JggDvj20050416
2007-06-20
(June 20, 2007)

DVJ Perspective on: Timing and synchronization for time-sensitive applications in bridges local area networks

Draft 0.711

Contributors:
See page xx.

Abstract: This working paper provides background and introduces possible higher level concepts for the development of Audio/Video bridges (AVB).
Keywords: audio, visual, bridge, Ethernet, time-sensitive

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.

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
P.O. Box 1331
Piscataway, NJ 08855-1331
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.

Editors' Foreword

Comments on this draft are encouraged. **PLEASE NOTE: All issues related to IEEE standards presentation style, formatting, spelling, etc. should be addressed, as their presence can often obfuscate relevant technical details.**

By fixing these errors in early drafts, readers can devote their valuable time and energy to comments that materially affect either the technical content of the document or the clarity of that technical content. Comments should not simply state what is wrong, but also what might be done to fix the problem.

Information on 802.1 activities, working papers, and email distribution lists etc. can be found on the 802.1 Website:

http://ieee802.org/1/

Use of the email distribution list is not presently restricted to 802.1 members, and the working group has had a policy of considering ballot comments from all who are interested and willing to contribute to the development of the draft. Individuals not attending meetings have helped to identify sources of misunderstanding and ambiguity in past projects. Non-members are advised that the email lists exist primarily to allow the members of the working group to develop standards, and are not a general forum.

Comments on this document may be sent to the 802.1 email reflector, to the editors, or to the Chairs of the 802.1 Working Group and Interworking Task Group.

This draft was prepared by:

David V James
JGG
3180 South Court
Palo Alto, CA 94306
+1.650.494.0926 (Tel)
+1.650.954.6906 (Mobile)
Email: dvj@alum.mit.edu

Chairs of the 802.1 Working Group and Audio/Video Bridging Task Group:.

Michael Johas Teener
Chair, 802.1 Audio/Video Bridging Task
Broadcom Corporation
3151 Zanker Road
San Jose, CA
95134-1933
USA
+1 408 922 7542 (Tel)
+1 831 247 9666 (Mobile)
Email:mikejt@broadcom.com

Tony Jeffree
Group Chair, 802.1 Working Group
11A Poplar Grove
Sale
Cheshire
M33 3AX
UK
+44 161 973 4278 (Tel)
+44 161 973 6534 (Fax)
Email: tony@jeffree.co.uk

Introduction to IEEE Std 802.1AS™

(This introduction is not part of P802.1AS, IEEE Standard for Local and metropolitan area networks—Timing and synchronization for time-sensitive applications in bridged local area networks.)

This standard specifies the protocol and procedures used to ensure that the synchronization requirements are met for time sensitive applications, such as audio and video, across bridged and virtual bridged local area networks consisting of LAN media where the transmission delays are fixed and symmetrical; for example, IEEE 802.3 full duplex links. This includes the maintenance of synchronized time during normal operation and following addition, removal, or failure of network components and network reconfiguration. The design is based on concepts developed within the IEEE Std 1588, and is applicable in the context of IEEE Std 802.1D and IEEE Std 802.1Q.

Synchronization to an externally provided timing signal (e.g., a recognized timing standard such as UTC or TAI) is not part of this standard but is not precluded.

Version history

Version	Date	Edits by	Comments
0.082	2005-04-28	DVJ	Updates based on 2005Apr27 meeting discussions
0.085	2005-05-11	DVJ	– Updated list-of-contributors, page numbering, editorial fixes.
0.088	2005-06-03	DVJ	– Application latency scenarios clarified.
0.090	2005-06-06	DVJ	– Misc. editorials in bursting and bunching annex.
0.092	2005-06-10	DVJ	– Extensive cleanup of Clause 5 subscription protocols.
0.121	2005-06-24	DVJ	– Extensive cleanup of clock-synchronization protocols.
0.127	2005-07-04	DVJ	– Pacing descriptions greatly enhanced.
0.200	2007-01-23	DVJ	Removal of non time-sync related information, initial layering proposal.
0.207	2007-02-01	DVJ	Updates based on feedback from Monterey 802.1 meeting. – Common entity terminology; Ethernet type code expanded.
0.216	2007-02-17	DVJ	Updates based on feedback from Chuck Harrison: – linkDelay based only on syntonization to one's neighbor. – Time adjustments based on observed grandMaster rate differences.
0.224	2007-03-03	DVJ	Updates for whiplash free PLL cascading.
0.230	2007-03-05	DVJ	Major changes: – simplified back-interpolation – first iteration on an Ethernet-PON interface – client-level clock-master and clock-slave interfaces defined
0.243	2007-04-20	DVJ	– Revised GrandSync entity illustrations – General cleanup
0.708	2007-05-30	DVJ	– Simulation results provided within an annex – Extensive code revisions for simplicity & clarity. – Interpolation better described.
—	TBD	—	—

Formats

In many cases, readers may elect to provide contributions in the form of exact text replacements and/or additions. To simplify document maintenance, contributors are requested to use the standard formats and provide checklist reviews before submission. Relevant URLs are listed below:

General: http://grouper.ieee.org/groups/msc/WordProcessors.html
Templates: http://grouper.ieee.org/groups/msc/TemplateTools/FrameMaker/
Checklist: http://grouper.ieee.org/groups/msc/TemplateTools/Checks2004Oct18.pdf

TBDs

Further definitions are needed in the following areas:

a) Should low-rate *leapSeconds* occupy space in timeSync frames, if this information rarely changes?
b) What other (than *leapSeconds*) low-rate information should be transferred between stations?
c) When the grand-master changes, how should the new grand-master affect change:
    1) Transition immediately to the rate of its reference clock.
    2) Transition slowly (perhaps 1ppm/s) between previous and reference clock rates.

Contents

List of figures 9

List of tables 11

1. Overview 13

1.1 Scope 13
1.2 Purpose 13
1.3 Introduction 13

2. References 15

3. Terms, definitions, and notation 16

3.1 Conformance levels 16
3.2 Terms and definitions 16
3.3 State machines 17
3.4 Arithmetic and logical operators 19
3.5 Numerical representation 19
3.6 Field notations 20
3.7 Bit numbering and ordering 21
3.8 Byte sequential formats 22
3.9 Ordering of multibyte fields 22
3.10 MAC address formats 23
3.11 Informative notes 24
3.12 Conventions for C code used in state machines 24

4. Abbreviations and acronyms 25

5. Architecture overview 27

5.1 Application scenarios 27
5.2 Design methodology 28
5.3 Grand-master selection 29
5.4 Synchronized-time distribution 31
5.5 Cascaded clock topologies 34
5.6 Time-affiliation adjustments 36
5.7 Sampling offset/rate conversion 37
5.8 Distinctions from IEEE Std 1588 40

6. GrandSync operation 41

6.1 Overview 41
6.2 Service interface primitives 43
6.3 GrandSync state machine 48

7. ClockMaster/ClockSlave state machines 51

7.1 Overview 51
7.2 ClockMaster service interfaces 52
7.3 ClockMaster state machine 53
7.4 ClockSlave service interfaces 56

7.5 ClockSlave state machine ... 57

8. Ethernet full duplex (EFDX) state machines ... 60

8.1 Overview ... 60
8.2 timeSyncEfdx frame format ... 63
8.3 TimeSyncRxEfdx state machine ... 65
8.4 TimeSyncTxEfdx state machine ... 68

9. Wireless state machines ... 72

9.1 Overview ... 72
9.2 Service interface definitions ... 74
9.3 TimeSyncRxR11v state machine ... 77
9.4 TimeSyncTxR11v state machine ... 80

10. Ethernet passive optical network (EPON) state machines ... 84

10.1 Overview ... 84
10.2 timeSyncEpon frame format ... 85
10.3 TimeSyncRxEpon service interface primitives ... 86
10.4 TimeSyncRxEpon state machine ... 87
10.5 TimeSyncTxEpon service interface primitives ... 90
10.6 TimeSyncTxEpon state machine ... 91

Annex A (informative) Bibliography ... 95

Annex B (informative) Time-scale conversions ... 96

B.1 Overview ... 96
B.2 TAI and UTC ... 96
B.3 NTP and GPS ... 97
B.4 Time-scale conversions ... 98
B.5 Time zones and GMT ... 99

Annex C (informative) Simulation results (preliminary) ... 100

C.1 Simulation environment ... 100
C.2 Initialization transients ... 101
C.3 Steady-state interpolation errors ... 102
C.4 Steady-state extrapolation errors ... 103

Annex D (informative) Bridging to IEEE Std 1394 ... 104

D.1 Hybrid network topologies ... 104

Annex E (informative) Time-of-day format considerations ... 106

E.1 Possible time-of-day formats ... 106

Annex F (informative) C-code illustrations ... 109

List of figures

Figure 1.1—Topology and connectivity 14
Figure 3.1—Bit numbering and ordering 21
Figure 3.2—Byte sequential field format illustrations 22
Figure 3.3—Multibyte field illustrations 22
Figure 3.4—Illustration of fairness-frame structure 23
Figure 3.5—MAC address format 23
Figure 3.6—48-bit MAC address format 24
Figure 5.1—Garage jam session 27
Figure 5.2—Possible looping topology 28
Figure 5.3—Timing information flows 29
Figure 5.4—Grand-master precedence flows 30
Figure 5.5—Grand-master preference 30
Figure 5.6—Hierarchical flows 31
Figure 5.7—Time-synchronization flows 32
Figure 5.8—Intermediate-bridge responsibilities 32
Figure 5.9—Cumulative sync-interval bunching 34
Figure 5.10—Cumulative sync-interval bunching 35
Figure 5.11—Mixed sync-interval systems 35
Figure 5.12—Receive/transmit adjustments 36
Figure 5.13—Extrapolation for *grandTime* 37
Figure 5.14—Extrapolation for *grandTime* 37
Figure 5.15—Interpolation for *grandTimeA* 38
Figure 5.16—Interpolation of *extraTimeD* 39
Figure 6.1—GrandSync interface model 41
Figure 6.2—GrandSync service-interface components 42
Figure 6.3—Global-time subfield format 44
Figure 6.4—precedence subfields 45
Figure 6.5—*clockID* format 45
Figure 6.6—Global-time subfield format 46
Figure 6.7—extraTime format 46
Figure 6.8—*snapTime* format 46
Figure 7.1—ClockMaster interface model 52
Figure 8.1—EFDX-link interface model 60
Figure 8.2—Contents of rxSync/txSync indications 60
Figure 8.3—Link-delay compensation 61
Figure 8.4—timeSyncEfdx frame format 63
Figure 9.1—R11v interface model 72
Figure 9.2—Formats of wireless-dependent times 72
Figure 9.3—802.11v time-synchronization interfaces 73
Figure 10.1—PON interface model 84
Figure 10.2—Format of PON-dependent times 84

Figure 10.3—timeSyncEpon frame format 85
Figure 10.4—tickTime format 86
Figure C.1—Time-synchronization flows 100
Figure C.2—Startup transients with 8 stations 101
Figure C.3—Startup transients with 64 stations 101
Figure C.4—Time interpolation with 8 stations 102
Figure C.5—Time interpolation with 64 stations 102
Figure C.6—Time extrapolation with 8 stations 103
Figure C.7—Time extrapolation with 64 stations 103
Figure D.1—IEEE 1394 leaf domains 104
Figure D.2—IEEE 802.3 leaf domains 104
Figure D.3—Time-of-day format conversions 105
Figure D.4—Grand-master precedence mapping 105
Figure E.1—Global-time subfield format 106
Figure E.2—IEEE 1394 timer format 106
Figure E.3—IEEE 1588 timer format 107
Figure E.4—EPON timer format 107

List of tables

Table 3.1—State table notation example 18
Table 3.2—Special symbols and operators 19
Table 3.3—Names of fields and sub-fields 20
Table 3.4—*wrap* field values 21
Table 6.1—GrandSync state table 50
Table 7.1—ClockMaster state machine table 55
Table 7.2—ClockSlave state table 59
Table 8.1—Clock-synchronization intervals 64
Table 8.2—TimeSyncRxEfdx state machine table 67
Table 8.3—TimeSyncTxEfdx state machine table 71
Table 9.1—TimeSyncRxR11v state machine table 79
Table 9.2—TimeSyncTxR11v state table 82
Table 10.1—TimeSyncRxEpon state machine table 89
Table 10.2—TimeSyncTxEpon state machine table 93
Table B.1—Time-scale parameters 96
Table B.2—Time-scale conversions 98

DVJ Perspective on: Timing and synchronization for time-sensitive applications in bridges local area networks

1. Overview

1.1 Scope

This draft specifies the protocol and procedures used to ensure that the synchronization requirements are met for time sensitive applications, such as audio and video, across bridged and virtual bridged local area networks consisting of LAN media where the transmission delays are fixed and symmetrical; for example, IEEE 802.3 full duplex links. This includes the maintenance of synchronized time during normal operation and following addition, removal, or failure of network components and network reconfiguration. It specifies the use of IEEE 1588 specifications where applicable in the context of IEEE Std 802.1D and IEEE Std 802.1Q. Synchronization to an externally provided timing signal (e.g., a recognized timing standard such as UTC or TAI) is not part of this standard but is not precluded.

1.2 Purpose

This draft enables stations attached to bridged LANs to meet the respective jitter, wander, and time synchronization requirements for time-sensitive applications. This includes applications that involve multiple streams delivered to multiple endpoints. To facilitate the widespread use of bridged LANs for these applications, synchronization information is one of the components needed at each network element where time-sensitive application data are mapped or demapped or a time sensitive function is performed. This standard leverages the work of the IEEE 1588 WG by developing the additional specifications needed to address these requirements.

1.3 Introduction

1.3.1 Background

Ethernet has successfully propagated from the data center to the home, becoming the wired home computer interconnect of choice. However, insufficient support of real-time services has limited Ethernet’s success as a consumer audio-video interconnects, where IEEE Std 1394 Serial Bus and Universal Serial Bus (USB) have dominated the marketplace. Success in this arena requires solutions to multiple topics:

a) Discovery. A controller discovers the proper devices and related streamID/bandwidth parameters to allow the listener to subscribe to the desired talker-sourced stream.

b) Subscription. The controller commands the listener to establish a path from the talker. Subscription may pass or fail, based on availability of routing-table and link-bandwidth resources.

c) Synchronization. The distributed clocks in talkers and listeners are accurately synchronized. Synchronized clocks avoid cycle slips and playback-phase distortions.

d) Pacing. The transmitted classA traffic is paced to avoid other classA traffic disruptions.

This draft covers the “Synchronization” component, assuming solutions for the other topics will be developed within other drafts or forums.

1.3.2 Interoperability

AVB time synchronization interoperates with existing Ethernet, but the scope of time-synchronization is limited to the AVB cloud, as illustrated in Figure 1.1; less-precise time-synchronization services are available everywhere else. The scope of the AVB cloud is limited by a non-AVB capable bridge or a half-duplex link, neither of which can support AVB services.

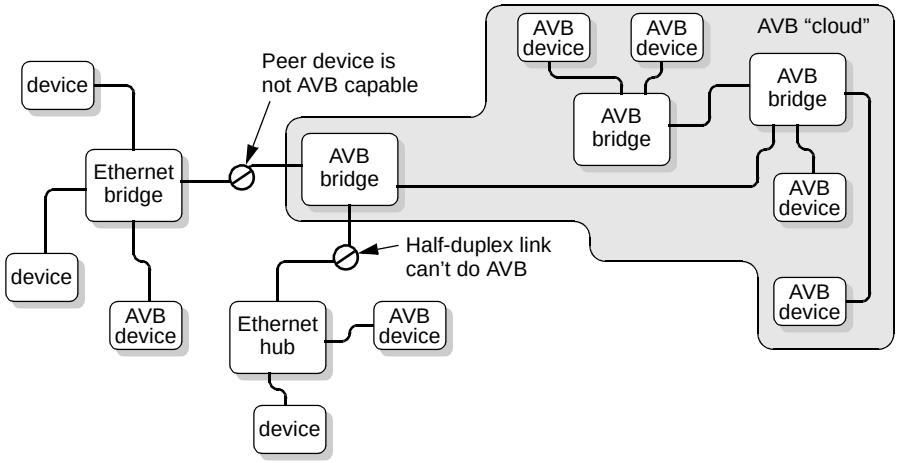

Figure 1.1—Topology and connectivity

Separation of AVB devices is driven by the requirements of AVB bridges to support subscription (bandwidth allocation) and pacing of time-sensitive transmissions, as well as time-of-day clock-synchronization.

1.3.3 Document structure

The clauses and annexes of this working paper are listed below.

- Clause 1: Overview
- Clause 2: References
- Clause 3: Terms, definitions, and notation
- Clause 4: Abbreviations and acronyms
- Clause 5: Architecture overview
- Clause 8: Ethernet full duplex (EFDX) state machines
- Annex A: Bibliography
- Annex D: Bridging to IEEE Std 1394
- Annex E: Time-of-day format considerations
- Annex F: C-code illustrations

2. References

The following documents contain provisions that, through reference in this working paper, constitute provisions of this working paper. All the standards listed are normative references. Informative references are given in Annex A. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this working paper are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

ANSI/ISO 9899-1990, Programming Language-C.[1,2]

IEEE Std 802.1D-2004, IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges.

[1]Replaces ANSI X3.159-1989

[2]ISO documents are available from ISO Central Secretariat, 1 Rue de Varembe, Case Postale 56, CH-1211, Geneve 20, Switzerland/Suisse; and from the Sales Department, American National Standards Institute, 11 West 42 Street, 13th Floor, New York, NY 10036-8002, USA

3. Terms, definitions, and notation

3.1 Conformance levels

Several key words are used to differentiate between different levels of requirements and options, as described in this subclause.

3.1.1 may: Indicates a course of action permissible within the limits of the standard with no implied preference (“may” means “is permitted to”).

3.1.2 shall: Indicates mandatory requirements to be strictly followed in order to conform to the standard and from which no deviation is permitted (“shall” means “is required to”).

3.1.3 should: An indication that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (“should” means “is recommended to”).

3.2 Terms and definitions

For the purposes of this working paper, the following terms and definitions apply. The Authoritative Dictionary of IEEE Standards Terms [B2] should be referenced for terms not defined in the clause.

3.2.1 bridge: A functional unit interconnecting two or more networks at the data link layer of the OSI reference model.

3.2.2 clock master: A bridge or end station that provides the link clock reference.

3.2.3 clock slave: A bridge or end station that tracks the link clock reference provided by the clock master.

3.2.4 cyclic redundancy check (CRC): A specific type of frame check sequence computed using a generator polynomial.

3.2.5 grand clock master: The clock master selected to provide the network time reference.

3.2.6 link: A unidirectional channel connecting adjacent stations (half of a span).

3.2.7 listener: A sink of a stream, such as a television or acoustic speaker.

3.2.8 local area network (LAN): A communications network designed for a small geographic area, typically not exceeding a few kilometers in extent, and characterized by moderate to high data transmission rates, low delay, and low bit error rates.

3.2.9 MAC client: The layer entity that invokes the MAC service interface.

3.2.10 medium (plural: **media):** The material on which information signals are carried; e.g., optical fiber, coaxial cable, and twisted-wire pairs.

3.2.11 medium access control (MAC) sublayer: The portion of the data link layer that controls and mediates the access to the network medium. In this working paper, the MAC sublayer comprises the MAC datapath sublayer and the MAC control sublayer.

3.2.12 network: A set of communicating stations and the media and equipment providing connectivity among the stations.

3.2.13 plug-and-play: The requirement that a station perform classA transfers without operator intervention (except for any intervention needed for connection to the cable).

3.2.14 protocol implementation conformance statement (PICS): A statement of which capabilities and options have been implemented for a given Open Systems Interconnection (OSI) protocol.

3.2.15 span: A bidirectional channel connecting adjacent stations (two links).

3.2.16 station: A device attached to a network for the purpose of transmitting and receiving information on that network.

3.2.17 topology: The arrangement of links and stations forming a network, together with information on station attributes.

3.2.18 transmit (transmission): The action of a station placing a frame on the medium.

3.2.19 unicast: The act of sending a frame addressed to a single station.

3.3 State machines

3.3.1 State machine behavior

The operation of a protocol can be described by subdividing the protocol into a number of interrelated functions. The operation of the functions can be described by state machines. Each state machine represents the domain of a function and consists of a group of connected, mutually exclusive states. Only one state of a function is active at any given time. A transition from one state to another is assumed to take place in zero time (i.e., no time period is associated with the execution of a state), based on some condition of the inputs to the state machine.

The state machines contain the authoritative statement of the functions they depict. When apparent conflicts between descriptive text and state machines arise, the order of precedence shall be formal state tables first, followed by the descriptive text, over any explanatory figures. This does not override, however, any explicit description in the text that has no parallel in the state tables.

The models presented by state machines are intended as the primary specifications of the functions to be provided. It is important to distinguish, however, between a model and a real implementation. The models are optimized for simplicity and clarity of presentation, while any realistic implementation might place heavier emphasis on efficiency and suitability to a particular implementation technology. It is the functional behavior of any unit that has to match the standard, not its internal structure. The internal details of the model are useful only to the extent that they specify the external behavior clearly and precisely.

3.3.2 State table notation

NOTE—The following state machine notation was used within 802.17, due to the exactness of C-code conditions and the simplicity of updating table entries (as opposed to 2-dimensional graphics).
Early state table descriptions can be converted (if necessary) into other formats before publication.

Each row of the table is preferably provided with a brief description of the condition and/or action for that row. The descriptions are placed after the table itself, and linked back to the rows of the table using numeric tags.

State machines may be represented in tabular form. The table is organized into two columns: a left hand side representing all of the possible states of the state machine and all of the possible conditions that cause transitions out of each state, and the right hand side giving all of the permissible next states of the state machine as well as all of the actions to be performed in the various states, as illustrated in Table 3.1. The syntax of the expressions follows standard C notation (see 3.12). No time period is associated with the transition from one state to the next.

Table 3.1—State table notation example

Current		Row	Next	
state	**condition**		**action**	**state**
START	sizeOfMacControl > spaceInQueue	1	—	START
	passM == 0	2		
	—	3	TransmitFromControlQueue();	FINAL
FINAL	SelectedTransferCompletes()	4	—	START
	—	5	—	FINAL

Row 3.1-1: Do nothing if the size of the queued MAC control frame is larger than the PTQ space.
Row 3.1-2: Do nothing in the absence of MAC control transmission credits.
Row 3.1-3: Otherwise, transmit a MAC control frame.

Row 3.1-4: When the transmission completes, start over from the initial state (i.e., START).
Row 3.1-5: Until the transmission completes, remain in this state.

Each combination of current state, next state, and transition condition linking the two is assigned to a different row of the table. Each row of the table, read left to right, provides: the name of the current state; a condition causing a transition out of the current state; an action to perform (if the condition is satisfied); and, finally, the next state to which the state machine transitions, but only if the condition is satisfied. The symbol "—" signifies the default condition (i.e., operative when no other condition is active) when placed in the condition column, and signifies that no action is to be performed when placed in the action column. Conditions are evaluated in order, top to bottom, and the first condition that evaluates to a result of TRUE is used to determine the transition to the next state. If no condition evaluates to a result of TRUE, then the state machine remains in the current state. The starting or initialization state of a state machine is always labeled "START" in the table (though it need not be the first state in the table). Every state table has such a labeled state.

Each row of the table is preferably provided with a brief description of the condition and/or action for that row. The descriptions are placed after the table itself, and linked back to the rows of the table using numeric tags.

3.4 Arithmetic and logical operators

In addition to commonly accepted notation for mathematical operators, Table 3.2 summarizes the symbols used to represent arithmetic and logical (boolean) operations. Note that the syntax of operators follows standard C notation (see 3.12).

Table 3.2—Special symbols and operators

Printed character	Meaning
&&	Boolean AND
||	Boolean OR
!	Boolean NOT (negation)
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
=	Assignment operator
//	Comment delimiter

3.5 Numerical representation

NOTE—The following notation was taken from 802.17, where it was found to have benefits:
– The subscript notation is consistent with common mathematical/logic equations.
– The subscript notation can be used consistently for all possible radix values.

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

Decimal numbers are represented in their usual 0, 1, 2, … format. Hexadecimal numbers are represented by a string of one or more hexadecimal (0-9,A-F) digits followed by the subscript 16, except in C-code contexts, where they are written as `0x123EF2` etc. Binary numbers are represented by a string of one or more binary (0,1) digits, followed by the subscript 2. Thus the decimal number “26” may also be represented as “$1A_{16}$” or “11010_2”.

MAC addresses and OUI/EUI values are represented as strings of 8-bit hexadecimal numbers separated by hyphens and without a subscript, as for example “01-80-C2-00-00-15” or “AA-55-11”.

3.6 Field notations

3.6.1 Use of italics

All field names or variable names (such as *level* or *myMacAddress*), and sub-fields within variables (such as *thisState.level*) are italicized within text, figures and tables, to avoid confusion between such names and similarly spelled words without special meanings. A variable or field name that is used in a subclause heading or a figure or table caption is also italicized. Variable or field names are not italicized within C code, however, since their special meaning is implied by their context. Names used as nouns (e.g., subclassA0) are also not italicized.

3.6.2 Field conventions

This working paper describes fields within packets or included in state-machine state. To avoid confusion with English names, such fields have an italics font, as illustrated in Table 3.3.

Table 3.3—Names of fields and sub-fields

Name	Description
newCRC	Field within a register or frame
thisState.level	Sub-field within field *thisState*
thatState.rateC[*n*].*c*	Sub-field within array element *rateC[n]*

Run-together names (e.g., *thisState*) are used for fields because of their compactness when compared to equivalent underscore-separated names (e.g., *this_state*). The use of multiword names with spaces (e.g., “This State”) is avoided, to avoid confusion between commonly used capitalized key words and the capitalized word used at the start of each sentence.

A sub-field of a field is referenced by suffixing the field name with the sub-field name, separated by a period. For example, *thisState.level* refers to the sub-field *level* of the field *thisState*. This notation can be continued in order to represent sub-fields of sub-fields (e.g., *thisState.level.next* is interpreted to mean the sub-field *next* of the sub-field *level* of the field *thisState*).

Two special field names are defined for use throughout this working paper. The name *frame* is used to denote the data structure comprising the complete MAC sublayer PDU. Any valid element of the MAC sublayer PDU, can be referenced using the notation *frame.xx* (where *xx* denotes the specific element); thus, for instance, *frame.serviceDataUnit* is used to indicate the *serviceDataUnit* element of a frame.

Unless specifically specified otherwise, reserved fields are reserved for the purpose of allowing extended features to be defined in future revisions of this working paper. For devices conforming to this version of this working paper, nonzero reserved fields are not generated; values within reserved fields (whether zero or nonzero) are to be ignored.

3.6.3 Field value conventions

This working paper describes values of fields. For clarity, names can be associated with each of these defined values, as illustrated in Table 3.4. A symbolic name, consisting of upper case letters with underscore separators, allows other portions of this working paper to reference the value by its symbolic name, rather than a numerical value.

Table 3.4—*wrap* field values

Value	Name	Description
0	STANDARD	Standard processing selected
1	SPECIAL	Special processing selected
2,3	—	Reserved

Unless otherwise specified, reserved values allow extended features to be defined in future revisions of this working paper. Devices conforming to this version of this working paper do not generate nonzero reserved values, and process reserved fields as though their values were zero.

A field value of TRUE shall always be interpreted as being equivalent to a numeric value of 1 (one), unless otherwise indicated. A field value of FALSE shall always be interpreted as being equivalent to a numeric value of 0 (zero), unless otherwise indicated.

3.7 Bit numbering and ordering

Data transfer sequences normally involve one or more cycles, where the number of bytes transmitted in each cycle depends on the number of byte lanes within the interconnecting link. Data byte sequences are shown in figures using the conventions illustrated by Figure 3.1, which represents a link with four byte lanes. For multi-byte objects, the first (left-most) data byte is the most significant, and the last (right-most) data byte is the least significant.

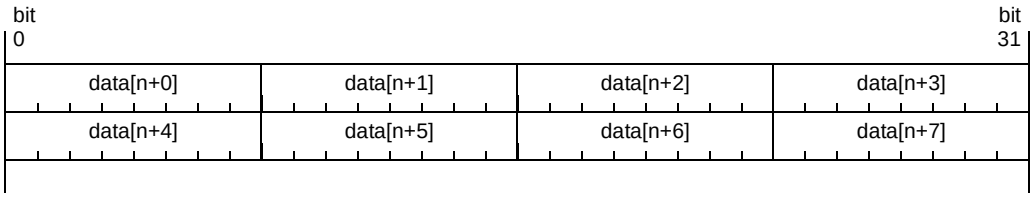

Figure 3.1—Bit numbering and ordering

Figures are drawn such that the counting order of data bytes is from left to right within each cycle, and from top to bottom between cycles. For consistency, bits and bytes are numbered in the same fashion.

NOTE—The transmission ordering of data bits and data bytes is not necessarily the same as their counting order; the translation between the counting order and the transmission order is specified by the appropriate reconciliation sublayer.

3.8 Byte sequential formats

Figure 3.2 provides an illustrative example of the conventions to be used for drawing frame formats and other byte sequential representations. These representations are drawn as fields (of arbitrary size) ordered along a vertical axis, with numbers along the left sides of the fields indicating the field sizes in bytes. Fields are drawn contiguously such that the transmission order across fields is from top to bottom. The example shows that *field1*, *field2*, and *field3* are 1-, 1- and 6-byte fields, respectively, transmitted in order starting with the *field1* field first. As illustrated on the right hand side of Figure 3.2, a multi-byte field represents a sequence of ordered bytes, where the first through last bytes correspond to the most significant through least significant portions of the multi-byte field, and the MSB of each byte is drawn to be on the left hand side.

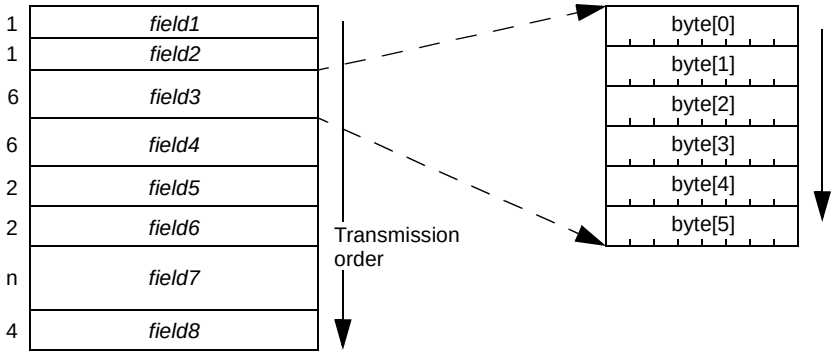

Figure 3.2—Byte sequential field format illustrations

NOTE—Only the left-hand diagram in Figure 3.2 is required for representation of byte-sequential formats. The right-hand diagram is provided in this description for explanatory purposes only, for illustrating how a multi-byte field within a byte sequential representation is expected to be ordered. The tag “Transmission order” and the associated arrows are not required to be replicated in the figures.

3.9 Ordering of multibyte fields

In many cases, bit fields within byte or multibyte objects are expanded in a horizontal fashion, as illustrated in the right side of Figure 3.3. The fields within these objects are illustrated as follows: left-to-right is the byte transmission order; the left-through-right bits are the most significant through least significant bits respectively.

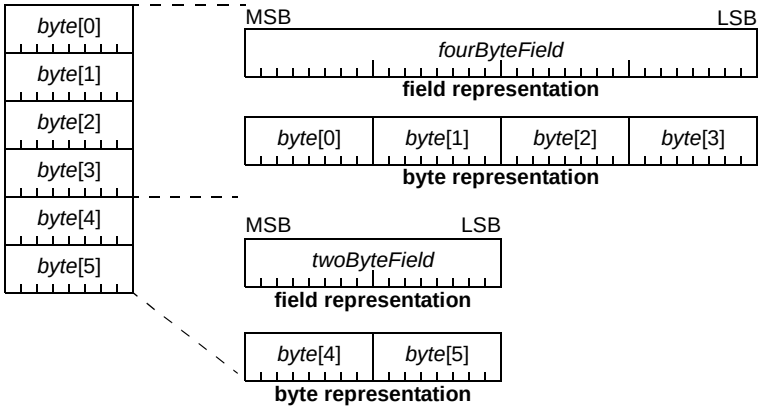

Figure 3.3—Multibyte field illustrations

The first *fourByteField* can be illustrated as a single entity or a 4-byte multibyte entity. Similarly, the second *twoByteField* can be illustrated as a single entity or a 2-byte multibyte entity.

NOTE—The following text was taken from 802.17, where it was found to have benefits:
The details should, however, be revised to illustrate fields within an AVB frame header *serviceDataUnit*.

To minimize potential for confusion, four equivalent methods for illustrating frame contents are illustrated in Figure 3.4. Binary, hex, and decimal values are always shown with a left-to-right significance order, regardless of their bit-transmission order.

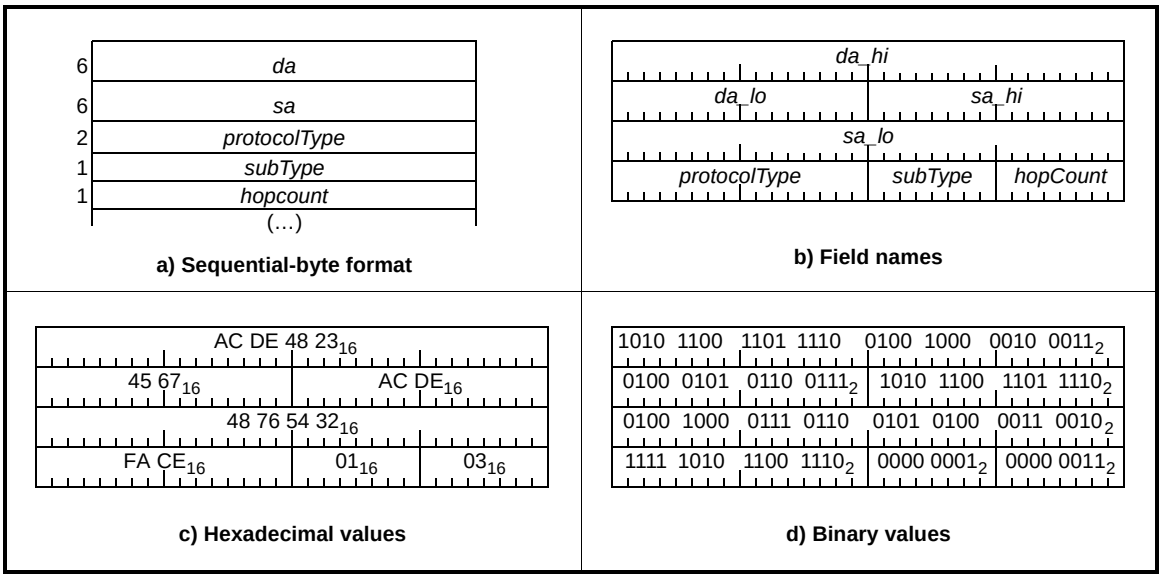

Figure 3.4—Illustration of fairness-frame structure

3.10 MAC address formats

The format of MAC address fields within frames is illustrated in Figure 3.5.

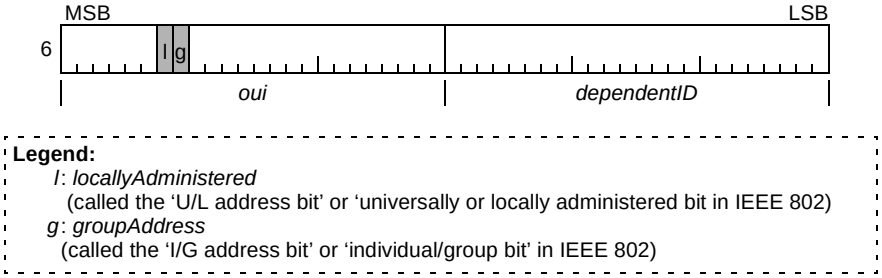

Figure 3.5—MAC address format

3.10.1 *oui*: A 24-bit organizationally unique identifier (OUI) field supplied by the IEEE/RAC for the purpose of identifying the organization supplying the (unique within the organization, for this specific context) 24-bit *dependentID*. (For clarity, the *locallyAdministered* and *groupAddress* bits are illustrated by the shaded bit locations.)

3.10.2 *dependentID*: An 24-bit field supplied by the *oui*-specified organization. The concatenation of the *oui* and *dependentID* provide a unique (within this context) identifier.

To reduce the likelihood of error, the mapping of OUI values to the *oui/dependentID* fields are illustrated in Figure 3.6. For the purposes of illustration, specific OUI and *dependentID* example values have been assumed. The two shaded bits correspond to the *locallyAdministered* and *groupAddress* bit positions illustrated in Figure 3.5.

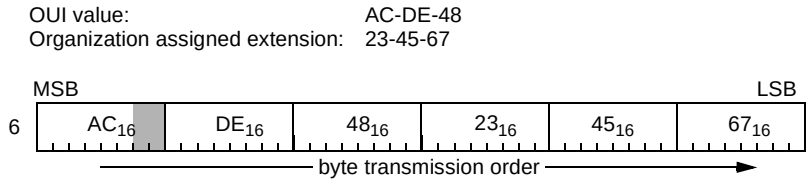

Figure 3.6—48-bit MAC address format

3.11 Informative notes

Informative notes are used in this working paper to provide guidance to implementers and also to supply useful background material. Such notes never contain normative information, and implementers are not required to adhere to any of their provisions. An example of such a note follows.

NOTE—This is an example of an informative note.

3.12 Conventions for C code used in state machines

Many of the state machines contained in this working paper utilize C code functions, operators, expressions and structures for the description of their functionality. Conventions for such C code can be found in Annex F.

4. Abbreviations and acronyms

This working paper contains the following abbreviations and acronyms:

AP	access point
AV	audio/video
AVB	audio/video bridging
AVB network	audio/video bridged network
BER	bit error ratio
BMC	best master clock
BMCA	best master clock algorithm
CRC	cyclic redundancy check
EFDX	Ethernet full duplex
EPON	Ethernet passive optical network
FIFO	first in first out
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
ITU	International Telecommunication Union
LAN	local area network
LSB	least significant bit
MAC	medium access control
MAN	metropolitan area network
MSB	most significant bit
OSI	open systems interconnect
PDU	protocol data unit
PHY	physical layer
PLL	phase-locked loop
PTP	Precision Time Protocol
R11V	radio 802.11v
RFC	request for comment
RPR	resilient packet ring
VOIP	voice over internet protocol

5. Architecture overview

5.1 Application scenarios

5.1.1 Garage jam session

As an illustrative example, consider AVB usage for a garage jam session, as illustrated in Figure 5.1. The audio inputs (microphone and guitar) are converted, passed through a guitar effects processor, two bridges, mixed within an audio console, return through two bridges, and return to the ear through headphones.

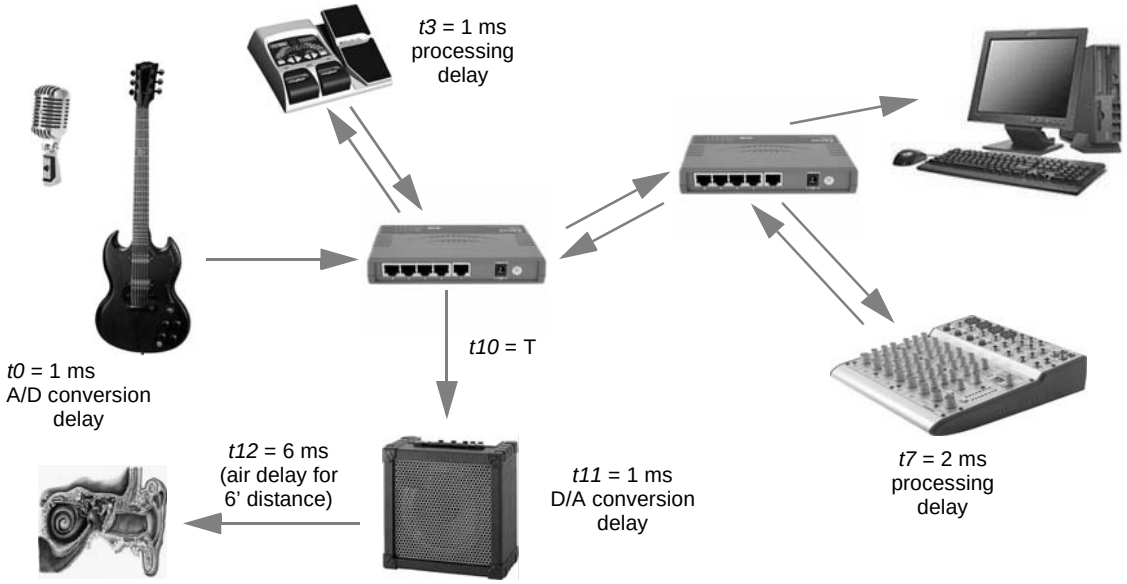

Figure 5.1—Garage jam session

Using Ethernet within such systems has multiple challenges: low-latency and tight time-synchronization. Tight time synchronization is necessary to avoid cycle slips when passing through multiple processing components and (ultimately) to avoid under-run/over-run at the final D/A converter's FIFO. The challenge of low-latency transfers is being addressed in other forums and is outside the scope of this draft.

5.1.2 Looping topologies

Bridged Ethernet networks currently have no loops, but bridging extensions are contemplating looping topologies. To ensure longevity of this standard, the time-synchronization protocols are tolerant of looping topologies that could occur (for example) if the dotted-line link were to be connected in Figure 5.2.

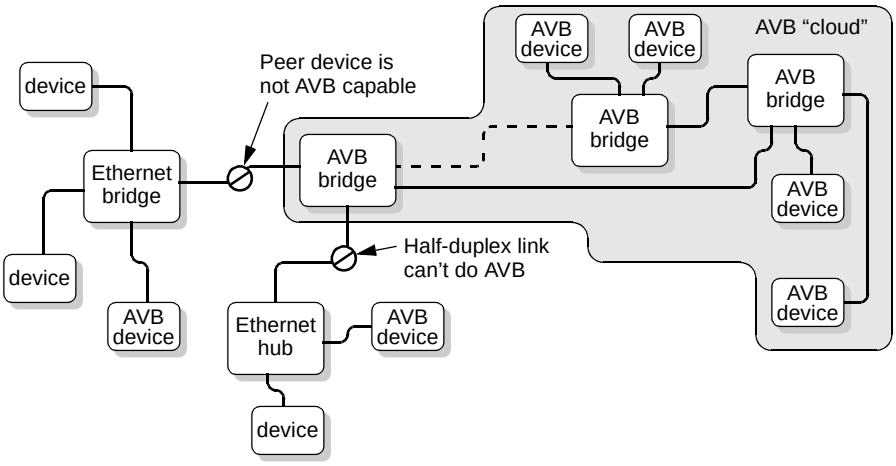

Figure 5.2—Possible looping topology

Separation of AVB devices is driven by the requirements of AVB bridges to support subscription (bandwidth allocation) and pacing of time-sensitive transmissions, as well as time-of-day clock-synchronization.

5.2 Design methodology

5.2.1 Assumptions

This working paper specifies a protocol to synchronize independent timers running on separate stations of a distributed networked system, based on concepts specified within IEEE Std 1588-2002. Although a high degree of accuracy and precision is specified, the technology is applicable to low-cost consumer devices. The protocols are based on the following design assumptions:

a) Each end station and intermediate bridges provide independent clocks.
b) All clocks are accurate, typically to within ±100PPM.
c) Details of the best time-synchronization protocols are physical-layer dependent.

5.2.2 Objectives

With these assumptions in mind, the time synchronization objectives include the following:

a) Precise. Multiple timers can be synchronized to within 10's of nanoseconds.
b) Inexpensive. For consumer AVB devices, the costs of synchronized timers are minimal. (GPS, atomic clocks, or 1PPM clock accuracies would be inconsistent with this criteria.)
c) Scalable. The protocol is independent of the networking technology. In particular:
    1) Cyclical physical topologies are supported.
    2) Long distance links (up to 2 kM) are allowed.
d) Plug-and-play. The system topology is self-configuring; no system administrator is required.

5.2.3 Strategies

Strategies used to meet these objectives include the following:

a) Precision is achieved by calibrating and adjusting *grandTime* clocks.
   1) Offsets. Offset value adjustments eliminate immediate clock-value errors.
   2) Rates. Rate value adjustments reduce long-term clock-drift errors.

b) Simplicity is achieved by the following:
   1) Concurrence. Most configuration and adjustment operations are performed concurrently.
   2) Feed-forward. PLLs are unnecessary within bridges, but possible within applications.
   3) Frequent. Frequent (nominally 100 Hz) interchanges reduces needs for overly precise clocks.

5.3 Grand-master selection

5.3.1 Grand-master overview

Clock synchronization involves streaming of timing information from a grand-master timer to one or more slave timers. Although primarily intended for non-cyclical physical topologies (see Figure 5.3a), the synchronization protocols also function correctly on cyclical physical topologies (see Figure 5.3b), by activating only a non-cyclical subset of the physical topology.

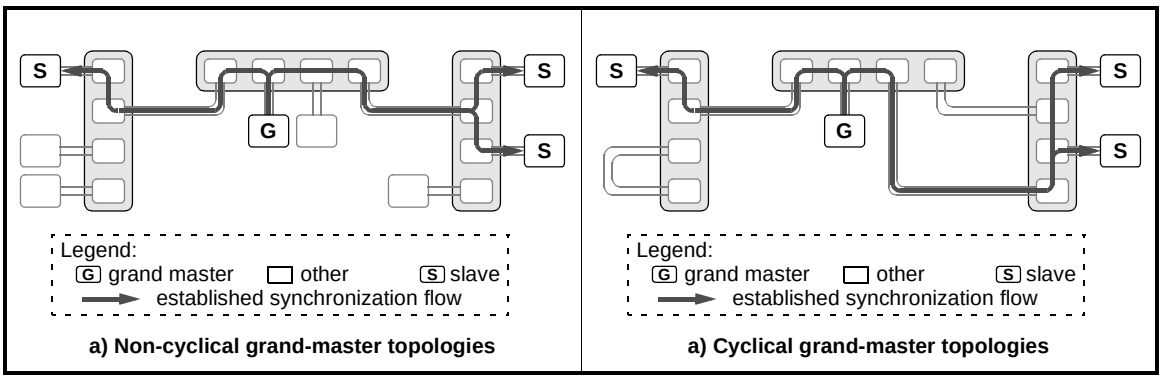

Figure 5.3—Timing information flows

In concept, the clock-synchronization protocol starts with the selection of the reference-timer station, called a grand-master station (oftentimes abbreviated as grand-master). Every AVB-capable station is grand-master capable, but only one is selected to become the grand-master station within each network. To assist in the grand-master selection, each station is associated with a distinct preference value; the grand-master is the station with the “best” preference values. Thus, time-synchronization services involve two subservices, as listed below and described in the following subclauses.

a) Selection. Looping topologies are isolated (from a time-synchronization perspective) into a spanning tree. The root of the tree, which provides the time reference to others, is the grand master.

b) Distribution. Synchronized time is distributed through the grand-master’s spanning tree.

5.3.2 Grand-master selection

As part of the grand-master selection process, stations forward the best of their observed preference values to neighbor stations, allowing the overall best-preference value to be ultimately selected and known by all. The station whose preference value matches the overall best-preference value ultimately becomes the grand-master.

The grand-master station observes that its precedence is better than values received from its neighbors, as illustrated in Figure 5.4a. A slave stations observes its precedence to be worse than one of its neighbors and forwards the best-neighbor precedence value to adjacent stations, as illustrated in Figure 5.4b. To avoid cyclical behaviors, a *hopCount* value is associated with preference values and is incremented before the best-precedence value is communicated to others.

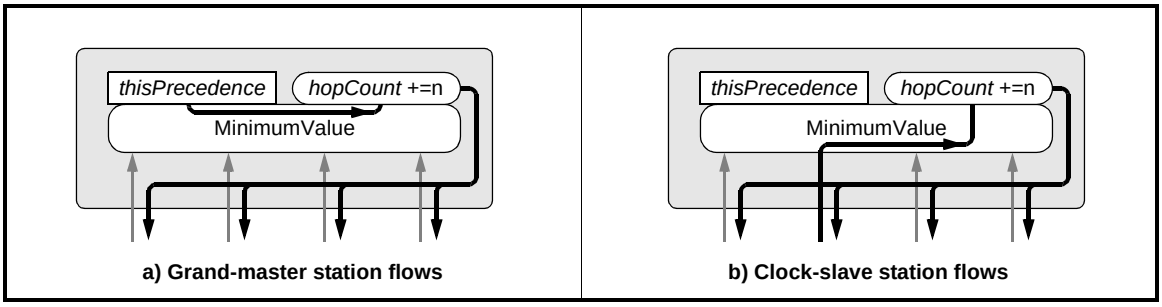

Figure 5.4—Grand-master precedence flows

When stabilized, the value of *n* equals one and the *hopCount* value reflects the distance between this station and its grand master, in units of hops-between-bridges. Other values are used to quickly stabilize systems with rogue frames, as summarized in Equation 5.1.

```
#define HOPS 255                                                              (5.1)
n = (frame.hopCount > hopCount) ? (HOPS - frame.hopCount) / 2 : 1;
```

NOTE—A rogue frame circulates at a high precedence, in a looping manner, where the source stations is no longer present (or no longer active) and therefore cannot remove the circulating frame. The super-linear increase in *n* is intended to quickly scrub rogue frames, when the circulation loop consists of less than HOPS stations.

5.3.3 Grand-master preference

Grand-master preference is based on the concatenation of multiple fields, as illustrated in Figure 5.5. The *port* value is used within bridges, but is not transmitted between stations.

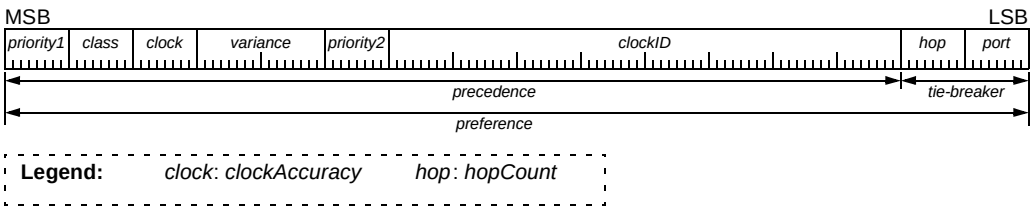

Figure 5.5—Grand-master preference

This format is similar to the format of the spanning-tree precedence value, but a wider *clockID* is provided for compatibility with interconnects based on 64-bit station identifiers.

5.4 Synchronized-time distribution

5.4.1 Hierarchical grand masters

Clock-synchronization information conceptually flows from a grand-master station to clock-slave stations, as illustrated in Figure 5.6a. A more detailed illustration shows pairs of synchronized clock-master and clock-slave components, as illustrated in Figure 5.6b. The active clock agents are illustrated as black-and-white components; the passive clock agents are illustrated as grey-and-white components.

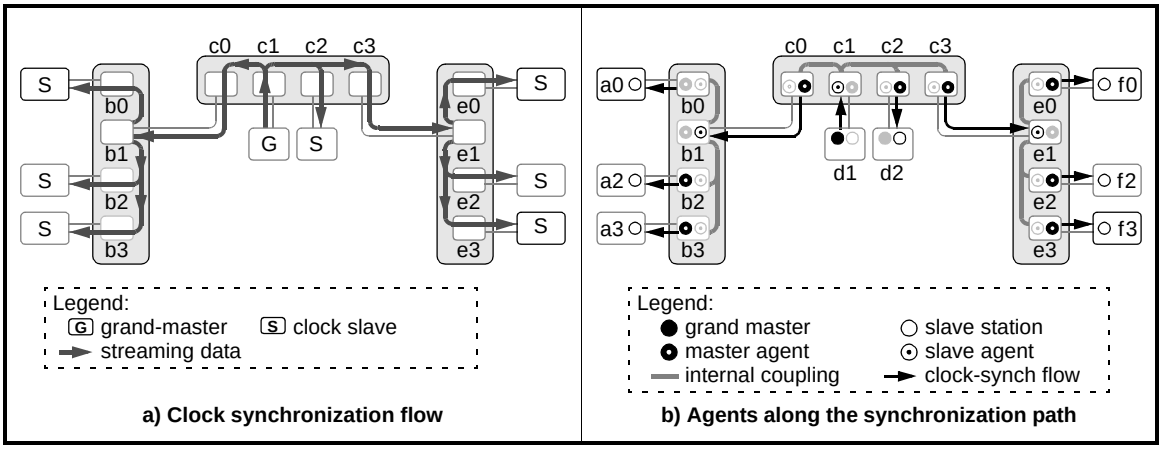

Figure 5.6—Hierarchical flows

Internal communications distribute synchronized time from clock-slave agents b1, c1, and e1 to the other clock-master agents on bridgeB, bridgeC, and bridgeE respectively. Within a clock-slave, precise time synchronization involves adjustments of timer value and rate-of-change values.

Time synchronization yields distributed but closely-matched *grandTime* values within stations and bridges. No attempt is made to eliminate intermediate jitter with bridge-resident jitter-reducing phase-lock loops (PLLs) but application-level phase locked loops (not illustrated) are expected to filter high-frequency jitter from the supplied *grandTime* values.

5.4.2 Time-synchronization flows

Time-reference information is created at a ClockSource entity, flows through multiple intermediate entities, and is consumed at one or more ClockSink entities, as illustrated in Figure 5.7. Within this illustration, the clock-master station (containing the ClockSource entity) and the clock-slave station (containing the ClockSink entity) are illustrated as multipurpose bridges. Either of the ClockMaster and ClockSlave stations could also be end stations (not illustrated), wherein no MAC-relay functionality is required.

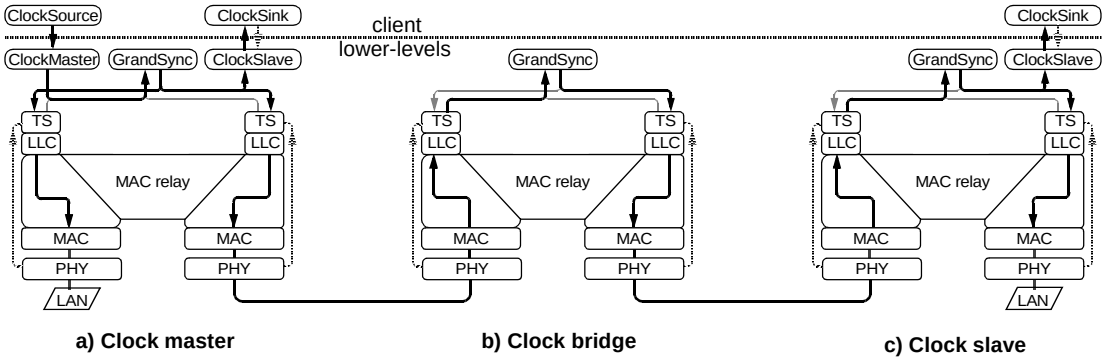

Figure 5.7—Time-synchronization flows

Entities within the intermediate bridge (see Figure 5.7b) are responsible for performing three distinct (and largely decoupled) functions, as illustrated in Figure 5.8. A clock-slave port (see Figure 5.8a) is responsible for compensating for time-reference transmission delays between this station and its neighbor. The GrandSync entity (see Figure 5.8b) is responsible for selecting the timeSync PDUs from the grand-master station; only thus selected PDUs are forwarded to transmitter ports.

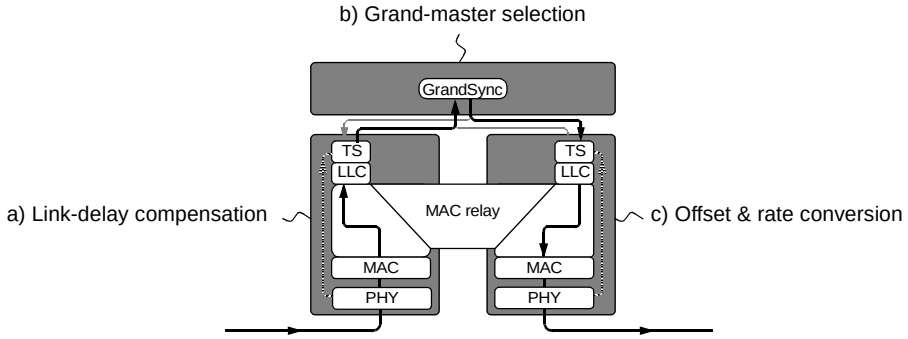

Figure 5.8—Intermediate-bridge responsibilities

The clock-master port (see Figure 5.8c) is responsible for revising the GrandSync-supplied timeSync PDUs to supply the appropriate media-dependent service-interface parameters and/or frames. Since the transmission times and rates may differ from those on the clock-slave port, the clock-master port is responsible for interpolating/extrapolating between previously received time samples to generate parameters corresponding to the recently observed transmit-snapshot.

5.4.2.1 Clock-master flows

Referring now to the clock-master (Figure 5.7-a) station. This clock-master station comprises client-level ClockSink as well as ClockSource entities. The ClockSink entity is provided so that the client-clock can be synchronized to the network clock, whenever another station is selected to become that grand-master. (The ClockSource entity on the grand-master station provides the network-synchronized time reference.)

The ClockSource time-reference interfaces indirectly to the GrandSync entity via a ClockMaster entity. The ClockMaster entity supplements the clock-synchronization provided by the ClockSource entity with additional information (such as the grand-master precedence) that is needed by the GrandSync entity.

The GrandSync entity is responsible for selecting the preferred time-reference port from among the possible direct-attached ClockSource and bus-bridge-port entities. The selection is based on user-preference, clock-property, topology, and unique-clock-identifier information.

The GrandSync entity echoes the time-synchronization information from (what it determines to be) the preferred port. Information from lower-preference ports is continuously monitored to detect preference changes (typically due to attach or detach of clock-master capable stations). In the absence of such changes, time-reference information in PDUs from lower-preference ports is ignored.

The GrandSync entity's echoed time-reference information is observed by the directly-attached ClockSlave and bridge-port entities. The information forms the basis for the time-synchronization information forwarded to other indirectly-attached ClockSlave entities through this station's bus-bridge ports

5.4.2.2 Bus-bridge flows

Referring now to the bus-bridge (Figure 5.7-b) station. This bus-bridge station comprises port and GrandSync entities. Both ports are responsible for forwarding their received time-reference information to the GransSync entity.

The bus bridge's GrandSync entity is responsible for selecting the preferred time-reference port. The selection is based on user-preference, clock-property, topology, and unique-clock-identifier information provided indirectly by remote ClockSource entities.

The GrandSync entity echoes the time-synchronization information from (what it determines to be) the preferred port. Information from lower-preference ports is continuously monitored to detect preference changes (typically due to attach or detach of clock-master capable stations). In the absence of such changes, time-reference information in PDUs from lower-preference ports is ignored.

The GrandSync entity's echoed time-reference information is observed by all bridge-port entities (including the source port). The information forms the basis for the time-synchronization information forwarded to other indirectly-attached ClockSlave entities through this station's bus-bridge ports.

5.4.2.3 Clock-slave flows

Referring now to the clock-slave (Figure 5.7-c) station. This clock-slave station comprises port, GrandSync, and ClockSlave entities, as well as a client-level ClockSink entity. All ports are responsible for forwarding their received time-reference information to the GransSync entity.

As always, the GrandSync entity is responsible for selecting the preferred time-reference port from among the possible direct-attached ClockSource and bus-bridge-port entities. The GrandSync entity echoes the time-synchronization information from (what it determines to be) the preferred port.

The GrandSync entity's echoed time-reference information is observed by the station-local ClockSlave entity. The ClockSlave entity removes the extraneous grand-master preference information and re-times its transmissions to match the client's time-request rate. The time-reference information is then passed to the ClockSink client.

5.4.2.4 Time-stamp flows

Referring now to the hashed PHY-to-TS lines within Figure 5.7 stations. Maintaining an accurate time reference relies on the presence of accurate time-stamp hardware capabilities in or near the media-dependent PHY. A bypass path is thus required at the receiver, so that the time-stamp can be affiliated with the arriving timeSync information, before the SDU or service-interface parameters are processed by the time-synchronization (TS) entity above the MAC.

A similar bypass path is also required at the transmitter, so that the time-stamp of a transmitted frame can become known to the time-synchronization (TS) entity above the MAC. For simplicity and convenience, this time-stamp information is not placed into the transmitted frame, but (via processing by the time-synchronization entity) can be placed within later transmissions.

5.5 Cascaded clock topologies

5.5.1 Cascaded clocking limitations

The naive approach towards forwarding time-synchronization information is to quickly propagate time-reference snapshots through successive stations. Unfortunately, relatively small (¼ interval) worst-case residence-time delays in each station can cause significant bunching on relevant topologies, as illustrated in Figure 5.9.

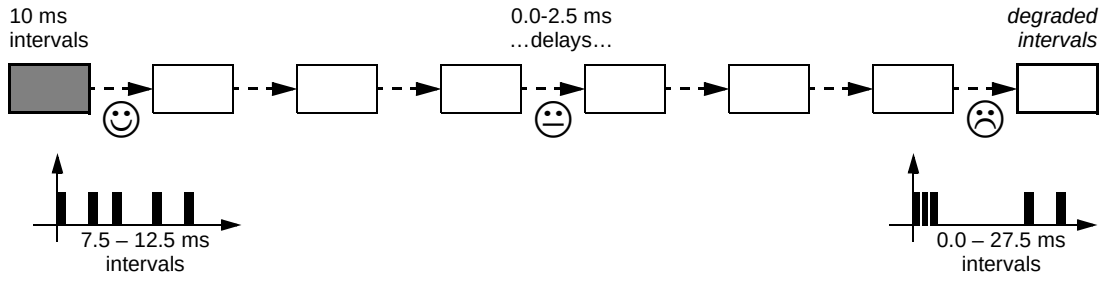

Figure 5.9—Cumulative sync-interval bunching

Techniques for avoiding such bunching are well known and practiced in the form of reclocked synchronous circuits. For example, Ethernet stations accept (baud-rate) information at a closely matched input clock rate, reclock the data with a local clock reference, and then forward the reclocked information without degrading data-jitter performance.

Applying these techniques to clock-sync transmission is straightforward. Rather than quickly forwarding these frames, their information is saved. That saved information is then forwarded in the same periodic fashion, based on local-station timing, as illustrated in Figure 5.10. While such reclocked systems more susceptible to gain-peaking/whiplash effects, their inherent design and verification simplicities favor their use.

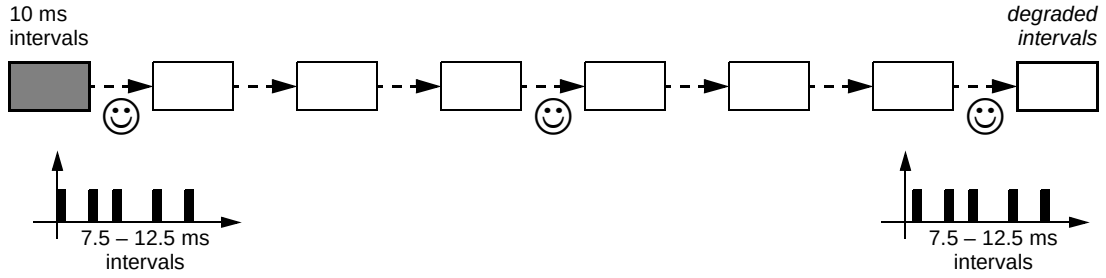

Figure 5.10—Cumulative sync-interval bunching

5.5.2 Mixed sync-interval systems

The reclocked sync-interval strategy is compatible with bridged mixed-media systems. The persistent or transient sync-interval rate of an intermediate (perhaps longer or more power sensitive) link could be less than the rate assumed for the clock-master, as illustrated in the center of Figure 5.11. Similarly, wireless links could base their timing events on triggers initiated by the clock-slave station, as illustrated in the right side of Figure 5.11.

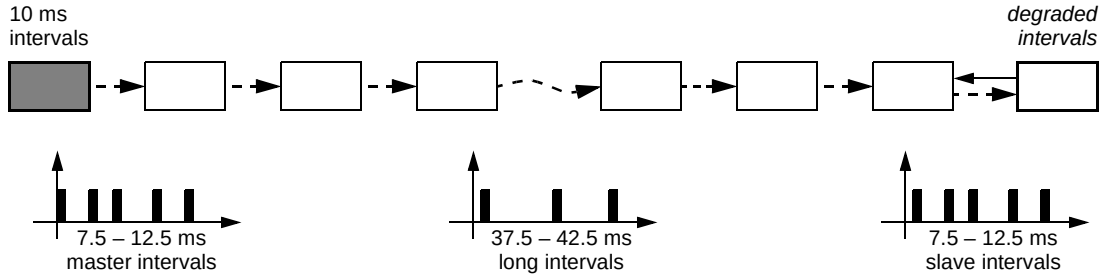

Figure 5.11—Mixed sync-interval systems

Other flow-through clocking designs would require special “boundary clock” architectures to support such mixed systems. With the interval retiming strategy, the additional (specification and implementation) complexities of such boundary-clock architectures are easily avoided.

5.6 Time-affiliation adjustments

5.6.1 Distinct receive/transmit adjustments

Distinct forms of time-delay adjustments occur at the receive (clock-slave) and transmit (clock-master) ports, as illustrated in Figure 5.12. The details of these time-delay adjustments are media-dependent, but the high-level concepts are the same.

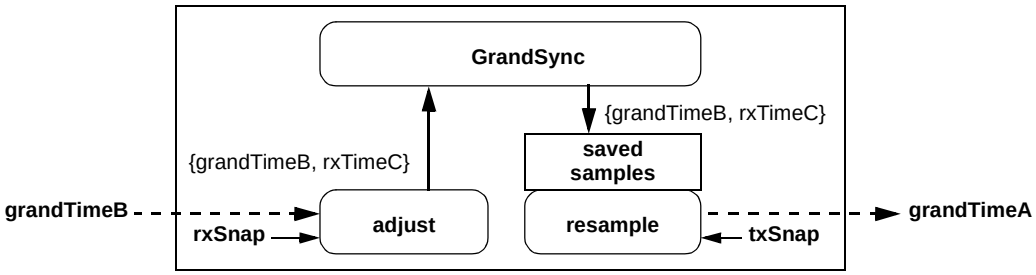

Figure 5.12—Receive/transmit adjustments

When a frame with the clock-master sourced *grandTimeB* is received, a snapshot of the station-local time is taken; that snapshot is called *rxSnap*. A more accurate *rxTimeC* snap-shot value is formed by compensating by the relatively-constant precomputed *linkDelay* value, as follows:

rxTimeC = rxSnap – linkDelay

The delay-compensated {*grandTimeB*, *rxTimeC*} affiliation parameters are passed to the GrandSync entity. That GrandSync entity ignores PDUs from lower-precedence stations, echoing only PDUs from the (perceived to be) grand-master station.

The echoed delay-compensated {*grandTimeB*, *rxTimeC*} affiliation parameters are saved in storage at each of the clock-master ports (this is an architectural model; implementations need not replicate physical storage). The forwarded *grandTimeA* value (which is renamed *grandTimeB* when received at the next station) is derived by interpolating between (or extrapolating from) previously saved samples.

5.7 Sampling offset/rate conversion

Each clock-master port is responsible for using its received {*grandTimeB*, *stationTimeB*} and converting them into the distinct {*grandTimeA*, *stationTimeA*} affiliations that are transmitted to its neighbor. Since the values of *stationTimeB* and *stationTimeA* are (by convention) coupled to the receive and transmit times, this update involves computation of *grandTimeA* values based on observed {*grandTimeA*, *stationTimeB*} values.

5.7.1 Forward interpolation inaccuracies

A typical design approach (and that used by IEEE Std 1588) views the received {*grandTime*, *stationTime*} affiliations as points on a curve, sampled at received-snapshot times *rx*[*n*]. The objective is to generate the distinct set of {*grandTime*, *tx*[*m*]} affiliations by extrapolating from a distinct set of receive-snapshot times *rx*[*n*], as illustrated in Figure 5.13.

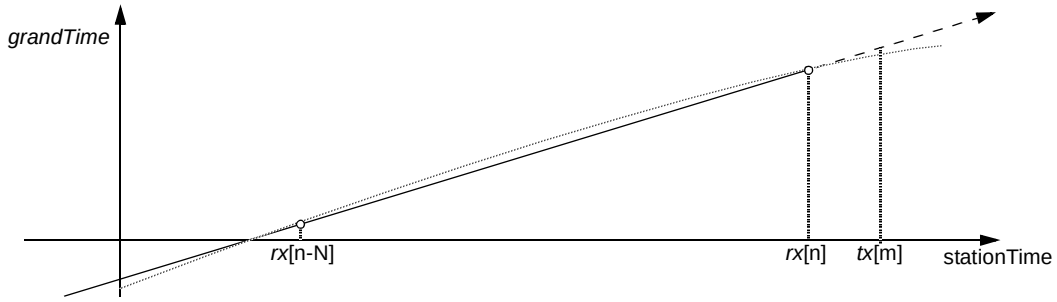

Figure 5.13—Extrapolation for *grandTime*

Extrapolation techniques exhibit gain peaking at frequencies whose wavelength is twice the {*rx*[*n*-N],*rx*[*n*]} slope-averaging interval, because the extrapolated value can exceed what would have been the sampled time value. A cascade of multiple stations emphasizes the gain-peaking inaccuracies, allowing errors to accumulate in an $O(N^2)$ fashion.

5.7.2 Forward interpolation inaccuracies

To reduce gain-peaking effects, the resampling computation can be migrated to a safe-interpolation domain. This involves subtracting a *backTime* constant from *tx*[*m*], yielding a new time *tb*[*m*], for which a less gain-peaking sensitive interpolation is viable, as illustrated in Figure 5.14. In concept, the stale (but {*grandTime*[*m*], *tb*[*m*]} affiliations could be passed to the terminal clock-slave stations, wherein a single extrapolation-to-the-future accumulation could be performed. A preferred technique is to compensate the interpolation result on an per-station basis as the time-reference flows towards the clock-slave station, as discussed in the following subclauses.

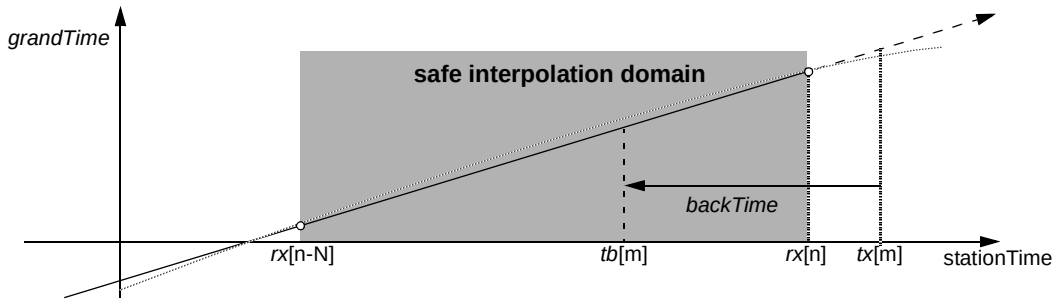

Figure 5.14—Extrapolation for *grandTime*

Extrapolation techniques exhibit gain peaking at frequencies whose wavelength is twice the {*rx*[*n*-N],*rx*[*n*]} slope-averaging interval, because the extrapolated value can exceed what would have been the sampled time value. A cascade of multiple stations emphasizes the gain-peaking inaccuracies, allowing errors to accumulate in an $O(N^2)$ fashion.

5.7.3 Backward interpolation

5.7.3.1 Interpolation of *grandTime*

A more-scalable backward-interpolation approach also views the received {*grandTimeB*, *stationTimeB*} affiliations as points on a curve, sampled at received-snapshot times *rx*[*n*]. However, the objective is to generate the distinct set of {*grandTimeA*, *tx*[*m*]} affiliations by interpolating within a distinct set of receive-snapshot affiliations {*grandTimeB*[*n*], *rx*[*n*]), as illustrated in Figure 5.15.

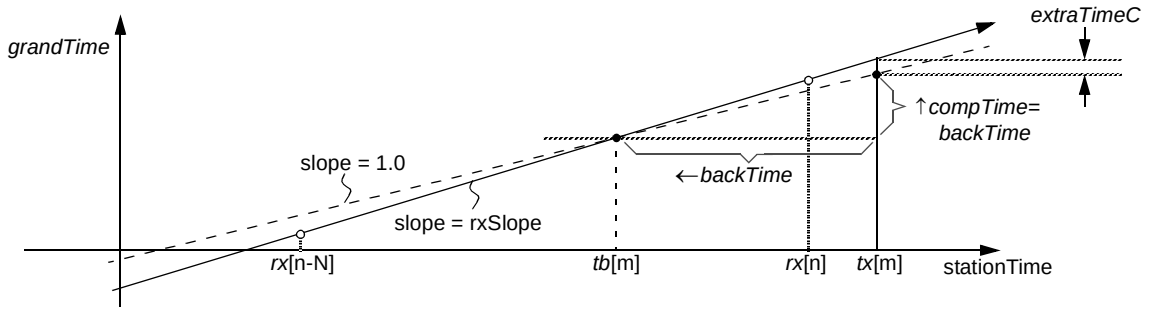

Figure 5.15—Interpolation for *grandTimeA*

grandTimeA[*m*] = *grandTimeB*[*m*] + *rxSlope* * ((*tx*[*m*] – *backTime*) – *rx*[*n*]) + *backTime*; (5.2)

grandTime0[*m*] is the value for the to-be-transmitted {*grandTime0*[*m*], *tx*[*m*]} affiliation.

backTime is a constant (sync-interval dependent) value.

rxSlope is the value of slope of previously sampled values, specified by Equation 5.3.

rxSlope = (*grandTimeB*[*n*] – *grandTimeB*[*n*-N]) / (*rx*[*n*] – *rx*[*n*-N]) (5.3)

Where:

grandTimeB[*n*] is the value from the previously received {*grandTimeB*[*n*], *rx*[*n*]} affiliation.

extraTimeC[*m*] = (*rxSlope* – ONE) * *backTime*; (5.4)

The advantage of this technique is the separation of *grandTime*[*m*] and *extra*[*m*] components. The interpolation process eliminates gain-peaking for the *grandTime*[*m*] value, thus reducing error effects when passing through multiple bridges. The sideband *extraTime* signal remains significant, and is therefore carried through bridges, so that the cumulative *grandTimed*[*m*]+*extraTime*[*m*] value can be passed to the end-point application.

From an intuitive perspective, the whiplash-free nature of the back-in-time interpolation is attributed to the use of interpolation (as opposed to extrapolation) protocols. Interpolation between input values never produces a larger output value, as would be implied by a gain-peaking (larger-than-unity gain) algorithm. A disadvantage of back-in-time interpolation is the requirement for a side-band *extraTime* communication channel, over which the difference between nominal and rate-normalized *backTime* values can be transmitted.

5.7.3.2 Averaging of *extraTime*

An averaging (rather than backward-interpolation) approach is applied to the received {*extraTimeB*, *stationTime*} affiliations as points on a curve, sampled at received-snapshot times *rx*[*n*]. The {*extraTimeD*, *tx*[*m*]} affiliations are produced by averaging recently observed *extraTimeB* values, as illustrated in Figure 5.16.

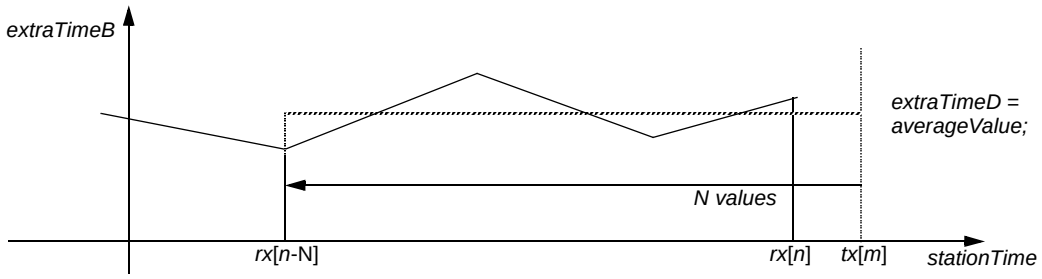

Figure 5.16—Interpolation of *extraTimeD*

$$extraTimeD[m] = (extraTime[n–N] + \ldots extraTime[n]) / N \quad (5.5)$$

$$extraTimeA[m] = extraTimeC[m] + extraTimeD[m]; \quad (5.6)$$

The to-be-transmitted value of *extraTimeA*[*m*] consists of a contribution *errorTimeC* (coming from this station's *grandTime* interpolation) and a contribution *extraTimeD* (accumulated from previous stations's *grandTime* interpolations). Note that the averaging of *extraB* values is effectively a low-pass filtering process that removes noise without causing a gain-peaking frequency response.

NOTE—For simplicity and scalability, the computed *extraTimeC* time is based on *n*, a fixed number of samples, where *n* is a convenient power-of-two in size.

5.8 Distinctions from IEEE Std 1588

Advantageous properties of this protocol that distinguish it from other protocols (including portions of IEEE Std 1588) include the following:

a) Synchronization between grand-master and local clocks occurs at each station:
   1) All bridges have a lightly filtered synchronized image of the grand-master time.
   2) End-point stations have a heavily filtered synchronized image of the grand-master time.

b) Time is uniformly represented as scaled integers, wherein 40-bits represent fractions-of-a-second.
   1) Grand-master time specifies seconds within a more-significant 40-bit field.
   2) Local time specifies seconds within a more-significant 8-bit field.

c) Locally media-dependent synchronized networks don't require extra time-snapshot hardware.

d) Error magnitudes are linear with hop distances; PLL-whiplash and $O(n^2)$ errors are avoided.

e) Multicast (one-to-many) services are not required; only nearest-neighbor addressing is assumed.

f) A relatively frequent 100 Hz (as compared to 1 Hz) update frequency is assumed:
   1) This rate can be readily implemented (in today's technology) for minimal cost.
   2) The more-frequent rate improves accuracy and reduces transient-recovery delays.
   3) The more-frequent rate reduces transient-recovery delays.

g) Only one frame type simplifies the protocols and reduces transient-recovery times. Specifically:
   1) Cable delay is computed at a fast rate, allowing clock-slave errors to be better averaged.
   2) Rogue frames are quickly scrubbed (2.6 seconds maximum, for 256 stations).
   3) Drift-induced errors are greatly reduced.

6. GrandSync operation

6.1 Overview

6.1.1 GrandSync behavior

This clause specifies the state machines that specify GrandSync-entity processing. The operations are described in an abstract way and do not imply any particular implementations or any exposed interfaces. There is not necessarily a one-to-one correspondence between the primitives and formal procedures and the interfaces in any particular implementation.

The GrandSync entity is responsible for monitoring time-sync PDUs via the TIME_SYNC.indication service primitive, selectively echoing a subset of these PDUs vai the TIME_SYNC.request service primitive, as follows:

a) When a preferred time-sync related TIME_SYNC.indication arrives:
   1) The grand-master preference and port-timeout parameters are saved.
   2) TIME_SYNC.indication parameters are echoed in TIME_SYNC.request parameters.
   3) The arrival time is recorded, for the purpose of monitoring port timeouts.

b) Arriving non-preferred TIME_SYNC.indications are discarded.
   The intent is to echo only PDUs from the currently selected grand-master port.

c) If the preferred-port timeout is exceeded, the preferred-port parameters are reset.
   The intent is to restart grand-master selection based on the remaining candidate ports.

6.1.2 GrandSync interface model

The time-synchronization service model assumes the presence of one or more time-synchronized AVB ports communicating with a MAC relay, as illustrated in Figure 6.1. All components are assumed to have access to a common free-running (not adjustable) *localTime* value.

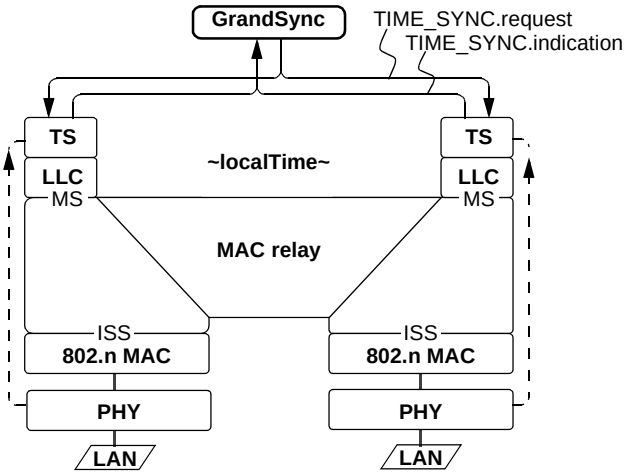

Figure 6.1—GrandSync interface model

A received MAC frame is associated with link-dependent timing information, processed within the TimeSync (TS) state machine, and passed to the GrandSync protocol entity. The GrandSync state machine (illustrated with a darker boundary) is responsible for saving time parameters from observed TIME_SYNC.indication parameters and generating TIME_SYNC.request parameters for delivery to other ports.

The preference of the time-sync PDUs determines whether the PDU content is ignored by the GrandSync protocol entity or modified and redistributed to the attached TS state machines. The sequencing of this state machine is specified by Table 6.1; details of the computations are specified by the C-code of Annex F.

Information exchanged with the GrandSync entity includes a source-*port* identifier, *hops*&*precedence* information for grand-master selection, a globally synchronized *grandTime*, a station-local *snapTime*, and a cumulative *extraTime*, as illustrated in Figure 6.2. A clock-slave end-point can filter the sum of *grandTime* and *extraTime* values, thereby yielding its image of the globally synchronized *grandTime* value.

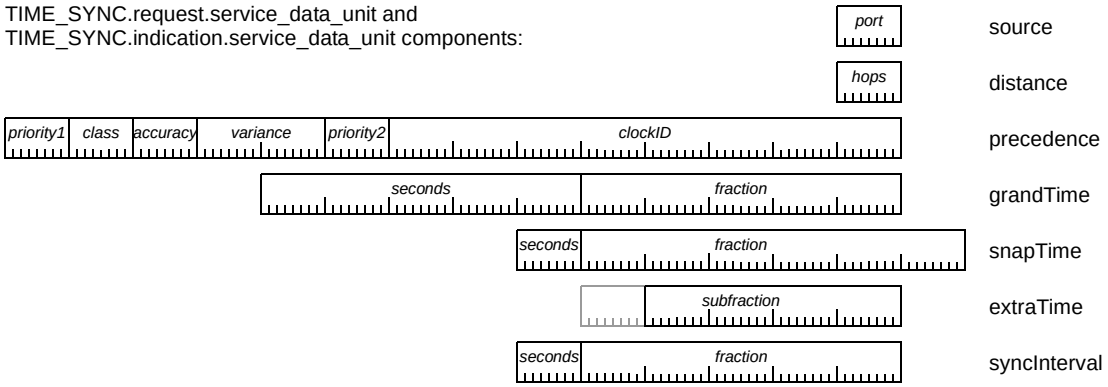

Figure 6.2—GrandSync service-interface components

NOTE—The *syncInterval* value is relative static and could (if desired) be communicated by access to port-specific resources. If this alternative configuration mechanism is preferred, this content will be removed from the service interface contents.

NOTE—The *snapTime* value has additional precision, when compared to the similar externally visible *localTime* value, to minimize the effects of numerical rounding when transferring values between computational entities within the bridge.

6.2 Service interface primitives

6.2.1 TIME_SYNC.indication

6.2.1.1 Function

Provides the GrandSync protocol entity with clock-synchronization parameters derived from PDUs sent from attached media-dependent ports. The information is sufficient to identify a single clock-slave port (typically the closest-to-grand-master port) and to disseminate grand-master supplied clock-synchronization information to other ports.

6.2.1.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
TIME_SYNC.indication {
    destination_address,      // Destination address
    source_address,           // Optional
    priority,                 // Forwarding priority
    service_data_unit,        // Delivered content
    {                         // Contents of the service_data_unit
        protocolType,         // Distinguishes AVB frames from others
        function,             // Distinguishes between timeSync and other AVB frames
        version,              // Distinguishes between timeSync frame versions
        precedence,           // Precedence for grand-master selection
        grandTime,            // Global-time snapshot (1-cycle delayed)
        extraTime,            // Accumulated grandTime error
        sourcePort,           // Identifies the source port
        hopCount,             // Distance from the grand-master station
        snapTime,             // Local-time snapshot (1-cycle delayed)
        syncInterval          // Nominal timeSync transmission interval
    }
}
```

NOTE—The *grandTime* field has a range of approximately 36,000 years, far exceeding expected equipment life-spans. The *localTime* and *linkTime* fields have a range of 256 seconds, far exceeding the expected timeSync frame transmission interval. These fields have a 1 pico-second resolution, more precise than the expected hardware snapshot capabilities. Future time-field extensions are therefore unlikely to be necessary in the future.

The parameters of the MA_DATA.indication are described as follows:

6.2.1.2.1 destination_address: A 48-bit field that allows the frame to be conveniently stripped by its downstream neighbor. The destination_address field contains an otherwise-reserved group 48-bit MAC address (TBD).

6.2.1.2.2 source_address: A 48-bit field that specifies the local station sending the frame. The source_address field contains an individual 48-bit MAC address (see 3.10), as specified in 9.2 of IEEE Std 802-2001.

6.2.1.2.3 priority: Specifies the priority associated with content delivery.

6.2.1.2.4 service_data_unit: A multi-byte field that provides information content.

For GrandSync-entity time-sync interchanges, the service_data_unit consists of the following subfields:

6.2.1.2.5 *protocolType*: A 16-bit field contained within the payload that identifies the format and function of the following fields.

6.2.1.2.6 *function*: An 8-bit field that distinguishes the timeSync frame from other AVB frame type.

6.2.1.2.7 *version*: An 8-bit field that identifies the version number associated with of the following fields. TBD—A more exact definition of version is needed.

6.2.1.2.8 *precedence*: A 14-byte field that specifies grand-master selection precedence (see 6.2.1.4).

6.2.1.2.9 *grandTime*: An 80-bit field that specifies a grand-master synchronized time (see 6.2.1.6).

6.2.1.2.10 *extraTime*: A 32-bit field that specifies the cumulative grand-master synchronized-time error. (Propagating *extraTime* and *grandTime* separately eliminates whiplash associated with cascaded PLLs.)

6.2.1.2.11 *sourcePort*: An 8-bit field that identifies the port that sourced the encapsulating content.

6.2.1.2.12 *hopCount*: An 8-bit field that identifies the maximum number of hops between the talker and associated listeners.

6.2.1.2.13 *snapTime*: A 56-bit field that specifies the local free-running time within this station, when the previous timeSync frame was received (see 6.2.1.8).

6.2.1.2.14 *syncInterval*: A 48-bit field that specifies the nominal period between timeSync frame transmissions.

NOTE—The *syncInterval* value is a port-specific constant value which (for apparent simplicity) has been illustrated as a relayed frame parameter. Other abstract communication techniques (such as access to shared design constants) might be selected to communicate this information, if requested by reviewers for consistency with existing specification methodologies.

6.2.1.3 Version format

For compatibility with existing 1588 time-snapshot, a single bit within the version field is constrained to be zero, as illustrated in Figure 6.3. The remaining *versionHi* and *versionLo* fields shall have the values of 0 and 1 respectively.

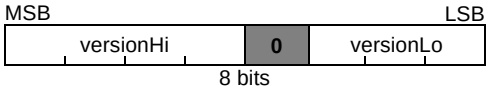

Figure 6.3—Global-time subfield format

6.2.1.4 *precedence* subfields

The precedence field includes the concatenation of multiple fields that are used to establish precedence between grand-master candidates, as illustrated in Figure 6.4.

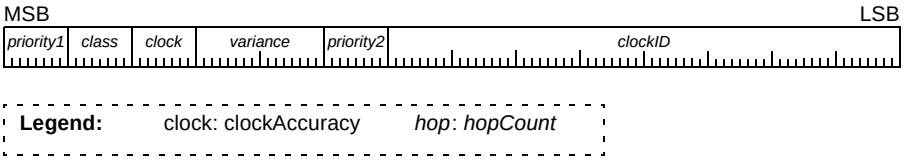

Figure 6.4—*precedence* subfields

6.2.1.4.1 *priority1*: An 8-bit field that can be configured by the user and overrides the remaining *precedence*-resident precedence fields.

6.2.1.4.2 *class*: An 8-bit precedence-selection field defined by the like-named IEEE-1588 field.

6.2.1.4.3 *clockAccuracy*: An 8-bit precedence-selection field defined by the like-named IEEE-1588 field.

6.2.1.4.4 *variance*: A 16-bit precedence-selection field defined by the like-named IEEE-1588 field.

6.2.1.4.5 *priority2*: A 8-bit field that can be configured by the user and overrides the remaining *precedence*-resident *clockID* field.

6.2.1.4.6 *clockID*: A 64-bit globally-unique field that ensures a unique precedence value for each potential grand master, when {*priority1*, *class*, *clockAccuracy*, *variance*, *priority2*} fields happen to have the same value (see 6.2.1.5).

6.2.1.5 *clockID* subfields

The 64-bit *clockID* field is a unique identifier. For stations that have a uniquely assigned 48-bit *macAddress*, the 64-bit *clockID* field is derived from the 48-bit MAC address, as illustrated in Figure 6.5.

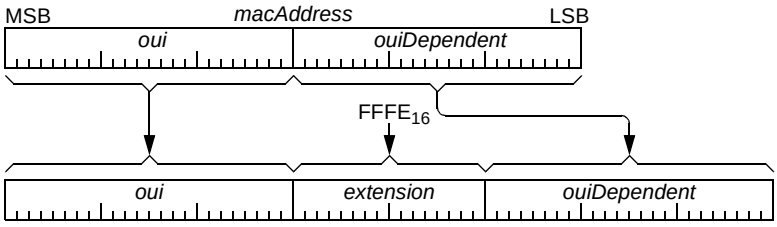

Figure 6.5—*clockID* format

6.2.1.5.1 *oui*: A 24-bit field assigned by the IEEE/RAC (see 3.10.1).

6.2.1.5.2 *extension*: A 16-bit field assigned to encapsulated EUI-48 values.

6.2.1.5.3 *ouiDependent*: A 24-bit field assigned by the owner of the *oui* field (see 3.10.2).

6.2.1.6 Global-time subfield formats

Time-of-day values within a frame are based on seconds and fractions-of-second values, consistent with IETF specified NTP[B7] and SNTP[B8] protocols, as illustrated in Figure 6.6.

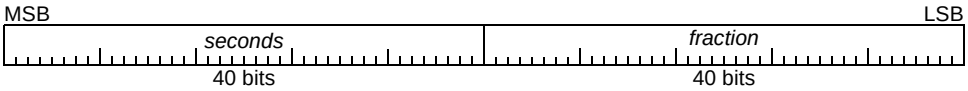

Figure 6.6—Global-time subfield format

6.2.1.6.1 *seconds*: A 40-bit signed field that specifies time in seconds.

6.2.1.6.2 *fraction*: A 40-bit unsigned field that specifies a time offset within each *second*, in units of 2^{-40} second.

The concatenation of these fields specifies a 96-bit *grandTime* value, as specified by Equation 6.1.

$$grandTime = seconds + (fraction / 2^{40}) \quad (6.1)$$

6.2.1.7 *extraTime*

The error-time values within a frame are based on a selected portion of a fractions-of-second value, as illustrated in Figure 6.7. The 40-bit signed *fraction* field specifies the time offset within a *second*, in units of 2^{-40} second.

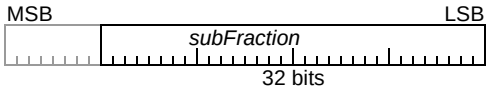

Figure 6.7—extraTime format

6.2.1.8 *snapTime* formats

The *snapTime* value within a frame is based on seconds and fractions-of-second field values, as illustrated in Figure 6.8. The 48-bit *fraction* field specifies the time offset within the *second*, in units of 2^{-48} second.

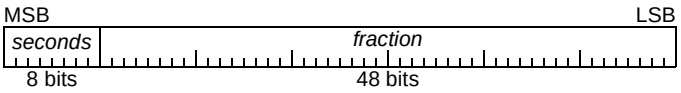

Figure 6.8—*snapTime* format

6.2.1.9 When generated

The time-sync related TIME_SYNC.indication service primitive is generated when new time-sync information is available. Such information could change the selection of the grand-master or could provide a more-recent {*grandTime*, *stationTime*} time affiliation necessary for maintaining accurate grand-master synchronized time references.

6.2.1.10 Effect of receipt

Receipt of the service primitive by the GrandSync entity triggers an update of the grand-master selection information. If the grand-master selection determines the source-port to be the preferred port, its provided

{*grandTime, stationTime*} time affiliation is also echoed to the attached entities, via invocation of the TIME_SYNC.request service primitive.

6.2.2 TIME_SYNC.request

6.2.2.1 Function

Communicates GrandSync protocol-entity supplied information to attached media-dependent ports. The information is sufficient for attached ports to update/propagate grand-master clock-synchronization parameters.

6.2.2.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
TIME_SYNC.request
{
    destination_address,      // Destination address
    source_address,           // Optional
    priority,                 // Forwarding priority
    service_data_unit,        // Delivered content
    {                         // Contents of the service_data_unit
        protocolType,         // Distinguishes AVB frames from others
        function,             // Distinguishes between timeSync and other frames
        version,              // Distinguishes between timeSync frame versions
        precedence,           // Precedence for grand-master selection
        grandTime,            // Global-time snapshot (1-cycle delayed)
        extraTime,            // Accumulated grandTime error
        sourcePort,           // Identifies the source port
        hopCount,             // Distance from the grand-master station
        snapTime,             // Local-time snapshot (1-cycle delayed)
        syncInterval          // Nominal timeSync transmission interval
    }
}
```

The parameters of the TIME_SYNC.request are described in 6.2.1.2.

6.2.2.3 When generated

Generated by the GrandSync entity upon receipt of a time-sync related TIME_SYNC.indication from a preferred (by grand-master selection protocol) source port.

6.2.2.4 Effect of receipt

Receipt of the service primitive by a ClockSlave or TS entity updates entity storage. This storage update allows the destination-port to provide accurate {*grandTime, stationTime*} affiliations during later time-sync information transmissions.

6.3 GrandSync state machine

6.3.1 Function

The GrandSync state machine is responsible for observing TIME_SYNC.indication parameters, selecting PDUs with preferred time-sync content, and echoing this content in following TIME_SYNC.request parameters.

6.3.2 State machine definitions

AVB identifiers
- Assigned constants used to specify AVB frame parameters.
  - AVB_FUNCTION—The function code that corresponds to a time-sync frame.
    - value—TBD.
  - AVB_MCAST—The multicast destination address corresponding to the adjacent neighbor.
    - value—TBD.
  - AVB_TYPE—The *protocolType* corresponding that uniquely identifies time-sync SDUs.
    - value—TBD.
  - AVB_VERSION—The number that uniquely identifies this version of time-sync SDUs.
    - value—TBD.

LAST_HOP
- A constant that specifies the largest possible *hopCount* value.
  - value—255

NULL
- A constant indicating the absence of a value that (by design) cannot be confused with a valid value.

ONES
- A large constant wherein all binary bits of the numerical representation are set to one.

queue values
- Enumerated values used to specify shared FIFO queue structures.
  - Q_MS_IND—Queue identifier for TIME_SYNC.indication transfers.
  - Q_MS_REQ—Queue identifier for TIME_SYNC.request transfers.

6.3.3 State machine variables

ePtr
- A pointer to entity-dependent storage, where that storage comprises the following:
  - *lastTime*—Time of the last best-preference update, used for timeout purposes.
  - *rxSaved*—A copy of the best-preference GrandSync PDU parameters.

new, old
- Local variables consisting of concatenated *preference*, *hopCount*, and *port* parameters.

rsPtr
- A pointer to the service-data-unit portion of *rxInfo* storage.

rxInfo
- Parameters associated with an TIME_SYNC.indication (see 6.2.1.2), comprising the following:
  - *destination_address*, *source_address*, *service_data_unit*
- Where *service_data_unit* comprises:
  - *extraTime*, *function*, *grandTime*, *hopCount*, *precedence*, *protocolType*, *snapTime*, *syncInterval*, *version*

rxPtr
- A pointer to the *rxInfo* storage.

stationTime
A shared value representing current time within each station.
Within the state machines of this standard, this is assumed to have two components, as follows:
seconds—An 8-bit unsigned value representing seconds.
fraction—An 40-bit unsigned value representing portions of a second, in units of 2^{-40} second.
ssPtr
A pointer to the service-data-unit portion of *ePtr->rxSaved* storage.
sxPtr
A pointer to the *ePtr->rxSaved* storage.
tsPtr
A pointer to the service-data-unit portion of *txInfo* storage.
txInfo
Parameters associated with an TIME_SYNC.request (see 6.2.1.2), comprising the following:
destination_address, *source_address*, *service_data_unit*
Where *service_data_unit* comprises:
extraTime, *function*, *grandTime*, *hopCount*, *precedence*,
protocolType, *snapTime*, *syncInterval*, *version*
txPtr
A pointer to the *txInfo* storage.

6.3.4 State machine routines

Dequeue(*queue*)
Returns the next available frame from the specified queue.
info—The next available parameters.
NULL—No parameters available.
Enqueue(*queue*, *info*)
Places the *info* parameters at the tail of the specified queue on all ports.
FormPreference(*precedence*, *hops*, *port*)
Forms a 16-byte *preference* by concatenating the following fields:
precedence (14 bytes)
hops (1 byte)
port (1 byte)
StationTime(*ePtr*)
Returns the value of the station's shared local timer, encoded as follows:
seconds—A 16-bit unsigned value representing seconds.
fraction—A 48-bit unsigned value representing portions of a second, in units of 2^{-40} second.
TimeSyncSdu(*info*)
Checks the frame contents to identify MS_DATAUNIT.indication frames.
TRUE—The frame is a timeSync frame.
FALSE—Otherwise.

6.3.5 GrandSync state table

The GrandSync state machine includes a media-dependent timeout, which effectively restarts the grand-master selection process in the absence of received timeSync frames, as specified by Table 6.1.

Table 6.1—GrandSync state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(rxInfo = Dequeue(Q_MS_IND)) != NULL	1	—	TEST
	(stationTime – ePtr->timer) > 4 * ePtr->syncInterval	2	ePtr->lastTime = stationTime; ssPtr->hopCount = ssPtr->sourePort = ssPtr->precedence = ONES;	START
	—	3	stationTime = StationTime();	
TEST	TimeSyncSdu(rsPtr) && rsPtr->hopCount != LAST_HOP	4	test = FormPreference(rsPtr->precedence, rsPtr->hopCount, rsPtr->port); best = FormPreference(ssPtr->precedence, ssPtr->hopCount, ePtr->sourcePort);	SERVE
	—	5	—	START
SERVE	rsPtr->port == ePtr->rxSourcePort	6	ePtr->lastTime = stationTime; *ssPtr = *tsPtr = *rsPtr;	HOPS
	test <= best	7		
	—	8	—	START
HOPS	rsPtr->hopCount > ssPtr->hopcount	9	tsPtr->hopCount = Min(LAST_HOP, 1 + (LAST_HOP + rsPtr->hopCount) / 2);	LAST
	—	10	tsPtr->hopCount = rsPtr->hopCount + 1;	

LAST	—	11	txPtr->destination_address = AVB_MCAST;
			txPtr->source_address = MacAddress(ePtr);
			tsPtr->protocolType = AVB_TYPE;
			tsPtr->function = AVB_FUNCTION;
			tsPtr->version = AVB_VERSION;
			Enqueue(Q_MS_REQ, txPtr);
				START

Row 6.1-1: Available indication parameters are processed.
Row 6.1-2: The absence of indications forces a timeout, after a entity-dependent delay
Row 6.1-3: Wait for changes of conditions.

Row 6.1-4: Still-active time-sync PDUs are processed further, based on grand-master preferences. The *new* and *old* preference values consist of *precedence*, *hopCount*, and *port* components.
Row 6.1-5: Other PDUs and over-aged indications are discarded.

Row 6.1-6: Same-port indications always have preference.
Row 6.1-7: Preferred preference-level indications are accepted.
Row 6.1-8: Other indications are discarded.

Row 6.1-9: Increasing *hopCount* values are indicative of a rogue frame and are therefore quickly quashed.
Row 6.1-10: Non-increasing *hopCount* values are incremented and are thus aged slowly.

Row 6.1-11: Reset the timeout timer; broadcast saved parameters to all ports (including the source).

7. ClockMaster/ClockSlave state machines

7.1 Overview

7.1.1 ClockMaster/ClockSlave behaviors

This clause specifies the state machines that specify ClockMaster and ClockSlave entity processing. The operations are described in an abstract way and do not imply any particular implementations or any exposed interfaces. There is not necessarily a one-to-one correspondence between the primitives and formal procedures and the interfaces in any particular implementation.

The ClockMaster entity is responsible for forwarding the grand-master time supplied by the ClockSource via the TIME_SOURCE.request service primitive, as follows:

a) A count value (that is incremented in sequential TIME_SOURCE.request PDUs) is checked.

b) The grand-master time parameter within the TIME_SYNC.request[*n*+1] PDU is associated with the TIME_SYNC.request[*n*] PDU arrival time.

c) The TIME_SYNC.request parameters are supplemented to form a TIME_SYNC.request PDU, which is then passed to the GrandSync entity.

The ClockSlave entity is responsible for extracting the grand-master time from TIME_SYNC.indications and supplying the current value to the ClockSink entity through the TIME_SINK service interfaces, as follows:

a) Grand-master time samples are extracted from GrandSync-supplied TIME_SYNC.request PDUs, and saved for computing grand-master times in following TIME_SINK.indication PDUs.

b) When triggered by a TIME_SINK.request indication, a TIME_SINK.indication PDU is delivered to the ClockSink state machine. That returned TIME_SINK.indication PDU supplies the grand-master time associated with the TIME_SINK.request invocation time.

7.1.2 ClockMaster/ClockSlave interface model

The time-synchronization service model assumes the presence of one or more grand-master capable entities communicating with the GrandSync state machine, as illustrated on the left side of Figure 7.1. A grand-master capable port is also expected to provide clock-slave functionality, so that any non-selected grand-master-capable station can synchronize to the selected grand-master station.

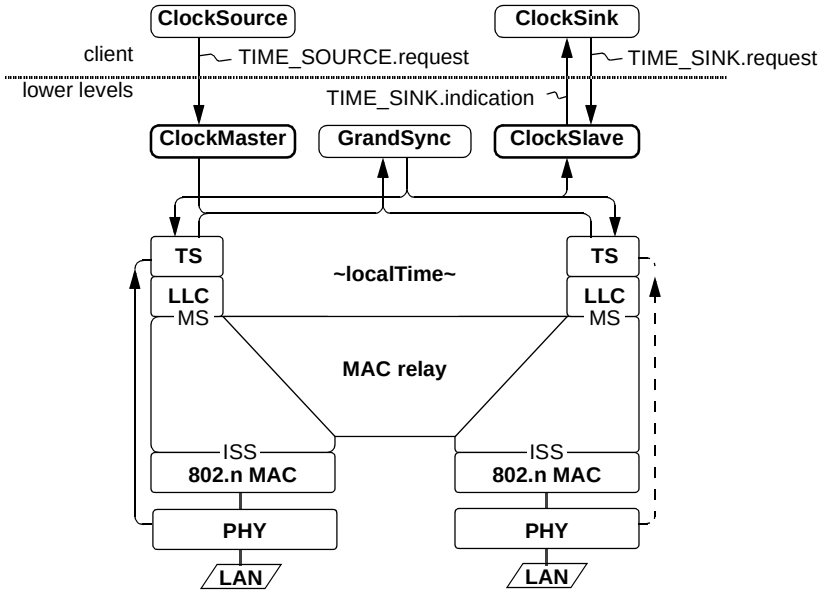

Figure 7.1—ClockMaster interface model

The clock-master ClockMaster state machine (illustrated with an italics name and darker boundary) is responsible for monitoring its port's TIME_SOURCE.request PDUs and sending TIME_SYNC.request frames. The sequencing of this state machine is specified by Table 7.1; details of the computations are specified by the C-code of Annex F.

The time-synchronization service model assumes the presence of one or more clock-slave capable time-sync entities communicating with a GrandSync protocol entity, as illustrated on the top-side of Figure 7.1. A non-talker clock-slave capable entity is not required to be grand-master capable.

The ClockSlave state machine (illustrated with an italics name and darker boundary) is responsible for saving time parameters from relayed TIME_SYNC.request frames and servicing TIME_SOURCE.request PDUs supplied by the associated clock-slave interface. The sequencing of this state machine is specified by Table 7.2; details of the computations are specified by the C-code of Annex F.

7.2 ClockMaster service interfaces

7.2.1 Shared service interfaces

The ClockMaster entity is coupled to the bridge ports TS entities via the defined time-sync related TIME_SYNC.indication service interface (see 6.2.1).

7.2.2 TIME_SOURCE.request service interface

7.2.2.1 Function

Provides the ClockMaster entity with clock-synchronization parameters derived from the reference clock. The information is sufficient to provide the ClockMaster with accurate {*grandTime*, *localTime*} associations. The ClockSource entity supplies the reference time for service-interface invocation *n* within the parameters of the next service-interface invocation *n*+1.

7.2.2.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
TIME_SOURCE.request {
    frameCount,          // An integrity-check that is incremented each invocation
    grandTime,           // Global-time snapshot (1-cycle delayed)
}
```

The parameters of the TIME_SOURCE.request service-interface primitive are described as follows:

7.2.2.2.1 *frameCount*: An 8-bit field that is incremented on each service-interface invocation.

7.2.2.2.2 *grandTime*: An 80-bit field that specifies the grand-master synchronized time within the source station, when the previous timeSync frame was transmitted (see 6.2.1.6).

7.2.2.3 When generated

The TIME_SOURCE.request service primitive is invoked by a client-resident ClockSource entity. The intent is to provide the ClockMaster with continuous/accurate updates from a ClockSource-resident clock reference.

7.2.2.4 Effect of receipt

Upon receipt by the ClockMaster entity, the encapsulated *grandTime* value is affiliated with the *stationTime* snapshot from the previous invocation; the resulting {*grandTime*, *stationTime*} affiliation is passed to the GrandSync entity for redistribution to other ClockSlave and TS entities.

7.3 ClockMaster state machine

7.3.1 State machine definitions

AVB identifiers
    Assigned constants used to specify AVB frame parameters (see 6.3.2).
        AVB_FUNCTION, AVB_MCAST, AVB_TYPE, AVB_VERSION
NULL
    A constant indicating the absence of a value that (by design) cannot be confused with a valid value.
COUNT
    A numerical constant equal to the range of the *info.frameCount* field value.
queue values
    Enumerated values used to specify shared FIFO queue structures.
        Q_CM_SET—The queue identifier associated with received clock-master sync frames.
        Q_MS_IND—A GrandSync queue identifier (see 6.3.2).

7.3.2 State machine variables

count
A transient value representing the expected value of the next *rxInfo.frameCount* field value.
ePtr
A pointer to an entity data structure with information comprising the following:
precedence—A 14-byte field that specifies the grand-master selection precedence.
rxSaved—Saved parameters from a received TIME_SOURCE.request primitive.
snapShot0—The *info.snapShot* field value from the last receive-port poke indication.
snapShot1—The value of the *ePtr->snapShot0* field saved from the last poke indication.
syncInterval—The expected rate of clockMaster service-interface invocations.

rxInfo
A contents of a higher-level supplied time-synchronization request, including the following:
frameCount—A value that increments on each TIME_SOURCE.request frame transmission.
grandTime—The value of grand-master time, when the previous TIME_SOURCE.request frame was sent.
rxPtr
A pointer to *rxInfo* storage.
stationTime
See 6.3.3.
sxPtr
A pointer to the *ePtr->rxSaved* storage.
tsPtr
A pointer to the service-data-unit portion of *txInfo* storage.
txInfo
Storage for to-be-transmitted TIME_SYNC.request parameters (see 6.2.2.2), comprising:
destination_address, *source_address*, *service_data_unit*
Where *service_data_unit* comprises:
extraTime, *function*, *grandTime*, *hopCount*, *precedence*,
protocolType, *snapTime*, *sourcePort*, *syncInterval*, *version*
txPtr
A pointer to *txInfo* storage.

7.3.3 State machine routines

Dequeue(*queue*)
Enqueue(*queue*, *info*)
SourcePort(*entity*)
StationTime(*entity*)
TimeSyncSdu(*info*)
See 6.3.4.

7.3.4 ClockMaster state table

The ClockMaster state table encapsulates clock-provided sync information into a MAC-relay frame, as illustrated in Table 7.1.

Table 7.1—ClockMaster state machine table

Current		Row	Next	
state	**condition**		**action**	**state**

START	(rxInfo = Dequeue(Q_CM_SET)) != NULL	1	ePtr->snapShot1 = ePtr->snapShot0;
			ePtr->snapShot0 = stationTime;
			count = (sxPtr->frameCount + 1) % COUNT;
			grandTime = rxPtr->grandTime;
			*sxPtr = rxInfo;
				SEND

	—	2	stationTime = StationTime(ePtr);	START

SEND	count == sxPtr->frameCount	3	txPtr->destination_address = AVB_MCAST;
			txPtr->source_address = MacAddress(ePtr);
			tsPtr->prototolType = AVB_TYPE;
			tsPtr->function = AVB_FUNCTION;
			tsPtr->version = AVB_VERSION;
			tsPtr->precedence = ePtr->precedence;
			tsPtr->hopCount = 0;
			tsPtr->sourcePort = SourcePort(ePtr);
			tsPtr->grandTime = grandTime;
			tsPtr->extraTime = 0;
			tsPtr->snapTime = ePtr->snapShot1;
			tsPtr->syncInterval = ePtr->syncInterval;
			Enqueue(Q_MS_IND, txInfo);
				START

	—	4	—	

Row 7.1-1: Update snapshot values on TIME_SOURCE.request request arrival.
Row 7.1-2: Wait for the next change of state.

Row 7.1-3: Sequential requests are forwarded as a TIME_SYNC.request to the GrandSync entity.
Row 7.1-4: Nonsequential requests are discarded.

7.4 ClockSlave service interfaces

7.4.1 Shared service interfaces

The ClockSlave entity is coupled to the GrandSync entity, via the defined TIME_SYNC.request service interface (see 6.2.2).

7.4.2 TIME_SINK.request service interface

7.4.2.1 Function

Triggers the ClockSlave entity to provide a {*grandTime*, *localTime*} association that is synchronized with the grand-master clock.

7.4.2.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
TIME_SINK.request {
    frameCount          // An integrity-check that is incremented each invocation
}
```

The parameters of the TIME_SOURCE.request service-interface primitive are described as follows:

7.4.2.2.1 *frameCount*: An 8-bit field that is incremented on each service-interface invocation.

7.4.2.3 When generated

The TIME_SINK.request service primitive is invoked by a client-resident ClockSink entity. The intent is to trigger the ClockSlave's invocation of a following TIME_SINK.indication primitive, thus providing the ClockSink entity with a recent {*grandTime*,*stationTime*} affiliation.

7.4.2.4 Effect of receipt

Upon receipt by a ClockSlave entity, a copy of the current *stationTime* value is saved and an invocation of a following TIME_SINK.indication primitive is triggered.

7.4.3 TIME_SINK.indication service interface

7.4.3.1 Function

Provides the ClockSync entity with clock-synchronization parameters derived from the reference clock. The information comprises {*frameCount*, *grandTime*} associations: *frameCount* is supplied by the previous TIME_SINK.request invocation; *grandTime* represents the invocation time of that preceding TIME_SINK.request service primitive.

7.4.3.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
TIME_SINK.indication {
    frameCount,         // Identifies the previous TIME_SINK.request invocation
    grandTime,          // Grand-master synchronized snapshot.
}
```

The parameters of the TIME_SINK.indication service-interface primitive are described as follows:

7.4.3.2.1 *frameCount*: An 8-bit field that copied from the like-named field of the previous TIME_SINK.request service-interface invocation.

7.4.3.2.2 *grandTime*: An 80-bit field that specifies the grand-master synchronized time within the ClockSlave entity, when the previous TIME_SINK.request service-interface was invoked.

7.4.3.3 When generated

The invocation of the TIME_SINK.indication service primitive is invoked by the receipt of a ClockSink supplied TIME_SINK.request PDU. The intent is to provide the ClockSink entity with a recent {*grandTime*,*stationTime*} affiliation.

7.4.3.4 Effect of receipt

Upon receipt by a ClockSink entity, the {*grandTime*,*stationTime*} affiliation is expected to be saved and (along with previously saved copies) used to adjust the rate of the grand-master synchronized ClockSink-resident clock.

7.5 ClockSlave state machine

7.5.1 Function

7.5.2 State machine definitions

NULL
A constant indicating the absence of a value that (by design) cannot be confused with a valid value.
queue values
Enumerated values used to specify shared FIFO queue structures.
Q_MS_REQ—A GrandSync queue identifier (see 6.3.2).
Q_CS_REQ—The queue identifier associated with TIME_SINK.request requests.
Q_CS_IND—The queue identifier associated with TIME_SINK.indication indications.

7.5.3 State machine variables

ePtr
A pointer to entity-dependent information, including the following:
rxSaved—A copy of the GrandSync supplied MA_DATAUNIT.request value.
syncInterval—The expected service rate of TIME_SINK.request services.
baseTimer—Recently saved time events, each consisting of the following:
index—Index into the *timed*[ ] array, where last times were stored.
range—Number of entries within the *timed*[ ] array
timed[range]—Recently saved time events, each consisting of the following:
grandTime—A previously sampled grand-master synchronized time.
extraTime—The residual error associated with the sampled *grandTime* value.
stationTime—The station-local time affiliated with the sampled *grandTime* value.
cxInfo
A contents of a higher-level supplied time-synchronization request, including the following:
frameCount—A value that increments on each TIME_SOURCE.request PDU transfer.
nextTime
Storage representing *grandTime* and *extraTime* values returned from call to *NextTimed*( ).

rsPtr
A pointer to the service-data-unit portion of *rxInfo*.
rxInfo
A contents of a GrandSync supplied TIME_SYNC.request (see 6.2.2), including the following:
destination_address, source_address, service_data_unit
Where *service_data_unit* comprises:
extraTime, function, grandTime, protocolType, snapTime, version
rxPtr
A pointer to *rxInfo*.
rxSyncInterval
The synchronization interval of this station's GrandSync-selected clock-slave port.
stationTime
See 6.3.3.
ssPtr
A pointer to the service-data-unit portion of the *ePtr->rxSaved* storage
sxPtr
A pointer to the *ePtr->rxSaved* storage
timePtr
A pointer to the *ePtr->timed*[ ] array storage
txInfo
A contents of a ClockSlave supplied TIME_SINK.indication (see 6.2.2), comprising the following:
frameCount—The saved value of the like named field from the previous TIME_SINK.request PDU.
grandTime—The grand-master synchronized time sampled during the TIME_SINK.request transfer.
txPtr
A pointer to *txInfo* storage.
txSyncInterval
The synchronization interval of this ClockSlave entity.

7.5.4 State machine routines

Dequeue(*queue*)
Enqueue(*queue, info*)
See 6.3.4.
NextSaved(*btPtr, rateInterval, grandTime, extaTime, thisTime*)
Saves *grandTime, extraTime* values associated with a snapshot taken at *thisTime*, with the saved values spanning a *rateInterval* specified interval.
NextTimed(*btPtr, stationTime, backInterval*)
Returns *grandTime* and *extraTime* values associated with a snapshot taken at *stationTime*, back-interpolated by a *backInterval* time, based on previous received-time information saved in the *btPtr* referenced data structure.
StationTime(*entity*)
See 6.3.4.
TimeSyncSdu(*info*)
See 7.3.3.

7.5.5 ClockSlave state table

The ClockSlave state machine includes a media-dependent timeout, which effectively disconnects a clock-slave port in the absence of received timeSync frames, as illustrated in Table 7.2.

Table 7.2—ClockSlave state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(rxInfo = Dequeue(Q_MS_REQ)) != NULL	1	—	TEST

	((cxInfo = Dequeue(Q_CS_REQ)) != NULL	2	rxSyncInterval = ssPtr->syncInterval;
			txSyncInterval = ePtr->syncInterval;
			backInterval =
			(3 * rxSyncInterval + txSyncInterval) / 2;
			nextTimes =
			NextTimed(btPtr, stationTime, backInterval);
			txPtr->count = cxInfo.count;
			txPtr->grandTime =
			nextTimes.grandTime + nextTimes.extraTime;
			Enqueue(Q_CS_IND, txInfo);
				START

	—	3	stationTime = StationTime(ePtr);	
TEST	TimeSyncSdu(rsPtr)	4	*sxPtr = *rxPtr; NextSaved(btPtr, rateInterval, rsPtr->grandTime; rsPtr->extraTime, rsPtr->snapTime);	START
	—	5	—	

Row 7.2-1: The received TIME_SYNC.request parameters are dequeued for checking.
Row 7.2-2: A clock-slave request generates an affiliated information-providing indication.
The affiliated indication has the sequence-count information provided by the request.
The delivered end-point *grandTime* value is the sum of delivered *grandTime* and *extraTime* values.
The requested content is queued for delivery to the higher-level client.
Row 7.2-3: Wait for the next change-of-conditions.

Row 7.2-4: Validated GrandSync entity requests are accepted; its time parameters are saved.
The back-interpolation time is estimated from the *syncInterval* times of the source and clock slave.
(This back-interpolation time is used by *NextTimed*( ), which provides transmission-time estimates.)
Row 7.2-5: Wait for the next change-of-conditions.

8. Ethernet full duplex (EFDX) state machines

8.1 Overview

This clause specifies the state machines that support 802.3 Ethernet full duplex (EFDX) bridges. The operations are described in an abstract way and do not imply any particular implementations or any exposed interfaces. There is not necessarily a one-to-one correspondence between the formal specification and the interfaces in any particular implementation.

8.1.1 EFDX link indications

The duplex-link TimeSyncRxEfdx state machines are provided with snapshots of timeSync-frame reception and transmission times, as illustrated by the ports within Figure 8.1. These link-dependent indications can be different for bridge ports attached to alternative media.

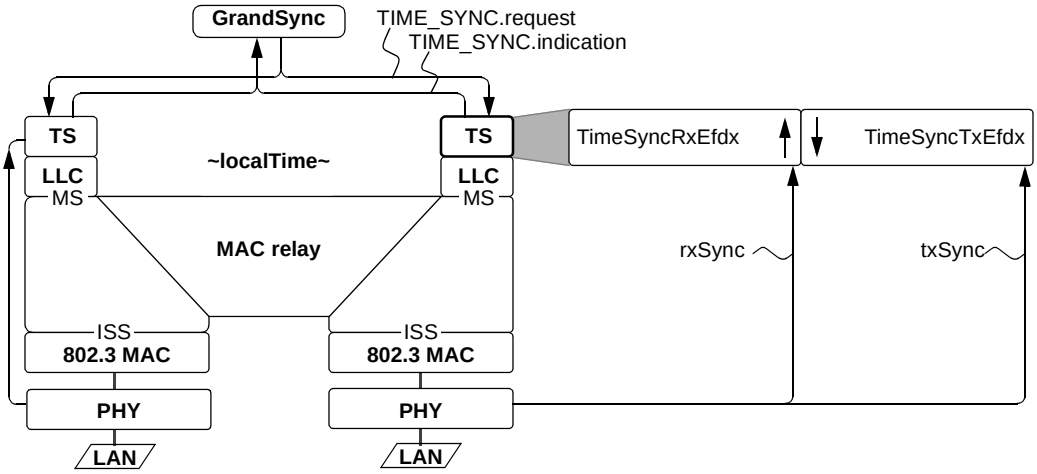

Figure 8.1—EFDX-link interface model

The rxSync and txSync indications provide a tag (to reliably associate them with MAC-supplied timeSync frames) and a *localTime* stamp indicating when the associated timeSync frame was received, as illustrated within Figure 8.2.

Figure 8.2—Contents of rxSync/txSync indications

8.1.2 Link-delay compensation

Synchronization accuracies are affected by the transmission delays associated with transmissions over links between bridges. To compensate for these transmission delays, the receive port is responsible for compensating {*grandTime*, *stationTime*} affiliations by the (assumed to be constant) frame-transmission delay.

The clock-slave entity uses the computed cable-delay measurement and is therefore (in concept) responsible for initiating such measurements. Cable-delay measurements begin with the transmission of frame F1 between the clock-slave and clock-master stations and conclude with the a clock-master response, a transmission of frame F2 between the clock-master to clock-slave stations, as illustrated in Figure 8.3.

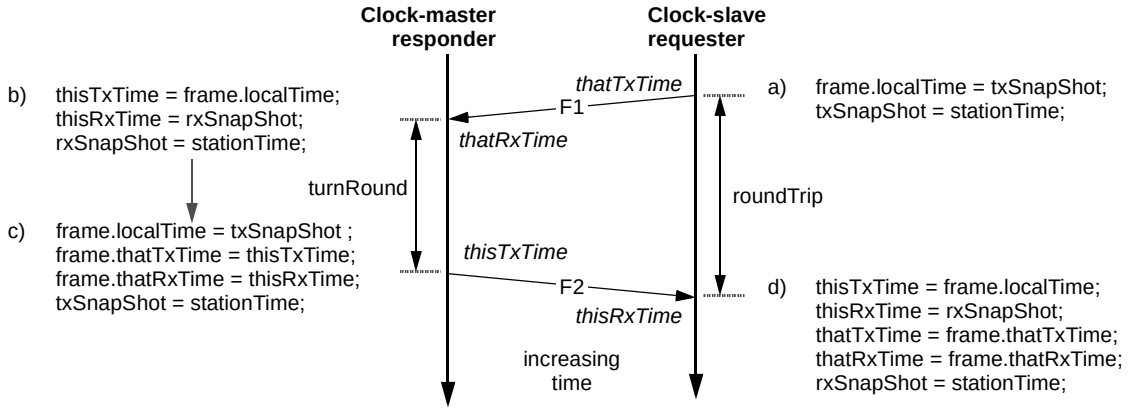

Figure 8.3—Link-delay compensation

The cable-delay computations are performed in multiple steps, as follows:

a) The F1-frame transmission involves multiple steps:
   1) The *txSnapShot* value (time of the last F1 transmission) is copied to *frame*.*localTime* storage.
   2) Remaining fields are copied into *frame* storage; the *frame*-storage content is transmitted.
   3) The *txSnapShot* value is set to the frame-F1 transmission time, for next step (a) usage.

b) The F1-frame reception involves multiple steps:
   1) The *frame*.*localTime* value is copied to a port-local *thisTxTime* field, for next step (c) usage.
   2) The *rxSnapShot* value (time of the last F1 reception) is copied to a port-local *thisRxTime* field.
   3) The *rxSnapShot* value is set to the F1-frame reception time, for next step (b) usage.

c) The F2-frame transmission involves multiple steps:
   1) The *txSnapShot* value (time of the last F1 transmission) is copied to *frame*.*localTime* storage.
   2) The receive-port *thisTxTime* value is copied to *frame*.*thatTxTime* storage.
   3) The receive-port *thisRxTime* value is copied to *frame*.*thatRxTime* storage.
   4) Remaining fields are copied into *frame* storage; the *frame*-storage content is transmitted.
   5) The *txSnapShot* value is set to the frame-F2 transmission time, for next step (c) usage.

d) The F2-frame reception involves multiple steps:
   1) The *frame*.*localTime* value is copied to a port-local *thisTxTime* field.
   2) The *rxSnapShot* value (time of the last F2 reception) is copied to a port-local *thisRxTime* field.
   3) The *frame*.*thatTxTime* value is copied to a port-local *thatTxTime* field.
   4) The *frame*.*thatRxTime* value is copied to a port-local *thatRxTime* field.
   5) The *rxSnapShot* value is set to the F2-frame reception time, for next step (d) usage.

At the conclusion of these steps, the values returned to the clock-slave requester include the values below. (Within Figure 8.3, these values are also illustrated in the center, at their source, using a distinct italic font.)

— *thatTxTime*. The clock-slave transmit time.
— *thatRxTime*. The clock-master receipt time.
— *thisTxTime*. The clock-master transmit time.
— *thisRxTime*. The clock-slave receipt time.

Based on the preceding listed values, Equation 8.1 defines the computations for computing *linkDelay.* Although not explicitly stated, the best accuracy can be achieved by performing these computation every cycle.

linkDelay = (*roundTrip* – *turnRound*) / 2; (8.1)
roundTrip = *thisRxTime* – *thatTxTime*;
turnRound = (*thisTxTime* – *thatRxTime*) * *ratesRatio*;
Where:
ratesRatio ≅ (*deltaRxTime* / *deltaTxTime*);

The value of *ratesRatio* is necessary to maintain tight accuracies in the presence of significant (±200 PPM) differences in clock-master/clock-slave timing references and significant (multiple milliseconds) *turnRound* delays. This value is also readily computed from the preceding listed values, as specified by Equation 8.2.

ratesRatio[*n*] = (*thisRxTime*[*n*] – *thisRxTime*[*n*-N]) / (*thisTxTime*[*n*] – *thisTxTime*[*n*-N]); (8.2)

NOTE—For 802.3 and other inexpensive interconnects, the processing of slow-rate PDUs is oftentimes performed by firmware and (due to interrupt and processing delays) the turn-around delays can be much larger than the packet-transmission times.

The cable-delay computations assume the transmission delays associated with frame F1 and frame F2 are equal and constant. If the duplex links within a span have different propagation delays, these *linkDelay* calculations do not correspond to the different propagation delays, but represent the average of the two link delays. Implementers have the option of manually specifying the link-delay differences via MIB-accessible parameters, within tightly-synchronized systems where this inaccuracy might be undesirable.

This cable-delay calculation does not rely on the particular timings of F1 and F2 frame transmissions. These transmissions can be triggered independently (as opposed to one triggered by the other) and could occur at different rates (although the accuracies are limited by the slower rate). As a direct benefit of these independence properties, distinct interlocks or timeouts for expected-but-corrupted-and-not-delivered transmissions are unnecessary.

Furthermore, there is no need to transport F1 and F2 content in distinct frames. The contents of clock-slave affiliated F1 and clock-master affiliated F2 frames can be merged and transported within the same frame. Thus, distinct frame types and/or transmission timings are unnecessary; the link-delay calibration protocols do nothing to prevent the same frame from communicating master-to-slave and slave-to-master link delays, in addition to the baseline grand-master timing and selection parameters.

8.2 timeSyncEfdx frame format

8.2.1 timeSyncEfdx fields

EFDX time-synchronization (timeSyncEfdx) frames facilitate the synchronization of neighboring clock-master and clock-slave stations. The frame, which is normally sent at 10ms intervals, includes time-snapshot information and the identity of the network's clock master, as illustrated in Figure 8.4. The gray boxes represent physical layer encapsulation fields that are common across Ethernet frames.

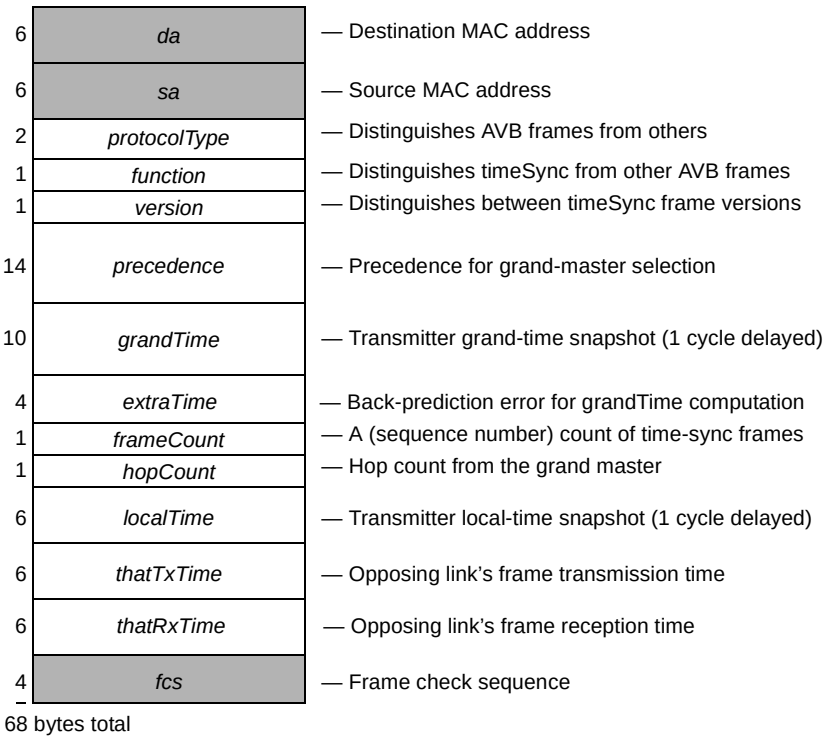

Figure 8.4—timeSyncEfdx frame format

NOTE— Existing 1588 time-snapshot hardware captures the values between byte-offset 34 and 45 (inclusive). The location of the *frameCount* field (byte-offset 44) has been adjusted to ensure this field can be similarly captured for the purpose of unambiguously associating timeSync-packet snapshots (that bypass the MAC) and timeSync-packet contents (that pass through the MAC).

The 48-bit ***da*** (destination address), 48-bit ***sa*** (source address) field, 16-bit ***protocolType,*** 8-bit ***function***, 8-bit ***version***, 14-byte ***precedence***, 80-bit ***grandTime***, 32-bit ***extraTime***, 8-bit ***hopCount***, and 6-byte ***localTime*** field are specified in 6.2.1.2.

8.2.1.1 *frameCount*: An 8-bit field that is incremented by one between successive timeSync frame transmission.

8.2.1.2 *thatTxTime*: A 48-bit field that specifies the local free-running time within the source station, when the previous timeSync frame was transmitted on the opposing link (see 6.2.1.8).

8.2.1.3 *thatRxTime*: A 48-bit field that specifies the local free-running time within the target station, when the previous timeSync frame was received on the opposing link (see 6.2.1.8).

8.2.1.4 *fcs*: A 32-bit (frame check sequence) field that is a cyclic redundancy check (CRC) of the frame.

8.2.2 Clock-synchronization intervals

Clock synchronization involves synchronizing the clock-slave clocks to the reference provided by the grand clock master. Tight accuracy is possible with matched-length duplex links, since bidirectional frame transmissions can cancel the cable-delay effects.

Clock synchronization involves the processing of periodic events. Multiple time periods are involved, as listed in Table 8.1. The clock-period events trigger the update of free-running timer values; the period affects the timer-synchronization accuracy and is therefore constrained to be small.

Table 8.1—Clock-synchronization intervals

Name	Time	Description
clock-period	< 20 ns	Resolution of timer-register value updates
send-period	10 ms	Time between sending of periodic timeSync frames between adjacent stations
slow-period	100 ms	Time between computation of clock-master/clock-slave rate differences

The send-period events trigger the interchange of timeSync frames between adjacent stations. While a smaller period (1 ms or 100 μs) could improve accuracies, the larger value is intended to reduce costs by allowing computations to be executed by inexpensive (but possibly slow) bridge-resident firmware.

The slow-period events trigger the computation of timer-rate differences. The timer-rate differences are computed over two slow-period intervals, but recomputed every slow-period interval. The larger 100 ms (as opposed to 10 ms) computation interval is intended to reduce errors associated with sampling of clock-period-quantized slow-period-sized time intervals.

8.3 TimeSyncRxEfdx state machine

8.3.1 Function

The TimeSyncRxEfdx state machine is responsible for monitoring its port's rxSync indications, receiving MAC-supplied frames, and sending MAC-relay frames. The sequencing of this state machine is specified by Table 8.2; details of the computations are specified by the C-code of Annex F.

8.3.2 State machine definitions

LAST_HOP
: A constant representing the largest-possible frame.hopCount value.
  value—255.

NULL
: A constant indicating the absence of a value that (by design) cannot be confused with a valid value.

queue values
: Enumerated values used to specify shared FIFO queue structures.
  Q_MS_IND—The queue identifier associated with MAC frames sent into GrandSync.
  Q_ES_IND—The queue identifier associated with the received MAC frames.
  Q_RX_SYNC—The queue identifier associated with rxSync, sent from the lower levels.

8.3.3 State machine variables

cableDelay
: Values (possibly scaled integers) representing cable-delay times.

count
: A transient value representing the expected value of the next *rxInfo.frameCount* value.

cxInfo
: A contents of a lower-level supplied time-synchronization poke indication, including the following:
  frameCount—The value of the like-named field within the last timeSync packet arrival.
  snapTime—The value of *stationTime* associated with the last timeSync packet arrival.

cxPtr
: A pointer to *cxInfo* storage.

ePtr
: A pointer to a data structure that contains port-specific information comprising the following:
  frameCount—The value of *frameCount* within the last received frame.
  rated—The ratio of the local-station and remote-station local-timer rates.
  snapCount—The value of *frameCount* saved from the last snapshot indication.
  snapShot0—The *info.snapShot* field value from the last receive-port snapshot indication.
  snapShot1—The value of the *ePtr->snapShot0* field at the snapshot indication.
  times[*N*]—An array of time groups, where each array elements consists of:
    thisTime—The local receive time associated with received time-sync frames.
    thatTime—The remote transmit time associated with received time-sync frames.

ratesRatio
: A variable representing the ratio of this station's timer to this port's neighbor timer.

roundTrip
: A variable representing the time between transmit-to-neighbor and receive-from-neighbor events.

rsPtr
: A pointer to the service-data-unit portion of *rxInfo* storage.

rxInfo
Storage for received time-sync PDUs, comprising:
destination_address, *source_address*, *service_data_unit*
Where *service_data_unit* comprises:
extraTime, *frameCount*, *function*, *grandTime*, *hopCount*, *localTime*, *protocolType*, *precedence*, *thatTxTime*, *thatRxTime*, *version*

rxPtr
A pointer to the *rxInfo* storage.

stationTime
See 6.3.3.

thisDelay, *thatDelay*, *thatDelay*, *thisDelta*, *thisTime*, *thatTime*, *tockTime*
Values (possibly scaled integers) representing intermediate local-time values.

tsPtr
A pointer to service-data-unit portion of *txInfo* storage.

turnRound
A variable representing the time between receive-at-neighbor and transmit-from-neighbor events.

txInfo
Storage for information sent to the GrandSync entity, comprising:
destination_address, *source_address*, *service_data_unit*
Where *service_data_unit* comprises:
extraTime, *sourcePort*, *function*, *grandTime*, *hopCount*, *localTime*, *protocolType*, *precedence*, *syncInterval*, *version*

txPtr
A pointer to *txInfo* storage.

8.3.4 State machine routines

Dequeue(*queue*)

Enqueue(*queue*, *info*)
See 6.3.4.

Min(*x*, *y*)
Returns the minimum of *x* and *y* values.

RemoteRate(*times*)
The ratio of local-to-remote *localTime* rates is computed from samples within the of *times* array.
Each times-array element contains two times:
thisTime - the receive time of the frame.
thatTime - the transmit time of the frame.

SourcePort(*entity*)
See 7.3.3.

StationTime(*entity*)

TimeSyncSdu(*info*)
See 6.3.4.

8.3.5 TimeSyncRxEfdx state machine table

The TimeSyncRxEfdx state machine associates PHY-provided sync information with arriving timeSync frames and forwards adjusted frames to the MAC-relay function, as illustrated in Table 8.2.

Table 8.2—TimeSyncRxEfdx state machine table

Current		Row	Next	
state	**condition**		**action**	**state**

START	(cxInfo = Dequeue(Q_RX_SYNC)) != NULL	1	ePtr->snapShot1 = ePtr->snapShot0;
			ePtr->snapShot0 = cxPtr->localTime;
			ePtr->snapCount = cxPtr->frameCount;
			match= (rxPtr->frameCount==ePtr->snapCount);
				START

	(rxInfo=Dequeue(Q_ES_IND)) != NULL	2	count = (ePtr->rxFrameCount + 1) % COUNT; ePtr->rxFrameCount = rxPtr->frameCount;	TEST
	match	3	—	PAIR
	—	4	stationTime = StationTime(ePtr);	START
TEST	!TimeSyncSdu(rsPtr)	5	Enqueue(Q_CS_IND, rxPtr);	START
	rxPtr->hopCount == LAST_HOP	6	—	
	count != rxPtr->frameCount	7	—	
	—	8	match= (rxPtr->frameCount==ePtr->snapCount);	

PAIR	—	9	ePtr->times[0].thisTime = ePtr->snapShot1;
			ePtr->times[1].thatTime = rsPtr->localTime;
			ratesRatio = RemoteRate(ePtr->times);
			roundTrip = localTime – ePtr->thatTxTime;
			turnRound =
			rsPtr->localTime – rsPtr->thatRxTime;
			cableDelay =
			Min(0, roundTrip – (turnRound * ratesRatio));
			txPtr->destination_address =
			rxPtr->destination_address;
			txPtr->source_address = rxPtr->source_address;
			tsPtr->protocolType = rsPtr->protocolType;
			tsPtr->function = rsPtr->function;
			tsPtr->version = rsPtr->version;
			tsPtr->grandTime = rsPtr->grandTime;
			tsPtr->extraTime = rsPtr->extraTime;
			tsPtr->snapTime = ePtr->snapShot1 – cableDelay;
			tsPtr->sourcePort = SourcePort(ePtr);
			tsPtr->hopCount = rsPtr->hopCount;
			tsPtr->syncInterval = ePtr->syncInterval;
			Enqueue(Q_MR_HOP, relayFrame);
				START

Row 8.2-1: Update snapshot values on timeSync frame arrival.
Row 8.2-2: Initiate inspection of frames received from the lower-level MAC.
Row 8.2-3: Generate a GrandSync PDUs using matching snapshot and frame information.
Row 8.2-4: Wait for the next change-of-state.

Row 8.2-5: The non-timeSync frames are passed through.
Row 8.2-6: Over-aged timeSync frames are discarded.
Row 8.2-7: Non-sequential timeSync frames are ignored.
Row 8.2-8: Associated snapshot and frame information trigger a GrandSync indication generation.

Row 8.2-9: Generate a time-sync GrandSync indication from saved snapshot and frame information.

8.4 TimeSyncTxEfdx state machine

8.4.1 Function

The TimeSyncTxEfdx state machine is responsible for saving time parameters from relayed timedSync frames and forming timeSync frames for transmission over the attached link.

8.4.2 State machine definitions

NULL
A constant indicating the absence of a value that (by design) cannot be confused with a valid value.
queue values
Enumerated values used to specify shared FIFO queue structures.
Q_MR_HOP—The queue identifier associated with frames sent from the relay.
Q_ES_REQ—The queue identifier associated with frames sent to the MAC.
Q_TX_SYNC—The queue identifier associated with txSync, sent from the lower levels.
T10ms
A constant the represents a 10 ms value.

8.4.3 State machine variables

backInterval
A variable that represents the back-interpolation interval for transmit-time affiliations.
cxInfo
A contents of a lower-level supplied time-synchronization poke indication, including the following:
snapCount—The value of the like-named field within the last timeSync packet arrival.
snapTime—The value of *stationTime* associated with the last timeSync packet arrival.
dPtr
A pointer this port's associated TimeSyncRxEfdx-entity storage.

ePtr
A pointer to a data structure that contains port-specific information comprising the following:
baseTimer—Recently saved time events, each consisting of the following:
index—Index into the *timed*[ ] array, where last times were stored.
range—Number of entries within the *timed*[ ] array
timed[range]—Recently saved time events, each consisting of the following:
grandTime—A previously sampled grand-master synchronized time.
extraTime—The residual error associated with the sampled *grandTime* value.
stationTime—The station-local time affiliated with the sampled *grandTime* value.
frameCount—A consistency-check identifier that is incremented on each transmission.
lastTime—The last transmit time, saved for timeout purposes.
rxSaved—A copy of the last received GrandSync parameters.
syncInterval—The expected interval between successive time-sync transmissions.
txSnapCount—The *frameCount* value associated with the last transmission.
txSnapTime—The *stationTime* value associated with the last transmission.
rsPtr
A pointer to service-data-unit portion of *rxInfo* storage.
rxInfo
Storage for received time-sync PDUs from the GrandSync entity, comprising:
destination_address, source_address, service_data_unit
Where *service_data_unit* comprises:
extraTime, function, grandTime, hopCount,
precedence, protocolType, snapTime, syncInterval, version
rxPtr
A pointer to *rxInfo* storage.
rxSyncInterval
A variable that represents the sync-interval associated with this station's clock-slave port.
stationTime
See 6.3.3.
ssPtr
A pointer to the service-data-unit portion of the *ePtr->rxSaved* storage
sxPtr
A pointer to the *ePtr->rxSaved* storage.
tsPtr
A pointer to service-data-unit portion of *txInfo* storage.
txInfo
Storage for to-be-transmitted time-sync PDUs, comprising:
destination_address, source_address, service_data_unit
Where *service_data_unit* comprises:
extraTime, function, frameCount, grandTime, hopCount, localTime,
precedence, protocolType, thatRxTime, thatTxTime, version
txPtr
A pointer to *txInfo* storage.
txSyncInterval
A variable that represents the sync-interval associated with this clock-master port.

8.4.4 State machine routines

Dequeue(*queue*)
Enqueue(*queue, info*)
See 6.3.4.
NextSaved(*btPtr, rateInterval, grandTime, extaTime, thisTime*)
NextTimed(*btPtr, stationTime, backInterval*)
See 7.5.4.

StationTime(*entity*)
See 7.3.3.
TimeSyncSdu(*info*)
See 6.3.4.

8.4.5 TimeSyncTxEfdx state machine table

The TimeSyncTxEfdx state machine includes a media-dependent timeout, which effectively disconnects a clock-slave port in the absence of received timeSync frames, as illustrated in Table 8.3.

Table 8.3—TimeSyncTxEfdx state machine table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(rxInfo = Dequeue(Q_MS_REQ)) != NULL	1	—	TEST
	(stationTime – ePtr->lastTime) > T10ms	2	ePtr->lastTime = stationTime;	SEND
	(cxInfo = Dequeue(Q_TX_SYNC)) != NULL	3	ePtr->txSnapTime = cxPtr->snapTime; ePtr->txSnapCount = cxPtr->frameCount;	START

	—	4	stationTime = StationTime(ePtr);
			rxSyncInterval = ssPtr->syncInterval;
			txSyncInterval = ePtr->syncInterval;
			backInterval =
			(3 * rxSyncInterval + txSyncInterval) / 2;
			rateInterval =
			backInterval + (3 * txSyncInterval) / 2;	

TEST	TimeSyncSdu(rsPtr)	5	ePtr->rxSaved = rxInfo; NextSaved(btPtr, rateInterval,rsPtr->grandTime, rsPtr->extraTime, rsPtr->snapTime);	START
	—	6	Enqueue(Q_ES_REQ, rxPtr);	

SEND	—	7	dPtr = PortPair(ePtr);
			nextTimes =
			NextTimed(btPtr, stationTime, backInterval);
			ePtr->syncInterval, ePtr->timed);
			ePtr->txFrameCount =
			(ePtr->txSnapCount + 1) % COUNT;
			txPtr->destination_address =
			sxPtr->destination_address;
			txPtr->source_address = sxPtr->source_address;
			tsPtr->protocolID = ssPtr->protocolID;
			tsPtr->function = ssPtr->function;
			tsPtr->version = ssPtr->version;
			tsPtr->hopCount = ssPtr->hopCount;
			tsPtr->frameCount = ssPtr->frameCount;
			tsPtr->grandTime = nextTimes.grandTime;
			tsPtr->extraTime = nextTimes.extraTime;
			tsPtr->localTime = ePtr->txSnapTime;
			tsPtr->thatTxTime = dPtr->thisTxTime;
			tsPtr->thatRxTime = dPtr->thisRxTime;
			Enqueue(Q_ES_REQ, txPtr);
				START

Row 8.3-1: Relayed frames are further checked before being processed.
Row 8.3-2: Transmit periodic timeSync frames.
Row 8.3-3: Update snapshot values on timeSync frame departure.
Row 8.3-4: Wait for the next change-of-state.

Row 8.3-5: The timeSync PDUs are checked further.
Row 8.3-6: The non-timeSync PDUs are passed through.

Row 8.3-7: Active timeSync frames are cable-delay compensated and passed through.

9. Wireless state machines

EDITOR DVJ NOTE—This clause is based on indirect knowledge of the 802.11v specifications, as interpreted by the author, and have not been reviewed by the 802.1 or 802.11v WGs. The intent was to provide a forum for evaluation of the media-independent MAC-relay interface, while also triggering discussion of 802.11v design details. As such, this clause is highly preliminary and subject to change.
Specifically, we have not resolved the grouping of information that is transferred through the service interfaces (currently written as all) and the information that would be transferred through standard MAC frames (currently written as none).

9.1 Overview

This clause specifies the state machines that support wireless 802.11v-based bridges. The operations are described in an abstract way and do not imply any particular implementations or any exposed interfaces. There is not necessarily a one-to-one correspondence between the formal specification and the interfaces in any particular implementation.

9.1.1 Link-dependent indications

The wireless 802.11v TimeSyncR11v state machines are provided with MAC service-interface parameters, as illustrated within Figure 9.1. These link-dependent indications can be different for bridge ports attached to alternative media.

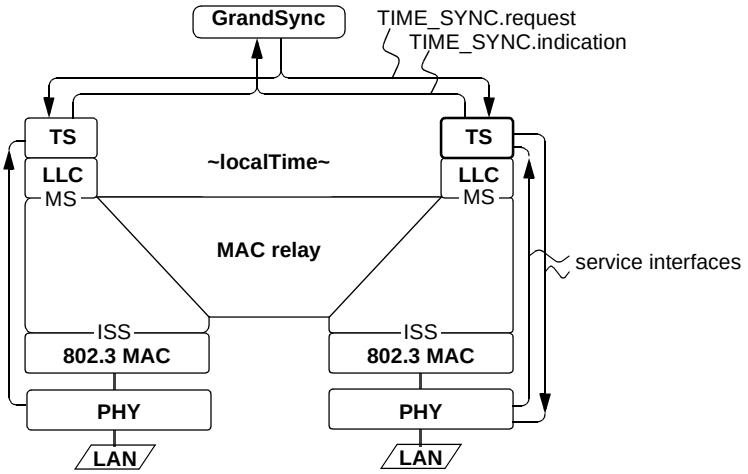

Figure 9.1—R11v interface model

The rxSync and txSync indications are localized communications between the MAC-and-PHY and are not directly visible to the a TimeSync state machines. Client-level interface parameters include the timing information, based on the formats illustrated within Figure 9.2.

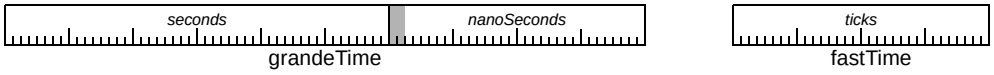

Figure 9.2—Formats of wireless-dependent times

9.1.2 Service interface overview

A sequence of 802.11v TimeSync service interface actions is illustrated in Figure 9.3 and summarized below:

a) A periodic clock-slave trigger initiates the initial MLME_PRESENCE_REQUEST.request action.

b) The clock-master gets an MLME_PRESENCE_REQUEST.indication upon request receipt.
The clock-slave gets an MLME_PRESENCE_REQUEST.confirm when the ack is returned.

c) The clock-master processes the MLME_PRESENCE_REQUEST.indication parameters, returning them in MLME_PRESENCE_RESPONSE.request parameters for the clock-slave station.

d) The clock-slave gets an MLME_PRESENCE_RESPONSE.indication upon response receipt.
The clock- master gets an MLME_PRESENCE_RESPONSE.confirm when the ack is returned.

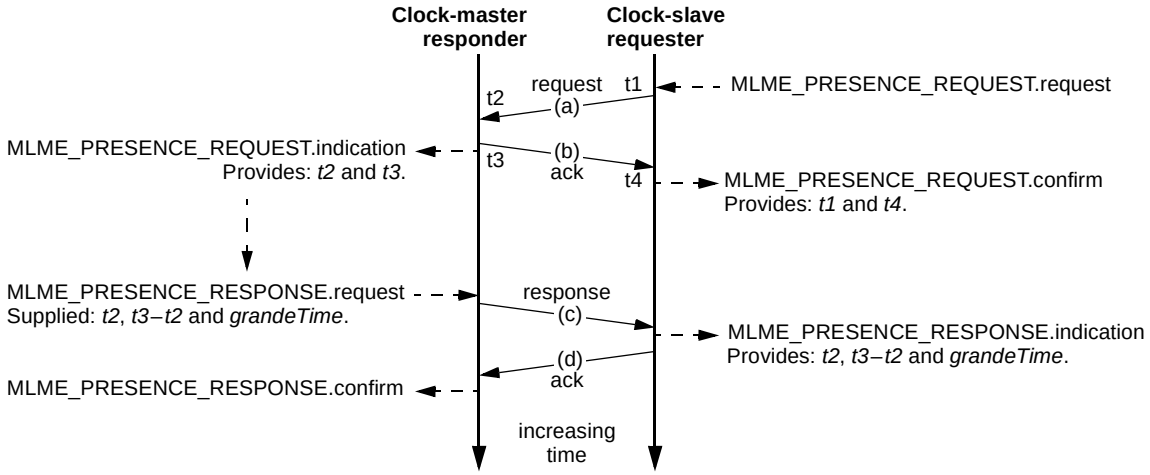

Figure 9.3—802.11v time-synchronization interfaces

The properties of these service interfaces are summarized below:

MLME_PRESENCE_REQUEST.request
Generated periodically by the clock-slave entity.
Triggers a (Figure 9.3a) request to fetch clock-master resident timing parameters.
MLME_PRESENCE_REQUEST.indication
Generated after receiving a (Figure 9.3a) request.
Provides *t2* and *t3* timing information to the clock-master entity.
MLME_PRESENCE_REQUEST.confirm
Generated after the (Figure 9.3b) request-ack is returned.
Provides *time1* and *time4* timing information to the clock-slave entity.
Confirms completion of the request transmission.

MLME_PRESENCE_RESPONSE.request
Generated shortly after processing a received (Figure 9.3a) request
Triggers a (Figure 9.3-c) response to update clock-slave resident timing parameters.
MLME_PRESENCE_RESPONSE.indication
Generated in response to receiving a (Figure 9.3c) response.
Provides *time2*, *time3–time2*, and *grandeTime* information to the clock-slave entity.
MLME_PRESENCE_RESPONSE.confirm
Generated after the (Figure 9.3d) ack is returned.
Confirms completion of the response transmission.

9.2 Service interface definitions

9.2.1 MLME_PRESENCE_REQUEST.request

9.2.1.1 Function

The service interface triggers the sending of a (Figure 9.3a) request from the clock-slave requester to the clock-master responder. A snapshot of the transmit time is also saved for deferred transmission/processing.

9.2.1.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
MLME_PRESENCE_REQUEST.request {
    other_arguments      // Arguments for other purposes
}
```

9.2.1.3 When generated

Generated periodically by a receive TS port as the first phase of a time-sync information transfer.

9.2.1.4 Effect of receipt

Upon receipt by a transmit TS port, an MLME_PRESENCE_REQUEST.indication is invoked; times of the arriving (Figure 9.3a) request and departing (Figure 9.3b) request-ack are both passed within this indication.

9.2.2 MLME_PRESENCE_REQUEST.indication

9.2.2.1 Function

The receipt of a (Figure 9.3a) request from the clock-slave requester triggers the return of an (Figure 9.3b) request-ack from the clock-master port. The transfer of an MLME_PRESENCE_REQUEST.indication to the clock-master provides snapshots of the (Figure 9.3a) request-receipt time as well as the following (Figure 9.3b) ack-transmit time.

9.2.2.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
MLME_PRESENCE_REQUEST.indication {
    other_arguments,     // Arguments for other purposes
    time_t2,             // Arrival time of request
    time_t3              // Departure time of request-ack
}
```

9.2.2.3 When generated

Generated by the receipt of a (Figure 9.3a) request during the first phase of a time-sync transfer.

9.2.2.4 Effect of receipt

Upon receipt, the times of the arriving (Figure 9.3a) request and (Figure 9.3b) request-ack are both saved for deferred processing.

9.2.3 MLME_PRESENCE_REQUEST.confirm

9.2.3.1 Function

The receipt of a (Figure 9.3b) request-ack at the clock-slave requester triggers the invocation of the MLME_PRESENCE_REQUEST.confirm service interface. The transmit time of the original (Figure 9.3a) request and the receive time of the recent (Figure 9.3b) request-ack are both provided.

9.2.3.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
MLME_PRESENCE_REQUEST.indication {
    other_arguments,    // Arguments for other purposes
    time_t1,            // Departure time of request
    time_t4             // Arrival time of confirm
}
```

9.2.3.3 When generated

Generated by the receipt of a (Figure 9.3b) request-ack during the initial phases of a time-sync transfer.

9.2.3.4 Effect of receipt

Upon receipt, the transmit time of the previous (Figure 9.3a) request and receive time of the recent (Figure 9.3b) request-ack are both saved for deferred processing.

9.2.4 MLME_PRESENCE_RESPONSE.request

9.2.4.1 Function

After the initial phases, a clock-slave requester triggers the transfer of an MLME_PRESENCE_RESPONSE.request. The transmit time of the original (Figure 9.3a) request, the transmit time of the recent (Figure 9.3b) request-ack, and the current time-sync related information are all included in the service primitives.

9.2.4.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
MLME_PRESENCE_RESPONSE.request {
    other_arguments,    // Arguments for other purposes
    time_t2             // Arrival time of request
    time_t32,           // Turn-round time
    grande_time         // Current media-dependent time
}
```

9.2.4.3 When generated

Triggered at the clock-master by the servicing of an MLME_PRESENCE_REQUEST.indication, indicating the completion of the initial time-sync phase.

9.2.4.4 Effect of receipt

Upon receipt, an MLME_PRESENCE_RESPONSE.indication is invoked, to provide the clock-slave with sufficient information to send a GrandSync PDU.

9.2.5 MLME_PRESENCE_RESPONSE.indication

9.2.5.1 Function

Additional information is provided to a clock-slave port. Along with previous information (saved earlier for deferred processing), the clock-slave has sufficient information to send a GrandSync PDU.

9.2.5.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
MLME_PRESENCE_RESPONSE.indication {
    other_arguments,    // Arguments for other purposes
    time_t2             // Arrival time of request
    time_t32,           // Turn-round time
    level_time,         // Current media-dependent time
}
```

9.2.5.3 When generated

Triggered at the clock-slave by the receipt of a (Figure 9.3c) response, nearing the completion of the final time-sync phases.

9.2.5.4 Effect of receipt

Upon receipt, the clock-slave is provided with sufficient information to send a GrandSync PDU.

9.2.6 MLME_PRESENCE_RESPONSE.confirm

9.2.6.1 Function

Confirmation is provided to the clock-master, confirming clock-slave has sufficient information to send a GrandSync PDU.

9.2.6.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
MLME_PRESENCE_RESPONSE.confirm {
    other_arguments,    // Arguments for other purposes
}
```

9.2.6.3 When generated

Triggered at the clock-master by the receipt of a (Figure 9.3-d) response-ack, at the completion of the final time-sync phases.

9.2.6.4 Effect of receipt

Upon receipt, the clock-master is provided with a time-sync success status.

9.3 TimeSyncRxR11v state machine

9.3.1 Function

The TimeSyncRxR11v state machine consumes primitives provided by the MAC service interface and (in response) generates frames for the GrandSync entity.

9.3.2 State machine definitions

NULL
A constant indicating the absence of a value that (by design) cannot be confused with a valid value.

queue values
Enumerated values used to specify shared FIFO queue structures.
- Q_MS_IND—Queue identifier associated with the GrandSync receive port.
- Q_S1_REQ—Queue identifier for MLME_PRESENCE_REQUEST.request parameters.
- Q_S1_CON—Queue identifier for MLME_PRESENCE_REQUEST.confirm parameters.
- Q_S2_IND—Queue identifier for MLME_PRESENCE_RESPONSE.indication parameters.

9.3.3 State machine variables

backTime
A variable representing the lapsed time since the remote request-ack transmission.

con1
A set of values returned within the MLME_PRESENCE_REQUEST.request service primitive:
- *time_t1*—A local-timer snapshot at the (Figure 9.3a) request transmission.
- *time_t4*—A local-timer snapshot at the (Figure 9.3b) request-ack reception.

ind2
A set of values returned within the MLME_PRESENCE_RESPONSE.indication service primitive:
- *grandeTime*—A remote snapshot of grandTime at the request-ack transmission.
- *time_t2*—A remote-timer snapshot at the (Figure 9.3a) request reception.
- *time_t3*—A remote-timer snapshot at the (Figure 9.3b) request-ack transmission.

ePtr
Points to entity-specific storage, comprising the following:
- *lastTime*—The time of the last request transmission, for pacing periodic transmissions.
- *roundTrip*—Saved (*con1.time_t4*−*con1.time_t4*) value.
- *rsInfo*—Saved grand-master selection values.
- *rxFastTimed*—Saved *args2.fastTimed* value.
- *snapTime*—Saved *con1.time_t4* value.
- *syncInterval*—The sync-interval associated with this clock-slave port.

grandTime
An variable representing the normalized/synchronized grand-master time.

lapseTime
An variable representing the lapsed time since the request-ack reception.

localTime
A variable representing the calibrated one-way link delay.

radioDelay
An variable representing the round-trip transmission delay.

radioTime
An variable representing the current time, in media-specific units.

req1

A set of values returned within the MLME_PRESENCE_REQUEST.request service primitive, consisting of other (unrelated) parameters.

stationTime

See 6.3.3.

rsPtr

A pointer to the *rsInfo* portion of *ePtr* referenced storage.

turnRound

An variable representing the difference between local time-sync transmit and receive times.

turnStart

An variable representing the remote time-sync transmit time.

txInfo

Storage for information sent to the GrandSync entity, comprising:

destination_address, *source_address*, *service_data_unit*

Where *service_data_unit* comprises:

extraTime, *sourcePort*, *function*, *grandTime*, *hopCount*, *localTime*, *protocolType*, *precedence*, *syncInterval*, *version*

txPtr

A pointer to *txInfo* storage.

9.3.4 State machine routines

Dequeue(*queue*)

Enqueue(*queue, info*)

See 6.3.4.

R11vTime(*entity*)

Returns the local media-dependent free-running timer.

SourcePort(*entity*)

See 7.3.3.

StationTime(*entity*)

TimeSyncSdu(*info*)

See 6.3.4.

9.3.5 TimeSyncRxR11v state table

The TimeSyncRxR11v state machine consumes MAC-provided service-primitive information and forwards adjusted frames to the MAC-relay function, as illustrated in Table 9.1.

Table 9.1—TimeSyncRxR11v state machine table

Current		Row	Next	
state	**condition**		**action**	**state**

START	(stationTime – ePtr->lastTime) > ePtr->syncInterval	1	ePtr->lastTime = stationTime;
			req1 = SetupReq1();
			Enqueue(Q_S1_REQ, req1);
				START

	(con1 = Dequeue(Q_S1_CON)) != NULL	2	ePtr->snapTime = con1.time_t4; ePtr->roundTrip = con1.time_t4 – con1.time_t1;	
	(ind2 = Dequeue(Q_S2_IND)) != NULL	3	turnStart = rxPtr->time_t2; turnRound = rxPtr->time_t32;	SINK
	—	4	stationTime = StationTime(ePtr); radioTime = R11vTime(ePtr);	START

SINK	—	5	linkDelay= (ePtr->roundTrip−turnRound) / 2;
			lapseDelay= (radioTime – ePtr->snapTime);
			backTime =
			R11vToStation(lapseDelay + linkDelay);
			grandTime =
			GrandeToGrand(rxptr->grandeTime);
			txPtr->destination_address = AVB_MCAST;
			txPtr->source_address = TBD;
			tsPtr->protocolType = AVB_TYPE;
			tsPtr->function = AVB_FUNCTION;
			tsPtr->version = AVB_VERSION;
			tsPtr->precedence = rsPtr->precedence;
			tsPtr->sourcePort = SourcePort(ePtr);
			tsPtr->hopCount = rsPtr->hopCount;
			tsPtr->grandTime = grandTime;
			tsPtr->extraTime = rsPtr->extraTime;
			tsPtr->snapTime = stationTime – backTime;
			Enqueue(Q_MS_IND, txPtr);
				START

Row 9.1-1: Requests are sent at a periodic rate.
Row 9.1-2: Save the times that are available when the request-ack returns.
Row 9.1-3: Capture the parameters when the MLME_PRESENCE_RESPONSE.indication returns.
Row 9.1-4: Update times while waiting for state changes.

Row 9.1-5: Send accumulated/supplemented information to the GrandSync entity.

9.4 TimeSyncTxR11v state machine

9.4.1 Function

The TimeSyncTxR11v state machine consumes GrandSync-generated frames, to maintain estimates of the current {*grandTime,stationTime*} and {*errorTime,stationTime*} affiliations. The TimeSyncTxR11v state machine also provides time-synchronization information through the MAC service interface, in response to clock-slave initiated requests.

9.4.2 State machine definitions

NULL
A constant indicating the absence of a value that (by design) cannot be confused with a valid value.
queue values
Enumerated values used to specify shared FIFO queue structures.
Q_MS_REQ—The queue identifier associated with frames sent from the GrandSync entity.
Q_S1_IND—Queue identifier for MLME_PRESENCE_REQUEST.indication parameters.
Q_S2_REQ—Queue identifier for MLME_PRESENCE_RESPONSE.request parameters.
Q_S2_ACK—Queue identifier for MLME_PRESENCE_RESPONSE.confirm parameters.

9.4.3 State machine variables

backInterval
A variable that represents the back-interpolation interval for transmit-time affiliations.
btPtr
A pointer to the ePtr->baseTimer storage.
con2
Unrelated values returned within the MLME_PRESENCE_RESPONSE.confirm service primitive:
ePtr
A pointer to the entity-specific storage containing the following:
baseTimer—Recently saved time events, each consisting of the following:
index—Index into the *timed*[ ] array, where last times were stored.
range—Number of entries within the *timed*[ ] array
timed[range]—Recently saved time events, each consisting of the following:
grandTime—A previously sampled grand-master synchronized time.
extraTime—The residual error associated with the sampled *grandTime* value.
stationTime—The station-local time affiliated with the sampled *grandTime* value.
lastTime—The last transmit time, saved for pacing transmissions.
rxSaved—A copy of the last received GrandSync parameters.
syncInterval—The expected interval between successive time-sync transmissions.
ind1
Values returned within the MLME_PRESENCE_REQUEST.indication service primitive:
time_t2—A local snapshot at the time of (Figure 9.3a) request reception.
time_t3—A local snapshot at the time of (Figure 9.3b) request-ack transmission.
radioTime
A variable representing the media-dependent station-local time.
rateInterval
A variable representing the time interval over which the grandTime rate is measured.
req2
Values provided to the MLME_PRESENCE_REQUEST.request service primitive:
grandeTime—A local snapshot of the grandTime as the request-ack transmission.
time_t2—Previously saved *ind1.time_t2* value.
time_t32—Previously saved (*ind1.time_t3–ind1.time_t2*) value.

rsPtr
A pointer to service-data-unit portion of *rxInfo* storage.
rxInfo
Storage for received time-sync PDU from the GrandSync entity, comprising:
destination_address, source_address, service_data_unit
Where *service_data_unit* comprises:
extraTime, function, grandTime, hopCount,
precedence, protocolType, snapTime, syncInterval, version
rxPtr
A pointer to *rxInfo* storage.
rxSyncInterval
A variable that represents the sync-interval associated with this station's clock-slave port.
ssPtr
A pointer to the service-data-unit portion of the *ePtr->rxSaved* storage
stationTime
A shared value representing current time. There is one instance of this variable for each station. Within the state machines of this standard, this is assumed to have two components, as follows:
seconds—An 8-bit unsigned value representing seconds.
fraction—An 40-bit unsigned value representing portions of a second, in units of 2^{-40} second.
sendTime
A variable representing the local time estimate of the remote request-ack transmission time.
sxPtr
A pointer to the *ePtr->rxSaved* storage.
timeT2
A variable that represents the request receipt time.
timeT3
A variable that represents the request-ack transmit time.
txSyncInterval
A variable that represents the sync-interval associated with this port.

9.4.4 State machine routines

Dequeue(*queue*)
Enqueue(*queue, info*)
See 6.3.4.
NextSaved(*btPtr*, *rateInterval*, *grandTime*, *extaTime*, *thisTime*)
See 7.5.4.
NextTimed(*btPtr, stationTime, backInterval*)
See 7.5.4.
StationTime(*entity*)
See 7.3.3.
TimeSyncSdu(*info*)
See 6.3.4.

9.4.5 TimeSyncTxR11v state table

NOTE—This state machine is preliminary; sequence timeouts have not been considered.

The TimeSyncTxR11v state machine includes a media-dependent timeout, which effectively disconnects a clock-slave port in the absence of received timedSync frames, as illustrated in Table 9.2.

Table 9.2—TimeSyncTxR11v state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(rxInfo = Dequeue(Q_MS_REQ)) != NULL	1	—	SINK
	(ind1 = Dequeue(Q_S1_IND)) != NULL	2	timeT2 = ind1.time_t2; timeT3 = ind1.time_t3;	SEND
	(con2 = Dequeue(Q_S2_CON)) != NULL	3	—	START

	—	4	stationTime = StationTime(ePtr);
			radioTime = R11vTime(ePtr);
			rxSyncInterval = ssPtr->syncInterval;
			txSyncInterval = ePtr->syncInterval;
			backInterval =
			(3 * rxSyncInterval + txSyncInterval) / 2;
			rateInterval =
			backInterval + (3 * txSyncInterval) / 2;
				START

SINK	TimeSyncSdu(rsPtr)	5	ePtr->rxSaved = rxInfo; NextSaved(btPtr, rateInterval, rsPtr->grandTime, rsPtr->extraTime, stationTime);	SERVE
	—	6	Enqueue(Q_ES_REQ, rxPtr);	START

SEND	—	7	sendTime = stationTime –
			((radioTime – timeT2) * RADIO_TIME);
			nextTimes =
			NextTimed(btPtr, stationTime, backInterval);
			req2.time_t2 = timeT2;
			req2.time_t32 = timeT3 – timeT2;
			req2.grandeTime =
			GrandToR11v(nextTimes.grandTime);
			// If possible for extraTime.
			req2.extraTime = nextTimes.extraTime;
			Enqueue(Q_S2_REQ, req2);
				WAIT2

Row 9.2-1: GrandSync generated frames are further checked before being processed.
Row 9.2-2: Save parameters from a service-interface primitive call.
Row 9.2-3: The final acknowledge provides a completion indication.
Row 9.2-4: Wait for the next change-of-state.

Row 9.2-5: Parameters from timeSync PDUs are saved.
Row 9.2-6: The contents of non-timeSync PDUs are passed through.

Row 9.2-7: Provide parameters for the MLME_PRESENCE_RESPONSE.response interface.

10. Ethernet passive optical network (EPON) state machines

NOTE—This clause is based on indirect knowledge of the Ethernet-PON specifications, as interpreted by the author, and have not been reviewed by the 802.1 or 802.3 WGs. The intent was to provide a forum for evaluation of the GrandSync interfaces, while also triggering discussion of 802.3-PON design details. As such, the contents are highly preliminary and subject to change.

10.1 Overview

This clause specifies the state machines that support Ethernet passive optical network (EPON) based bridges. The operations are described in an abstract way and do not imply any particular implementations or any exposed interfaces. There is not necessarily a one-to-one correspondence between the formal specification and the interfaces in any particular implementation.

10.1.1 Link-dependent indications

The TimeSyncEpon state machines have knowledge of network-local synchronized *ticksTime* timers. With this knowledge, the TimeSyncEpon state machines can operated on frames received from the LLC, as illustrated in Figure 10.1. Link-dependent indications could be required for bridge ports attached to alternative media.

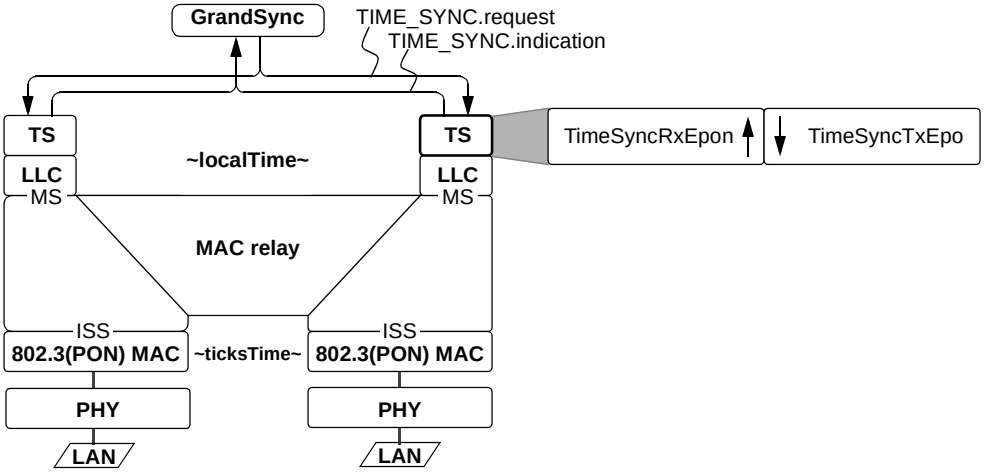

Figure 10.1—PON interface model

The *localTime* values are represented as timers that are incremented once every 16 ns interval, as illustrated on the left side of Figure 10.2. Each synchronized local timer is roughly equivalent to a 6-bit *sec* (seconds) field and a 26-bit *fraction* (fractions of second) field timer, as illustrated on the right side of Figure 10.2.

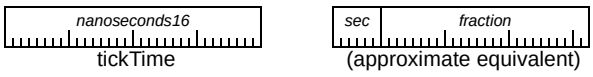

Figure 10.2—Format of PON-dependent times

The Ethernet-PON MAC is supplied with frame transmit/receive snapshots, but these are transparent-to and not-used-by the TimeSync state machine. Instead, these are used to synchronize the *ticksTime* values in associated MACs and the TimeSyncEpon state machines have access to these synchronized *ticksTime* values.

10.1.2 Link-delay compensation

The synchronized-clock accuracies are influenced by the transmission delays between ports. To compensate for these transmission delays, the receive port is normally responsible for compensating {*grandTime, ticksTime*} affiliations by the (assumed to be constant) frame transmission delay.

The Ethernet-PON MAC provides access to a subnet-synchronized media-dependent *ticksTime* timer. Thus, the {*grandTime, ticksTime*} affiliation specified the transmitter remains valid within the receiver and transmission-delay compensation (in this sense) is unnecessary.

However, each time-sync related GrandSync PDU includes an {*grandTime, stationTime*} affiliation, wherein *stationTime* represents a recent snapshot of a shared station-local clock. To provide such an affiliation, the transmission delay (measured as a *ticksTime* difference) is scaled and subtracted from the *stationTime* that is sampled when the conversion is performed. Thus, no additional receiver snapshot hardware is required.

10.2 timeSyncEpon frame format

The timeSyncEpon frames facilitate the synchronization of neighboring clock-master and clock-slave stations. The frame, which is normally sent at 10 ms intervals, includes time-snapshot information and the identity of the network's clock master, as illustrated in Figure 10.3. The gray boxes represent physical layer encapsulation fields that are common across Ethernet frames.

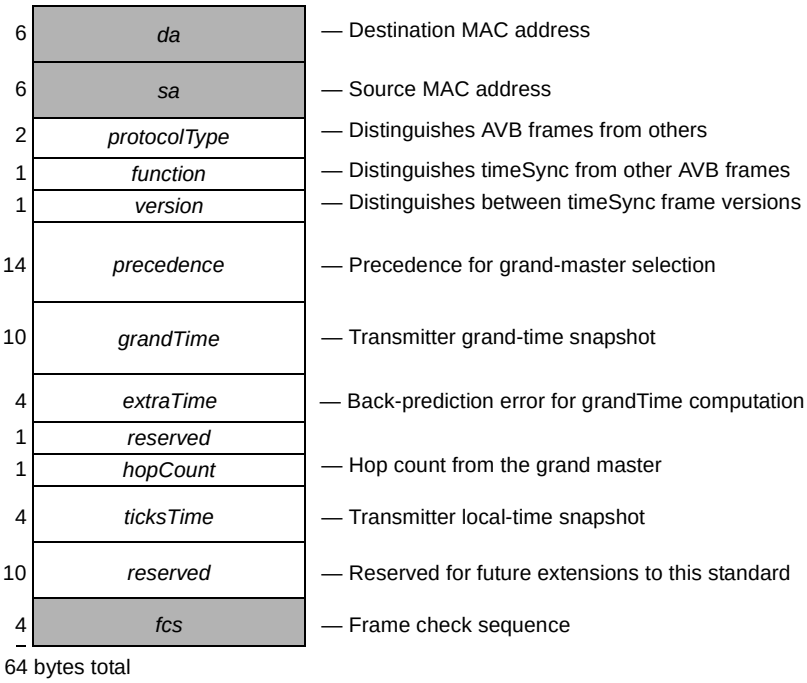

Figure 10.3—timeSyncEpon frame format

The 48-bit ***da*** (destination address), 48-bit ***sa*** (source address) field, 16-bit ***protocolType,*** 8-bit ***function***, 8-bit ***version***, 14-byte ***precedence***, 80-bit ***grandTime***, 32-bit ***extraTime***, and 8-bit ***hopCount*** fields are specified in 6.2.1.2.

10.2.1 *ticksTime*: A value representing local time in units of a 16 ns timer ticks, as illustrated in Figure 10.4.

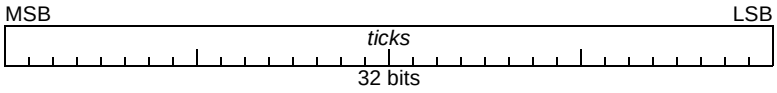

Figure 10.4—tickTime format

10.3 TimeSyncRxEpon service interface primitives

10.3.1 ES_UNITDATA.indication

10.3.1.1 Function

Provides the TimeSyncRxEpon entity with clock-synchronization parameters derived from arriving time-sync frames.

10.3.1.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
ES_UNITDATA.indication {
    destination_address,      // Destination address
    source_address,           // Optional
    priority,                 // Forwarding priority
    service_data_unit,        // Delivered content
    {                         // Contents of the service_data_unit
        protocolType,         // Distinguishes AVB frames from others
        function,             // Distinguishes between timeSync and other AVB frames
        version,              // Distinguishes between timeSync frame versions
        precedence,           // Precedence for grand-master selection
        grandTime,            // Global-time snapshot (1-cycle delayed)
        extraTime,            // Accumulated grandTime error
        hopCount,             // Distance from the grand-master station
        ticksTime             // Local-time snapshot (1-cycle delayed)
    }
}
```

The parameters of the MA_DATA.indication are described as follows:

The 48-bit **destination_address**, 48-bit **source_address**, and 8-bit **priority** field are specified in 6.2.1.2.

The service_data_unit consists of subfields; for content exchanged with the GrandTime protocol entity, these fields include the following.

The 16-bit ***protocolType***, 8-bit ***function***, 8-bit ***version***, 14-byte ***precedence***, 80-bit ***grandTime***, 32-bit ***extraTime***, and 8-bit ***hopCount*** fields are specified in 6.2.1.2.

10.3.1.2.1 *frameCount*: An 8-bit consistency-check field that increments on successive frames.

10.3.1.2.2 *ticksTime*: A 32-bit field that specifies the local free-running time within this subnet, when the previous timeSync frame was received (see 10.2.1).

10.3.1.3 When generated

The service primitive is generated upon the receipt of a time-sync related frame delivered from the MAC. The intent is to facilitate reformatting and snapshot-time adjustment before the content of that frame is delivered to the ClockMaster and TS entities.

10.3.1.4 Effect of receipt

The service primitive invokes processing of time-sync related content and forwarding of unrelated content. For time-sync related content, the processing included reformatting and compensation for receive-link transmission delays.

10.4 TimeSyncRxEpon state machine

10.4.1 Function

The TimeSyncRxEpon state machine is responsible for receiving MAC-supplied frames, converting their media-dependent parameters, and sending normalized MAC-relay frames. The sequencing of this state machine is specified by Table 10.1; details of the computations are specified by the C-code of Annex F.

10.4.2 State machine definitions

NULL
A constant indicating the absence of a value that (by design) cannot be confused with a valid value.
queue values
Enumerated values used to specify shared FIFO queue structures.
Q_MS_IND—Associated with the GrandSync entity (see 6.3.2).
Q_ES_IND—The queue identifier associated with the received MAC frames.

10.4.3 State machine variables

ePtr
A pointer to a entity-specific data structure comprising the following:
syncInterval—The expected interval between time-sync frame transmissions.
backTime
A value representing the time lapse between transmission of reception of the timeSync frame.
rsPtr
A pointer to the service-data-unit portion of the *rxInfo* storage.
rxInfo
A storage location for received service-interface parameters, comprising:
destination_address, *source_address*, *service_data_unit*
Where *service_data_unit* comprises:
extraTime, *function*, *grandTime*, *hopCount*,
precedence, *protocolType*, *ticksTime*, *version*
rxPtr
A pointer to the *rxInfo* storage location.
tsPtr
A pointer to the service-data-unit portion of the *txInfo* storage.
txInfo
A storage location for to-be-transmitted TIME_SYNC.indication parameters, comprising:
destination_address, *source_address*, *service_data_unit*
Where *service_data_unit* comprises:
extraTime, *function*, *grandTime*, *hopCount*, *precedence*,
protocolType, *snapTime*, *ticksTime*, *version*
txPtr
A pointer to the *txInfo* storage location.

10.4.4 State machine routines

Dequeue(*queue*)
Enqueue(*queue*, *info*)
See 6.3.4.
SourcePort(*entity*)
See 7.3.3.
TicksTime(*entity*)
Returns the value of the station’s shared media-dependent subnet-synchronized timer.
This 32-bit timer is incremented once at the end of each 16 ns interval.
TicksToTime(*ticks*)
Returns the time duration of *stationTime* that corresponds to the time duration specified in *ticks*.
TimeSyncSdu(*info*)
See 6.3.4.

10.4.5 TimeSyncRxEpon state machine table

The TimeSyncRxEpon state machine associates PHY-provided sync information with arriving timeSync frames and forwards adjusted frames to the MAC-relay function, as illustrated in Table 8.2.

Table 10.1—TimeSyncRxEpon state machine table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(rxInfo = Dequeue(Q_RX_MAC)) **!**= NULL	1	—	TEST
	—	2	stationTime = StationTime(ePtr); ticksTime = EponTime(ePtr);	START
TEST	TimeSyncSdu(rsPtr)	3	*rxPtr = rxInfo;	SYNC
	—	4	Enqueue(Q_MS_IND, txInfo);	START

SYNC	rsPtr->hopCount **!**= LAST_HOP	5	backTime = ticksTime – rsPtr->ticksTime);
			compTime =
			stationTime –TicksToTime(backTime);
			txPtr->destination_address =
			rxPtr->destination_address;
			txPtr->source_address = rxPtr->source_address;
			tsPtr->protocolType = rsPtr->protocolType;
			tsPtr->function = rsPtr->function;
			tsPtr->version = rsPtr->version;
			tsPtr->precedence = rsPtr->precedence;
			tsPtr->grandTime = rsPtr->grandTime;
			tsPtr->extraTime = rsPtr->extraTime;
			tsPtr->snapTime = compTime;
			tsPtr->sourcePort = SourcePort(ePtr);
			tsPtr->hopCount = frame.hopCount;
			tsPtr->syncInterval = ePtr->syncInterval;
			Enqueue(Q_MS_IND, txInfo);
				START

	—	6	—	

Row 10.1-1: Initiate inspection of frames received from the lower-level MAC.
Row 10.1-2: Wait for the next frame to arrive.

Row 10.1-3: The timeSync frames are checked further.
Row 10.1-4: The non-timeSync frames are passed through.

Row 10.1-5: Active timeSync frames are adjusted for transfer delays and passed through.
Row 10.1-6: Overly-aged timeSync frames are discarded.

10.5 TimeSyncTxEpon service interface primitives

10.5.1 ES_UNITDATA.request

10.5.1.1 Function

Provides the Ethernet-PON entity with clock-synchronization parameters for constructing departing time-sync frames.

10.5.1.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
ES_UNITDATA.request
{
    destination_address,      // Destination address
    source_address,           // Optional
    priority,                 // Forwarding priority
    service_data_unit,        // Delivered content
    {                         // Contents of the service_data_unit
        protocolType,         // Distinguishes AVB frames from others
        function,             // Distinguishes between timeSync and other frames
        version,              // Distinguishes between timeSync frame versions
        precedence,           // Precedence for grand-master selection
        grandTime,            // Global-time snapshot (1-cycle delayed)
        extraTime,            // Accumulated grandTime error
        hopCount,             // Distance from the grand-master station
        ticksTime             // Local-time snapshot (1-cycle delayed)
    }
}
```

The parameters of the MA_UNITDATA.request are described in 10.3.1.2.

10.5.1.3 When generated

The service primitive is generated at a periodic rate, for the purposes of synchronizing the *grandTime* values resident in other stations.

10.5.1.4 Effect of receipt

The service primitive triggers the transmission of a timeSync frame on the affiliated port.

10.6 TimeSyncTxEpon state machine

10.6.1 Function

The TimeSyncTxEpon state machine is responsible for modifying time-sync TIME_SYNC.request parameters to form timeSync frames for transmission over the attached link.

10.6.2 State machine definitions

NULL
A constant indicating the absence of a value that (by design) cannot be confused with a valid value.
queue values
Enumerated values used to specify shared FIFO queue structures.
Q_MS_REQ—Associated with the GrandSync entity (see 6.3.2).
Q_ES_REQ—The queue identifier associated with frames sent to the MAC.
T10ms
A constant the represents a 10 ms value.

10.6.3 State machine variables

backInterval
A variable that represents the back-interpolation interval for transmit-time affiliations.
ePtr
A pointer to a entity-specific data structure comprising the following:
baseTimer—Recently saved time events, each consisting of the following:
index—Index into the *timed*[ ] array, where last times were stored.
range—Number of entries within the *timed*[ ] array
timed[range]—Recently saved time events, each consisting of the following:
grandTime—A previously sampled grand-master synchronized time.
extraTime—The residual error associated with the sampled *grandTime* value.
stationTime—The station-local time affiliated with the sampled *grandTime* value.
lastTime—The last PDU-transmit time; used to space periodic transmissions.
rxSaved—A copy of the last received GrandSync parameters.
syncInterval—The expected interval between time-sync frame transmissions.
rsPtr
A pointer to the service-data-unit portion of *rxInfo* storage.
rxInfo
Storage for the contents of GrandSync PDUs, comprising:
destination_address, *source_address*, *service_data_unit*
Where *service_data_unit* comprises:
extraTime, *function*, *grandTime*, *hopCount*, *precedence*,
protocolType, *snapTime*, *syncInterval*, *version*
rxPtr
A pointer to the *rxInfo* storage.
rxSyncInterval
A variable that represents the sync-interval associated with this station’s clock-slave port.
stationTime
See 6.3.3.
ssPtr
A pointer to the service-data-unit portion of the *ePtr->rxSaved* storage
sxPtr
A pointer to the *ePtr->rxSaved* storage.
tsPtr
A pointer to the service-data-unit portion of *txInfo* storage.

txInfo

Storage for a to-be-transmitted MAC frame, comprising:

destination_address, *source_address*, *service_data_unit*

Where *service_data_unit* comprises:

extraTime, *function*, *grandTime*, *hopCount*,
protocolType, *precedence*, *ticksTime*, *version*

txPtr

A pointer to the *txInfo* storage.

ticksTime

A 32-bit shared value representing Ethernet-PON media-dependent time; incremented every 16 ns.

10.6.4 State machine routines

Dequeue(*queue*)

Enqueue(*queue*, *info*)

See 6.3.4.

NextTimed(*btPtr*, *stationTime*, *backInterval*)

See 7.5.4.

SourcePort(*entity*)

See 7.3.3.

StationTime(*entity*)

See 6.3.4.

TicksTime(*entity*)

See 10.4.4.

TimeSyncSdu(*info*)

See 6.3.4.

10.6.5 TimeSyncTxEpon state machine table

The TimeSyncTxEpon state machine includes a media-dependent timeout, which effectively disconnects a clock-slave port in the absence of received timeSyncEpon frames, as illustrated in Table 10.2.

Table 10.2—TimeSyncTxEpon state machine table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(rxInfo = Dequeue(Q_MS_REQ)) != NULL	1	—	SINK
	(stationTime – ePtr->lastTime) > T10ms	2	ePtr->lastTime = stationTime;	SEND
	—	3	stationTime = StationTime(ePtr); ticksTime = TicksTime(ePtr);	START
SINK	TimeSyncSdu(rsPtr)	4	ePtr->rxSaved = rxInfo;	START
	—	5	Enqueue(Q_ES_REQ, rxPtr);	

SEND	—	6	rxSyncInterval = ssPtr->syncInterval;
			txSyncInterval = ePtr->syncInterval;
			backInterval =
			(3 * rxSyncInterval + txSyncInterval) / 2;
			nextTimes =
			NextTimed(btPtr, stationTime, backInterval);
			txPtr->destination_address =
			sxPtr->destination_address;
			txPtr->source_address = sxPtr->source_address;
			tsPtr->protocolType = ssPtr->protocolType;
			tsPtr->function = ssPtr->function;
			tsPtr->version = ssPtr->version;
			tsPtr->precedence = ssPtr->precedence;
			tsPtr->hopCount = ssPtr->hopCount;
			tsPtr->grandTime = nextTimes.grandTime;
			tsPtr->extraTime = nextTimes.extraTime;
			tsPtr->ticksTime = ticksTime;
			Enqueue(Q_ES_REQ, txPtr);
				START

Row 10.2-1: Relayed frames are further checked before being processed.
Row 10.2-2: Transmit periodic timeSync frames.
Row 10.2-3: Wait for the next change-of-state.

Row 10.2-4: The timeSync PDU is saved and processed further.
Row 10.2-5: Non-timeSync PDUs are retransmitted in the standard fashion.

Row 10.2-6: Format and transmit the media-specific timeSync frame.

Annexes

Annex A

(informative)

Bibliography

[B1] IEEE 100, The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition.[1]

[B2] IEEE Std 802-2002, IEEE Standards for Local and Metropolitan Area Networks: Overview and Architecture.

[B3] IEEE Std 801-2001, IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture.

[B4] IEEE Std 802.1D-2004, IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges.

[B5] IEEE Std 1394-1995, High performance serial bus.

[B6] IEEE Std 1588-2002, IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.

[B7] IETF RFC 1305: Network Time Protocol (Version 3) Specification, Implementation and Analysis, David L. Mills, March 1992[2]

[B8] IETF RFC 2030: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI, D. Mills, October 1996.

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).

[2]IETF publications are available via the World Wide Web at http://www.ietf.org.

Annex B

(informative)

Time-scale conversions

B.1 Overview

For historical reasons, time is specified in a variety of ways as listed in Table B.1. GPS, PTP, and TAI times are based on values yielded by atomic clocks and advance on each second. NTP and UTC times are similar, but are occasionally adjusted by one leap-second, to account for differences between the atomic clocks and the rotation time of the earth.

Table B.1—Time-scale parameters

	Time scale				
Parameter	GPS	PTP	TAI	NTP	UTC
approximate epoch	1980-01-06 1999-08-22	1970-01-01	1972-01-01*	1900-01-01	1972-01-01*
representation	weeks.seconds	seconds	YYYY-MM-DD hh:mm:ss	seconds	YYYY-MM-DD hh:mm:ss
rollover (years)	19.7	8,925,513	10,000	136.19	10,000
leapSeconds	no			yes	

Notes:

* The TAI time when TAI and UTC were first specified to deviate by only integer seconds. (There is no true epoch for the TAI and UTC time scales.)

GPS global positioning satellite

NTP Network Time Protocol

PTP Precision Time Protocol (commonly used in POSIX)

TAI International Atomic Time (from the French term *Temps Atomique International*)

UTC Coordinated Universal Time (a compromise between the English and French):
English speakers wanted the initials of their language: CUT for "coordinated universal time"
French speakers wanted the initials of their language: TUC for "temps universel coordonné".

B.2 TAI and UTC

TAI and UTC are international standards for time based on the SI second; both are expressed in days, hours, minutes and seconds. TAI is implemented by a suite of atomic clocks and forms the timekeeping basis for other time scales in common use. The rate at which UTC time advances is normally identical to the rate of TAI. An exception is an occasion when UTC is modified by adding or subtracting leap seconds.

Prior to 1972-01-01, corrections to the offset between UTC and TAI were made in fractions of a second. After 1972-01-01, leap-second corrections are applied to UTC preferably following second 23:59:59 of the last day of June or December. As of 2006-01-01, TAI and UTC times differed by +33 seconds.

In POSIX based computer systems, the common time conversion algorithms can produce the correct ISO 8601-2004 printed representation format “YYYY-MM-DD hh:mm:ss” for both TAI and UTC.

The PTP epoch is set such that a direct application of the POSIX algorithm to a PTP time-scale timestamp yields the ISO 8601-2004 printed representation of TAI. Subtracting the current *leapSeconds* value from a PTP timestamp prior to applying the POSIX algorithm yields the ISO 8601-2004 printed representation of UTC. Conversely, applying the inverse POSIX algorithm and adding *leapSeconds* converts from the ISO 8601-2004 printed form of UTC to the form convenient for generating a PTP timestamp.

Example: The POSIX algorithm applied to a PTP timestamp value of 8 seconds yields 1970-01-01 00:00:08 (eight seconds after midnight on 1970-01-01 TAI). At this time the value of *leapSeconds* was approximately 8 seconds. Subtracting this 8 seconds from this time yields 1970-01-01 00:00:00 UTC.

Example: The POSIX algorithm applied to a PTP timestamp value of 0 seconds yields 1970-01-01 00:00:00 TAI. At this time the value of *leapSeconds* was approximately 8 seconds. Subtracting this 8 seconds from this time yields 1969-12-31 23:59:52 UTC.

B.3 NTP and GPS

Two standard time sources of particular interest in implementing PTP systems: NTP and GPS. Both NTP and GPS systems are expected to provide time references for calibration of the grand-master supplied PTP time.

NTP represents seconds as a 32 bit unsigned integer that rolls-over every 2^{32} seconds $\approx$ 136 years, with the first such rollover occurring in the year 2036. The precision of NTP systems is usually in the millisecond range.

NTP is a widely used protocol for synchronizing computer systems. NTP is based on sets of servers, to which NTP clients synchronize. These servers themselves are synchronized to time servers that are traceable to international standards.

NTP provides the current time. In NTP version 4, the current *leapSeconds* value and warning flags marking indicating when a *leapSecond* will be inserted at the end of the current UTC day. The NTP clock effectively stops for one second when the leap second is inserted.

GPS time comes from a global positioning satellite system, GPS, maintained by the U.S. Department of Defense. The precision of GPS system is usually in the 10-100 ns range. GPS system transmissions represent the time as {*weeks*, *secondsInWeek*}, the number of weeks since the GPS epoch and the number of seconds since the beginning of the current week.

GPS also provides the current *leapSeconds* value, and warning flags marking the introduction of a leap second correction. UTC and TAI times can be computed solely based the information contained in the GPS transmissions.

GPS timing receivers generally manage the epoch transitions (1024-week rollovers), providing the correct time (YYYY-MM-DD hh:mm:ss) in TAI and/or UTC time scales, and often also local time; in addition to providing the raw GPS week, second of week, and leap second information.

B.4 Time-scale conversions

Previously discussed representations of time can be readily converted to/from PTP *time* based on a constant offsets and the distributed *leapSeconds* value, as specified in Table B.2. Within Table B.2, all variables represent integers; '/' and '%' represent a integer divide and remainder operation, respectively.

Table B.2—Time-scale conversions

ta		**PTP value *tb*:**
name	format	
GPS	weeks:seconds	tb = ta.seconds + 315 964 819 + (gpsRollovers * 1024 + ta.weeks) * (7 * DAYSECS);
		ta.weeks = (tb – 315 964 819) / (7 * DAYSECS); ta.days = (tb – 315 964 819) % (7 * DAYSECS);
TAI	date{YYYY,MM,DD}:time{hh,mm,ss}	tb = DateToDays("1970-01-01", ta.date) * DAYSECS + ((ta.time.hh * 24) + ta.time.mm) *60) + ta.time.ss;

		secs = tb % DAYSECS;
		ta.date = DaysToDate("1970-01-01", tb / DAYSECS);
		ta.time.hh = secs / 3600;
		ta.time.mm = (secs % 3600)/60;
		ta.time.ss = (secs % 60);

NTP	seconds	tb = (ta + leapSeconds) – 2208988800;
		ta = (ta – leapSeconds) + 2208988800;
UTC	date{YYYY,MM,DD}:time{hh,mm,ss}	tb = DateToDays("1970-01-01", ta.date) * DAYSECS + ((ta.time.hh * 24) + ta.time.mm) *60) + ta.time.ss + leapSeconds;

		tc = tb – leapSeconds;
		secs = tc % DAYSECS;
		ta.date = DaysToDate("1970-01-01", tc/DAYSECS);
		ta.time.hh = secs / 3600;
		ta.time.mm = (secs % 3600)/60;
		ta.time.ss = (secs % 60);

Note:

gpsRollovers Currently equals 1; changed from 0 to 1 between 1999-08-15 and 1999-08-22.
DAYSECS The number of seconds within a day: (60*60*24).
leapSeconds Extra seconds to account for variations in the earth-rotation times: 33 on 2006-01-01.
DateToDays For arguments DateToDays(*past*, *present*), returns days between *past* and *present* dates.
DaysToDate For arguments DaysToDate(*past*, *days*), returns the current date, *days* after the *past* date.

B.5 Time zones and GMT

The term Greenwich Mean Time (GMT) once referred to mean solar time at the Royal Observatory in Greenwich, England. GMT now commonly refers to the time scale UTC; or the UK winter time zone (Western European Time, WET). Such GMT references are strictly speaking incorrect; but nevertheless quite common. The following representations correspond to the same instant of time:

18:07:00 (GMT), commonplace usage	13:07:00 (Eastern Standard Time, EST)
18:07:00 (UTC)	1:07 PM (Eastern Standard Time, EST)
18:07:00 (Western European Time, WET)	10:07:00 (Pacific Standard Time, PST)
6:07 PM (Western European Time, WET)	10:07 AM (Pacific Standard Time, PST)

Annex C

(informative)

Simulation results (preliminary)

C.1 Simulation environment

This annex describes several simulations performed with the intent of comparing time-extrapolation and time-interpolation algorithms. To reduce possibilities of code-conversion errors, the simulation model executes the C code of Annex F. Simulation time is based on a 128-bit *systemTime*, represented by 64-bit seconds and fractions-of-second components, to ensure that precision and range are not constraining factors.

The simulation consists of *bridgeCount* identical super-bridge components, as illustrated in Figure C.1. For generality and uniformity, each bridge includes ClockMaster and ClockSlave entities. The smallest MAC address is assigned to the left-most station; for other stations, the address is incremented for each sequential right-side bridge. The simulations assumed *bridgeCount* values of 8 (the assumed AVB diameter) and 64 (a reasonable IEEE 802.17 ring diameter).

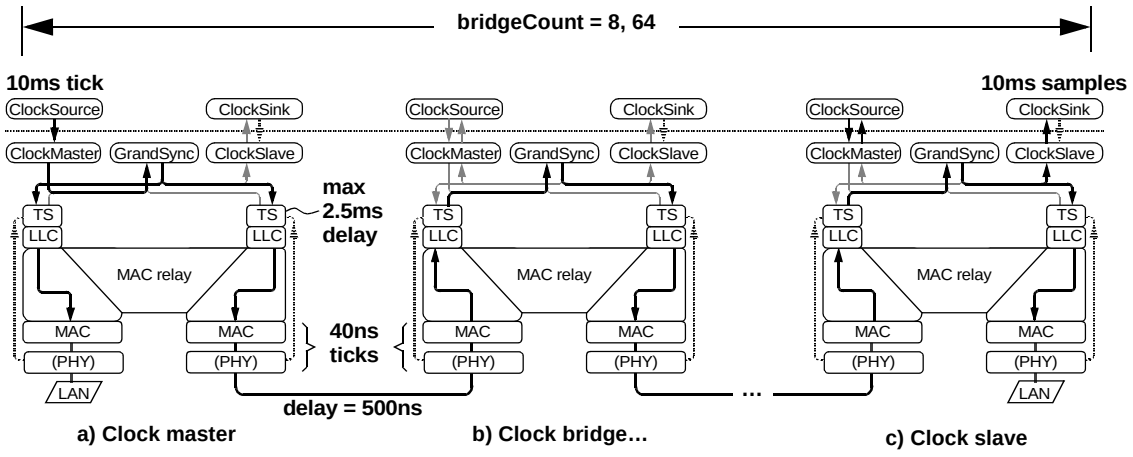

Figure C.1—Time-synchronization flows

The transmit portion of the TS component (emulated by the DuplexTxExec routine) introduces a random delay of no more than 2.5 ms, thus emulating delays consistent with the 10 ms sync-frame transmission rate. A 20 ns sampling clock ambiguity (corresponding to 25 MHz) is incorporated into the MAC component (emulated by the DupMacTxExec routine).

The cable is modeled as a symmetric 500ns delay, corresponding to a cable length of approximately 100 meters.

Station clock accuracies are assigned randomly/uniformly within the range of the allowed ±100 PPM deviation from the simulation's emulated/exact *systemTime* reference.

NOTE—Please be tolerant of the editor of this document, who just downloaded the gnuplot application and fft4 library today. These initial cut-and-paste of plots are primitive (to be improved, when EPS or other formats are understood) and no noise-spectrum plots (to better illustrate gain peaking) are currently available. Improvements expected soon…

C.2 Initialization transients

C.2.1 Cascaded 8 stations

A significant expected initialization transient is observed when all stations simultaneously start operations, as illustrated in Figure C.2. This can be contributed to inaccurate initial estimates of receiver's link-delay and transmitter's rate estimations. The transient delays (although significant) are much less than expected from designs based on many-sample grand-master rate-syntonization delays within bridges.

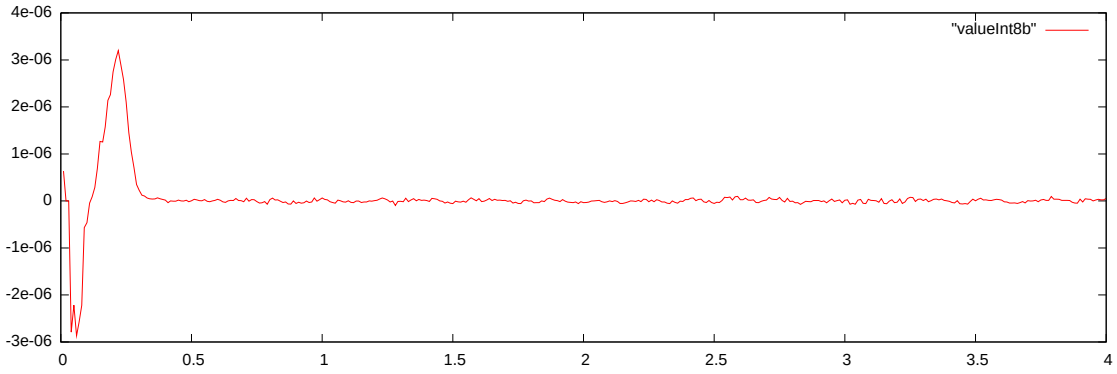

Figure C.2—Startup transients with 8 stations

C.2.2 Cascaded 64 stations

The length of the initialization transient increases when the number of bridges is increased to 64, as illustrated in Figure C.3. The much-longer duration of such transients is perhaps tolerable, but illustrates the desire to avoid extrapolation-based on many-sample grand-master rate-syntonization delays within bridges.

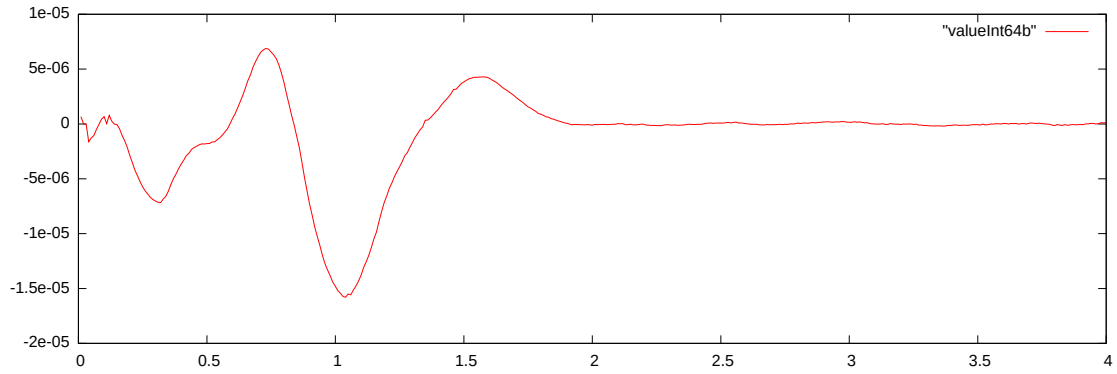

Figure C.3—Startup transients with 64 stations

C.3 Steady-state interpolation errors

C.3.1 Time interpolation with 8 stations

Simulations indicate modest peak-to-peak errors for 8-bridge topologies when interpolation-based protocols are used, as illustrated in Figure C.4.

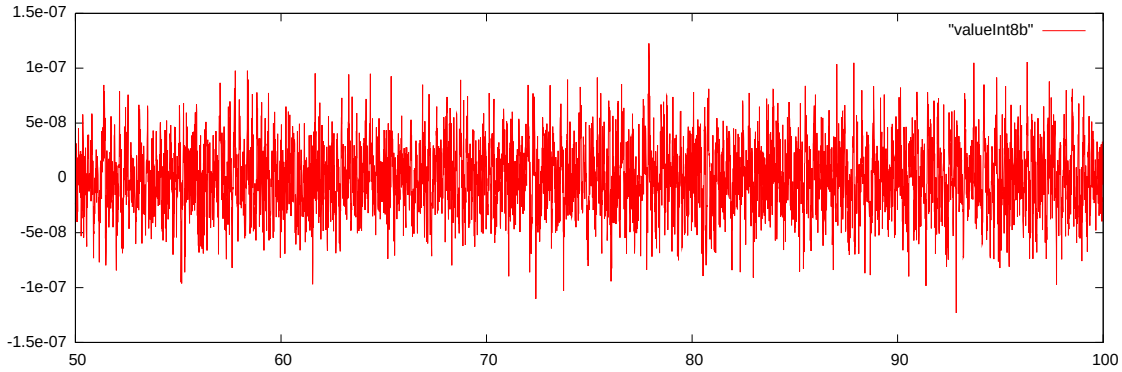

Figure C.4—Time interpolation with 8 stations

C.3.2 Time interpolation with 64 stations

Simulations indicate modest peak-to-peak error increases for 64-bridge topologies (as expected to 8-bridge topologies) when interpolation-based protocols are used, as illustrated in Figure C.5. The data is consistent with less-than-linear expectations, due to statistical averaging and intermediate interpolation filtering.

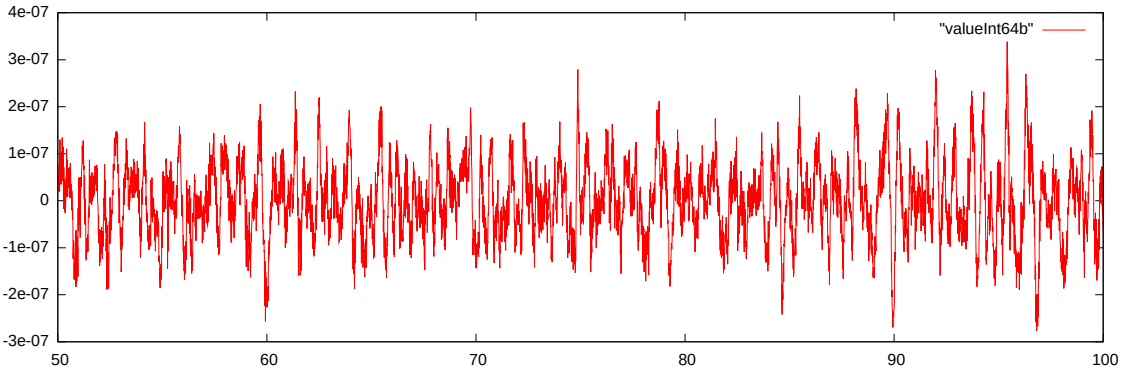

Figure C.5—Time interpolation with 64 stations

C.4 Steady-state extrapolation errors

C.4.1 Time extrapolation with 8 stations

Simulations indicate approximately twice the errors for 8-bridge topologies when extrapolation-based protocols (as opposed to interpolation-based protocols) are used, as illustrated in Figure C.6.

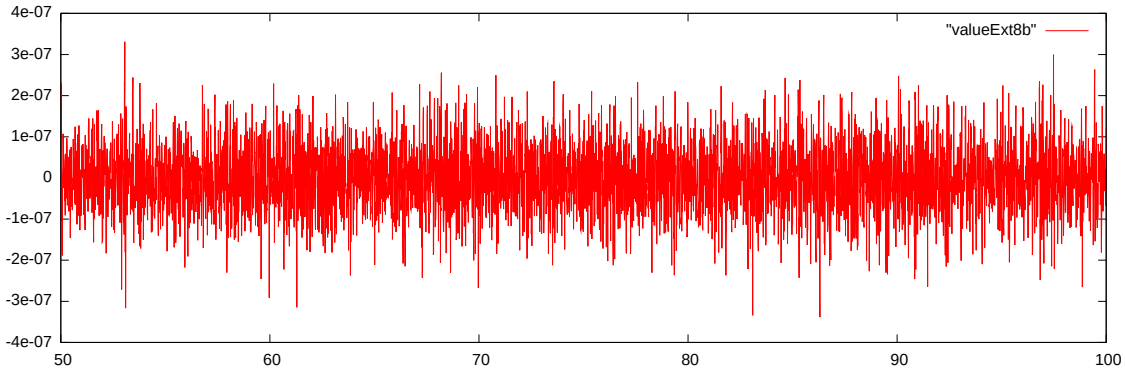

Figure C.6—Time extrapolation with 8 stations

C.4.2 Time extrapolation with 64 stations

Simulations indicate significantly larger peak-to-peak errors for 64-bridge topologies when extrapolation-based protocols (as opposed to interpolation-based protocols) are used, as illustrated in Figure C.7.

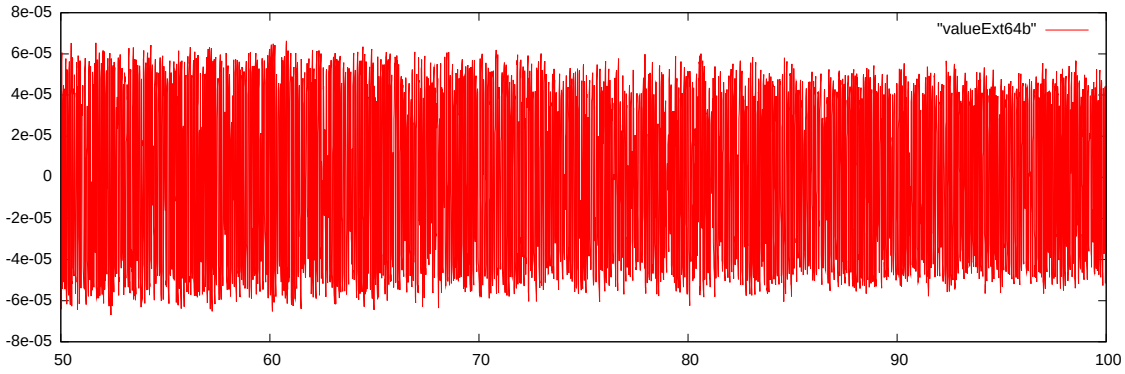

Figure C.7—Time extrapolation with 64 stations

Annex D

(informative)

Bridging to IEEE Std 1394

To illustrate the sufficiency and viability of the AVB time-synchronization services, the transformation of IEEE 1394 packets is illustrated.

D.1 Hybrid network topologies

D.1.1 Supported IEEE 1394 network topologies

This annex focuses on the use of AVB to bridge between IEEE 1394 domains, as illustrated in Figure D.1. The boundary between domains is illustrated by a dotted line, which passes through a SerialBus adapter station.

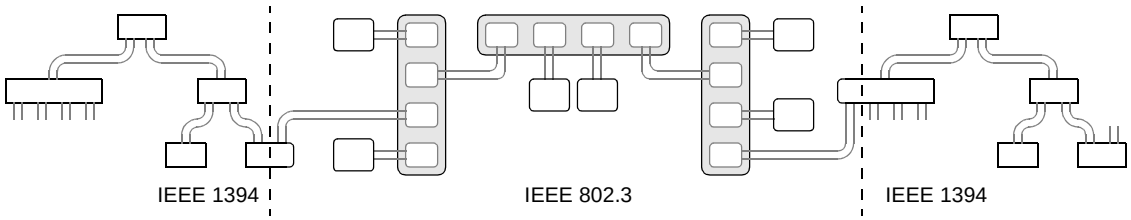

Figure D.1—IEEE 1394 leaf domains

D.1.2 Unsupported IEEE 1394 network topologies

Another approach would be to use IEEE 1394 to bridge between IEEE 802.3 domains, as illustrated in Figure D.2. While not explicitly prohibited, architectural features of such topologies are beyond the scope of this working paper.

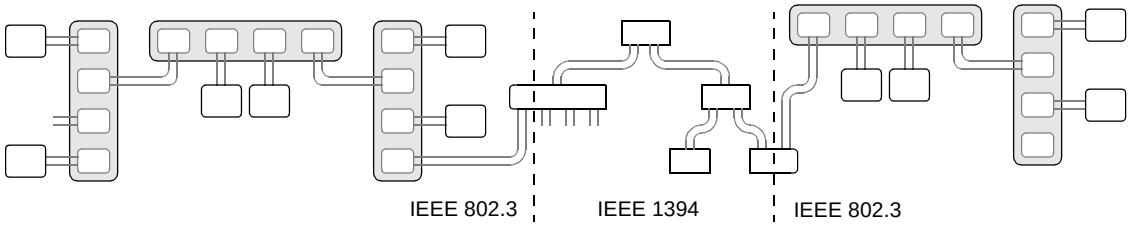

Figure D.2—IEEE 802.3 leaf domains

D.1.3 Time-of-day format conversions

The difference between AVB and IEEE 1394 time-of-day formats is expected to require conversions within the AVB-to-1394 adapter. Although multiplies are involved in such conversions, multiplications by constants are simpler than multiplications by variables. For example, a conversion between AVB and IEEE 1394 involves no more than two 32-bit additions and one 16-bit addition, as illustrated in Figure D.3.

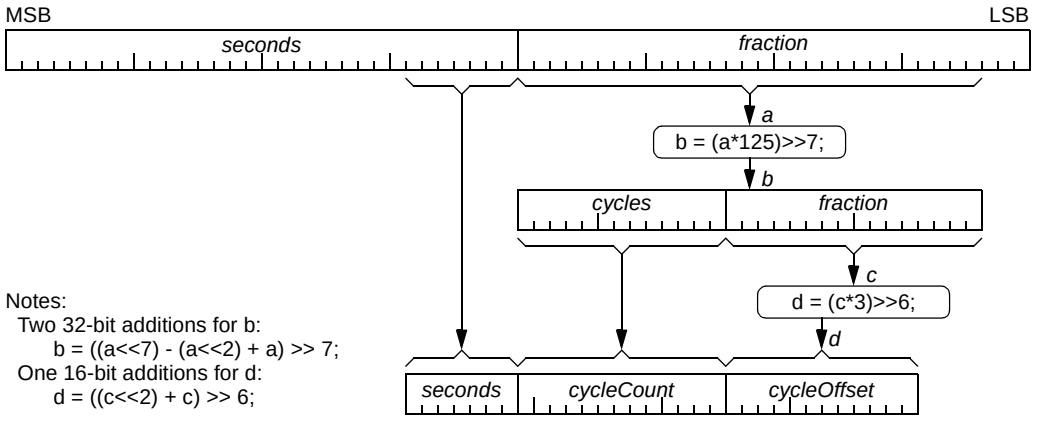

Figure D.3—Time-of-day format conversions

D.1.4 Grand-master precedence mappings

Compatible formats allow either an IEEE 1394 or IEEE 802.3 stations to become the network's grand-master station. While difference in format are present, each format can be readily mapped to the other, as illustrated in Figure D.4:

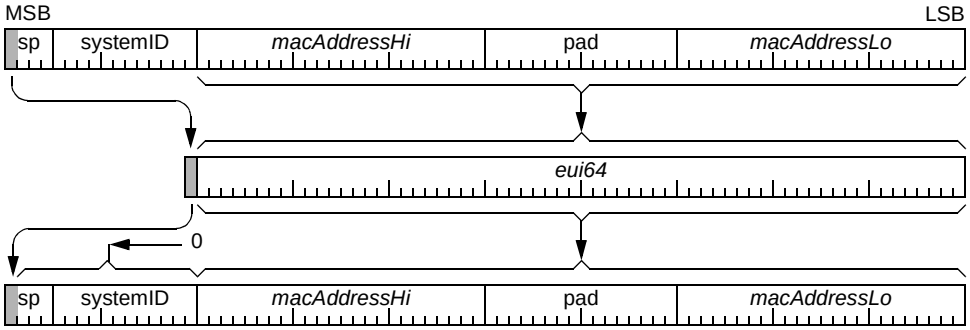

Figure D.4—Grand-master precedence mapping

Annex E

(informative)

Time-of-day format considerations

To better understand the rationale behind the ‘extended binary’ timer format, various possible formats are described within this annex.

E.1 Possible time-of-day formats

E.1.1 Extended binary timer formats

The extended-binary timer format is used within this working paper and summarized herein. The 64-bit timer value consist of two components: a 40-bit *seconds* and 40-bit *fraction* fields, as illustrated in Figure E.1.

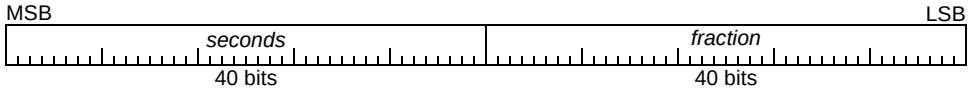

Figure E.1—Global-time subfield format

The concatenation of 40-bit *seconds* and 40-bit *fraction* field specifies an 80-bit *time* value, as specified by Equation E.1.

$$time = seconds + (fraction / 2^{40}) \quad \text{(E.1)}$$

Where:

seconds is the most significant component of the time value.

fraction is the less significant component of the time value.

E.1.2 IEEE 1394 timer format

An alternate “1394 timer” format consists of *secondCount*, *cycleCount*, and *cycleOffset* fields, as illustrated in Figure E.2. For such fields, the 12-bit *cycleOffset* field is updated at a 24.576MHz rate. The *cycleOffset* field goes to zero after 3071 is reached, thus cycling at an 8kHz rate. The 13-bit *cycleCount* field is incremented whenever *cycleOffset* goes to zero. The *cycleCount* field goes to zero after 7999 is reached, thus restarting at a 1Hz rate. The remaining 7-bit *secondCount* field is incremented whenever *cycleCount* goes to zero.

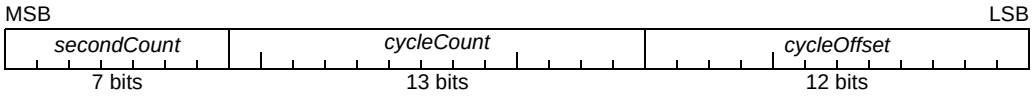

Figure E.2—IEEE 1394 timer format

E.1.3 IEEE 1588 timer format

IEEE Std 1588-2002 timer format consists of seconds and nanoseconds fields components, as illustrated in Figure E.3. The nanoseconds field must be less than 10^9; a distinct *sign* bit indicates whether the time represents before or after the epoch duration.

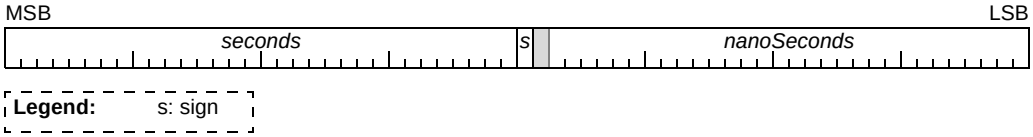

Figure E.3—IEEE 1588 timer format

E.1.4 EPON timer format

The IEEE 802.3 EPON timer format consists of a 32-bit scaled nanosecond value, as illustrated in Figure E.4. This clock is logically incremented once each 16 ns interval.

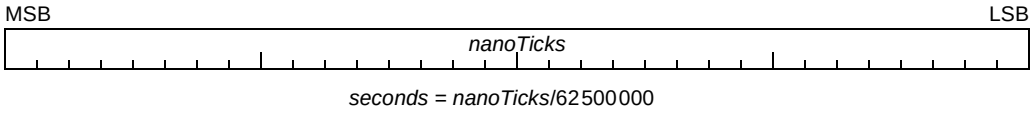

Figure E.4—EPON timer format

Annex F

(informative)

C-code illustrations

NOTE—This annex is provided as a placeholder for illustrative C-code. Locating the C code in one location (as opposed to distributed throughout the working paper) is intended to simplify its review, extraction, compilation, and execution by critical reviewers.
Also, placing this code in a distinct Annex allows the code to be conveniently formatted in 132-character landscape mode. This eliminates the need to truncate variable names and comments, so that the resulting code can be better understood by the reader.

This Annex provides code examples that illustrate the behavior of AVB entities. The code in this Annex is purely for informational purposes, and should not be construed as mandating any particular implementation. In the event of a conflict between the contents of this Annex and another normative portion of this standard, the other normative portion shall take precedence.

The syntax used for the following code examples conforms to ANSI X3T9-1995.

```
// ********************************************************************************************************************************
//                                                                                                                1         1         1         1
//       1         2         3         4         5         6         7         8         9         0         1         2         3
//3456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
// ********************************************************************************************************************************
// ****************************************** by Dr. David V James, 2007-05-18 ****************************************************
// ********************************************************************************************************************************

// NOTE--The following code is portable with respect to endian ordering,
// but (for clarity and simplicity) assumes availability of 64-bit integers.

// TBDs:
// Active:
//   Suggested changes for clarity (anonymous reviewer 2007-05-11):
//     Separate partition for the bulk of header
//     Make TBDs explicit
//     Consider name change: state machines => queue service routines
//     Better use of spaces and comments
//   FFT and spectrum analysis via utilities or piped process
// Completed:
//     Initial checks to be more descriptive, as in "GroupAsserts"
//     Consistent terminology: backInterval
//     Sequence of tests within looks, possibly with "serviced", as in:
//       for (checkForMore = TRUE; checkForMore == TRUE; ) {
//         if (something) {
//             checkForMore = TRUE;
//         }
//         ...
//       } // for(;;) ends here
//     Ports=>queues, from a naming perspective
//     Separate initialization from routines
//     rating => rateRatio0, etc. for similar name usage
//     matched to something more descriptive, as in "countsAreEqual"
//     Ethernet-duplex, Ethernet-pon, more descriptive names to be used

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "avbHead.h"

// ****************************************************************************
// Time formats used within this simulation are listed below.
// The layout is half scale; each '+' mark represents a byte boundary,
// not a bit-boundary (as is true in other narrow-format conventions.
// The high-level timings are based on largeTime and smallTime values.
//
//                              largeTime
//  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
//  |            seconds            |            fraction           |
//  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
//  Used for: Simulation time base  :
//  Features: Near-infinite resolution and range
//                                  :
```

```
//                                :
//                            smallTime
//                        +---+---+---+---+---+---+---+---+
//                        |seconds|        fraction       |
//                        +---+---+---+---+---+---+---+---+
//  Used for: Station-local time base
//  Features: Nearly-a-day range; near femto-second resolution
//                                :
// ********************************:****************************************
// Time formats used within application-specific frames include a
//  * grandTime - never-overflows globally-synchronized time
//  * localTime - per-station locally-managed time
//  * tinyTime  - extra part of grandTime (a small value)
//  * ticksTime - an application-specific variant of localTime
//                                :
//                            grandTime
//              +---+---+---+---+---+---+---+---+---+---+
//              |      seconds      |      fraction     |
//              +---+---+---+---+---+---+---+---+---+---+
//  Used for: Frames, grand-master time synchronization
//  Features: Thousands-of-years range; pico-second resolution
//                                :
//                            localTime
//                            +---+---+---+---+---+---+---+
//                            |sec|     fraction      |
//                            +---+---+---+---+---+---+---+
//  Used for: Frames, cable-delay measurements
//  Features: Minutes of range; pico-second resolution
//                                :
//                                :       tinyTime
//                                + - +---+---+---+---+
//                                :   |  subfraction  |
//                                + - +---+---+---+---+
//                                :
//  Used for: Sideband grandTime error accumulation
//  Features: Minutes of range; pico-second resolution
//                                :
//                                :    ticksTime
//                             +---+---+---+---+---+---+---+
//                             |       tickCounts      |
//                             +---+---+---+---+---+---+---+
//  Used for: Exists on 802.3-EPON, 802.11v-Radio
//  Features: Sufficient range; nano-second-like resolution
//
// ***************************************************************************


// ***************************************************************************
// ***************************** Assumed integer values ******************************
// ***************************************************************************

// typedef unsigned char       uint8_t;          // 1-byte unsigned integer
// typedef unsigned short      uint16_t;         // 2-byte unsigned integer
// typedef unsigned int        uint32_t;         // 4-byte unsigned integer
// typedef unsigned long long  uint64_t;         // 8-byte unsigned integer
```

```
// typedef signed char         int8_t;           // 1-byte signed integer
// typedef signed short        int16_t;          // 2-byte signed integer
// typedef signed int          int32_t;          // 4-byte signed integer
// typedef signed long long    int64_t;          // 8-byte signed integer


// ********************************************************************************
// ******** Revise timeSync frame parameters as the actual values become known *********
// ********************************************************************************

// Unique identifier values
#define AVB_MCAST     0                         // Neighbor multicast address
#define AVB_PROTOCOL  0                         // The protocolType for AVB
#define AVB_FUNCTION  0                         // The timeSync function
#define AVB_VERSION   1                         // The timeSync version

#define DEFAULT_DUPLEX_RX_RANGE 16

#define DEBUG 0

// Generic macro definitions
#define ARRAY_SIZE(x) (sizeof(x)/sizeof(x[0]))
#define BITS(type) (8 * sizeof(type))
#define CLIP_RATE(x, y) (((x) > SMALL_ONE + (y)) ? \
 (SMALL_ONE + (y)) : (((x) < (SMALL_ONE - (y)) ? SMALL_ONE - (y) : (x))))              // Clip within specified rate
#define CLIP_SIZE(x, y) ((x) > (y) ? (y) : ((x) < (-(y)) ? (-(y)) : (x)))             // Clip within specified value
#define CLOCK_MASTER_PORT_ID 255                                                      // Clock-master port identifier
#define COUNT 256                                                                     // Number of frameCount values
#define EXTRA 16
#define FULL  1                                                                       // Queue-full error status
#define LAST_HOP 255                                                                  // Largest hop-count value
#define LARGE_10ms SmallToLarge(SMALL_10ms)
#define LARGE_HALF (ONE << 63)
#define LARGE_TOCK (ONE << 62)
#define MASK(bits) ((ONE << bits) - 1)
#define MASK32 (ONES >> 32)
#define MAX(a, b) ((a) > (b) ? (a) : (b))                                             // Maximum value definition
#define MIN(a, b) ((a) > (b) ? (b) : (a))                                             // Minimum value definition
#define MTU_SIZED 2048                                                                // Maximum-sized transfer
#define OK 0                                                                          // Non-error status
#define ONE ((uint64_t)1)                                                             // Wide "1" constant
#define ONES (~(uint64_t)0)                                                           // Wide "FF..FF" constant
#define PLUS(a, b, c) (((a) + (b) + (c)) % (c))
#define PON_TICK_TIME (DivideSmall(16 * (ONE << 32), 1000000000))
#define PPM100 ((SMALL_ONE * 100) / 1000000)                                          // Scaled 100PPM fraction.
#define PPM250 ((SMALL_ONE * 250) / 1000000)                                          // Scaled 250PPM fraction.
#define RADIO_TICK_TIME DivideSmall(1 << (32 - 9), 1000000000 >> 9)                   // Ratio radio-ns to localTime
#define RESIDENCE_DELAY ((SMALL_ONE + RandomMagOne()) / 800)                          // A 2.5ms max residence time
#define SMALL_10ms (SMALL_ONE / 100)                                                  // A 10ms smallTime interval
#define SMALL_ONE ((int64_t)(ONE << 48))                                              // Scaled fraction for 1.0
#define TESTING_OUI ((uint64_t)0Xcab1ed << 24)
#define TLIMIT  255
#define WIDE_MIN(a, b) (WideCompare((a), (b)) <= 0 ? (a) : (b))
```

```
#define CommonCheck(comPtr) (assert(comPtr != NULL), \
  assert(comPtr->rootLink != NULL), assert(comPtr->pairLink != NULL))

#define SetRxQueue1Ptrs(comPtr, ptr0) (CommonCheck(comPtr), \
  assert(comPtr->rxPortCount == 1), ptr0 = &(comPtr->rxPortPtr[0]) )
#define SetRxQueue2Ptrs(comPtr, ptr0, ptr1) (CommonCheck(comPtr), \
   assert(comPtr->rxPortCount == 2), \
   ptr0 = &(comPtr->rxPortPtr[0]), ptr1 = &(comPtr->rxPortPtr[1]) )
#define SetRxQueue3Ptrs(comPtr, ptr0, ptr1, ptr2) (CommonCheck(comPtr), \
   assert(comPtr->rxPortCount == 3), ptr0 = &(comPtr->rxPortPtr[0]), \
   ptr1 = &(comPtr->rxPortPtr[1]), ptr2 = &(comPtr->rxPortPtr[2]) )
#define SetTxQueue1Ptrs(comPtr, ptr0) (CommonCheck(comPtr), \
   assert(comPtr->txPortCount == 1), ptr0 = &(comPtr->txPortPtr[0]) )
#define SetTxQueue2Ptrs(comPtr, ptr0, ptr1) (CommonCheck(comPtr), \
   assert(comPtr->txPortCount == 2), \
   ptr0 = &(comPtr->txPortPtr[0]), ptr1 = &(comPtr->txPortPtr[1]) )

#define RxPortPtr(comPtr, count) (&(comPtr->rxPortPtr[count]))
#define StationTime(comPtr) (comPtr->smallTime)
#define SystemTime(comPtr)  (comPtr->largeTime)
#define TxPortPtr(comPtr, count) (&(comPtr->txPortPtr[count]))

#define PrecedenceToEui64(a) (a.lower)

#define SizePlus(set) (sizeof(set) + EXTRA)

#define LargeToSmall(a) WideExtract(a, 16)
#define SmallToLarge(a) WideShift(SignedToWide(a), -16)
#define SmallToGrand(a) WideShift(SignedToWide(a), 8)
#define SmallToLocal(a) ((a) >> 8)
#define SmallAsLocal(a) ((a) & (ONES >> 8))
#define TinyToGrand(x)  (SignedToWide((int64_t)(x)))
#define TinyToSmall(a)  (((int64_t)(a)) << 8)
#define TinyToLarge(a)  (SignedToWide(((int64_t)(a)) << 24))

typedef enum {
  INTERPOLATE,
  EXTRAPOLATE
} GuessMode;

enum {
  FALSE,
  TRUE,
  WAIT
};

enum {
  BODY,
  LIST,
  BOTH
};

enum {
  TALK_QUIET,
```

```
  TALK_GSYNC,
  TALK_FRAME
};

enum {
  VOCAL_QUIET,
  VOCAL_DEBUG,
  VOCAL_PAIRS
};

// Field extract/deposit definitions
#define FieldToSigned(fPtr, field) \
 FrameToValue((uint8_t *)(&(fPtr->field)), sizeof(fPtr->field), TRUE)                  // Convert field to signed
#define FieldToUnsign(fPtr, field) \
 FrameToValue((uint8_t *)(&(fPtr->field)), sizeof(fPtr->field), FALSE)                 // Convert field to unsigned
#define WideToFrame(value, fPtr, field) \
 ValueToFrame(value, (uint8_t *)(&(fPtr->field)), sizeof(fPtr->field))                 // Convert field to unsigned
#define LongToFrame(value, fPtr, field) \
 ValueToFrame(SignedToWide(value), (uint8_t *)(&(fPtr->field)), sizeof(fPtr->field))

#define DeQueue(a, b) Dequeue(a, (uint8_t *)b, sizeof(*b))
#define EnQueue(a, b) Enqueue(a, (uint8_t *)b, sizeof(*b))

typedef struct {                                // Double-precise integers
    uint64_t upper;                             // More significant portion
    uint64_t lower;                             // Less significant portion
} WideUnsigned;


#ifndef AVB_TIMES

#define NLIMIT   63

typedef struct {                                // Double-precise integers
    int64_t  upper;                             // More significant portion
    uint64_t lower;                             // Less significant portion
} WideSigned;

typedef int32_t      TicksTime;                 // Link-dependent time
typedef int64_t      SmallTime;                 // Local time reference
typedef WideSigned   LargeTime;                 // General 128-bit timers
#endif


typedef uint8_t      Boolean;                   // True or false
typedef uint8_t      Port;                      // Received port number
typedef uint8_t      Class;                     // 1588: clock class
typedef uint8_t      HopCount;                  // 1588: distance from GM
typedef uint16_t     Variance;                  // 1588: clock error variance
typedef int32_t      TinyTime;                  // Extra part of GM time
typedef int64_t      LocalTime;                 // Compacted SmallTime
typedef WideSigned   GrandTime;                 // 1588: grand-master time
typedef WideSigned   Precedence;                // Fields {priorities,clockID}
typedef WideUnsigned Preference;                // Fields {precedence,hops,port}
```

```
// ********************************************************************************
// *************************** Communication components ***************************
// ********************************************************************************

typedef struct {
    LargeTime  largeTime;                       // Grand-master synchronized
    SmallTime  extraTime;                       // Extra part for largeTime
    SmallTime  smallTime;                       // Station's free-running
    uint16_t   extraCount;                      // Count of extra-values sampling
} BaseTimes;

typedef struct {                                // Time-result collection
    GrandTime grandTime;                        // Grand-master synchronized
    TinyTime  extraTime;                        // Side-band extra values
    GrandTime totalTime;                        // Precise grandTime+extraTime
} NextTimes;


// ********************************************************************************
// ************************** Formal interface exchanges **************************
// ********************************************************************************

typedef struct {
    uint8_t   frameCount[1];                    // Sequential consistency check
    uint8_t   grandTime[10];                    // Received grand-master time
} ClockMasterSet;

typedef struct {
    uint8_t   frameCount[1];                    // Sequential consistency check
} ClockSlaveReq;

typedef struct {
    uint8_t   frameCount[1];                    // Sequential consistency check
    uint8_t   grandTime[10];                    // Provided grand-master time
} ClockSlaveRes;

typedef struct {                                // Time-sync frame parameters
    uint8_t protocolType[2];                    // Protocol identifier
    uint8_t function[1];                        // Identifies timeSync frame
    uint8_t version[1];                         // Specific format identifier
    uint8_t precedence[14];                     // Grand-master precedence
    uint8_t grandTime[10];                      // Grand-master time
    uint8_t extraTime[4];                       // Extra part of grandTime
    uint8_t sourcePort[1];                      // Transmit sequence number
    uint8_t hopCount[1];                        // GM hop-count distance
    uint8_t smallTime[8];                       // Local-time reference
    uint8_t syncInterval[6];                    // Opposite-link transmit time
} SyncSduData;

typedef struct {                                // MS_UNITDATA.request
    uint8_t     destination_address[6];         // Destination address
    uint8_t     source_address[6];              // Source address
    uint8_t     priority[1];                    // Delivery priority
    SyncSduData service_data_unit;              // Data content
```

```
} GrandSyncReq;

typedef struct {                                // MS_UNITDATA.indication
    uint8_t     destination_address[6];         // Destination address
    uint8_t     source_address[6];              // Source address
    uint8_t     priority[1];                    // Delivery priority
    SyncSduData service_data_unit;              // Data content
} GrandSyncInd;

typedef struct {                                // Time-sync frame parameters
    uint8_t       protocolType[2];              // Protocol identifier
    uint8_t       function[1];                  // Identifies timeSync frame
    uint8_t       version[1];                   // Specific format identifier
    uint8_t       precedence[14];               // Grand-master precedence
    uint8_t       grandTime[10];                // Grand-master time
    uint8_t       extraTime[4];                 // Extra part of grandTime
    uint8_t       frameCount[1];                // Transmit sequence number
    uint8_t       hopCount[1];                  // GM hop-count distance
    uint8_t       localTime[6];                 // Transmitted timeSync time
    uint8_t       thatTxTime[6];                // Opposite-link transmit time
    uint8_t       thatRxTime[6];                // Opposite-link received time
} EfdxSduData;

typedef struct {                                // Time-sync frame parameters
    uint8_t       da[6];                        // Destination address
    uint8_t       sa[6];                        // Source address
    uint8_t       protocolType[2];              // Protocol identifier
    uint8_t       function[1];                  // Identifies timeSync frame
    uint8_t       version[1];                   // Specific format identifier
    uint8_t       precedence[14];               // Grand-master precedence
    uint8_t       grandTime[10];                // Grand-master time
    uint8_t       extraTime[4];                 // Extra part of grandTime
    uint8_t       frameCount[1];                // Transmit sequence number
    uint8_t       hopCount[1];                  // GM hop-count distance
    uint8_t       localTime[6];                 // Transmitted timeSync time
    uint8_t       thatTxTime[6];                // Opposite-link transmit time
    uint8_t       thatRxTime[6];                // Opposite-link received time
    uint8_t       fcs[4];                       // Opposite-link received time
} EfdxMacFrame;

typedef struct {                                // Time-sync frame parameters
    uint8_t       destination_address[6];       // Destination address
    uint8_t       source_address[6];            // Source address
    uint8_t       priority[1];                  // Delivery priority
    EfdxSduData    service_data_unit;            // Efdx service-data-unit
} EfdxMacInd;
typedef EfdxMacInd EfdxMacReq;

typedef struct {
    uint8_t   frameCount[1];                    // Sequential consistency check
    uint8_t   smallTime[8];                     // Common station-local time
} EfdxRxInfo;

typedef struct {
    uint8_t   frameCount[1];                    // Sequential consistency check
```

```
    uint8_t   smallTime[8];                 // Common station-local time
} EfdxTxInfo;

typedef struct {                            // Time-sync frame parameters
    uint8_t protocolType[2];                // Protocol identifier
    uint8_t function[1];                    // Identifies timeSync frame
    uint8_t version[1];                     // Specific format identifier
    uint8_t precedence[14];                 // Grand-master precedence
    uint8_t grandTime[10];                  // Grand-master time
    uint8_t extraTime[4];                   // Extra part of grandTime
    uint8_t frameCount[1];                  // Transmit sequence number
    uint8_t hopCount[1];                    // GM hop-count distance
    uint8_t ticksTime[4];                   // Local timing reference
} SyncSduEpon;

typedef struct {                            // Time-sync frame parameters
    uint8_t       destination_address[6];   // Destination address
    uint8_t       source_address[6];        // Source address
    uint8_t       priority[1];              // Delivery priority
    SyncSduEpon   service_data_unit;        // Efdx service-data-unit
} EponMacInd;
typedef EponMacInd EponMacReq;

typedef struct {
    uint32_t   reserved;                    // Reserved
} R11vInfo1Req;

typedef struct {
    uint32_t ticksTime2;                    // Received snapshot
    uint32_t ticksTime3;                    // Transmit snapshot
} R11vInfo1Ind;

typedef struct {
    uint32_t ticksTime1;                    // Transmit snapshot
    uint32_t ticksTime4;                    // Received snapshot
} R11vInfo1Con;

typedef struct {
    uint32_t   ticksTime4;                  // Received snapshot
    uint32_t   roundTrip;                   // Duration snapshot
    GrandTime  levelTime;                   // Grand-master like
    TinyTime   extraTime;                   // Extra part of levelTime
    Precedence precedence;                  // Grand-master error
    HopCount   hopCount;                    // Grand-master error
} R11vInfo2Req;

typedef R11vInfo2Req R11vInfo2Ind;

typedef struct {
    uint32_t   reserved;                    // Reserved
} R11vInfo2Con;


// ********************************************************************************
// ****************************** Defined entities ********************************
```

```
// ********************************************************************************

enum {
    Q_RX00_LAST,
    Q_TX00_LAST = 0
};

enum {
    Q_RX01_BASE,
    Q_RX11_LAST,
    Q_TX01_BASE = 0,
    Q_TX11_LAST
};

enum {
    Q_RX02_BASE,
    Q_RX12_NEXT,
    Q_RX22_LAST,
    Q_TX02_BASE = 0,
    Q_TX12_NEXT,
    Q_TX22_LAST
};

enum {
    Q_RX03_BASE,
    Q_RX13_NEXT,
    Q_RX23_PLUS,
    Q_RX33_LAST,
    Q_TX03_BASE = 0,
    Q_TX13_NEXT,
    Q_TX23_NEXT,
    Q_TX33_LAST
};

// ****************************** GrandSync entity ***********************************

typedef struct {                                // GrandSync entity state
    Precedence     precedence;                  // Grand-master precedence
    Port           sourcePort;                  // Source-port identifier
    HopCount       hopCount;                    // Synchronization interval
    SmallTime      syncInterval;                // Synchronization interval
} GrandSyncSaved;

typedef struct {                                // GrandSync entity state
    Common         common;                      // Common simulation state
    LocalTime      lastTime;                    // Timeout, set on activity
    GrandSyncSaved rxSaved;                     // Saved indication
} GrandSyncEntity;


// ***************************** ClockMaster entity **********************************

typedef struct {                                // Client-clock master
    Common           common;                    // Common simulation state
    uint8_t          rxFrameCount;              // Consistency-check count
```

```
    Precedence       precedence;                // Grand-master precedence
    SmallTime        syncInterval;              // Synchronization interval
    SmallTime        snapShot0;                 // Recent snapshot
    SmallTime        snapShot1;                 // Remote snapshot
} ClockMasterEntity;


// ***************************** ClockSlave entity ***********************************

typedef struct {                                // Client-clock slave
    SmallTime      syncInterval;
} ClockSlaveSaved;

typedef struct {                                // Client-clock slave
    Boolean          validated;                 // Validated; operational
    uint16_t         headIndex;                 // Recent interval index
    uint16_t         tailIndex;                 // Oldest interval index
    SmallTime        interval;                  // Rate-averaging interval
    BaseTimes        times[64];                 // txTimes value array
} BaseTimer;

typedef struct {                                // Client-clock slave
    Common           common;                    // Common simulation info
    uint8_t          frameCount;                // Consistency-check count
    SmallTime        syncInterval;              // Synchronization interval
    SmallTime        snapShot0;                 // Recent snapshot
    SmallTime        snapShot1;                 // Remote snapshot
    ClockSlaveSaved  rxSaved;                   // Saved rx information
    BaseTimer        baseTimer;                 // Receive-time history
} ClockSlaveEntity;


// ***************************** duplex-Ethernet *****************************

typedef struct {                                // EFDX receive
    SmallTime      thisTime;                    // Saved previous snapshot
    SmallTime      thatTime;                    // Saved previous thisTxTime
} PastTimes;

typedef struct {                                // EFDX receive
    Boolean        validated;                   // Validity indication
    uint16_t       headIndex;                   // Recent interval index
    uint16_t       tailIndex;                   // Oldest interval index
    SmallTime      interval;                    // Rate-averaging interval
    PastTimes      times[64];                   // Larger than ever needed
} RxTimer;

typedef struct {                                // EFDX receive
    Common         common;                      // Common simulation info
    Boolean        txReady;                     // Cable-delay valid
    LocalTime      syncInterval;                // Clock-master's tockTime
    uint16_t       snapCount;                   // The indication's frameCount
    uint16_t       frameCount;                  // The timeSync's frameCount
    SmallTime      snapShot0;                   // This frame's arrival time
    SmallTime      snapShot1;                   // Past frame's arrival time
```

```
    LocalTime      thisTxTime;                  // Frame transmission time
    LocalTime      thisRxTime;                  // Frame reception time
    LocalTime      thatRxTime;                  // Frame reception time
    EfdxMacInd     savedInd;                    // Received timeSync indication
    RxTimer        rxTimer;                     // For computing rateRatio (new)
} EfdxRxEntity;

typedef struct {                                // EFDX transmit
    uint64_t       da;                          // destination_address
    uint64_t       sa;                          // source_address
    uint16_t       type;                        // Received protocolType
    uint8_t        function;                    // Specified function (AVB)
    uint8_t        version;                     // Version number in AVB
    WideSigned     precedence;                  // Grand-master preference
    HopCount       hopCount;                    // Grand-master distance
    SmallTime      syncInterval;                // Sync transmit interval
} EfdxTxSaved;

typedef struct {                                // EFDX transmit
    Common         common;                      // Common simulation info
    GuessMode      guessMode;                   // Estimating next value
    Boolean        rxReady;                     // Sinking rx initialized
    Boolean        txReady;                     // Sending tx initialized
    SmallTime      execTime;                    // Next transmission time
    SmallTime      lastTime;                    // Periodic transmission time
    uint8_t        frameCount;                  // The timeSync frame count
    SmallTime      syncInterval;                // Sync interval duration
    EfdxTxSaved    rxSaved;                     // Received GrandSync request
    uint8_t        sxSnapCount;                 // Received MAC snapshot
    SmallTime      sxSnapTimed;                 // Received MAC snapshot
    BaseTimer      baseTimer;                   // Receive-time history
} EfdxTxEntity;


// ****************************** 802.11v radio ******************************

typedef struct {                                // 802.11v wireless receive
    Common         common;                      // Common simulation info
    LocalTime      syncInterval;                // Clock-master's tockTime
    TicksTime      turnRound;                   // Turn-round delay times
} R11vRxEntity;

typedef struct {                                // 802.11v wireless transmit
    WideSigned     precedence;                  // Grand-master preference
    HopCount       hopCount;                    // hopCount
    SmallTime      syncInterval;
} R11vTxSaved;

typedef struct {                                // 802.11v wireless transmit
    Common         common;                      // Common simulation info
    LocalTime      syncInterval;                // Clock-master's tockTime
    LocalTime      pastTime;                    // Back-interpolation time
    LocalTime      lastTime;                    // Last transmission
    R11vTxSaved    rxSaved;                     // Saved parameters
    BaseTimer      baseTimer;                   // Receive-time history
```

```
    TicksTime      snapShot1;                  // Saved ticksTime1
    TicksTime      roundTrip;                  // Saved ticksTime4-ticksTime1
    TicksTime      rxTurnRound;                // Turn-round delay times
    TicksTime      snapShot4;                  // Saved ticksTime4
    TicksTime      rxRoundTrip;                // Saved ticksTime4-ticksTime1
    Boolean        respondNow;
} R11vTxEntity;


// *************************** Ethernet-PON entity ***************************

typedef struct {                               // Etherent-PON receive
    Common         common;                     // Common simulation info
    LocalTime      syncInterval;               // Clock-master's tockTime
} EponRxEntity;

typedef struct {                               // Ethernet-PON transmit
    uint64_t       da;                         // destination_address
    uint64_t       sa;                         // source_address
    Precedence     precedence;                 // Grand-master precedence
    HopCount       hopCount;                   // hopCount
    SmallTime      syncInterval;
} EponTxSaved;

typedef struct {                               // Ethernet-PON transmit
    Common         common;                     // Common simulation info
    LocalTime      syncInterval;               // Clock-master's tockTime
    SmallTime      lastTime;                   // Last time checkpoint
    EponTxSaved    rxSaved;
    BaseTimer      baseTimer;                  // Receive-time history
} EponTxEntity;


// External control parameters
GuessMode argGuessMode = INTERPOLATE;
SmallTime argMegaHertz = 0;
uint16_t  argBridgeCount = 10;
uint32_t  argFirstSecs = 0;
uint32_t  argFinalSecs = 150;
uint32_t  argVocalType = VOCAL_DEBUG;


// Standard state-machine routines
Common        *GrandSyncExec(Common *, char *);
Common        *ClockMasterExec(Common *, char *);
Common        *ClockSlaveExec(Common *, char *);
Common        *EfdxRxExec(Common *, char *);
Common        *EfdxTxExec(Common *, char *);
Common        *EfdxSpanExec(Common *, char *);
Common        *R11vRxExec(Common *, char *);
Common        *R11vTxExec(Common *, char *);
Common        *EponRxExec(Common *, char *);
Common        *EponTxExec(Common *, char *);
```

```
void           CommonChecks(Common *, uint16_t, uint16_t);
Boolean        PortsCheck(Common *, char *);
Common        *CommonCreate(Common *, uint16_t,
                 Common *(Common *, char *), uint16_t, uint16_t, uint16_t);
Port           PortID(Common *);
TicksTime      EponTime(EponRxEntity  *);
TicksTime      R11vTime(R11vRxEntity *);

SmallTime      NextRate(RxTimer *, SmallTime, SmallTime, SmallTime, SmallTime);
void           NextSaved(BaseTimer *, SmallTime, SmallTime, LargeTime, SmallTime, SmallTime);
NextTimes      NextTimed(BaseTimer *, SmallTime, SmallTime);

// A minimalist double-width integer library
int            WideCompare(WideSigned, WideSigned);
int            WideCompareUnsigned(WideUnsigned, WideUnsigned);
int64_t        WideExtract(WideSigned, uint8_t);
WideSigned     WideAddition(WideSigned, WideSigned);
WideSigned     WideSubtract(WideSigned, WideSigned);
WideSigned     WideShift(WideSigned, int16_t);
WideSigned     WideNegate(WideSigned);
WideSigned     WideMultiply(int64_t, int64_t);

int64_t        DivideSmall(int64_t, int64_t);
int64_t        MultiplySmall(int64_t, int64_t);
SmallTime      ClockTicks(int64_t, uint64_t);

double         LargeToFloat(LargeTime);
GrandTime      LargeToGrand(LargeTime);
double         LocalToFloat(LocalTime);
SmallTime      LocalToSmall(LocalTime);
double         GrandToFloat(GrandTime);
LargeTime      GrandToLarge(GrandTime);
TinyTime       SmallToTiny(SmallTime);
double         SmallToFloat(SmallTime);
double         TinyToFloat(TinyTime);


// Other routines
uint64_t       Eui48ToEui64(uint64_t);
Precedence     FieldsToPrecedence(uint8_t, uint8_t, uint16_t, uint8_t, uint64_t);
WideSigned     FrameToValue(uint8_t *, uint16_t, Boolean);
Preference     FormPreference(WideSigned, uint8_t, uint8_t);
GrandTime      GrandToLevel(GrandTime);
GrandTime      LevelToGrand(GrandTime);
uint64_t       MacAddress(Common *);
SmallTime      RandomMagOne();
WideSigned     SignedToWide(int64_t);
Boolean        TimeSyncSdu(SyncSduData *);
void           ValueToFrame(WideSigned, uint8_t *, uint16_t);

Entry         *Bequeue(RxPort *);
uint32_t       Dequeue(RxPort *, void *, unsigned);
Boolean        Enqueue(TxPort *, void *, unsigned);
void           SleepOnRoot(Common *, LargeTime);
void           SleepOnBase(Common *, LocalTime);
```

```
char          *StrPair(char *, char *, char *, uint16_t);


// ********************************************************************************
// ************************* GrandSync state-machine routine **********************
// ********************************************************************************

Common *GrandSyncInit(Common *, char *);

// Sets common state to allow grandTime values to be back-interpolated
//   arguments:
//     comPtr - associated state-maintaining data structure
//     name -   initialization trigger and assigned entity name
Common *
GrandSyncExec(Common *comPtr, char *name) {
    uint8_t rxInfo[SizePlus(GrandSyncInd)], txInfo[SizePlus(GrandSyncReq)];
    GrandSyncEntity *ePtr = (GrandSyncEntity *)comPtr;
    GrandSyncInd   *rxPtr = (GrandSyncInd *)rxInfo;
    GrandSyncReq   *txPtr = (GrandSyncReq *)txInfo;
    GrandSyncSaved *sxPtr = &(ePtr->rxSaved);
    SyncSduData    *rsPtr = &(rxPtr->service_data_unit);
    SyncSduData    *tsPtr = &(txPtr->service_data_unit);
    RxPort    *rxQueuePtr;
    TxPort    *txQueuePtr;
    GrandTime  rxGrandTime;
    Preference test, past;
    Precedence rxPrecedence;
    SmallTime  nextTime, rxSmallTime, rxSyncInterval, stationTime;
    TinyTime   rxExtraTime;
    HopCount   rxHopCount;
    Port       rxSourcePort;
    uint64_t   macAddress;
    uint32_t   sized;
    uint16_t   accelerated, count, hopCount;
    Boolean    serviced, better;

    if (name != NULL)                                                    // Initialization
        return(GrandSyncInit(comPtr, name));
    SetRxQueue1Ptrs(comPtr, rxQueuePtr);
    SetTxQueue1Ptrs(comPtr, txQueuePtr);

    do {
        serviced = FALSE;
        // ***************************************************************************
        // ***** Processing arriving GrandSync MA_UNITDATA.indication parameters *******
        // ***************************************************************************
        stationTime = StationTime(comPtr);                               // Station's localTime
        sized = DeQueue(rxQueuePtr, &rxInfo);                            // Check rx queue
        if (sized == sizeof(GrandSyncInd) && TimeSyncSdu((SyncSduData *)rsPtr)) {   // Verify parameters
            serviced = TRUE;
            rxPrecedence =   FieldToUnsign(rsPtr, precedence);           // Precedence value
            rxHopCount =     FieldToUnsign(rsPtr, hopCount).lower;       // Hop-count distance
            rxSourcePort =   FieldToUnsign(rsPtr, sourcePort).lower;     // Received port identifier
            rxGrandTime =    FieldToSigned(rsPtr, grandTime);            // Grand-master time
            rxExtraTime =    FieldToSigned(rsPtr, extraTime).lower;      // Extra part of grandTime
```

```
            rxSmallTime =    FieldToSigned(rsPtr, smallTime).lower;                                  // Internal station time
            rxSyncInterval = FieldToUnsign(rsPtr, syncInterval).lower;                               // Sync-interval time

            test = FormPreference(rxPrecedence, rxHopCount, rxSourcePort);                           // Test preference
            past = FormPreference(sxPtr->precedence, sxPtr->hopCount, sxPtr->sourcePort);            // Past preference
            better = rxSourcePort == sxPtr->sourcePort || WideCompareUnsigned(test, past) <= 0;      // This one is better
            if (rxHopCount != LAST_HOP && better) {                                                  // and is also valid
                ePtr->lastTime = stationTime;                                                        // Update timeout timer
                count = sxPtr->hopCount;                                                             // Saved hopCount value
                sxPtr->precedence = rxPrecedence;                                                    // Saved GM values
                sxPtr->sourcePort = rxSourcePort;                                                    //  ...
                sxPtr->hopCount = rxHopCount;                                                        //  ...
                sxPtr->syncInterval = rxSyncInterval;                                                //  ...

                accelerated = 1 + (LAST_HOP + rxHopCount) / 2;                                       // Accelerated aging
                hopCount = MIN(LAST_HOP, rxHopCount > count ? accelerated : rxHopCount + 1);        // if receiver is aged

                // ************************************************************************
                // ******* Create and transmit MA_UNITDATA.indication parameters *******
                // ************************************************************************
                macAddress = MacAddress(comPtr);
                LongToFrame(AVB_MCAST,      txPtr, destination_address);                             // Neighbor multicast address
                LongToFrame(macAddress,     txPtr, source_address);                                  // This port's MAC address
                LongToFrame(AVB_PROTOCOL,   tsPtr, protocolType);                                    // The AVB protocol
                LongToFrame(AVB_FUNCTION,   tsPtr, function);                                        // The timeSync frame in AVB
                LongToFrame(AVB_VERSION,    tsPtr, version);                                         // This version number
                WideToFrame(rxPrecedence,   tsPtr, precedence);                                      // Create new precedende
                LongToFrame(hopCount,       tsPtr, hopCount);                                        // Create GM distance
                LongToFrame(rxSourcePort,   tsPtr, sourcePort);                                      // Create port indentifier
                WideToFrame(rxGrandTime ,   tsPtr, grandTime);                                       // Echo grandTime
                LongToFrame(rxExtraTime,    tsPtr, extraTime);                                       // Echo extraTime
                LongToFrame(rxSmallTime,    tsPtr, smallTime);                                       // Echo localTime
                LongToFrame(rxSyncInterval, tsPtr, syncInterval);                                    // Echo synch interval
                EnQueue(txQueuePtr, txPtr);                                                          // Enqueue the result;
            }
        }

        assert(sxPtr->syncInterval != 0);                                                            // Consistency check
        nextTime = ePtr->lastTime + 4 * sxPtr->syncInterval;                                         // Timeout threshold
        if ((stationTime - nextTime) >= 0) {                                                         // Timeout actions
            serviced = TRUE;
            sxPtr->precedence.upper = sxPtr->precedence.lower = ONES;                                // Worst precedence &
            sxPtr->hopCount = sxPtr->sourcePort = 0XFF;                                              // worst tie-breakers
            ePtr->lastTime = stationTime;                                                            // Resets the timeout
            // printf("GrandSync (timeout):\n");
        }
    } while (serviced == TRUE);
    assert((nextTime - stationTime) > 0);
    SleepOnBase(comPtr, nextTime);
    return(NULL);
}

Common *
GrandSyncInit(Common *oldPtr, char *string) {
    Common        *comPtr;
```

```
    GrandSyncEntity *ePtr;
    GrandSyncSaved *sxPtr;
    char temp[TLIMIT+1], name[NLIMIT+1], data[NLIMIT+1], *nextPtr;

    comPtr = CommonCreate(oldPtr, sizeof(GrandSyncEntity), &GrandSyncExec, TYPE_ENTITY, Q_RX11_LAST, Q_TX11_LAST);
    if (comPtr != NULL) {
        ePtr = (GrandSyncEntity *)comPtr;
        sxPtr = &(ePtr->rxSaved);
        sxPtr->syncInterval = SMALL_10ms;
        sxPtr->precedence.upper = sxPtr->precedence.lower = ONES;

        nextPtr = StrPair(strncpy(temp, string, TLIMIT), name, data, NLIMIT);
        for ( ; nextPtr != NULL && name[0] != '\0'; nextPtr = StrPair(nextPtr, name, data, NLIMIT)) {
            if (strcmp(name, "name") == 0)
                strcpy(comPtr->name, data);
            else if (strcmp(name, "rx0") == 0)
                assert(comPtr->rxPortCount > 0), strcpy(comPtr->rxPortPtr[0].name, data);
            else if (strcmp(name, "tx0") == 0)
                assert(comPtr->txPortCount > 0), strcpy(comPtr->txPortPtr[0].name, data);
        }
        assert(PortsCheck(comPtr, nextPtr) == TRUE);
    }
    return(comPtr);
}


// ********************************************************************************
// ********************** ClockMaster state-machine routines **********************
// ********************************************************************************

Common *ClockMasterInit(Common *, char *);
// Provide the clock source information, retransmits to GrandSync
//   arguments:
//     comPtr - associated state-maintaining data structure
//     name -   initialization trigger and assigned entity name
Common *
ClockMasterExec(Common *comPtr, char *name) {
    uint8_t cmInfo[SizePlus(ClockMasterSet)], txInfo[SizePlus(GrandSyncInd)];
    ClockMasterEntity *ePtr = (ClockMasterEntity *)comPtr;
    ClockMasterSet   *rxPtr = (ClockMasterSet *)cmInfo;
    GrandSyncInd     *txPtr = (GrandSyncInd *)txInfo;
    SyncSduData      *tsPtr = &(txPtr->service_data_unit);
    RxPort   *rxQueuePtr;
    TxPort   *txQueuePtr;
    GrandTime grandTime;
    SmallTime stationTime;
    uint64_t  macAddress;
    uint32_t  sized;
    uint16_t  count, frameCount;
    uint8_t   portID;
    Boolean   serviced;

    if (name != NULL)                                                    // The entity name
        return(ClockMasterInit(comPtr, name));                           // for initialization
    SetRxQueue1Ptrs(comPtr, rxQueuePtr);
```

```
    SetTxQueue1Ptrs(comPtr, txQueuePtr);

    macAddress = MacAddress(comPtr);
    portID = PortID(comPtr);

    do {
        serviced = FALSE;
        // ********************************************************************************
        // ***** Processing arriving GrandSync clock-master indication parameters ******
        // ********************************************************************************
        stationTime = StationTime(comPtr);
        sized = DeQueue(rxQueuePtr, rxPtr);
        if (sized > 0) {
            serviced = TRUE;
            assert(sized == sizeof(ClockMasterSet));
            ePtr->snapShot1 = ePtr->snapShot0;
            ePtr->snapShot0 = stationTime;
            count = (ePtr->rxFrameCount + 1) % COUNT;
            frameCount = FieldToUnsign(rxPtr, frameCount).lower;
            grandTime  = FieldToSigned(rxPtr, grandTime);
            ePtr->rxFrameCount = frameCount;
            if (count == frameCount) {

                // ********************************************************************************
                // ****** Creation and transmit of MA_UNITDATA.indication parameters *******
                // ********************************************************************************

                LongToFrame(AVB_MCAST,           txPtr, destination_address);                    // Neighbor multicast address
                LongToFrame(macAddress,          txPtr, source_address);                         // This port's MAC address
                LongToFrame(AVB_PROTOCOL,        tsPtr, protocolType);                           // The AVB protocol
                LongToFrame(AVB_FUNCTION,        tsPtr, function);                               // The timeSync frame in AVB
                LongToFrame(AVB_VERSION,         tsPtr, version);                                // This version number
                WideToFrame(ePtr->precedence,    tsPtr, precedence);                             // Create new precedence
                LongToFrame(0,                   tsPtr, hopCount);                               // Initial GM distance
                LongToFrame(portID,              tsPtr, sourcePort);                             // Create port indentifier
                WideToFrame(grandTime,           tsPtr, grandTime);                              // Report grandTime
                LongToFrame(ePtr->snapShot1,     tsPtr, smallTime);                              // Report smallTime
                LongToFrame(0,                   tsPtr, extraTime);                              // Initial extraTime
                LongToFrame(ePtr->syncInterval,  tsPtr, syncInterval);                           // Sync-frame interval
                // TBD: fcs, priority, ...
                EnQueue(txQueuePtr, txPtr);                                                      // Enqueue the result;
            }
        }
    } while (serviced == TRUE);
    return(NULL);
}

Common *
ClockMasterInit(Common *oldPtr, char *string) {
    Common          *comPtr;
    ClockMasterEntity *ePtr;
    uint64_t  macAddress;
    char temp[TLIMIT+1], name[NLIMIT+1], data[NLIMIT+1], *nextPtr;

    comPtr = CommonCreate(oldPtr, sizeof(ClockMasterEntity), &ClockMasterExec, TYPE_RX_PORT, Q_RX11_LAST, Q_TX11_LAST);
```

```
    if (comPtr != NULL) {
        macAddress = MacAddress(comPtr);                                    // Get MAC address
        comPtr->portID = CLOCK_MASTER_PORT_ID;                              // Set port identifier
        comPtr->portLink = comPtr;
        ePtr = (ClockMasterEntity *)comPtr;
        ePtr->syncInterval = SMALL_10ms;
        ePtr->precedence.upper = 0;
        ePtr->precedence.lower = Eui48ToEui64(macAddress);

        nextPtr = StrPair(strncpy(temp, string, TLIMIT), name, data, NLIMIT);
        for ( ; name[0] != '\0'; nextPtr = StrPair(nextPtr, name, data, NLIMIT)) {
            if (strcmp(name, "name") == 0)
                strncpy(comPtr->name, data, NLIMIT);
            else if (strcmp(name, "rx0") == 0)
                assert(comPtr->rxPortCount >= 1), strcpy(comPtr->rxPortPtr[0].name, data);
            else if (strcmp(name, "tx0") == 0)
                assert(comPtr->txPortCount >= 1), strcpy(comPtr->txPortPtr[0].name, data);
        }
        assert(PortsCheck(comPtr, nextPtr) == TRUE);
    }
    return(comPtr);
}


// ********************************************************************************
// ********************** ClockSlave state-machine routines ***********************
// ********************************************************************************

Common *ClockSlaveInit(Common *, char *);
// Generates a GrandSync indication, after being triggered
//   arguments:
//     comPtr - associated state-maintaining data structure
//     name -   initialization trigger and assigned entity name
Common *
ClockSlaveExec(Common *comPtr, char *name) {
    uint8_t sxInfo[SizePlus(ClockSlaveReq)], rxInfo[SizePlus(GrandSyncReq)], txInfo[SizePlus(ClockSlaveRes)];
    ClockSlaveEntity *ePtr = (ClockSlaveEntity *)comPtr;
    ClockSlaveSaved *rcPtr = &(ePtr->rxSaved);
    BaseTimer       *btPtr = &(ePtr->baseTimer);
    ClockSlaveReq   *sxPtr = (ClockSlaveReq *)sxInfo;
    GrandSyncReq    *rxPtr = (GrandSyncReq *)rxInfo;
    ClockSlaveRes   *txPtr = (ClockSlaveRes *)txInfo;
    SyncSduData     *rsPtr = &(rxPtr->service_data_unit);
    RxPort    *cxPortPtr, *rxQueuePtr;
    TxPort    *txQueuePtr;
    NextTimes  nextTimes;
    LargeTime  systemTime;
    GrandTime  grandTime;
    SmallTime  backInterval, rateInterval, smallTime, stationTime, syncInterval;
    TinyTime   extraTime;
    uint32_t   sized;
    uint8_t    frameCount;
    Boolean    serviced;

    if (name != NULL)                                                       // The entity's name
```

```
        return(ClockSlaveInit(comPtr, name));                                                   // for initialization

    SetRxQueue2Ptrs(comPtr, rxQueuePtr, cxPortPtr);
    SetTxQueue1Ptrs(comPtr, txQueuePtr);
    stationTime = StationTime(comPtr);
    do {
        serviced = FALSE;
        systemTime = SystemTime(comPtr);
        backInterval = (3 * rcPtr->syncInterval + ePtr->syncInterval) / 2;
        rateInterval = backInterval + (3 * ePtr->syncInterval) / 2;

        // ************************************************************************
        // ******* Processing arriving GrandSync ClockSlave.request parameters *********
        // ************************************************************************
        sized = DeQueue(cxPortPtr, sxPtr);
        if (sized != 0) {
            serviced = TRUE;
            assert(sized == sizeof(ClockSlaveReq));
            frameCount = FieldToUnsign(sxPtr, frameCount).lower;
            nextTimes =  NextTimed(btPtr, stationTime, backInterval);
            grandTime = nextTimes.totalTime;
            LongToFrame(frameCount, txPtr, frameCount);                                          // Tag from the request
            WideToFrame(grandTime,  txPtr, grandTime);                                           // Associated grandTime
            EnQueue(txQueuePtr, txPtr);                                                          // Enqueue the result;
        }

        // ************************************************************************
        // ******* Processing arriving GrandSync MA_UNITDATA.request parameters ********
        // ************************************************************************
        sized = DeQueue(rxQueuePtr, rxPtr);
        if (sized == sizeof(GrandSyncReq) && TimeSyncSdu((SyncSduData *)rsPtr)) {
            serviced = TRUE;
            syncInterval = FieldToSigned(rsPtr, syncInterval).lower;
            grandTime =    FieldToSigned(rsPtr, grandTime);
            extraTime =    FieldToSigned(rsPtr, extraTime).lower;
            smallTime =    FieldToSigned(rsPtr, smallTime).lower;
            rcPtr->syncInterval = syncInterval;
            NextSaved(btPtr, ePtr->syncInterval, rateInterval, GrandToLarge(grandTime), TinyToSmall(extraTime), smallTime);
        }
    } while (serviced == TRUE);                                                             // Return tagged indication
    return(NULL);
}

Common *
ClockSlaveInit(Common *oldPtr, char *string) {
    Common         *comPtr;
    ClockSlaveEntity *ePtr;
    BaseTimer        *btPtr;
    char temp[TLIMIT+1], name[NLIMIT+1], data[NLIMIT+1], *nextPtr;

    comPtr = CommonCreate(oldPtr, sizeof(ClockSlaveEntity), &ClockSlaveExec, TYPE_TX_PORT, Q_RX22_LAST, Q_TX11_LAST);
    if (comPtr != NULL) {
        ePtr = (ClockSlaveEntity *)comPtr;
        ePtr->syncInterval = SMALL_10ms;
        ePtr->rxSaved.syncInterval = SMALL_10ms;
```

```
        btPtr = &(ePtr->baseTimer);
        btPtr->validated = 0;
        btPtr->headIndex = btPtr->tailIndex = 0;

        nextPtr = StrPair(strncpy(temp, string, TLIMIT), name, data, NLIMIT);
        for ( ; name[0] != '\0'; nextPtr = StrPair(nextPtr, name, data, NLIMIT)) {
            if (strcmp(name, "name") == 0)
                strncpy(comPtr->name, data, NLIMIT);
            else if (strcmp(name, "rx0") == 0)
                assert(comPtr->rxPortCount > 0), strcpy(comPtr->rxPortPtr[0].name, data);
            else if (strcmp(name, "rx1") == 0)
                assert(comPtr->rxPortCount > 1), strcpy(comPtr->rxPortPtr[1].name, data);
            else if (strcmp(name, "tx0") == 0)
                assert(comPtr->txPortCount >= 1), strcpy(comPtr->txPortPtr[0].name, data);
        }
        assert(PortsCheck(comPtr, nextPtr) == TRUE);
    }
    return(comPtr);
}


// ********************************************************************************
// ********************* Ethernet-duplex state-machine routines ***********************
// ********************************************************************************

Common *EfdxRxInit(Common *, char *);
// Receives duplex-Ethernet SDUs, retransmits them to GrandSync
//   arguments:
//     comPtr - associated state-maintaining data structure
//     name -   initialization trigger and assigned entity name
Common *
EfdxRxExec(Common *comPtr, char *name) {
    uint8_t    rxInfo[MTU_SIZED], txInfo[MTU_SIZED], sxInfo[SizePlus(EfdxRxInfo)];
    EfdxRxEntity  *ePtr =  (EfdxRxEntity *)comPtr;
    RxTimer      *btPtr = &(ePtr->rxTimer);
    EfdxRxInfo   *sxPtr =  (EfdxRxInfo *)sxInfo;
    EfdxMacInd   *rxPtr =  (EfdxMacInd *)rxInfo;
    GrandSyncInd *txPtr =  (GrandSyncInd *)txInfo;
    EfdxSduData  *rsPtr = &(rxPtr->service_data_unit);
    SyncSduData  *tsPtr = &(txPtr->service_data_unit);
    EfdxMacInd   *dxPtr = &(ePtr->savedInd);
    EfdxSduData  *dsPtr = &(dxPtr->service_data_unit);
    RxPort    *rxQueuePtr, *sxQueuePtr;
    TxPort    *txQueuePtr;
    Precedence precedence;
    GrandTime  grandTime;
    SmallTime  cableDelay, compRxTime, rateRatio, smallTime, stationTime, thisRxTime, thisTxTimed;
    LocalTime  roundTrip, thatTxTime, thatRxTime, thisTxTime, turnRound, turnRound0;
    TinyTime   extraTime;
    uint64_t   da, sa;
    uint32_t   sized;
    uint16_t   frameCount, protocol, rxHeadSize, snapCount, txHeadSize;
    uint8_t    function, guess, hopCount, portID, version;
    Boolean    countsAreEqual, serviced;
```

```
if (name != NULL)                                                          // The entity's name
    return(EfdxRxInit(comPtr, name));                                      // for initialization.
SetRxQueue2Ptrs(comPtr, rxQueuePtr, sxQueuePtr);
SetTxQueue1Ptrs(comPtr, txQueuePtr);

portID = PortID(comPtr);
countsAreEqual = 0;
do {
    stationTime = StationTime(comPtr);                                     // Station's localTime
    serviced = FALSE;

    // ****************************************************************************
    // ****************** Processing arriving timeSync snapshots ******************
    // ****************************************************************************
    sized = DeQueue(sxQueuePtr, &sxInfo);
    if (sized != 0) {
        serviced = TRUE;
        assert(sized == sizeof(EfdxRxInfo));
        snapCount = FieldToUnsign(sxPtr, frameCount).lower;
        smallTime = FieldToUnsign(sxPtr, smallTime).lower;
        ePtr->snapShot1 = ePtr->snapShot0;
        ePtr->snapShot0 = smallTime;
        ePtr->snapCount = snapCount;
        countsAreEqual = (snapCount == ePtr->frameCount);
    }

    // ****************************************************************************
    // ******************** Processing arriving timeSync frames *******************
    // ****************************************************************************
    sized = Dequeue(rxQueuePtr, rxPtr, sizeof(rxInfo));
    if (sized != 0) {
        serviced = TRUE;
        if (sized == sizeof(EfdxMacInd) && TimeSyncSdu((SyncSduData *)rsPtr)) {  // Any timeSync frame
            frameCount = FieldToUnsign(rsPtr, frameCount).lower;               // has an actual and
            guess = (ePtr->frameCount + 1) % COUNT;                            // expected frameCount
            ePtr->frameCount = frameCount;                                     // Update frameCount
            if (frameCount == guess) {                                         // An expected frame
                bcopy(rxPtr, dxPtr, sizeof(EfdxMacInd));                       // Save valid frames
                countsAreEqual = (frameCount == ePtr->snapCount);              // Enable processing
            }
        } else {
            da = FieldToUnsign(rxPtr, destination_address).lower;              // Extract destination
            sa = FieldToUnsign(rxPtr, source_address).lower;                   // Extract source
            LongToFrame(da, txPtr, destination_address);                       // Deposit destination
            LongToFrame(sa, txPtr, source_address);                            // Deposit source
            rxHeadSize = (void *)rsPtr - (void *)rxPtr;                        // Received header size
            txHeadSize = (void *)tsPtr - (void *)txPtr;                        // Transmit header size
            bcopy(rsPtr, tsPtr, sized - rxHeadSize);                           // Copy entire SDU
            Enqueue(txQueuePtr, txPtr, sized + txHeadSize - rxHeadSize);       // Enqueue the contents
        }
    }

    if (countsAreEqual == TRUE) {
        serviced = TRUE;
```

```
            countsAreEqual = FALSE;
            da =         FieldToUnsign(dxPtr, destination_address).lower;
            sa =         FieldToUnsign(dxPtr, source_address).lower;
            protocol =   FieldToUnsign(dsPtr, protocolType).lower;
            function =   FieldToUnsign(dsPtr, function).lower;
            version =    FieldToUnsign(dsPtr, version).lower;
            precedence = FieldToUnsign(dsPtr, precedence);
            hopCount =   FieldToUnsign(dsPtr, hopCount).lower;
            frameCount = FieldToUnsign(dsPtr, frameCount).lower;
            thatTxTime = FieldToSigned(dsPtr, thatTxTime).lower;
            thatRxTime = FieldToSigned(dsPtr, thatRxTime).lower;
            grandTime =  FieldToSigned(dsPtr, grandTime);
            extraTime =  FieldToSigned(dsPtr, extraTime).lower;
            thisTxTime = FieldToSigned(dsPtr, localTime).lower;
            thisRxTime = ePtr->snapShot1;
            ePtr->thisTxTime = thisTxTime;                                          // Saved for transmit
            ePtr->thisRxTime = SmallToLocal(thisRxTime);                            // over returning link

            if (ePtr->thatRxTime != thatRxTime) {
                ePtr->thatRxTime = thatRxTime;
                ePtr->txReady = TRUE;
            }
            thisTxTimed = thisRxTime - LocalToSmall(SmallToLocal(thisRxTime) - thisTxTime);
            if (ePtr->txReady == TRUE)
                rateRatio = NextRate(btPtr, thisRxTime, thisTxTimed, ePtr->syncInterval, btPtr->interval);
            else
                rateRatio = SMALL_ONE;

            if (btPtr->validated != TRUE || ePtr->txReady != TRUE)
                hopCount = LAST_HOP;
            roundTrip =  LocalToSmall(ePtr->thisRxTime - thatTxTime);               // Round-trip delay
            turnRound0 = LocalToSmall(thisTxTime - thatRxTime);                     // Turn-around delay
            turnRound = MultiplySmall(turnRound0, rateRatio);                       // Normalized turnRound
            cableDelay = MAX(0, roundTrip - turnRound) / 2;                         // Cable-delay computed
            compRxTime = thisRxTime - cableDelay;                                   // Cable-delay adjustment

            // *************************************************************************
            // ************** Update revised service-data-unit parameters **************
            // *************************************************************************
            LongToFrame(da,                 txPtr, destination_address);            // The destination address
            LongToFrame(sa,                 txPtr, source_address);                 // The source address
            LongToFrame(protocol,           tsPtr, protocolType);                   // The protocol identifier
            LongToFrame(function,           tsPtr, function);                       // The function identifier
            LongToFrame(version,            tsPtr, version);                        // The function identifier
            WideToFrame(precedence,         tsPtr, precedence);                     // GM selection precedence
            LongToFrame(hopCount,           tsPtr, hopCount);                       // GM hop-count distance
            LongToFrame(portID,             tsPtr, sourcePort);                     // Source-port identifier
            WideToFrame(grandTime,          tsPtr, grandTime);                      // grandTime at snapShot
            LongToFrame(extraTime,          tsPtr, extraTime);                      // Next extraTime value
            LongToFrame(compRxTime,         tsPtr, smallTime);                      // Transmitted frame time
            LongToFrame(ePtr->syncInterval, tsPtr, syncInterval);                   // Sync transmit interval
            EnQueue(txQueuePtr, txPtr);                                             // Enqueue the result
        }
    } while (serviced == TRUE);
    return(NULL);
```

```
}

Common *
EfdxRxInit(Common *oldPtr, char *string) {
    Common     *comPtr;
    EfdxRxEntity *ePtr;
    RxTimer     *btPtr;
    char temp[TLIMIT+1], name[NLIMIT+1], data[NLIMIT+1], *nextPtr;

    comPtr = CommonCreate(oldPtr, sizeof(EfdxRxEntity), &EfdxRxExec, TYPE_RX_PORT, Q_RX22_LAST, Q_TX11_LAST);
    if (comPtr != NULL) {
        ePtr = (EfdxRxEntity *)comPtr;
        ePtr->syncInterval = SMALL_10ms;
        ePtr->snapCount = ePtr->frameCount = -1;
        ePtr->txReady = FALSE;

        btPtr = &(ePtr->rxTimer);
        btPtr->validated = FALSE;
        btPtr->interval = SMALL_10ms * 20;
        btPtr->headIndex = btPtr->tailIndex = 0;

        nextPtr = StrPair(strncpy(temp, string, TLIMIT), name, data, NLIMIT);
        for ( ; name[0] != '\0'; nextPtr = StrPair(nextPtr, name, data, NLIMIT)) {
            if (strcmp(name, "name") == 0)
                strcpy(comPtr->name, data);
            else if (strcmp(name, "rx0") == 0)
                assert(comPtr->rxPortCount > 0), strcpy(comPtr->rxPortPtr[0].name, data);
            else if (strcmp(name, "rx1") == 0)
                assert(comPtr->rxPortCount > 1), strcpy(comPtr->rxPortPtr[1].name, data);
            else if (strcmp(name, "tx0") == 0)
                assert(comPtr->txPortCount > 0), strcpy(comPtr->txPortPtr[0].name, data);
        }
        assert(PortsCheck(comPtr, nextPtr) == TRUE);
    }
    return(comPtr);
}

SmallTime
NextRate(RxTimer *btPtr, SmallTime thisRxTime, SmallTime thisTxTime, SmallTime interval0, SmallTime interval1)
{
    PastTimes *timePtr;
    SmallTime rateRatio0, rateRatio1, thatDelta, thisDelta;
    uint16_t headIndex, tailIndex, lastIndex, limit;
    uint8_t i;

    assert(btPtr != NULL);                                                  // Verify the pointer
    timePtr = btPtr->times;                                                 // Array value pointer
    limit = ARRAY_SIZE(btPtr->times);                                       // Array-size limits

    if (btPtr->headIndex == btPtr->tailIndex) {                             // Unitialized array
        assert(btPtr->validated == FALSE);                                  // has no validated
        btPtr->headIndex = 1, btPtr->tailIndex = 0;                         // Initialize index
        timePtr[0].thisTime = thisRxTime;                                   // and tail-indexed
        timePtr[0].thatTime = thisTxTime;                                   // data values
    }
```

```
    headIndex = btPtr->headIndex;                                                   // The head and last
    lastIndex = PLUS(headIndex, -1, limit);                                         // index values within
    assert(headIndex < limit && lastIndex < limit);                                 // the circular buffer

    if ((timePtr[headIndex].thatTime - timePtr[lastIndex].thatTime) > (interval0 / 2)) {  // Time to advance
        btPtr->headIndex = headIndex = PLUS(headIndex, 1, limit);                   // increment headIndex
        btPtr->validated = TRUE;                                                    // Set when ready
    }
    timePtr[headIndex].thisTime = thisRxTime;                                       // Save received time
    timePtr[headIndex].thatTime = thisTxTime;                                       // Save transmit time
    if (btPtr->validated == FALSE)                                                  // Until times change,
        return(SMALL_ONE);                                                          // assume slope==1

    for (i = 0; i < 2; i += 1, btPtr->tailIndex = tailIndex) {                      // Check tailIndex twice
        tailIndex = PLUS(btPtr->tailIndex, 1, limit);                               // Next tailIndex value
        if (tailIndex == headIndex)                                                 // The tailIndex can
            break;                                                                  // never equal headIndex
         if (thisTxTime - timePtr[tailIndex].thatTime <= interval1)                 // Update tailIndex if
            break;                                                                  // range is maintained
    }
    tailIndex = btPtr->tailIndex;

    thisDelta = thisRxTime - timePtr[tailIndex].thisTime;                           // Received interval
    thatDelta = thisTxTime - timePtr[tailIndex].thatTime;                           // Transmit interval
    assert(thatDelta != 0);                                                         // Must have changed
    rateRatio0 = DivideSmall(thisDelta, thatDelta);                                 // Compute the rate
    rateRatio1 = CLIP_RATE(rateRatio0, PPM250);                                     // Clip within 250PPM
    return(rateRatio1);
}

Common *EfdxTxInit(Common *, char *);
// Receives GrandSync SDUs, retransmits them as duplex-Ethernet SDUs
//   arguments:
//     comPtr - associated state-maintaining data structure
//     name -   initialization trigger and assigned entity name
Common *
EfdxTxExec(Common *comPtr, char *name) {
    uint8_t txInfo[MTU_SIZED], rxInfo[MTU_SIZED], sxInfo[SizePlus(EfdxTxInfo)];
    EfdxTxEntity  *ePtr = (EfdxTxEntity *)comPtr;
    BaseTimer    *btPtr = &(ePtr->baseTimer);
    GrandSyncReq *rxPtr = (GrandSyncReq *)rxInfo;
    EfdxTxSaved  *rcPtr = &(ePtr->rxSaved);
    EfdxMacReq   *txPtr = (EfdxMacReq *)txInfo;
    SyncSduData  *rsPtr = &(rxPtr->service_data_unit);
    EfdxSduData  *tsPtr = &(txPtr->service_data_unit);
    EfdxTxInfo   *sxPtr = (EfdxTxInfo *)sxInfo;
    EfdxRxEntity *dPtr;
    RxPort    *rxQueuePtr, *sxQueuePtr;
    TxPort    *txQueuePtr;
    NextTimes  nextTimes;
    GrandTime  grandTime;
    SmallTime  backInterval, execTime, nextTime, rateInterval, smallTime, stationTime, wakeTime;
    LocalTime  localTime;
    TinyTime   extraTime;
    uint16_t   rxHeadSize, txHeadSize, size, sized;
```

```
    Boolean    serviced;

    if (name != NULL)                                                       // The entity name
        return(EfdxTxInit(comPtr, name));                                   // for initialization.
    SetRxQueue2Ptrs(comPtr, rxQueuePtr, sxQueuePtr);
    SetTxQueue1Ptrs(comPtr, txQueuePtr);
    assert((dPtr = (EfdxRxEntity *)comPtr->pairLink) != NULL);              // Associated receiver

    do {
        serviced = FALSE;
        stationTime = StationTime(comPtr);                                  // Station's localTime
        backInterval = (3 * rcPtr->syncInterval + ePtr->syncInterval) / 2;
        rateInterval = backInterval + (3 * ePtr->syncInterval) / 2;

        // ********************************************************************************
        // ****************** Processing arriving timeSync snapshots ******************
        // ********************************************************************************
        sized = DeQueue(sxQueuePtr, &sxInfo);
        if (sized != 0) {
            serviced = TRUE;
            assert(sized == sizeof(EfdxTxInfo));
            ePtr->sxSnapCount = FieldToUnsign(sxPtr, frameCount).lower;
            ePtr->sxSnapTimed = FieldToUnsign(sxPtr, smallTime).lower;
            ePtr->txReady = TRUE;
        }

        // ********************************************************************************
        // *************** Processing arrived MS_DATAUNIT.request frames ***************
        // ********************************************************************************
        sized = DeQueue(rxQueuePtr, &rxInfo);
        if (sized != 0) {
            serviced = TRUE;
            if (sized != sizeof(GrandSyncReq) || !TimeSyncSdu(rsPtr)) {
                LongToFrame(rcPtr->da, txPtr, destination_address);         // The destination_address
                LongToFrame(rcPtr->sa, txPtr, source_address);              // The source_address
                rxHeadSize = (void *)rsPtr - (void *)rxPtr;                 // Received header size
                txHeadSize = (void *)tsPtr - (void *)txPtr;                 // Transmit header size
                bcopy(rsPtr, tsPtr, sized - rxHeadSize);                    // Copy entire SDU
                size = sized + txHeadSize - rxHeadSize;                     // Size of enqueued frame
                Enqueue(txQueuePtr, txPtr, size);                           // Enqueue for transmission
            } else {
                rcPtr->da =           FieldToUnsign(rxPtr, destination_address).lower;
                rcPtr->sa =           FieldToUnsign(rxPtr, source_address).lower;
                rcPtr->type =         FieldToUnsign(rsPtr, protocolType).lower;
                rcPtr->function =     FieldToUnsign(rsPtr, function).lower;
                rcPtr->version =      FieldToUnsign(rsPtr, version).lower;
                rcPtr->hopCount =     FieldToUnsign(rsPtr, hopCount).lower;
                rcPtr->precedence =   FieldToUnsign(rsPtr, precedence);
                rcPtr->syncInterval = FieldToUnsign(rsPtr, syncInterval).lower;
                grandTime =           FieldToSigned(rsPtr, grandTime);
                extraTime =           FieldToSigned(rsPtr, extraTime).lower;
                smallTime =           FieldToSigned(rsPtr, smallTime).lower;
                NextSaved(btPtr, ePtr->syncInterval, rateInterval,
                 GrandToLarge(grandTime), TinyToSmall(extraTime), smallTime);
```

```
//              if (ePtr->guessMode == EXTRAPOLATE)
//                  ePtr->execTime = stationTime + RESIDENCE_DELAY;
            }
        }

        // ***************************************************************************
        // ****************** Preparing transmitted timeSync frames ******************
        // ***************************************************************************
        nextTime = ePtr->lastTime + ePtr->syncInterval;
        if ((stationTime - nextTime) >= 0) {                                    // Next sync transmission
            serviced = TRUE;
            ePtr->lastTime = nextTime;                                          // Restart 10ms timer
            ePtr->execTime = stationTime + RESIDENCE_DELAY;
        }
        execTime = ePtr->execTime;
        if ((stationTime - execTime) >= 0) {
            serviced = TRUE;
            ePtr->execTime = stationTime + LARGE_TOCK;                          // Indefinite future
            ePtr->frameCount = (ePtr->frameCount + 1) % COUNT;                  // Incremented counter
            if (!ePtr->txReady)
                ePtr->sxSnapTimed = (stationTime - ePtr->syncInterval);
            localTime = SmallToLocal(ePtr->sxSnapTimed);
            switch(ePtr->guessMode) {
            case INTERPOLATE:
                nextTimes =  NextTimed(btPtr, ePtr->sxSnapTimed, backInterval);
                break;
            case EXTRAPOLATE:
                nextTimes = NextTimed(btPtr, ePtr->sxSnapTimed, (SmallTime)0);
                break;
            }

            LongToFrame(rcPtr->da,           txPtr, destination_address);       // The destination_address
            LongToFrame(rcPtr->sa,           txPtr, source_address);            // The source_address
            LongToFrame(AVB_PROTOCOL,        tsPtr, protocolType);              // The protocol identifier
            LongToFrame(AVB_FUNCTION,        tsPtr, function);                  // The basic function
            LongToFrame(AVB_VERSION,         tsPtr, version);                   // and version identifier
            WideToFrame(rcPtr->precedence,   tsPtr, precedence);                // Grand-master precedence
            LongToFrame(rcPtr->hopCount,     tsPtr, hopCount);                  // The ~GM distance.
            LongToFrame(ePtr->frameCount,    tsPtr, frameCount);                // Source-port identifier
            WideToFrame(nextTimes.grandTime, tsPtr, grandTime);                 // grandTime at snapShot
            LongToFrame(nextTimes.extraTime, tsPtr, extraTime);                 // Next extraTime value
            LongToFrame(localTime,           tsPtr, localTime);                 // Transmitted frame time
            LongToFrame(dPtr->thisTxTime,    tsPtr, thatTxTime);                // Opposing transmit time
            LongToFrame(dPtr->thisRxTime,    tsPtr, thatRxTime);                // Opposing received time
            EnQueue(txQueuePtr, txPtr);                                         // Enqueue the result
        }
        wakeTime = (execTime - nextTime) > 0 ? nextTime : execTime;
    } while (serviced == TRUE);
    assert((wakeTime - stationTime) > 0);
    SleepOnBase(comPtr, wakeTime);
    return(NULL);
}

Common *
EfdxTxInit(Common *oldPtr, char *string) {
```

```
    Common     *comPtr;
    EfdxTxEntity *ePtr;
    BaseTimer   *btPtr;
    EfdxTxSaved *rcPtr;
    char temp[TLIMIT+1], name[NLIMIT+1], data[NLIMIT+1], *nextPtr;

    comPtr = CommonCreate(oldPtr, sizeof(EfdxTxEntity), &EfdxTxExec, TYPE_TX_PORT, Q_RX22_LAST, Q_TX11_LAST);
    if (comPtr != NULL) {
        ePtr = (EfdxTxEntity *)comPtr;
        rcPtr = &(ePtr->rxSaved);
        ePtr->syncInterval = rcPtr->syncInterval = SMALL_10ms;
        ePtr->guessMode = INTERPOLATE;

        rcPtr->precedence.upper = rcPtr->precedence.lower = ONES;
        rcPtr->hopCount = 0XFF;

        btPtr = &(ePtr->baseTimer);
        btPtr->validated = 0;
        btPtr->headIndex = btPtr->tailIndex = 0;

        nextPtr = StrPair(strncpy(temp, string, TLIMIT), name, data, NLIMIT);
        for ( ; name[0] != '\0'; nextPtr = StrPair(nextPtr, name, data, NLIMIT)) {
            if (strcmp(name, "name") == 0)
                strncpy(comPtr->name, data, NLIMIT);
            else if (strcmp(name, "rx0") == 0)
                assert(comPtr->rxPortCount > 0), strcpy(comPtr->rxPortPtr[0].name, data);
            else if (strcmp(name, "rx1") == 0)
                assert(comPtr->rxPortCount > 1), strcpy(comPtr->rxPortPtr[1].name, data);
            else if (strcmp(name, "tx0") == 0)
                assert(comPtr->txPortCount > 0), strcpy(comPtr->txPortPtr[0].name, data);
            else if (strcmp(name, "guessMode") == 0)
                ePtr->guessMode = atoi(data);
        }
        assert(PortsCheck(comPtr, nextPtr) == TRUE);
    }
    return(comPtr);
}

// The NextTimed() routine computes grandTime based on current txTime and
// previously sampled rxTimes information. The computation effect is:
//   1) Step back in time by a duration backInterval, to tbTime
//   2) Interpolate between rxTimes[n-N] and rxTimes[n+0], yielding tiTime
//   3) Extrapolate the tiTime forward, assuming slope==1, yielding grandTime
//   4) Extrapolate the tiTime forward, assuming rateRatio, yielding totalTime
//      The rateRatio is the ratio of grandTime to stationTime changes.
//   5) Forward {extraTime = totalTime - grandTime) along with grandTime.
// The incoming extraTime is also filtered, but not extrapolated forward:
//   1) Step back in time by a duration backInterval, to tbTime
//   2) Interpolate between rxTimes[n-N] and rxTimes[n+0], yielding extraTime
// The value of of backInterval is based on worst-case latencies:
//   backInterval = (3 * thatInterval + thisInterval) / 2
//   thatInterval - is the syncInterval for the selected clock-slave
//   thisInterval - is the syncInterval for this clock-master port

NextTimes
```

```
NextTimed(BaseTimer *btPtr, SmallTime txTime, SmallTime backInterval) {
    BaseTimes *timePtr = btPtr->times;
    BaseTimes  thisTimes, pastTimes;
    NextTimes  nextTimes;
    LargeTime  largeTime0, largeTime1;
    SmallTime  deltaTime, extraDelta, extraTime0, extraTime1, extraTime2, extraTime3,
               grandDelta, rateRatio0, rateRatio1, smallDelta, weight;
    uint16_t   headIndex, tailIndex, extraCount, i;

    assert(btPtr != NULL);
    headIndex = btPtr->headIndex;
    tailIndex = btPtr->tailIndex;
    thisTimes = timePtr[headIndex];
    pastTimes = timePtr[tailIndex];

    grandDelta = LargeToSmall(WideSubtract(thisTimes.largeTime, pastTimes.largeTime));
    extraDelta = thisTimes.extraTime - pastTimes.extraTime;
    smallDelta = thisTimes.smallTime - pastTimes.smallTime;
    if (smallDelta == 0) {
        grandDelta = smallDelta = 2 * SMALL_10ms;
        extraDelta = 0;
    }

    weight = DivideSmall((txTime - backInterval) - thisTimes.smallTime, smallDelta);
    rateRatio0 = DivideSmall(grandDelta, smallDelta);
    rateRatio1 = CLIP_RATE(rateRatio0, PPM250);
    deltaTime = MultiplySmall((rateRatio1 - SMALL_ONE), backInterval);
    if (rateRatio1 != rateRatio0)
        grandDelta = MultiplySmall(rateRatio1, smallDelta);

    largeTime0 = WideAddition(thisTimes.largeTime, SmallToLarge(MultiplySmall(grandDelta, weight)));
    largeTime1 = WideAddition(largeTime0, SmallToLarge(backInterval));

    // Average the accumulated extraTime values...
    extraTime0 = extraCount = 0;
    for (i = tailIndex; ; i = PLUS(i, 1, ARRAY_SIZE(btPtr->times))) {
        extraTime0 += timePtr[i].extraTime;
        extraCount += timePtr[i].extraCount;
        if (i == headIndex)
            break;
    }
    assert(headIndex == tailIndex || extraCount != 0);
    extraTime1 = (extraCount != 0) ? (extraTime0 / extraCount) : 0;
    extraTime2 = extraTime1 + deltaTime;
    extraTime3 = CLIP_SIZE(extraTime2, (SMALL_ONE / 256) - 1);

    nextTimes.grandTime = LargeToGrand(largeTime1);
    nextTimes.extraTime = SmallToTiny(extraTime3);
    nextTimes.totalTime = LargeToGrand(WideAddition(largeTime1, SmallToLarge(extraTime3)));
    return(nextTimes);
}

void
NextSaved(BaseTimer *btPtr, SmallTime interval0, SmallTime interval1,
 LargeTime largeTime, SmallTime extraTime, SmallTime stationTime) {
```

```
    BaseTimes *timePtr = btPtr->times;
    uint16_t headIndex, tailIndex, lastIndex, limit;
    uint8_t i;

    assert(btPtr != NULL);                                                  // Validate the pointer
    if (btPtr->headIndex == btPtr->tailIndex) {                             // If not initialized
        assert(btPtr->validated == FALSE);                                  // Indicate not valid
        btPtr->headIndex = 1, btPtr->tailIndex = 0;                         // Initial index values
        timePtr[0].largeTime = largeTime;                                   // Initialize GM time
        timePtr[0].smallTime = stationTime;                                 // and the other time
        timePtr[0].extraTime = extraTime;                                   // values, including
        timePtr[0].extraCount = 1;                                          // an errorTime count
    }
    limit = ARRAY_SIZE(btPtr->times);                                       // Array index limit
    headIndex = btPtr->headIndex;                                           // Current-head index
    lastIndex = PLUS(headIndex, -1, limit);                                 // Previous-head index
    assert(headIndex < limit && lastIndex < limit);                         // Index limitatins
    if (timePtr[headIndex].smallTime == stationTime)
        return;

    if ((timePtr[headIndex].smallTime - timePtr[lastIndex].smallTime) > (interval0 / 2)) {  // If time has changed
        btPtr->headIndex = headIndex = PLUS(headIndex, 1, limit);           // Advance the headIndex
        timePtr[headIndex].extraCount = timePtr[headIndex].extraTime = 0;   // Initialize extraTime
        btPtr->validated = TRUE;                                            // and set validated
    }
    timePtr[headIndex].largeTime = largeTime;                               // Saved GM time
    timePtr[headIndex].smallTime = stationTime;                             // Saved station time
    timePtr[headIndex].extraTime += extraTime;                              // Averaged time value
    timePtr[headIndex].extraCount += 1;                                     // is accumulated

    for (i = 0; i < 2; i += 1, btPtr->tailIndex = tailIndex) {              // Check tailIndex twice
        tailIndex = PLUS(btPtr->tailIndex, 1, limit);                       // Next tailIndex value
        if (tailIndex == headIndex)                                         // The tailIndex must
            break;                                                          // not equal headIndex
        if (stationTime - timePtr[tailIndex].smallTime <= interval1)        // The tailIndex must
            break;                                                          // maintain the interval
    }
}


// ********************************************************************************
// ***************** Wireless 802.11v wireless state-machine routines *****************
// ********************************************************************************

Common *R11vRxInit(Common *, char *);
// Receives radio service-interface parameters, sends GrandSync an SDU
//   arguments:
//     comPtr - associated state-maintaining data structure
//     name -   initialization trigger and assigned entity name
Common *
R11vRxExec(Common *comPtr, char *name) {
    uint8_t r1Info[SizePlus(R11vInfo1Ind)], r2Info[SizePlus(R11vInfo2Ind)], txInfo[MTU_SIZED];
    R11vRxEntity   *ePtr =  (R11vRxEntity *)comPtr;
    R11vInfo1Ind  *r1Ptr =  (R11vInfo1Ind *)r1Info;
    R11vInfo2Ind  *r2Ptr =  (R11vInfo2Ind *)r2Info;
```

```
    GrandSyncInd  *txPtr =  (GrandSyncInd *)txInfo;
    SyncSduData   *tsPtr = &(txPtr->service_data_unit);
    RxPort    *r1QueuePtr, *r2QueuePtr;
    TxPort    *txQueuePtr;
    GrandTime  grandTime;
    SmallTime  stationTime, cableDelay, totalDelay, smallTime;
    TinyTime   extraTime;
    TicksTime  ticksTime;
    uint64_t   da, sa;
    uint32_t   sized;
    uint8_t    hopCount, portID;
    Boolean    serviced;

    if (name != NULL)                                                    // The entity name
        return(R11vRxInit(comPtr, name));                               // for initialization.
    SetRxQueue2Ptrs(comPtr, r1QueuePtr, r2QueuePtr);
    SetTxQueue1Ptrs(comPtr, txQueuePtr);

    portID = PortID(comPtr);
    do {
        stationTime = StationTime(comPtr);                               // Station's localTime
        serviced = FALSE;

        // ***************************************************************************
        // ****** Processing arriving MLME_PRESENCE_REQUEST.indication snapshots *******
        // ***************************************************************************
        sized = DeQueue(r1QueuePtr, &r1Info);
        if (sized != 0) {
            assert(sized == sizeof(R11vInfo1Ind));
            ePtr->turnRound = r1Ptr->ticksTime3 - r1Ptr->ticksTime2;
        }

        // ***************************************************************************
        // ****** Processing arriving MLME_PRESENCE_RESPONSE.indication snapshots ******
        // ***************************************************************************
        sized = DeQueue(r2QueuePtr, &r2Info);
        if (sized != 0) {
            assert(sized == sizeof(R11vInfo2Ind));
            serviced = TRUE;
            ticksTime = R11vTime(ePtr);                                 // Station local times
            cableDelay = MIN(0, r2Ptr->roundTrip - ePtr->turnRound) / 2;  // Cable delay ticks
            totalDelay = cableDelay + (ticksTime - r2Ptr->ticksTime4);    // Elapsed time
            grandTime = LevelToGrand(r2Ptr->levelTime);                   // Grand-master time
            hopCount = r2Ptr->hopCount;                                   // Grand-master time
            extraTime = r2Ptr->extraTime;                                 // Grand-master time
            smallTime = stationTime - MultiplySmall(totalDelay, RADIO_TICK_TIME);  // Local-time stamp

            // ************************************************************************
            // ************** Creation of service-data-unit parameters ****************
            // ************************************************************************
            LongToFrame(da,                txPtr, destination_address);   // Destination address
            LongToFrame(sa,                txPtr, source_address);        // Source-port identifier
            LongToFrame(hopCount,          tsPtr, hopCount);              // The ~GM distance.
            LongToFrame(portID,            tsPtr, sourcePort);            // Source-port identifier
            WideToFrame(grandTime,         tsPtr, grandTime);             // grandTime at snapShot
```

```
            LongToFrame(extraTime,          tsPtr, extraTime);                                  // Next extraTime value
            LongToFrame(smallTime,          tsPtr, smallTime);                                  // Transmitted frame time
            LongToFrame(ePtr->syncInterval, tsPtr, syncInterval);                               // Sync transmit interval
            EnQueue(txQueuePtr, txPtr);                                                         // Enqueue the result
        }
    } while (serviced == TRUE);
    return(NULL);
}

Common *
R11vRxInit(Common *oldPtr, char *string) {
    Common     *comPtr;
    R11vRxEntity *ePtr;
    char temp[TLIMIT+1], name[NLIMIT+1], data[NLIMIT+1], *nextPtr;

    comPtr = CommonCreate(oldPtr, sizeof(R11vRxEntity), &R11vRxExec, TYPE_RX_PORT, Q_RX22_LAST, Q_TX11_LAST);
    if (comPtr != NULL) {
        ePtr = (R11vRxEntity *)comPtr;                                                  // Setup entity pointer
        ePtr->syncInterval = SMALL_10ms;                                                 // Set default interval

        nextPtr = StrPair(strncpy(temp, string, TLIMIT), name, data, NLIMIT);
        for ( ; name[0] != '\0'; nextPtr = StrPair(nextPtr, name, data, NLIMIT)) {
            if (strcmp(name, "name") == 0)
                strncpy(comPtr->name, data, NLIMIT);
            else if (strcmp(name, "rx0") == 0)
                assert(comPtr->rxPortCount > 0), strcpy(comPtr->rxPortPtr[0].name, data);
            else if (strcmp(name, "rx1") == 0)
                assert(comPtr->rxPortCount > 1), strcpy(comPtr->rxPortPtr[1].name, data);
            else if (strcmp(name, "tx0") == 0)
                assert(comPtr->txPortCount > 0), strcpy(comPtr->txPortPtr[0].name, data);
        }
        assert(PortsCheck(comPtr, nextPtr) == TRUE);
    }
    return(comPtr);
}


Common *R11vTxInit(Common *, char *);

// Receives radio GrandSync SDUs, retransmits as service-interface parameters
//   arguments:
//     comPtr - associated state-maintaining data structure
//     name -   initialization trigger and assigned entity name
Common *
R11vTxExec(Common *comPtr, char *name) {
    uint8_t rxInfo[MTU_SIZED], c2Info[SizePlus(R11vInfo2Con)], c1Info[SizePlus(R11vInfo1Con)];
    R11vTxEntity  *ePtr =  (R11vTxEntity *)comPtr;
    BaseTimer    *btPtr = &(ePtr->baseTimer);
    GrandSyncReq *rxPtr =  (GrandSyncReq *)rxInfo;
    R11vTxSaved  *sxPtr = &(ePtr->rxSaved);
    SyncSduData  *rsPtr = &(rxPtr->service_data_unit);
    R11vInfo2Con *c2Ptr =  (R11vInfo2Con *)c2Info;
    R11vInfo1Con *c1Ptr =  (R11vInfo1Con *)c1Info;
    R11vRxEntity *dPtr;
    R11vInfo1Req  r1Info, *r1Ptr = &r1Info;
```

```
    R11vInfo2Req *r2Ptr, r2Info;
    TxPort        *r1QueuePtr, *r2QueuePtr;
    RxPort        *c1QueuePtr, *c2QueuePtr, *rxQueuePtr;
    NextTimes      nextTimes;
    GrandTime      grandTime;
    SmallTime      backInterval, lapseTime, nextTime, rateInterval, stationTime, smallTime;
    TicksTime      ticksTime;
    TinyTime       extraTime;
    uint32_t       sized;
    Boolean        serviced;

    if (name != NULL)                                                                   // The entity name
        return(R11vTxInit(comPtr, name));                                              // for initialization
    SetRxQueue3Ptrs(comPtr, c1QueuePtr, c2QueuePtr, rxQueuePtr);
    SetTxQueue2Ptrs(comPtr, r1QueuePtr, r2QueuePtr);

    assert((dPtr = (R11vRxEntity *)(comPtr->pairLink)) != NULL);                       // Receiver pair
    do {
        stationTime = StationTime(comPtr);                                              // Station's localTime
        serviced = TRUE;
        backInterval = (3 * sxPtr->syncInterval + ePtr->syncInterval) / 2;
        rateInterval = backInterval + (3 * ePtr->syncInterval) / 2;

        // ***************************************************************************
        // ***** Preparing transmitted MLME_PRESENCE_RESPONSE.request information ******
        // ***************************************************************************
        nextTime = ePtr->lastTime + SMALL_10ms;
        if ((stationTime - nextTime) >= 0) {                                            // Next sync transmission
            serviced = TRUE;
            ePtr->lastTime = nextTime;                                                  // Restart 10ms timer
            EnQueue(r2QueuePtr, r1Ptr);                                                 // Enqueue the trigger
        }

        // ***************************************************************************
        // *************** Processing arrived MS_DATAUNIT.request frames ***************
        // ***************************************************************************
        sized = DeQueue(rxQueuePtr, rxPtr);
        if (sized != 0) {
            serviced = TRUE;
            assert(sized == sizeof(GrandSyncReq));
            sxPtr->hopCount =     FieldToUnsign(rsPtr, hopCount).lower;
            sxPtr->precedence =   FieldToUnsign(rsPtr, precedence);
            sxPtr->syncInterval = FieldToUnsign(rsPtr, syncInterval).lower;
            grandTime =           FieldToSigned(rsPtr, grandTime);
            extraTime =           FieldToSigned(rsPtr, extraTime).lower;
            smallTime =           FieldToSigned(rsPtr, smallTime).lower;
            NextSaved(btPtr, ePtr->syncInterval, rateInterval, GrandToLarge(grandTime), TinyToSmall(extraTime), smallTime);
        }

        // ***************************************************************************
        // ****** Processing arriving MLME_PRESENCE_REQUEST.confirm information ********
        // ***************************************************************************
        sized = DeQueue(c1QueuePtr, &c1Info);
        if (sized != 0) {
            serviced = TRUE;
```

```
            assert(sized == sizeof(R11vInfo1Con));
            ePtr->snapShot1 = c1Ptr->ticksTime1;
            ePtr->snapShot4 = c1Ptr->ticksTime4;
            ePtr->respondNow = TRUE;
        }

        // ***************************************************************************
        // ***** Preparing transmitted MLME_PRESENCE_RESPONSE.request inforomation *****
        // ***************************************************************************
        if (ePtr->respondNow == TRUE) {                                    // Next sync transmission
            serviced = TRUE;
            ePtr->respondNow = TRUE;                                       // Clear the trigger
            ticksTime = R11vTime(dPtr);
            lapseTime = ticksTime - ePtr->snapShot4;                       // Elapsed time
            smallTime = stationTime - MultiplySmall(lapseTime, RADIO_TICK_TIME);  // Extrapolate stationTime
            nextTimes = NextTimed(btPtr, smallTime, backInterval);

            r2Ptr = &r2Info;
            r2Ptr->ticksTime4 = ePtr->snapShot4;                           // Snapshot time transfer
            r2Ptr->roundTrip = ePtr->roundTrip;                            // Snapshot diff transfer
            r2Ptr->levelTime = GrandToLevel(nextTimes.grandTime);          // Grand-master radio time
            r2Ptr->extraTime = nextTimes.extraTime;                        // Grand-master extra time
            r2Ptr->precedence = sxPtr->precedence;                         // GM precedence
            r2Ptr->hopCount = sxPtr->hopCount;                             // GM hop-count
            EnQueue(r2QueuePtr, r2Ptr);                                    // Enqueue the result
        }

        // ***************************************************************************
        // ****** Processing arriving MLME_PRESENCE_RESPONSE.confirm information *******
        // ***************************************************************************
        sized = DeQueue(c2QueuePtr, &c2Info);
        if (sized != 0) {
            serviced = TRUE;
            assert(sized == sizeof(R11vInfo2Con));
            assert(c2Ptr != NULL);
        }
    } while (serviced == TRUE);
    assert((nextTime - stationTime) > 0);
    SleepOnBase(comPtr, nextTime);
    return(NULL);
}

Common *
R11vTxInit(Common *oldPtr, char *string) {
    Common     *comPtr;
    R11vTxEntity *ePtr;
    BaseTimer   *btPtr;
    char temp[TLIMIT+1], name[NLIMIT+1], data[NLIMIT+1], *nextPtr;

    comPtr = CommonCreate(oldPtr, sizeof(R11vTxEntity), &R11vTxExec, TYPE_TX_PORT, Q_RX33_LAST, Q_TX22_LAST);
    if (comPtr != NULL) {
        ePtr = (R11vTxEntity *)comPtr;
        btPtr = &(ePtr->baseTimer);
        btPtr->validated = 0;
        btPtr->headIndex = btPtr->tailIndex = 0;
```

```
        nextPtr = StrPair(strncpy(temp, string, TLIMIT), name, data, NLIMIT);
        for ( ; name[0] != '\0'; nextPtr = StrPair(nextPtr, name, data, NLIMIT)) {
            if (strcmp(name, "name") == 0)
                strncpy(comPtr->name, data, NLIMIT);
            else if (strcmp(name, "rx0") == 0)
                assert(comPtr->rxPortCount > 0), strcpy(comPtr->rxPortPtr[0].name, data);
            else if (strcmp(name, "rx1") == 0)
                assert(comPtr->rxPortCount > 1), strcpy(comPtr->rxPortPtr[1].name, data);
            else if (strcmp(name, "rx2") == 0)
                assert(comPtr->rxPortCount > 2), strcpy(comPtr->rxPortPtr[2].name, data);
            else if (strcmp(name, "tx0") == 0)
                assert(comPtr->txPortCount > 0), strcpy(comPtr->txPortPtr[0].name, data);
            else if (strcmp(name, "tx1") == 0)
                assert(comPtr->txPortCount > 1), strcpy(comPtr->txPortPtr[1].name, data);
        }
        assert(PortsCheck(comPtr, nextPtr) == TRUE);
    }
    return(comPtr);
}

Common *EponRxInit(Common *, char *);

// ********************************************************************************
// ********************** Ethernet-PON state-machine routines *********************
// ********************************************************************************

// Receives Ethernet-PON SDUs, reformats and sends to GrandSync
//   arguments:
//     comPtr - associated state-maintaining data structure
//     name -   initialization trigger and assigned entity name
Common *
EponRxExec(Common *comPtr, char *name) {
    uint8_t rxInfo[MTU_SIZED], txInfo[MTU_SIZED];
    EponRxEntity *ePtr  = (EponRxEntity  *)comPtr;
    EponMacInd   *rxPtr = (EponMacInd *)rxInfo;
    GrandSyncInd *txPtr = (GrandSyncInd *)txInfo;
    SyncSduEpon  *rsPtr = &(rxPtr->service_data_unit);
    SyncSduData  *tsPtr = &(txPtr->service_data_unit);
    RxPort     *rxQueuePtr;
    TxPort     *txQueuePtr;
    GrandTime   grandTime;
    SmallTime   smallTime, stationTime;
    TinyTime    extraTime;
    TicksTime   lapseTime, ticksTime, ponTime;
    uint64_t    da, sa;
    uint32_t    sized;
    uint8_t     hopCount, portID;
    Boolean     serviced;

    if (name != NULL)                                                    // The entity name
        return(EponRxInit(comPtr, name));                                // for initialization.
    SetRxQueue1Ptrs(comPtr, rxQueuePtr);
    SetTxQueue1Ptrs(comPtr, txQueuePtr);
```

```
    portID = PortID(comPtr);
    do {
        serviced = FALSE;
        stationTime = StationTime(comPtr);                                          // Station's localTime

        // ***************************************************************************
        // ******************** Processing arriving timeSync frames ********************
        // ***************************************************************************
        sized = DeQueue(rxQueuePtr, rxPtr);
        if (sized != 0) {
            serviced = TRUE;
            assert(sized == sizeof(EponMacInd));
            ponTime = EponTime(ePtr);

            // ***************************************************************************
            // ***** Extract frame parameters and perform basic consistency checks *****
            // ***************************************************************************
            da =         FieldToUnsign(rxPtr, destination_address).lower;
            sa =         FieldToUnsign(rxPtr, source_address).lower;
            hopCount =   FieldToUnsign(rsPtr, hopCount).lower;
            grandTime =  FieldToSigned(rsPtr, grandTime);
            extraTime =  FieldToSigned(rsPtr, extraTime).lower;
            ticksTime =  FieldToSigned(rsPtr, ticksTime).lower;
            lapseTime =  ponTime - ticksTime;
            smallTime =  stationTime - MultiplySmall(lapseTime, PON_TICK_TIME);

            // ***************************************************************************
            // ************** Update revised service-data-unit parameters **************
            // ***************************************************************************
            LongToFrame(da,                 txPtr, destination_address);            // Destination address
            LongToFrame(sa,                 txPtr, source_address);                 // Source-port identifier
            LongToFrame(hopCount,           tsPtr, hopCount);                       // The ~GM distance.
            LongToFrame(portID,             tsPtr, sourcePort);                     // Source-port identifier
            WideToFrame(grandTime,          tsPtr, grandTime);                      // grandTime at snapShot
            LongToFrame(extraTime,          tsPtr, extraTime);                      // Next extraTime value
            LongToFrame(smallTime,          tsPtr, smallTime);                      // Transmitted frame time
            LongToFrame(ePtr->syncInterval, tsPtr, syncInterval);                   // Sync transmit interval
            EnQueue(txQueuePtr, txPtr);                                             // Enqueue the result
        }
    } while (serviced == TRUE);
    return(NULL);
}

Common *EponRxInit(Common *oldPtr, char *string) {
    Common      *comPtr;
    EponRxEntity  *ePtr;
    char temp[TLIMIT+1], name[NLIMIT+1], data[NLIMIT+1], *nextPtr;

    comPtr = CommonCreate(oldPtr, sizeof(EponRxEntity), &EponRxExec, TYPE_RX_PORT, Q_RX11_LAST, Q_TX11_LAST);
    if (comPtr != NULL) {
        ePtr = (EponRxEntity *)comPtr;
        ePtr->syncInterval = SMALL_10ms;
        nextPtr = StrPair(strncpy(temp, string, TLIMIT), name, data, NLIMIT);
        for ( ; name[0] != '\0'; nextPtr = StrPair(nextPtr, name, data, NLIMIT)) {
            if (strcmp(name, "name") == 0)
```

```
                strncpy(comPtr->name, data, NLIMIT);
            else if (strcmp(name, "rx0") == 0)
                assert(comPtr->rxPortCount > 0), strcpy(comPtr->rxPortPtr[0].name, data);
            else if (strcmp(name, "tx0") == 0)
                assert(comPtr->txPortCount > 0), strcpy(comPtr->txPortPtr[0].name, data);
        }
        assert(PortsCheck(comPtr, nextPtr) == TRUE);
    }
    return(comPtr);
}


Common *EponTxInit(Common *, char *);

// Receives GrandSync SDU, reformats/resends as Ethernet-PON SDU
//   arguments:
//     comPtr - associated state-maintaining data structure
//     name -   initialization trigger and assigned entity name
Common *
EponTxExec(Common *comPtr, char *name) {
    uint8_t rxInfo[MTU_SIZED], txInfo[MTU_SIZED];
    EponTxEntity *ePtr  =  (EponTxEntity *)comPtr;
    BaseTimer    *btPtr = &(ePtr->baseTimer);
    GrandSyncReq *rxPtr =  (GrandSyncReq *)rxInfo;
    EponTxSaved  *sxPtr = &(ePtr->rxSaved);
    EponMacReq   *txPtr =  (EponMacReq *)&txInfo;
    SyncSduData  *rsPtr = &(rxPtr->service_data_unit);
    SyncSduEpon  *tsPtr = &(txPtr->service_data_unit);
    RxPort     *rxQueuePtr;
    TxPort     *txQueuePtr;
    NextTimes   nextTimes;
    GrandTime   grandTime;
    SmallTime   backInterval, smallTime, rateInterval;
    TicksTime   ticksTime;
    TinyTime    extraTime;
    LocalTime   nextTime, stationTime;
    uint32_t    sized;
    Boolean     serviced;

    if (name != NULL)                                                   // The entity name
        return(EponTxInit(comPtr, name));                               // for initialization
    SetRxQueue1Ptrs(comPtr, rxQueuePtr);
    SetTxQueue1Ptrs(comPtr, txQueuePtr);

    do {
        serviced = FALSE;
        stationTime = StationTime(comPtr);                              // Station's localTime
        backInterval = (3 * sxPtr->syncInterval + ePtr->syncInterval) / 2;
        rateInterval = backInterval + (3 * ePtr->syncInterval) / 2;

        // ********************************************************************
        // ************** Processing arrived MS_DATAUNIT.request frames **************
        // ********************************************************************
        sized = DeQueue(rxQueuePtr, rxPtr);
        if (sized != 0) {
```

```
            serviced = TRUE;
            assert(sized == sizeof(GrandSyncReq));
            sxPtr->da =           FieldToUnsign(rxPtr, destination_address).lower;
            sxPtr->sa =           FieldToUnsign(rxPtr, source_address).lower;
            sxPtr->hopCount =     FieldToUnsign(rsPtr, hopCount).lower;
            sxPtr->precedence =   FieldToUnsign(rsPtr, precedence);
            sxPtr->syncInterval = FieldToUnsign(rsPtr, syncInterval).lower;
            grandTime =           FieldToSigned(rsPtr, grandTime);
            extraTime =           FieldToSigned(rsPtr, extraTime).lower;
            smallTime =           FieldToSigned(rsPtr, smallTime).lower;
            NextSaved(btPtr, ePtr->syncInterval, rateInterval, GrandToLarge(grandTime), TinyToSmall(extraTime), smallTime);
        }

        // ***************************************************************************
        // ****************** Preparing transmitted timeSync frames ******************
        // ***************************************************************************
        nextTime = ePtr->lastTime + SMALL_10ms;
        if ((stationTime - nextTime) >= 0) {                                    // Next sync transmission
            serviced = TRUE;
            ePtr->lastTime = nextTime;                                          // Restart 10ms timer
            ticksTime = EponTime((EponRxEntity  *)comPtr);                      // Get localTime values
            nextTimes = NextTimed(btPtr, stationTime, backInterval);
            LongToFrame(sxPtr->da,            txPtr, destination_address);      // The destination_address
            LongToFrame(sxPtr->sa,            txPtr, source_address);           // The source_address
            WideToFrame(sxPtr->precedence,    tsPtr, precedence);               // GM precedence
            LongToFrame(sxPtr->hopCount,      tsPtr, hopCount);                 // GM distance
            WideToFrame(nextTimes.grandTime,  tsPtr, grandTime);                // grandTime at snapShot
            LongToFrame(nextTimes.extraTime,  tsPtr, extraTime);                // Next grandTime value
            LongToFrame(ticksTime,            tsPtr, ticksTime);                // Next extraTime value
            EnQueue(txQueuePtr, txPtr);                                         // Enqueue the result
        }
    } while (serviced == TRUE);
    assert((nextTime - stationTime) > 0);
    SleepOnBase(comPtr, nextTime);
    return(NULL);
}

Common *
EponTxInit(Common *oldPtr, char *string) {
    Common     *comPtr;
    EponTxEntity *ePtr;
    BaseTimer   *btPtr;
    char temp[TLIMIT+1], name[NLIMIT+1], data[NLIMIT+1], *nextPtr;

    comPtr = CommonCreate(oldPtr, sizeof(EponTxEntity), &EponTxExec, TYPE_TX_PORT, Q_RX11_LAST, Q_TX11_LAST);
    if (comPtr != NULL) {
        ePtr = (EponTxEntity *)comPtr;
        btPtr = &(ePtr->baseTimer);
        btPtr->validated = 0;
        btPtr->headIndex = btPtr->tailIndex = 0;

        nextPtr = StrPair(strncpy(temp, string, TLIMIT), name, data, NLIMIT);
        for ( ; name[0] != '\0'; nextPtr = StrPair(nextPtr, name, data, NLIMIT)) {
            if (strcmp(name, "name") == 0)
                strncpy(comPtr->name, data, NLIMIT);
```

```
            else if (strcmp(name, "rx0") == 0)
                assert(comPtr->rxPortCount > 0), strcpy(comPtr->rxPortPtr[0].name, data);
            else if (strcmp(name, "tx0") == 0)
                assert(comPtr->txPortCount > 0), strcpy(comPtr->txPortPtr[0].name, data);
        }
        assert(PortsCheck(comPtr, nextPtr) == TRUE);
    }
    return(comPtr);
}
```