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

Draft 0.207

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 front-page list of contributors – Updated book for continuous pages (Clause 1 discontinuity fixed) – Miscellaneous editing 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, based on 2005Jun08 teleconference review comments.
0.121	2005Jun24	DVJ	– Extensive cleanup of clock-synchronization protocols, base on 2005Jun22 teleconference review comments.
0.127	2005Jul04	DVJ	– Pacing descriptions greatly enhanced.
0.200	2007Jan23	DVJ	Removal of non time-sync related information. Update based on recent teleconference suggestion (layering), as well as input available from others' drafts.
0.207	2007Feb01	DVJ	Updates based on feedback from Monterey 802.1 meeting. – Common entity terminology updated to avoid MAC-level confusion. – Expansion codes provided after the Ethernet type code. – Additional details of the common-entity services.
—	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

Topics for discussion

Readers are encouraged to provide feedback in all areas, although only the following areas have been identified as specific areas of concern.

a) Layering. Should be reviewed.

TBDs

Further definitions are needed in the following areas:

a) How are leap-seconds handled?

b) How are rate differences distributed? Avoid whiplash?

c) When the grand-master changes, should the new clock transition to it free-run rate instantaneously or migrate there slowly over time?

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.

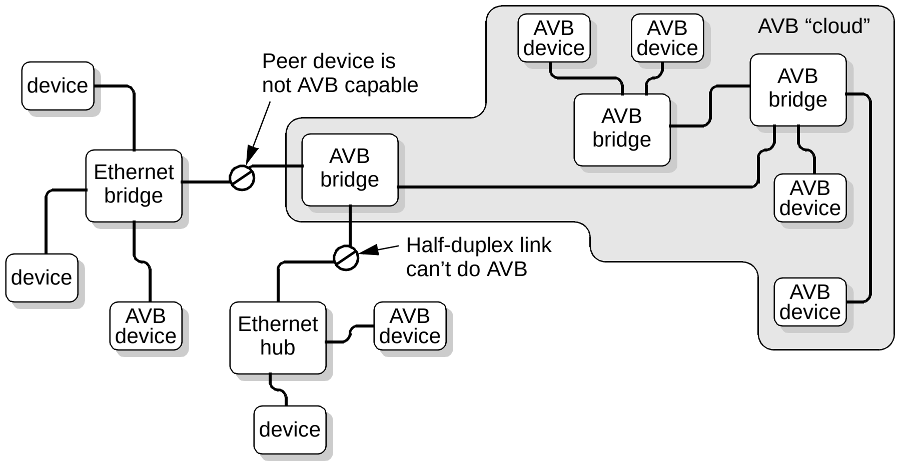


Figure 1.1—Topology and connectivity

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 6: Common entity service and reference model
- Clause 7: Ethernet duplex-cable time synchronization
- 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.

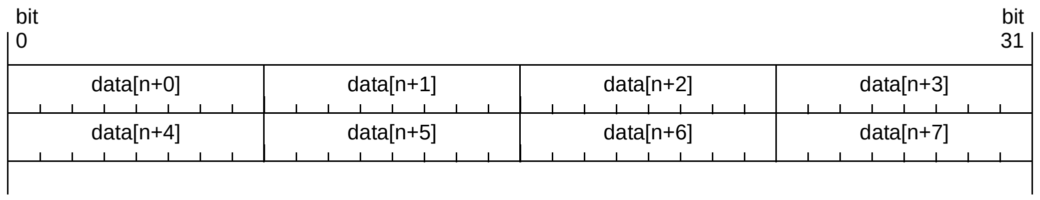


Figure 3.1—Bit numbering and ordering

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.

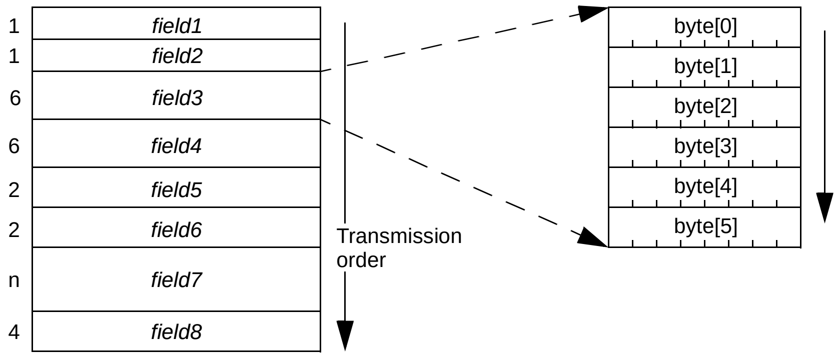


Figure 3.2—Byte sequential field format illustrations

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.

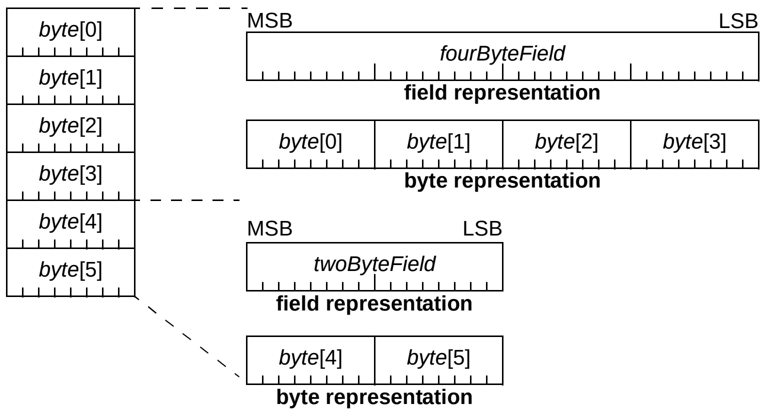


Figure 3.3—Multibyte field illustrations

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.

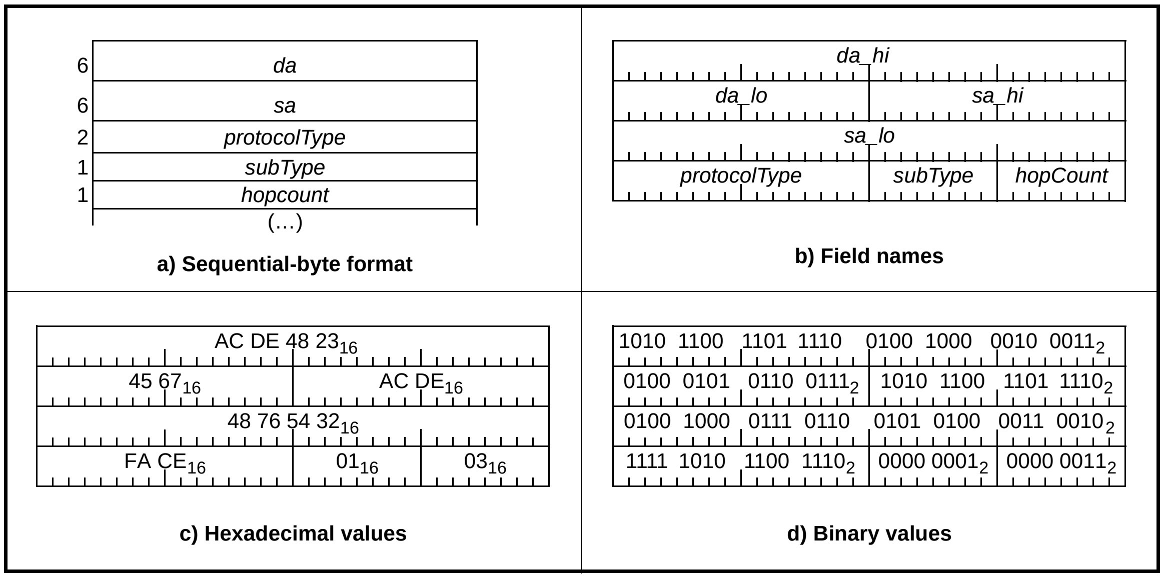


Figure 3.4—Illustration of fairness-frame structure

3.10 MAC address formats

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

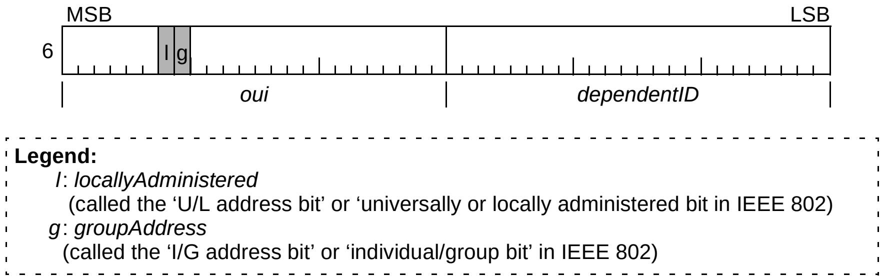

MSB
LSB
6
l
g
oui
dependentID
Legend:
l: localllyAdministered
(called the 'U/L address bit' or 'universally or locally administered bit in IEEE 802)
g: groupAddress
(called the 'I/G address bit' or 'individual/group bit' in IEEE 802)

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.

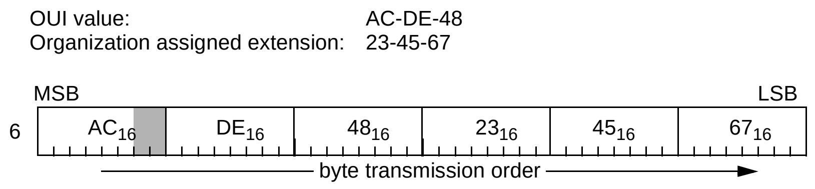


Figure 3.6—48-bit MAC address format

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

NOTE—This clause should be skipped on the first reading (continue with Clause 5).
This text has been lifted from the P802.17 draft standard, which has a relative comprehensive list. Abbreviations/acronyms are expected to be added, revised, and/or deleted as this working paper evolves.

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.

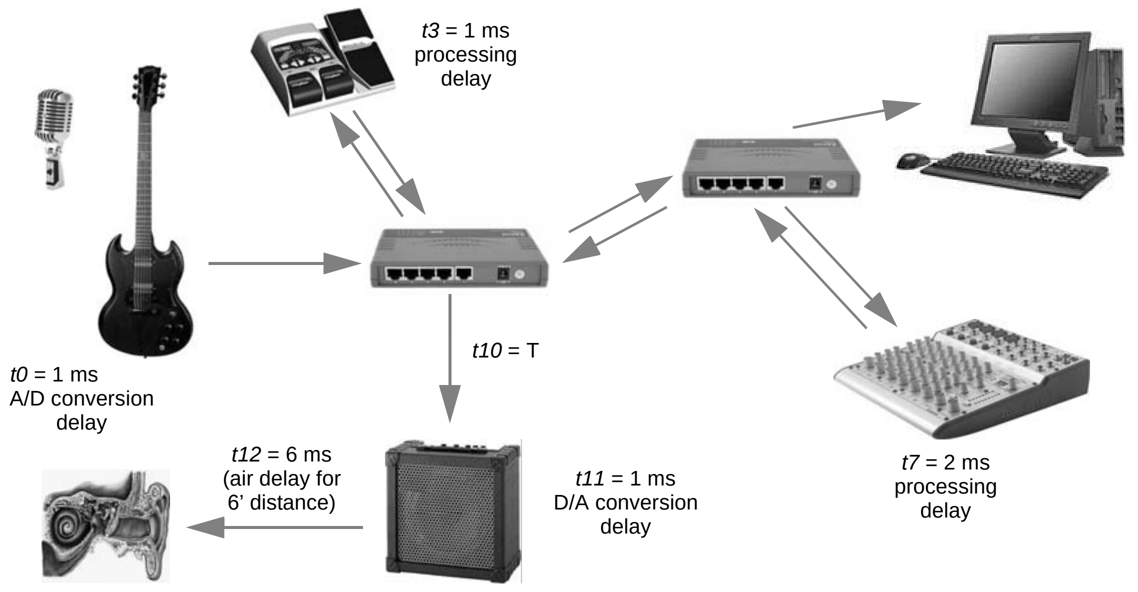


Figure 5.1—Garage jam session

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.

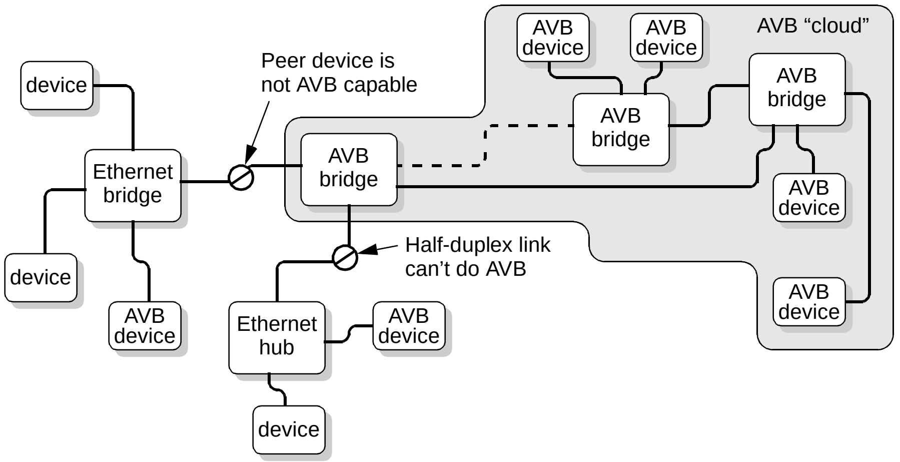


Figure 5.2—Possible looping topology

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 *gmTime* 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 Time-synchronization facilities

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.

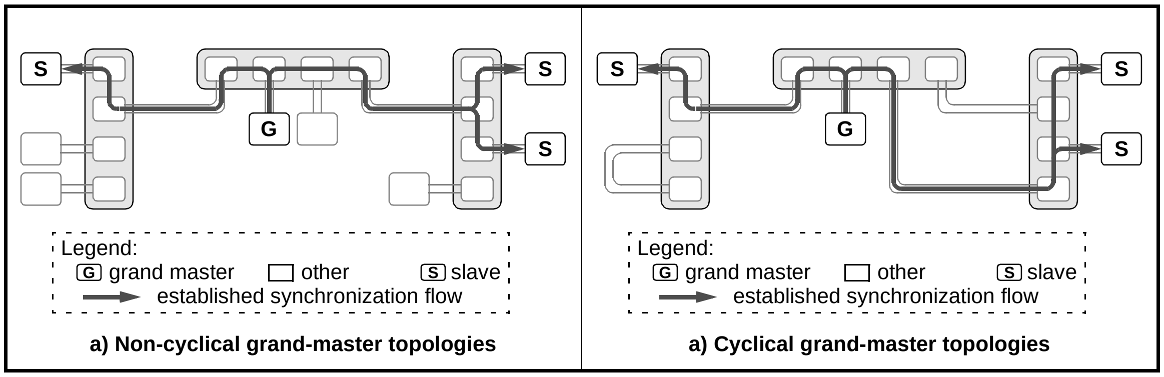

S
S
S
G
S
S
S
G
Legend:
G grand master
other
S slave
established synchronization flow
Legend:
G grand master
other
S slave
established synchronization flow
a) Non-cyclical grand-master topologies
a) Cyclical grand-master topologies

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.

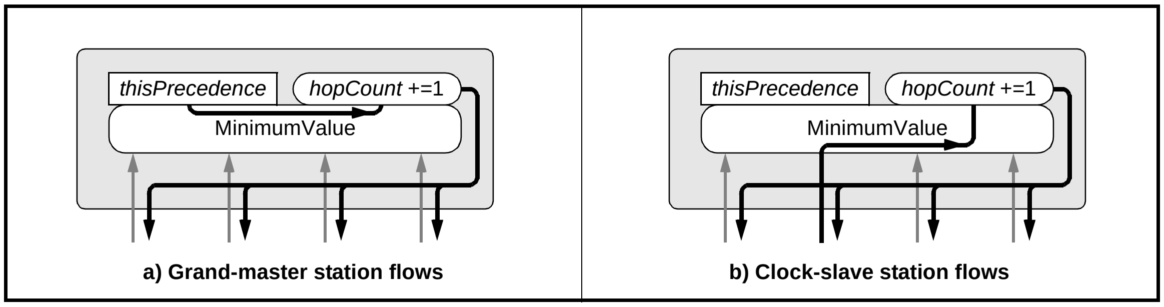
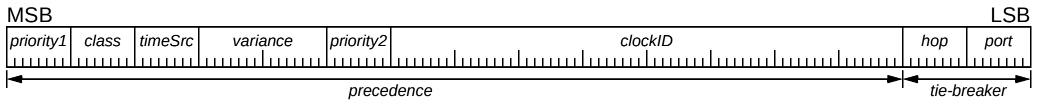


Figure 5.4—Grand-master precedence flows

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.



Legend: timeSrc: timeSource *hop*: *hopCount*

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.

5.3.4 Synchronized-time distribution

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

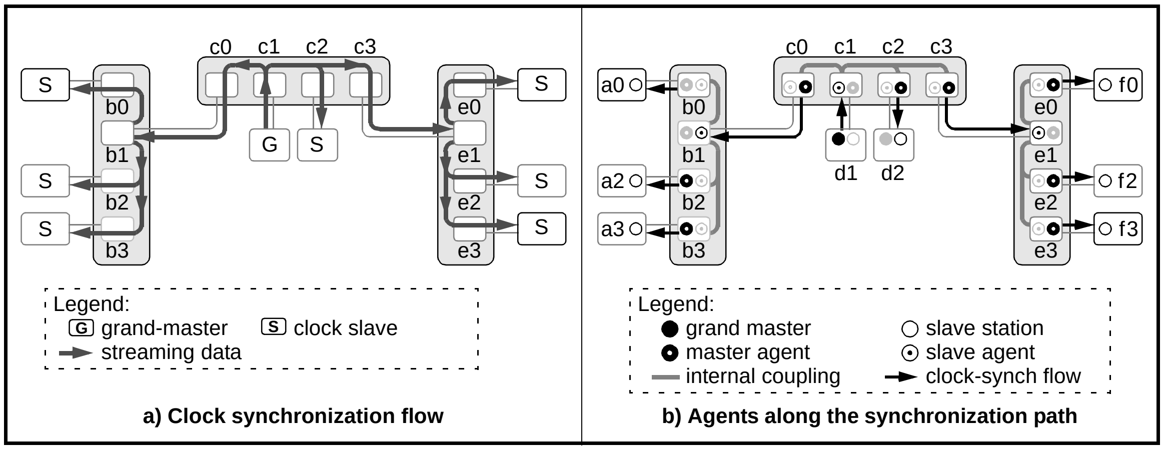


Figure 5.6—Hierarchical flows

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

Time synchronization yields distributed but closely-matched *gmTime* 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 *gmTime* values

5.4 Common-entity service model

The time-synchronization service model assumes the presence of one or more time-synchronized AVB ports communicating with a shared common entity, as illustrated in Figure 5.7. The receive portion of each port provides grand-master precedence information, which assists in the grand-master selection process. The transmit portion of each port provides the common entity with a local time and (in response) receives status containing the grand-master precedence and the client's synchronized version of the grand-master time. 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.

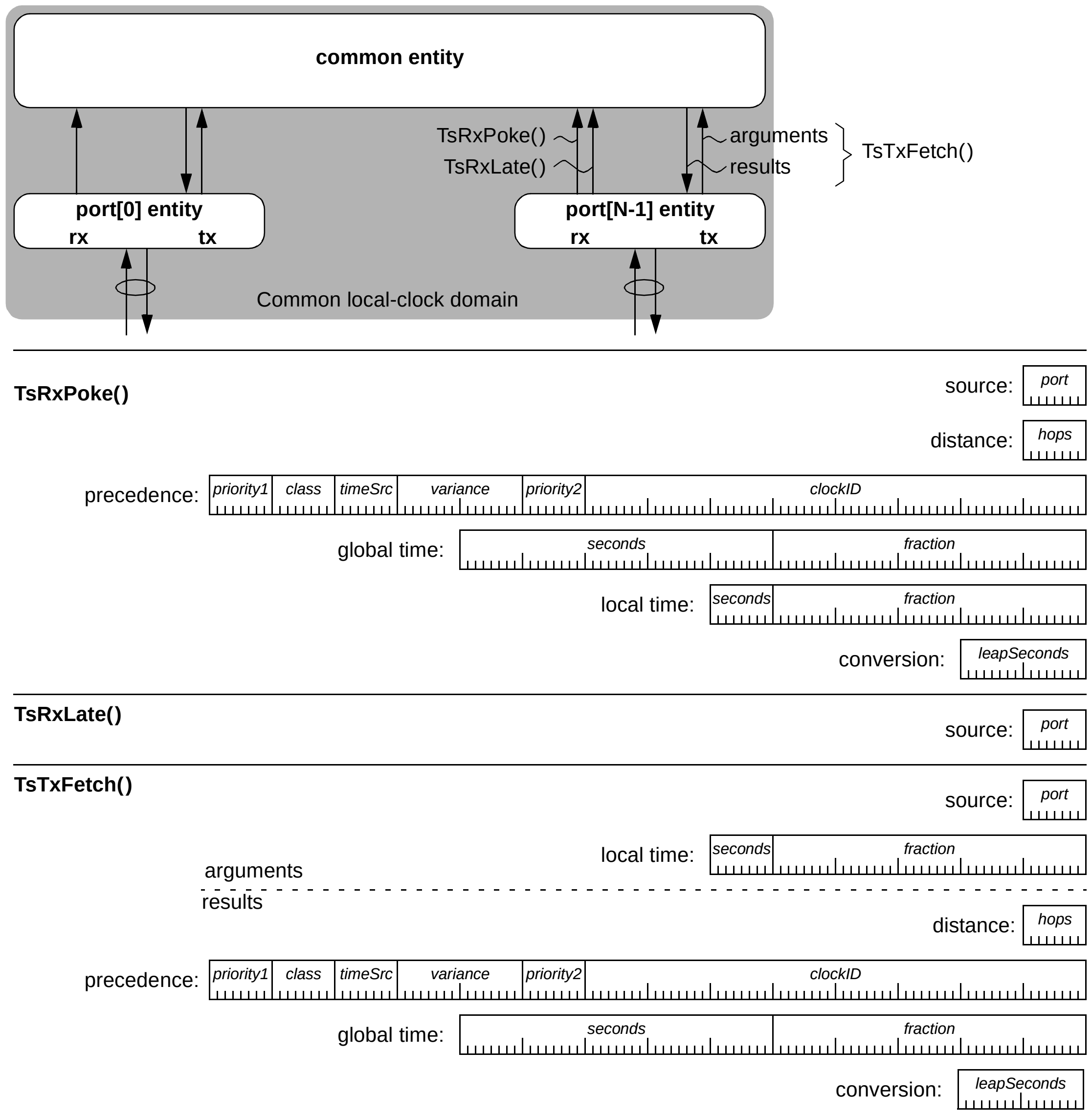


Figure 5.7—AVB common-entity service interface model

5.5 Rate-difference adjustments

If the absence of rate adjustments, significant *gmTime* errors can accumulate between send-period updates, as illustrated on the left side of Figure 5.8. 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.

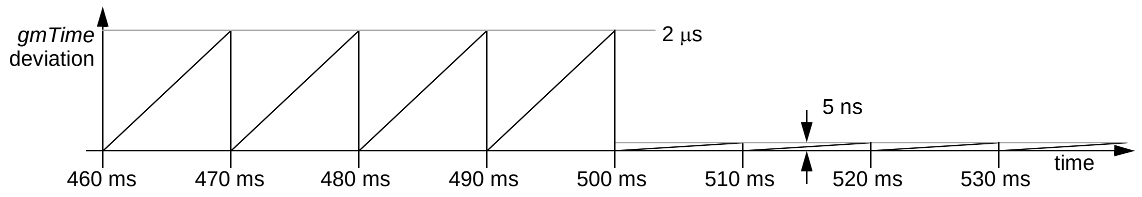


Figure 5.8—Rate-adjustment effects

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

The differences in rates could easily be reduced to less than 1 PPM, assuming a 200 ms measurement interval (based on a 100 ms slow-period interval) and a 100 ns arrival/departure sampling error. A clock-rate adjustment at time 100 ms could thus reduce the clock-drift related errors to less than 5 ns. At this point, the timer-offset measurement errors (not clock-drift induced errors) dominate the clock-synchronization error contributions.

The preceding discussion illustrates the need for timer-rate (as well as timer-offset) adjustments. To avoid PLL-chain whiplash type of effects, the rates are computed as follows:

a) Each clock-slave station computes the rate difference between its local timer and the grand master.
b) When interpolating between known grand-master times, the station uses a scaled local timer, wherein that scaled local timer has been adjusted to compensate for drifts from the grand master.

5.6 Key distinctions from IEEE Std 1588

Although based on the concepts of IEEE Std 1588, this draft is different in multiple ways:

a) All bridges are boundary clocks, since they compensate their received time by pass-through delays, by setting the client time on received frames, then transmitting the current client time when frames are transmitted. There are no transparent (in 1588 terminology) bridges.

b) To simplify computations, time is uniformly represented as a 80-bit scaled signed integer.

c) For Ethernet, a higher update frequency of 100 Hz is assumed. This reduces timeouts for failed grand masters, and worst-case times for clear the network of rogue packets, while also reducing timer-value drifts between updates.

d) For Ethernet, 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. Common entity service and reference model

6.1 Overview

This clause specifies the services provided to the common-entity. The services 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 Overview of common services

The services provided by the common-services sublayer are listed below. For each port, these services are invoked periodically at modest processing rates (100Hz, for Ethernet MACs).

a) The receive port provides grand-master precedence and time, as well as local time, to the client.
b) The transmit port provides the client with a recent-past local time and (in response) the client provides the grand-master precedence and time as status.

6.3 MAC services to the client layer

The services of a layer or sublayer are the set of capabilities that it offers to the next higher (sub)layer. The services specified in this standard are described by abstract service primitives and parameters that characterize each service. This definition of a service is independent of any particular implementation.

The following three service primitives are defined for the client interfaces, and shall be implemented.

— TsRxPoke():	Common entity gets information for grand-master selection and synchronization.
— TsRxLate():	Common entity gets a timeout indication (timeSync frames are absent.)
— TxTxFetch():	Port entity provided with timeSync frame-generation parameters.

6.3.1 TsRxPoke service interface

6.3.1.1 Function

The TsRxPoke primitive provides the common entity with an association of local time and the grand-master time (as received from the closer-to grand-master station), as well as the grand-master precedence and leap-seconds parameters.

6.3.1.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
TsRxPoke
(
    port,
    this_time,
    hop_count,
    gm_precedence,
    gm_time,
    leap_seconds
)
```

The parameters of TsRxPoke( ) are described below.

port
: An 8-bit value that identifies the port that provided the service primitive values.

local_time
: A 32-bit value representing the local-time value associated with the sampled gm_time value.

hop_count
: An 8-bit value representing the distance from the grand-master station, measured in links.

gm_precedence
: Provides the recently observed grand-master predecence, or NULL if such as precedence has not been received within a physical-layer dependent timeout interval. Fields within this component include the following (more-significant fields are listed first):

priority1	A 4-bit user-selectable overriding precedence.
class	An 8-bit identifier that identifies the type of grand-master.
variance	A 16-bit value that identifies the grand-master clock quality.
priority2	A 4-bit user-selectable tie-breaking precedence.
clockID	An 8-bit value that, when prepended with the tech, is globally unique.

gm_time
: Provides the value of the distributed grand-master time, compensated for intermediate delays. Fields within this component include the following (more-significant fields are listed first):

seconds	A signed 48-bit value representing seconds.
fraction	An unsigned 32-bit value representing fractions-of-seconds.

leap_seconds
: A parameter passed from the grand master for the purpose of occassionally jumping ahead seconds.

6.3.1.3 When generated

The TsRxPoke( ) primitive is invoked by the port entity whenever new knowledge of time or the grand-master precedence is generated.

6.3.1.4 Effect of receipt

The receipt of the TsRxPoke( ) supplied information causes the client entity to adjust its image the grand master and (when appropriate) the grand-master time and rate, as specified by the code of Annex F.

6.3.2 TsRxLate service interface

6.3.2.1 Function

The TsRxLate primitive provides the common entity with an indication of a link timeout, corresponding to the continued absence of received timeSync frames.

6.3.2.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
TsRxLate                    // Service primitive
(
    port                    // Argument
)
```

The parameters of TsRxLate( ) are described below.

port
    An 8-bit value that identifies the port that generated the service primitive.

6.3.2.3 When generated

The TsRxLate( ) primitive is invoked by the receive-port entity whenever new knowledge of time or the grand-master precedence is generated.

6.3.2.4 Effect of receipt

The receipt of the TsRxLate( ) supplied timeout-event information that allows the common entity to ignore the inactive port when performing grand-master selection.

6.3.3 TsTxFetch

6.3.3.1 Function

The TsTxFetch primitive provides the port entity with common-entity supplied information necessary for generating the next timeSync frame.

6.3.3.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
    (                               // Returned parameters
        hop_count,
        gm_precedence,
        gm_time,
        leap_seconds
    )
    TsTxFetch                       // Service primitive
    (
        port,                       // Argument parameters
        local_time
    )
```

The argument parameters of TsTxFetch() are described below.

port
: An 8-bit value that identifies the port that provided the service primitive values.

local_time
: Provides a local time value (as a basis for returning the corresponding grand-master time).

The returned parameters for the associated TsTxFetch() service primitive are described below.

hop_count
: An 8-bit value representing the distance from the grand-master station, measured in links.

gm_precedence
: Provides the client’s recently observed best grand-master precedence. See xx for details.

gm_time
: Provides the value of the distributed grand-master time, compensated for intermediate delays. See xx for details.

leap_seconds
: A parameter passed from the grand master for the purpose of occassionally jumping ahead seconds.

6.3.3.3 When generated

The TsTxFetch() primitive is generated by a transmit-port entity when timing information from the common entity is needed (for a duplex Ethernet link, this is shortly after a timeSync packet transmission).

6.3.3.4 Effect of receipt

The receipt of the TsTxFetch() primitive causes the common entity to return the desired result parameters, as specified by the TsTxFetch() routine of Annex F.

7. Ethernet duplex-cable time synchronization

7.1 Design methodology

7.1.1 Assumptions

Support of duplex-link Ethernet is 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) Point-to-point transmit/receive duplex connections are provided.

b) Transmit/receive propagation delays within duplex cables are well matched.

7.1.2 Strategies

Strategies used to meet these objectives include the following:

a) Precision is achieved by calibrating and adjusting *gmTime* 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) Symmetric. Clock-master/clock-slave computations are similar (only slave results are saved).
   2) Periodic. Messages are sent periodically, rather than in timely response to other requests.
   3) Frequent. Frequent (typically 1 kHz) interchanges reduces needs for precise clocks.

c) Balanced functionality.
   1) Low-rate. Complex computations are infrequent and can be readily implemented in firmware.
   2) High-rate. Frequent computations are simple and can be readily implemented in hardware.

7.2 Time-synchronization operation

7.2.1 Periodic packet transmissions

Time-synchronization involves periodic not-necessarily synchronized packet transmissions between adjacent stations, as illustrated in Figure 7.1a. The transmitted frame contains the following information:

select—specifies grand-master precedence
global—an estimation of the grand-master time
local—a sampling of the station-local time
delta—derived parameters from the neighbor, returned in a following cycle.

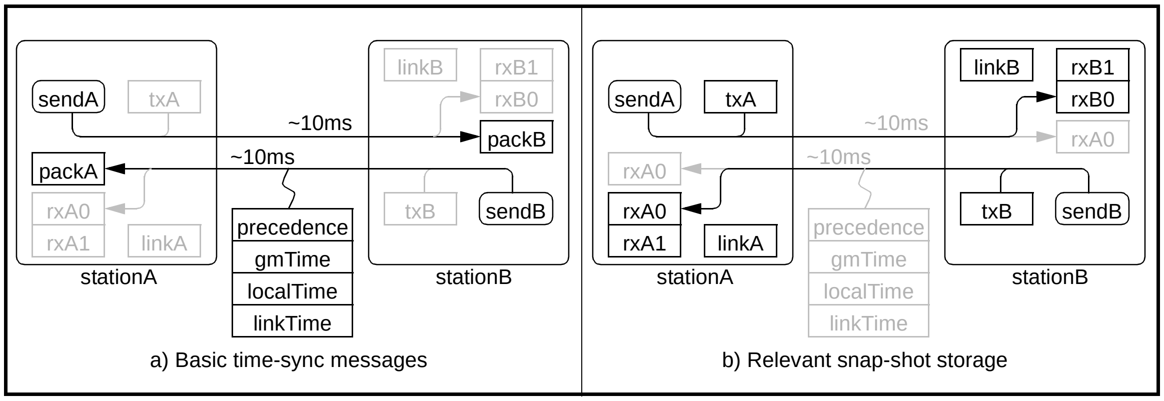


Figure 7.1—Timer snapshot locations

Snapshots are taken when packets are transmitted (illustrated as *txA* and *txB*) and received (illustrated as *rxA* and *rxB*), as illustrated in Figure 7.3b. The transmitted stopshot *txA* is placed into the next frame that is transmitted, as *packetA.localTime*, along with grand-master time *packetA.gmTime* sampled at this time. The transmitted stopshot *txB* is similarly placed into the next frame that is transmitted, as *packetB.localTime*, along with grand-master time *packetB.gmTime* sampled at this time.

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 computed value of *linkA* is the difference between the received *packetFromB.localTime* value and the previous *rxA* snapshot, as specified by Equation 7.1. Similarly, *linkB* (the value transmitted from stationB to stationA) is specified by Equation 7.2.

$$linkA = rxA1 - packetFromB.localTime; \quad (7.1)$$
$$linkB = rxB1 - packetFromA.localTime; \quad (7.2)$$

The value of the intermediate span delay is readily derived from these values. At stationA and stationB, these computations are specified by Equation 7.3 and Equation 7.4, respectively.

$$cableDelayComputedAtA = (linkA + packetFromB.delta)/2; \quad (7.3)$$
$$cableDelayComputedAtB = (packetFromA.delta + linkB)/2; \quad (7.4)$$

7.2.2 Timer snapshot locations

Each port entity is responsible for removing and inserting frames, when processed through the ReceivePort and TransmitPort state machines, as illustrated on the left of Figure 7.2. Other ports could have distinct media-dependent state machines, but share the same common-entity interface, as illustrated on the right of Figure 7.2.

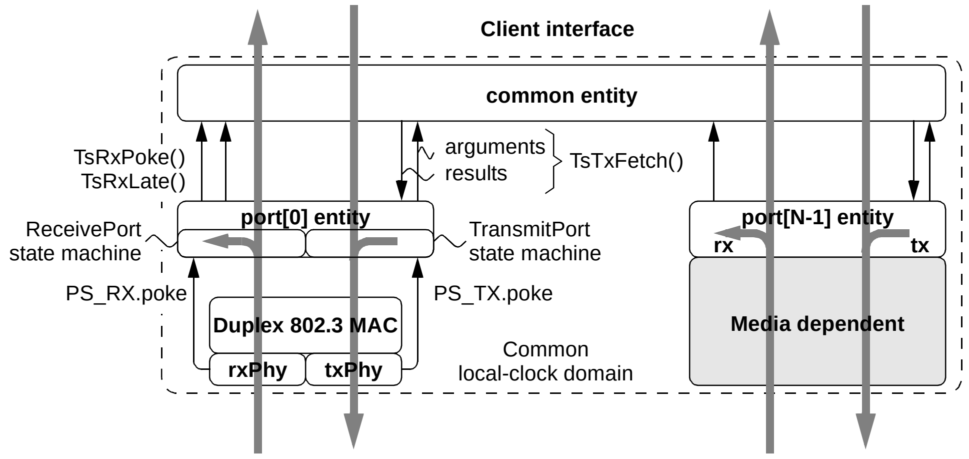


Figure 7.2—Duplex 802.3 MAC interface model

The physical entity that generates PS_RX.poke and PS_TX.poke indications is deliberately left ambiguous. Mandatory jitter-error accuracies are sufficiently loose to allow transmit/receive snapshot circuits to be located with the MAC, as illustrated in Figure 7.3a. Vendors may elect to further reduce timing jitter by latching the receive/transmit times within the PHY, where the uncertain FIFO latencies can be best avoided.

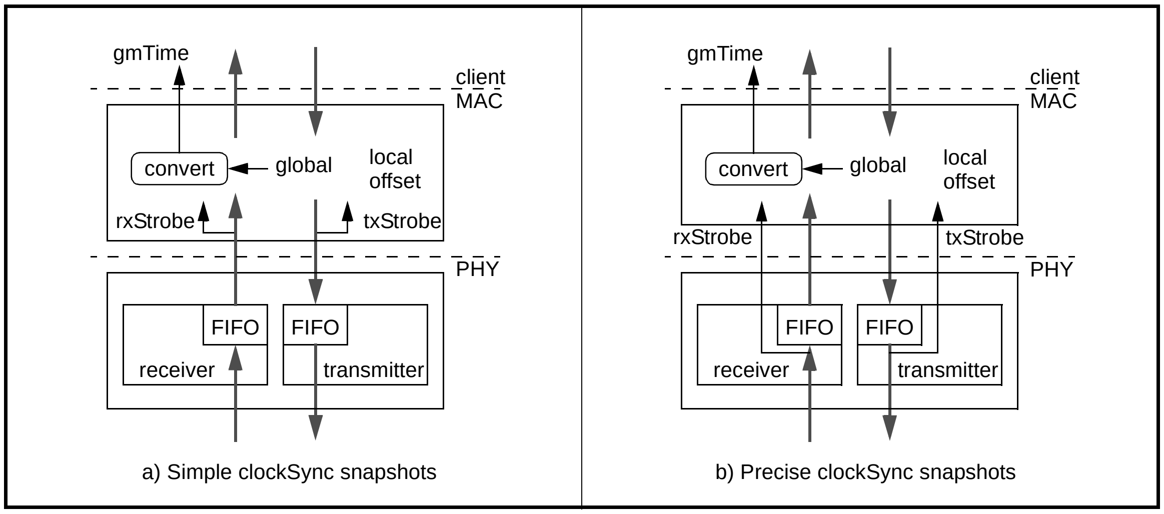


Figure 7.3—Timer snapshot locations

7.3 Port-level service primitives

7.3.1 PS_RX.poke service interface

7.3.1.1 Function

The PS_RX.poke primitive provides the ReceivePort state machine with knowledge of lower-level port-specific time-synchronization information.

7.3.1.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
PS_RX.poke
(
    local_time,
    gm_time
)
```

The parameters provided by the PS_RX.poke service interface are described below.

local_time
  A 40-bit value representing the local-time value associated with the sampled gm_time value.
gm_time
  Provides the value of the distributed grand-master time, compensated for intermediate delays. Fields within this component include the following (more-significant fields are listed first):

	seconds	A signed 48-bit value representing seconds.
|	fraction	An unsigned 32-bit value representing fractions-of-seconds.

NOTE—Within the context of 7.2.1, stationB would provide the following parameters:
local_time—The value of *rxA1*.
gm_time—The sum of two components, *packetFromB.global+cableDelayComputedAtA*, where:
packetFromB.gmTime—the value received from the clock-master stationB.
cableDelayComputedAtA—the most-recent computed value.

7.3.1.3 When generated

The PS_RX.poke primitive is invoked by the port entity whenever new knowledge of time or the grand-master precedence is generated.

7.3.1.4 Effect of receipt

The receipt of the PS_RX.poke supplied information causes the ReceivePort state machine to update common-entity parameters, as specified by the state-machine Table 7.2.

7.3.2 PS_TX.poke service interface

7.3.2.1 Function

The PS_TX.poke primitive provides the TransmitPort state machine with an indication of the last timeSync frame transmission.

7.3.2.2 Semantics of the service primitive

The semantics of the primitives are as follows:

```
PS_TX.poke              // Service primitive
(
    local_time          // Argument
)
```

The parameters of PS_TX.poke are described below.

local_time
A 40-bit value representing the local-time value associated with the sampled gm_time value.

7.3.2.3 When generated

The PS_TX.poke primitive is invoked by the transmit-phy entity whenever a timeSync frame is transmitted.

7.3.2.4 Effect of receipt

The receipt of the PS_TX.poke supplied information causes the TransmitPort state machine to generate next-frame parameters, as specified by the state-machine Table 7.3.

7.3.3 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.

7.4 timeSync frame format

7.4.1 timeSync fields

Clock synchronization (timeSync) frames facilitate the synchronization of neighboring clock span-master and clock span-slave stations. The frame, which is normally sent once each isochronous cycle, includes time-snapshot information and the identity of the network's clock master, as illustrated in Figure 7.4. The gray boxes represent physical layer encapsulation fields that are common across Ethernet frames.

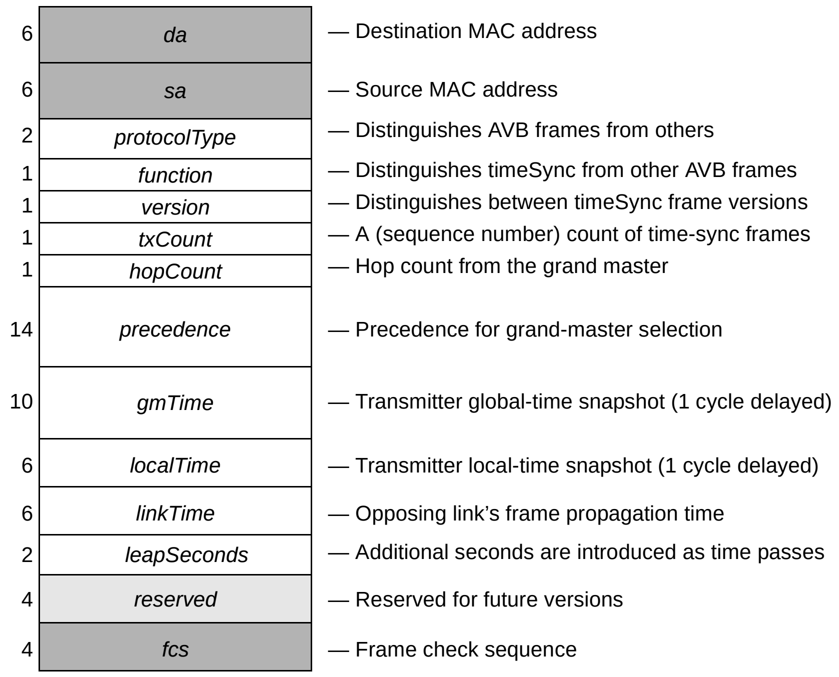

6 da — Destination MAC address
6 sa — Source MAC address
2 protocolType — Distinguishes AVB frames from others
1 function — Distinguishes timeSync from other AVB frames
1 version — Distinguishes between timeSync frame versions
1 txCount — A (sequence number) count of time-sync frames
1 hopCount — Hop count from the grand master
14 precedence — Precedence for grand-master selection
10 gmTime — Transmitter global-time snapshot (1 cycle delayed)
6 localTime — Transmitter local-time snapshot (1 cycle delayed)
6 linkTime — Opposing link's frame propagation time
2 leapSeconds — Additional seconds are introduced as time passes
4 reserved — Reserved for future versions
4 fcs — Frame check sequence

Figure 7.4—timeSync frame format

NOTE—The *gmTime* 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.

7.4.1.1 *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).

7.4.1.2 *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.

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

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

7.4.1.5 *version*: An 8-bit field that identifies the format and function of the following fields.

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

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

7.4.1.8 ***precedence*****:** A 14-byte field that has specifies precedence in the grand-master selection protocols (see 7.4.2).

7.4.1.9 ***gmTime*****:** An 80-bit field that specifies the grand-master synchronized time within the source station, when the previous timeSync frame was transmitted (see 7.4.4).

7.4.1.10 ***localTime*****:** A 48-bit field that specifies the local free-running time within the source station, when the previous timeSync frame was transmitted (see 7.4.5).

7.4.1.11 ***linkTime*****:** A 48-bit field that specifies the differences between timeSync-packet receive and transmit times, as measured in fractions-of-second on the opposing link (see 7.4.5).

7.4.1.12 ***leapSeconds*****:** A 16-bit field that specifies the number of seconds that should be added to the *gmTime* value, when converting between xx and yy values. (On IEEE-1588, this is called the *UTCOffset* field.)

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

7.4.2 *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 7.5.

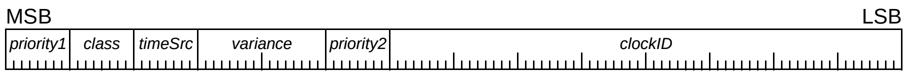


Figure 7.5—*precedence* subfields

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

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

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

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

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

7.4.2.6 ***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 7.4.3).

7.4.3 *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 7.6.

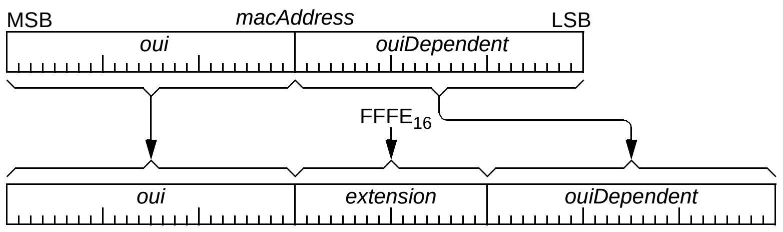


Figure 7.6—*clockID* format

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

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

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

7.4.4 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 7.7.

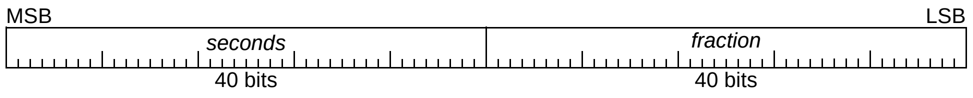


Figure 7.7—Global-time subfield format

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

7.4.4.2 *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 *gmTime* value, as specified by Equation 7.5.

$$gmTime = seconds + (fraction / 2^{40}) \tag{7.5}$$

7.4.5 Local time formats

The local-time values within a frame are based on a fractions-of-second value, as illustrated in Figure 7.8. The 40-bit *fraction* field specifies the time offset within the *second*, in units of 2^{-40} second.

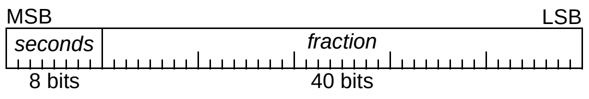


Figure 7.8—Local time format

7.5 ReceivePort state machine

7.5.1 Function

The ReceivePort state machine is responsible for monitoring PS_RX.poke indications and timeSync frames received from the lower layers. It calibrates cable delays and invokes the common time-synchronization entity. The sequencing of this state machine is specified by Table 7.2; most of the computations are specified by the C-code of Annex F.

7.5.2 ReceivePort 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_RX_LL—The queue identifier associated with MAC frames from the lower levels.
- Q_RX_PP—The queue identifier associated with the receive-port poke indications.
- Q_RX_UL—The queue identifier associated with MAC frames for the upper levels.

T100ms
A constant the represents a 100 ms value.

7.5.3 ReceivePort 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.

fields
A group of parameters extracted from a frame, including the following:
- hop_count—The content of the *exSyncFrame.hopCount* field.
- precedence—The content of the *exSyncFrame.precedence* field.
- gm_time—The content of the *exSyncFrame.gmTime* field.
- leap_seconds—The content of the *exSyncFrame.leapSeconds* field.

frame
The contents of a MAC-supplied frame.

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

port
A data structure containing port-specific information comprising the following:
- *rxSyncFrame*—The value of the previously observed timeSync frame.
- *rxPokeCount*—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 *port.rxSnapShot1* field saved from the last poke indication.
- *rxStartTime*—The value of *currentTime* when syncFrame was received, used for timeouts.

7.5.4 ReceivePort 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.

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

7.5.5 ReceivePort state machine table

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

Table 7.2—ReceivePort state machine table

Current		**Row**	**Next**	
state	**condition**		**action**	**state**
START	(frame = Dequeue(Q_RX_LL)) != NULL	1	—	CHECK
	(info = Dequeue(Q_RX_PP)) != NULL	2	port.rxSnapShot1 = rxSnapShot0; port.rxSnapShot0 = info.local_time; port.rxPokeCount = info.frame_count;	FINAL
	(currentTime – port.rxStartTime) > T500ms	3	TsRxLate(port.commonPtr, port.portID);	START
	—	4	—	
CHECK	!TimeSyncFrame(frame)	5	Enqueue(Q_RX_UL, frame);	START
	frame.count != port.rxFrameCount+1	6	port.rxFrameCount = frame.count;	
	—	7	port.rxSyncFrame = frame; port.rxFrameCount = frame.count;	FINAL
FINAL	port.rxFrame.frameCount != port.rxPokeCount	8	—	START
	—	9	rxFields = PsRxParse(&(port)); TsRxPoke(port.commonPtr, port.portID, port.rxSnapShot1, rxFields.hop_count, rxFields.precedence, rxFields.gm_time, rxFields.leap_seconds);	

Row 7.2-1: Initiate inspection of frames received from the lower-level MAC.
Row 7.2-2: Process poke information received from the lower-levels. Time snapshots are updated, the count identifier is saved, and port calculations (leading to an invoking of the common entity) are performed.
Row 7.2-3: The absence of timeSync frames is communicated to the common entity.
Row 7.2-4: Wait for the next timeSync frame or poke indication.

Row 7.2-5: Pass through all non-timeSync frames.
Row 7.2-6: Discard non-sequential frames.
Row 7.2-7: Sequential timeSync frames are processed.

Row 7.2-8: Inhibit processing when the frame and poke counts are different.
Row 7.2-9: Invoke common-entity processing when the frame and poke counts are the same.

7.6 TransmitPort state machine

7.6.1 Function

The TransmitPort state machine is responsible for monitoring the PS_TX.poke indications and (in response) generating the delayed periodic timeSync frames; these timeSync frames are passed through lower layers towards the transmit port. The TransmitPort state machine accesses the common time-synchronization entity to extract the desired parameters, by invoking PsTx() which then invokes the TsTxFetch() primitive.

7.6.2 TransmitPort state machine definitions

NULL
    See 7.5.2.
queue values
    Enumerated values used to specify shared FIFO queue structures.
        Q_TX_LL—The queue identifier associated with MAC frames queued to lower layers.
        Q_TX_PP—The queue identifier associated with the transmit-port poke indications.
        Q_TX_UL—The queue identifier associated with MAC frames dequeued from upper layers.
SEND_INTERVAL
    A constant the represents a 10 ms value.

7.6.3 TransmitPort state machine variables

frame
    See 7.5.3.
info
    A contents of a lower-level supplied time-synchronization poke indication, including the following:
        local_time—The value of *currentTime* associated with the last timeSync packet departure.
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.
        txSendTime—The value of *currentTime* when timeSync frame was enqueued, used for pacing.

7.6.4 TransmitPort state machine routines

Dequeue(*queue*)
Enqueue(*queue*)
    See 7.5.4.

FreeQueue(*queue*)

Identifies whether space is available for another frame to be placed into the specified queue.

TRUE—Space for another frame is available.

FALSE—Otherwise.

PsTx(*portPtr*)

Forms the next to-be-transmitted timeSync frame, based on port parameters.

7.6.5 TransmitPort state machine table

The TransmitPort state machine includes a media-dependent timeout, which effectively delays the timeSync frame transmissions to their desired periodic rate, as illustrated in Table 7.3.

Table 7.3—TransmitPort state machine table

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

START	(info = Dequeue(Q_TX_PP)) != NULL	1	port.txSnapShot = info.local_time;
			txFields= TsTxFetch(port.commonPtr,
			port.portID,
			port.txSnapShot);
			port.txFrame = PsTxForm(&port,
			txFields.hop_count,
			txFields.prededence,
			txFields.gm_time,
			txFields.leap_seconds);
				START

	FreeQueue(Q_TX_LL)	2	—	FREE
	—	3	—	START

FREE	(currentTime – port.txSendTime) >= SEND_INTERVAL &&
	port.txSyncFrame != NULL
		4	Enqueue(Q_TX_LL, port.txSyncFrame);
			port.txFrame = NULL;
			port.txSendTime = currentTime;
				START

	(frame = Dequeue(Q_TX_UL)) != NULL	5	Enqueue(Q_TX_LL, frame);	
	—	6	—	

EDITOR NOTE—The intent is to minimize the periodic transmission requirements, so they can be implemented in the most inexpensive way. The preceding state machine may therefore be modified, to better illustrate that the periodic nature could be based on either independent port activities or centralized common-entity synchronization.

Row 7.2-1: Process poke information received from the transmit lower-level.
Row 7.2-2: Check for free space within the transmit queue.
Row 7.2-3: Otherwise, wait for the next event.

Row 7.2-4: Save any preformed timeSync frame.
Row 7.2-5: Pass through all non-timeSync frames.
Row 7.2-6: Otherwise, continue checking for the next event.

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

Table B.1—Time-scale conversions

Acronym	Name	Row	offset	Algorithm
PTP	Precision Time protocol	1	0	time = gmTime + offset;
GPS	global positioning satellite	2	–315 964 819	
UTC	Coordinated Universal Time	3	TBD	time = gmTime + offset – leapSeconds;
NTP	Network Time Protocol	4	+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.

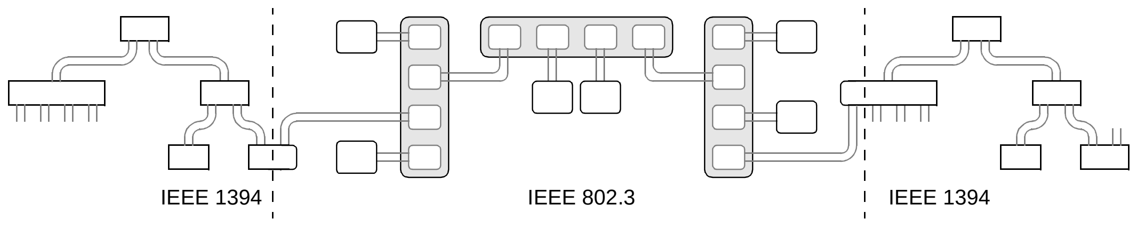


Figure C.1—IEEE 1394 leaf domains

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.

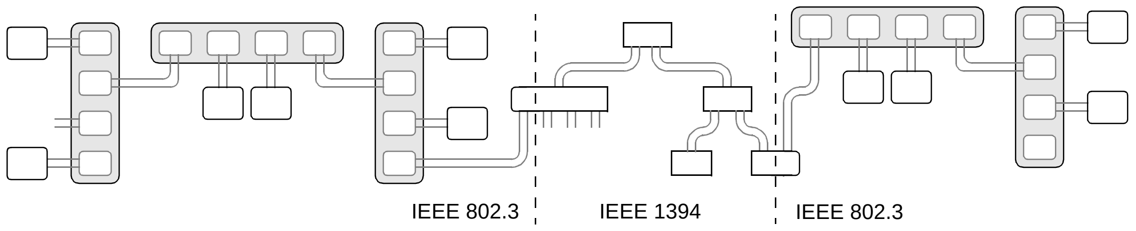


Figure C.2—IEEE 802.3 leaf domains

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.

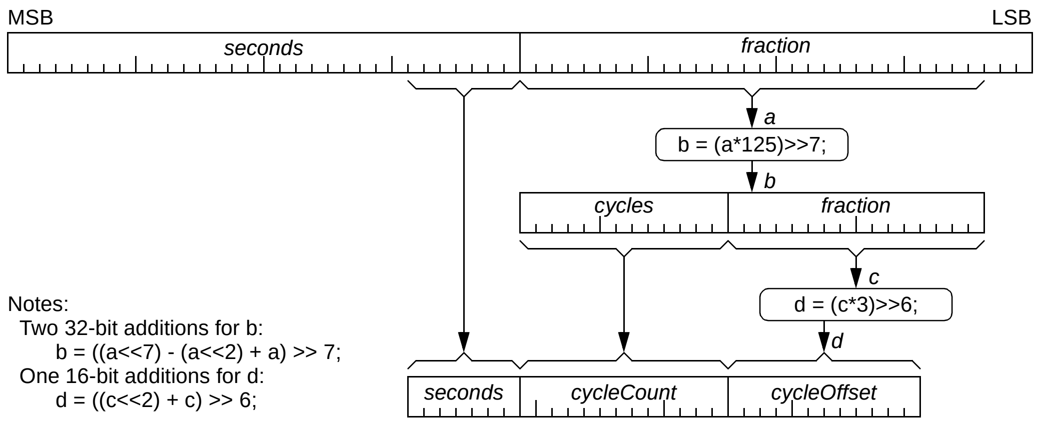


Figure C.3—Time-of-day format conversions

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:

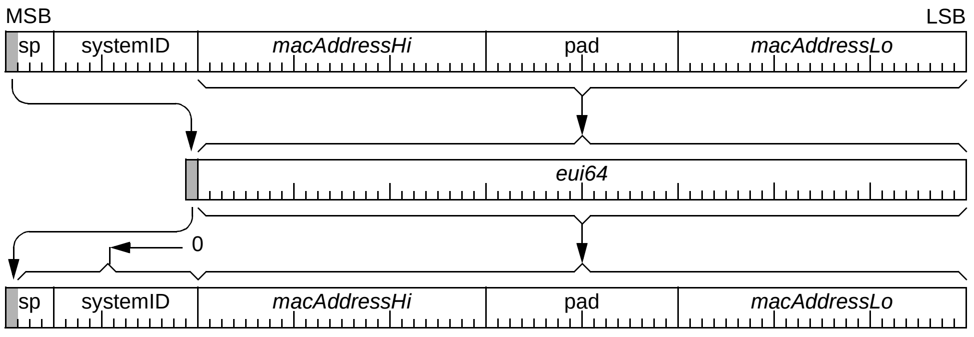


Figure C.4—Grand-master precedence mapping

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, other formats are evaluated and compared 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 32-bit *seconds* and 32-bit *fraction* fields, as illustrated in Figure 5.1.

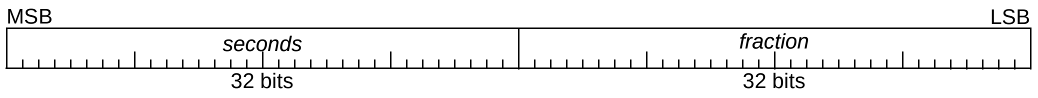


Figure 5.1—Complete seconds timer format

The concatenation of 32-bit *seconds* and 32-bit *fraction* field specifies a 64-bit *time* value, as specified by Equation E.1.

$$time = seconds + (fraction / 2^{32}) \qquad \text{(E.1)}$$

Where:

seconds is the most significant component of the time value (see Figure 5.1).

fraction is the less significant component of the time value (see Figure 5.1).

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 3171 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.

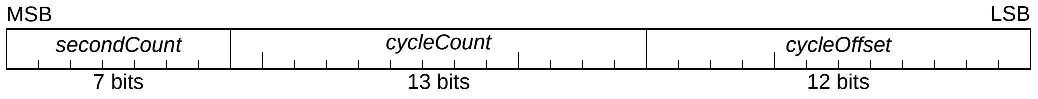


Figure E.2—IEEE 1394 timer format

E.1.3 IEEE 1588 timer format

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

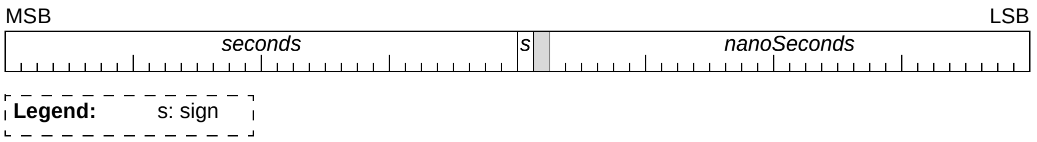


Figure E.3—IEEE 1588 timer format

E.1.4 EPON timer format

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

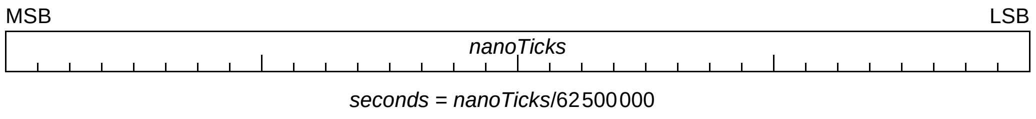


Figure E.4—EPON timer format

E.1.5 Compact seconds timer format

An alternate "compact seconds" format could consist of 8-bit *seconds* and 24-bit *fraction* fields, as illustrated in Figure E.5. This would provided similar resolutions to the IEEE 1394 timer format, without the complexities associated with its binary coded decimal (BCD) like encoding.

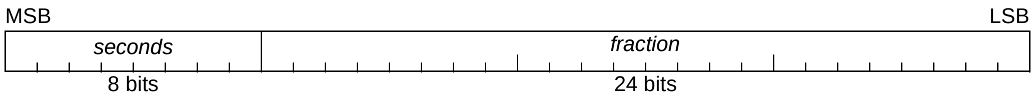


Figure E.5—Compact seconds timer format

E.1.6 Nanosecond timer format

An alternate "nanosecond" format could consists of 2-bit *seconds* and 30-bit *nanoSeconds* fields, as illustrated in Figure E.6.

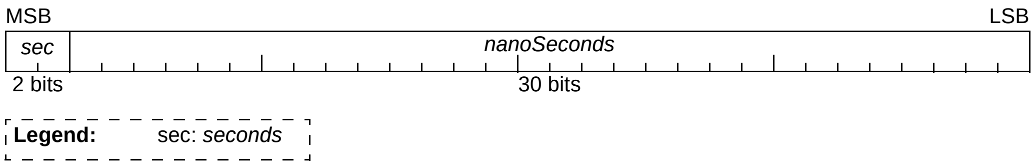


Figure E.6—Nanosecond timer format

E.2 Time format comparisons

To better understand the relative benefits of different time formats, the relevant properties are summarized in Table E.1. Counter complexity is not included in the comparison, since the digital logic complexity is comparable for all formats.

Table E.1—Time format comparison

Name	Subclause	Range	Precision	Arithmetic	Seconds	Defined standards
Column	—	1	2	3	4	5
extended binary	TBD	136 years	232 ps	Good	Good	RFC 1305 NTP, RFC 2030 SNTPv4
IEEE 1394	E.1.2	128 s	30 ns	Poor	Good	IEEE 1394
IEEE 1588	E.1.3	272 years	1 ns	Fair	Good	IEEE 1588
IEEE 802 (EPON)	E.1.4	69 s	16 ns	Good	Poor	IEEE 802.3
compact seconds	E.1.5	256 s	60 ns	Best	Good	—
nanoseconds	E.1.6	4 s	1 ns	Best	Poor	—

Column 1: A desirable property is the support of a wide range of second values, to eliminate the need for defining/coordinating/implementing auxiliary seconds-synchronization protocols. The 136-year range of the extended binary format is sufficient for this purpose.

Column 2: A desirable property is a fine-grained resolution, sufficient to measure each bit-transmission times. The ‘extened binary’ provides the most precision; exceeds the resolution of expected cost-effective time-capture circuits.

Column 3: Computation of time differences involves the subraction of two timer-snapshot values. Subtraction of ‘extended binary’ numbers involving standard 64-bit binary arithmetic; no special field-overlow compensations are required. Only the less precise ‘compact seconds’ and nanoseconds formats are simpler, due to the reduced 32-bit size of the timer values.

Column 4: Time values must oftentimes be compared to externally provided values (e.g., timers extracted from GPS or stratum-clock sources). For these purposes, the availability of a seconds component is desired. The ‘extended binary’ format provides a seconds component that can be easily extracted or such purposes.

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
//34567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012

// 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
// *****************************************************************************************
#define NEIGHBOR  0                                                                     // Neighbor multicast address.
#define AVB_TYPE  0                                                                     // The protocolType for AVB.
#define TIME_SYNC 0                                                                     // The timeSync function.
#define VERSION_0 0                                                                     // The timeSync version.


#define FALSE 0
#define TRUE  1
#define TIMEOUT TRUE
#define HUGE (0x7FFFFFFF)                                                               // Biggest 32-bit positive integer
#define SCALE 4096                                                                      // Scales 100PPM to 1/2
#define MAX(a, b) ((a) < (b) ? (b) : (a))                                               // Maximum value definition
#define MIN(a, b) ((a) > (b) ? (b) : (a))                                               // Minimum value definition
#define CLIP(x) ((x) > HUGE ? HUGE : ((x) < -HUGE ? -HUGE : (x)))                      // Clip to |x| <= 2**31-1,
#define DEAD 255                                                                        // Unused port number
#define T100ms ((((uint32_t)1)<<31)/5)                                                  // A 100ms timing interval
#define HopsBits (8 * Sizeof(Hops))
#define PortBits (8 * Sizeof(Port))
#define MASK(bits) (((uint64_t)1 << bits) - 1)
#define BITS(type) (8 * sizeof(type))

#define FieldToSigned(fPtr, field) \
 FrameToValue(fPtr, (uint8_t *)(&(fPtr->field)), sizeof fPtr->field, TRUE)             // Convert field to signed
#define FieldToUnsigned(fPtr, field) \
 FrameToValue(fPtr, (uint8_t *)(&(fPtr->field)), sizeof fPtr->field, FALSE)            // Convert field to unsigned
#define BigToFrame(value, fPtr, field) \
 ValueToFrame(value, fPtr, (uint8_t *)(&(fPtr->field)), sizeof fPtr->field)            // Convert field to unsigned
#define LongToFrame(value, fPtr, field) \
 ValueToFrame(LongToBig(value), fPtr, (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   Hops;
typedef uint8_t   Port;
typedef uint16_t  Variance;
typedef int16_t   LeapSeconds;
typedef uint32_t  Priorities;
typedef int64_t   LocalTime;
typedef BigNumber GrandTime;
typedef BigNumber Preference;                                           // Fields {priorities,clockID}
typedef BigNumber Precedence;                                           // Fields {preference,hops,port}

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 txCount[1];                                                 // Transmit count (sequence number)
    uint8_t hopCount[1];                                                // Hop-count from the grand master
    uint8_t precedence[14];                                             // Grand-master precedence
    uint8_t gmTime[10];                                                 // Grand-master time  (for last frame)
    uint8_t localTime[6];                                               // Local-station time (for last frame)
    uint8_t linkTime[6];                                                // Apparent link delay
    uint8_t leapSeconds[2];                                             // Leap seconds compensation
    uint8_t reserved[4];                                                // Apparent link delay
    uint8_t fcs[4];                                                     // CRC integrity check
} TimeSync;

typedef struct                                                          // Common entity state
{
    BigNumber thisPrecedence;                                           // more-significant portion
    BigNumber bestPreference;                                           // Grand-master preference
    uint8_t bestPort;                                                   // Incoming from the grand-master
    LocalTime thisSnap;                                                 // Sampled this_time value
    GrandTime thatSnap;                                                 // Sampled gm_time value
    LocalTime thisShot;                                                 // The last rate snapshot, local time
    GrandTime thatShot;                                                 // The last rate snapshot, gmTime
    int32_t diffRate;                                                   // Rate difference from grand-master
} CommonData;

typedef struct                                                          // Port entity state
{
    CommonData *comPtr;                                                 // Pointer to common data
    uint64_t  macAddress;                                               // MAC address of the port

    uint8_t   portID;                                                   // Destinctive port identifier
    uint8_t   rxPokeCount;                                                             // The information-poke count.
    uint8_t   rxFrameCount;                                                            // The timeSync frame count.
    LocalTime linkTime;                                                                // Apparent link delay
    LocalTime rxSnapShop0;                                                             // This frame’s arrival time
    LocalTime rxSnapShop1;                                                             // Past frame’s arrival time
    TimeSync  rxSyncFrame;                                                             // Received timeSync frame.

    uint8_t   txFrameCount;                                                            // The timeSync frame count.
    LocalTime txSendTime;                                                              // TimeSync activation time
    LocalTime txSnapShot;                                                              // Transmit frame snapshot
    TimeSync  txSyncFrame;                                                             // To be transmitted timeSync.
} PortData;

typedef struct                                                                         // Returned values for TsTx()
{
    uint8_t hop_count;                                                                 // Updated hop count
    BigNumber precedence;                                                              // Grand-master precedence
    GrandTime gm_time;                                                                 // Grand-master time
    uint16_t leap_seconds;                                                             // Leap-seconds for time.
} TxFields;

typedef struct
{
    Hops hop_count;
    Precedence precedence;
    GrandTime gm_time;
    LeapSeconds leap_seconds;
} RxFields;


LocalTime localTime;                                                                   // Shared time reference
CommonData commonData;                                                                 // Common state information

// 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(uint64_t, uint64_t);

// Other routines
Precedence  FieldsToPrecedence(uint8_t, uint8_t, uint16_t, uint8_t, uint64_t);
BigNumber   FrameToValue(TimeSync *, uint8_t *, uint16_t, Boolean);
BigNumber   FormPreference(BigNumber, uint8_t, uint8_t);
BigNumber   LongToBig(LocalTime);
Port        PreferenceToPort(Precedence);
RxFields    PsRxParse(PortData *);
TimeSync    PsTx(PortData *pPtr);
void        TsRxPoke(CommonData *, Port, LocalTime, Hops, BigNumber, BigNumber, LeapSeconds);
void        TsRxLate(CommonData *, uint8_t);
TxFields    TsTxFetch(CommonData *, uint8_t, LocalTime);
void        ValueToFrame(BigNumber, TimeSync *, uint8_t *, uint16_t);
```

```
// ***************************************************************************************
// Port-specific routines, called by corresponding state machines.
// ***************************************************************************************

RxFields
PsRxParse(PortData *pPtr)
{
    RxFields result;
    TimeSync *fPtr;
    GrandTime gmTime;
    uint64_t cableDelay, localTime, linkTime;
    int32_t deltaTime;

    assert(pPtr != NULL);
    fPtr = &(pPtr->rxSyncFrame);
    localTime = FieldToSigned(fPtr, localTime).lower;                                   // Frame transmission time.
    linkTime =  FieldToSigned(fPtr, linkTime).lower;                                    // Opposing link delay.
    gmTime =    FieldToSigned(fPtr, gmTime);                                            // GM synchronized time.
    pPtr->linkTime = deltaTime = (pPtr->rxSnapShop1 - localTime);                      // Cable delay is average of
    cableDelay = MIN(0, (deltaTime + linkTime)/2);                                     // ingress/egress link delays.
    result.gm_time = BigAddition(gmTime, LongToBig(cableDelay));                       // Adjust time with cable delay;

    result.hop_count =    FieldToUnsigned(fPtr, hopCount).lower;                       // Hops from the GM station.
    result.precedence =   FieldToUnsigned(fPtr, precedence);                           // GM precedence value.
    result.leap_seconds = FieldToSigned(fPtr, leapSeconds).lower;                      // GM precedence value.
    return(result);
}

TimeSync
PsTxFrame(PortData *pPtr,
 Hops hop_count, Precedence precedence, GrandTime gm_time, LeapSeconds leap_seconds)
{
    TxFields fields;
    TimeSync frame, *fPtr;

    assert(pPtr != NULL);
    fPtr = &frame;
    fields = TsTxFetch(pPtr->comPtr, pPtr->portID, pPtr->txSnapShot);                  // Fetch relevant parameters.

    LongToFrame(NEIGHBOR,            fPtr, da);                                         // Neighbor multicast address.
    LongToFrame(pPtr->macAddress,    fPtr, sa);                                         // This port’s MAC address.
    LongToFrame(AVB_TYPE,            fPtr, protocolType);                               // The AVB protocol.
    LongToFrame(TIME_SYNC,           fPtr, function);                                   // The timeSync frame in AVB.
    LongToFrame(VERSION_0,           fPtr, version);                                    // This version number.
    LongToFrame(pPtr->txFrameCount,  fPtr, txCount);                                    // The sequence number.
    LongToFrame(pPtr->txSnapShot,    fPtr, localTime);                                  // Local time (of last frame).
    LongToFrame(pPtr->linkTime,      fPtr, linkTime);                                   // Observed link delays.
    LongToFrame(0,                   fPtr, reserved);                                   // Reserved (zero for now).

    LongToFrame(hop_count+1,  fPtr, hopCount);                                          // The GM distance.
    BigToFrame(precedence,    fPtr, precedence);                                        // The GM precedence.
    BigToFrame(gm_time,       fPtr, gmTime);                                            // The GM time (of last frame).
    LongToFrame(leap_seconds, fPtr, leapSeconds);                                       // Leap seconds.
    pPtr->txFrameCount += 1;                                                            // Increment for next frame.
    return(frame);                                                                      // Return frame for transmission.
```

```
}

// ***************************************************************************************
// Common parameter-access routines, called by the port-specific routines.
// ***************************************************************************************

void
TsRxPoke(CommonData *cPtr, Port port, LocalTime this_time, Hops hop_count,
 BigNumber gm_precedence, GrandTime gm_time, LeapSeconds leap_seconds)
{
    BigNumber rxPreference, myPreference;
    LocalTime thisTime;
    uint32_t localDelta, gmDelta;
    uint8_t past;
    Boolean tiny, same, less, more, dead;

    assert(cPtr != NULL);                                                   // GM selection follows
    past = PreferenceToPort(cPtr->bestPreference);                          // The clock-slave port
    myPreference = FormPreference(cPtr->thisPrecedence, 0, 255);            // The station precedence
    rxPreference = FormPreference(gm_precedence, hop_count, port);          // Receive port precedence
    tiny = (BigCompare(rxPreference, myPreference) < 0);                    // Deferred to this station
    same = (port == past);                                                  // On the clock-slave port
    less = (BigCompare(rxPreference, cPtr->bestPreference) < 0);            // The preference is less
    more = (BigCompare(rxPreference, cPtr->bestPreference) > 0);            // The preference is more
    dead = (hop_count == DEAD);                                             // Lifetime has expired
    if (same && (tiny || dead)) {
        cPtr->bestPreference = myPreference;                                // Station becomes the GM.
    } else {
        if (!dead && (more || (same && less))) {
            cPtr->bestPreference = rxPreference;                            // Port is/becomes the GM.
            cPtr->thisShot = this_time;                                     // Restart the rate
            cPtr->thatShot = gm_time;                                       // compensation shapshots
        }
    }
    past = PreferenceToPort(cPtr->bestPreference);                          // Revised clock-slave portID


    // **********************************************************************
    // Following assumes that clock-rate differences jump when the GM changes.
    // A slow transition could be easily implemented, if desired slope is known.
    // **********************************************************************

    if (past == 255) {                                                      // The grandmaster does the following
        cPtr->thisSnap = thisTime = localTime;                              // Snapshots are exact
        cPtr->thatSnap = LongToBig(localTime);                              // Snapshots are exact
        cPtr->diffRate = 0;                                                 // The clock-rate difference is zero
        return;
    }

    if (port != past)                                                       // Ignore non-clock-slaves
        return;
    cPtr->thisSnap = this_time;                                             // Save’s time parameters
    cPtr->thatSnap = gm_time;                                               // supplied by the port,
    localDelta = (this_time - cPtr->thisShot);                              // Wait a longer interval before
    if (localDelta < T100ms)                                                // computing the rate difference.
```

```
        return;
    gmDelta = BigSubtract(gm_time, cPtr->thatShot).lower;                               // Global timer changes
    cPtr->diffRate = DivideHi(gmDelta - localDelta, localDelta);                        // Save rate difference.
    cPtr->thisShot = this_time;                                                         // The local-time snapshot
    cPtr->thatShot = gm_time;                                                           // The grand-master snapshot
}

void
TsRxLate(CommonData *cPtr, uint8_t port)
{
    uint8_t past;

    assert(cPtr != NULL);                                                               // GM selection follows
    past = PreferenceToPort(cPtr->bestPreference);                                      // The clock-slave port
    if (port == past)                                                                   // Ignore non clock-slave.
        cPtr->bestPreference = FormPreference(cPtr->thisPrecedence, 0, 255);            // The station precedence
    return;
}

TxFields
TsTxFetch(CommonData *cPtr, uint8_t port, LocalTime this_time)
{
    TxFields values;
    uint64_t delayed;

    assert(cPtr != NULL);
    delayed = (this_time - cPtr->thisSnap);                                             // Time since observation
    delayed += MultiplyHi(delayed, cPtr->diffRate) / SCALE;                             // Scaled rate difference
    values.precedence = BigShift(cPtr->bestPreference, 16);                             // Extract the precedence
    values.hop_count = BigShift(cPtr->bestPreference, 8).lower & 0XF;                   // Extract the hop count
    values.gm_time = BigAddition(cPtr->thatSnap, LongToBig(delayed));                   // Computed GM time
    return(values);                                                                     // Returned parameters
}


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

BigNumber                                                                               // Extracts field of frame,
FrameToValue(TimeSync *fPtr, 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
}

void                                                                                    // Place fields into frame,
ValueToFrame(BigNumber value, TimeSync *fPtr, uint8_t *fieldPtr, uint16_t length)        // signed properties ignored.
{
    int i;
    uint8_t *cPtr;

    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.
// ************************************************************************************

Hops
PreferenceToHops(BigNumber preference)
{
    Hops result;

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

Port
PreferenceToPort(Precedence preference)
{
    Hops result;

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

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);
}

BigNumber
LongToBig(int64_t number)
{
    BigNumber result;

    result.lower = number;
    result.upper = 0;
    if (number< 0)
        result.upper -= 1;
    return(result);
}

BigNumber
FormPreference(BigNumber precedence, Hops hopCount, Port port)
{
    BigNumber result;

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

BigNumber                                                                                  // Addition of BigNumbers
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);
}

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);
}
```

```
// Currently written assuming largest is best.
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
}

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);
}

int64_t
MultiplyHi(uint64_t value2, int32_t value1)
{
    int64_t result;

    result = (value2 >> 32) * value1 + (((value2 & 0XFFFFFFFF) * value1) >> 32);         // Compute the result;
    return(result);                                                                          // return the result.
}

int64_t                                                                                      // Scaled divide for ranges
```

```
DivideHi(uint64_t valueA, uint64_t valueB)                                        // valueA,valueB < 2**48
{
    int64_t data, div0, div1, div2, result;

    data = (valueA << 12);
    div0 = data / valueB;
    data = (data % valueB) << 16;
    div1 = data / valueB;
    data = (data % valueB) << 16;
    div2 = data / valueB;
    result = (div0 << 32) + (div1 << 16) + div2;
    return(result);                                                               // Return the result.
}
```