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

Draft 0.239

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

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	2005Apr28	DVJ	Updates based on 2005Apr27 meeting discussions
0.085	2005May11	DVJ	– Updated list-of-contributors, page numbering, editorial fixes.
0.088	2005Jun03	DVJ	– Application latency scenarios clarified.
0.090	2005Jun06	DVJ	– Misc. editorials in bursting and bunching annex.
0.092	2005Jun10	DVJ	– Extensive cleanup of Clause 5 subscription protocols.
0.121	2005Jun24	DVJ	– Extensive cleanup of clock-synchronization protocols.
0.127	2005Jul04	DVJ	– Pacing descriptions greatly enhanced.
0.200	2007Jan23	DVJ	Removal of non time-sync related information, initial layering proposal.
0.207	2007Feb01	DVJ	Updates based on feedback from Monterey 802.1 meeting. – Common entity terminology; Ethernet type code expandability.
0.216	2007Feb17	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	2007Mar03	DVJ	Updates for whiplash free PLL cascading.
0.230	2007Mar05	DVJ	Major changes: – simplified back-interpolation – first interation on an Ethernet-PON interface – client-level clock-master and clock-slave interfaces defined
—	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

List of tables

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.

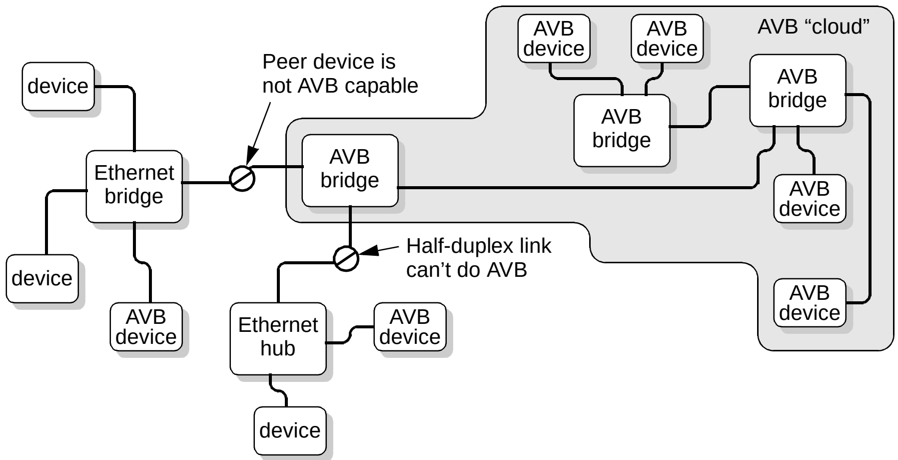


Figure 1.1—Topology and connectivity

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 7: Duplex-link state machines
- Annex A: Bibliography
- Annex C: Bridging to IEEE Std 1394
- Annex D: Review of possible alternatives
- 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.

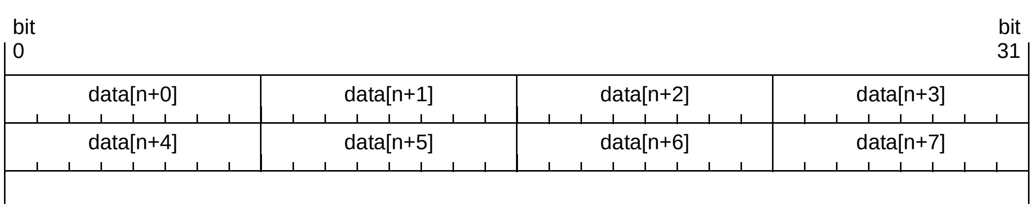


Figure 3.1—Bit numbering and ordering

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.

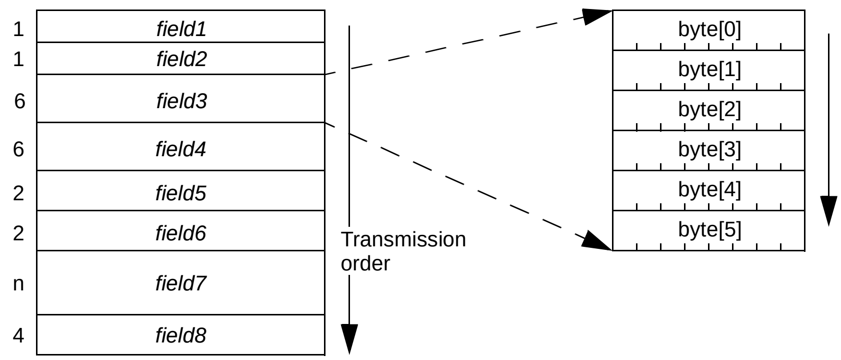


Figure 3.2—Byte sequential field format illustrations

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.

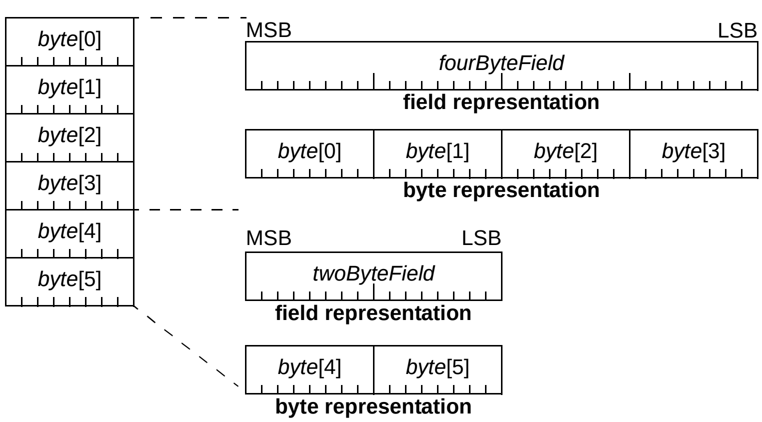


Figure 3.3—Multibyte field illustrations

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.

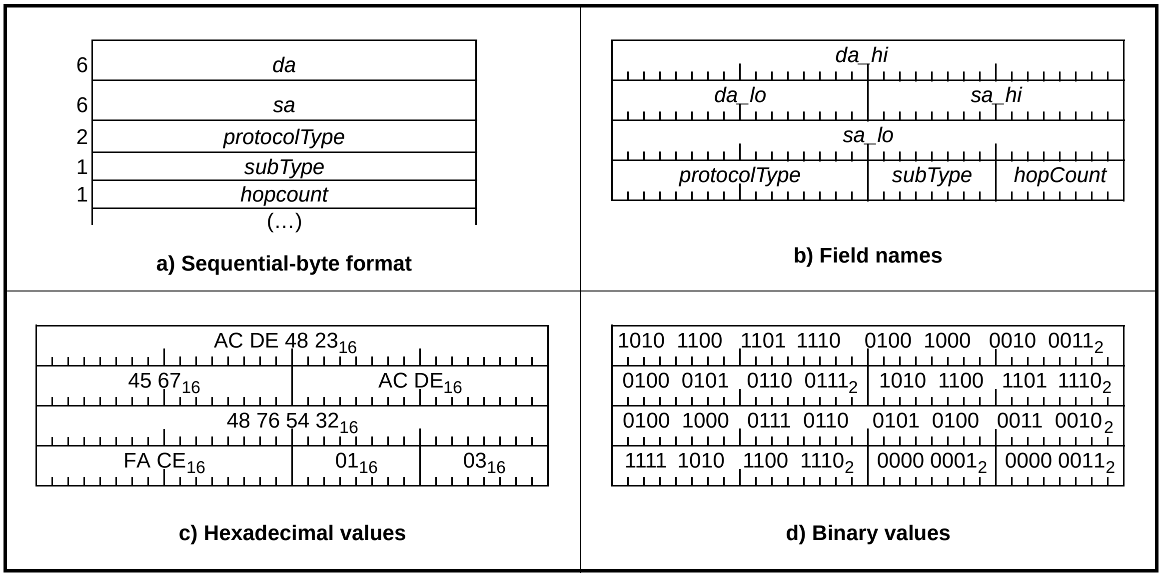


Figure 3.4—Illustration of fairness-frame structure

3.10 MAC address formats

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

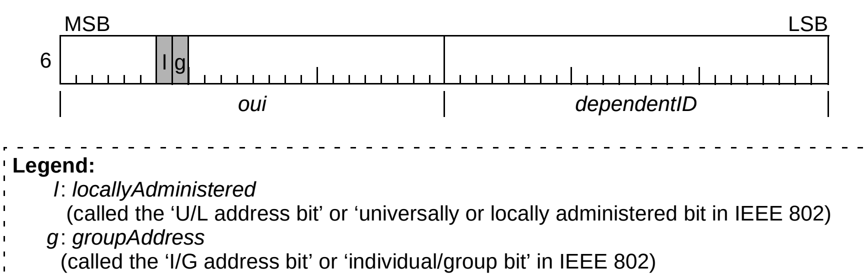


Figure 3.5—MAC address format

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.

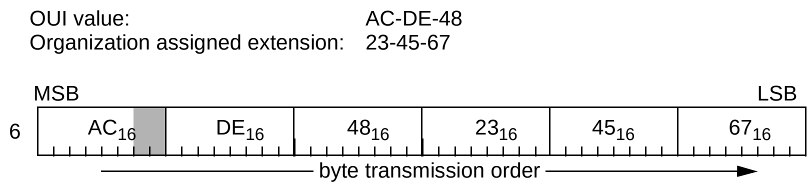


Figure 3.6—48-bit MAC address format

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
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
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.

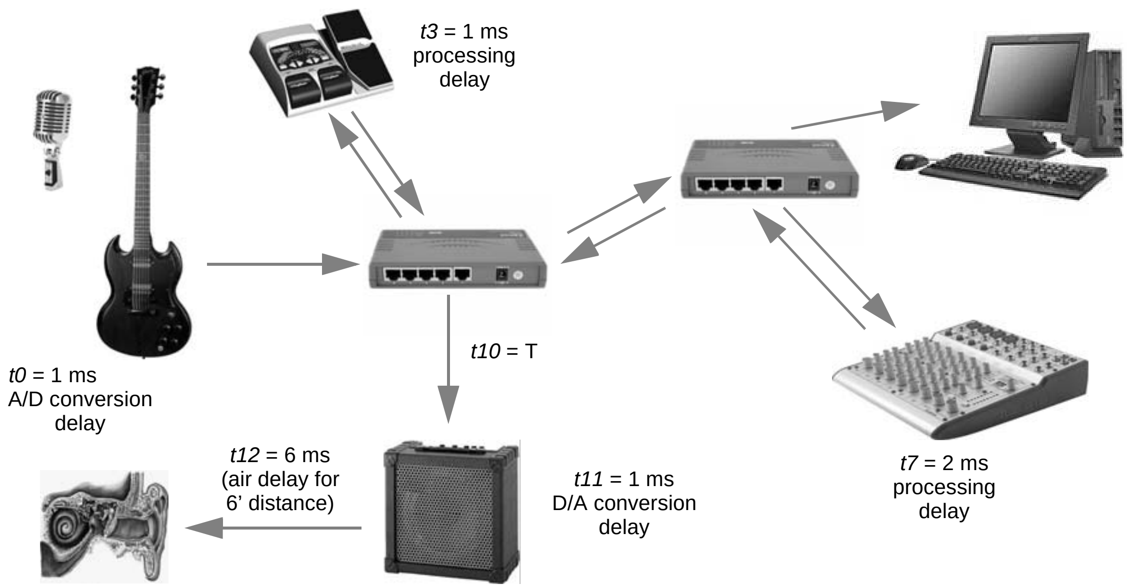


Figure 5.1—Garage jam session

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.

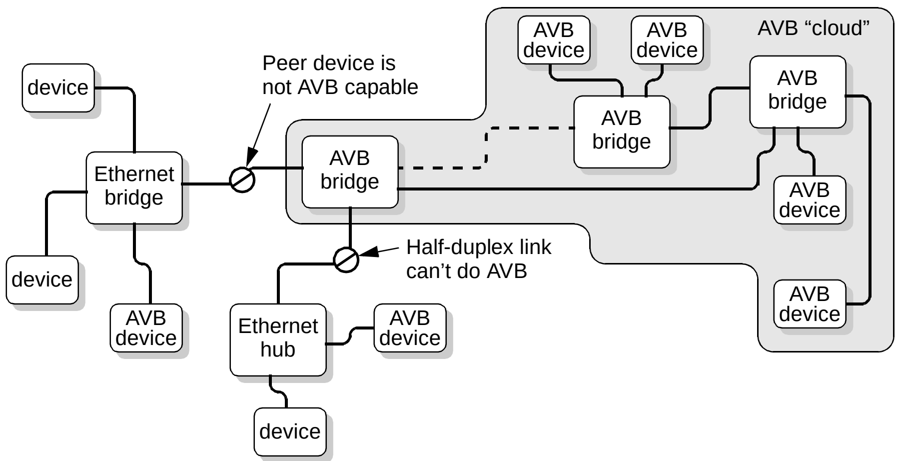


Figure 5.2—Possible looping topology

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.

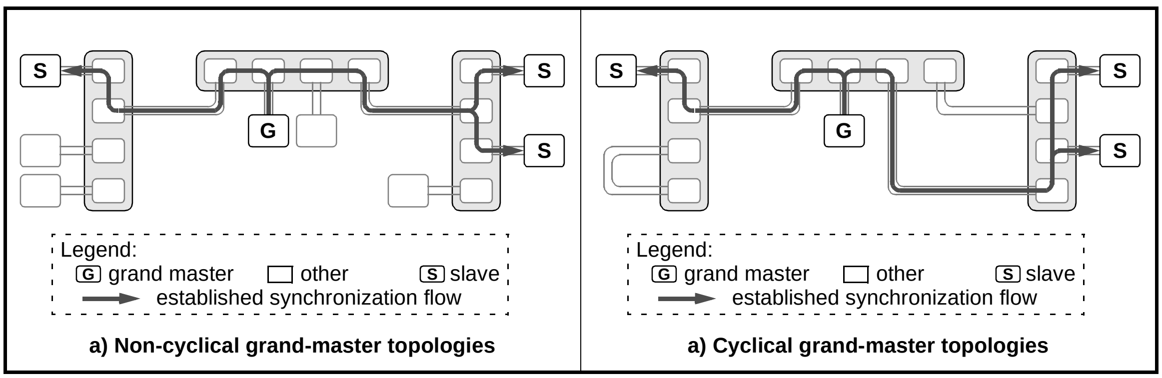


Figure 5.3—Timing information flows

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.

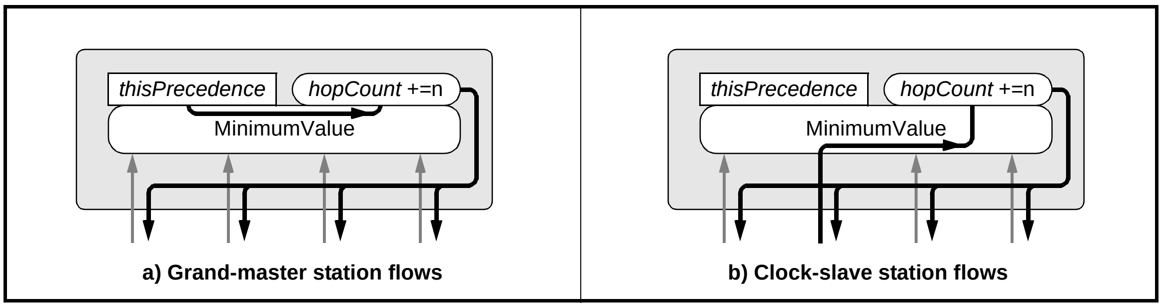


Figure 5.4—Grand-master precedence flows

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

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.

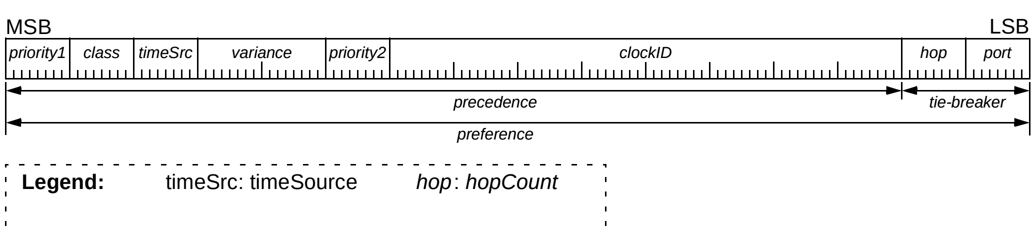


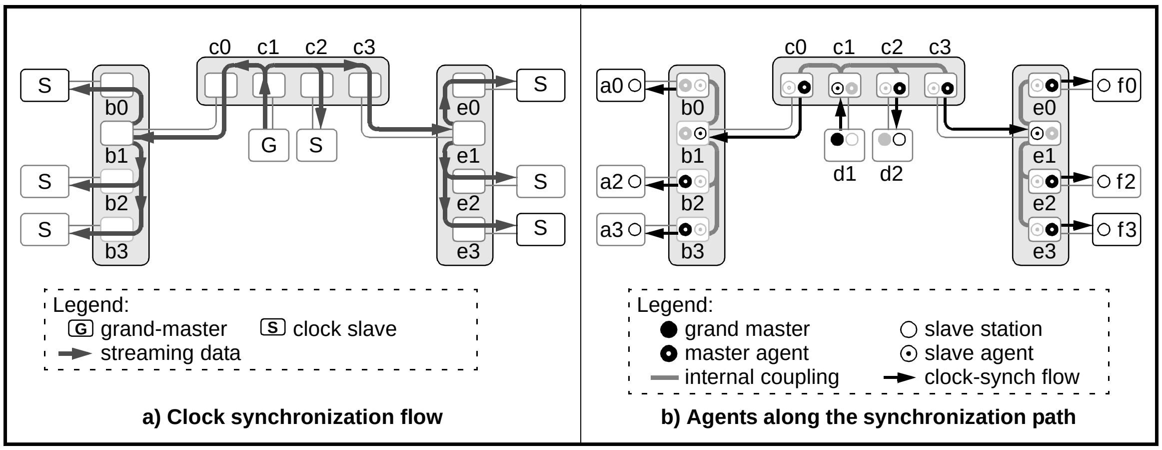
Figure 5.5—Grand-master selector

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.



a) Clock synchronization flow **b) Agents along the synchronization path**

Figure 5.6—Hierarchical flows

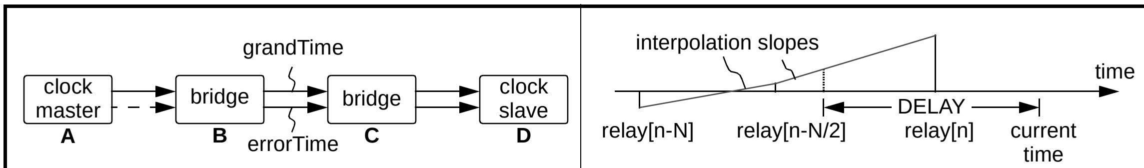
Internal communications distribute synchronized time from clock-slave agents b1, c1, and e1 to the other clock-master agents on bridgeB, bridgeC, and bridgeE respectively. Within a clock-slave, precise time synchronization involves adjustments of timer value and rate-of-change values.

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 Back-in-time interpolation (no gain-peaking)

A transient phenomenon associated with cascaded PLLs is called whiplash or gain-peaking, depending on how the phenomenon is observed. A whiplash effect is visible as ringing after a injected spike and/or a step change in frequency. The gain-peaking effect is visible as a frequency gain, that becomes increasingly larger through cascaded PLLs, for selected frequencies. For basic cascaded PLLs (see Figure 5.7a), this phenomenon is unavoidable, although its effects can be reduced through careful design or manual tuning of peaking frequencies.

To avoid this phenomenon when passing through multiple bridges, two signal values are transmitted over intermediate hops: *grandTime* and *errorTime* (see Figure 5.7a). For stability, the *grandTime* value corresponds to an interpolated DELAY time in the past (DELAY is typically assumed to be four transmission intervals). For accuracy, the *errorTime* value represents errors due to differences in DELAY, as measured by local-clock and syntonized-clock timers.



a) Cascaded PLL-error propagation

b) Interpolation of *grandTime* into the “past”

Figure 5.7—Cascaded PLL designs

Within the context of Figure 5.7a, the clock-master stationA could send time-varying *grandTime* values and a zero-valued *errorTime* value. The stationB bridge outputs a revised rate-interpolated whiplash-free *grandTime* value, along with nonzero *errorTime* values.

The stationC bridge behaves similarly; producing a whiplash-free *grandTime* output along with revised *errorTime* values. The propagation of (relatively DC-free) *errorTime* values is deferred for a DELAY-time interval, so that new values can be conveniently interpolated between past-observed values.

The concept of whiplash-free interpolation assumes the presence of relatively stable clock rates. The next *grandTime* output value *out*[*m*] is computed by interpolating between the last *grandTime* output value *out*[*m*-1] and the most-recent *relay*[*n*]-supplied *grandTime* values, as illustrated in Figure 5.7b. To compensate for the back-in-time error, the value of *out*[*m*]+DELAY is transmitted as the current *grandTime* value.

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 *errorTime* communication channel, over which the difference between nominal and rate-normalized DELAY values can be transmitted.

5.5 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 assured.
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 abstractions

6.1 Overview

This clause specifies the state machines that support media-independent 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.

6.2 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. 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 preference of the passed frames determine whether the frame is ignored by the GrandSync protocol entity or modified and redistributed to the remaining TimeSync state machines.

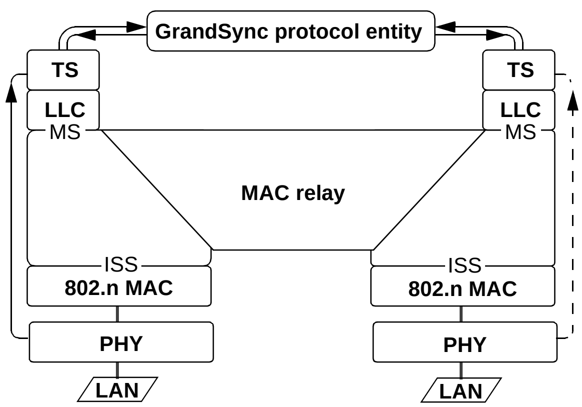


Figure 6.1—GrandSync interface model

All components are assumed to have access to a common free-running (not adjustable) local timer. There is not necessarily a one-to-one correspondence between the primitives and formal procedures and the interfaces in any particular implementation.

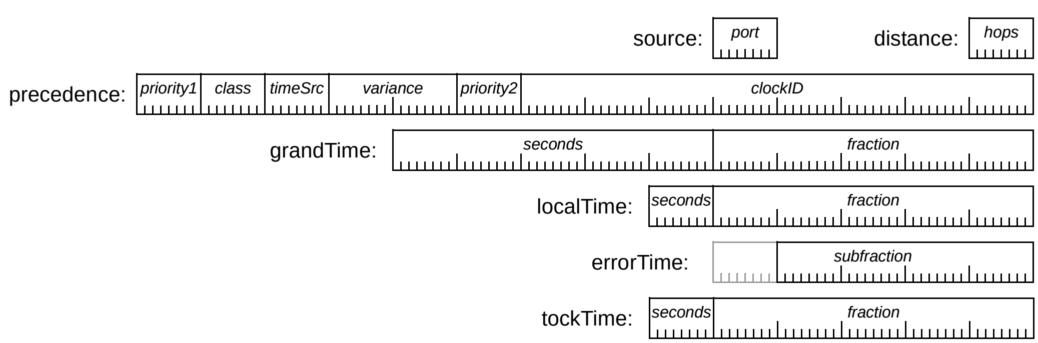


Figure 6.2—GrandSync interface components

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

6.3 GrandSync interface parameters

6.3.1 MA_SYNC.indication

6.3.1.1 Function

6.3.1.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
MA_SYNC.indication {
        da,              // Destination address
        sa,              // Optional
        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)
        errorTime,       // Accumulated grandTime error
        port,            // Identifies the source port
        hopCount,        // Distance from the grand-master station
        localTime,       // Local-time snapshot (1-cycle delayed)
        tockTime         // 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:

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

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

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

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

6.3.1.2.5 ***version*****:** An 8-bit field that identifies the format and function of the following fields (see xx).

6.3.1.2.6 ***precedence*****:** A 14-byte field that has specifies precedence in the grand-master selection protocols (see 6.3.1.4).

6.3.1.2.7 ***grandTime*****:** An 80-bit field that specifies the grand-master synchronized time within the source station, when the previous timeSync frame was transmitted (see 6.3.1.6).

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

6.3.1.2.9 ***port*****:** An 8-bit field that identifies the port that sourced the timedSync frame.

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

6.3.1.2.11 ***localTime*****:** A 48-bit field that specifies the local free-running time within this station, when the previous timeSync frame was received (see 6.3.1.8).

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

6.3.1.2.13 ***localTime*****:** A 48-bit field that specifies the local free-running time within the source station, when the previous timeSync frame was transmitted (see 6.3.1.8).

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

NOTE—The *tockTime* 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.

6.3.1.3 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.

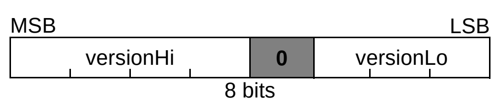


Figure 6.3—Global-time subfield format

6.3.1.4 *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.

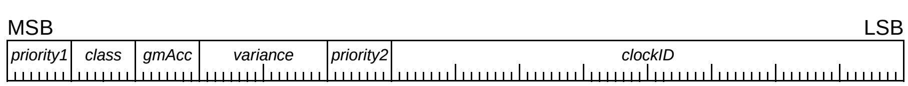


Figure 6.4—*precedence* subfields

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

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

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

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

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

6.3.1.4.6 ***clockID*****:** A 64-bit globally-unique field that ensures a unique precedence value for each potential grand master, when {*priority1*, *class*, *variance*, *priority2*} fields happen to have the same value (see 6.3.1.5).

6.3.1.5 *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.

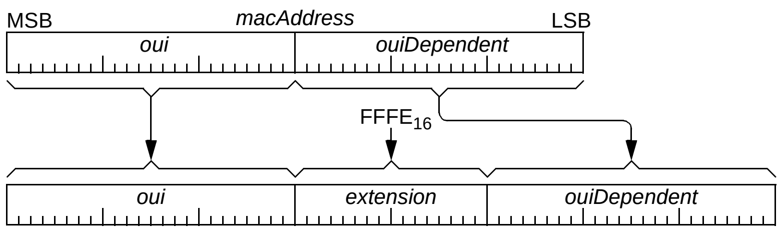


Figure 6.5—*clockID* format

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

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

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

6.3.1.6 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.

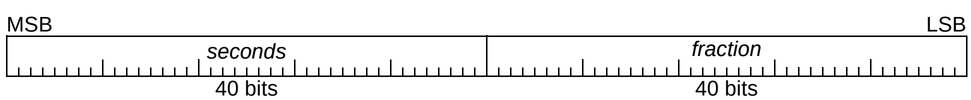


Figure 6.6—Global-time subfield format

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

6.3.1.6.2 ***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.

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

6.3.1.7 *errorTime* format

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.

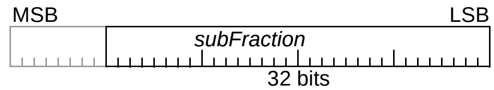


Figure 6.7—errorTime format

6.3.1.8 *localTime* formats

The *localTime* value within a frame is based on a fractions-of-second value, as illustrated in Figure 6.8. The 40-bit *fraction* field specifies the time offset within the *second*, in units of 2^{-40} second.

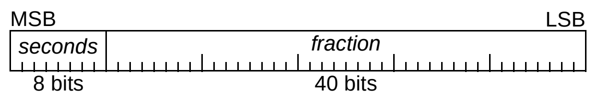


Figure 6.8—*localTime* format

6.3.2 MA_SYNC.request

6.3.2.1 Function

6.3.2.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
MA_SYNC.request {
        da,             // Destination address
        sa,             // Optional
        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)
        errorTime,      // Accumulated grandTime error
        port,           // Identifies the source port
        hopCount,       // Distance from the grand-master station
        localTime,      // Local-time snapshot (1-cycle delayed)
        tockTime        // Nominal timeSync transmission interval
}
```

The parameters of the MA_DATA.request are described in 6.3.1.2.

6.4 GrandSync state machine

6.4.1 Function

The GransSync protocol entity assumes the presence of one or more TimeSync entities communicating with a GrandSync protocol entity, as illustrated on the top-side of Figure 6.9. A listener-only clock-slave capable entity is not required to be grand-master capable

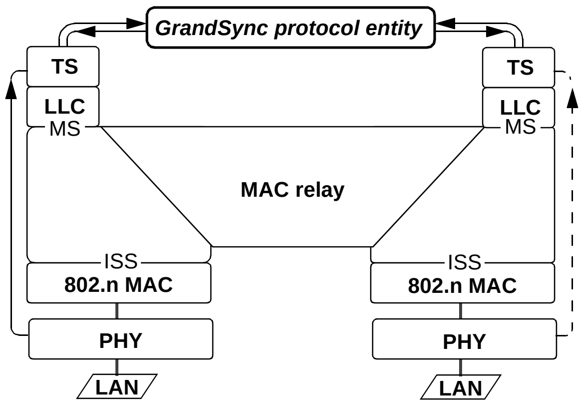


Figure 6.9—GransSync interface model

The GrandSync state machine (illustrated with an italics name and darker boundary) is responsible for saving time parameters from observed MA_SYNC.indication parameters and generating MA_SYNC.request parameters for delivery to other ports. 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.

6.4.2 State machine definitions

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_MA_IND—The queue identifier associated with indications from TS entities.
  - Q_MA_REQ—The queue identifier associated with requests to TS entities.

6.4.3 State machine variables

curentTime
: A local variable representing current time.

gPtr
: A pointer to the GrandSync data structure, where the data structure includes the following:
  - *info*—The parameters associated with an MA_SYNC.indication (see 6.3.1.2).
  - *timer*—Tthe time of the last observed MA_SYNC.indication, used for timeout purposes.

preferenceNew, *preferenceOld*
: Local variables consisting of concatenated *preference*, *hopCount*, and *port* parameters.

rxInfo
: The parameters associated with an MA_SYNC.indication (see 6.3.1.2).

txInfo
: The parameters associated with an MA_SYNC.request (see 6.3.1.2).

6.4.4 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.

GetLocalTime(*gPtr*)

Returns the value of the station's shared local timer, encoded 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.

6.4.5 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 6.2.

Table 6.1—GrandSync state table

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

START	(rxInfo = Dequeue(Q_MA_IND))
	!= NULL
		1	// Summary of PreferenceBetter()
			preferenceNew = MERGE(rxInfo.precedence,
			rxInfo.hopCount, rxInfo.port);
			preferenceOld = MERGE(gPtr->info.precedence,
			gPtr->info.hopCount, gPtr->info.port);
			better = (preferenceNew <= preferenceOld;)
				SERVE

	(currentTime – gPtr->timer)
	> 4 * gPtr->info.tockTime
		2	// Summary of PreferenceTimeout()
			gPtr->info.hopCount = gPtr->info.port =
			gPtr->info.precedence = ONES;
			gPtr->timer = currentTime;
				START

	—	3	currentTime = GetLocalTime();	
SERVE	rxInfo.hopCount == HOPS	4	—	START
	rxInfo.portID == gPtr->info.portID	5	gPtr->info = txInfo = rxInfo;	NEAR
	better	6		
	—	7	—	START
NEAR	rxInfo.hopCount > gmPtr->info.hopCount	8	txInfo.hopCount = MIN(HOPS, 1 + (HOPS + rxInfo.hopCount) / 2);	LAST
	—	9	txInfo.hopCount = rxInfo.hopCount + 1;	
LAST	—	10	gPtr->timer = currentTime; Enqueue(Q_MA_REQ, txInfo);	START

Row 6.1-1: Available indication information is processed; the preference comparison is precomputed. The *preferenceNew* and *preferenceOld* values consist of *precedence*, *hopCount*, and *port* values.
Row 6.1-2: The absence of indications forces the timeout, after a port-specific delay
Row 6.1-3: Wait for changes of conditions.

Row 6.1-4: Aged indications are discarded.
Row 6.1-5: Same-port indications always have preference.
Row 6.1-6: Preferred indications are accepted.
Row 6.1-7: Other indications are discarded.

Row 6.1-8: Increasing *hopCount* values are indicative of a rogue frame and are therefore quickly quashed.
Row 6.1-9: Non-increasing *hopCount* values are incremented and are thus aged slowly.

Row 6.1-10: Accept the preferred-indication parameters and reset the timeout timer.
Retransmit accepted indication parameters to all attached ports, including the source port.

6.5 TimeSyncTxSlave state machine

6.5.1 Function

The time-synchronization service model assumes the presence of one or more clock-slave capable TimeSync entities communicating with a GransSync protocol entity, as illustrated on the top-side of Figure 6.10. A listener-only clock-slave capable entity is not required to be grand-master capable

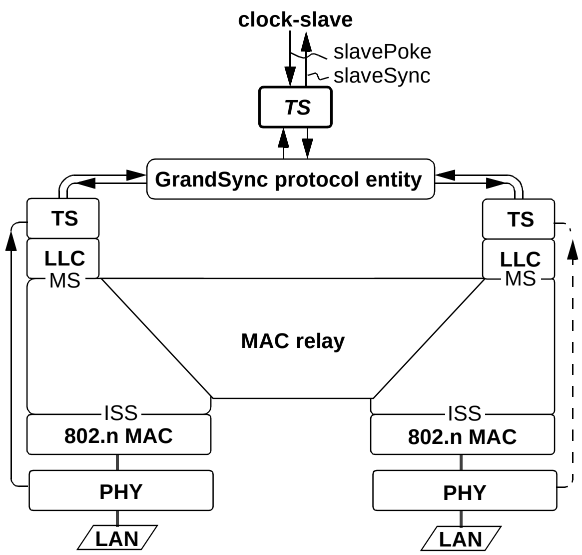


Figure 6.10—Clock-slave interface model

The TimeSyncTxSlave state machine (illustrated with an italics name and darker boundary) is responsible for saving time parameters from relayed timedSync frames and servicing time-sync requests from the attached clock-slave interface. The sequencing of this state machine is specified by Table 6.2; details of the computations are specified by the C-code of Annex F.

6.5.2 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_MA_REQ—The queue identifier associated with information from the GrandSync entity.
    Q_CI_REQ—The queue identifier associated with slavePoke requests.
    Q_CI_IND—The queue identifier associated with slaveSync indications.
T10ms
  A constant the represents a 10 ms value.

6.5.3 State machine variables

curentTime
  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.

frame
The contents of a MAC-supplied frame.

req
A contents of a higher-level supplied time-synchronization request, including the following:
infoCount—A value that increments on each masterSync frame transmission.

res
A contents of a lower-level provided time-synchronization response, including the following:
infoCount—The value of *currentTime* associated with the last timeSync packet arrival.
grandTime—The value of grand-master synchronized time, at the localTime snapshot.

port
A data structure of port-specific information sufficient to compute grand-master synchronized time.

6.5.4 State machine routines

Dequeue(*queue*)
Returns the next available frame from the specified queue.
frame—The next available frame.
NULL—No frame available.

Enqueue(*queue*)
Places the frame at the tail of the specified queue.

FrameToSlave(*pPtr*, *localTime*)
Computes the globalTime value at localTime, as specified by the C code of Annex F.

RelayToState(*pPtr*, *frame*, *currentTime*)
Copies a high-preference MAC-relay frame to port storage, as specified by the C code of Annex F. (Low preference MAC-relay frames are simply discarded.)

TimeSyncFrame(*frame*)
Checks the frame contents to identify timeSync frame.
TRUE—The frame is a timeSync frame.
FALSE—Otherwise.

6.5.5 TimeSyncTxClock state table

The TimeSyncTxClock 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 6.2.

Table 6.2—TimeSyncTxClock state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(rxInfo = Dequeue(Q_MA_REQ)) **!**= NULL	1	GrandToState(pPtr, rxInfo);	START

	((reqInfo = Dequeue(Q_CI_REQ)) **!**= NULL	2	// Summary of TimeSyncTxClockA()
			grandTimes = StateToGrand(pPtr, currentTime);
			resInfo.count = reqInfo.count;
			resInfo.grandTime =
			grandTimes.grandTime+grandTimes.errorTime;
			Enqueue(Q_CI_IND, resInfo);	

	—	3	currentTime = GetLocalTime(pPtr);	

Row 6.2-1: The received MA_SYNC.request frames set clock-interpolation parameters.
Row 6.2-2: A clock-slave request generates an affiliated information-providing indication.
Row 6.2-3: Wait for the next change-of-conditions.

6.6 TimeSyncRxClock state machine

6.6.1 Function

The time-synchronization service model assumes the presence of one or more grand-master capable entities communicating with a MAC relay, as illustrated on the left side of Figure 6.11. A grand-master capable port may also provide clock-slave functionality, so that any non-selected clock-master capable station can synchronize to the selected grand-master station..

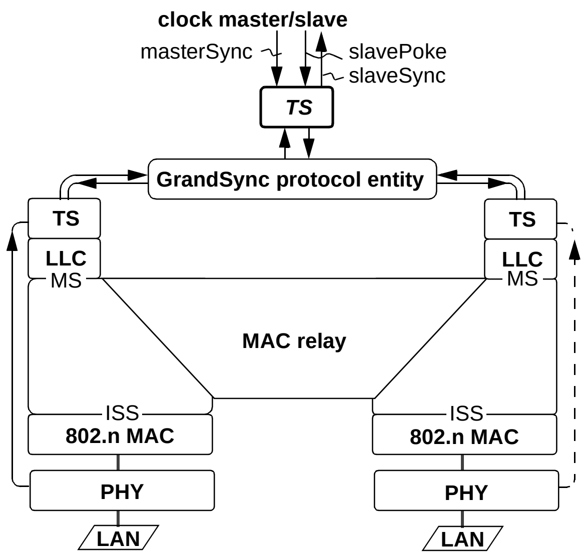


Figure 6.11—Clock-master interface model

The clock-master TimeSyncRxClock state machine (illustrated with an italics name and darker boundary) is responsible for monitoring its port's masterSync requests and sending MAC-relay frames. The sequencing of this state machine is specified by Table 6.3; details of the computations are specified by the C-code of Annex F.

6.6.2 State machine definitions

NULL
  A constant indicating the absence of a value that (by design) cannot be confused with a valid value.
CYCLE
  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_CE_SET—The queue identifier associated with received clock-master sync frames.
    Q_MA_IND—The queue identifier for information sent from the GrandSync entity.

6.6.3 State machine variables

info
  A contents of a higher-level supplied time-synchronization request, including the following:
    grandTime—The value of grand-master time, when the previous masterSync frame was sent.
    frameCount—A value that increments on each masterSync frame transmission.
next
  A transient value representing the expected value of the next *info*.*frameCount* field value.

port
A data structure containing port-specific information comprising the following:
rxSyncFrame—The next frame to be transmitted over the MAC-relay.
rxFrameCount—The value of *frameCount* within the last received frame.
rxSnapShot0—The *info.snapShot* field value from the last receive-port poke indication.
rxSnapShot1—The value of the *pPtr->rxSnapShot0* field saved from the last poke indication.

6.6.4 State machine routines

Dequeue(*queue*)
Returns the next available frame from the specified queue.
frame—The next available frame.
NULL—No frame available.
Enqueue(*queue*)
Places the frame at the tail of the specified queue.

6.6.5 TimeSyncRxClock state table

The TimeSyncRxClock state table encapsulates clock-provided sync information into a MAC-relay frame, as illustrated in Table 6.3.

Table 6.3—TimeSyncRxClock state machine table

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

START	(info = Dequeue(Q_CI_SET))
	!= NULL
		1	// Summary of TimeSyncRxClockA
			pPtr->rxSnapShot1 = pPtr->rxSnapShot0;
			pPtr->rxSnapShot0 = currentTime;
			count= (pPtr->rxFrameCount+1)%COUNT;
			pPtr->rxFrameCount = infoReq.frameCount;
			wrong = (count != infoReq.frameCount);
				SEND

	—	2	currentTime = GetLocalTime(pPtr);	START
SEND	wrong	3	—	START

	—	4	// Summary of TimeSyncRxClockB
			txPtr = &(txInfo);
			SetupInfo(pPtr, txPtr);
			txPtr->hopCount = 0;
			txPtr->precedence = pPtr->precedence;;
			txPtr->grandTime = info.grandTime;
			txPtr->errorTime = 0;
			txPtr->localTime = pPtr->rxSnapShot1;
			Enqueue(Q_MA_IND, txInfo);	

Row 6.3-1: Update snapshot values on masterSync request arrival.
Row 6.3-2: Wait for the next masterSync request arrival.

Row 6.3-3: Nonsequential requests are discarded.
Row 6.3-4: Sequential requests are forwarded over the MAC-relay.

7. Duplex-link state machines

7.1 Overview

This clause specifies the state machines that support duplex-link 802.3-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.

7.1.1 Duplex-link indications

The duplex-link TimeSyncRxDuplex state machines are provided with snapshots of timeSync-frame reception and transmission times, as illustrated by the ports within Figure 7.1. These link-dependent indications can be different for bridge ports attached to alternative media..

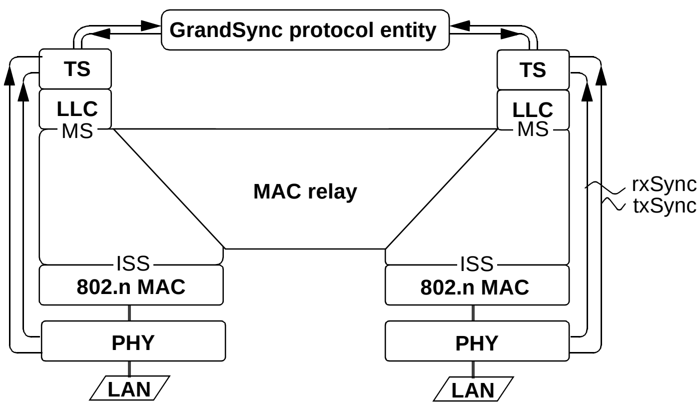


Figure 7.1—Duplex-link interface model

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 7.2.

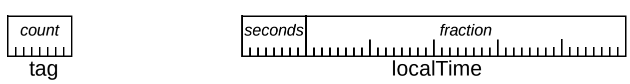


Figure 7.2—Contents of rxSync/txSync indications

7.1.2 Rate-normalization requirements

If the absence of rate adjustments, significant *grandTime* errors can accumulate between periodic updates, as illustrated in Figure 7.3. The 2 μs deviation is due to the cumulative effect of clock drift, over the 10 ms send-period interval, assuming clock-master and clock-slave crystal deviations of –100 PPM and +100 PPM respectively.

While this regular sawtooth is illustrated as a highly regular (and thus perhaps easily filtered) function, irregularities could be introduced by changes in the relative ordering of clock-master and clock-slave

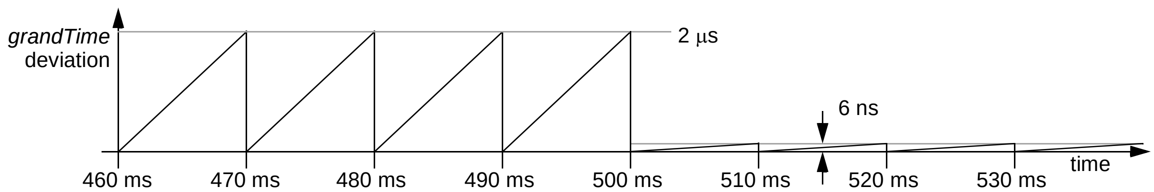


Figure 7.3—Rate-adjustment effects

transmissions, or transmission delays invoked by asynchronous frame transmissions. Tracking peaks/valleys or filtering such irregular functions are thought unlikely to yield similar *grandTime* deviation reductions.

To reduce such time deviations, a lower-rate (currently assumed to be 80 ms) activity measures the ratio of each station's frequency to that of its adjacent neighbor. When these calibration factors are applied, the effects of rate differences are easily be reduced to less than 1 PPM, based on the aforementioned time-accuracy assumptions. At this point, the timer-offset measurement errors (not clock-drift induced errors) dominate the clock-synchronization error contributions.

7.1.3 Duplex-link delays

On some forms of duplex-link media, time-synchronization involves periodic not-necessarily synchronized packet transmissions between adjacent stations, as illustrated in Figure 7.4a. The transmitted frame contains the following information:

precedence—Specifies the grand-master precedence.
grandTime—An estimation of the grand-master time.
localTime—A sampling of the neighbor's local time.
thatTxTime—The adjacent link's timeSync transmit time.
thatRxTime—The adjacent link's timeSync receive time.

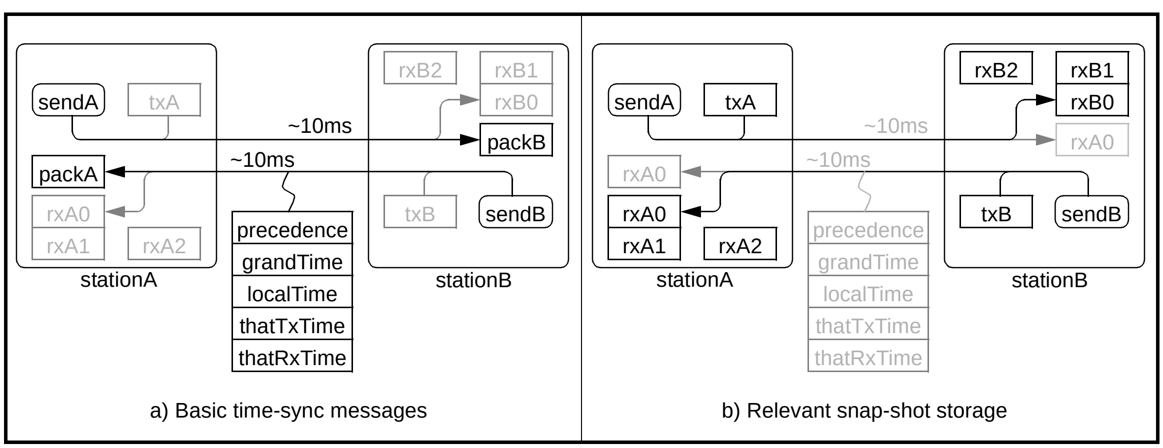


Figure 7.4—Timer snapshot locations

Snapshots are taken when packets are transmitted (illustrated as *txA* and *txB*) and received (illustrated as *rxA* and *rxB*), as illustrated in Figure 7.4b. The receive snapshot is double buffered, in that the value of *rxB0* is copied to *rxB1* when the *rxB0* snapshot is taken. Similarly, the value of *rxA0* is copied to *rxA1* when the *rxA0* snapshot is taken.

The physical entity that triggers the received-frame and transmitted-frame snapshot operations is deliberately left ambiguous. Mandatory jitter-error accuracies are sufficiently loose to allow transmit/receive snapshot circuits to be located with the MAC. Vendors may elect to further reduce timing jitter by latching the receive/transmit times within the PHY, where the uncertain FIFO latencies can be more easily avoided.

The the timeSync frame arrives from stationA, the frame's *localTime* value is copied to the rxB2 register, and is simultaneously available with the updated rxB1 snapshot value. Similarly, when the timeSync frame arrives from stationB, the frame's *localTime* value is copied to the rxA2 register, and is simultaneously available with the updated rxA1 snapshot value.

For stationB, the values inserted into each frame include the following:

localTime—The txB value, representing the last timeSync frame-transmission time on this link.
thatTxTime—The rxB2 value, representing a timeSync frame-transmission time on the other link.
thatRxTime—The rxB1 value, representing a timeSync frame-reception time on the other link.
grandTime—The computed grand-master time associated with the co-resident *localTime* value.

For stationA, the values inserted into each frame include the following:

localTime—The txA value, representing the last timeSync frame-transmission time on this link.
thatTxTime—The rxA2 value, representing a timeSync frame-transmission time on the other link.
thatRxTime—The rxA1 value, representing a timeSync frame-reception time on the other link.
grandTime—The computed grand-master time associated with the co-resident *localTime* value.

Assuming the local stationA and stationB timers have the same frequencies and the two links on the span have identical delays, the link delay can be computed at stationB and stationA, based on the contents of the most-recently received timeSync frame, as specified by Equation 7.1 and Equation 7.2 respectively.

$$linkDelayB = ((\mathrm{rxB1} - \mathrm{frame}.thatTxTime) - (\mathrm{frame}.localTime - \mathrm{frame}.thatRxTime))/2; \quad (7.1)$$
$$linkDelayA = ((\mathrm{rxA1} - \mathrm{frame}.thatTxTime) - (\mathrm{frame}.localTime - \mathrm{frame}.thatRxTime))/2; \quad (7.2)$$

If the stationA-to-stationB and stationB-to-stationA links 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.

7.1.4 Received timeSync computations

The baseline link-delay calculations of 7.1.3 are sufficient for 802.11v and other interconnects wherein the timeSync turn-around latencies are tightly controlled by the MAC. For 802.3 and other interconnects, the turnaround times can be done above the MAC and can be much larger than the packet-transmission times. For such media, the duplex-link delay calculations must be compensated by measured differences in adjacent-station clock rates, as discussed within this subclause.

Assuming the local stationA and stationB timers have the different frequencies and the two links on the span have identical delays, the link delay can be computed at stationB based on the contents of the most-recently received timeSync frame.

NOTE—The *rating* portion of the *linkDelay* computation is based on the station-local time within adjacent-neighbor exchanges and is therefore unaffected by discontinuities in the distributed grand-master time reference.

7.1.5 Transmitted timeSync computations

At the bridge's co-resident clock-master port, the current *grandTime* value is estimated by interpolating a fixed local-timer amount (40 ms) into the past, as summarized by Equation 7.3. The input error value is similarly interpolated into the past and incremented by the local-error contribution.

(7.3)

```
// Update information when transmitted frame is formed.
// This code summarizes the behavior of StateToTimes() in Annex F.
tockTime = (2 * (TOCK_TIME + MIN(TOCK_TIME, relay.tockTime))) // Sampling interval
delay = (tockTime - ((THIS_TOCK + relay.tockTxTime) / 2))     // Ensures interpolation
lapseTime = txB - delay;                                      // Back-in-time location
if (lapseTime < localTime0) {                                 // Remote interpolation:
  grandRated = grandRate1;                                    // based on grand rate;
  errorRated = errorRate1;                                    // based on rate
} else {                                                      // Recent interpolation:
  grandRated = grandRate0;                                    // based on grand rate;
  errorRated = errorRate0;                                    // based on recent rate
}
grandTime = grandTime1 + (lapseTime-localTime1)*grandRated;   // Grand-time estimate
errorTime = errorTime1 + (lapseTime-localTime1)*errorRated;   // Error-time estimate
errorPlus = errorTimer + delay * (rating - ONE);              // adds to cumulative
frame.grandTime = grandTimer;                                 // Extrapolate to future
frame.localTime = txB;                                        // Transmit snapshot
frame.errorTime = (errorTime + errorPlus);                    // adds to cumulative
```

7.2 timeSyncDuplex frame format

7.2.1 timeSyncDuplex fields

Duplex-link time-synchronization (timeSyncDuplex) 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 7.5. The gray boxes represent physical layer encapsulation fields that are common across Ethernet frames.

6	*da*	— Destination MAC address
6	*sa*	— Source MAC address
2	*protocolType*	— Distinguishes AVB frames from others
1	*function*	— Distinguishes timeSync from other AVB frames
1	*version*	— Distinguishes between timeSync frame versions
14	*precedence*	— Precedence for grand-master selection
10	*grandTime*	— Transmitter grand-time snapshot (1 cycle delayed)
4	*errorTime*	— Back-prediction error for grandTime computation
1	*frameCount*	— A (sequence number) count of time-sync frames
1	*hopCount*	— Hop count from the grand master
6	*localTime*	— Transmitter local-time snapshot (1 cycle delayed)
6	*thatTxTime*	— Opposing link's frame transmission time
6	*thatRxTime*	— Opposing link's frame reception time
4	*fcs*	— Frame check sequence
68 bytes total		

Figure 7.5—timeSyncDuplex 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 ***errorTime***, 8-bit ***hopCount***, and 6-byte ***localTime*** field are specified in 6.3.1.2.

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

7.2.1.2 *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 6.3.1.8).

7.2.1.3 *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 6.3.1.8).

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

7.2.2 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 messages can cancel the cable-delay effects.

Clock synchronization involves the processing of periodic events. Multiple time periods are involved, as listed in Table 7.1. 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 7.1—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.

7.3 TimeSyncRxDuplex state machine

7.3.1 Function

The TimeSyncRxDuplex 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 7.2; details of the computations are specified by the C-code of Annex F.

7.3.2 State machine definitions

HOPS
    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_MR_HOP—The queue identifier associated with MAC frames sent into the relay.
        Q_RX_MAC—The queue identifier associated with the received MAC frames.
        Q_RX_SYNC—The queue identifier associated with rxSync, sent from the lower levels.

7.3.3 State machine variables

cableDelay, *cableDelay0*
    Scaled integers representing cable-delay times.
curentTime
    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.
delta0, *delta1*
    Scaled integers representing times since the recent time-rating snapshots.
fPtr
    A pointer to a MAC-supplied *frame* (see below).
frame
    A MAC-supplied frame (see xx); the frame comprising the following.
        grandTime—A value synchronized to the grand-master time.
        localTime—The local time associated with the *grandTime* value.
        frameCount—A value that is incremented for successive timeSync transmissions.
        hopCount—Distance from the grand-master station, measured in station-to-station hops.
info
    A contents of a lower-level supplied time-synchronization poke indication, including the following:
        localTime—The value of *currentTime* associated with the last timeSync packet arrival.
        frameCount—The value of the like-named field within the last timeSync packet arrival.
pPtr
    A pointer to a data structure that contains port-specific information comprising the following:
        rxFrameCount—The value of *frameCount* within the last received frame.
        rxRated—The ratio of the local-station and remote-station local-timer rates.
        rxSnapCount—The value of *info.frameCount* saved from the last poke indication.
        rxSnapShot0—The *info.snapShot* field value from the last receive-port poke indication.
        rxSnapShot1—The value of the *pPtr->rxSnapShot1* field saved from the last poke indication.
        rxSyncFrame—The value of the previously observed timeSync frame.

rPtr

A pointer to a MAC-relay frame (see xx); the frame comprising the following.

grandTime—A value synchronized to the grand-master time.

localTime—The local time associated with the *grandTime* value.

sourcePort—Identifies the source port that generated the MAC-relay frame.

hopCount—Distance from the grand-master station, measured in station-to-station hops.

thisDelay, *thatDelay*, *thatDelay*, *thisDelta*, *thisTime*, *thatTime*, *tockTime*

Scaled integer representing intermediate local-time values.

7.3.4 State machine routines

BigAddition(x, y)

Returns the sum of 128-bit x and y values.

Dequeue(*queue*)

Returns the next available frame from the specified queue.

frame—The next available frame.

NULL—No frame available.

DivideHi(x, y)

Returns $(x/y)\times 2^{40}$ for integer values of x and y.

(This represents x/y, when y is assumed to be a scaled-integer.

DuplexToRelay(*pPtr*, *frame*)

Computes the average link-delay, based on neighbor-syntonized timers.

The averaged link-delay value is added to the frame, which is then forwarded over the MAC-relay.

Enqueue(*queue*)

Places the frame at the tail of the specified queue.

LongToBig(x)

Returns a sign-extended 128-bit version of value x.

MIN(x, y)

Returns the minimum of x and y values.

MultiplyHi(x, y)

Returns $(x\times y)\times 2^{-40}$ for integer values of x and y.

(This represents $x\times y$, when y is assumed to be a scaled-integer.

TimeSyncFrame(*frame*)

Checks the frame contents to identify timeSync frame.

TRUE—The frame is a timeSync frame.

FALSE—Otherwise.

7.3.5 TimeSyncRxDuplex state machine table

The TimeSyncRxDuplex state machine associates PHY-provided sync information with arriving timeSync frames and forwards adjusted frames to the MAC-relay function, as illustrated in Table 7.2

.

Table 7.2—TimeSyncRxDuplex state machine table

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

START	(info = Dequeue(Q_RX_SYNC))
	!= NULL
		1	// Summary of TimeSyncRxDuplexA()
			pPtr->rxSnapShot1 = pPtr->rxSnapShot0;
			pPtr->rxSnapShot0 = info.localTime;
			pPtr->rxSnapCount = info.frameCount;
				PAIR

	(frame = Dequeue(Q_RX_MAC))
	!= NULL
		2	// Summary of TimeSyncRxDuplexB()
			fPtr = &duplexFrame;
			count = (pPtr->rxFrameCount + 1) % COUNT;
			pPtr->rxFrameCount = fPtr->frameCount;
			wrong = (count **!**= fPtr->frameCount);
				TEST

	—	3	currentTime = GetLocalTime(pPtr);	START
TEST	**!**TimeSyncFrame(frame)	4	Enqueue(Q_MR_HOP, frame);	START
	fPtr->hopCount == HOPS	5	—	
	wrong	6	—	
	—	7	rPtr = &(relayFrame);	PAIR

PAIR	fPtr->frameCount **==**
	pPtr->rxSnapCount
		8	// Summary of TimeSyncRxDuplexC()
			thatTime = fPtr->localTime;
			thisTime = pPtr->rxSnapShot1;
			pPtr->rxThisTxTime = thatTime;
			pPtr->rxThisRxTime = thisTime;
			tockTime = pPtr->txThisTock;
			recent =
			thisTime – pPtr->rxThisTime0 >= 3 * tockTime;
			remote =
			thisTime – pPtr->rxThisTime1 >= 8 * tockTime;
				NEXT

	—	9	—	START

NEXT	recent && remote	10	// Summary of TimeSyncRxDuplexD()
			thisDelta = thisTime – pPtr->rxThisTime1;
			thatDelta = thatTime – pPtr->rxThatTime1;
			pPtr->rxRated = DivideHi(thisDelta, thatDelta);
			pPtr->rxThisTime1 = pPtr->rxThisTime0;
			pPtr->rxThatTime1 = pPtr->rxThatTime0;
			pPtr->rxThisTime0 = thisTime;
			pPtr->rxThatTime0 = thatTime;
				LAST

	—	11	—	

Table 7.2—TimeSyncRxDuplex state machine table

Current		Row	Next	
state	condition		action	state

LAST	—	12	// Summary of TimeSyncRxRelayC()
			rPtr = &relayFrame;
			localTime = pPtr->rxSnapShot1;
			roundTrip = localTime – pPtr->thatTxTime;
			turnRound = fPtr->localTime – fPtr->thatRxTime;
			cableDelay = MIN(0, roundTrip –
			 MultiplyHi(turnRound, pPtr->rxRated));
			SetRelayFrame(pPtr, rPtr);
			rPtr->grandTime = fPtr->grandTime + cableDelay;
			rPtr->localTime = localTime;
			rPtr->hopCount = fPtr->hopCount;
			Enqueue(Q_MR_HOP, relayFrame);
				START

Row 7.2-1: Update snapshot values on timeSync frame arrival.
Row 7.2-2: Initiate inspection of frames received from the lower-level MAC.
Row 7.2-3: Wait for the next change-of-state.

Row 7.2-4: The non-timeSync frames are passed through.
Row 7.2-5: Discard obsolete timeSync frames.
Row 7.2-6: Non-sequential frames are discarded.
Row 7.2-7: Sequential timeSync frames are processed.

Row 7.2-8: Inhibit processing when the frame and snap-shot counts are different.
Row 7.2-9: Broadcast revised timeSync frames over the MAC-relay.

Row 7.2-10: Periodic neighbor-timer ratings are performed.
Row 7.2-11: To reduce computation loads, neighbor-timer ratings are infrequently performed.
Row 7.2-12: The grand-master time is compensated by the timer-rate differences.

7.4 TimeSyncTxDuplex state machine

7.4.1 Function

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

7.4.2 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_TX_MAC—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.

7.4.3 State machine variables

curentTime

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.

frame

The contents of a MAC-supplied frame.

info

A contents of a lower-level supplied time-synchronization poke indication, including the following:

localTime—The value of *currentTime* associated with the last timeSync packet arrival.

frameCount—The value of the like-named field within the last timeSync packet arrival.

port

A data structure containing port-specific information comprising the following:

txSnapShot—The value of the *info*.*time* field saved from the last transmit-port poke indication.

txSyncFrame—The value of the next to-be-transmitted timeSync frame.

txSeenTime—The *currentTime* value when the last timeSync frame was received.

txSentTime—The *currentTime* value when the last timeSync frame enqueued for transmission.

7.4.4 State machine routines

Dequeue(*queue*)

Returns the next available frame from the specified queue.

frame—The next available frame.

NULL—No frame available.

Enqueue(*queue*)

Places the frame at the tail of the specified queue.

StateToTimes(*pPtr*, *frame*)

Transfers the frame to the MAC, as specified by the C code of Annex F.

RelayToState(*pPtr*, *frame*)

Copies a high-preference MAC-relay frame to port storage, as specified by the C code of Annex F. (Low preference MAC-relay frames are simply discarded.)

TimeSyncFrame(*frame*)

Checks the frame contents to identify timeSync frame.

TRUE—The frame is a timeSync frame.

FALSE—Otherwise.

7.4.5 TimeSyncTxDuplex state machine table

The TimeSyncTxDuplex 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 7.3.

Table 7.3—TimeSyncTxDuplex state machine table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(frame = Dequeue(Q_MR_HOP)) **!**= NULL	1	—	SINK
	(currentTime – pPtr->txSentTime) > T10ms	2	pPtr->txLastTime = currentTime;	SEND

	(info = Dequeue(Q_TX_SYNC))
	!= NULL
		3	// Summary of TimeSyncTxRelayA()
			pPtr->txSnapShot = info.localTime;
			pPtr->txSnapCount = info.frameCount;
				START

	—	4	currentTime = GetLocalTime(pPtr);	
SINK	**!**TimeSyncFrame(frame)	5	RelayToMac(pPtr, frame);	START
	RelayToState(pPtr, frame) == TOP	6	pPtr->txTestTimer = currentTime;	
	—	7	—	
SEND	pPtr->txHopCount >= HOPS	8	—	START

	—	9	// Summary of TimeSyncTxRelayB()
			dPtr = &duplexFrame;
			bothTimes =
			StateToGrand(pPtr, pPtr->txSnapShot);
			pPtr->txFrameCount =
			(pPtr->txSnapCount + 1) % COUNT;
			SetDuplexFrame(pPtr, dPtr);
			dPtr->hopCount = pPtr->txHopCount;
			dPtr->frameCount = pPtr->txFrameCount;
			dPtr->grandTime = bothTimes.grandTime;
			dPtr->errorTime = bothTimes.errorTime;
			dPtr->localTime = pPtr->txSnapShot;
			dPtr->rxThatTxTime = pPtr->rxThisTxTime;
			dPtr->rxThatRxTime = pPtr->rxThisRxTime;
			Enqueue(Q_TX_MAC, duplexFrame);	

Row 7.3-1: Relayed frames are further checked before being processed.
Row 7.3-2: Transmit periodic timeSync frames.
Row 7.3-3: Update snapshot values on timeSync frame departure.
Row 7.3-4: Wait for the next change-of-state.

Row 7.3-5: Non-timeSync frames are retransmitted in the standard fashion.
Row 7.3-6: Relevant timeSync parameters are saved for the next periodic transmission.
Row 7.3-7: MAC-relay frames from non grand-master stations are discarded.

Row 7.3-8: Discard obsolete timeSync frames.
Row 7.3-9: Form the next timeSync frame; enqueue this frame for immediate transmission.

8. Wireless state machines

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.

8.1 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.

8.2 Link-dependent indications

The wireless 802.11v TimeSyncRadio state machines are provided with MAC service-interface parameters, as illustrated within Figure 8.1. These link-dependent indications can be different for bridge ports attached to alternative media..

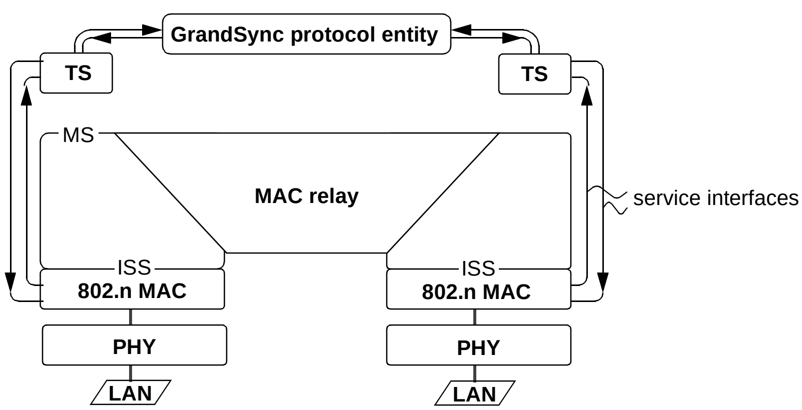


Figure 8.1—Radio interface model

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 8.2.

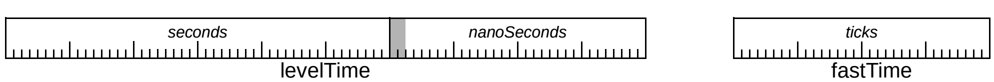


Figure 8.2—Formats of wireless-dependent times

8.3 Service interface overview

A sequence of 802.11v TimeSync service interface actions is illustrated in Figure 8.3. A periodic trigger is assumed to initiate the initial MLME_PRESENCE_REQUEST.request action. Processing of the returned MLME_PRESENCE_REQUEST.confirm triggers the following MLME_PRESENCE_RESPONSE.request action. The sequence completes with the final MLME_PRESENCE_RESPONSE.confirm action.

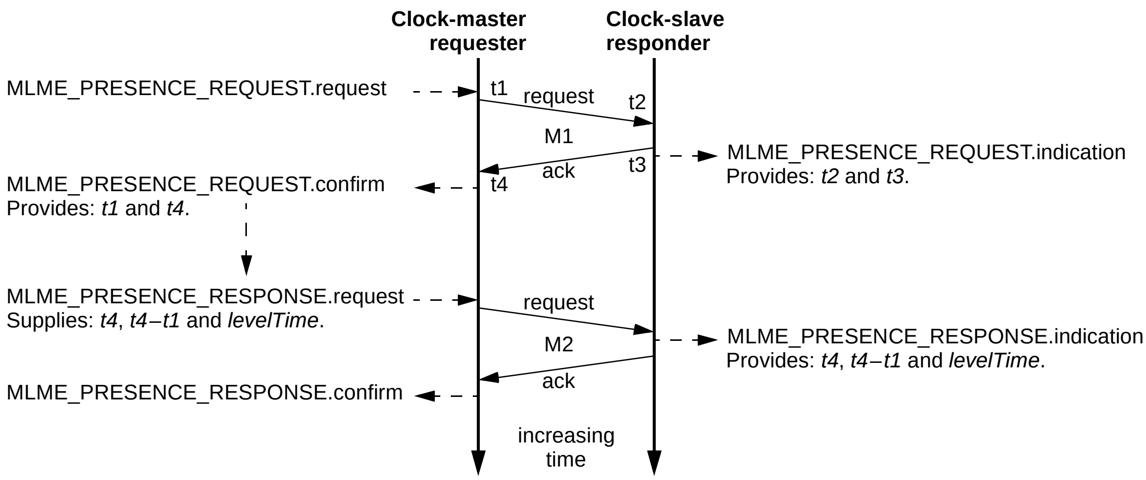


Figure 8.3—802.11v time-synchronization interfaces

The properties of these service interfaces are summarized below:

MLME_PRESENCE_REQUEST.request
- Generated periodically by the clock-master entity.
- Triggers an M1.request, to update clock-slave resident timing parameters.

MLME_PRESENCE_REQUEST.indication
- Generated in response to receiving an M1.request message.
- Provides *t2* and *t3* timing information to the clock-slave entity.

MLME_PRESENCE_REQUEST.confirm
- Generated after the M1.ack message is returned.
- Provides *time1* and *time4* timing information to the clock-master entity.

MLME_PRESENCE_RESPONSE.request
- Generated shortly after processing a returned M1.ack message
- Triggers an M1.request, to update clock-slave resident timing parameters.

MLME_PRESENCE_RESPONSE.indication
- Generated in response to receiving an M2.request message.
- Provides *time4*, *time4–time1*, and *levelTime* information to the clock-slave entity.

MLME_PRESENCE_RESPONSE.confirm
- Generated after the M2.ack message is returned.
- Confirms completion of the time-synchronization exchange.

8.4 TimeSyncRxRadio state machine

8.4.1 Function

The TimeSyncRxRadio state machine consumes primitives provided by the MAC service interface and (in response) generates MAC-relay frames.

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—Queue identifier associated with MAC frames sent into the relay.
    Q_S1_IND—Queue identifier for MLME_PRESENCE_REQUEST.indication parameters.
    Q_S2_IND—Queue identifier for MLME_PRESENCE_RESPONSE.indication parameters.

8.4.3 State machine variables

args1
  A set of values returned within the MLME_PRESENCE_REQUEST.indication service primitive:
    fastTime2—A local-timer snapshot corresponding to the time of M1.request reception.
    fastTime3—A local-timer snapshot corresponding to the time of M1.ack transmission.
args2
  A set of values provided to the MLME_PRESENCE_RESPONSE.indication service primitive:
    fastTime4—A neighbor-timer snapshot corresponding to the time of M1.ack reception.
    fastTimed—A neighbor-timer snapshot corresponding to a time difference:
      (M2.request transmission) – (M1.request transmission)
    levelTime—Grand-master synchronized time at the *fastTime4* neighbor-time snapshot.
curentTime
  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.
frame
  The contents of a MAC-supplied frame.
port
  A data structure containing port-specific information, including the following:
    rxFastTime2—Saved *args1*.*fastTime2* value.
    rxFastTime3—Saved *args1*.*fastTime3* value.
    rxFastTime4—Saved *args2*.*fastTime4* value.
    rxFastTimed—Saved *args2*.*fastTimed* value.
    rxLevelTime—Saved *args2*.*levelTime* value.

8.4.4 State machine routines

DequeueService(*queue*)
  Returns service parameters from the specified queue.
    args—The next available service parameters.
    NULL—No frame available.
Enqueue(*queue*)
  Places the frame at the tail of the specified queue.

PonToRelay(*pPtr*)
Computes the average link-delay, based on neighbor-syntonized timers.
The averaged link-delay value is added to the frame, which is then forwarded over the MAC-relay.

8.4.5 TimeSyncRxRadio state table

The TimeSyncRxRadio state machine consumes MAC-provided service-primitive information and forwards adjusted frames to the MAC-relay function, as illustrated in Table 8.1.

Table 8.1—TimeSyncRxRadio state machine table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(req1 = Dequeue(Q_S1_IND)) != NULL	1	// Summary of TimeSyncRxRadio1Indicate() pPtr->rxTurnRound = req1.fastTime3 – req1.fastTime2;	WAIT
	—	2	localTimes = RadioLocalTimes(pPtr);	START

WAIT	(req2 = Dequeue(Q_S2_IND))
	!= NULL
		3	// Summary of TimeSyncRxRadio2Indicate()
			rPtr = &(relayFrame);
			twice = req2.roundTrip – pPtr->rxTurnRound;
			moved = localTimes.ticksTime – req2.fastTime4;
			SetRelayFrame(pPtr, rPtr);
			rPtr->grandTime =
			RadioToGrand(req2.radioTime) +
			MultiplyHi((twice/2) + moved, RADIO_TIME);
			rPtr->localTime = localTimes.localTime;
			Enqueue(Q_MR_HOP, relayFrame);
				START

	—	4		—

Row 8.1-1: Wait until indication parameters become available.
Row 8.1-2: Update snapshot values based on MLME_PRESENCE_REQUEST.indication parameters.

Row 8.1-3: Wait until indication parameters become available.
Row 8.1-4: Update snapshot values based on MLME_PRESENCE_RESPONSE.indication parameters. Based on those parameters, generate a timeSync frame for MAC-relay transmission.

8.5 TimeSyncTxRadio state machine

8.5.1 Function

The TimeSyncTxRadio state machine consumes MAC-relay frames and (in response) generates calls to the time-synchronization related MAC service interface.

8.5.2 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_MR_HOP—The queue identifier associated with MAC frames sent into the relay.
    - Q_RX_MAC—The queue identifier associated with the received MAC frames.
    - Q_RX_SYNC—The queue identifier associated with rxSync, sent from the lower levels.

8.5.3 State machine variables

args1
: A set of values returned within the MLME_PRESENCE_REQUEST.confirm service primitive:
    - *time1*—A local-timer snapshot corresponding to the time of M1.request transmission.
    - *time2*—A local-timer snapshot corresponding to the time of M2.request reception.

args2
: A set of values provided to the MLME_PRESENCE_REQUEST.request service primitive:
    - *time1*—The value of the previously returned *args1*.*time1* value.
    - *timed*—The difference of previously returned values: *args1*.*time4* – *args1*.*time1*.
    - *levelTime*—The value of *grandTime* associated with the returned *args1*.*time1* timer.

curentTime
: 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.

frame
: The contents of a MAC-supplied frame.

reqArgs
: MLME_PRESENCE_REQUEST.request parameters unrelated to time-synchronization services.

resArgs
: MLME_PRESENCE_RESPONSE.request parameters unrelated to time-synchronization services.

port
: A data structure containing port-specific information for determining *grandTime* values.

8.5.4 State machine routines

Dequeue(*queue*)
: Returns the next available frame from the specified queue.
    - *frame*—The next available frame.
    - NULL—No frame available.

EnqueueService(*queue*)
: Places the service-interface parameters in the specified queue.

StateToTimes(*pPtr*, *frame*)
: Transfers the non-timeSync frame to the MAC.

RelayToState(*pPtr*, *frame*)

Copies a high-preference MAC-relay frame to port storage, as specified by the C code of Annex F. (Low preference MAC-relay frames are simply discarded.)

TimeSyncFrame(*frame*)

Checks the frame contents to identify timeSync frame.

TRUE—The frame is a timeSync frame.

FALSE—Otherwise.

8.5.5 TimeSyncTxRadio state table

NOTE—The following state machine is highly preliminary; sequence timeouts and grand-master selection code are not yet included.

The TimeSyncTxRadio 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 8.2.

Table 8.2—TimeSyncTxRadio state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(frame = Dequeue(Q_MR_HOP)) != NULL	1	—	SINK
	(currentTime – pPtr->txSentTime) > T10ms	2	pPtr->txSentTime = currentTime;	SEND

	(con1 = Dequeue(Q_S1_CON))
	!= NULL
		3	// Summary of TimeSyncRxRadio1Confirm()
			pPtr->txSnapShot1 = con1.ticksTime1;
			pPtr->txRoundTrip =
			con1.ticksTime4 – con1.ticksTime1;
			phase2 = TRUE;
				WAIT

	—	4	localTimes = RadioLocalTimes(pPtr);	START
SINK	!TimeSyncFrame(frame)	5	RelayToMac(pPtr, frame);	START
	RelayToState(pPtr, frame) == TOP	6	pPtr->txTestTimer = currentTime;	
	—	7	—	
SEND	pPtr->txFrame.hopCount == HOPS	8	—	START
	—	9	EnqueueService(Q_S1_REQ, reqArgs);	WAIT1

WAIT1	phase2 == TRUE	10	// Summary of TimeSyncTxRadio2Request()
			lapseTime =
			localTimes.radioTime – pPtr->txSnapShot4;
			localTime = localTimes.localTime –
			MultiplyHi(lapseTime, RADIO_TIME);
			grandTimes = StateToGrand(pPtr, localTime);
			req2.ticksTime4 = pPtr->txSnapShot4;
			req2.roundTrip = pPtr->txRoundTrip;
			req2.levelTime =
			GrandToRadio(grandTimes.grandTime);
			req2.errorTime = grandTimes.errorTime;
			req2.precedence = pPtr->txPrecedence;
			req2.hopCount = pPtr->txHopCount;
			EnqueueService(Q_S2_REQ, req2);
				WAIT2

	—	11		—
WAIT2	(args2 = Dequeue(Q_S2_CON)) == NULL	12	—	WAIT1
	—	13	—	START

Row 8.2-1: Relayed frames are further checked before being processed.
Row 8.2-2: Initiate periodic service-interface primitive calls.
Row 8.2-4: Wait for the next change-of-state.

Row 8.2-5: Non-timeSync frames are retransmitted in the standard fashion.
Row 8.2-6: Relevant timeSync parameters are saved for the next periodic transmission.
Row 8.2-7: MAC-relay frames from non grand-master stations are discarded.

Row 8.2-8: Discard obsolete timeSync frames.
Row 8.2-9: Transmit parameters through the MLME_PRESENCE_REQUEST.request interface.

Row 8.2-10: Wait for parameters arriving through the MLME_PRESENCE_REQUEST.confirm interface.
Row 8.2-11: Transmit parameters through the MLME_PRESENCE_RESPONSE.request interface.

Row 8.2-12: Wait for parameters arriving through the MLME_PRESENCE_RESPONSE.confirm interface.
Row 8.2-13: Confirm completion and continue.

9. Ethernet-PON 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 media-independent MAC-relay interface, while also triggering discussion of 802.3-PON design details. As such, the contents are highly preliminary and subject to change.

9.1 Overview

This clause specifies the state machines that support Ethernet-PON 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 TimeSyncPon state machines have knowledge of network-local synchronized timers. With this knowledge, the TimeSyncPon state machines can operated on frames received from the LLC, as illustrated in Figure 9.1. Link-dependent indications could be required for bridge ports attached to alternative media..

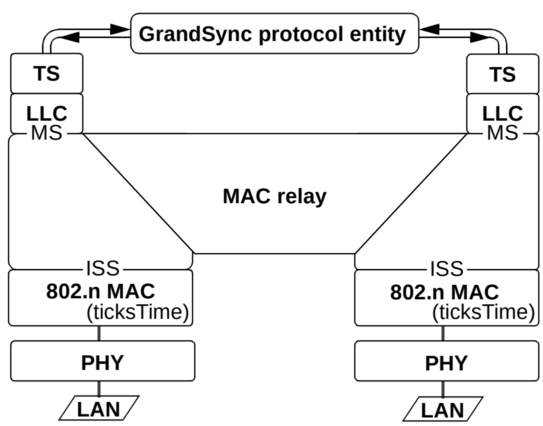


Figure 9.1—PON interface model

The *localTime* values are represented as timers that are incremented once every 16 ns interval, as illustrated on the left side of Figure 9.2. 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 9.2.

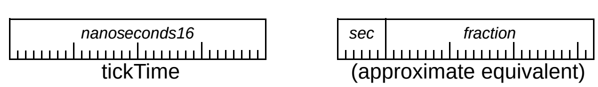


Figure 9.2—Format of PON-dependent times

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 TimeSyncPon state machines have access to these synchronized *ticksTime* values.

9.2 timeSyncPon frame format

The timeSyncPon 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 9.3. The gray boxes represent physical layer encapsulation fields that are common across Ethernet frames.

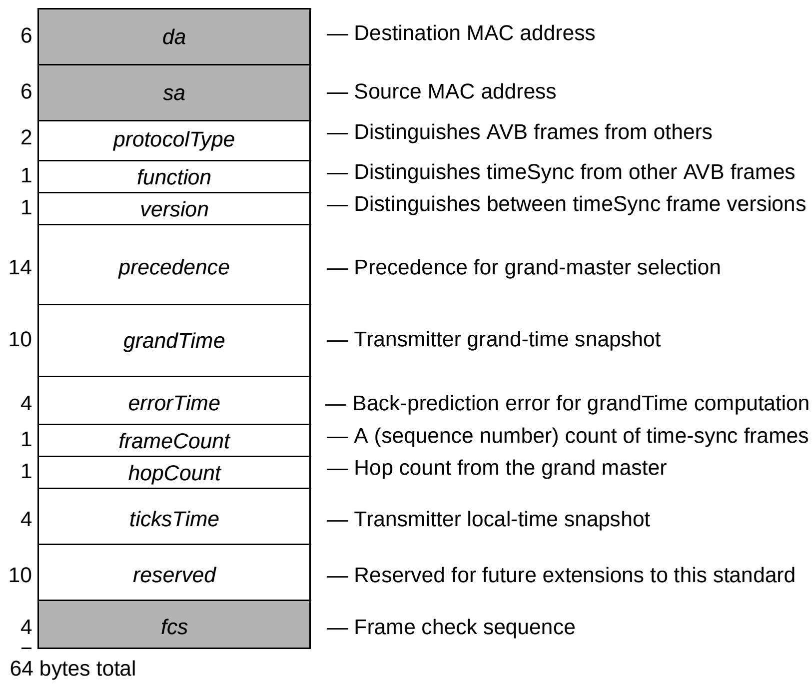


Figure 9.3—timeSyncPon frame format

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 ***errorTime***, and 8-bit ***hopCount*** fields are specified in 6.3.1.2.

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

9.2.2 *ticksTime*: A value representing local time in units of a 16 ns timer ticks, as illustrated in Figure 9.4.

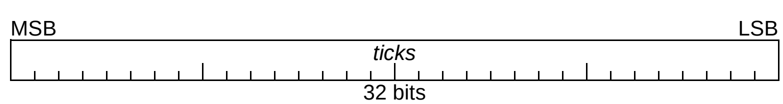


Figure 9.4—tickTime format

9.3 TimeSyncRxPon state machine

9.3.1 Function

The TimeSyncRxPon 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 9.1; details of the computations are specified by the C-code of Annex F.

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_MR_HOP—The queue identifier associated with MAC frames sent into the relay.
        Q_RX_MAC—The queue identifier associated with the received MAC frames.

9.3.3 State machine variables

frame
    The contents of a MAC-supplied frame.
port
    A data structure containing port-specific information comprising the following:
        rxFrame—The last received frame.
        rxFrameCount—The value of *frameCount* within the last received frame.
        rxSyncFrame—The value of the previously received timeSync frame.

9.3.4 State machine routines

Dequeue(*queue*)
    Returns the next available frame from the specified queue.
        frame—The next available frame.
        NULL—No frame available.
Enqueue(*queue*)
    Places the frame at the tail of the specified queue.
PonToRelay(*pPtr*, *frame*)
    Computes the average link-delay, based on neighbor-syntonized timers.
    The averaged link-delay value is added to the frame, which is then forwarded over the MAC-relay.
TimeSyncFrame(*frame*)
    Checks the frame contents to identify timeSync frame.
        TRUE—The frame is a timeSync frame.
        FALSE—Otherwise.

9.3.5 TimeSyncRxPon state machine table

The TimeSyncRxPon state machine associates PHY-provided sync information with arriving timeSync frames and forwards adjusted frames to the MAC-relay function, as illustrated in Table 7.2.

Table 9.1—TimeSyncRxPon state machine table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(frame = Dequeue(Q_RX_MAC)) **!**= NULL	1	—	TEST
	—	2	ponTimes = PonLocalTimes(pPtr);	START
TEST	**!**TimeSyncFrame(frame)	3	Enqueue(Q_MR_HOP, frame);	START

	—	4	// Summary of TimeSyncRxPon()
			rPtr = &relayFrame;
			SetRelayFrame(pPtr, rPtr);
			lapseTime = ponTimes.ponTime – frame.ticksTime;
			rPtr->grandTime = frame.grandTime;
			rPtr->errorTime = frame.errorTime;
			rPtr->localTime = ponsTimes.localTime –
			MultiplyHi(lapseTime, PON_TIME);
			rPtr->hopCount = frame.hopCount;
			Enqueue(Q_MR_HOP, relayFrame);	

Row 9.1-1: Initiate inspection of frames received from the lower-level MAC.
Row 9.1-2: Wait for the next frame to arrive.

Row 9.1-3: The non-timeSync frames are passed through.
Row 9.1-4: Sequential timeSync frames are processed.

9.4 TimeSyncTxPon state machine

9.4.1 Function

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

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_MR_HOP—The queue identifier associated with frames sent from the relay.
Q_TX_MAC—The queue identifier associated with frames sent to the MAC.
T10ms
A constant the represents a 10 ms value.

9.4.3 State machine variables

curentTime

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.

frame

The contents of a MAC-supplied frame.

port

A data structure containing port-specific information comprising the following:

txSyncFrame—The value of the next to-be-transmitted timeSync frame.

txSeenTime—The *currentTime* value when the last timeSync frame was received.

tickTime

A shared value representing current time. There is one instance of this synchronized variable for each port. This 32-bit counter is incremented once very 16 ns.

9.4.4 State machine routines

Dequeue(*queue*)

Returns the next available frame from the specified queue.

frame—The next available frame.

NULL—No frame available.

Enqueue(*queue*)

Places the frame at the tail of the specified queue.

StateToTimes(*pPtr, frame*)

Transfers the frame to the MAC, as specified by the C code of Annex F.

RelayToState(*pPtr, frame, currentTime*)

Copies a high-preference MAC-relay frame to port storage, as specified by the C code of Annex F. (Low preference MAC-relay frames are simply discarded.)

TimeSyncFrame(*frame*)

Checks the frame contents to identify timeSync frame.

TRUE—The frame is a timeSync frame.

FALSE—Otherwise.

9.4.5 TimeSyncTxPon state machine table

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

Table 9.2—TimeSyncTxPon state machine table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(frame = Dequeue(Q_MR_HOP)) **!**= NULL	1	—	SINK
	(currentTime – pPtr->txSentTime) > T10ms	2	pPtr->txLastTime = currentTime;	SEND
	—	3	currentTime = GetLocalTime(pPtr);	START
SINK	**!**TimeSyncFrame(frame)	4	RelayToMac(pPtr, frame);	START
	RelayToState(pPtr, frame) == TOP	5	pPtr->txTestTimer = currentTime;	
	—	6	—	
SEND	pPtr->txFrame.hopCount == HOPS	7	—	START

	—	8	// Summary of TimeSyncTxPon()
			dPtr = &ponFrame;
			localTimes = PonLocalTimes(pPtr);
			grandTimes = StateToGrand(pPtr, localTime);
			SetPonFrame(pPtr, dPtr);
			dPtr->precedence = pPtr->txPrecedence;
			dPtr->hopCount = pPtr->txHopCount;
			dPtr->grandTime = bothTimes.grandTime;
			dPtr->errorTime = bothTimes.errorTime;
			dPtr->ticksTime = dualTimes.ticksTime;
			EnqueueService(Q_S2_REQ, ponFrame);	

Row 9.2-1: Relayed frames are further checked before being processed.
Row 9.2-2: Transmit periodic timeSync frames.
Row 9.2-3: Wait for the next change-of-state.

Row 9.2-4: Non-timeSync frames are retransmitted in the standard fashion.
Row 9.2-5: Relevant timeSync parameters are saved for the next periodic transmission.
Row 9.2-6: MAC-relay frames from non grand-master stations are discarded.

Row 9.2-7: Discard obsolete timeSync frames.
Row 9.2-8: Form the next timeSync frame and enqueue this frame for immediate transmission.

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

The synchronized value of *grandTime* (grand-master time) is based on the Precision Time Protocol (PTP). Time is measured in international seconds since the start of January 1, 1970 Greenwich Mean Time (GMT). Other representations of time can be readily derived from the values of *grandTime* and a distributed *leapSeconds* value, as specified in Table B.1.

Table B.1—Time-scale conversions

Acronym	Name	Row	offset	Algorithm
PTP	Precision Time protocol	1	0	time = grandTime + offset;
GPS	global positioning satellite	2	–315 964 819	
TAI	????	3	????	
UTC	Coordinated Universal Time	4	TBD	time = grandTime + offset – leapSeconds;
NTP	Network Time Protocol	5	+2 208 988 800	

NOTE—The PTP time is commonly used in POSIX algorithms for converting elapsed seconds to the ISO 8601-2000 printed representation of time of day.

Annex C

(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.

C.1 Hybrid network topologies

C.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 C.1. The boundary between domains is illustrated by a dotted line, which passes through a SerialBus adapter station.

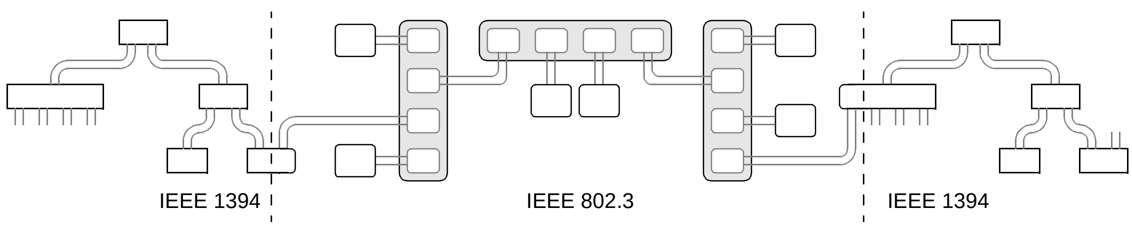


Figure C.1—IEEE 1394 leaf domains

C.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 C.2. While not explicitly prohibited, architectural features of such topologies are beyond the scope of this working paper.

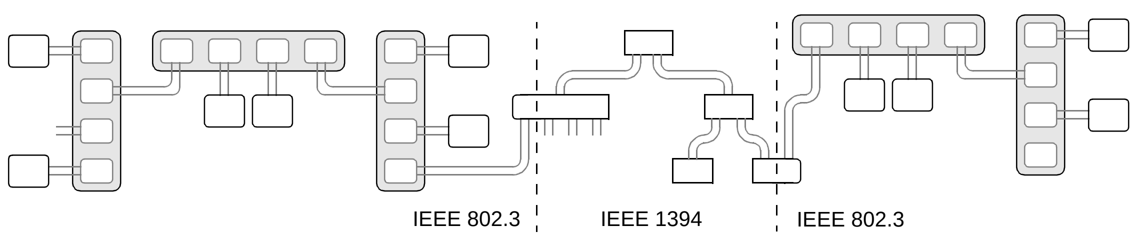


Figure C.2—IEEE 802.3 leaf domains

C.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 C.3.

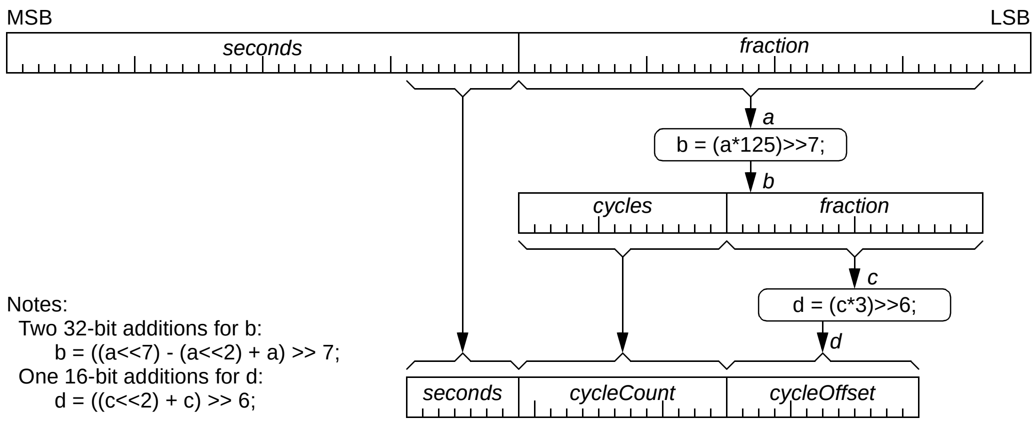


Figure C.3—Time-of-day format conversions

C.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 C.4:

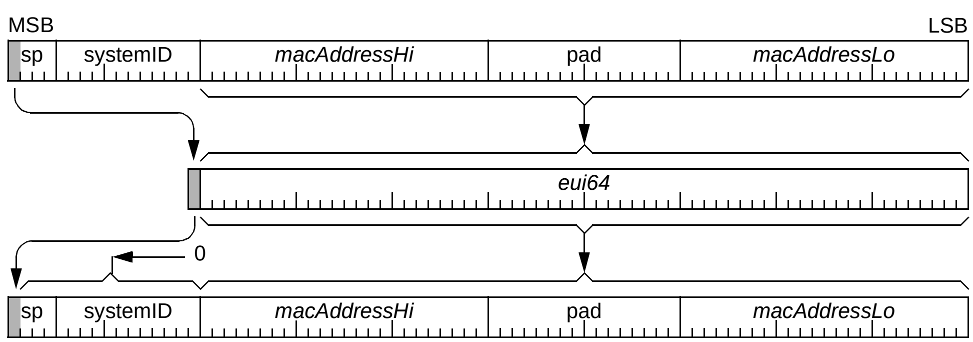


Figure C.4—Grand-master precedence mapping

Annex D

(informative)

Review of possible alternatives

D.1 Clock-synchronization alternatives

NOTE—This tables has not been reviewed for considerable time and is thus believed to be inaccurate. However, the list is being maintained (until it can be updated) for its usefulness as talking points.

A comparison of the AVB and IEEE 1588 time-synchronization proposals is summarized in Table D.1.

Table D.1—Protocol comparison

Properties		Row	Descriptions	
state			**AVB-SG**	**1588**
timeSync MTU <= Ethernet MTU		1	yes	
No cascaded PLL whiplash		2	yes	
Number of frame types		3	1	> 1
Phaseless initialization sequencing		4	yes	no
Topology		5	duplex links	general
Grand-master precedence parameters		6	spanning-tree like	special
Rogue-frame settling time, per hop		7	10 ms	1 s
Arithmetic complexity	numbers	8	64-bit binary	2 x 32-bit binary
	negatives	9	2’s complement	signed
Master transfer discontinuities	rate	10	gradual change	
	offset limitations	11	duplex-cable match sampling error	
Firmware friendly	no delay constraints	12	yes	
	n-1 cycle sampling	13	yes	
Time-of-day value precision	offset resolution	14	233 ps	
	overflow interval	15	136 years	

Row 1: The size of a timeSync frame should be no larger than an Ethernet MTU, to minimize overhead.
AVB-SG: The size of a timeSync frame is an Ethernet MTU.
1588: The size of a timeSync frame is (to be provided).

Row 2: Cascaded phase-lock loops (PLLs) can yield undesirable whiplash responses to transients.
AVB-SG: There are no cascaded phase-lock loops.
1588: There are multiple initialization phases (to be provided).

Row 3: There number of frame types should be small, to reduce decoding and processing complexities.
AVB-SG: Only one form of timeSync frame is used.
1588: Multiple forms of timeSync frames are used (to be provided).

Row 4: Multiple initialization phases adds complexity, since miss-synchronized phases must be managed.
AVB-SG: There are no distinct initialization phases.
1588: There are multiple initialization phases (to be provided).

Row 5: Arbitrary interconnect topologies should be supported.
AVB-SG: Topologies are constrained to point-to-point full-duplex cabling.
1588: Supported topologies include broadcast interconnects.

Row 6: Grand-master selection precedence should be software configurable, like spanning-tree parameters.
AVB-SG: Grand-master selection parameters are based on spanning-tree parameter formats.
1588: Grand-master selection parameters are (to be provided).

Row 7: The lifetime of rogue frames should be minimized, to avoid long initialization sequences.
AVB-SG: Rogue frame lifetimes are limited by the 10 ms per-hop update latencies.
1588: Rogue frame lifetimes are limited by (to be provided).

Row 8: The time-of-day formats should be convenient for hardware/firmware processing.
AVB-SG: The time-of-day format is a 64-bit binary number.
1588: The time-of-day format is a (to be provided).

Row 9: The time-of-day negative-number formats should be convenient for hardware/firmware processing.
AVB-SG: The time-of-day format is a 2’s complement binary number.
1588: The time-of-day format is a (to be provided).

Row 10: The rate discontinuities caused by grand-master selection changes should be minimal.
AVB-SG: Smooth rate-change transitions with a 2.5 second time constant is provided.
1588: (To be provided).

Row 11: The time-of-day discontinuities caused by grand-master selection changes should be minimal.
AVB-SG: Maximum time-of-day errors are limited by cable-length asymmetry and time-snapshot errors.
1588: (To be provided).

Row 12: Firmware friendly designs should not rely on fast response-time processing.
AVB-SG: Response processing time have no significant effect on time-synchronization accuracies.
1588: (To be provided).

Row 13: Firmware friendly designs should not rely on immediate or precomputed snapshot times.
AVB-SG: Snapshot times are never used within the current cycle, but saved for next-cycle transmission.
1588: (To be provided).

Row 14: The fine-grained time-of-day resolution should be small, to facilitate accurate synchronization.
AVB-SG: The 64-bit time-of-day timer resolution is 233 ps, less than expected snapshot accuracies.
1588: (To be provided).

Row 15: The time-of-day extent should be sufficiently large to avoid overflows within one’s lifetime.
AVB-SG: The 64-bit time-of-day timer overflows once every 136 years.
1588: (To be provided).

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 5.1.

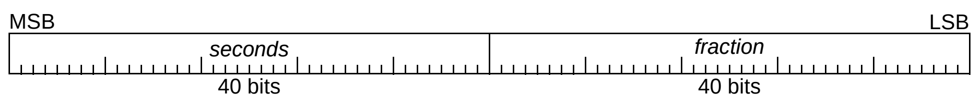


Figure 5.1—Global-time subfield format

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

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

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.

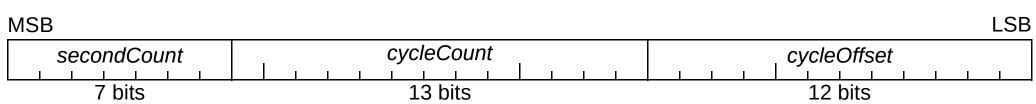


Figure E.2—IEEE 1394 timer format

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.

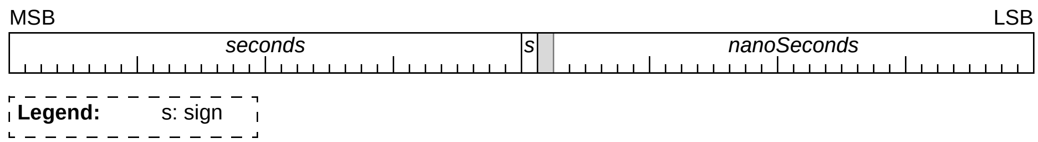
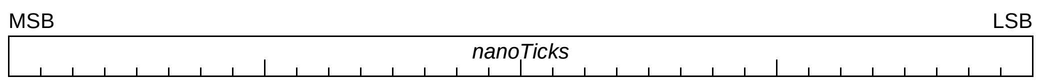


Figure E.3—IEEE 1588 timer format

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.



seconds = *nanoTicks*/62 500 000

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
//345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901212

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

#include <assert.h>
#include <stdio.h>

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

// Unique identifier values
#define AVB_TYPE  0                                                                      // The protocolType for AVB.
#define NEIGHBOR  0                                                                      // Neighbor multicast address.
#define TIME_SYNC 0                                                                      // The timeSync function.
#define VERSION_A 1                                                                      // The timeSync version.

// Generic macro definitions
#define ABS(a) ((a) < 0 ? (-a) : (a))                                                    // Minimum value definition
#define BITS(type) (8 * sizeof(type))
#define CLIP_RATE(x, y) ((x) > ONE+(y) ? ONE+(y) : ((x) < ONE-(y) ? ONE-(y) : (x)))      // Clip within specified rate
#define CLIP_SIZE(x, y) ((x) > (y) ? (y) : ((x) < (y) ? (y) : (x)))                      // Clip within specified value
#define COUNT 256                                                                        // Number of frameCount values
#define FALSE 0
#define HOPS 255                                                                         // Largest hop-count value
#define MASK(bits) (((uint64_t)1 << bits) - 1)
#define MIN(a, b) ((a) > (b) ? (b) : (a))                                                // Minimum value definition
#define ONE ((uint64_t)1 << 40)                                                          // Scaled fraction for 1.0
#define PON_TIME (DivideHi(16 * ((uint64_t)1 << 32), 1000000000))
#define PPM250 ((ONE * 250) / 1000000)                                                   // Scaled 250PPM fraction.
#define RADIO_TIME DivideHi(1 << (32 - 9), 1000000000 >> 9)                              // Ratio radio-ns to localTime
#define T10ms (ONE / 100)                                                                // A 10ms error interval
#define TIMEOUT TRUE
#define TOP   0
#define TRUE  1

// Field extract/deposit definitions
#define FieldToSigned(fPtr, field) \
 FrameToValue((uint8_t *)(&(fPtr->field)), sizeof fPtr->field, TRUE)                     // Convert field to signed
```

```
#define FieldToUnsigned(fPtr, field) \
 FrameToValue((uint8_t *)(&(fPtr->field)), sizeof fPtr->field, FALSE)            // Convert field to unsigned
#define BigToFrame(value, fPtr, field) \
 ValueToFrame(value, (uint8_t *)(&(fPtr->field)), sizeof fPtr->field)            // Convert field to unsigned
#define LongToFrame(value, fPtr, field) \
 ValueToFrame(LongToBig(value), (uint8_t *)(&(fPtr->field)), sizeof fPtr->field)

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

typedef uint8_t   Boolean;
typedef uint8_t   Class;
typedef uint8_t   HopCount;
typedef uint8_t   Port;
typedef uint16_t  Variance;
typedef uint32_t  ErrorTime;
typedef uint32_t  PonTime;
typedef uint32_t  RadioTime;
typedef int64_t   LocalTime;
typedef BigNumber GrandTime;
typedef BigNumber Precedence;                                                      // Fields {priorities,clockID}
typedef BigNumber Preference;                                                      // Fields {precedence,hops,port}

typedef struct
{                                                                                  // Double-precise integers
    GrandTime grandTime;                                                           // Grand-master synchronized
    LocalTime errorTime;                                                           // Side-band error values
} GrandTimes;

typedef struct
{                                                                                  // Double-precise integers
    LocalTime localTime;                                                           // Local free-running
    RadioTime radioTime;                                                           // Side-band error values
} RadioTimes;

typedef struct
{                                                                                  // Double-precise integers
    LocalTime localTime;                                                           // Local free-running
    PonTime   ponTime;                                                             // PON media-dependent
} PonTimes;


typedef struct
{
    Precedence precedence;
    HopCount   hopCount;
    Port       port;
} TimeSyncInfo;

typedef struct
{
    TimeSyncInfo info;
```

```
    LocalTime    timer;
} GrandSyncInfo;


// May be obsoleted
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 (for last frame)
    uint8_t errorTime[4];                       // Cumulative GM-time errors
    uint8_t sourcePort[1];                      // Transmit count (sequence number)
    uint8_t hopCount[1];                        // Hop-count from the grand master
    uint8_t localTime[6];                       // Transmitted timeSync time
    uint8_t tockTxTime[6];                      // Opposite-link transmit time
} TimeSyncRelay;

typedef struct
{
    uint8_t   frameCount;                       // Sequential consistency check
    GrandTime grandTime;                        // Received grand-master time
} ClockInfoReq;

typedef struct
{
    uint8_t   infoCount;                        // Sequential consistency check
    GrandTime grandTime;                        // Grand-master time
} ClockInfoRes;


typedef struct
{
    uint8_t   frameCount;                       // Sequential consistency check
    LocalTime localTime;                        // Station-local time
} DuplexRxInfo;

typedef struct
{
    uint8_t   frameCount;                       // Sequential consistency check
    LocalTime localTime;                        // Station-local time
} DuplexTxInfo;

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 (for last frame)
```

```
    uint8_t errorTime[4];                                                  // Cumulative GM-time errors
    uint8_t frameCount[1];                                                 // Transmit count (sequence number)
    uint8_t hopCount[1];                                                   // Hop-count from the grand master
    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];                                                        // CRC integrity check
} TimeSyncDuplex;

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 (for last frame)
    uint8_t errorTime[4];                                                  // Cumulative GM-time errors
    uint8_t frameCount[1];                                                 // Transmit count (sequence number)
    uint8_t hopCount[1];                                                   // Hop-count from the grand master
    uint8_t ticksTime[4];                                                  // Transmitted timeSync time
    uint8_t unused[8];                                                     // Opposite-link transmit time
    uint8_t fcs[4];                                                        // CRC integrity check
} TimeSyncPon;


typedef struct
{
    uint32_t ticksTime2;                                                   // Received snapshot
    uint32_t ticksTime3;                                                   // Transmit snapshot
} RadioInfo1Ind;

typedef struct
{
    uint32_t ticksTime1;                                                   // Transmit snapshot
    uint32_t ticksTime4;                                                   // Received snapshot
} RadioInfo1Con;

typedef struct
{
    uint32_t   ticksTime4;                                                 // Received snapshot
    uint32_t   roundTrip;                                                  // Duration snapshot
    GrandTime  levelTime;                                                  // Grand-master like
    ErrorTime  errorTime;                                                  // Grand-master error
    Precedence precedence;                                                 // Grand-master error
    HopCount   hopCount;                                                   // Grand-master error
} RadioInfo2Req;

typedef struct                                                             // Port entity state
{
    uint64_t       macAddress;                                             // MAC address of the port
    uint8_t        portID;                                                 // Destinctive port identifier
    BigNumber      txPrecedence;                                           // Grand-master preference
    uint8_t        txHopCount;                                             // Next hop-count value
```

```
    LocalTime      txTestTimer;                        // Relay-frame received time
    LocalTime      txThisTock;                         // Relay-frame tock-time
    LocalTime      txThatTock;                         // Relay-frame tock-time
    LocalTime      txTockTime;                         // Clock-master’s tockTime
    LocalTime      txPastTime;                         // Back-interpolation time
    BigNumber      txPreference;                       // Grand-master preference
    LocalTime      txGrandRate0;                       // Recent grandTime rating
    LocalTime      txGrandRate1;                       // Remote grandTime rating
    LocalTime      txErrorRate0;                       // Recent errorTime rating
    LocalTime      txErrorRate1;                       // Remote errotTime rating
    GrandTime      txGrandTime0;                       // Recent grandTime endpoint
    GrandTime      txGrandTime1;                       // Remote grandTime midpoint
    LocalTime      txLocalTime0;                       // Recent localTime endpoint
    LocalTime      txLocalTime1;                       // Remote localTime midpoint
    LocalTime      txErrorTime0;                       // Recent errorTime endpoint
    LocalTime      txErrorTime1;                       // Remote errorTime midpoint
} PortData;

typedef struct                                         // Port entity state
{
    uint64_t       macAddress;                         // MAC address of the port
    uint8_t        portID;                             // Destinctive port identifier
    BigNumber      txPrecedence;                       // Grand-master preference
    uint8_t        txHopCount;                         // Next hop-count value
    LocalTime      txTestTimer;                        // Relay-frame received time
    LocalTime      txThisTock;                         // Relay-frame tock-time
    LocalTime      txThatTock;                         // Relay-frame tock-time
    LocalTime      txTockTime;                         // Clock-master’s tockTime
    LocalTime      txPastTime;                         // Back-interpolation time
    BigNumber      txPreference;                       // Grand-master preference
    LocalTime      txGrandRate0;                       // Recent grandTime rating
    LocalTime      txGrandRate1;                       // Remote grandTime rating
    LocalTime      txErrorRate0;                       // Recent errorTime rating
    LocalTime      txErrorRate1;                       // Remote errotTime rating
    GrandTime      txGrandTime0;                       // Recent grandTime endpoint
    GrandTime      txGrandTime1;                       // Remote grandTime midpoint
    LocalTime      txLocalTime0;                       // Recent localTime endpoint
    LocalTime      txLocalTime1;                       // Remote localTime midpoint
    LocalTime      txErrorTime0;                       // Recent errorTime endpoint
    LocalTime      txErrorTime1;                       // Remote errorTime midpoint

    LocalTime      rxSnapShot0;                        // This frame’s arrival time
    LocalTime      rxSnapShot1;                        // Past frame’s arrival time
    Precedence     rxPrecedence;                       // Station’s precedence
    uint8_t        rxFrameCount;                       // Clock-master frameCount
} PortDataClock;

typedef struct                                         // Port entity state
{
    uint64_t       macAddress;                         // MAC address of the port
    uint8_t        portID;                             // Destinctive port identifier
    BigNumber      txPrecedence;                       // Grand-master preference
    uint8_t        txHopCount;                         // Next hop-count value
    LocalTime      txTestTimer;                        // Relay-frame received time
    LocalTime      txThisTock;                         // Relay-frame tock-time
```

```
    LocalTime      txThatTock;                                       // Relay-frame tock-time
    LocalTime      txTockTime;                                       // Clock-master’s tockTime
    LocalTime      txPastTime;                                       // Back-interpolation time
    BigNumber      txPreference;                                     // Grand-master preference
    LocalTime      txGrandRate0;                                     // Recent grandTime rating
    LocalTime      txGrandRate1;                                     // Remote grandTime rating
    LocalTime      txErrorRate0;                                     // Recent errorTime rating
    LocalTime      txErrorRate1;                                     // Remote errotTime rating
    GrandTime      txGrandTime0;                                     // Recent grandTime endpoint
    GrandTime      txGrandTime1;                                     // Remote grandTime midpoint
    LocalTime      txLocalTime0;                                     // Recent localTime endpoint
    LocalTime      txLocalTime1;                                     // Remote localTime midpoint
    LocalTime      txErrorTime0;                                     // Recent errorTime endpoint
    LocalTime      txErrorTime1;                                     // Remote errorTime midpoint

    LocalTime      txSnapShot;                                       // Transmit frame snapshot
    uint8_t        txFrameCount;                                     // The timeSync frame count.
    uint8_t        txSnapCount;                                      // The indication’s frameCount
    uint8_t        rxSnapCount;                                      // The indication’s frameCount
    uint8_t        rxFrameCount;                                     // The timeSync’s frameCount
    LocalTime      rxSnapShot0;                                      // This frame’s arrival time
    LocalTime      rxSnapShot1;                                      // Past frame’s arrival time
    LocalTime      rxThisTxTime;                                     // Frame transmission time
    LocalTime      rxThisRxTime;                                     // Frame reception time
    LocalTime      rxThisTime0;                                      // Same as rxSnapShot[n-2]
    LocalTime      rxThatTime0;                                      // Same as frame.localTime[n-2]
    LocalTime      rxThisTime1;                                      // Same as rxSnapShot[n-1]
    LocalTime      rxThatTime1;                                      // Same as frame.localTime[n-1]
    uint64_t       rxRated;                                          // Rate difference from neighbor
} PortDataDuplex;

typedef struct                                                       // Port entity state
{
    uint64_t       macAddress;                                       // MAC address of the port
    uint8_t        portID;                                           // Destinctive port identifier
    BigNumber      txPrecedence;                                     // Grand-master preference
    uint8_t        txHopCount;                                       // Next hop-count value
    LocalTime      txTestTimer;                                      // Relay-frame received time
    LocalTime      txThisTock;                                       // Relay-frame tock-time
    LocalTime      txThatTock;                                       // Relay-frame tock-time
    LocalTime      txTockTime;                                       // Clock-master’s tockTime
    LocalTime      txPastTime;                                       // Back-interpolation time
    BigNumber      txPreference;                                     // Grand-master preference
    LocalTime      txGrandRate0;                                     // Recent grandTime rating
    LocalTime      txGrandRate1;                                     // Remote grandTime rating
    LocalTime      txErrorRate0;                                     // Recent errorTime rating
    LocalTime      txErrorRate1;                                     // Remote errotTime rating
    GrandTime      txGrandTime0;                                     // Recent grandTime endpoint
    GrandTime      txGrandTime1;                                     // Remote grandTime midpoint
    LocalTime      txLocalTime0;                                     // Recent localTime endpoint
    LocalTime      txLocalTime1;                                     // Remote localTime midpoint
    LocalTime      txErrorTime0;                                     // Recent errorTime endpoint
    LocalTime      txErrorTime1;                                     // Remote errorTime midpoint

    LocalTime      txLocalTime;                                      // Normalized standard time
```

```
    PonTime         txPonTime;                                  // Media-dependent time
} PortDataPon;

typedef struct                                                  // Port entity state
{
    uint64_t        macAddress;                                 // MAC address of the port
    uint8_t         portID;                                     // Destinctive port identifier
    BigNumber       txPrecedence;                               // Grand-master preference
    uint8_t         txHopCount;                                 // Next hop-count value
    LocalTime       txTestTimer;                                // Relay-frame received time
    LocalTime       txThisTock;                                 // Relay-frame tock-time
    LocalTime       txThatTock;                                 // Relay-frame tock-time
    LocalTime       txTockTime;                                 // Clock-master's tockTime
    LocalTime       txPastTime;                                 // Back-interpolation time
    BigNumber       txPreference;                               // Grand-master preference
    LocalTime       txGrandRate0;                               // Recent grandTime rating
    LocalTime       txGrandRate1;                               // Remote grandTime rating
    LocalTime       txErrorRate0;                               // Recent errorTime rating
    LocalTime       txErrorRate1;                               // Remote errotTime rating
    GrandTime       txGrandTime0;                               // Recent grandTime endpoint
    GrandTime       txGrandTime1;                               // Remote grandTime midpoint
    LocalTime       txLocalTime0;                               // Recent localTime endpoint
    LocalTime       txLocalTime1;                               // Remote localTime midpoint
    LocalTime       txErrorTime0;                               // Recent errorTime endpoint
    LocalTime       txErrorTime1;                               // Remote errorTime midpoint

    RadioTime       txSnapShot1;                                // Saved ticksTime1
    RadioTime       txRoundTrip;                                // Saved ticksTime4-ticksTime1
    RadioTime       rxTurnRound;                                // Turn-round delay times
    RadioTime       txSnapShot4;                                // Saved ticksTime4
    RadioTime       rxRoundTrip;                                // Saved ticksTime4-ticksTime1
} PortDataRadio;

// Basic interface routines
Boolean         TimeSyncRxClockA(PortDataClock *, ClockInfoReq);    // Check frame's validity
TimeSyncRelay   TimeSyncRxClockB(PortDataClock *, ClockInfoReq);    // Generate MAC-relay frame
ClockInfoRes    TimeSyncTxClock(PortDataClock *, uint8_t);          // Clock-slave updates

Boolean         RelayToState(PortData *, TimeSyncRelay);            // Standard interpolation
Boolean         PreferenceBetter(GrandSyncInfo *, TimeSyncInfo);    // Detects a better preference
void            PreferenceTimeout(PortData *pPtr);                  // Sets precedence to worst

void            TimeSyncRxDuplexA(PortDataDuplex *, DuplexRxInfo);
Boolean         TimeSyncRxDuplexB(PortDataDuplex *, TimeSyncDuplex);
Boolean         TimeSyncRxDuplexC(PortDataDuplex *, TimeSyncDuplex);
void            TimeSyncRxDuplexD(PortDataDuplex *, TimeSyncDuplex);
TimeSyncRelay   TimeSyncRxDuplexE(PortDataDuplex *, TimeSyncDuplex);
void            TimeSyncTxDuplex(PortDataDuplex *, TimeSyncDuplex *);
void            SetDuplexFrame(PortData *, TimeSyncDuplex *);

void            TimeSyncRxRadio1Indicate(PortDataRadio *, RadioInfo1Ind);
TimeSyncRelay   TimeSyncRxRadio2Indicate(PortDataRadio *, RadioInfo2Req);
void            TimeSyncTxRadio1Confirm(PortDataRadio *, RadioInfo1Con);
RadioInfo2Req   TimeSyncTxRadio2Request(PortDataRadio *);
```

```
TimeSyncRelay  TimeSyncRxPon(PortDataPon *, TimeSyncPon);                  // For Ethernet-PON
TimeSyncPon    TimeSyncTxPon(PortDataPon *);                               //  "
void           SetPonFrame(PortData *, TimeSyncPon *);                     //  "

GrandTimes     StateToGrand(PortData *, LocalTime);                        // localTime=>grandTime
void           SetRelayFrame(PortData *, TimeSyncRelay *);                 // Set relay'd timeSync
PonTimes       PonLocalTimes(PortDataPon *);                               // Get localTime/ticksTime
RadioTimes    RadioLocalTimes(PortDataRadio *);                       // Get localTime/ticksTime
LocalTime      GetLocalTime(PortData *);                                   // Get localTime


// A minimalist double-width integer library
BigNumber      BigAddition(BigNumber, BigNumber);
int            BigCompare(BigNumber, BigNumber);
BigNumber      BigShift(BigNumber, int8_t);
BigNumber      BigSubtract(BigNumber, BigNumber);
int64_t        MultiplyHi(uint64_t, int32_t);
int64_t        DivideHi(int64_t, int64_t);

// Other routines
Precedence     FieldsToPrecedence(uint8_t, uint8_t, uint16_t, uint8_t, uint64_t);
BigNumber      FrameToValue(uint8_t *, uint16_t, Boolean);
BigNumber      FormPreference(BigNumber, uint8_t, uint8_t);
BigNumber      LongToBig(LocalTime);
Port           PreferenceToPort(Preference);
HopCount       PreferenceToHops(Preference);
void           ValueToFrame(BigNumber, uint8_t *, uint16_t);
GrandTime      LevelToGrand(GrandTime);
GrandTime      GrandToLevel(GrandTime);


// ***************************************************************************************
// Standard routines, called by corresponding state machines.
// ***************************************************************************************

// Sets common state to allow grandTime values to be back-interpolated
//   Arguments:
//     gPtr   - associated state-maintaining data structure
//     rxInfo - MA_SYNC.indication parameters
Boolean
PreferenceBetter(GrandSyncInfo *gPtr, TimeSyncInfo rxInfo)
{
    Preference preferenceNew, preferenceOld;

    assert(gPtr != NULL);
    preferenceNew =
     FormPreference(rxInfo.precedence, rxInfo.hopCount, rxInfo.port);
    preferenceOld =
     FormPreference(gPtr->info.precedence, gPtr->info.hopCount, gPtr->info.port);
    return(BigCompare(preferenceNew,preferenceOld) >= 0);
}


// Sets common state to allow grandTime values to be back-interpolated
//   Arguments:
```

```
//     pPtr    - associated state-maintaining data structure
//     rxFrame - MAC-relay frame contents
Boolean
RelayToState(PortData *pPtr, TimeSyncRelay rxFrame)
{
    TimeSyncRelay *rxPtr;
    Preference sentPreference, bestPreference;
    Precedence precedence;
    GrandTime grandTime;
    LocalTime currentTime, localTime, errorTime, thisDelta0, thisDelta1, myLocalTime;
    LocalTime grandDelta, grandRated, errorRated;
    LocalTime thisTock, thatTock, tockTime;
    uint8_t hopCount, newHops, oldHops, sourcePort;
    Boolean best, none, same;

    assert(pPtr != NULL);
    rxPtr = &rxFrame;

    sourcePort = FieldToUnsigned(rxPtr, sourcePort).lower;                          // Source-port value
    hopCount =   FieldToUnsigned(rxPtr, hopCount).lower;                            // Hop-count parameter
    precedence = FieldToUnsigned(rxPtr, precedence);                                // GM precedence value
    grandTime =  FieldToSigned(rxPtr,   grandTime);                                 // Grand-master time value
    errorTime =  FieldToUnsigned(rxPtr, errorTime).lower;                           // Grand-master error value
    localTime =  FieldToSigned(rxPtr,   localTime).lower;                           // Neighbor-local time value
    thatTock =   FieldToSigned(rxPtr,   tockTxTime).lower;                          // Neighbor-local time value
    currentTime = GetLocalTime((PortData *)pPtr);                                   // Current localTime value

    sentPreference = FormPreference(precedence, hopCount, sourcePort);              // Received port precedence
    bestPreference = pPtr->txPreference;                                            // Previous best precedence
    same = (PreferenceToPort(bestPreference) == sourcePort);                        // This was preferred port
    best = (BigCompare(sentPreference, bestPreference) <= 0) && (hopCount != HOPS); // This port is preferred
    none = (PreferenceToHops(bestPreference) == HOPS);                              // Obsolete hop count
    if (!same && !best && !none)                                                    // Not-higher preference
        return(!TOP);                                                               // updates are ignored

    oldHops = PreferenceToHops(bestPreference);                                     // Previous hopCount value
    pPtr->txPreference = sentPreference;                                            // Update the preference
    newHops = PreferenceToHops(bestPreference);                                     // Updated hopCount value

    pPtr->txTestTimer = myLocalTime;                                                // Timeout reset from now
    if (newHops <= oldHops)                                                         // Normal operation yields
        pPtr->txHopCount = newHops + 1;                                             // hop-count from source
    else                                                                            // Apparent looping forces
        pPtr->txHopCount = MIN(HOPS, newHops + 1 + (HOPS + newHops) / 2);           // accelerated aging
    pPtr->txThatTock = thatTock;                                                    // Neighbor-local time value
    thisTock = pPtr->txThisTock;                                                    // Neighbor-local time value
    pPtr->txTockTime = tockTime = (2 * (thisTock + MIN(thisTock, thatTock)));       // Interpolation-rate update
    pPtr->txPastTime = tockTime - (pPtr->txThisTock + pPtr->txThatTock) / 2;        // Past interpolation delay

    thisDelta0 = (localTime - pPtr->txLocalTime0);
    thisDelta1 = (localTime - pPtr->txLocalTime1);
    if (thisDelta0 > (tockTime - thisTock) && thisDelta1 > 2 * tockTime)            // Minimum sampling interval
    {
        pPtr->txLocalTime1 = pPtr->txLocalTime0;                                    // Saved localTime[n-1]
        pPtr->txGrandTime1 = pPtr->txGrandTime0;                                    // Saved grandTime[n-1]
```

```
        pPtr->txErrorTime1 = pPtr->txErrorTime0;                                          // Saved errorTime[n-1]
        pPtr->txGrandRate1 = pPtr->txGrandRate0;                                          // Saved grandRate[n-1]
        pPtr->txErrorRate1 = pPtr->txErrorRate0;                                          // Saved errorRate[n-1]

        grandDelta = BigSubtract(grandTime, pPtr->txGrandTime0).lower;                    // The grandTime advance
        grandRated = DivideHi(grandDelta, thisDelta0);                                    // Baseline grandRate[n]
        pPtr->txGrandRate0 = CLIP_RATE(grandRated, PPM250);                               // In-bound grandRate[n]
        errorRated = DivideHi(errorTime - pPtr->txErrorTime0, thisDelta0);                // Baseline errorRate[n]
        pPtr->txErrorRate0 = errorRated;                                                  // In-bound errorRate[n]

        pPtr->txGrandTime0 = grandTime;                                                   // Saved grandTime[n]
        pPtr->txLocalTime0 = localTime;                                                   // Saved localTime[n]
        pPtr->txErrorTime0 = errorTime;                                                   // Saved errorTime[n]
    }
    return(TOP);
}

// Checks for standard clock-master sequence-count consistency
//   Arguments:
//     pPtr    - associated state-maintaining data structure
//     infoReq - clock-master inormation with count value
Boolean
TimeSyncRxClockA(PortDataClock *pPtr, ClockInfoReq infoReq)
{
    uint8_t count;

    assert(pPtr != NULL);                                                                 // Code-correctness check
    pPtr->rxSnapShot1 = pPtr->rxSnapShot0;                                                // Save snapshot delayed
    pPtr->rxSnapShot0 = GetLocalTime((PortData *)pPtr);                                   // Snapshot localTime value
    count = (pPtr->rxFrameCount + 1) % COUNT;                                             // Frame count expectation
    pPtr->rxFrameCount = infoReq.frameCount;                                              // update frameCount value
    return(count != infoReq.frameCount);                                                  // Is frameCount consist?
}


// Generates a timeSyncRelay frame, based on clock-master inputs
//   Arguments:
//     pPtr    - associated state-maintaining data structure
//     infoReq - clock-master inormation with count value
TimeSyncRelay
TimeSyncRxClockB(PortDataClock *pPtr, ClockInfoReq infoReq)
{
    TimeSyncRelay *txPtr, result;

    assert(pPtr != NULL);                                                                 // Code-correctness check
    txPtr = &result;                                                                      // Frame storage preparation

    SetRelayFrame((PortData *)pPtr, txPtr);                                               // Standard relay-frame info
    LongToFrame(0,                  txPtr, hopCount);                                     // Clock's GM distance
    BigToFrame(pPtr->rxPrecedence,  txPtr, precedence);                                   // Clock's GM precedence
    BigToFrame(infoReq.grandTime,   txPtr, grandTime);                                    // Clock's GM time
    LongToFrame(0,                  txPtr, errorTime);                                    // Zero errorTime value
    LongToFrame(pPtr->rxSnapShot0,  txPtr, localTime);                                    // Associated localTime
    return(result);
}
```

```
// Generates a clock-master indication, after being triggered
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     infoCount - clock-master request sequence number
ClockInfoRes
TimeSyncTxClock(PortDataClock *pPtr, uint8_t infoCount)
{
    GrandTimes grandTimes;
    ClockInfoRes result;
    LocalTime currentTime;

    assert(pPtr != NULL);                                                          // Code-correctness check
    currentTime = GetLocalTime((PortData *)pPtr);                                  // Snapshot localTime value
    grandTimes = StateToGrand((PortData *)pPtr, currentTime);                      // Interpolated times
    result.infoCount = infoCount;                                                  // Tag from the request
    result.grandTime =                                                             // Combine the grandTime
     BigAddition(grandTimes.grandTime, LongToBig(grandTimes.errorTime));           // and errorTime values
    return(result);                                                                // Return tagged indication
}

// Restores the precedence level after missing grand-master indications
//   Arguments:
//     pPtr      - associated state-maintaining data structure
void
PreferenceTimeout(PortData *pPtr)
{
    assert(pPtr != NULL);                                                          // Code-correctness check
    pPtr->txPreference = FormPreference(pPtr->txPrecedence, 255, 255);             // Worst-case precedence
    pPtr->txTestTimer = GetLocalTime(pPtr);
}

// >>>> THIS CODE IS CURRENTLY STUBBED TO SUPPORT COMPILATION <<<<
// Returns the times associated with this station:
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//   Results:
//     localTime -- normalized 48-bit local-time
LocalTime
GetLocalTime(PortData *pPtr)
{
    LocalTime localTime;

    assert(pPtr != NULL);                                                          // Pointer consistency
    localTime = 0;                                                                 // To-be-customized
    return(localTime);                                                             // Returned value
}

// ***************************************************************************************
// TimeSync specific library, called by state machines.
// ***************************************************************************************
```

```
// Returns the times associated with this station:
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     localTime - station-local time base for returned values
//   Results:
//     grandTime - normalized 80-bit grand-master synchronized
//     errorTime - normalized 40-bit grand-master compensation
GrandTimes
StateToGrand(PortData *pPtr, LocalTime localTime)
{
    GrandTimes grandTimes;
    LocalTime lapseTime, grandRated, errorRated, localDiff, grandDiff, errorHere;

    assert(pPtr != NULL);                                                   // Code-correctness check
    lapseTime = localTime - pPtr->txPastTime;                               // Back-in-time placement
    if (lapseTime < pPtr->txLocalTime1)
    {                                                                       // Before pivot; based
        grandRated = pPtr->txGrandRate1;                                    // on remote grandRate
        errorRated = pPtr->txErrorRate1;                                    // and remote errorRate
    } else {                                                                // After pivot; based
        grandRated = pPtr->txGrandRate0;                                    // on recent grandRate
        errorRated = pPtr->txErrorRate0;                                    // and recent errorRate
    }

    localDiff = lapseTime - pPtr->txLocalTime1;                             // Local time after pivot
    grandDiff = pPtr->txPastTime + MultiplyHi(localDiff, grandRated);       // Grand time after pivot
    grandTimes.grandTime = BigAddition(pPtr->txGrandTime1, LongToBig(grandDiff));        // Interpolated grandTime
    errorHere = pPtr->txErrorTime1 + MultiplyHi(localDiff, errorRated);     // Interpolated errorTime
    grandTimes.errorTime = errorHere + pPtr->txPastTime * (grandRated - ONE);              // Back-in-time errors
    return(grandTimes);                                                     // Return updated times
}


// Sets the common information associated with MAC-relay frames:
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     txPtr     - pointer to associated MAC-relay frame
//   Results:
//     properly initialized values
void
SetRelayFrame(PortData *pPtr, TimeSyncRelay *txPtr)
{
    LongToFrame(NEIGHBOR,         txPtr, da);                               // Neighbor multicast address
    LongToFrame(pPtr->macAddress, txPtr, sa);                               // This port's MAC address
    LongToFrame(AVB_TYPE,         txPtr, protocolType);                     // The AVB protocol
    LongToFrame(TIME_SYNC,        txPtr, function);                         // The timeSync frame in AVB
    LongToFrame(VERSION_A,        txPtr, version);                          // This version number
    LongToFrame(pPtr->portID,     txPtr, sourcePort);                       // Source-port identifier
    LongToFrame(pPtr->txThisTock, txPtr, tockTxTime);                       // Source sampling rate
}


// ***************************************************************************************
// Ethernet-duplex routines, called by corresponding state machines.
```

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

// Updates state when a receive-PHY indication is observed:
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     rxInfo    - receive-PHY snapshot indication
void
TimeSyncRxDuplexA(PortDataDuplex *pPtr, DuplexRxInfo rxInfo)
{
    assert(pPtr != NULL);
    pPtr->rxSnapShot1 = pPtr->rxSnapShot0;
    pPtr->rxSnapShot0 = rxInfo.localTime;
    pPtr->rxSnapCount = rxInfo.frameCount;
}

// Checks for sequential frameCount consistency errors
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     rxFrame   - received frame with frameCount field
Boolean
TimeSyncRxDuplexB(PortDataDuplex *pPtr, TimeSyncDuplex rxFrame)
{
    TimeSyncDuplex *rxPtr;
    uint8_t frameCount, count;

    assert(pPtr != NULL);
    rxPtr = &rxFrame;
    frameCount = FieldToUnsigned(rxPtr, frameCount).lower;
    count = (pPtr->rxFrameCount + 1) % COUNT;
    pPtr->rxFrameCount = frameCount;
    return(count != frameCount);
}

// Determines when periodic neighbor-rate calibrations are required
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     rxFrame   - received frame with frameCount field
Boolean
TimeSyncRxDuplexC(PortDataDuplex *pPtr, TimeSyncDuplex rxFrame)
{
    TimeSyncDuplex *rxPtr;
    LocalTime thisTime;
    Boolean recent, remote;

    assert(pPtr != NULL);
    rxPtr = &rxFrame;

    thisTime = pPtr->rxSnapShot1;                                            // Frame arrival time
    recent = (thisTime - pPtr->rxThisTime0) >= (3 * pPtr->txTockTime);       // Advanced from recent past
    remote = (thisTime - pPtr->rxThisTime1) >= (8 * pPtr->txTockTime);       // Advanced from remote past
    return(recent && remote);                                                // Rate sampling indication
}

// Performs periodic neighbor-rate calibrations.
//   Arguments:
```

```
//     pPtr      - associated state-maintaining data structure
//     rxFrame   - received frame with frameCount field
void
TimeSyncRxDuplexD(PortDataDuplex *pPtr, TimeSyncDuplex rxFrame)
{
    TimeSyncDuplex *rxPtr;
    LocalTime thisDelta;
    LocalTime thatTime, thatDelta;

    assert(pPtr != NULL);
    rxPtr = &rxFrame;
    thatTime =   FieldToSigned(rxPtr,   localTime).lower;                        // Frame transmission time.

    thisDelta = pPtr->rxSnapShot1  - pPtr->rxThisTime1;                          // Station's timer changes
    thatDelta = thatTime - pPtr->rxThatTime1;                                    // Neighbor's timer changes
    pPtr->rxThisTime1 = pPtr->rxThisTime0;                                       // The local-time snapshot
    pPtr->rxThatTime1 = pPtr->rxThatTime0;                                       // The grand-master snapshot
    pPtr->rxThisTime0 = pPtr->rxSnapShot1;                                       // The local-time snapshot
    pPtr->rxThatTime0 = thatTime;                                                // The grand-master snapshot
    pPtr->rxRated = DivideHi(thatDelta, thisDelta);                              // Neighbor's timer rating
}

// Forms MAC-relay frame; grand-master time compensated for cable delay
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     rxFrame   - received frame with frameCount field
TimeSyncRelay
TimeSyncRxDuplexE(PortDataDuplex *pPtr, TimeSyncDuplex rxFrame)
{
    TimeSyncDuplex *rxPtr;
    TimeSyncRelay relayFrame, *txPtr;
    GrandTime grandTime;
    LocalTime thisTxTime, thatTxTime, thatRxTime, localTime;
    LocalTime roundTrip, turnRound, cableDelay;
    uint8_t hopCount;

    assert(pPtr != NULL);
    rxPtr = &rxFrame;
    txPtr = &relayFrame;

    hopCount =   FieldToUnsigned(rxPtr, hopCount).lower;                         // Hops from the GM station
    grandTime =  FieldToSigned(rxPtr,   grandTime);                              // Grand-master time
    thisTxTime = FieldToSigned(rxPtr,   localTime).lower;                        // Frame transmission time
    thatTxTime = FieldToSigned(rxPtr,   thatTxTime).lower;                       // Opposing transmit time
    thatRxTime = FieldToSigned(rxPtr,   thatRxTime).lower;                       // Opposing received time

    localTime = pPtr->rxSnapShot1;                                               // Frame-arrival time
    roundTrip = (localTime - thatTxTime);                                        // Looped-response delay
    turnRound = (thisTxTime - thatRxTime);                                       // Remote-response delay
    cableDelay = MIN(0, roundTrip - MultiplyHi(turnRound, pPtr->rxRated));       // Computed cable delay
    grandTime = BigAddition(grandTime, LongToBig(cableDelay));                   // Delay compensations

    pPtr->rxThisTxTime = thisTxTime;                                             // This link's sampled values
    pPtr->rxThisRxTime = localTime = pPtr->rxSnapShot1;                          // go-back on opposing link
```

```
    SetRelayFrame((PortData *)pPtr, txPtr);                              // Set basic parameters
    BigToFrame(grandTime,  txPtr, grandTime);                            // Compensated GM time
    LongToFrame(localTime, txPtr, localTime);                            // Observed arrival time
    LongToFrame(hopCount,  txPtr, hopCount);                             // Hops from grand master
    return(relayFrame);
}


// Updates state when a transmit-PHY indication is observed:
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     txInfo    - transmit-PHY snapshot indication
void
TimeSyncTxDuplexA(PortDataDuplex *pPtr, DuplexTxInfo txInfo)
{
    assert(pPtr != NULL);
    pPtr->txSnapShot =  txInfo.localTime;
    pPtr->txSnapCount = txInfo.frameCount;
}


// Forms a MAC-level frame for duplex-line transmission
//   Arguments:
//     pPtr        - associated state-maintaining data structure
//   Result:
//     duplexFrame - MAC frame for duplex-link transmission
TimeSyncDuplex
TimeSyncTxDuplexB(PortDataDuplex *pPtr)
{
    TimeSyncDuplex duplexFrame, *txPtr;
    GrandTimes grandTimes;
    uint8_t frameCount;

    assert(pPtr != NULL);                                                // Code-correctness check
    txPtr = &duplexFrame;                                                // Pointer to results
    grandTimes = StateToGrand((PortData *)pPtr, pPtr->txSnapShot);       // Interpolated times
    pPtr->txFrameCount = (frameCount = (pPtr->txSnapCount + 1) % COUNT); // Increment frameCount

    SetDuplexFrame((PortData *)pPtr, txPtr);                             // Duplex Ethernet frame
    LongToFrame(pPtr->txHopCount,    txPtr, hopCount);                   // The ~GM distance.
    LongToFrame(frameCount,          txPtr, frameCount);                 // Source-port identifier
    BigToFrame(grandTimes.grandTime,  txPtr, grandTime);                 // grandTime at txSnapShot
    LongToFrame(grandTimes.errorTime, txPtr, errorTime);                 // Next errorTime value
    LongToFrame(pPtr->txSnapShot,    txPtr, localTime);                  // Transmitted frame time
    LongToFrame(pPtr->rxThisTxTime,  txPtr, thatTxTime);                 // Opposing transmit time
    LongToFrame(pPtr->rxThisRxTime,  txPtr, thatRxTime);                 // Opposing received time
    return(duplexFrame);
}


// Sets the common information associated with MAC-relay frames:
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     txPtr     - pointer to associated duplex-link frame
//   Results:
```

```
//     properly initialized values
void
SetDuplexFrame(PortData *pPtr, TimeSyncDuplex *txPtr)
{
    LongToFrame(NEIGHBOR,         txPtr, da);                          // Neighbor multicast address
    LongToFrame(pPtr->macAddress, txPtr, sa);                          // This port's MAC address
    LongToFrame(AVB_TYPE,         txPtr, protocolType);                // The AVB protocol
    LongToFrame(TIME_SYNC,        txPtr, function);                    // The timeSync frame in AVB
    LongToFrame(VERSION_A,        txPtr, version);                     // This version number
}


// ***********************************************************************************
// Wireless 802.11v wireless routines, called by corresponding state machines.
// ***********************************************************************************

// Updates state when a MLME_PRESENCE_REQUEST.indication is received.
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     info1Ind  - indication parameters
void
TimeSyncRxRadio1Indicate(PortDataRadio *pPtr, RadioInfo1Ind info1Ind)
{
    assert(pPtr != NULL);
    pPtr->rxTurnRound = info1Ind.ticksTime3 - info1Ind.ticksTime2;
}


// Generates MAC-relay frames based on MLME_PRESENCE_RESPONSE.indication parameters
//   Arguments:
//     pPtr     - associated state-maintaining data structure
//     info2Req - information supplied by the service interface
//   Result     - a timedSync frame destined for the MAC relay
TimeSyncRelay
TimeSyncRxRadio2Indicate(PortDataRadio *pPtr, RadioInfo2Req info2Req)
{
    TimeSyncRelay result, *txPtr;
    GrandTime grandTime;
    RadioTimes localTimes;
    LocalTime twice, moved;

    assert(pPtr != NULL);
    txPtr = &result;

    localTimes = RadioLocalTimes(pPtr);                                // Station local times
    twice = info2Req.roundTrip - pPtr->rxTurnRound;                    // Cable delay ticks
    moved = localTimes.radioTime - info2Req.ticksTime4;                // Elapsed time
    grandTime = BigAddition(LevelToGrand(info2Req.levelTime),          // Grand-master time
     LongToBig(MultiplyHi((twice/2) + moved, RADIO_TIME)));

    SetRelayFrame((PortData *)pPtr, txPtr);                            // Set basic parameters
    BigToFrame(grandTime,            txPtr, grandTime);                // Passing GM time.
    LongToFrame(localTimes.localTime, txPtr, localTime);               // Observed rx-snapshot time.
    return(result);
}
```

```
// Generates parameters for MLME_PRESENCE_REQUEST.confirm.
//   Arguments:
//     pPtr       - associated state-maintaining data structure
//     rxInfo1Req - returned from MLME_PRESENCE_REQUEST.confirm
void
TimeSyncTxRadio1Confirm(PortDataRadio *pPtr, RadioInfo1Con info1Con)
{
    assert(pPtr != NULL);
    pPtr->txSnapShot1 = info1Con.ticksTime1;
    pPtr->txSnapShot4 = info1Con.ticksTime4;
}

// Generates parameters for MLME_PRESENCE_RESPONSE.request.
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//   Result:
//     time4     - the requester’s concluding time snapshot
//     time5     - the observation interval: time4-time1
//     radioTime - A re-encoded version of grandTime
RadioInfo2Req
TimeSyncTxRadio2Request(PortDataRadio *pPtr)
{
    RadioInfo2Req result;
    GrandTimes grandTimes;
    RadioTimes localTimes;
    LocalTime localTime;
    RadioTime lapseTime;

    assert(pPtr != NULL);                                                       // Code-correctness check
    localTimes = RadioLocalTimes(pPtr);
    lapseTime = localTimes.radioTime - pPtr->txSnapShot4;                       // Elapsed time
    localTime = localTimes.localTime - MultiplyHi(lapseTime, RADIO_TIME);       // Extrapolate localTime
    grandTimes = StateToGrand((PortData *)pPtr, localTime);

    result.ticksTime4 = pPtr->txSnapShot4;                                      // Snapshot time transfer
    result.roundTrip = pPtr->txRoundTrip;                                       // Snapshot diff transfer
    result.levelTime = GrandToLevel(grandTimes.grandTime);                      // Grand-master radio time
    result.errorTime = grandTimes.errorTime;                                    // Grand-master error time
    result.precedence = pPtr->txPrecedence;                                     // Grand-master error time
    result.hopCount = pPtr->txHopCount;                                         // Grand-master error time
    return(result);
}

// Should return the times associated with this station:
//  localTime -- normalized 48-bit local-time
//  radioTime -- media-dependent 32-bit time
RadioTimes
RadioLocalTimes(PortDataRadio *pPtr)
{
    RadioTimes localTimes;

    assert(pPtr != NULL);
    localTimes.localTime = localTimes.radioTime = 0;
```

```
    return(localTimes);
}



// ************************************************************************************
// Ethernet-PON routines, called by corresponding state machines.
// ************************************************************************************

// Forms MAC-relay frame; localTime compensated for transmission delay
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     rxFrame   - received frame with frameCount field
TimeSyncRelay
TimeSyncRxPon(PortDataPon *pPtr, TimeSyncPon rxFrame)
{
    TimeSyncPon *rxPtr;
    TimeSyncRelay result, *txPtr;
    GrandTime grandTime, errorTime;
    LocalTime localTime, lapseTime;
    PonTimes ponTimes;
    PonTime ponTime;
    HopCount hopCount;

    assert(pPtr != NULL);
    rxPtr = &rxFrame;
    txPtr = &result;
    result = *((TimeSyncRelay *)&rxFrame);

    ponTimes = PonLocalTimes(pPtr);                                                       // Station local times
    grandTime = FieldToSigned(rxPtr, grandTime);                                          // Grand-master time
    errorTime = FieldToSigned(rxPtr, errorTime);                                          // Error in grand-master time
    ponTime   = FieldToSigned(rxPtr, ticksTime).lower;                                    // Frame transmission time
    hopCount  = FieldToUnsigned(rxPtr, hopCount).lower;                                   // Distance to grand-master
    lapseTime = ponTimes.ponTime - ponTime;                                               // Passed-time compensation
    localTime = ponTimes.localTime - MultiplyHi(lapseTime, PON_TIME);                     // Passed-time compensation

    SetRelayFrame((PortData *)pPtr, txPtr);                                                // Set basic parameters
    BigToFrame(grandTime,  txPtr, grandTime);                                              // Passing GM time
    BigToFrame(errorTime,  txPtr, errorTime);                                              // Passing GM error
    LongToFrame(localTime, txPtr, localTime);                                              // Observed rx-snapshot time
    LongToFrame(hopCount,  txPtr, hopCount);                                               // Distance to grand-master
    return(result);
}



// Forms a MAC-level frame for Ethernet-PON transmission
//   Arguments:
//     pPtr     - associated state-maintaining data structure
//   Result:
//     ponFrame - MAC frame for Ethernet-PON transmission
TimeSyncPon
TimeSyncTxPon(PortDataPon *pPtr)
{
```

```
    TimeSyncPon *txPtr, ponFrame;
    GrandTimes grandTimes;
    PonTimes localTimes;

    assert(pPtr != NULL && txPtr != NULL);                                      // Code-correctness check
    txPtr = &ponFrame;
    localTimes = PonLocalTimes(pPtr);                                           // Get localTime values
    grandTimes = StateToGrand((PortData *)pPtr, localTimes.localTime);          // Get grandTime values

    SetPonFrame((PortData *)pPtr,     txPtr);                                   // Base EthernetPon frame
    LongToFrame(pPtr->txHopCount,     txPtr, hopCount);                         // The GM distance.
    BigToFrame(grandTimes.grandTime,  txPtr, grandTime);                        // grandTime at txSnapShot
    LongToFrame(grandTimes.errorTime, txPtr, errorTime);                        // Next errorTime value
    LongToFrame(localTimes.ponTime,   txPtr, ticksTime);                        // Transmitted frame time
    return(ponFrame);
}


// Sets the common information associated with Ethernet-PON frames:
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     txPtr     - pointer to associated Ethernet-PON frame
//   Results:
//     properly initialized values
void
SetPonFrame(PortData *pPtr, TimeSyncPon *txPtr)
{
    LongToFrame(NEIGHBOR,         txPtr, da);                                   // Neighbor multicast address
    LongToFrame(pPtr->macAddress, txPtr, sa);                                   // This port's MAC address
    LongToFrame(AVB_TYPE,         txPtr, protocolType);                         // The AVB protocol
    LongToFrame(TIME_SYNC,        txPtr, function);                             // The timeSync frame in AVB
    LongToFrame(VERSION_A,        txPtr, version);                              // This version number
}


// Should return the times associated with this station:
//  localTime -- normalized 48-bit local-time
//  ticksTime -- media-dependent 32-bit time
PonTimes
PonLocalTimes(PortDataPon *pPtr)
{
    PonTimes localTimes;

    assert(pPtr != NULL);
    localTimes.localTime = localTimes.ponTime = 0;
    return(localTimes);
}


// ***************************************************************************************
// Alignment and endian-order independent frame-extraction routines.
// ***************************************************************************************


// Extracts & sign-extends a specified field to a 16-byte result.
```

```
//   fieldPtr - the starting address for the source field
//   length   - the length of the source field
//   sign     - differentiates between unsigned and signed fields:
//                0 - an unsigned field
//                1 - a signed field
BigNumber                                                                          // Extracts field of frame,
FrameToValue(uint8_t *fieldPtr, uint16_t length, Boolean sign)                     // as signed or unsigned.
{
    BigNumber result;                                                              // The 128-bit signed result.
    uint8_t *cPtr;
    int i;

    cPtr = fieldPtr;                                                               // Start from first byte
    if (sign && (int8_t)(cPtr[0]) < 0)                                             // Check for sign extension
        result.upper = result.lower = (int64_t)-1;                                 // 1's extended if negative
    else                                                                           // otherwise,
        result.upper = result.lower = 0;                                           // 0's extended.

    for (i = length - 1; i >= 0; i -= 1, cPtr += 1)                                // Step through bytes
        if (length >= 8)
            result.upper |= *cPtr << (8 * ( i % 8));                               // First bytes into upper
        else
            result.lower |= *cPtr << (8 * ( i % 8));                               // Final byes into lower
    return(result);                                                                // Return BigNumber result
}


// Copies the less-significant portion of a 16-byte argument to a specified field location.
//   value -    a 16-byte value, consisting of upper and lower components
//   fieldPtr - the starting address for the copied field
//   length   - the length of the copied field
void                                                                               // Place fields into frame,
ValueToFrame(BigNumber value, uint8_t *fieldPtr, uint16_t length)                  // signed properties ignored.
{
    int i;
    uint8_t *cPtr;

    assert(fieldPtr != NULL);
    cPtr = fieldPtr;                                                               // First byte location
    for (i = length - 1; i >= 0; i -= 1, cPtr += 1)                                // Step through the bytes
        if (length >= 8)
            *cPtr = value.upper >> (8 * ( i % 8));                                 // First bytes from upper
        else                                                                       // as well as the
            *cPtr = value.lower >> (8 * ( i % 8));                                 // final bytes from lower.
}


// ************************************************************************************
// Supporting library-like routines.
// ************************************************************************************


// Converts a seconds:nanoseconds value to seconds.fraction scaled integer
//   Arguments:
```

```
//     pPtr      - associated state-maintaining data structure
//     value     - formatted seconds:nanoseconds grand-master time
//   Results:
//     grandTime - formatted seconds.fraction scaled integer
GrandTime
LevelToGrand(GrandTime value)
{
    GrandTime seconds, grandTime;
    LocalTime lessor, partial;

    lessor = value.lower & (uint64_t)0XFFFFFFFF;
    partial = DivideHi(MIN(1000000000, lessor), 1000000000);
    seconds = BigShift(BigShift(value, 32), -40);
    grandTime = BigAddition(seconds, LongToBig(partial));
    return(grandTime);
}


// Converts a seconds.fraction scaled integer to seconds:nanoseconds value
//   Arguments:
//     pPtr      - associated state-maintaining data structure
//     value     - Formatted seconds.fraction grand-master time
//   Results:
//     grandTime - Formatted seconds:nanoseconds grand-master time
GrandTime
GrandToLevel(GrandTime value)
{
    GrandTime seconds, result;
    LocalTime lessor, partial;

    lessor  = value.lower & (((uint64_t)1 << 48) - 1);
    partial = MultiplyHi(lessor, 1000000000);
    seconds = BigShift(BigShift(value, 40), -32);
    result  = BigAddition(seconds, LongToBig(partial));
    return(result);
}


// Extracts the hopCount field from within the 16-byte preference:
//   preference - a 16-byte preference value, consisting of upper and lower components
//   result -     the 8-bit hopCount-field value
HopCount
PreferenceToHops(BigNumber preference)
{
    HopCount result;

    result = (preference.lower >> BITS(Port)) & MASK(BITS(HopCount));
    return(result);
}


// Extracts the port field from within the 16-byte preference:
//   preference - a 16-byte preference value, consisting of upper and lower components
//   result -     the 8-bit port-field value
Port
```

```
PreferenceToPort(Precedence preference)
{
    Port result;

    result = (preference.lower & MASK(BITS(Port)));
    return(result);
}


// Concatensated subfields into a larger precedence field
//   Arguments:
//     priority1  - user-assigned more-significant priority field
//     class      - characteristic of the clock
//     variance   - characteristic of the clock quality
//     priority2  - user-assigne less-significant priority field
//     clockID    - 64-bit EUI-64 (or near equivalent) field
//   Result:
//     precedence - the concatenated arguments
Precedence
FieldsToPrecedence(uint8_t priority1, Class class, Variance variance, uint8_t priority2, uint64_t clockID)
{
    BigNumber result;
    uint32_t fields;

    fields = (priority1 & MASK(4));
    fields <<= BITS(class);
    fields  |= class & MASK(BITS(class));
    fields <<= BITS(variance);
    fields  |= variance & MASK(BITS(variance));
    fields <<= 4;
    fields  |= priority2 & MASK(4);
    result.upper = fields;
    result.lower = clockID;
    return(result);
}


// Converts between integer precisions:
//   number - a signed 64-bit integer
//   result - a signed 128-bit integer,
//            consisting of upper and lower parts
BigNumber
LongToBig(int64_t number)
{
    BigNumber result;

    result.lower = number;                                              // LSBs are the same
    result.upper = 0;                                                   // Zero sign-extended
    if (number < 0)                                                     // Negative numbers are
        result.upper -= 1;                                              // ones sign-extended
    return(result);                                                     // Returned 128-bit result
}


// Forms an 16-byte precedence from smaller components:
```

```
//   precedence - a 14-byte grand-master weighting (lowest is best)
//   hopCount   - the distance from the grand-master, in station-to-station hops
//   port       - the port that sourced the preference value
BigNumber
FormPreference(BigNumber precedence, HopCount hopCount, Port port)
{
    BigNumber result;

    result = BigShift(precedence, -8 * (int)(sizeof(HopCount) + sizeof(Port)) );         // Left-shift precedence
    result.lower |= (hopCount << (8 * sizeof(Port))) | port;                              // Merge in hopCount&port
    return(result);                                                                       // Return the result
}


// Forms a 16-byte arithmetic sum of two values:
//   a -      A 16-byte argument, with upper and lower components.
//   b -      A 16-byte argument, with upper and lower components.
//   result - A 16-byte summation: a+b.
BigNumber
BigAddition(BigNumber a, BigNumber b)
{
    BigNumber result;
    uint32_t sum, carry;

    result.lower = sum = a.lower + b.lower;                                               // Addition of the LSBs
    carry = (sum < a.lower) ? 1 : 0;                                                      // Determine the carry.
    result.upper += a.upper + b.upper + carry;                                            // Addition of the MSBs
    return(result);
}


// Forms a 16-byte arithmetic difference of two values:
//   a -      A 16-byte argument, with upper and lower components.
//   b -      A 16-byte argument, with upper and lower components.
//   result - A 16-byte difference: a-b.
BigNumber
BigSubtract(BigNumber a, BigNumber b)
{
    BigNumber result;
    uint32_t sum, borrow;

    result.upper = sum = a.lower - b.lower;                                               // Addition of the LSBs
    borrow = (sum > a.lower) ? 1 : 0;                                                     // Determine the borrow.
    result.upper += a.upper + b.upper - borrow;                                           // Addition of the MSBs
    return(result);
}


// Forms a 16-byte arithmetic difference of two values:
//   a -      A 16-byte argument, with upper and lower components.
//   b -      A 16-byte argument, with upper and lower components.
//   result - The result of a signed arithmetic comparison:
//              1 - Corresponds to: a > b
//              0 - Corresponds to: a == b
//             -1 - Corresponds to: a < b
```

```
int
BigCompare(BigNumber a, BigNumber b)
{
    if (a.upper != b.upper)                                                // More significant compare
        return(a.upper > b.upper ? 1 : -1);
    if (a.lower != b.lower)                                                // Less significant compare
        return(a.lower > b.lower ? 1 : -1);
    return(0);                                                             // Comparison returns equal
}


// Right shifts the 16-byte arguments by a shift-specified amount:
//   a -      A 16-byte argument, with upper and lower components.
//   b -      A signed shift amount.
//   result - The shifted/sign-extended result: a >> b.
BigNumber
BigShift(BigNumber value, int8_t shift)
{
    BigNumber result;
    int8_t rightShift, leftShift;

    if (shift == 0)
        return(value);
    if (shift > 0)
    {
        rightShift = shift;
        if (rightShift >= 64)
        {
            result.lower = (value.upper >> (rightShift % 64));
            result.upper = (value.upper > 0 ? 0 : -1);
        } else {
            result.lower = (value.upper << (64 - rightShift)) | (value.lower >> rightShift);
            result.upper = (value.upper >> rightShift);
        }
    } else {
        leftShift = shift;
        if (leftShift >= 64)
        {
            result.upper = value.lower << (leftShift % 64);
            result.lower = 0;
        } else {
            result.upper = (value.upper << leftShift) | (value.lower >> (64 - leftShift));
            result.lower = (value.lower << leftShift);
        }
    }
    return(result);
}


// Multiplies a signed integer by a signed scaled-integer,
// returning a scaled-integer product:
//   a -      A 64-bit argument.
//   b -      A 64-bit argument.
//   result - The multiplied value: (a * b) >> 40.
```

```
int64_t
MultiplyHi(uint64_t value1, int32_t value2)
{
    int64_t upper, lower;

    upper = (value1 >> 40) * value2;                                    // Add the upper
    lower = ((value1 & (uint64_t)0XFFFFFF) * value2) >> 40;             // to the lower
    return(upper + lower);                                              // for the result.
}


// Divides a signed integer by a signed scaled-integer,
// returning a scaled-integer result:
//   a -      A 64-bit argument, the numerator.
//   b -      A 64-bit argument, the denominator.
//   result - The divided value: (a / b) << 40.
int64_t                                                                 // x = (a << 32)/b, for
DivideHi(int64_t a, int64_t b)                                          // for b < 2**48
{
    int64_t sum, rem;
    Boolean flip;

    flip = ((a ^ b) < 0);                                               // Ensure positive args
    a = (a < 0) ? -a : a;                                               // for all possible
    b = (b < 0) ? -b : b;                                               // argument values.

    sum = a / b;                                                        // The normal divide
    rem = (a % b) << 16;                                                // Prepare the remainder
    sum = (sum << 16) + rem / b;                                        // Scaled by 2**16
    rem = (rem % b) << 16;                                              // Prepare the remainder
    sum = (sum << 16) + rem / b;                                        // Scaled by 2**32
    rem = (rem % b) << 8;                                               // Prepare the remainder
    sum = (sum << 8) + rem / b;                                         // Scaled by 2**40
    return(flip ? -sum : sum);                                          // Correctly signed result
}
```