall be set to the initial values specified in 5.2.
c) Each portal shall clear its *coportal_update_required*, *mute*, *panicking,* and *transmit_virtual_ID_map* flags to FALSE.
d) The bridge shall activate implementation-dependent phase-lock synchronization of the alpha portal's CYCLE_TIME.*cycle_offset* to the prime portal's CYCLE_TIME.*cycle_offset*.
e) Each portal shall initialize its normalized topology information (see Annex G) as if it were the only node on its local bus.

Upon completion of initialization, the bridge is configured as a solitary clan that consists of a prime portal and its alpha co-portal; neither portal has an assigned bus ID nor an assigned virtual ID. At this time, either portal may initiate an arbitrated (short) bus reset and transmit a self-ID packet whose *L* bit is one and whose *brdg* field is three.

NOTE—While power reset initialization is in progress, there are two ways for a bridge portal to conceal its capabilities: a) report an *L* bit value of zero or b) report an *L* bit value of one and a *brdg* value of zero. The first method is the simpler but the second method could be useful if the node implements unit architectures separate from its bridge portal capability. In either case, the bridge portal advertises its existence after completing power reset initialization by enabling its link layer (if not already enabled), setting its *brdg* variable to three, and initiating an arbitrated (short) bus reset.

If a bridge's portals are in separate power domains, the power state of each portal affects the behavior of the other. If either portal detects that its co-portal is unpowered, it shall either a) disable its link layer and initiate an arbitrated (short) bus reset (the self-ID packet *L* bit shall be zero) or b) disable its bridge functions and initiate an arbitrated (short) bus reset (the self-ID packet *brdg* field shall be zero). Either state shall persist so long as the co-portal remains unpowered. A portal that disables its link layer while its co-portal is unpowered shall either ignore link-on packets or enable its link but report a *brdg* value of zero. When a portal whose link layer or bridge functions are disabled because its co-portal is unpowered detects that its co-portal is powered, it shall perform the actions described at the beginning of this clause.

10.2 Bus reset operations

Bus reset might be a local event that requires little action on the part of bridge portals and bridge-aware devices or it might signal a net topology change that requires net update. All net updates are triggered by bus reset but not all bus resets cause net update. A change in net topology is associated with one or more of the following events, each of which results in a bus reset:

— Connection has been lost to a portion of the previous net because one or more bridge portals were removed.

— Connection has been established with a different net (or a different part of the existing net) because one or more bridge portals were inserted.
— Updated route information has arrived at a bridge portal on the local bus because of the insertion or removal of one or more bridge portals on a remote bus.

In the first two cases, a bus reset occurs because of the local insertion or removal, while in the last case the portal that receives the updated route information initiates an arbitrated (short) bus reset.

10.2.1 Quarantine

Upon detection of a bus reset, a bridge portal shall perform the following actions:

a) Local subactions queued for transmission and pending local transactions shall be handled as specified by IEEE 1394 (in this respect, a bridge portal behaves as a bridge-aware device; see 9.2.3 for more detail).
b) All global subactions queued for transmission and all pending global transactions shall be held in suspense until the value of QUARANTINE.*net_update* is determined. A global subaction is either a request or response subaction whose *destination_ID* or *source_ID* (or both) contains a global node ID or a GASP subaction whose *sy* value is greater than or equal to eight. A global transaction is one initiated by a request subaction whose *source_ID* contains a global node ID.
c) If the self-ID packets observed subsequent to the bus reset are inconsistent, as specified by IEEE 1394, the bridge portal shall initiate an arbitrated (short) bus reset and await the self-ID packets to follow. The portal shall not proceed to the next step until consistent self-ID packets are received.
d) Once a consistent set of self-ID packets has been observed, the bridge portal shall determine the value of QUARANTINE.*net_update* (as specified by 5.2.1) to ascertain both whether quarantine exists and the identity of the coordinator, which is the bridge portal with the largest physical ID. The coordinator retains its role until a subsequent bus reset.
e) If the portal is mute, it shall make certain that the *coportal_update_required* flag is FALSE. If the portal is not mute and *coportal_update_required* is TRUE, it shall remain TRUE. Otherwise, if the portal is not mute and *coportal_update_required* is FALSE, it shall determine the value of *coportal_update_required* from the self-ID packets. The portal shall determine the flag's value by the same algorithm used to derive the value of QUARANTINE.*net_update* (see 5.2.1) except that the bridge portal's own self-ID packet shall be excluded from the analysis.
f) If QUARANTINE.*net_update* is zero, net topology has not changed and net update is not underway. The bridge portal shall resume normal operation and shall disregard the remainder of 10.2 until invoked from the beginning by a subsequent bus reset. The coordinator might be required to assign virtual IDs to newly inserted nodes, as specified by 11.1.
g) Otherwise, when QUARANTINE.*net_update* is one, the bridge portal shall discard all global subactions queued for transmission and cancel all pending global transactions. The coordinator shall initiate or resume net update, as described below.

Net update requires the coordinator to collect and collate information from all of the bridge portals in order to resolve net topology to a consistent state. Two different types of allocation maps are derived from all of the bridge portals' route maps: clan allocation maps and a net allocation map. Clan allocation maps, one calculated for each clan represented on the bus, are ultimately merged into a single net allocation map, which is then distributed to all of the bridge portals. Once this is accomplished, the coordinator signals the completion of net update to bridge portals. These steps are described in the remainder of 10.2.

Because of time-critical nature of net update, while QUARANTINE.*net_update* is one, the coordinator may set its own PRIORITY_BUDGET.*pri_req* to a value less than or equal to 63 minus the number of nodes on the local bus. This permits the coordinator to use priority arbitration both for read requests as it gathers clan and route map information from all the portals and for write requests when it transmits UPDATE ROUTES messages to the portals.

10.2.2 Loop detection and elimination

The first step in net update is the identification and elimination, within each clan, of routing loops. A routing loop exists when more than one alpha portal from the same clan is connected to the bus. If loops are detected within a clan, the coordinator mutes one or more of the clan's alpha portals to eliminate the loops (see 10.4). In order to promote stability of cycle time distribution from the net cycle master, the coordinator mutes the alpha portals farthest from the net cycle master, as measured by the count of bridge hops to the clan's prime portal. While the coordinator checks for routing loops within a clan, it also determines whether net update might be in a pathological state that will not complete correctly. If the value of an alpha portal's CLAN_INFO.*hops_to_prime* is greater than the value of any subordinate portal's CLAN_INFO.*hops_to_prime*, the coordinator shall initiate net panic as specified in 10.6. Table 10-1 specifies the functional behavior of the coordinator for a single clan.

Table 10-1 — Loop detection and elimination within a clan

```
#include "csr.h"
#include "global.h"

VOID loopDetect(OCTLET targetClanEUI64) {

   BYTE i, redundantAlpha;
   OCTLET clanEUI64[63];
   CLAN_INFO_CSR clanInfo[63];      /* Local copy of other portal's CLAN_INFO */
   INT alphaPortals = 0, maxAlphaHopsToPrime, minSubordinateHopsToPrime = 1024;

   for (i = 0; i < 63; i++)         /* Process clan's portals */
      if ((brdg[i] & BRIDGE) == BRIDGE) {
         readBlock(i, CLAN_EUI_64, sizeof(OCTLET), &clanEUI64[i]);
         if (clanEUI64[i] != targetClanEUI64)
            continue;               /* Not part of the clan under investigation */
         readQuadlet(i, CLAN_INFO, &clanInfo[i]);
         if (clanInfo[i].alpha)
            alphaPortals++;
      }
   while (alphaPortals > 1) {       /* Eliminate all the clan's routing loops */
      maxAlphaHopsToPrime = 0;
      for (i = 0; i < 63; i++) {
         if (clanEUI64[i] != targetClanEUI64)
            continue;
         if (clanInfo[i].alpha)
            if (clanInfo[i].hopsToPrime > maxAlphaHopsToPrime) {
               maxAlphaHopsToPrime = clanInfo[i].hopsToPrime;
               redundantAlpha = i;  /* Candidate alpha portal to be muted */
            }
      }
      muteBridge(redundantAlpha);   /* Take the redundant alpha portal out of the loop */
      clanInfo[redundantAlpha].alpha = FALSE;
      alphaPortals--;
   }
   for (i = 0; i < 63; i++) {       /* Loops eliminated; check for panic .... */
      if (clanEUI64[i] != targetClanEUI64)
         continue;
      if (clanInfo[i].alpha)
         maxAlphaHopsToPrime = clanInfo[i].hopsToPrime;
      else if (clanInfo[i].hopsToPrime < minSubordinateHopsToPrime)
         minSubordinateHopsToPrime = clanInfo[i].hopsToPrime;
   }
   if (maxAlphaHopsToPrime > minSubordinateHopsToPrime)
      netPanic();                   /* Net topology might be inconsistent: PANIC! */

}
```

NOTE—A coordinator may eliminate loops within clans one by one or, if memory permits, in parallel. The algorithm describes this process as it applies to a single clan, but this is not intended to constrain implementations.

10.2.3 Clan allocation maps

Once loops have been eliminated, the coordinator shall collect routing information. For each clan on the bus, the coordinator creates a clan allocation map as follows:

a) The coordinator shall initialize the clan allocation map to CLEAN for all possible bus ID entries.
b) From each of the clan's portals connected to the local bus, the coordinator shall obtain both the current route map and CLAN_INFO.*bus_ID*. When processing its own clan, coordinator shall include its own route map and CLAN_INFO.*bus_ID* in the creation of the clan allocation map.
c) The coordinator shall update the entry in the clan allocation map that corresponds to a portal's CLAN_INFO.*bus_ID*. If the entry is CLEAN it shall be changed to LOCAL,[24] if the entry is VALID it shall be changed to DIRTY; otherwise, it shall remain unchanged.
d) Each portal's route map shall be used to transform the clan allocation map according to the function represented by Table 10-2. The cells in the table contain the resultant clan allocation map entry for all combinations of current clan allocation map and route map entries.

Table 10-2 — Clan allocation map transform

		Route map entry			
		CLEAN	**FORWARD**	**VALID**	**DIRTY**
Clan allocation map entry	**CLEAN**	CLEAN	VALID	CLEAN	DIRTY
	LOCAL	LOCAL	DIRTY	LOCAL	DIRTY
	VALID	VALID	DIRTY	VALID	DIRTY
	DIRTY	DIRTY	DIRTY	DIRTY	DIRTY

e) The creation of a clan allocation map proceeds to completion as route map data is obtained from each of the clan's portals on the bus. The process is repeated for other clans present on the bus.

NOTE—A coordinator may collect and process the clan allocation maps serially or, if memory permits, in parallel. Although the description of the algorithm is from the viewpoint of serial collation, this is not intended to constrain implementations.

Table 10-3 specifies the functional behavior of the algorithm used to create a clan allocation map for a particular clan. The function takes an uninitialized clan allocation map as an argument and returns the compilation of the data from the route maps of all of the clan's portals on the bus.

Table 10-3 — Creation of a clan allocation map (Sheet 1 of 2)

```
#include "global.h"
#include "csr.h"

VOID collectClanMap(OCTLET targetClanEUI64, ROUTE_STATE (*clanMap)[1023]) {

   BYTE i;
   OCTLET clanEUI64;
   CLAN_INFO_CSR clanInfo;
   ROUTE_STATE routeMap[1023];
   INT j;

   for (i = 0; i < MAX_BUS_ID; i++)            /* Initialize clan allocation map */
      *clanMap[i] = CLEAN;
   for (i = 0; i < 63; i++)                    /* Process clan's portals */
      if ((brdg[i] & BRIDGE) == BRIDGE) {
         readBlock(i, CLAN_EUI_64, sizeof(OCTLET), &clanEUI64);
         if (clanEUI64 != targetClanEUI64)
```

[24] LOCAL is used by the coordinator in the derivation of a clan allocation map; it is never observable on the bus. Since the FORWARD value used within bridge portal route maps is not needed within a clan allocation map, LOCAL can be encoded by the same value.

Table 10-3 — Creation of a clan allocation map (Sheet 2 of 2)

```
            continue;                          /* Not part of the scrutinized clan */
         readQuadlet(i, CLAN_INFO, &clanInfo);
         if (*clanMap[clanInfo.busID] == CLEAN)
            *clanMap[clanInfo.busID] = LOCAL;   /* This portal's local bus ID */
         else if (*clanMap[clanInfo.busID] == VALID)
            *clanMap[clanInfo.busID] = DIRTY;   /* Artifact from prior routing loop */
         readBlock(i, ROUTE_MAP, sizeof(routeMap), &routeMap);
         for (j = 0; j < 1023; j++)
            if (routeMap[j] == DIRTY)
               *clanMap[j] == DIRTY;            /* DIRTY overrides all other states */
            else switch (*clanMap[j]) {
               case CLEAN:
                  if (routeMap[j] == FORWARD)
                     *clanMap[j] = VALID;       /* This bus ID is routed by the clan */
                  break;

               case LOCAL:
               case VALID:
                  if (routeMap[j] == FORWARD)
                     *clanMap[j] = DIRTY;       /* Duplicated bus ID! */
                  break;
            }
      }

}
```

After all clan allocation maps have been collected, the coordinator shall attempt to select a survivor clan from those clans with an alpha portal connected to the local bus. If there is a single alpha portal whose CLAN_INFO.*preferred_clan* bit is one, its clan shall be the survivor. If there is more than one preferred clan represented by an alpha portal, the value of each clan's CLAN_EUI_64 register shall be used to select the survivor clan. The EUI-64 comparison shall be performed in byte-wise reverse order, *i.e.*, for the purpose of the unsigned comparison, the least significant byte shall be treated as if it were most significant, the most significant byte shall be treated as if it were least significant, and the interior bytes shall be compared in reverse order of their normal significance. The survivor clan shall be the clan with the largest byte-wise reversed EUI-64. When no preferred clans are represented on the local bus, the survivor shall be selected from whatever alpha portals are present. If there is more than one alpha portal, byte-wise reverse comparison of EUI-64s shall be used to select the survivor clan.

If no alpha portals are connected to the local bus, the coordinator shall select itself as the prime portal of a new clan. The new clan is a survivor clan in the sense that the coordinator retains the routing information and clan allocation map of its former clan. The new clan's *prime_portal_EUI_64* shall be set to the value of the coordinator's EUI-64, its *preferred_clan* shall either be zero or set to a field-configured value and its *hops_to_prime* shall be zero.

Once the survivor clan has been selected, the coordinator shall update all nodes' BUS_TIME registers *via* a broadcast write request of the survivor clan's bus time. If the coordinator's clan is the survivor clan, the bus time value may be taken from the coordinator's BUS_TIME register; otherwise, it shall be obtained from the survivor clan alpha portal's BUS_TIME register.

Next, the coordinator shall update the clan allocation maps it created. First, in the survivor clan's allocation map, all LOCAL entries except one (*e.g.*, the LOCAL entry that corresponds to the survivor clan's alpha portal CLAN_INFO.*bus_ID*) shall be changed to DIRTY. In the victim clan allocation maps, each LOCAL entry shall be changed to CLEAN if and only if the corresponding entry in the survivor clan's allocation map is also LOCAL; otherwise, it shall be changed to DIRTY. After all the victim clan allocation maps have been processed, if no LOCAL entry exists in the survivor clan's allocation map, the survivor clan's *bus_ID* field shall be set to $3FF_{16}$. Otherwise, *bus_ID* shall be set to the bus ID that corresponds to the LOCAL entry in the survivor clan's allocation map, which afterward shall be changed to VALID.

NOTE—In some cases, the choice of the LOCAL entry that corresponds to the survivor clan's alpha portal CLAN_INFO.*bus_ID* might not be optimal. For example, CLAN_INFO.*bus_ID* could be equal to $3FF_{16}$ or one or more of the corresponding entries in the victim clan allocation maps could be other than CLEAN or LOCAL. In either case, upon completion of net update the bus will have an unassigned bus ID. A coordinator may choose to retain the survivor clan's "best" LOCAL entry according to implementation-dependent criteria.

The coordinator shall not update its CLAN_EUI_64 or CLAN_INFO registers at this time, but shall save the survivor clan's *prime_portal_EUI_64*, *preferred_clan*, *bus_ID,* and *hops_to_prime* information for inclusion in the UPDATE ROUTES message to be sent to all portals at the conclusion of net update.

10.2.4 Net allocation map

Subsequent to the creation of a clan allocation map for each clan represented on the bus, the coordinator shall combine the clan allocation maps into a net allocation map according to the function represented by Table 10-4. The cells in the table contain the resultant net allocation map entry for all combinations of current net allocation map and clan allocation map entries. The net allocation map shall be initialized to CLEAN for all possible bus ID entries before updates from the clan allocation maps are applied.

Table 10-4 — Net allocation map transform

		Clan allocation map entry		
		CLEAN	**VALID**	**DIRTY**
Net allocation map entry	**CLEAN**	CLEAN	VALID	DIRTY
	VALID	VALID	DIRTY	DIRTY
	DIRTY	DIRTY	DIRTY	DIRTY

Table 10-5 specifies the algorithm described above. The coordinator is assumed to have initialized the net allocation map to CLEAN prior to the first invocation of `createNetMap()` and to call the function successively with each clan's allocation map.

Table 10-5 — Creation of a net allocation map

```
#include "csr.h"
#include "global.h"

VOID createNetMap(ROUTE_STATE (*netMap)[1023], ROUTE_STATE (*clanMap)[1023]) {

   INT i;

   for (i = 0; i < MAX_BUS_ID; i++)
      if (*clanMap[i] == DIRTY)
         *netMap[i] = DIRTY;           /* DIRTY besmirches everything it touches */
      else if (*netMap[i] == CLEAN) {
         if (*clanMap[i] == VALID)
            *netMap[i] = VALID;        /* VALID in one clan and unused in others --> VALID */
      } else if (*netMap[i] == VALID)
         if (*clanMap[i] != CLEAN)
            *netMap[i] = DIRTY;        /* VALID in more than one clan --> DIRTY */

}
```

The net allocation map contains the intended status of all bus IDs in the emergent net created as a result of the net topology changes. Only bus IDs that were unused in all of the clans prior to the bus reset remain unused (CLEAN) in the reconfigured net. Bus IDs previously valid in a particular clan remain valid in the reconfigured net only if they were unused in all the other clans. Collisions between bus IDs in use by more than one clan render the bus IDs dirty throughout the reconfigured net.

Once the coordinator has derived the net allocation map, it shall be transmitted to all bridge portals on the bus (including the coordinator itself) as part of an UPDATE ROUTES message (see 6.7), which shall also include the survivor clan's *prime_portal_EUI_64*, *preferred_clan*, *bus_ID,* and *hops_to_prime* information. Each portal that receives an UPDATE ROUTES message shall indicate acceptance or rejection of the message by a response of *resp_complete* or *resp_conflict_error*, respectively (see 10.5.1). If the message is accepted, prior to the return of *resp_complete* the portal shall transform its own state and, if required by its *coportal_update_required* flag, propagate the UPDATE ROUTES message to its co-portal. By this iterative propagation, the updated net configuration information eventually reaches the entire net of interconnected buses.

10.2.5 Net update completion

Once the coordinator has transmitted the UPDATE ROUTES message to each of the bridge portals on the bus and received a successful completion response from each portal, it shall transmit a broadcast write to the QUARANTINE register. The *orphan* bit shall be one and the *net_update* bit shall be zero. When QUARANTINE.*net_update* is zeroed, a bridge portal shall set its *brdg* variable to two and shall change all DIRTY entries in its route map to CLEAN. The *net_update* bit transition from one to zero ends the quarantine period and signals the completion of net update; bridge portals may transmit global subactions as permitted by the information in their route maps.

When net update completes, the coordinator shall update virtual IDs as specified by 11.1 and the alpha portal, if its CLAN_INFO.*bus_ID* is equal to $3FF_{16}$, shall request a bus ID assignment as specified by 11.2.

After QUARANTINE.*net_update* is zero, the coordinator should divide the local bus' surplus priority arbitration budget (which is equal to 63 minus the number of local nodes) among the bridge portals connected to the bus, including itself. Care should be taken to determine each portal's PRIORITY_BUDGET.*pri_max* before writing a nonzero value to PRIORITY_BUDGET.*pri_req*.

A bridge portal might fail to receive a broadcast write addressed to its QUARANTINE register and, consequently, remain unaware that *net_update* is zero. A bridge portal may use either of two methods to obtain the current value of *net_update*. If, subsequent to receipt of an UPDATE ROUTES message (and without an intervening bus reset) the bridge portal observes a subaction whose *destination_ID* contains a global node ID, the portal can reliably infer that QUARANTINE.*net_update* is zero. Alternatively, the bridge portal may start a one-second timer when an UPDATE ROUTES message is received; if the timer expires and *net_update* is one, the portal may read the coordinator's QUARANTINE register to determine the value of *net_update*. While QUARANTINE.*net_update* is one, the portal may continue to read the coordinator's QUARANTINE register at intervals no shorter than one second.

10.3 Coherency during net update

The routing, clan affiliation, and other information stored by bridge portals collectively form a distributed database that accurately and consistently describes net topology during normal operation, *i.e.*, when QUARANTINE.*net_update* is zero. During net update, it is critically important that this information remain coherent. Net update algorithms that effect coherency among the portals connected to the same bus are organized around bus reset, which is observed simultaneously by all portals and therefore provides a synchronization point. Net update algorithms that effect coherency between a portal and its co-portal are organized around a shared semaphore, which also provides a synchronization point. Although implementation details of the semaphore are left to the bridge designer, this standard specifies its use.

Three of the messages specified in 6.6.4 are critical to net update—BUS ID ANNOUNCEMENT, MUTE, and UPDATE ROUTES. A fourth message, PANIC, is also critical (it preempts net update), but its processing is sufficiently different that it is specified separately in 10.6. Each message's processing by a portal depends upon its source; Table 10-6 specifies the behavior of a portal that receives a BUS ID ANNOUNCEMENT, MUTE, or UPDATE ROUTES message from a portal connected to the local bus, as well as the source portal's actions after it receives the response subaction for the message.

Table 10-6 — Semaphore interlock for net update messages

Semaphore Status	Message	Recipient Portal Action	Source Portal Action (after response)
Available	BUS ID ANNOUNCEMENT	Once the semaphore is obtained, the portal shall process the message without interruption, transfer it to its co-portal and wait for the release of the semaphore before returning *resp_complete*.	Continue with bus ID announcements.
	MUTE		Continue with net update.
	UPDATE ROUTES		
Unavailable	BUS ID ANNOUNCEMENT	The portal shall discard the message and return *resp_conflict_error*.	Retransmit the message[a] until a successful completion response is received or a bus reset initiates or restarts net update.
	MUTE		
	UPDATE ROUTES		

[a] If the source portal is transmitting other messages in parallel, it shall suspend their transmission until successful retransmission of the refused message or bus reset.

Table 10-7 specifies the behavior of a portal that receives a BUS ID ANNOUNCEMENT, MUTE, or UPDATE ROUTES message from its co-portal.

Table 10-7 — Net update interactions for messages received from the co-portal

QUARANTINE.*net_update*	Message	Portal Action	Net Update Required
Zero	BUS ID ANNOUNCEMENT	The portal shall release the semaphore and process the message without interruption. If net update is required, the portal shall set its *brdg* variable to three and generate an arbitrated (short) bus reset to initiate or restart net update.	No
	MUTE		No
	UPDATE ROUTES		Yes
One	BUS ID ANNOUNCEMENT		Yes
	MUTE		No
	UPDATE ROUTES		Yes

Although bus reset shall not interrupt message processing (regardless of its source), if the message was received from the coordinator, bus reset cancels the message's pending transaction response; *resp_complete* is not returned to the coordinator upon completion of message processing.

If a portal, as specified by Table 10-7, initiates bus reset to start or restart net update, it may do so before it completely processes the message—but until the portal observes bus reset, it shall ignore all write requests addressed to its QUARANTINE register. In the time between initiating bus reset and the completion of updates to its own route map, the portal shall refuse all read requests addressed to its ROUTE_MAP register with *resp_conflict_error*.

10.4 Mute bridge portals

Bridge portals are muted to eliminate routing loops in the net topology. The coordinator transmits a MUTE message to an alpha portal whose routing connections with its co-portal are to be severed; the alpha portal in turn communicates the MUTE message to its co-portal. When an alpha portal receives a MUTE message from the coordinator, it shall not respond with a terminal acknowledgment; unless it transmits a busy acknowledgment, it shall transmit *ack_pending*. Next, the alpha portal shall attempt to obtain control of the semaphore shared with its co-portal. If it fails to obtain control of the semaphore, the alpha portal shall not process the MUTE message but shall transmit *resp_conflict_error* in response. Otherwise, it shall clear its *coportal_update_required* flag to FALSE, zero its CLAN_INFO.*alpha* bit, set all FORWARD entries in its route map to VALID, and transfer the MUTE message to its co-portal. Once these steps are complete, the now subordinate portal shall set its *mute* flag to TRUE, wait for the co-portal to release the semaphore, and then respond to the MUTE message with *resp_complete*.

When a bridge portal receives a MUTE message from its co-portal, it shall release the semaphore, discard all queued global subactions, set all FORWARD entries in its route map to VALID, clear its *coportal_update_required* flag to FALSE, and set its *mute* flag to TRUE. No other action is necessary; the net update in which its co-portal is a participant should propagate and ultimately reach the now mute portal.

NOTE—Between the time the portal becomes mute and the time net update arrives at its local bus, global subactions that would otherwise be forwarded by the portal receive no acknowledgment. No problem is created by the missing acknowledgment, since net update terminates all outstanding global subactions and all pending global transactions.

Mute bridge portals do not forward any asynchronous or isochronous subactions to their co-portal. Although each portal continues to respond normally to request subactions on its local bus and participate in net update (it might be the coordinator), all traffic across its internal fabric is interdicted: the bridge is effectively disabled.

A mute bridge portal autonomously resumes normal operations (unmutes) if it changes its clan allegiance during net update (see 10.5.1).

10.5 Route map updates

A bridge portal that receives an UPDATE ROUTES message typically updates its own route map and might be required to propagate the message to its co-portal. The transformation applied to the portal's route map depends upon whether the UPDATE ROUTES message was received from the coordinator or from the bridge portal's co-portal. Once a portal starts processing an UPDATE ROUTES, it shall complete the process (*e.g.*, bus reset shall not cancel the message processing even though the response subaction for the UPDATE ROUTES message is discarded).

Independent of the source of the message, the coordinator or the co-portal, a bridge portal that processes an UPDATE ROUTES message shall set CLAN_INFO.*bus_ID* to $3FF_{16}$ (unassigned) if the portal's current bus ID is marked DIRTY in the net allocation map. In a separate process (described in 11.2), the alpha portal for the bus shall obtain a new bus ID and assign it to the local bus.

The functional behavior of a bridge portal that receives an UPDATE ROUTES message, whether from the coordinator or its co-portal, is explained in narrative text in 10.5.1 and 10.5.2 and specified in detail by Table 10-8.

Table 10-8 — Route map and clan allegiance updates (Sheet 1 of 3)

```
#include "csr.h"
#include "global.h"

BYTE updateRouteMap(OCTLET newClanEUI64, ROUTE_STATE (*netMap)[1023], BOOLEAN fromCoportal,
                    BOOLEAN preferredClan, DOUBLET busID, DOUBLET hopsToPrime) {

   UNSIGNED i;
   BOOLEAN resetBus = FALSE;

   if (ownEUI64 == newClanEUI64 && hopsToPrime != 0)
      netPanic();                          /* Prime portal should be zero hops from itself */
   else if (!fromCoportal && clanEUI64 != newClanEUI64 && hopsToPrime == 1022)
      netPanic();                          /* Too many bridges in the net! */
   if (fromCoportal) {                     /* UPDATE ROUTES message from co-portal */
      releaseSemaphore();                           /* Let co-portal know we got the message */
      resetBus = TRUE;                              /* Net update is necessary */
      if (clanEUI64 != newClanEUI64)                /* Has our clan allegiance changed? */
         clanInfo.alpha = TRUE;
      clanEUI64 = newClanEUI64;                     /* Update clan allegiance ... */
      clanInfo.preferredClan = preferredClan;
      clanInfo.hopsToPrime = hopsToPrime;           /* ... and hops to prime */
      if (mute) {                                   /* Is the portal mute? */
        memset(routeMap, CLEAN, sizeof(routeMap));   /* Yes, set all route map entries to CLEAN ... */
         mute = FALSE;                              /* ... and unmute */
         coportalUpdateRequired = TRUE;             /* New information at the end of net update */
      }
```

Table 10-8 — Route map and clan allegiance updates (Sheet 2 of 3)

```
      for (i = 0; i < MAX_BUS_ID; i++)
         switch (*netMap[i]) {
            case CLEAN:
               if (routeMap[i] == FORWARD)
                  routeMap[i] = DIRTY;             /* This bus is no longer routable */
               break;

            case VALID:
               if (routeMap[i] == CLEAN)
                  routeMap[i] = FORWARD;           /* New addition to the clan */
               else if (routeMap[i] == DIRTY)
                 coportalUpdateRequired = TRUE;  /* New net update on co-portal's bus before we finish */
               break;

            case DIRTY:
               routeMap[i] = DIRTY;                /* Collision elsewhere in the net */
               break;
         }
   } else {                                 /* UPDATE ROUTES message from coordinator */
      if (mute && clanEUI64 != newClanEUI64)    /* When clan allegiance changes, prepare to unmute */
         coportalUpdateRequired = TRUE;         /* Make sure co-portal unmutes */
      if (coportalUpdateRequired)
         if (!getSemaphore())                   /* Interlock with co-portal */
            return(RESP_CONFLICT_ERROR);        /* Coordinator should wait ... */
      for (i = 0; i < MAX_BUS_ID; i++)         /* Use net allocation map to update our route map */
         switch (*netMap[i]) {
            case CLEAN:
               if (routeMap[i] == VALID)
                  routeMap[i] = DIRTY;          /* Newcomer elsewhere in the net */
               break;

            case VALID:
               if (routeMap[i] == CLEAN)
                  routeMap[i] = VALID;          /* New addition to the clan */
               break;

            case DIRTY:
               routeMap[i] = DIRTY;             /* DIRTY trumps everything else */
               break;
         }
      if (clanEUI64 != newClanEUI64)            /* Has our clan allegiance changed? */
         if (mute) {
            mute = FALSE;                       /* Clan allegiance change forces unmute */
            if (coportalPrime) {                /* Exception case */
               if (clanInfo.preferredClan == preferredClan) {  /* Equivalent clan preferences? */
                  i = sizeof(OCTLET);           /* Number of bytes in an EUI-64 */
                  do {                          /* Byte-wise reversed EUI-64 comparison */
                     i--;
                     if (((BYTE *) &newClanEUI64)[i] < ((BYTE *) &clanEUI64)[i]) {
                        resetBus = TRUE;        /* We have a winning EUI-64! */
                        break;
                     }
                  } while (i > 0);
               } else if (clanInfo.preferredClan)  /* Our clan preferred? */
                  resetBus = TRUE;              /* Yes, throw our hat in the ring */
            }
         } else
            clanInfo.alpha = FALSE;             /* We can't possibly be the alpha portal */
      if (resetBus) {                           /* Restart net update? */
        clanInfo.alpha = TRUE;                /* Declare ourselves a competitor for survivor clan */
         newClanEUI64 = clanEUI64;              /* We'll win, so adjust UPDATE ROUTES message */
         preferredClan = clanInfo.preferredClan;
         clanInfo.hopsToPrime = hopsToPrime = 1;   /* Invariant when coportalPrime is TRUE */
      } else {
         clanEUI64 = newClanEUI64;              /* Update clan allegiance */
```

Table 10-8 — Route map and clan allegiance updates (Sheet 3 of 3)

```
         clanInfo.preferredClan = preferredClan;
       clanInfo.hopsToPrime = hopsToPrime;   /* Portals on the same bus are equidistant from prime */
      }
      clanInfo.busID = busID;                  /* Local bus ID may have changed */
      if (coportalUpdateRequired) {
         if (clanInfo.alpha && ownEUI64 != clanEUI64)
            hopsToPrime--;                     /* Subordinate co-portal is closer to prime */
         else
            hopsToPrime++;                     /* Alpha co-portal is farther from prime */
         sendRouteInfo(newClanEUI64, netMap, preferredClan, hopsToPrime);
         waitSemaphoreFree();                  /* Co-portal should release the semaphore ... */
         coportalUpdateRequired = FALSE;
      }
   }
   if (routeMap[clanInfo.busID] == DIRTY)       /* Have we lost our bus ID? */
      clanInfo.busID = 0x3FF;                   /* Special value for "bus ID unassigned" */
   if (resetBus) {                              /* One or more bus IDs invalidated? */
      brdg[ownLocalID] = BRIDGE_CHANGED_STATE;  /* Insure net update commences (or resumes) */
      SB_CONTROL.request = RESET;               /* Signal changed net topology to local bus */
      return(RESP_CONFLICT_ERROR);
   } else
      return(RESP_COMPLETE);                    /* Complete pending UPDATE ROUTES write request */

}
```

10.5.1 UPDATE ROUTES message received from the coordinator

When a bridge portal receives an UPDATE ROUTES message from the coordinator, it shall not respond with a terminal acknowledgment; unless it transmits a busy acknowledgment, it shall transmit *ack_pending*. Next, the portal shall determine whether net update is in a pathological state that never completes. If the portal's EUI-64 is equal to the value of the UPDATE ROUTES message *prime_portal_EUI_64* field and the message's *hops_to_prime* field is nonzero, the portal shall initiate net panic as specified in 10.6.

A portal whose *mute* flag is TRUE shall inspect the UPDATE ROUTES message. If the values of the mute portal's CLAN_EUI_64 register and the *prime_portal_EUI_64* field from the UPDATE ROUTES message are different, the portal shall set its *coportal_update_required* flag to TRUE.

Before processing the UPDATE ROUTES message, the portal shall test its *coportal_update_required* flag. If the flag is TRUE, the portal shall attempt to obtain control of the semaphore shared with its co-portal. If the portal fails to obtain control of the semaphore, it shall not process the message but shall transmit *resp_conflict_error* in response. Otherwise, the portal shall transform its route map according to the function described by Table 10-9.

Table 10-9 — Route map transform (UPDATE ROUTES message from coordinator)

		Net allocation map entry		
		CLEAN	**VALID**	**DIRTY**
Route map entry	**CLEAN**	CLEAN	VALID	DIRTY
	FORWARD	—[a]	FORWARD	DIRTY
	VALID	DIRTY	VALID	DIRTY
	DIRTY	—[a]	—[a]	DIRTY

[a] This combination cannot occur.

After transforming its route map, a portal whose *mute* flag is FALSE shall copy the *preferred_clan* bit and the *bus_ID* and *hops_to_prime* fields from the UPDATE ROUTES message to their corresponding locations in its CLAN_INFO register and shall update its clan allegiance to match the *prime_portal_EUI_64* field from the UPDATE ROUTES message. If the

bridge portal's clan allegiance is unchanged, its status as either an alpha or subordinate portal shall also be unchanged. Otherwise, when a bridge portal changes its clan allegiance in response to an UPDATE ROUTES message received from the coordinator, it shall become a subordinate portal.

A portal whose *mute* flag is TRUE shall inspect the UPDATE ROUTES message before processing its remaining fields:

a) If the values of the mute portal's CLAN_EUI_64 register and the *prime_portal_EUI_64* field from the UPDATE ROUTES message are equal, the portal shall remain mute and shall update its CLAN_EUI_64 and CLAN_INFO registers with the information in the UPDATE ROUTES message and shall skip the remaining steps below. Otherwise, the mute portal shall clear its *mute* flag to FALSE and shall continue with the following steps.
b) If the formerly mute portal's co-portal is not a prime portal, the portal shall update its CLAN_EUI_64 and CLAN_INFO registers with the information in the UPDATE ROUTES message.
c) When the formerly mute portal's co-portal is a prime portal, if the portal's CLAN_INFO.*preferred_clan* bit and the UPDATE ROUTES message *preferred_clan* bit are equal, the portal shall modify the message as specified below if and only if the value of the portal's CLAN_EUI_64 register is larger than, in byte-wise reversed comparison, *prime_portal_EUI_64*.
d) When the formerly mute portal's co-portal is a prime portal, if the *preferred_clan* bits have different values, the portal shall modify the UPDATE ROUTES message as specified below if and only if the portal's CLAN_INFO.*preferred_clan* bit is one.
e) If none of the above conditions is satisfied, the formerly mute portal shall update its CLAN_EUI_64 and CLAN_INFO registers with the information in the UPDATE ROUTES message.

If either condition c) or d) is satisfied, the formerly mute portal's clan would have been selected as the survivor clan if it had been eligible (*i.e.*, if the portal's CLAN_INFO.*alpha* had been one). In order to effect this desirable outcome, the formerly mute portal shall set its CLAN_INFO.*alpha* bit to one, change the UPDATE ROUTES message's *prime_portal_EUI_64* field to the value of its own CLAN_EUI_64 register, change the message's *preferred_clan* bit to the value of its own CLAN_INFO.*preferred_clan* bit, change both the message's *hops_to_prime* field and its own CLAN_INFO.*hops_to_prime* field to one and copy the message's *bus_ID* field to CLAN_INFO.*bus_ID*.

Once the UPDATE ROUTES message has been processed, the bridge portal shall test its *coportal_update_required* flag. If the flag is FALSE, the portal shall transmit *resp_complete* in response to the UPDATE ROUTES message. Otherwise, an alpha portal that is not a prime portal shall decrement the message's *hops_to_prime* field while a prime or subordinate portal shall increment it. After modifying the UPDATE ROUTES message, the portal shall propagate it to its co-portal, wait for the co-portal to release the semaphore, clear the *coportal_update_required* flag to FALSE, and either initiate an arbitrated (short) bus reset or transmit *resp_complete* in response to the UPDATE ROUTES message. The portal shall initiate bus reset only if it had been mute but now intends to compete to become the survivor clan's alpha portal, in which case it shall set *brdg* to three before initiating bus reset.

NOTE—The requirements of this clause obligate a bridge portal implementation to be able to ascertain if its co-portal is prime, determine whether the semaphore is available, complete the required updates, transfer the UPDATE ROUTES message to the co-portal, and wait for the co-portal to release the semaphore—all within the time limit established by its SPLIT_TIMEOUT register.

Unless the bridge portal initiates an arbitrated (short) bus reset, after it has completed the steps above and subject to its updated route map information, the portal shall resume forwarding global subactions to its co-portal. The portal is not yet enabled to transmit global subactions.

10.5.2 UPDATE ROUTES message received from co-portal

When a bridge portal receives an UPDATE ROUTES message from its co-portal, it shall release the semaphore to confirm receipt of the message. Then it shall determine whether net update is in a pathological state that never completes. Such a state exists if either of the following conditions is met:

— The portal's EUI-64 is equal to the value of the UPDATE ROUTES message *prime_portal_EUI_64* field and the *hops_to_prime* field in the message is nonzero; or

— The value of the UPDATE ROUTES message *hops_to_prime* field equals 1023.

If either of the previous is true, the portal shall initiate net panic as specified in 10.6.

Otherwise, if net panic is not required, a portal whose *mute* flag is TRUE shall clear it to FALSE and set all of its route map entries to CLEAN. Whether the portal's *mute* flag was FALSE when the UPDATE ROUTES message was received from the co-portal or was cleared to FALSE by receipt of the message, the portal shall transform its route map according to the function specified by Table 10-10.

Table 10-10 — Route map transform (UPDATE ROUTES message from co-portal)

		Net allocation map entry		
		CLEAN	**VALID**	**DIRTY**
Route map entry	**CLEAN**	CLEAN	FORWARD	DIRTY
	FORWARD	DIRTY	FORWARD	DIRTY
	VALID	—[a]	VALID	DIRTY
	DIRTY	DIRTY	DIRTY[b]	DIRTY

[a] This combination cannot occur.
[b] Set the *coportal_update_required* flag TRUE.

In addition to updating its route map, the portal shall update its CLAN_EUI_64 register with the value of the *prime_portal_EUI_64* field in the UPDATE ROUTES message. If the bridge portal's clan allegiance is unchanged, its status as either an alpha or subordinate portal shall also be unchanged. Otherwise, when a bridge portal changes its clan allegiance in response to an UPDATE ROUTES message received from its co-portal, it shall become an alpha portal. Next, the portal shall copy the *preferred_clan* bit and the *hops_to_prime* field to their corresponding locations in its CLAN_INFO register.

The receipt of an UPDATE ROUTES message from the co-portal necessitates bus reset in order to cause net update on the portal's bus. Before initiating an arbitrated (short) bus reset, the portal shall set its *brdg* variable to three and, until the bus reset is observed, shall ignore all write requests addressed to its QUARANTINE register. In the time between the bus reset and the completion of updates to its own route map, the portal shall refuse all read requests addressed to its ROUTE_MAP register with *resp_conflict_error*.

NOTE—Usually, an UPDATE ROUTES message should not be returned to the co-portal when net update completes, since most often the UPDATE ROUTES message would not contain new information and would serve only to create perpetual net update. The *coportal_update_required* flag is used to control the propagation of UPDATE ROUTES messages back to the co-portal (see 10.2.1).

10.6 Net panic

Under extraordinary circumstances,[25] net update can enter a pathological state that will either perpetuate indefinitely or terminate with inconsistent routing information. If net update is such a state, one of the following events happens:

— Within a single clan, the coordinator observes an alpha portal whose CLAN_INFO.*hops_to_prime* is greater than a subordinate portal's CLAN_INFO.*hops_to_prime*.
— A portal receives an UPDATE ROUTES message whose *prime_portal_EUI_64* field matches the portal's EUI-64 but whose *hops_to_prime* field is nonzero.
— A portal receives an UPDATE ROUTES message from its co-portal whose *hops_to_prime* field is 1023.

[25] If extremely rapid changes in net topology break or create routing loops too quickly for net update to establish a stable net topology before the next change, the net update algorithms might malfunction. This very unlikely possibility is reliably resolved by net panic.

Exit from this non-terminating state is effected by net panic, an extreme remedy in which all bridges in the net cease operating as bridges then, once the panic condition has been communicated to all bridge portals on a bus and its immediately adjacent buses, perform power reset initialization before resumption of bridge operations. Net panic operations shall preempt all other bridge portal operations, including net update.

A bridge portal that detects any of the conditions previous shall initiate net panic by initializing its *panicking* flag to TRUE, initializing its *local_bus_panic_complete* and *adjacent_bus_panic_complete* flags to FALSE, and zeroing both its *hops_since_panic* and *brdg* variables. Zeroing *brdg* shall cause the portal to stop all bridge functions. Once these steps are complete, the portal shall broadcast a PANIC message (see 6.6.4.4) and shortly thereafter shall generate an arbitrated (short) bus reset.

When a bridge portal receives a PANIC message from its local bus (*i.e.*, not from its co-portal), it shall set its *panicking* flag to TRUE, reset its *local_bus_panic_complete* and *adjacent_bus_panic_complete* flags to FALSE, update its *hops_since_panic* variable with the value from the PANIC message, and zero its *brdg* variable. As was the case for the portal that initiated net panic, zeroing *brdg* shall cause the portal to stop all bridge functions. Once *panicking* is TRUE, the portal shall ignore any subsequent PANIC messages received from its local bus.

While a bridge portal's *panicking* flag is TRUE, it shall ignore all net management messages, whether received from its co-portal or from the local bus. Although the messages are ignored, the portal shall confirm receipt of the message as appropriate, whether by the return of *resp_complete* or by the release of the semaphore shared with its co-portal. In addition, so long as the portal's *local_bus_panic_complete* flag is FALSE, it shall monitor self-ID packets observed after bus reset. If any self-ID packet contains a nonzero *brdg* field, the portal shall broadcast a PANIC message followed by an arbitrated (short) bus reset. Otherwise, the portal shall attempt to obtain control of the semaphore shared with its co-portal. The portal shall persist in its attempts to obtain control of the semaphore until it succeeds, at which time it shall transfer a PANIC message to its co-portal. The message's *hops_since_panic* field shall contain the current value of the portal's *hops_since_panic* variable. Once the PANIC message has been transferred to the co-portal, the portal shall wait for its co-portal to release the semaphore, after which the portal shall set its *local_bus_panic_complete* flag to TRUE.

Upon receiving a PANIC message from its co-portal, a portal shall set its *adjacent_bus_panic_complete* flag to TRUE. Subsequent actions depend upon the portal's state. If its *panicking* flag is TRUE, PANIC messages have already been broadcast on the local bus and no further action is necessary. Otherwise, if the flag is FALSE, the portal shall set *panicking* to TRUE, update its *hops_since_panic* variable to be one greater than the value obtained from the PANIC message, and zero its *brdg* variable (zeroing *brdg* shall cause the portal to stop all bridge functions). If the incremented *hops_since_panic* variable is less than 1023, the portal shall reset its *local_bus_panic_complete* flag to FALSE, broadcast a PANIC message, and subsequently generate an arbitrated (short) bus reset—after which the portal shall monitor self-ID packets as described in the preceding paragraph. Otherwise, when *hops_since_panic* equals 1023, the portal shall attempt to obtain control of the semaphore shared with its co-portal. The portal shall persist in its attempts to obtain control of the semaphore until it succeeds, at which point it shall transfer a PANIC message to its co-portal. The message's *hops_since_panic* field shall contain the current value of the portal's *hops_since_panic* variable. Once the PANIC message has been transferred to the co-portal, the portal shall wait for its co-portal to release the semaphore, after which the portal shall set its *local_bus_panic_complete* flag to TRUE.

As soon as both of a portal's *local_bus_panic_complete* and *adjacent_bus_panic_complete* flags are TRUE, the portal shall perform power reset initialization (see 10.1). When both the bridge's portals have completed initialization, each shall set its *brdg* variable to three and initiate an arbitrated (short) bus reset.

NOTE—Because each portal clears its normalized bus topology information during power reset initialization, if other bridge portals are connected to the local bus, their discovery subsequent to bus reset causes the *coportal_update_required* flag to be set to TRUE. This ensures that their route information is propagated through the entire net.

11. Global node ID management

Global node IDs consist of two parts, a 10-bit bus ID and a 6-bit virtual ID. Before a node can be addressed remotely, assigned values are necessary for both. This clause specifies how the alpha portal and coordinator manage these two global node ID components.

Although there are timing relationships between bus ID and virtual ID management and the net update operations described in Clause 10., both bus ID and virtual ID management are essentially independent of net update.

11.1 Virtual ID management

Bus reset triggers updates to virtual IDs, if any. All bridge portals, including the coordinator, shall maintain a correlation between physical IDs of connected nodes and their EUI-64s. In order to minimize reads of configuration ROM, this correlation shall be derived from an analysis of self-ID packets in combination with EUI-64 data retained from the bus topology configuration just prior to the bus reset. A bus reset could also trigger net update (described in Clause 10.), in which case the coordinator shall wait until QUARANTINE.*net_update* is zero before it determines whether updates to the VIRTUAL_ID_MAP register are necessary.

NOTE—Nodes whose configuration ROM is not in the general format specified by IEEE Std 1212-2001 are not assigned virtual IDs since they lack an EUI-64 in a well-known location.

The starting point of the virtual ID update process is the current clan's virtual ID map. If there was no net update associated with the bus reset or if the coordinator's clan allegiance did not change during net update, the coordinator shall obtain the current virtual ID map from its own VIRTUAL_ID_MAP register. Otherwise, it shall first read this register from the survivor clan's alpha portal. Once the coordinator obtains a current copy of the virtual ID map, it shall compare it against the EUI-64s of currently connected nodes as follows:

a) If a connected node's EUI-64 is present in the virtual ID map, no action is necessary. A virtual ID to EUI-64 mapping exists and is still valid.

b) Newly inserted nodes require update of their bus time with the clan's bus time and assignment of a virtual ID. The coordinator shall transmit the value of its own BUS_TIME by means of a write request addressed to the node's BUS_TIME register and shall select an unused virtual ID and associate it with the newly inserted node's EUI-64.

c) When no unallocated virtual IDs are available and CLAN_INFO.*bus_ID* is not equal to $3FF_{16}$, it is necessary to abandon the current bus ID before assigning new virtual IDs to the nodes connected to the bus. The coordinator shall zero its VIRTUAL_ID_MAP register, invalidate the current bus ID by marking it DIRTY in its own ROUTE_MAP register, set its *coportal_update_required* flag to TRUE, and set its *brdg* variable to three. Without regard for the recommendations of IEEE Std 1394a-2000 with respect to minimum intervals between successive bus resets, the coordinator shall initiate an arbitrated (short) bus reset to cause net update. The coordinator selected as a result of the bus reset starts the virtual ID management process afresh.

d) If the coordinator's VIRTUAL_ID_MAP register is unchanged since the last time QUARANTINE.*orphan* was zero, there is no need to write updated information to the other bridge portals. Otherwise, the coordinator shall write an updated virtual ID map to each bridge portal; the alpha portal shall be the last to receive the virtual ID map information. If the coordinator's CLAN_INFO.*bus_ID* is not equal to $3FF_{16}$, it shall transmit a broadcast write request to zero QUARANTINE.*orphan* to signal that all nodes on the bus are mapped to valid virtual IDs.

e) If the coordinator's CLAN_INFO.*bus_ID* is equal to $3FF_{16}$, although each local node possesses a virtual ID it lacks a global node ID. In this case, the value of all nodes' QUARANTINE.*orphan* bits shall remain one until subsequent assignment of a bus ID (see 11.2).

The algorithm described in the list above is specified in more detail by Table 11-1.

Table 11-1 — Virtual ID update after bus reset

```
#include “csr.h”
#include “global.h”

VOID updateVirtualIDMap(OCTLET physicalIDMap[64], BYTE alphaPhysicalID,
                        BYTE physicalToVirtual[64], BYTE virtualToPhysical[64]) {

   BYTE i, j;

   memset(physicalToVirtual, 0x3F, sizeof(physicalToVirtual));
   memset(virtualToPhysical, 0x3F, sizeof(virtualToPhysical));
   for (i = 0; i < 63; i++) {
      if (physicalIDMap[i] != 0) {     /* Is a node present on the bus? */
         for (j = 0; j++; j < 63)
            if (virtualIDMap[j] == physicalIDMap[i]) {
               physicalToVirtual[i] = j;  /* Update mapping from PHY ID to virtual ID ... */
               virtualToPhysical[j] = i;  /* ... and virtual ID to PHY ID */
               break;                  /* Node’s EUI-64 already assigned a virtual ID */
            }
         if (j == 63) {                /* Did we find a matching EUI-64? */
            for (j = 0; j++; j < 63)   /* No, attempt to allocate a virtual ID */
               if (virtualIDMap[j] == 0)
                  break;               /* Allocate first unused virtual ID */
            if (j == 63) {             /* No available virtual ID? */
               memset(virtualIDMap, 0, sizeof(virtualIDMap));  /* Discard all virtual IDs */
               routeMap[clanInfo.busID] = DIRTY;   /* Time to get a new bus ID! */
               clanInfo.busID = 0x3FF;
               brdg[ownLocalID] = BRIDGE_CHANGED_STATE;  /* Force net update */
               SB_CONTROL.request = RESET;         /* Initiate a (short) bus reset */
               return;  /* We’ll resume virtual ID update if we remain the coordinator */
            } else {
               physicalToVirtual[i] = j;
               virtualToPhysical[j] = i;
               virtualIDMap[j] = physicalIDMap[i]; /* Update virtual ID slot with EUI-64 */
               transmitVirtualIDMap = TRUE;        /* We’ll need to tell the other portals */
            }
         }
      }
   }
   if (virtualIDMap[63] != physicalIDMap[alphaPhysicalID]) {
      virtualIDMap[63] = physicalIDMap[alphaPhysicalID];
      transmitVirtualIDMap = TRUE;
   }
   if (transmitVirtualIDMap) {           /* Did VIRTUAL_ID_MAP change? */
      for (i = 0; i < 63; i++)           /* Yes, update other portals’ VIRTUAL_ID_MAP */
         if ((brdg[i] & BRIDGE) == BRIDGE && i != ownLocalID && i != alphaPhysicalID)
            writeBlock(i, VIRTUAL_ID_MAP, sizeof(VIRTUAL_ID_MAP), virtualIDMap);
      writeBlock(alphaPhysicalID, VIRTUAL_ID_MAP, sizeof(VIRTUAL_ID_MAP), virtualIDMap);
   }
   if (clanInfo.busID = 0x3FF) {         /* Does our bus have an assigned bus ID? */
      quarantine.orphan = 0;             /* Zero our own QUARANTINE.orphan bit ... */
      writeQuadlet(BROADCAST, QUARANTINE, &quarantine);  /* ... and everyone else’s */
   }

}
```

The input parameter `physicalIDMap` is an array of EUI-64s for connected nodes, indexed by physical ID, while the input parameter `alphaPhysicalID` is the physical ID of the alpha portal.[26] The contents of the coordinator’s VIRTUAL_ID_MAP register are represented by the global variable `virtualIDMap`, which has been updated, as necessary, with information obtained from the survivor clan’s alpha portal before `updateVirtualIDMap()` is invoked. The global array `brdg` is indexed by physical ID; each entry contains the indexed node’s *brdg* field collected from

[26] The coordinator knows the physical ID of the survivor clan’s alpha portal as a consequence of net update.

self-ID packet zero after the most recent bus reset. The output parameters `physicalToVirtual` and `virtualToPhysical` are a convenient representation of the relationship between physical and virtual IDs.[27] The first is indexed by physical ID and yields a virtual ID while the second is indexed by virtual ID and provides a physical ID. They are utilized when an initial entry portal transforms a subaction's *source_ID* from a local node ID into a global node ID or when a terminal exit portal performs the inverse transformation on a subaction's *destination_ID*.

A bridge portal might fail to receive a broadcast write addressed to its QUARANTINE register and, consequently, remain unaware that *net_update* is zero. A bridge portal may use either of two methods to obtain the current value of *net_update*. If, subsequent to receipt of an UPDATE ROUTES message (and without an intervening bus reset) the bridge portal observes a subaction whose *destination_ID* contains a global node ID, the portal can reliably infer that QUARANTINE.*net_update* is zero. Alternatively, the bridge portal may start a one-second timer when an UPDATE ROUTES message is received; if the timer expires and *net_update* is zero, the portal may read the coordinator's QUARANTINE register to determine the value of *net_update*. While QUARANTINE.*net_update* is one, the portal may continue to read the coordinator's QUARANTINE register at intervals no shorter than one second.

A bridge-aware node or bridge portal whose QUARANTINE.*orphan* bit is not zero two seconds after bus reset may read the coordinator's QUARANTINE register to obtain the value of the *orphan* bit; if zero, the portal shall clear its *transmit_virtual_ID_map* flag to FALSE. Bridge-aware nodes and bridge portals should not read the coordinator's register sooner than two seconds after the most recent bus reset nor more frequently than once a second thereafter. Bridge portals should be designed to complete virtual ID assignment within two seconds of bus reset. As an alternative to reading the coordinator's QUARANTINE register, a bridge portal that observes a global subaction whose *source_ID* contains a local node ID may zero QUARANTINE.*orphan*.

11.2 Bus ID management

As a consequence of net update, a bus could have a bus ID value of $3FF_{16}$ (unassigned).[28] Before any remote request or response subactions can be routed to or from nodes on such a bus, it is necessary to assign a unique bus ID and to update the route maps of bridge portals throughout the net. Whenever QUARANTINE.*net_update* changes from one to zero, the alpha portal on a bus without an assigned bus ID shall request an assignment from the prime portal; after a bus ID is assigned by the prime portal, the alpha portal shall communicate the bus ID to other portals on the bus.

An alpha portal initiates a request for a bus ID by instructing its co-portal to transmit a BUS ID request message towards the prime portal. Unless the co-portal is on the same bus as the prime portal (or is the prime portal itself), the *source_ID* field shall be equal to the co-portal's global node ID and the *destination_ID* field shall be equal to the local node ID of the local alpha portal. Otherwise, the *destination_ID* field shall be equal to the local node ID of the prime portal. When the co-portal lacks an assigned bus ID for its own bus it shall defer sending the BUS ID message until it has a bus ID assignment. The value of *bus_ID* in the request shall be equal to the alpha portal's EUI-64 modulo 1023.

When an alpha portal that is not also a prime portal receives a BUS ID request message, it shall transfer it, unaltered, to its co-portal. The co-portal shall update the *destination_ID* field as described above and shall forward the message to the next alpha portal or prime portal, as appropriate.

A prime portal that receives a BUS ID request message shall transmit a BUS ID response message that contains the assigned bus ID; the *destination_ID* field shall be set to the value of the *source_ID* field from the BUS ID request. If the message's requested *bus_ID* is available, the prime portal shall assign it to the requester. Otherwise, the prime portal shall

---

[27] The manner in which a bridge portal maintains correlation between a node's physical ID and its virtual ID is implementation-dependent and is derived from the physical ID to EUI-64 and virtual ID to EUI-64 maps that a bridge portal is required to maintain. The correlation between physical ID and virtual ID is established by each node's EUI-64.

[28] NODE_IDS.*bus_ID* retains a value of $3FF_{16}$ even after the bus has been assigned a bus ID by the prime portal. Bridge portals connected to the bus store the assigned bus ID as CLAN_INFO.*bus_ID* and use it to determine when local node ID to global node ID translations (or vice versa) are necessary.

assign the smallest unallocated bus ID that is also greater than the prime portal's EUI-64 modulo 1023; if no such bus ID exists, the prime portal shall assign the smallest unallocated bus ID. If there are no unallocated bus IDs, the prime portal shall return a BUS ID response with a bus ID equal to $3FF_{16}$.

When the prime portal distributes an assigned bus ID, there is no confirmation of its receipt by the requesting alpha portal. Consequently, the bus ID shall be marked as assigned until the next net update—even though the corresponding entry in the prime portal's route map is CLEAN. Upon completion of net update, the prime portal shall update its list of assigned bus IDs to include only those that correspond to VALID entries in the net allocation map received in the most recent UPDATE ROUTES message.

The recipient of a BUS ID response uses the *requester_EUI_64* field in the message header to determine what action, if any, should be taken. If the *requester_EUI_64* does not match the recipient's own EUI-64, the message shall be discarded. Otherwise, the assigned bus ID in the message shall be communicated to the recipient's co-portal by implementation-dependent means.

If an alpha portal requests a bus ID assignment from the prime portal and net update occurs before a BUS ID response message is received, the alpha portal shall repeat its request for a bus ID. In the absence of net update, if the alpha portal fails to receive a bus ID assignment within a reasonable, implementation-dependent time (*e.g.*, *hops_to_prime* seconds), it should repeat its request for a bus ID. Because BUS ID request or response messages might be delayed, not discarded, the alpha portal could ultimately receive multiple bus ID assignments. When an alpha portal whose CLAN_INFO.*bus_ID* is equal to $3FF_{16}$ receives an assigned bus ID other than $3FF_{16}$, it shall create a BUS ID ANNOUNCEMENT message whose *local_bus* bit is one and whose *bus_ID* field is equal to the assigned bus ID. The message shall be processed by the alpha portal as described below.

With one exception, bridge portals process a BUS ID ANNOUNCEMENT message differently according to its source: the alpha portal that originally requested a bus ID assignment, another portal on the local bus or the co-portal. Regardless of the source of the message, if the route map entry that corresponds to the message's *bus_ID* field is FORWARD, the portal shall set it to DIRTY, set its *coportal_update_required* flag to TRUE, set its *brdg* variable to three, and initiate an arbitrated (short) bus reset. Otherwise, the actions taken by a bridge portal upon receipt of a BUS ID ANNOUNCEMENT message are as follows:

— When the alpha portal that originally requested a bus ID assignment processes the BUS ID ANNOUNCEMENT message it created, it shall inspect the entry in its route map that corresponds to the *bus_ID* field. If the entry is not CLEAN, the portal shall discard the BUS ID ANNOUNCEMENT message. Otherwise, the portal shall attempt to obtain control of the semaphore shared with its co-portal. If the semaphore is unavailable, the portal shall defer processing the BUS ID ANNOUNCEMENT message until control is obtained or net update commences. After obtaining control, the portal shall set its *brdg* variable to three, shall update CLAN_INFO.*bus_ID* with the value of the message's *bus_ID* field and shall set the corresponding entry in its route map to VALID. Next, the portal shall temporarily zero the message's *local_bus* bit, transfer the message to its co-portal, and wait for it to release the semaphore. After the semaphore is released, the portal shall restore the message's *local_bus* bit to its prior value of one and shall individually transmit the BUS ID ANNOUNCEMENT message to all bridge portals on the local bus. Once this is complete, the portal shall set its *brdg* variable to two, zero QUARANTINE.*orphan* and shall transmit a broadcast write request to zero QUARANTINE.*orphan* for all nodes on the bus.

— A portal that received the message from another portal on the local bus and whose *mute* flag is FALSE shall attempt to obtain control of the semaphore shared with its co-portal. If the semaphore is unavailable, the portal shall not process the BUS ID ANNOUNCEMENT message but shall transmit *resp_conflict_error* in response. After obtaining control, the portal shall inspect the entry in its route map that corresponds to the *bus_ID* field. If the entry is not CLEAN, the portal shall discard the BUS ID ANNOUNCEMENT message. Otherwise, the portal shall set it to VALID and, if the message's *local_bus* bit is one, shall update CLAN_INFO.*bus_ID* with the value of the message's *bus_ID* field. Next, the portal shall zero the message's *local_bus* bit, transfer the message to its co-portal and wait for it to release the semaphore before transmitting *resp_complete*.

— A portal that received the message from another portal on the local bus and whose *mute* flag is TRUE shall inspect the entry in its route map that corresponds to the *bus_ID* field. If the entry is not CLEAN, the portal shall discard the BUS ID ANNOUNCEMENT message. Otherwise, the portal shall set it to VALID and, if the message's *local_bus* bit is one, shall update CLAN_INFO.*bus_ID* with the value of the message's *bus_ID* field. The portal shall not transfer the message to its co-portal but shall transmit *resp_complete*.
— A portal that received the message from its co-portal shall set its *brdg* variable to three, set the entry in its route map that corresponds to the message's *bus_ID* field to FORWARD, and release the semaphore. Next, the portal shall individually transmit the BUS ID ANNOUNCEMENT message to all bridge portals on the local bus. Once this is complete, the portal shall set its *brdg* variable to two.

NOTE—If a bus reset interrupts the process of distributing the BUS ID ANNOUNCEMENT messages to other bridge portals, the portal's *brdg* value of three guarantees that net update occurs. So long as the assigned bus ID is present in at least one CLAN_INFO register, net update completes the announcement process. There is no need for the portal to resume distribution of BUS ID ANNOUNCEMENT messages.

The recipient of a BUS ID ANNOUNCEMENT message might be a coordinator (other than the alpha portal that requested bus ID assignment) busy with virtual ID management. The receipt of a BUS ID ANNOUNCEMENT message shall interrupt virtual ID management in order to process the message—except that the message's transmission to other portals on the local bus, if required, should be deferred until virtual ID management completes.

Annex A

(normative)

Net correctness properties

A correctly configured net of interconnected Serial Buses is a directed spanning tree whose vertices are Serial Buses and whose edges are bridges specified by this standard. The spanning tree connects all Serial Buses (*i.e.*, a route exists from any bus to any other bus) without loops (*i.e.*, only one route exists between a pair of buses). From the viewpoint of a hypothetical external observer, it is trivial to determine whether a net satisfies these criteria. However, it is a subtler problem for an individual bridge or portal to determine if the net is correctly configured. The following correctness properties are observable by individual bridges and portals whenever QUARANTINE.*net_update* is zero after analysis of the self-ID packets observed subsequent to the most recent bus reset.

- For each bus within the net:
  - the values of all portals' CLAN_EUI_64 registers are equal;
  - the values of all portals' CLAN_INFO.*bus_ID* fields are equal;
  - for all *bus_ID* values less than $3FF_{16}$ and not equal to CLAN_INFO.*bus_ID*, either a) all portals' ROUTE_MAP[*bus_ID*] entries are CLEAN or b) one portal's ROUTE_MAP[*bus_ID*] entry is FORWARD and all other portals' ROUTE_MAP[*bus_ID*] entries are VALID;
  - the values of all portals' CLAN_INFO.*hops_to_prime* fields are equal; and
  - there is exactly one portal whose CLAN_INFO.*alpha* bit is one.
- For each bridge within the net:
  - the values of both portals' CLAN_EUI_64 registers are equal;
  - the values of both portals' *mute* flags are equal;
  - if *mute* is FALSE, for all *bus_ID* values less than $3FF_{16}$, either a) both portals' ROUTE_MAP[*bus_ID*] entries are CLEAN or b) one portal's ROUTE_MAP[*bus_ID*] entry is FORWARD and its co-portal's ROUTE_MAP[*bus_ID*] entry is VALID;
  - if *mute* is TRUE, for all *bus_ID* values less than $3FF_{16}$, both portals' ROUTE_MAP[*bus_ID*] entries are equal and either CLEAN or VALID;
  - there is at most one portal whose CLAN_INFO.*alpha* bit is one and whose EUI-64 is not equal to the value of its CLAN_EUI_64 register; and
  - unless both portals are mute, there is one portal whose CLAN_INFO.*alpha* bit is one, whose EUI-64 is not equal to the value of its CLAN_EUI_64 register and whose CLAN_INFO.*hops_to_prime* is one greater than its co-portal's CLAN_INFO.*hops_to_prime*.
- For each portal within a bridge:
  - if the value of the portal's QUARANTINE.*orphan* field is zero then the portal's CLAN_INFO.*bus_ID* field is not equal to $3FF_{16}$;
  - if the value of the portal's CLAN_INFO.*bus_ID* field is not equal to $3FF_{16}$ then the portal's ROUTE_MAP entry that corresponds to CLAN_INFO.*bus_ID* is VALID;
  - if the value of the portal's *mute* flag is TRUE, the portal's CLAN_INFO.*alpha* bit is one only if the portal's EUI-64 is equal to the value of its CLAN_EUI_64 register; and
  - if the portal's EUI-64 is equal to the value of its CLAN_EUI_64 register, the portal's CLAN_INFO.*alpha* bit is one and the value of the portal's CLAN_INFO.*hops_to_prime* field is zero.

The net update algorithms are designed either to insure that these conditions are satisfied or, if an incorrect condition is detected and cannot be remedied by continuing net update, to initiate an extreme response (net panic) that shuts down all bridge functions and subsequently initializes all bridges and correctly configures the net.

Annex B

(normative)

Minimum Serial Bus capabilities for bridge portals

In addition to the minimum capabilities defined by IEEE 1394 for isochronous resource manager-capable nodes, this annex specifies other capabilities or restrictions mandated by this standard for bridge portals.

A bridge portal compliant with this standard (and no other device, except as provided by a future IEEE standard) is specifically exempt from requirements b) and c) of IEEE Std 1394a-2000 Annex C1. It is essential for bridge operations that a portal be able both to snoop (receive asynchronous subactions whose *destination_ID* is not equal to the portal's node ID) and spoof (transmit asynchronous subactions whose *source_ID* is not equal to the portal's node ID).

A bridge portal shall be capable of receiving and transmitting primary packets whose data payload is less than or equal to 512 bytes; the *max_rec* field in a bridge portal's bus information block shall be greater than or equal to eight. This capability applies to subactions addressed to or originated by the bridge portal as well as to subactions snooped or retransmitted by the bridge portal.

A bridge portal 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, bridge portals 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. Bridge portals may automatically set *dreq* to zero (request initiation enabled) upon a power reset or a command reset.

While initializing after a power reset, a bridge portal whose link layer is active 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 bridge portal CSRs, are not accessible.

Annex C

(normative)

Pseudocode data structures and constants

Throughout this standard, C-like pseudocode is employed to permit more accurate specification of behaviors than is otherwise afforded by simple narration. Many of the variables used are defined and described in the functions themselves, but others are globally accessible to any functional block within a node. The definitions of those data structures and constants are gathered in this annex for convenience of reference.

C.1 Configuration ROM and control and status registers

When reference is made to the formal name of a CSR as specified by IEEE 1394 or this standard, for example BANDWIDTH_AVAILABLE or ROUTE_MAP. it shall be understood to mean the base address of the CSR within the node's memory space. Thus BANDWIDTH_AVAILABLE represents a value of FFFF F000 0220_{16} while ROUTE_MAP means FFFF F000 $1E00_{16}$. When it is necessary to refer to the current value of a CSR at the node itself, the formal name of the CSR is transformed into a similar name that is applied to a pseudocode variable or data structure, as set out in Table C-1. Only the CSRs referenced in the pseudocode are included.

Table C-1 — Configuration ROM and CSR data structures and constants (Sheet 1 of 3)

```
/* Configuration ROM */

struct {                          /* Bus information block */
   BYTE busName[4];               /* ASCII "1394" for IEEE 1394 */
   struct {
      BOOLEAN irmc:1;
      BOOLEAN cmc:1;
      BOOLEAN isc:1;
      BOOLEAN bmc:1;
      BOOLEAN pmc:1;
      BOOLEAN adjustable:1;
      BYTE reserved:2;
   } capabilities;
   BYTE cycleClkAcc;
   BYTE maxRec:4;
   BYTE reserved:2;
   BYTE maxROM:2;
   BYTE generation:4;
   BOOLEAN bridgeAware:1;
   BYTE linkSpd:2;
   OCTLET eui64;
} busInfoBlock;

struct {                          /* Bridge_Capabilities entry */
   BYTE key;
   BYTE reserved:2;
   BYTE maxIsoch:4;
   BYTE streams:6;
   DOUBLET latency:12;
} bridgeCapabilities;

/* CSR addresses and definitions */

#define STATE_SET 0xFFFFF0000004
#define NODE_IDS 0xFFFFF0000008
#define MESSAGE_REQUEST 0xFFFFF0000080
#define MESSAGE_RESPONSE 0xFFFFF00000C0
#define CYCLE_TIME 0xFFFFF0000200
```

Table C-1 — Configuration ROM and CSR data structures and constants (Sheet 2 of 3)

```
#define BUS_TIME 0xFFFFF0000204
#define QUARANTINE 0xFFFFF0000214
#define OMPR 0xFFFFF0000900
#define OPCR 0xFFFFF0000904
#define IMPR 0xFFFFF0000980
#define IPCR 0xFFFFF0000984
#define VIRTUAL_ID_MAP 0xFFFFF0001C00
#define ROUTE_MAP 0xFFFFF0001E00
#define CLAN_EUI_64 0xFFFFF0001F00
#define CLAN_INFO 0xFFFFF0001F08

struct {                          /* STATE_SET register */
   BYTE cmstr:1;
} stateSet;

struct {                          /* NODE_IDS register */
   QUADLET busID:10;
   QUADLET localID:6;
} nodeIDs;

struct {                          /* SPLIT_TIMEOUT register */
   QUADLET reserved:29;
   QUADLET seconds:3;
   QUADLET cycles:13;
   QUADLET reserved2:19;
} splitTimeout;

struct {                          /* CYCLE_TIME register */
   QUADLET secondsCount:7;        /* Least significant portion of BUS_TIME.secondsCount */
   QUADLET cycleCount:13;         /* Cycles (1/8000 second) count */
   QUADLET cycleOffset:12;        /* Ticks of 24.576 MHz clock (nominal 40 ns) */
} cycleTime;

UNSIGNED cycleOffsetThreshold;    /* Initialized to 3071 for nominal 125 us interval */

union {                           /* BUS_TIME register */
   QUADLET secondsCount;
   struct {
      QUADLET secondsCountHi:25;
      QUADLET secondsCountLo:7;
   };
} busTime;

typedef struct {                  /* Output master plug register (oMPR) */
   BYTE spd:2;
   BYTE broadcastBase:6;
   BYTE nonpersistentExt;
   BYTE persistentExt;
   BYTE reserved:1;
   BYTE xspd:2;
   BYTE outputPlugs:5;
} OMPR_CSR;

OMPR_CSR oMPR;

typedef struct {                  /* Output plug control register (oPCR) */
   BYTE online:1;
   BYTE broadcastConnection:1;
   BYTE pointToPointConnections:6;
   BYTE xspd:2;
   BYTE channel:6;
   BYTE spd:2;
   BYTE overhead:4;
   DOUBLET payload:10;
} OPCR_CSR;
```

Table C-1 — Configuration ROM and CSR data structures and constants (Sheet 3 of 3)

```
OPCR_CSR oPCR;

typedef struct {                /* Input master plug register (iMPR) */
   BYTE spd:2;
   BYTE reserved1:6;
   BYTE nonpersistentExt;
   BYTE persistentExt;
   BYTE reserved2:1;
   BYTE xspd:2;
   BYTE inputPlugs:5;
} IMPR_CSR;

IMPR_CSR iMPR;

typedef struct {                /* Input plug control register (iPCR) */
   BYTE online:1;
   BYTE broadcastConnection:1;
   BYTE pointToPointConnections:6;
   BYTE xspd:2;
   BYTE channel:6;
   BYTE spd:2;
   BYTE overhead:4;
   QUADLET payload:10;
} IPCR_CSR;

IPCR_CSR iPCR;

struct {                        /* QUARANTINE register */
   QUADLET orphan:1;
   QUADLET netUpdate:1;
   QUADLET reserved:30;
} quarantine;

OCTLET virtualIDMap[64];        /* VIRTUAL_ID_MAP register */



typedef enum {CLEAN=0,          /* ROUTE_MAP entry states */
              DIRTY=1,
              VALID=2,
              FORWARD=3} ROUTE_STATE;

ROUTE_STATE routeMap[1024];     /* ROUTE_MAP register */

#define LOCAL FORWARD           /* No FORWARD state in clan allocation maps; reuse
                                   value for definition of LOCAL */

OCTLET clanEUI64;               /* CLAN_EUI_64 register */

typedef struct {                /* CLAN_INFO register */
   BOOLEAN alpha:1;
   BOOLEAN preferredClan:1;
   DOUBLET busID:10;
   DOUBLET hopsToPrime:10;
   DOUBLET reserved:10;
} CLAN_INFO_CSR;

CLAN_INFO_CSR clanInfo;
```

C.2 Message and packet formats

The data structures for packets, packet headers, and net management messages are contained in Table C-2.

Table C-2 — Packet and message data structures and constants (Sheet 1 of 2)

```
typedef struct {        /* IEEE 1394 primary packet header */
   NODE_ID destination;
   BYTE tl:6;
   BYTE rt:2;
   BYTE tcode:4;
   BYTE pri:4;
   NODE_ID source;
   union {
      struct {
         BYTE rcode:4;
         BYTE reserved1:6;
         BYTE extRcode:6;
         NODE_ID proxy;
         DOUBLET reserved2;
      };
      OCTLET destinationOffset:24;
   };
   DOUBLET dataLength;
   BYTE snarf:2;
   BYTE sourcePortal:6;
   BYTE extTcode;
   QUADLET data[];
} PACKET;

typedef struct {
   BYTE hdrReserved1;
   enum {TIMEOUT=1,
         TIME_OFFSET=2,
         JOIN=16,
         LEAVE=17,
         LISTEN=18,
         RENEW=19,
         TEARDOWN=20,
         STREAM_STATUS=21,
         MUTE=0x80,
         BUS_ID=0x81,
         BUS_ID_ANNOUNCEMENT=0x82,
         PANIC=0x83} opcode;
   BYTE hdrReserved2;
   enum {OK=0,
         ERROR=1,
         CONNECTION_DELETED=2,
         INVALID_PLUG,
         PAYLOAD_TOO_BIG,
         RESOURCES_UNAVAILABLE,
         STREAM_CONFLICT,
         INVALID_TALKER,
         UNEXPECTED_ERROR,
         INVALID_STREAM,
         PENDING=0xFF} result;
   OCTLET requesterEUI64;
   OCTLET responderEUI64;
} BRIDGE_MSG_HDR;

typedef struct {
   BRIDGE_MSG_HDR;
   QUADLET remoteTimeout;     /* In units of 125 us */
   QUADLET maxRemote:4;       /* 2 ** (maxRemote + 1) bytes */
   QUADLET reserved:18;
   QUADLET hopCount:10;       /* Count of intervening bridges */
```

Table C-2 — Packet and message data structures and constants (Sheet 2 of 2)

```
} TIMEOUT_MSG;

typedef struct {
   BRIDGE_MSG_HDR;
   OCTLET talkerEUI64;        /* Uniquely identifies talker */
   DOUBLET maxPayload;        /* Largest data payload (in bytes) */
   DOUBLET latency;           /* Constant delivery delay to endpoint */
   DOUBLET window;            /* Time period over which aggregate applies */
   DOUBLET aggregatePayload;  /* Maximum payload (in bytes) in a single window */
   DOUBLET sourceQuantum;     /* Underlying "chunkiness" of source data stream */
   DOUBLET reserved1;
   QUADLET sourceBitRate;     /* Source data stream's rate (in bits per second) */
   DOUBLET controllerID;      /* Virtual node ID of controller */
   BYTE talkerIndex;          /* Identifies oPCR or other designator at talker */
   BYTE listenerIndex;        /* Identifies iPCR or other designator at listener */
   DOUBLET talkerID;          /* Virtual node ID of talker */
   DOUBLET listenerID;        /* Virtual node ID of listener */
   BYTE tspd:3;               /* Talker speed */
   BYTE lspd:3;               /* Listener speed */
   BYTE channel:6;            /* Channel (on the local bus */
   BYTE reserved2:2;
   BOOLEAN upstream:1;        /* Propagation direction of TEARDOWN message */
   QUADLET lifetime:17;       /* Remaining stream lifetime (seconds) */
} STREAM_MSG;

typedef struct {
   BRIDGE_MSG_HDR;
   QUADLET reserved:22;
   QUADLET busID:10;          /* Suggested or assigned bus ID */
} BUS_ID_MSG;

typedef struct {
   BRIDGE_MSG_HDR;
   OCTLET deltaCycles;        /* Cumulative time difference between two buses */
} TIME_OFFSET_MSG;

typedef struct {
   ROUTE_STATE netAllocationMap[1024];
   OCTLET primePortalEUI64;
   QUADLET clanInfo;
} UPDATE_ROUTES_MSG;
```

C.3 Global portal variables and external procedures

Common constant and type definitions, global bridge portal variables, and external procedures are contained in Table C-3.

Table C-3 — Constants, global portal variables and external procedures (Sheet 1 of 4)

```
/* Common constants and type definitions */

#define LOCAL_BUS_ID 0x3FF
#define NO_BUS_ID 0x3FF
#define MAX_BUS_ID 0x3FF
#define UNASSIGNED_CHANNEL 0xFF
#define BROADCAST 0x3F
#define DEFAULT_BROADCAST_CHANNEL 31

#define QUADLET_WRITE_REQUEST 0
#define BLOCK_WRITE_REQUEST 1
#define WRITE_RESPONSE 2
#define QUADLET_READ_REQUEST 4
#define BLOCK_READ_REQUEST 5
```

Table C-3 — Constants, global portal variables and external procedures (Sheet 2 of 4)

```
#define QUADLET_READ_RESPONSE 6
#define BLOCK_READ_RESPONSE 7
#define CYCLE_START 8
#define LOCK_REQUEST 9
#define ISOCHRONOUS 0x0A
#define LOCK_RESPONSE 0x0B

#define ACK_COMPLETE 1
#define ACK_PENDING 2
#define ACK_BUSY_X 4
#define ACK_BUSY_A 5
#define ACK_BUSY_B 6
#define ACK_ADDRESS_ERROR 0xF

#define RETRY_1 0
#define RETRY_X 1
#define RETRY_A 2
#define RETRY_B 3

#define RESP_COMPLETE 0
#define RESP_CONFLICT_ERROR 4
#define RESP_DATA_ERROR 5
#define RESP_TYPE_ERROR 6
#define RESP_ADDR_ERROR 7

#define REMOTE_WRITE_ACTIVE 0
#define EXT_LEGACY_QUARANTINE 16
#define EXT_INVALID_ROUTE 17
#define EXT_INVALID_GLOBAL_ID 18
#define EXT_PAYLOAD_TOO_BIG 20
#define EXT_CONGESTION 21
#define EXT_DATA_ERROR 22
#define EXT_NO_VIRTUAL_ID 23
#define EXT_UNSPECIFIED 0x3F

typedef enum {SIMPLE=0,
              BRIDGE_UNCHANGED_STATE=2,
              BRIDGE_CHANGED_STATE=3} NODE_STATE;

#define BRIDGE 2                /* The node is an IEEE 1394.1 bridge */
#define CHANGED_STATE 1         /* The net state has changed since the last “all clear” */

typedef enum {NO_SNARF=0,       /* Values for the snarf field */
              SNARF_TERMINAL_EXIT=1,
              SNARF_INITIAL_ENTRY=2,
              SNARF_ALL=3} SNARF;

typedef union {
   DOUBLET nodeID;
   struct {
      DOUBLET busID:10;
      BYTE localID:6;
   };
} NODE_ID;

typedef struct {
   OCTLET talkerEUI64;          /* Uniquely identify stream (together with talkerIndex) */
   BYTE talkerIndex;
   DOUBLET talkerID;            /* Needed if it has an oPCR */
   BYTE listenerIndex[64];      /* All possible local bus listeners */
   DOUBLET controllerID[64];    /* And their associated controllers */
   BYTE channel;                /* Initial value is UNASSIGNED */
   BYTE speed;
   DOUBLET maxPayload;          /* Data payload, in bytes */
   QUADLET bandwidth;           /* Bandwidth allocation units */
```

Table C-3 — Constants, global portal variables and external procedures (Sheet 3 of 4)

```
   enum {UNSPECIFIED=0,          /* Disjoint portal functions */
         LISTENER=1,             /* On talker's bus, also a reallocation proxy */
         TALKER=2,               /* Always a reallocation proxy */
         REALLOCATION_ONLY=3} portalRole;
   BYTE coportalChannel;         /* Channel used by our co-portal for this stream */
   QUADLET lifetime[64];         /* Remaining lifetime, in seconds, for local listeners */
   OCTLET localListeners;        /* Bit map, by virtual ID, of local listeners */
} STREAM_INFO;

/* Global bridge portal variables */

NODE_STATE brdg[63];           /* Each node's brdg variable (indexed by physical ID) obtained from */
                               /* self-ID packet 0 after each bus reset) */
BOOLEAN bridgeAware[63];       /* Collected from bus information block */
BOOLEAN coportalPrime;         /* TRUE when CLAN_EUI_64 equals co-portal's EUI-64 */
BOOLEAN coportalUpdateRequired;  /* Internal flag for UPDATE ROUTES processing */
BOOLEAN dualPhase;
BYTE maxInterportalSpeed[1023];  /* Maximum transmit speed on intermediate bus */
                               /* (all entries reset to S100 by net update) */
DOUBLET maxRequestForwardTime;   /* Maximum time (units of 125 us) to attempt to transmit a request */
DOUBLET maxResponseForwardTime;  /* Maximum time (units of 125 us) to attempt to transmit a response */
BOOLEAN mute;                  /* TRUE if portal is mute */
OCTLET ownEUI64;               /* Unique ID from our own bus information block */
NODE_ID ownGlobalID;
BYTE ownLocalID;               /* Our own PHY ID from NODE_IDS.localID */
BYTE ownVirtualID;             /* Our own virtual ID */
BYTE ownSpeed;                 /* Slower of our own link or PHY speed */
BYTE physicalToVirtual[63];    /* Maps 6-bit local ID to corresponding virtual ID */
UNSIGNED remoteTimeout[63];
STREAM_INFO streamInfo[64];    /* Actual number of stream descriptors implementation-dependent */
DOUBLET unmutedHopsToPrime;    /* CLAN_INFO.hops_to_prime value when MUTE message received */
BOOLEAN transmitVirtualIDMap;  /* TRUE if coordinator needs to distribute the virtual ID map */
BYTE virtualToPhysical[63];    /* Maps 6-bit virtual ID to corresponding local ID */


/* External procedures */

BOOLEAN allocateBandwidth(QUADLET bandwidth);
BYTE allocateChannel(BYTE channel);
STREAM_INFO *allocateResources(DOUBLET maxPayload, BYTE speed);
STREAM_INFO *allocateStreamDescriptor();
VOID deallocateBandwidth(QUADLET bandwidth);
VOID deallocateChannel(BYTE channel);
VOID deallocateResources(VOID *streamInfo);
VOID deallocateStreamDescriptor(VOID *streamInfo);
VOID deleteLocalListener(BYTE listenerVirtualID, STREAM_INFO *streamInfo);
VOID forwardMsg(DOUBLET destinationID, OCTLET csr, BYTE snarf, VOID *message);
BOOLEAN getSemaphore();        /* Returns TRUE if semaphore acquired */
STREAM_INFO *getStreamDescriptor(OCTLET talkerEUI64, BYTE talkerIndex);
VOID listenRequest(VOID *streamMsg, VOID *streamInfo);
VOID muteBridge(BYTE localID);
VOID netPanic();
BYTE pathSpeed(BYTE localID1, BYTE localID2);
INT pop();
VOID processNetManagementMessage();
BOOLEAN programInputPlug(BYTE localID, BYTE plugIndex, VOID *data);
BOOLEAN programOutputPlug(BYTE localID, BYTE plugIndex, VOID *data);
VOID push(INT phyID);
VOID queuePacket(BOOLEAN request, BOOLEAN transferToCoportal);
VOID readQuadlet(BYTE phyID, OCTLET csr, VOID *responseData);
VOID readBlock(BYTE phyID, OCTLET csr, DOUBLET size, VOID *responseData);
VOID releaseSemaphore();
VOID resetBus();
```

Table C-3 — Constants, global portal variables and external procedures (Sheet 4 of 4)

```
VOID sendRouteInfo(OCTLET newClanEUI64, VOID *netMap,
                   BOOLEAN preferredClan, INT hopsToPrime);
VOID streamStatusResponse(VOID *streamMsg, BYTE result);
VOID synthesizeResponse(VOID *request, BYTE rcode, BYTE extRcode);
VOID teardownRequest(BOOLEAN upstream, STREAM_INFO *streamInfo);
VOID transmitMsg(DOUBLET destinationID, OCTLET csr, BYTE snarf, VOID *message);
BYTE transmitPacket(VOID *packet);
VOID waitSemaphoreFree();
VOID writeBlock(BYTE phyID, OCTLET csr, DOUBLET size, VOID *requestData);
VOID writeQuadlet(BYTE phyID, OCTLET csr, VOID *requestData);
```

Annex D

(normative)

Transaction routing

For ease of comprehension, the transaction routing actions taken by bridge portals are presented in Clause 7. from the point of view of the source bus, the intermediate buses, and the destination bus. These behaviors can be merged into a single set of decision tables and C functions that describe the normal operations of any bridge portal with respect to bridge-bound and bus-bound request and response subactions routed by *destination_ID*.

This annex specifies the normative routing behavior of bridge portals. It is not intended to imply a particular implementation.

D.1 Bridge-bound subactions

The behavior of a bridge portal when it snoops packets on its local bus can be considered in two parts—time-critical operations likely to be implemented in hardware and other operations less sensitive to processing time. The latter, which might require the creation of a response subaction as well as the retransmission of the original snooped subaction, may be implemented in firmware or software.

The time-critical operations use the following parameters:

— The *destination_ID* from the snooped packet
— The subaction type, request or response
— The portal's local ID[29] and global node ID
— Routing information maintained by the portal for all possible bus IDs
— Whether the portal is the alpha portal for the bus

The result of the snooping operations is an action code that provides guidance for subsequent packet retransmission and response synthesis and, in some circumstances, the transmission of an acknowledge packet. Packets bound for the portal are signaled to the local transaction layer by a LK_DATA.*indication* as described by IEEE Std 1394-1995 and are not discussed in further detail in this annex. Table D-1 summarizes the time-critical behaviors of bridge portals when they snoop the local bus but omits the case when a busy acknowledgment is returned. If the acknowledgment transmitted is either *ack_busy_X*, *ack_busy_A,* or *ack_busy_B*, the output action code shall be IGNORE.

Table D-1 — Time-critical bridge-bound snooping

Destination						
bus_ID	***local_ID***	**`route`[*bus_ID*]**	**`alpha`**	***ack***	**Output**	**Comment**
$3FF_{16}$	$3F_{16}$	—	—	No	PORTAL	Broadcast packet for portal
	Equal to NODE_IDS.*local_ID*			Yes[a]		Packet addressed to portal's local address
	Not equal to NODE_IDS.*local_ID*			No	IGNORE	Packet addressed to another node's local address
Equal to portal's bus ID	—		No	No	IGNORE	Alpha portal responsible for these packets
	Not equal to portal's virtual ID		Yes	Yes	ECHO	Packet addressed to another node's global address

[29] The term "local ID" is defined by IEEE Std 1394a-2000; it refers to the 6-bit physical ID assigned to each node on a local bus as a consequence of bus reset.

Table D-1 — Time-critical bridge-bound snooping *(continued)*

Destination						
bus_ID	***local_ID***	**`route`[*bus_ID*]**	**`alpha`**	***ack***	**Output**	**Comment**
Equal to portal's global node ID			—	Yes[a]	PORTAL	Packet addressed to portal's global address
Neither $3FF_{16}$ nor equal to portal's bus ID	—	CLEAN or DIRTY	No	No	IGNORE	No valid routing exists for the packet, but only the alpha portal responds
			Yes	Yes[b]	INVALID	
		FORWARD	—		COPORTAL	The destination bus ID is valid and routed by this or another portal
		VALID		No	IGNORE	

[a] When a packet is addressed to the bridge portal, transmission of an appropriate acknowledgment is expected as a consequence of normal operations after an LK_DATA.*indication* is signaled to the transaction layer.

[b] The type of subaction observed, request or response, determines whether the entry portal transmits *ack_pending* or *ack_complete*, respectively.

Whenever a portal shall transmit an acknowledgement, a busy acknowledgement may be substituted if bridge resources are temporarily unavailable, in which case the output action code shall be IGNORE.

Once the time-critical operations are complete (and an acknowledge packet transmitted, as necessary, within the time permitted by IEEE 1394), additional scrutiny of the snooped packet is necessary before additional action is taken. When the action code output by the time-critical operations is other than IGNORE, the less time-critical operations are invoked; they use the following parameters:

— The action code produced by the snoop operations
— The *source_ID* and *tcode* from the snooped packet
— A mapping from 6-bit physical ID on the local bus to the corresponding virtual ID
— Whether the originator of the snooped packet is bridge-aware

The bridge's disposition of the subaction (discard or queue for retransmission by the same portal or its co-portal) and the synthesis of a response packet are specified by Table D-2.

Table D-2 — Bridge-bound response synthesis and subaction processing

	Source			Packet header		
Snoop output	***bus_ID***	**`virtual_ID`[*local_ID*]**	**Bridge aware**	***tcode***	***snarf***	**Action**
PORTAL	—					Signal the subaction to the local transaction layer as specified by IEEE Std 1394.[a]
INVALID						For a request subaction, synthesize a response that indicates *resp_address_error* with an *ext_rcode* of *ext_invalid_route*. For both request and response, discard the subaction.
ECHO	Not $3FF_{16}$	—			—	Discard the subaction. Caused by erroneous behavior of another bridge portal on the bus.
COPORTAL					0, 1, or 2	Queue the subaction for retransmission by the co-portal.
					3	Synthesize a write response that indicates *resp_complete* ("posted write" completion), process the net management message then queue the subaction for retransmission by this portal (ECHO) or the co-portal (COPORTAL).

Table D-2 — Bridge-bound response synthesis and subaction processing *(continued)*

Snoop output	Source		Bridge aware	Packet header		Action
	bus_ID	`virtual_ID`[*local_ID*]		*tcode*	*snarf*	
ECHO COPORTAL	$3FF_{16}$	Unmapped	—		—	For a request subaction, synthesize a response that indicates *resp_conflict_error* with an *ext_rcode* of *ext_no_virtual_ID*. For both request and response, discard the subaction.
		Mapped	No	Response	—	Replace the subaction's *source_ID* with the corresponding global node ID and queue for retransmission by this portal (ECHO) or the co-portal (COPORTAL).
			Yes	—	0 or 1	
					2 or 3	Synthesize a write response that indicates *resp_complete* ("posted write" completion), process the net management message then Replace the subaction's *source_ID* with the corresponding global node ID and queue for retransmission by this portal (ECHO) or the co-portal (COPORTAL).
ECHO COPORTAL	$3FF_{16}$	Mapped	No	Write request	—	Synthesize a write response that indicates *resp_complete* ("posted write" completion) and then replace the subaction's *source_ID* with the corresponding global node ID and queue for retransmission by this portal (ECHO) or the co-portal (COPORTAL).
				Read or lock request		Not permitted for legacy devices; discard the subaction then synthesize a response packet that indicates *resp_type_error* with an *ext_rcode* of *ext_legacy_quarantine*.

[a] In all cases except one, the *destination_ID* contains a local node ID. When the alpha portal is addressed globally by another node on the local bus, the *destination_ID* contains a global node ID.

D.2 Bus-bound subactions

A subaction queued for transmission by a bridge portal has already passed the acceptance test specified in D.1, either from its co-portal (a packet forwarded to a remote bus) or from the retransmitting portal itself (the echo of a virtually addressed packet back to its original bus). If the subaction's ultimate destination is not the bus to which the portal is connected, it is a packet in transit and is to be retransmitted at the fastest speed possible. Otherwise, when a packet has arrived at its destination bus it is necessary to convert the global *destination_ID* to an address on the local bus before retransmission. For both transit packets and those that have reached their destination bus, the bridge portal monitors the acknowledge packet and in some cases synthesizes a response packet to be returned to the subaction originator. Table D-3 describes the bridge portal's action according to the *destination_ID* and *tcode*.

Table D-3 — Bus-bound subaction processing

Destination				Packet header		
bus_ID	*local_ID*	phy_ID[*local_ID*]	Bridge aware	*tcode*	*snarf*	Action
Not equal to portal's bus ID	—				0, 1, or 2	Packet in transit to another bus. If the packet payload (or the payload anticipated in a response) can be transmitted at the intra-portal speed for the bus, retransmit the subaction without modification. Otherwise, synthesize response packet with *resp_type_error* and *ext_rcode* that indicates the payload is too large.
					3	Synthesize a write response that indicates *resp_complete* ("posted write" completion), process the net management message then retransmit the subaction.
Equal to portal's bus ID	$3F_{16}$	—			—	Broadcast is not supported for global node IDs. Synthesize response packet with *resp_address_error* and *ext_rcode* that indicates no global node ID. Discard the subaction.
	Not $3F_{16}$	$3F_{16}$	—	Request		Unknown global node ID. Synthesize response packet with *resp_address_error* and *ext_rcode* that indicates no global node ID. Discard the subaction.
				Response		Unknown global node ID; discard the subaction.
		Equal to portal's physical ID	—			Signal the subaction to the local transaction layer as specified by IEEE Std 1394. For request subactions, the local transaction layer is expected to signal an LK_DATA.*response* in return; if it indicates anything other than *ack_pending*, synthesize an appropriate response packet.
		Neither $3F_{16}$ nor equal to portal's physical ID	Yes	—	0 or 2	Replace packet's *destination_ID* with local node ID and transmit subaction on local bus.If an acknowledgment other than *ack_pending* is received, synthesize an appropriate response packet.
			No	Request		
					1 or 3	Synthesize a write response that indicates *resp_complete* ("posted write" completion), process the net management message then discard the subaction without transmission to *destination_ID*.
				Response		Discard the subaction.

When a bridge portal transmits a bus-bound request subaction, the acknowledge code returned by the recipient may cause the bridge portal to synthesize a response packet to be returned to the original requester. The same rules apply to both packets in transit and packets that have arrived at their destination bus, as summarized by Table D-4.

Table D-4 — Synthesized response determined by acknowledge code

Acknowledge code	Synthesized response code	Comment
ack_complete	*resp_complete*	Valid only for write requests; always return a write response (even to read or lock requests).
ack_pending	—	No response packet is created; the recipient of the request is expected to transmit the response.
ack_busy_X *ack_busy_A* *ack_busy_B*	*resp_conflict_error*	The response packet is created only after the bridge portal has exhausted the retry limits or relevant time-out period specified by this standard.
ack_tardy		A subsequent retry by the requester might succeed.
ack_conflict_error		

Table D-4 — Synthesized response determined by acknowledge code *(continued)*

Acknowledge code	Synthesized response code	Comment
ack_data_error	*resp_data_error*	Errors that complete the transaction and require the return of a response packet to the original requester.
ack_type_error	*resp_type_error*	
ack_address_error	*resp_address_error*	

When an intermediate bridge portal synthesizes a response, it shall set the *source_ID* field in the response to the value of *destination_ID* obtained from the corresponding request and set the *proxy_ID* field to its own global node ID. Otherwise, when a terminal exit portal synthesizes a response, it shall set the *source_ID* field to the global node ID of the destination node and zero the *proxy_ID* field. In both cases, the *ext_rcode* field shall be zero.

D.3 Transaction routing functions

C pseudocode functions and procedures for the bridge portal behaviors, specified by D.1 and D.2, are combined into Table D-5. The C pseudocode is not intended to restrict implementations to the particular approach illustrated; only the behaviors are normative.

Table D-5 — Transaction routing functions (Sheet 1 of 4)

```
#include “csr.h”
#include “global.h”
#include “packets.h”

BOOLEAN request(unsigned tcode) {

   switch (tcode) {
      case QUADLET_WRITE_REQUEST:
      case BLOCK_WRITE_REQUEST:
      case QUADLET_READ_REQUEST:
      case BLOCK_READ_REQUEST:
      case LOCK_REQUEST:
         return(TRUE);

      default:
         return(FALSE);
   }
}

/* The snoop() function operates on received packets indicated by the PHY
   to the link. Because of the timing constraints on the transmission of an
   acknowledge packet, it is likely that this functionality is implemented
   in hardware---but other methods are possible. */

enum {IGNORE, INVALID, PORTAL, COPORTAL, ECHO} snoop(NODE_ID destination, BYTE tcode) {

   if (destination.nodeID == 0xFFFF)      /* Local bus broadcast? */
      return(PORTAL);
   else if (destination.busID == 0x3FF)   /* Local bus packet? */
      return((destination.localID == nodeIDs.localID) ? PORTAL : IGNORE);
   else if (destination.busID == clanInfo.busID)   /* Local bus (but virtual address)? */
      if (destination.nodeID == ownGlobalID.nodeID)
         return(PORTAL);                  /* Normal 1394 processing */
      else if (clanInfo.alpha) {          /* Should we do anything? */
         PH_ACK.request = (request(tcode) ? ACK_PENDING : ACK_COMPLETE);
         return(ECHO);                    /* Virtualize, retransmit */
      } else
         return(IGNORE);                  /* Leave it to the alpha ... */
   else switch (routeMap[destination.busID]) {
      case DIRTY:
      case CLEAN:
```

Table D-5 — Transaction routing functions (Sheet 2 of 4)

```
         if (clanInfo.alpha) {            /* Are we required to take action? */
            PH_ACK.request = (request(tcode) ? ACK_ADDRESS_ERROR : ACK_COMPLETE);
            return(INVALID);              /* Yes, synthesize error response packet */
         } else
            return(IGNORE);               /* No, leave it to the alpha portal */

      case FORWARD:
         PH_ACK.request = (request(tcode) ? ACK_PENDING : ACK_COMPLETE);
         return(COPORTAL);

      case VALID:
         return(IGNORE);                  /* Routed by another portal */
   }
}

/* The bridgeBound() function is driven by the results of snooping on individual packets.
   Since most of its actions take the form of response packets and/or packets requeued by one of
   the bridge's portals for retransmission, it is less time sensitive than the snooping
   functions and is likely to be implemented in firmware or software. */

VOID bridgeBound(PACKET *packet) {

BOOLEAN coPortal = TRUE;               /* Default assumption (override if loopback to same bus) */

   switch (snoop(packet->destination, packet->tcode)) {
      case ECHO:
         if (packet->source.busID != 0x3FF)  /* Is the source address local? */
           break;                         /* No, ignore the packet (design error in another bridge!) */
         coPortal = FALSE;                /* Remaining processing same as FORWARD (drop through) */

      case FORWARD:
         if (packet->source.busID == 0x3FF) {/* Packet is originating from this bus */
         if (physicalToVirtual[packet->source.localID] == 0x3F) {   /* Has a virtual ID been mapped? */
               if (request(packet->tcode))   /* Request subaction? */
                  synthesizeResponse(packet, RESP_CONFLICT_ERROR, EXT_NO_VIRTUAL_ID);
               break;                  /* Discard subactions whose source has no virtual node ID */
            }
            if (bridgeAware[packet->source.localID] || !request(packet->tcode))
               if (packet->snarf == NO_SNARF || packet->snarf == SNARF_TERMINAL_EXIT)
                  ;     /* All un-snarfed subactions from bridge-aware nodes are OK, */
                        /* as are legacy device response subactions */
               else {   /* Subaction OK, but post completion and process before forwarding */
                  synthesizeResponse(packet, RESP_COMPLETE, 0);
                  processNetManagementMessage();
               }
            else if (packet->tcode == QUADLET_WRITE_REQUEST || packet->tcode == BLOCK_WRITE_REQUEST)
               if (remoteTimeout[packet->source.localID] == 0)    /* Another write outstanding? */
                synthesizeResponse(packet, RESP_COMPLETE, 0);   /* No, post "done" to legacy device */
               else {
                  synthesizeResponse(packet, RESP_CONFLICT_ERROR, REMOTE_WRITE_ACTIVE);
                  break;               /* Do NOT requeue the packet */
               }
            else {                      /* Legacy devices limited to write requests */
               synthesizeResponse(packet, RESP_TYPE_ERROR, EXT_LEGACY_QUARANTINE);
               break;
            }
            packet->source.busID = clanInfo.busID;          /* Substitute global bus ID ... */
          packet->source.localID = physicalToVirtual[packet->source.localID];  /* ...and virtual ID */
       } else if (packet->snarf == SNARF_ALL) {         /* Process snarfed packet before forwarding */
            synthesizeResponse(packet, RESP_COMPLETE, 0);
            processNetManagementMessage();
         }
         queuePacket(request(packet->tcode), coPortal);     /* Requeue packets for retransmission */
         break;
```

Table D-5 — Transaction routing functions (Sheet 3 of 4)

```
      case INVALID:
         if (request(packet->tcode))
            synthesizeResponse(packet, RESP_ADDR_ERROR, EXT_INVALID_ROUTE);
         break;

      case PORTAL:
         LK_DATA.indication = TRUE;        /* Portal local, usual processing per IEEE 1394 */
         break;
   }
}

/* The busBound() procedure processes packets forwarded across intermediate buses as well as
   packets that have arrived at their destination bus. Although the code below represents a
   firmware implementation, parts of the algorithm are readily adaptable to hardware. */

VOID busBound(PACKET *packet) {

   if (packet->destination.busID != clanInfo.busID)
     if (packet->snarf != SNARF_ALL)               /* Packet snarfed by non-terminal exit portal? */
         queuePacket(request(packet->tcode), FALSE);  /* No, just forward to next entry portal */
      else {
         synthesizeResponse(packet, RESP_COMPLETE, 0);
         processNetManagementMessage();
         queuePacket(request(packet->tcode), FALSE);
      }
   else if (packet->destination.localID == 0x3F)
      ;                                            /* Broadcast not supported for global node IDs */
   else if (packet->destination.localID == ownGlobalID.localID)
     LK_DATA.indication;                          /* Portal local, usual processing per IEEE 1394 */
   else if (virtualToPhysical[packet->destination.localID] == 0x3F)
    if (request(packet->tcode))                 /* Return error response for request subactions ... */
         synthesizeResponse(packet, RESP_ADDR_ERROR, EXT_INVALID_GLOBAL_ID);
      else
         ;                                          /* ... but just discard response subactions */
   else if (packet->snarf == SNARF_TERMINAL_EXIT || packet->snarf == SNARF_ALL) {
      synthesizeResponse(packet, RESP_COMPLETE, 0);
      processNetManagementMessage();
   } else if (bridgeAware[packet->destination.localID] || request(packet->tcode)) {
      packet->destination.busID = clanInfo.busID;
      packet->destination.localID = virtualToPhysical[packet->destination.localID];
      queuePacket(request(packet->tcode), FALSE);     /* Transmit packet to its final destination */
   }

}

/* This function creates a response packet in the extended format defined by P1394.1; the new
   information includes more detail about the nature of the error as well as the virtual node ID
   of the bridge portal that synthesized the response (the latter may be useful to diagnostic or
   management software that monitors the net). Once assembled, the response packet is queued for
   transmission by the appropriate bridge portal. */

VOID synthesizeResponse(PACKET *request, BYTE rcode, BYTE extRcode) {

PACKET response;

   memset(&response, 0, sizeof(response));
   response.destination.nodeID = request->source.nodeID;
   response.tl = request->tl;
   response.rt = (dualPhase) ? RETRY_1 : RETRY_X;
   if (request->tcode == QUADLET_WRITE_REQUEST || request->tcode == BLOCK_WRITE_REQUEST)
      response.tcode = WRITE_RESPONSE;
   else if (request->tcode == QUADLET_READ_REQUEST)
      response.tcode = QUADLET_READ_RESPONSE;
   else if (request->tcode == BLOCK_READ_REQUEST)
```

Table D-5 — Transaction routing functions (Sheet 4 of 4)

```
      response.tcode = BLOCK_READ_RESPONSE;
   else if (request->tcode == LOCK_REQUEST)
      response.tcode = LOCK_RESPONSE;
   response.source.nodeID = request->destination.nodeID;
   response.rcode = rcode;
   response.extRcode = extRcode;
   response.proxy.nodeID = ownGlobalID.nodeID;     /* For diagnostic and management purposes */
   queuePacket(FALSE, FALSE);                      /* Transmit response to original requester */

}
```

Annex E

(normative)

Discovery and enumeration protocol (DEP)

At the time of writing, the most common discovery and enumeration method used on a single Serial Bus is the exhaustive examination of configuration ROM for all 62 possible devices on the local bus. When this technique is combined with an intelligent analysis of self-ID packets that monitors local topology changes (see Annex G), the bus traffic required for discovery might be kept within manageable limits. However, this brute-force method does not scale when applied to the number of devices—in excess of 65 000—that could be present in a fully populated Serial Bus net. The discovery and enumeration protocol (DEP) specified within this annex is designed to provide a scalable method that reaches across bridges, but it is also suitable for use on the local bus alone. Devices that implement the protocol specified by this annex need not be compliant with other parts of this standard. Devices compliant solely with this annex shall zero the *bridge_aware* bit in their configuration ROM bus information block.

E.1 Message formats

DEP discovery requests and announcements shall be encapsulated within GASP transmitted on the default broadcast channel. GASP is specified by IEEE Std 1394a-2000; the structure of the data payload is reproduced in Figure E-1 for convenience of reference. The DEP request or announcement is contained within the shaded area, whose format is specified in E.1.1 through E.1.4, inclusive.

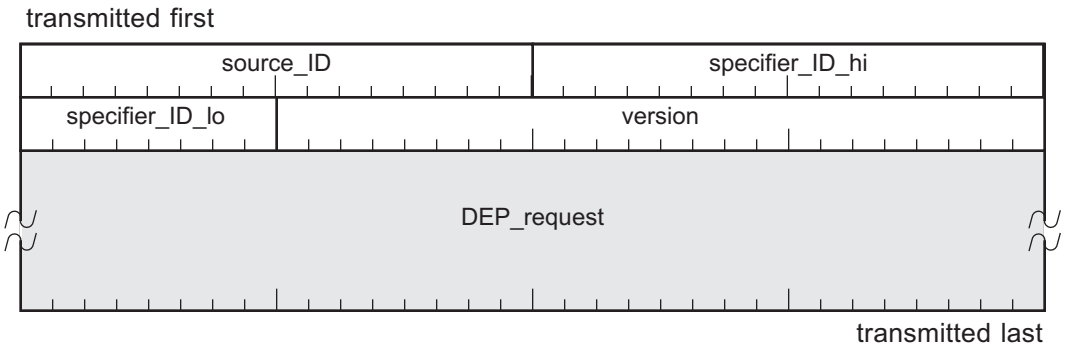


Figure E-1 — DEP request encapsulated within GASP data payload

The definition and usage of the *source_ID* field is specified by IEEE Std 1394a-2000. If the GASP has passed through one or more bridges, *source_ID* contains a global node ID.

The value of the *specifier_ID_hi* and *specifier_ID_lo* fields (collectively referred to as *specifier_ID*) shall be $00\,A03F_{16}$, the Organizationally Unique Identifier (OUI) granted to the IEEE Microprocessor Standards Committee (MSC) by the IEEE Registration Authority. This value indicates that the MSC and its Working Groups are responsible for the maintenance of this standard.

The value of the *version* field shall be $00\,0201_{16}$. The combined 48-bit value of the *specifier_ID* and *version* fields identifies this standard as the document that specifies the meaning of the *DEP_request* that follows.

The *DEP_request* field shall contain a DEP discovery request or announcement whose format is specified by this standard.

The originator of a DEP request or announcement encapsulated within GASP can control the scope of its broadcast by setting the *sy* field in the packet header to zero or eight (see 6.4).

Other DEP requests are encapsulated within a 64-byte block write request addressed to the MESSAGE_REQUEST register of a discovery proxy. IEEE Std 1212-2001 specifies the format and meaning of data payload written to this register; the structure of the data payload is reproduced in Figure E-2 for convenience of reference. The DEP request is contained within the shaded area, whose format is specified in E.1.3.

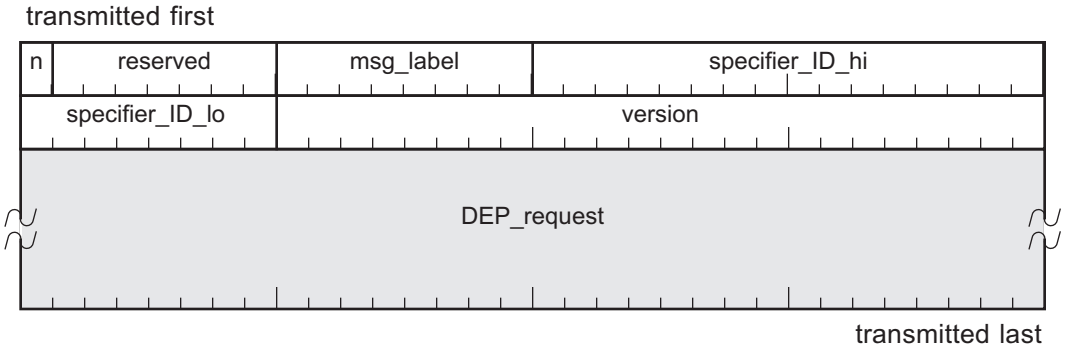


Figure E-2 — DEP request encapsulated within MESSAGE_REQUEST data payload

The definition and usage of the *notify* bit (abbreviated as *n* in Figure E-2) and the *msg_label*, field are specified by IEEE Std 1212-2001. The *notify* bit shall be one.

The value of the *specifier_ID_hi* and *specifier_ID_lo* fields (collectively referred to as *specifier_ID*) shall be $00\,A03F_{16}$, the Organizationally Unique Identifier (OUI) granted to the IEEE Microprocessor Standards Committee (MSC) by the IEEE Registration Authority. This value indicates that the MSC and its Working Groups are responsible for the maintenance of this standard.

The value of the *version* field shall be $00\,0201_{16}$. The combined 48-bit value of the *specifier_ID* and *version* fields identifies this standard as the document that specifies the meaning of the *DEP_request* that follows.

The *DEP_request* field shall contain a DEP request in a format specified by this standard.

DEP responses are encapsulated within a 64-byte block write request addressed to the MESSAGE_RESPONSE register of the node that originated the DEP request. IEEE Std 1212-2001 specifies the format and meaning of data payload written to the MESSAGE_RESPONSE register; the structure of the data payload is reproduced in Figure E-3 for convenience of reference. The DEP response is contained within the shaded area, whose format is specified in E.1.5.

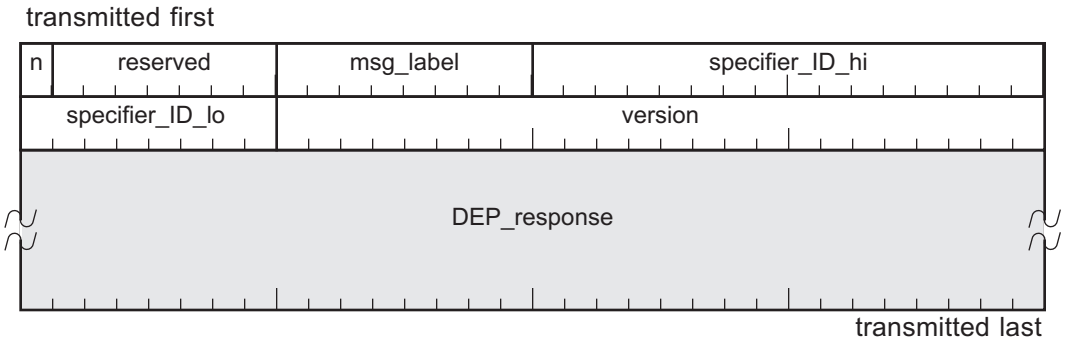


Figure E-3 — DEP response encapsulated within MESSAGE_RESPONSE data payload

The definition and usage of the *notify* bit (abbreviated as *n* in the figure above) and the *msg_label*, field are specified by IEEE Std 1212-2001.

For a DEP response associated with a DEP request received *via* GASP, the *notify* bit shall be zero. Otherwise, in response to a DEP request received at the responder's MESSAGE_REQUEST register, the value of the *notify* bit shall be in accordance with IEEE Std 1212-2001.

The *msg_label* field permits the recipient of a DEP response to correlate it with a previously transmitted DEP request. Its value shall be equal to *msg_label* in the DEP request to which this response corresponds.

The value of the *specifier_ID_hi* and *specifier_ID_lo* fields (collectively referred to as *specifier_ID*) shall be $00\,A03F_{16}$, the Organizationally Unique Identifier (OUI) granted to the IEEE Microprocessor Standards Committee (MSC) by the IEEE Registration Authority. This value indicates that the MSC and its Working Groups are responsible for the maintenance of this standard.

The value of the *version* field shall be $00\,0201_{16}$. The combined 48-bit value of the *specifier_ID* and *version* fields identifies this standard as the document that specifies the meaning of the *DEP_request* that follows.

The *DEP_response* field shall contain a DEP response in the format specified by this standard.

A node that transmits a DEP response shall do so within the time limit specified by its SPLIT_TIMEOUT register. The time period commences with the receipt of the DEP request to which the DEP response corresponds.

E.1.1 EUI-64 discovery request

This DEP request, illustrated by Figure E-4, asks a target node, uniquely identified by its EUI-64, to return a DEP response whose *source_ID* field shall contain a local or global node ID that can be used to address asynchronous requests directly to the node. If a discovery or service proxy is active for the target node, it is possible for the DEP requester to receive more than one response.

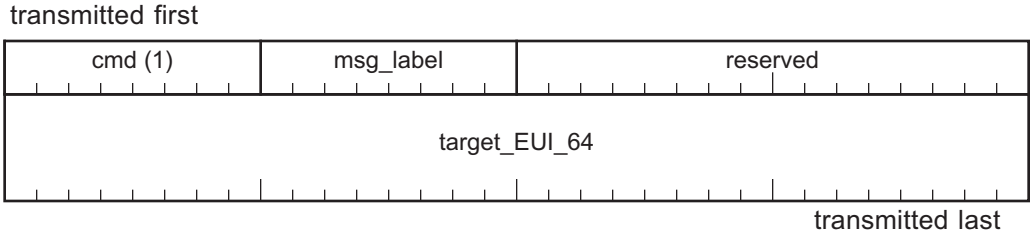


Figure E-4 — EUI-64 discovery request format

The *cmd* field identifies the message as a DEP EUI-64 discovery request; its value shall be one.

The meaning and usage of the *msg_label* field shall be determined by the originator of the DEP request. A node that transmits a DEP response in reply shall copy this value to the *msg_label* field in the MESSAGE_RESPONSE header that precedes the DEP response.

The *target_EUI_64* field contains the unique identifier of the sought-after node. The recipient of an EUI-64 discovery request whose bus information block contains an EUI-64 equal to this field is expected to transmit a DEP response to the requester's MESSAGE_RESPONSE register. A node that receives an EUI-64 discovery request for which *target_EUI_64* is not equal to the EUI-64 in its own bus information block shall not transmit a DEP response in reply unless it is a discovery or service proxy for the node identified by *target_EUI_64* (see E.2).

E.1.2 Keyword discovery request

This DEP request, illustrated by Figure E-5, asks the set of target devices identified by a set of keywords to return DEP responses whose *source_ID* field shall contain a node ID that can be used to address asynchronous requests directly to the node. The set of devices that matches the keyword criteria might be empty. The keyword discovery facility is not intended to conclusively identify nodes of interest; instead, it permits rapid identification of a (presumably manageable) set of nodes whose exact characteristics can be determined by inspection of their configuration ROM.

The keywords that are the target of the search are fully described in IEEE Std 1212-2001. Keywords deliberately lack formal definition, as it is intended that an evolutionary process govern which keywords are eventually found in common usage in particular classes of Serial Bus devices.

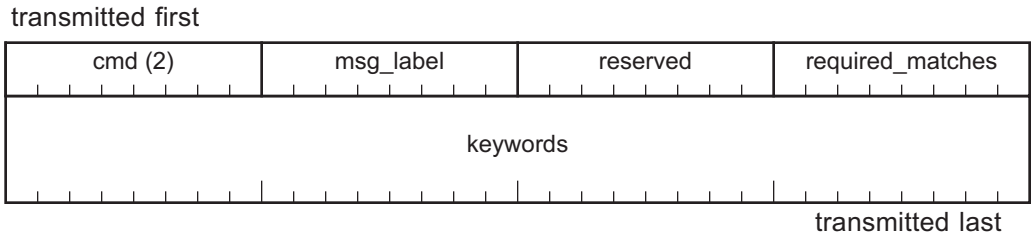


Figure E-5 — Keyword discovery request format

The *cmd* field identifies the message as a DEP keyword discovery request; its value shall be two.

The meaning and usage of the *msg_label* field shall be determined by the originator of the DEP request. A node that transmits a DEP response in reply shall copy this value to the *msg_label* field in the MESSAGE_RESPONSE header that precedes the DEP response.

The *required_matches* field shall specify the minimum number of keywords that shall be present in the recipient's configuration ROM in order for the recipient to satisfy the search criteria. For a nonzero value *n*, a recipient of a DEP keyword discovery request shall not respond unless the first *n* keywords specified by the request are present in the union of the recipient's keyword leaves.

The *keywords* field is a variable-length component of the keyword discovery request that shall contain a list of sought-after keywords. The format of this field shall be identical to that of a keyword leaf, as specified by IEEE Std 1212-2001, absent the length and CRC fields that commence leaves in configuration ROM. That is, the *keywords* field consists of individual keywords, each of which is a zero-terminated ASCII string. The number of keywords present in the field shall be greater than or equal to the value of *required_matches*. The *keywords* field shall be padded with bytes of zero to an integral number of quadlets.

Subject to the constraint imposed by a nonzero *required_matches* field, the recipient of a keyword discovery request is expected to transmit a DEP response to the requester's MESSAGE_RESPONSE register if one or more keywords specified in the *keywords* field are present in one or more of the recipient's keyword leaves.

E.1.3 Client ID request

This DEP request, illustrated by Figure E-6, asks a discovery proxy to return the global node ID of one of its clients.

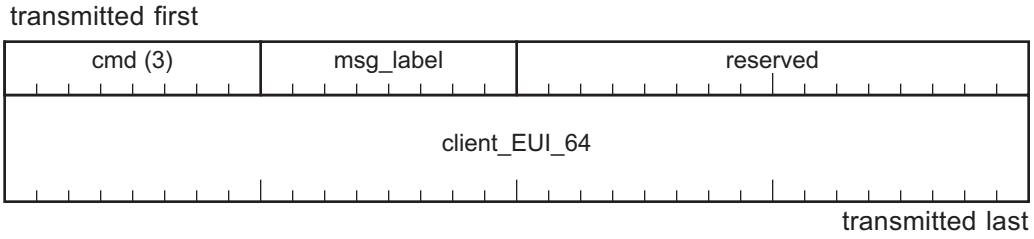


Figure E-6 — Client ID request format

The *cmd* field identifies the message as a DEP client ID request; its value shall be three.

The meaning and usage of the *msg_label* field shall be determined by the originator of the DEP request. A node that transmits a DEP response in reply shall copy this value to the *msg_label* field in the MESSAGE_RESPONSE header that precedes the DEP response.

The *client_EUI_64* field contains the unique identifier of the node whose global node ID is sought. A discovery proxy that receives a client ID request and recognizes the specified EUI-64 as belonging to one of its clients is expected to transmit a DEP response to the requester's MESSAGE_RESPONSE register. The discovery proxy should insure, by means beyond the scope of this standard, that the client's link and transaction layers are active before its transmits a client ID response to the requester.

E.1.4 Configuration ROM announcement

A node that wishes to inform other nodes, local or remote, of the current contents of its configuration ROM bus information block may transmit an announcement in the format illustrated by Figure E-7.

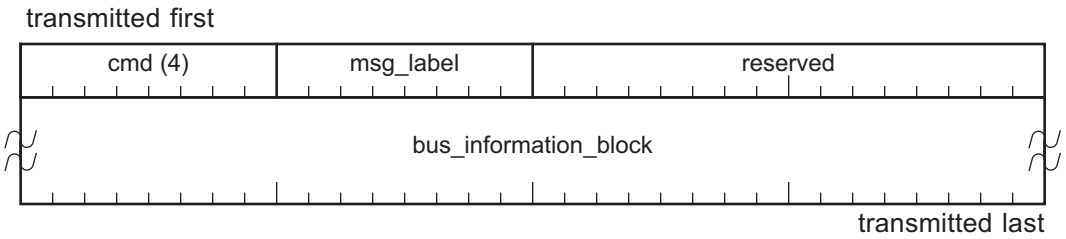


Figure E-7 — Configuration ROM announcement format

The *cmd* field identifies the message as a DEP client ID request; its value shall be four.

The *msg_label* field shall be zero

The *bus_information_block* field shall be equal to the contents of the transmitter's bus information block. When a Serial Bus node transmits a DEP configuration ROM announcement, the size of *bus_information_block* is four quadlets. In other cases, when the source is a node on a bus other than IEEE 1394 but compliant with the CSR architecture, the size of the field is determined by the applicable bus standard.

The circumstances in which a node transmits a DEP configuration ROM announcement are implementation-dependent, but it is strongly recommended that an announcement not be made unless the node has been connected to its bus or the *generation* field in its configuration ROM has changed.

The recipient of a DEP configuration ROM announcement can obtain the sender's likely global or local node ID from the *source_ID* field in the GASP header. However, the recipient shall verify that the global or local node ID corresponds to the node identified by the EUI-64 contained within the *bus_information_block* field before transmitting any request subactions to the node that could alter its state.

E.1.5 DEP responses

When the search criteria of either an EUI-64 or keyword discovery request are satisfied, either directly within the recipient's configuration ROM or indirectly *via* information known to a proxy, the expected response is a message of the format illustrated by Figure E-8 transmitted to the MESSAGE_RESPONSE register of the DEP requester. The same format is also used when a discovery proxy transmits a response to a client ID request.

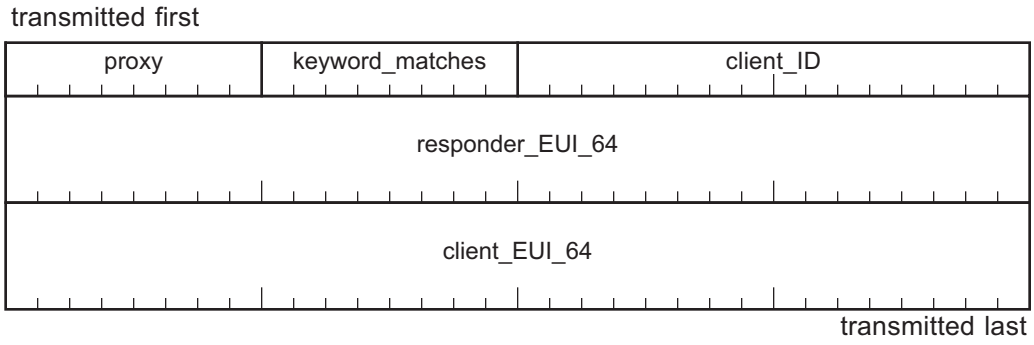


Figure E-8 — DEP response format

The *proxy* field shall indicate the type of node that transmitted the DEP response, as specified by the table below.

proxy	Description
0	Direct response from the target node itself.
1	Response from a discovery proxy.
2	Response from a service proxy.
$3 – FF_{16}$	Reserved for future standardization

See E.2 for additional information on the operations of proxies.

When a DEP response is transmitted in response to an EUI-64 discovery request, the *keyword_matches* field shall be zero. Otherwise, its value shall be nonzero and equal to the number of matches between the keyword list in the discovery request and keywords present in the union of the targeted device's keyword leaves. A keyword that is present in more than one leaf shall count as one match. When a DEP response is transmitted in reply to a keyword discovery request, the value of *keyword_matches* shall be greater than or equal to the value of *required_matches* in the discovery request.

When the *proxy* field is other than one (a response from a discovery proxy), the contents of *client_ID* are unspecified. Otherwise, the field may contain a node ID valid for reference to the device identified by *client_EUI_64*. If the identified device is not in a state responsive to transaction requests, the discovery proxy shall return a *client_ID* value of $FFFF_{16}$. When the *proxy* field is equal to one and the identified device is responsive to transaction requests, the discovery proxy that originates the response shall set *client_ID* to the local node ID of the identified device. If the DEP requester is identified by a global node ID, the discovery proxy shall also set the *snarf* field in the packet header to two; this causes

the initial entry portal to intercept the message and replace the local node ID in the *client_ID* field with a global node ID, set the *snarf* field to zero, and forward or echo the DEP response. Unlike other recipients of acknowledgments for packets with a nonzero *snarf* value (see 6.5), the DEP proxy shall not interpret *ack_pending* as transaction completion but shall await a response from the remote recipient.

The *responder_EUI_64* field shall contain the EUI-64 of the node that transmits the response.

When the *proxy* field is zero, the contents of *client_EUI_64* are unspecified. Otherwise, the *client_EUI_64* field shall be equal to the EUI-64 of the node on whose behalf the proxy has transmitted a DEP response.

E.2 Proxy functions

A DEP proxy is a device that responds to DEP requests on behalf of one or more clients located on the same bus as the proxy. Two types of proxy are defined by this standard—a discovery proxy and a service proxy. A discovery proxy permits DEP requesters to obtain the identity of a device that is temporarily unable to respond, as when the client device is in a power-saving state in which it cannot observe DEP requests. Discovery proxies service EUI-64 discovery requests and optionally service keyword discovery requests on behalf of their clients. A service proxy supplements or extends the functional capabilities of its client devices. For example, a vendor could make a service proxy available to customers in order to remedy a defect in an installed base of devices.

The details of communication between a proxy and its clients are beyond the scope of this standard.

E.2.1 Discovery proxy

A discovery proxy is a device that receives an EUI-64 discovery request, keyword discovery request, or client ID request and transmits a response on behalf of a client device, on the same bus as the discovery proxy, whose design or present state prevents it from observing or responding to the DEP request.

The details of a discovery proxy are largely vendor-dependent, but the following requirements apply:

— A discovery proxy intended to respond to EUI-64 discovery requests caches up to 62 EUI-64 values for nodes on the local bus along with a flag to indicate whether the proxy is enabled for a particular node. The discovery proxy shall obtain each node's unique ID, EUI-64, from the node's bus information block by means of two quadlet read transactions.

— A discovery proxy intended to respond to keyword discovery requests additionally caches an image of the master keyword leaf (see IEEE Std 1212-2001 for details) for each of its clients. This information need not be visible in the discovery proxy's configuration ROM, but is necessary in order to perform keyword matching against received discovery requests.

— A discovery proxy does not respond to a client ID request for one of its clients if the client's link is inactive or the client is otherwise incapable of responding to transaction requests. This might be the case if the client is unpowered or in a state that reduces its power consumption. A discovery proxy may defer response to the client ID request until the client is sufficiently operational to respond to transaction requests or may respond with a *client_ID* value of $FFFF_{16}$ to indicate that the client is not accessible. The methods by which a discovery proxy causes a client to become operational are beyond the scope of a standard.

— A discovery proxy should not transmit a DEP response when the client itself is capable of doing so itself. It might not be possible to guarantee that both client and proxy never transmit duplicate responses, but it should be avoided.

E.2.2 Service proxy

A service proxy is a device whose configuration ROM contains one or more unit directories that represent functions available not in the proxy itself but in a client device represented by the proxy. The service proxy receives commands and, after transforming them into a format recognizable by the client, relays them to the client device for execution. In order to perform this role, a service proxy is subject to the following requirements:

— A service proxy contains, within its own configuration ROM, structures that represent the functions and characteristics of the client devices. These include, but are not limited to, instance directories, keyword leaves and unit directories.
— A service proxy responds to EUI-64 and, optionally, keyword discovery requests as a discovery proxy does—the key difference is the presence of configuration ROM structures and associated CSR facilities that permit the client device to be controlled *via* the service proxy.

Details of service proxy operation are strongly dependent upon the unit architectures represented and are beyond the scope of this standard.

E.3 Implementation requirements

Bridge-aware devices and bridge portals shall implement support for DEP capabilities as specified by the table below:

DEP capability	Bridge-aware device	Bridge portal
EUI-64 discovery request	Required	Required
Keyword discovery request	Required	Required
Client ID request	Optional	Optional
Configuration ROM announcement	Optional	Optional

Annex F

(normative)

Plug control registers

The CSR Architecture reserves a portion of register space for bus-dependent uses. The addresses of these bus-dependent registers are specified in terms of byte offsets within register space, where the base of register space is FFFF F000 0000_{16} relative to zero. Table F-1 summarizes the optional plug control registers specified by this standard.

Table F-1 — Plug control registers

Offset	Name	Notes
900_{16}	OUTPUT_MASTER_PLUG	Common output plug controls for the node.
904_{16} – $97C_{16}$	OUTPUT_PLUG	Output plug control registers for individual channels, OUTPUT_PLUG[0] through OUTPUT_PLUG[30].
980_{16}	INPUT_MASTER_PLUG	Common input plug controls for the node.
984_{16} – $9FC_{16}$	INPUT_PLUG	Input plug control registers for individual channels, INPUT_PLUG[0] through INPUT_PLUG[30].

A node that implements plug control registers shall support quadlet read requests for all implemented registers. The node shall also support lock requests for all implemented registers so long as the *destination_offset* is quadlet aligned, the *extended_tcode* is equal to two (compare and swap) and the *data_length* is equal to four. Neither quadlet write nor block write requests shall be supported for any plug control registers, whether implemented or not. If an otherwise valid request is received for an unimplemented plug control register, the node should reject the request with a response of *resp_address_error* but may complete the transaction with *resp_complete* and response data of zeros.

A node may support block read requests addressed to the plug control register address space. If the combination of *destination_offset* and *data_length* for a block read request includes unimplemented plug control registers, the node may reject the request with a response of *resp_address_error*. However, if the node successfully completes the transaction, the response data returned for the unimplemented registers shall be zero.

F.1 OUTPUT_MASTER_PLUG register

The OUTPUT_MASTER_PLUG register, defined by Figure F-1, provides information about and permits control of common aspects of a node's OUTPUT_PLUG registers.

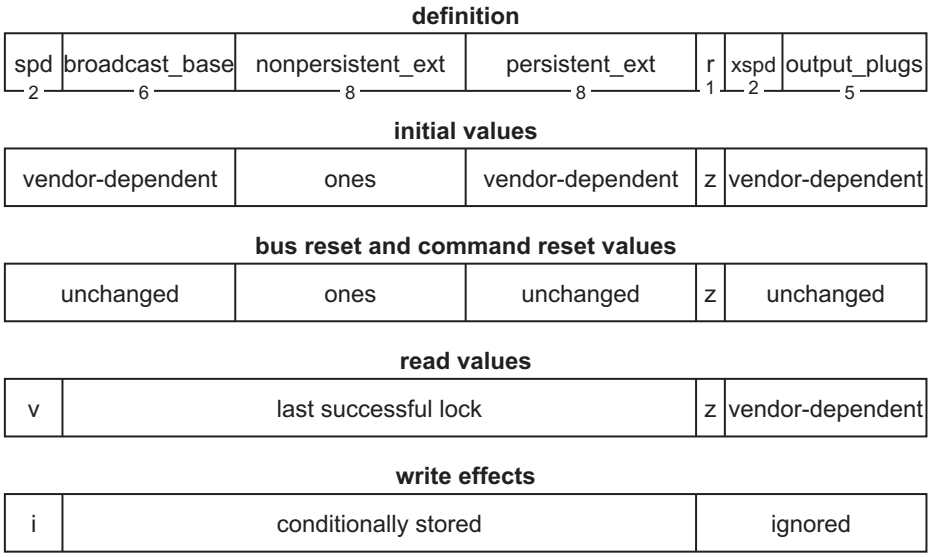


Figure F-1 — OUTPUT_MASTER_PLUG format

The ***spd*** field shall specify the maximum speed for isochronous data transmission that any of the OUTPUT_PLUG registers may use, as encoded by Table F-2.

Table F-2 — Speed encoding

Value of *spd*	Data rate
0	S100
1	S200
2	S400
3	Maximum data rate specified by *xspd*

The ***broadcast_base*** field shall specify the base channel number used to determine the channel number used for broadcast out connections. When a broadcast out connection is established for a plug for which a point-to-point connection does not simultaneously exist, the *channel* field of the OUTPUT_PLUG register shall be set to 63 if *broadcast_base* equals 63 and otherwise shall be set to (*broadcast_base* + *n*) modulo 63, where *n* is the ordinal of OUTPUT_PLUG[*n*].

The ***nonpersistent_ext*** and ***persistent_ext*** fields are reserved for future standardization.

When the *spd* field has a value of three, the ***xspd*** field shall specify the maximum speed for isochronous data transmission that any of the OUTPUT_PLUG registers may use, as encoded by Table F-3. Otherwise, if *spd* is less than three, *xspd* shall be zero.

Table F-3 — Extended speed encoding

Value of *xspd*	Data rate
0	S800
1	S1600
2	S3200
3	Reserved for future standardization

The ***output_plugs*** field shall specify the total count of OUTPUT_PLUG registers implemented by a node. Between zero and 31 OUTPUT_PLUG registers may be implemented. If one or more OUTPUT_PLUG registers are implemented, they shall lie within the contiguous address range FFFF F000 0904_{16} to FFFF F000 0900_{16} + 4 * *output_plugs*, inclusive.

F.2 OUTPUT_PLUG registers

Each OUTPUT_PLUG register, defined by Figure F-2, permits the description and control of both broadcast and point-to-point connections that originate with the associated plug. OUTPUT_PLUG registers shall be implemented within a contiguous address space and are referenced by an ordinal *n*, where *n* starts at zero; OUTPUT_PLUG[*n*] refers to the register addressable at FFFF F000 0904_{16} + 4 * *n*.

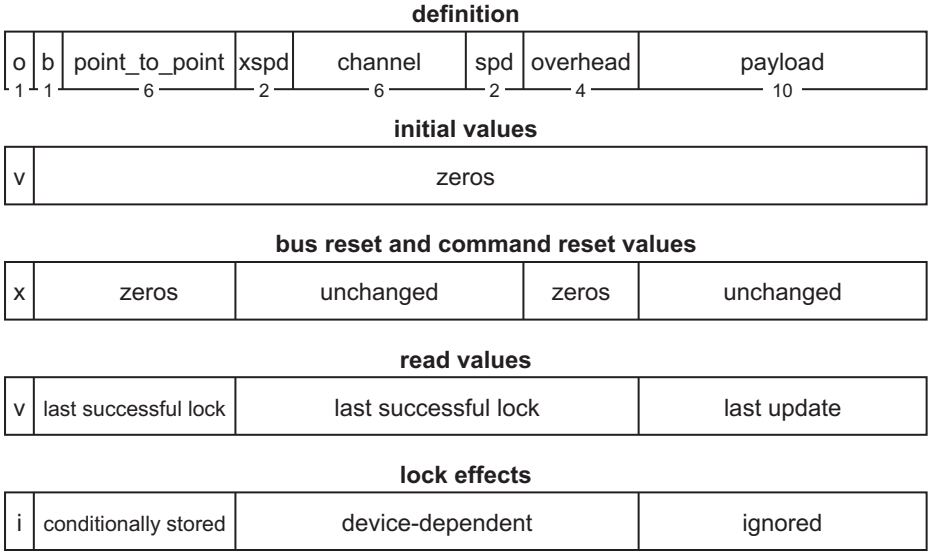


Figure F-2 — OUTPUT_PLUG format

The ***online*** bit (abbreviated as *o* in Figure F-2) shall specify the on-line status of the plug resources controlled by the OUTPUT_PLUG register. An *online* bit value of zero shall indicate that the plug is off-line and not capable of transmitting isochronous data. An *online* value of one shall indicate that the plug may be configured and used for isochronous data transmission.

NOTE—Plug status can change dynamically from off-line to on-line as device resources become unavailable or available. The causes of a change in plug status reported by the *online* bit are vendor-dependent.

The ***broadcast*** bit (abbreviated as *b* in Figure F-2) shall specify whether a broadcast connection exists for the output plug; a value of zero indicates that no such connection exists.

The ***point_to_point*** field shall specify the number of point-to-point connections that exist for the output plug.

When the *spd* field has a value of three, the ***xspd*** field shall specify the speed to be used for isochronous data transmissions for the plug, as encoded by Table F-3. Otherwise, the value of *xspd* shall be zero. If the *spd* field has a value of three and x*spd* is set to a value greater than the value of OUTPUT_MASTER_PLUG.*xspd*, isochronous data transmissions shall be disabled for the plug.

The ***channel*** field shall specify the channel number used in isochronous data transmissions for the plug.

The ***spd*** field shall specify the speed to be used for isochronous data transmissions for the plug, as encoded by Table F-2. If *spd* is set to a value greater than the value of the *spd* field in the OUTPUT_MASTER_PLUG register, isochronous data transmissions shall be disabled for the plug.

The ***overhead*** field shall encode a value used in the calculation of the isochronous bandwidth allocation necessary for isochronous data transmissions associated with the plug. Isochronous bandwidth is expressed in terms of bandwidth allocation units, as defined by IEEE 1394. One bandwidth allocation unit represents the time required to transmit one quadlet of data at the S1600 data rate, roughly 20 ns. If *overhead* is nonzero, the total bandwidth allocation necessary is expressed as $overhead * 32 + (payload + 3) * 2^{4 - (xspd + spd)}$. Otherwise, the total bandwidth allocation can be obtained from $512 + (payload + 3) * 2^{4 - (xspd + spd)}$. In the preceding formulae, *overhead*, *payload*, *spd,* and *xspd* represent the values of these fields in the OUTPUT_PLUG register.

NOTE—In the formulae above, there is a negative exponent at the S3200 data rate. When dividing by two at this data rate, the result should be rounded up to the next larger integer value.

The ***payload*** field shall specify the maximum number of quadlets that may be transmitted in a single isochronous packet for this plug. The interpretation of *payload* depends upon the value of OUTPUT_PLUG.*spd*. If *spd* is less than three, a *payload* value of zero indicates a maximum of 1024 quadlets; all other values represent a maximum of *payload* quadlets. Otherwise, if *spd* is equal to three, a *payload* value of zero indicates a maximum of $1024 * 2^{xspd + 1}$ quadlets; all other values represent a maximum of $payload * 2^{xspd + 1}$ quadlets.

NOTE—The value of *payload* does not include the isochronous header, header CRC or data CRC required as part of an isochronous packet; it counts only those quadlets that are part of the isochronous data payload.

F.3 INPUT_MASTER_PLUG register

The INPUT_MASTER_PLUG register, defined by Figure F-3, provides information about and permits control of common aspects of a node's INPUT_PLUG registers.

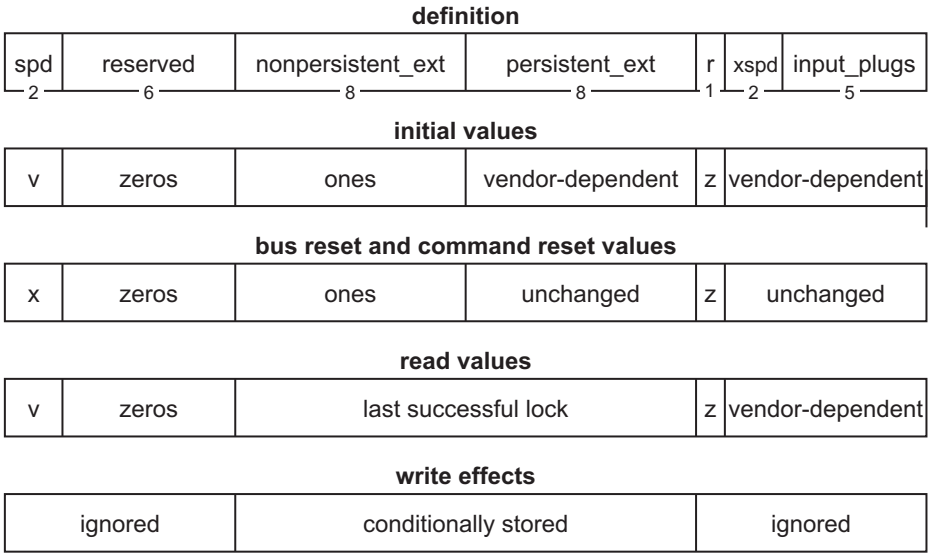


Figure F-3 — INPUT_MASTER_PLUG format

The ***spd*** field shall specify the maximum speed at which any of the node's input plugs may receive isochronous data, as encoded by Table F-2.

The ***nonpersistent_ext*** and ***persistent_ext*** fields are reserved for future standardization.

When the *spd* field has a value of three, the ***xspd*** field shall specify the maximum speed at which any of the node's input plugs may receive isochronous data, as encoded by Table F-3. If *spd* is less than three, the value of *xspd* shall be zero.

The ***input_plugs*** field shall specify the total count of INPUT_PLUG registers implemented by a node. Between zero and 31 INPUT_PLUG registers may be implemented. If one or more INPUT_PLUG registers are implemented, they shall lie within the contiguous address range FFFF F000 0984_{16} to FFFF F000 0980_{16} + 4 * *input_plugs*, inclusive.

F.4 INPUT_PLUG registers

Each INPUT_PLUG register, defined by Figure F-4, permits the description and control of point-to-point connections that terminate at the associated plug. INPUT_PLUG registers shall be implemented within a contiguous address space and are referenced by an ordinal *n*, where *n* starts at zero; INPUT_PLUG[*n*] refers to the register addressable at FFFF F000 0984_{16} + 4 * *n*.

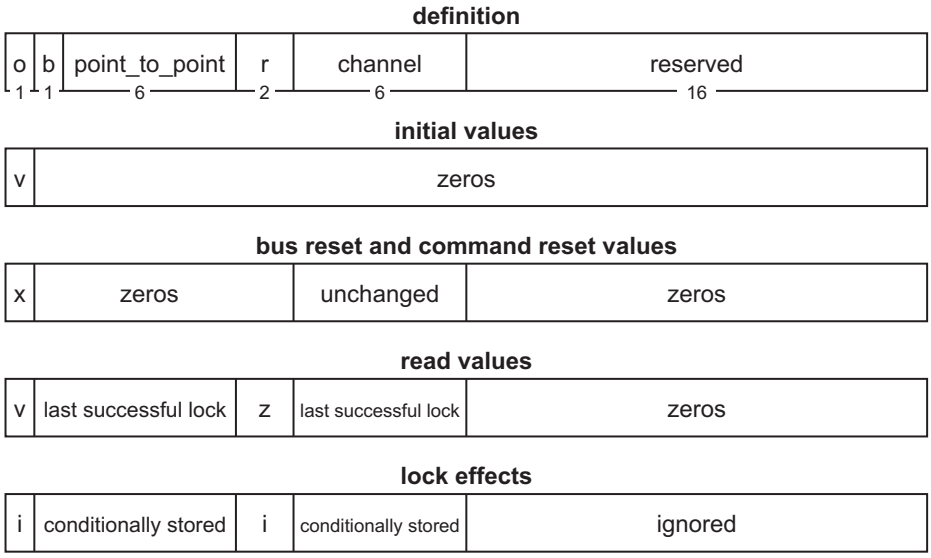


Figure F-4 — INPUT_PLUG format

The ***online*** bit (abbreviated as *o* in Figure F-4) bit shall specify the on-line status of the plug resources controlled by the INPUT_PLUG register. An *online* bit value of zero shall indicate that the plug is off-line and not capable of receiving isochronous data. An *online* value of one shall indicate that the plug may be configured and used for isochronous data reception.

NOTE—Plug status can change dynamically from off-line to on-line as device resources become unavailable or available. The causes of a change in plug status reported by the *online* bit are vendor-dependent.

The ***broadcast*** bit (abbreviated as *b* in Figure F-4) shall specify whether a broadcast connection exists for the input plug; a value of zero indicates that no such connection exists.

The ***point_to_point*** field shall specify the number of point-to-point connections that exist for the input plug.

The ***channel*** field shall specify the channel number used in isochronous data reception for the plug.

Annex G

(informative)

Bus topology analysis

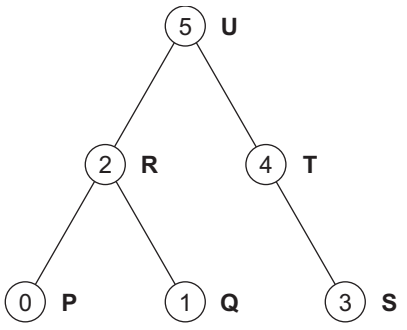
Subsequent to a bus reset, asynchronous transaction forwarding by bridges remains suspended until it is determined whether net topology has changed (10.2 describes this in detail). The timeliness of this determination has the potential to affect both the cost of bridge designs (larger buffers are required to keep remote transactions in suspense if a long time is required to confirm the absence of a net topology change) as well as the responsiveness of the entire net. As described in 10.2, an early step in determining whether a net topology change occurred is an analysis of the bus topology and unique identity of the connected nodes before and after the bus reset. The simplest method is to assume that the identity of all nodes could have changed and therefore to read the configuration ROM bus information block of each node to reestablish its unique identity (as determined by its EUI-64). This is inefficient and time-consuming; each node will arbitrate for the bus in attempts to read configuration ROM from all the other nodes. This flurry of activity occupies bus bandwidth needed for other time critical tasks, such as the reallocation of isochronous resources or net update. A superior method exists, one based upon topology information retained from conditions prior to the bus reset combined with an intelligent examination of the information contained in the set of self-ID packets.

NOTE—Because the algorithm relies on knowledge of the bus topology just prior to bus reset and compares it to the information in the self-ID packets, it is the essential that each set of self-ID packets associated with a bus reset be processed in the same order as the bus resets. If bus resets occur in rapid succession or interfere with completion of the self-identify phase, an implementation might lose part or all of one or more sets of self-ID packets, in which case there is no recourse except to read configuration ROM for all nodes on the bus.

This annex describes a method to perform intelligent analysis of self-ID packets to reestablish the EUI-64 identity of nodes connected before and after the bus reset. A node that uses this method derives a bus topology viewed from its own perspective, as if it were the root and all of its connections were child connections. The self-ID packets contain all the information necessary to derive such a normalized topology. The second step is the comparison of a saved copy of a normalized topology accurate before the bus reset with the newly derived normalized topology. When a child connection exists in the new map and a valid connection—either child or parent—existed in the prior map, the EUI-64 identity of the node is unchanged. Otherwise, when a new child connection is discovered, the EUI-64 identities of all nodes subsidiary to that connection are unknown and can be determined only by an examination of configuration ROM.

NOTE—Although this annex is part of a standard for bridges and the methods described are essential for bridges, they may be implemented separately by any node. Because these methods reduce the asynchronous traffic load on the bus subsequent to a bus reset, all nodes are strongly encouraged to implement topology analysis based upon self-ID packets.

The methods of self-ID packet analysis are discussed with reference to an example topology, illustrated by Figure G-1.



Self-ID packets

0	P	–	–
1	P	–	–
2	C	C	P
3	P	–	–
4	–	C	P
5	C	C	–

Figure G-1 — Reference topology (with self-ID packets)

In Figure G-1, nodes are represented by circles that contain their physical IDs. The root (physical ID 5 in this figure) is at the top. For the sake of brevity, a single letter shown to the right of each node represents its EUI-64. Although the PHY ports at each node are not explicitly numbered, this information can be deduced from the assigned physical IDs. The examples that follow assume that node identified by the letter T collects and analyzes the self-ID packets. The table to the right of Figure G-1 contains pertinent details from the set of self-ID packets generated for this topology; child and parent connections on a PHY port are abbreviated as C and P, respectively.

G.1 Topology analysis after power reset

For obvious reasons, a node that has just completed power reset has no knowledge of bus topology; from its perspective, the EUI-64 identity of all nodes is unknown. Part of the information the node requires is contained in any consistent set of self-ID packets collected after bus reset; these completely describe bus topology but they are not in a format useful for topology comparison subsequent to future bus resets. The first task is to reorganize this information into a normalized topology. The data structure that represents each node in the tree is represented graphically by Figure G-2.

Figure G-2 — Node data structure for normalized topology

The graphic representation shown in Figure G-2 is used in the figures that follow, which describe steps in the analysis of self-ID packets. The EUI-64 field represents the node's unique identifier (ultimately obtained from configuration ROM). The port fields are overloaded in the figures to represent either the port status reported in the self-ID packet (disconnected, child or parent) or a link to another node data structure in the normalized topology.[30] The overloading of these fields is more apparent in the C pseudocode definition of the data structure type (see Table G-1), in which they are separated from each other.

The first phase in the process is to analyze the self-ID packets in order to determine which nodes are connected to each other and by which numbered ports. The algorithm starts with the self-ID packets for the node with physical ID zero and concludes with the root. As each node's self-ID packets are encountered, transfer the port status into the corresponding node data structure. If the node is childless, also push the node's physical ID onto a stack; this information is used later in the process when the node's parent is encountered in the self-ID packets. In Figure G-3, the two self-ID packets that have been processed are shown shaded. Since both are childless, their physical IDs have been pushed onto the stack in the order they were encountered.

Figure G-3 — Self-ID packet topology analysis (nodes zero and one)

Whenever self-ID packets for a node indicate connected child ports, the processing is different. As before, copy the port status information from the self-ID packets to the node data structure—but do not push the node's physical ID onto the stack just yet. For each connected child port, pop a physical ID from the stack and establish a link between this node's data structure and the node data structure identified by the physical ID obtained from the stack As the stack values are

[30] Although PHYs may implement up to sixteen ports, the examples assume uniform use of three-port PHYs.

popped, the connections are made to this node's connected child ports in decreasing order. Upon completion, if the node has an unresolved parent connection, push the node's physical ID onto the stack. Figure G-4 shows the results of processing the self-ID packets for node two (shaded); links have been created to its children and its physical ID has been pushed onto the stack.

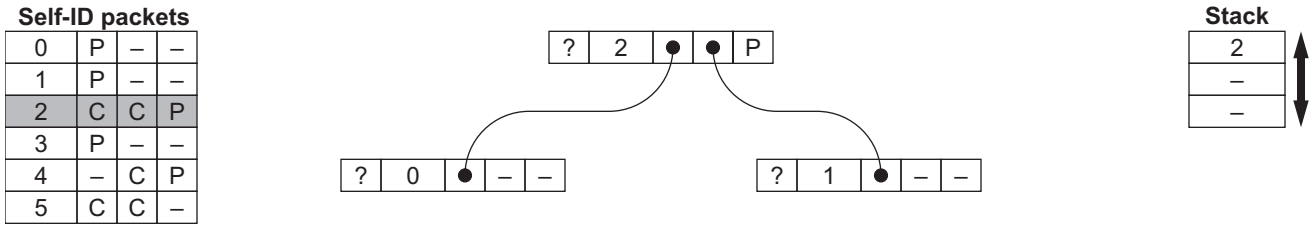


Figure G-4 — Self-ID packet topology analysis (node two)

This process continues as the physical ID of nodes with unresolved parent links are pushed onto the stack. Figure G-5 shows the result after processing another childless node.

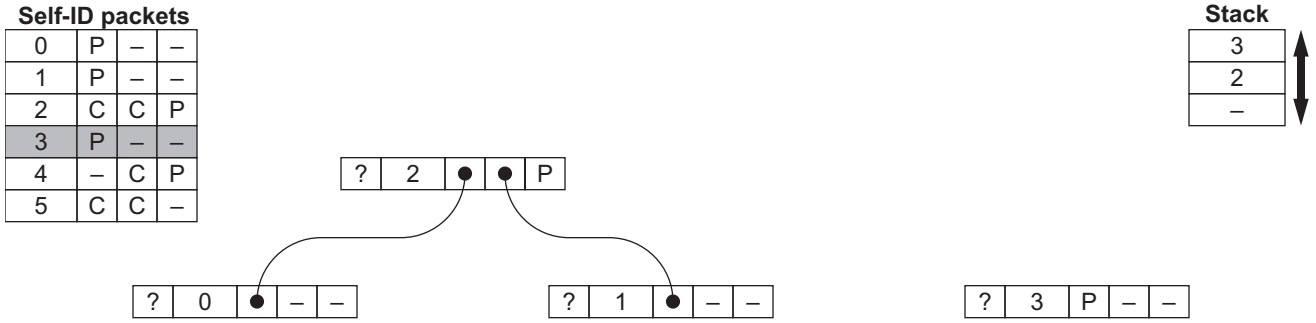


Figure G-5 — Self-ID packet topology analysis (node three)

Each time a node with connected child ports is encountered, corresponding physical ID entries are popped from the stack and links established in the topology that is under construction, as already described in association with Figure G-4. In the case of the node with physical ID four, the results are illustrated by Figure G-6.

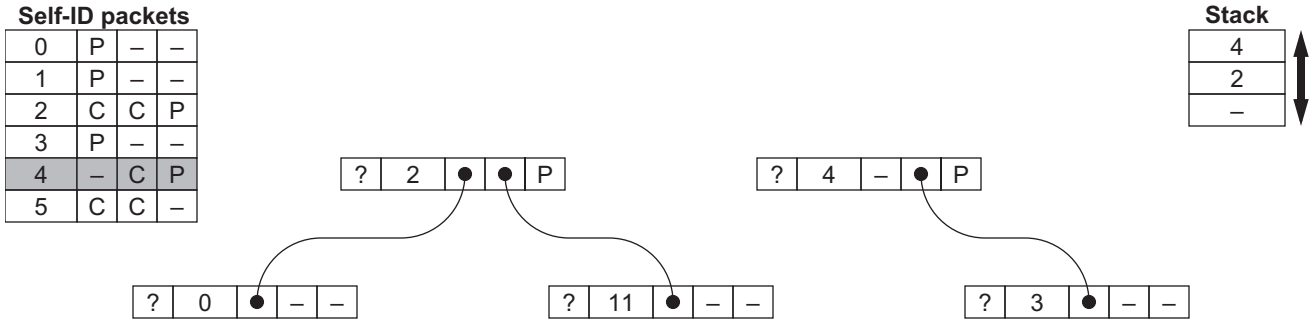


Figure G-6 — Self-ID packet topology analysis (node four)

Eventually the process comes to an end when the root node is encountered. All of its connected child ports are processed and links created in the topology. Since the root, by definition, has no connected parent, its physical ID is not pushed onto the stack. At this time, a complete topology from the perspective of the root has been derived from the self-ID packets, with the results shown by Figure G-7.

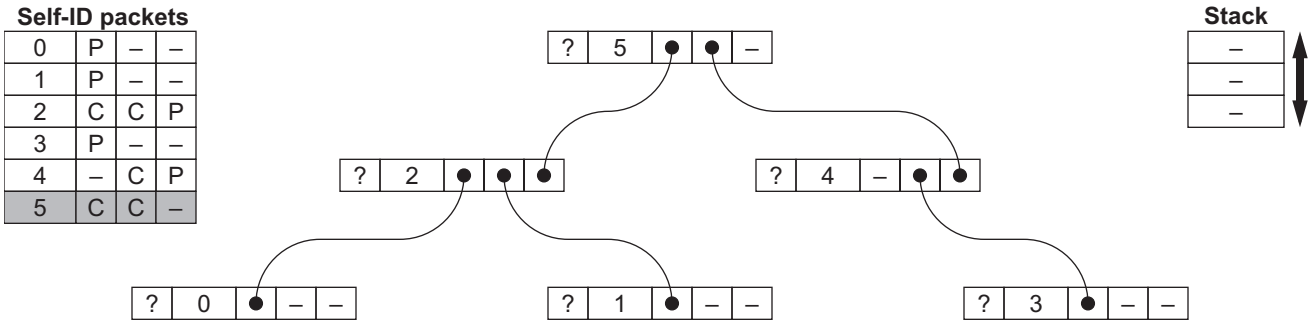


Figure G-7 — Self-ID packet topology analysis (node five)

Although Figure G-7 illustrates an accurate and complete bus topology, it is presented from the viewpoint of the root. In order to be useful for future comparisons with potentially changed topologies, the viewpoint is normalized to present the bus as if the node collecting the self-ID packets were the root. This is shown by Figure G-8, which assumes that node four is the observer.

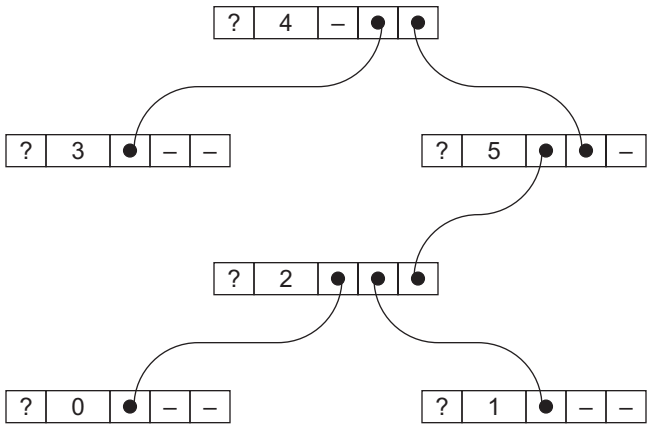


Figure G-8 — Normalized topology (relative to node four)

The normalized topology shown above is in a form suitable for comparison to the last known topology (prior to the bus reset). However, since this example assumes that the node has completed a power reset, there is no previous topology information and the only means to associate an EUI-64 identity with each node is to read its configuration ROM. In order to reduce asynchronous traffic congestion on the bus, read requests for configuration ROM should be deferred until one second after the most recent bus reset. Once configuration ROM has been read for all the unidentified nodes, the normalized topology is complete, as shown by Figure G-9.

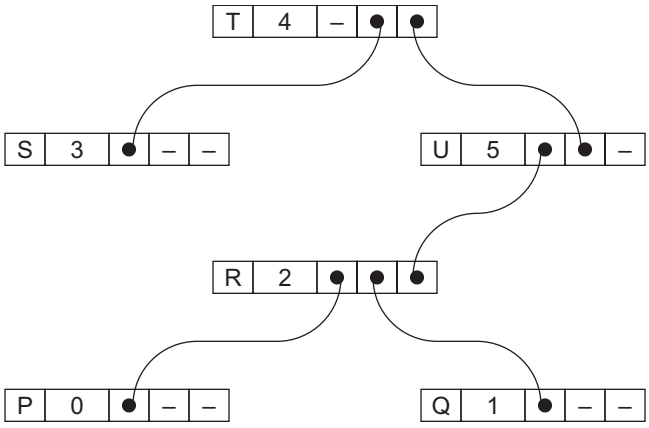


Figure G-9 — Normalized topology with EUI-64 information

All of the steps described previously are presented more formally in the C pseudocode excerpt in Table G-1. The algorithm, as embodied in the normalize_topology() procedure, assumes that a consistent set of self-ID packets has been observed and that their physical ID and port connection status have been transferred to an array in memory—the format of entries in this array is not identical to the self-ID packets themselves. Note that the code assumes that self-ID information is present for the node executing the algorithm; this may have been obtained from the node's PHY registers instead of directly from a self-ID packet.

Table G-1 — Topology analysis of self-ID packets (Sheet 1 of 2)

```
#include "csr.h"
#include "global.h"

typedef enum {DISCONNECTED=0, CHILD, PARENT} PORT;
typedef struct _NODE {
   OCTLET eui64;                /* Zero indicates unknown EUI-64 */
   PORT port[16];
   struct _NODE *link[16];
} NODE;

NODE node[63];                  /* Normalized topology derived from self-ID */
INT rootID;                     /* Set to root's physical ID after bus reset*/
struct {                        /* Port connection status information */
   PORT port[16];               /* Copied from each node's self-ID packets */
} selfID[63];

VOID normalize_topology() {

   INT i, j, m, n;

   memset(node, 0, sizeof(node));                  /* Clear the topology at the start */
   for (i = 0; i <= rootID; i++) {
      for (m = 16; m >= 0; --m) {
         node[i].port[m] = selfID[i].port[m];     /* Copy port connection status */
         if (selfID[i].port[m] == CHILD) {        /* Found a child connection? */
            j = pop();                            /* Yes, set j to child PHY ID */
            for (n = 0; n < 16; n++)              /* Scan for parent port */
               if (selfID[j].port[n] == PARENT) {
                  node[i].link[m] = &node[j];     /* Link from parent to child ... */
```

Table G-1 — Topology analysis of self-ID packets (Sheet 2 of 2)

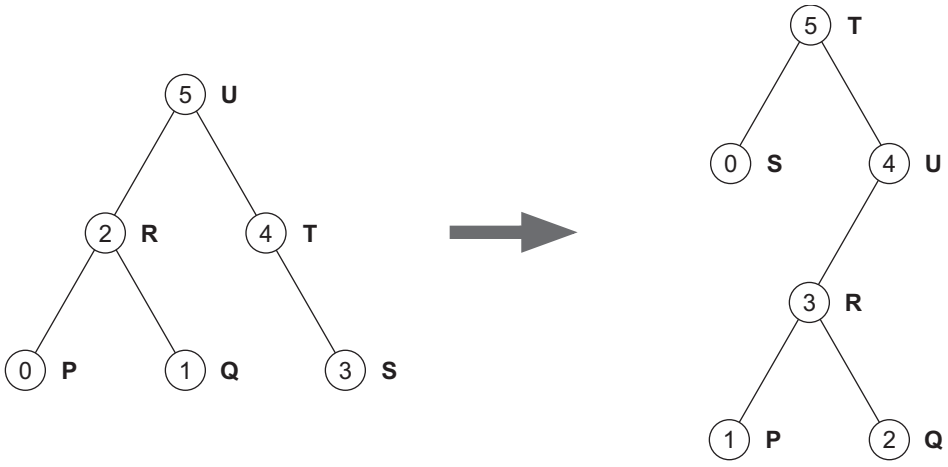
```
                    node[j].link[n] = &node[i];      /* ... and from child to parent */
                    break;                           /* Only one parent port per PHY */
                }
            }
        }
        if (i < rootID)
            push(i);       /* Remember this node for later parent link resolution */
    }

}
```

The normalized topology derived by this code should be saved for comparison with potentially changed topologies after future bus resets.

G.2 Topology analysis when the root changes

Even in cases where bus topology is unchanged before and after bus reset it is possible for the self-ID packets to differ. A common example occurs when the location of root changes, as when a node other than the current root has its root hold-off bit set prior to the bus reset. Figure G-10 illustrates the topologies (seen from the perspective of the root) and the self-ID packets generated when the root changes from node U to node T.



Self-ID packets

0	P	–	–
1	P	–	–
2	C	C	P
3	P	–	–
4	–	C	P
5	C	C	–

Self-ID packets

0	P	–	–
1	P	–	–
2	P	–	–
3	C	C	P
4	C	P	–
5	–	C	C

Figure G-10 — Reference topology (changed root)

A simple comparison of the self-ID packets fails to discern that the topologies are, in fact identical, but if the normalized topologies (derived as described in G.1) are compared it is apparent that the topologies are unchanged. First, the new set of self-ID packets is analyzed to produce the normalized topology illustrated by Figure G-11. At this point in the process, the only EUI-64 known with certainty is that of the node that observed the self-ID packets—in this case, the same node T used in the first example. Note that the new normalized topology reflects a changed physical ID for node T.

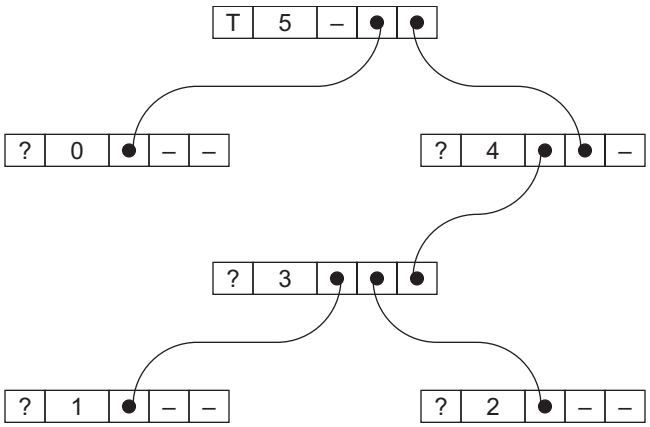


Figure G-11 — Normalized topology (relative to node five)

The next step is to compare the normalized topologies before and after bus reset in order to transfer as much EUI-64 information as possible from the saved information. The algorithm recursively traverses the two trees, starting from the position of the observing node. If a port is disconnected in the new topology, there is no need for any analysis. If the same port was connected to a child both before and after bus reset, the EUI-64 identity of the child is unchanged. The C pseudocode in Table G-2 describes the details of the algorithm.

Table G-2 — Normalized topology comparison

```
VOID updateEUI64(NODE *new, NODE *prior, NODE *parent) {

   int i;

   for (i = 0; i <= 16; i++) {              /* Compare old and new connection status on all ports */
      if (new->port[i] == DISCONNECTED)
         continue;                          /* Nothing to explore down this branch ... */
      if (new->link[i] == parent)            /* Does connection point to our parent? */
         continue;                          /* Been there, done that! */
      if (prior->port[i] != DISCONNECTED) {  /* Was same port connected in the prior topology? */
         new->link[i]->eui64 = prior->link[i]->eui64;     /* OK! We know it has the same EUI-64 */
         updateEUI64(new->link[i], prior->link[i], new);  /* Recurse down this branch of the tree */
      }
   }

}
```

The recursive process is started with a call to `updateEUI64()` with the parameters shown below:

`updateEUI64(&node[`phy_ID_{new}`], prior[`phy_ID_{prior}`], NULL);`

In the code excerpt above, `prior` and `new` are arrays of normalized topology information, the one from before the bus reset and the other current, while phy_ID_{prior} and phy_ID_{new} are the observing node's physical ID before and after the bus reset. Since the topologies are compared from the perspective of the observing node as if it were the root, there is no parent node and its parameter is null.

Upon completion of the algorithm, the normalized topology is updated with EUI-64 identity information from the prior topology, as shown by Figure G-12. Although the physical IDs of all the nodes have changed, their EUI-64 identities have been reestablished without recourse to reading configuration ROM.

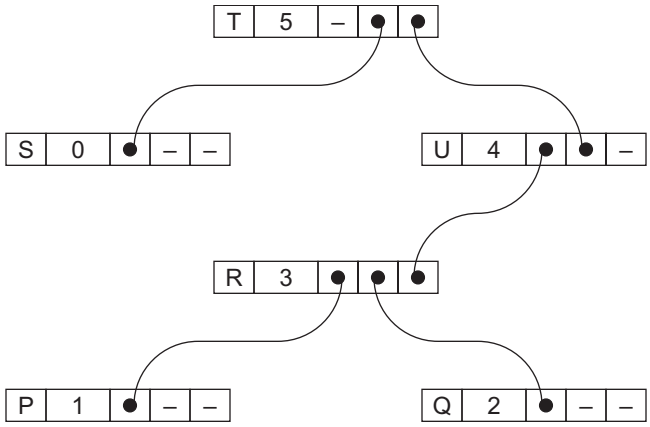
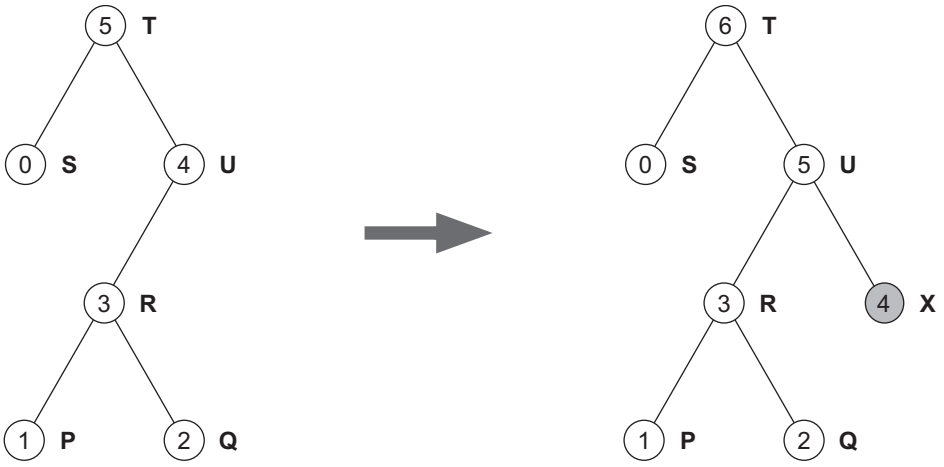


Figure G-12 — Normalized topology with EUI-64 information (changed root)

G.3 Topology analysis when a node is inserted

The methods already described in the context of detecting unchanged topologies are equally useful when a node is inserted. Consider the reference topology illustrated in the right half of Figure G-10. If a new node, with an EUI-64 represented by the letter X, is inserted, the topology is altered and a different set of self-ID packets is generated as shown by Figure G-13.



Self-ID packets

0	P	–	–
1	P	–	–
2	P	–	–
3	C	C	P
4	C	P	–
5	–	C	C

Self-ID packets

0	P	–	–
1	P	–	–
2	P	–	–
3	C	C	P
4	P	–	–
5	C	P	C
6	–	C	C

Figure G-13 — Reference topology (inserted node)

By the methods already described in G.1 and G.2, a normalized topology is derived and compared to the topology retained from before the node insertion. Figure G-14 illustrates the normalized topology with the addition of a new node.

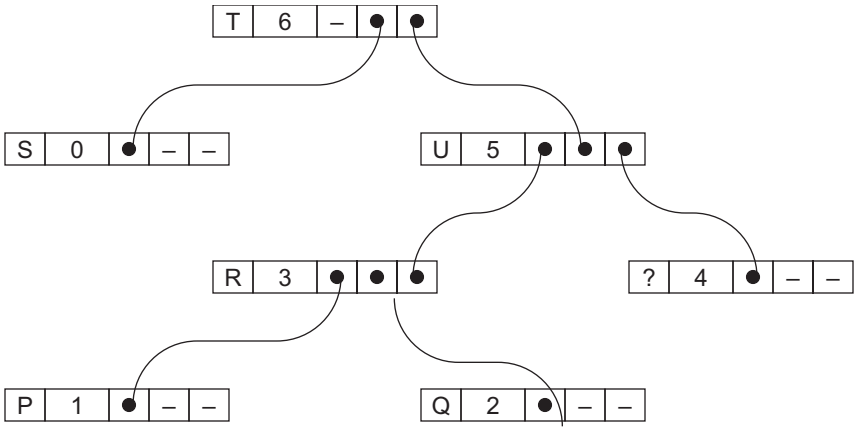


Figure G-14 — Normalized topology with EUI-64 information (inserted node)

At the completion of the recursive application of `updateEUI64()`, the nodes whose EUI-64 identity could not be deduced from topological analysis are left with zero for their EUI-64s. This is an invalid value for an EUI-64; it is necessary to read configuration ROM to obtain the EUI-64s for these nodes. Configuration ROM reads for newly inserted nodes should be deferred until at least one second since the most recent bus reset.

Annex H

(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 bridge-aware nodes and bridge portals are specified in 5.1. Figure H-1 shows a typical bus information block, root directory and bus-dependent information directory for a bridge portal. Configuration ROM for a bridge-aware node that is not a bridge portal omits the bus-dependent information directory.

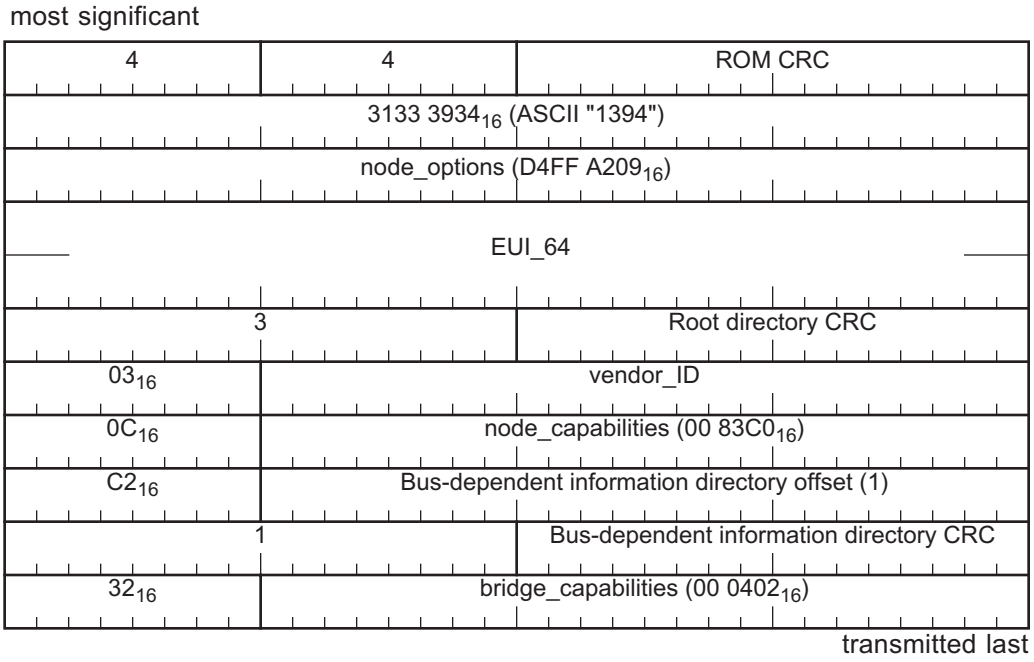


Figure H-1 — Bridge portal configuration ROM

The ROM CRC in the first quadlet of configuration ROM is calculated on the four quadlets of the bus information block that follow.

The *node_options* field represents a collection of bits and fields specified by this document. The value shown, D4FF $A209_{16}$, represents the fundamental characteristics of a bridge portal that is not isochronous-capable.[31] This value is composed of a *capabilities* field with a value of $D4_{16}$, a *cyc_clk_acc* field with a value of FF_{16}, a *max_rec* value of ten, a *max_ROM* value of two, a *bridge_aware* bit of one and a *link_spd* value of two. The *irmc*, *cmc*, *bmc,* and *adjustable* bits in the *capabilities* field are one to indicate that the portal is cycle master-, isochronous resource manager- and bus manager-capable and that it recognizes cycle master adjustment packets. The *max_rec* field describes a maximum payload of 2048 bytes for asynchronous subactions transferred to the co-portal (as well as in block requests addressed to the portal) while the *max_ROM* field indicates that configuration ROM supports block read requests up to 1024 bytes in length. The *link_spd* field describes a link that can operate at speeds up to S400.

[31] The bridge supports the transfer of isochronous streams (see the Bridge_Capabilities entry) but the bridge portal does not function as an endpoint for isochronous streams.

The value of the Node_Capabilities entry in the root directory, whose *key* field is $0C_{16}$, shows the *spt*, *64*, *fix*, *lst,* and *drq* bits set to one. This is a minimum requirement for nodes compliant with IEEE 1394.

The Bus_Dependent_Info entry in the root directory, with a key field of $C2_{16}$, addresses the bus-dependent information directory that immediately follows the root directory. Within that directory, the Bridge_Capabilities entry, with a key field of 32_{16}, describes a bridge portal that supports up to four isochronous streams simultaneously; when an isochronous stream is transferred across the bridge fabric to the co-portal, it incurs a constant delay of 250 μs.

Annex I

(informative)

Bibliography

[B1] IEC 61883-4: 2004, Consumer audio/video equipment—Digital interface—Part 4: MPEG2-TS data transmission.

[B2] IEEE 100,*The Authoritative Dictionary of IEEE Standard Terms*, Seventh Edition, New York, The Institute of Electrical and Electronics Engineers, Inc.

[B3] ISO/IEC 9899:1999, Programming languages—C.