JggDvj2005Apr16
February 14, 2007

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

Draft 0.214

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.

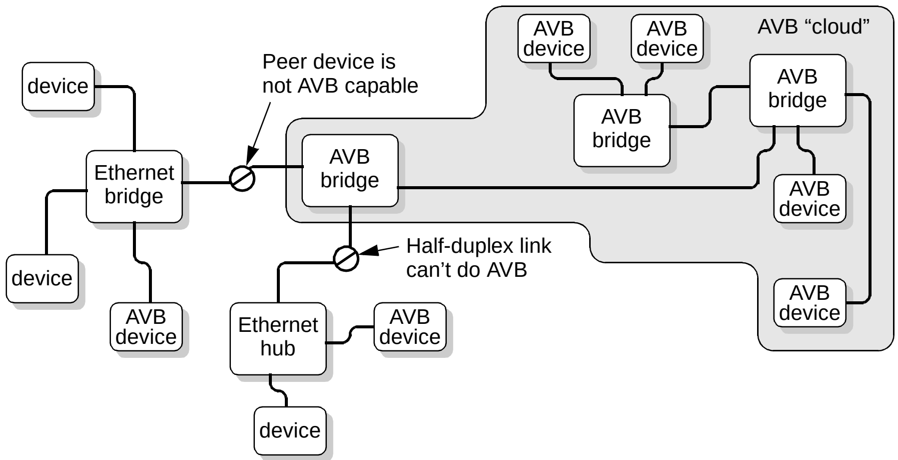


Figure 1.1—Topology and connectivity

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: Duplex-link state machines
— Annex A: Bibliography
— Annex C: Bridging to IEEE Std 1394
— Annex D: Review of possible alternatives
— Annex E: Time-of-day format considerations
— Annex F: C-code illustrations

2. References

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

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

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

[1]Replaces ANSI X3.159-1989

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

3. Terms, definitions, and notation

3.1 Conformance levels

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

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

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

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

3.2 Terms and definitions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 State machines

3.3.1 State machine behavior

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

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

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

3.3.2 State table notation

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

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

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

Table 3.1—State table notation example

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

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

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

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

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

3.4 Arithmetic and logical operators

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

Table 3.2—Special symbols and operators

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

3.5 Numerical representation

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

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

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

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

3.6 Field notations

3.6.1 Use of italics

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

3.6.2 Field conventions

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

Table 3.3—Names of fields and sub-fields

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

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

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

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

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

3.6.3 Field value conventions

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

Table 3.4—*wrap* field values

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

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

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

3.7 Bit numbering and ordering

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

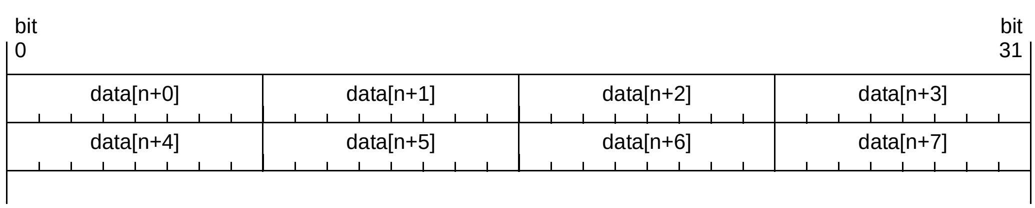


Figure 3.1—Bit numbering and ordering

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

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

3.8 Byte sequential formats

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

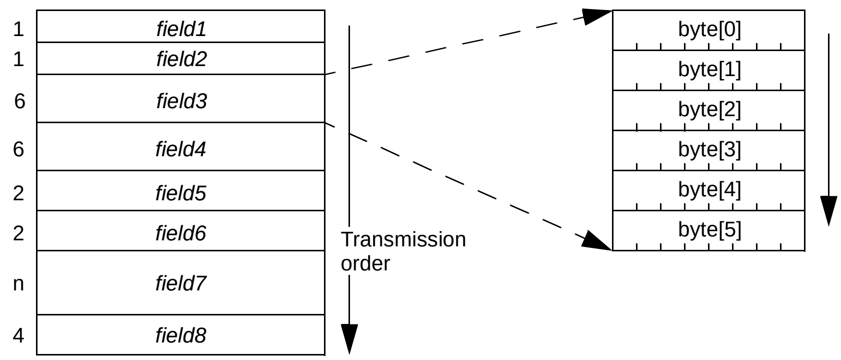


Figure 3.2—Byte sequential field format illustrations

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

3.9 Ordering of multibyte fields

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

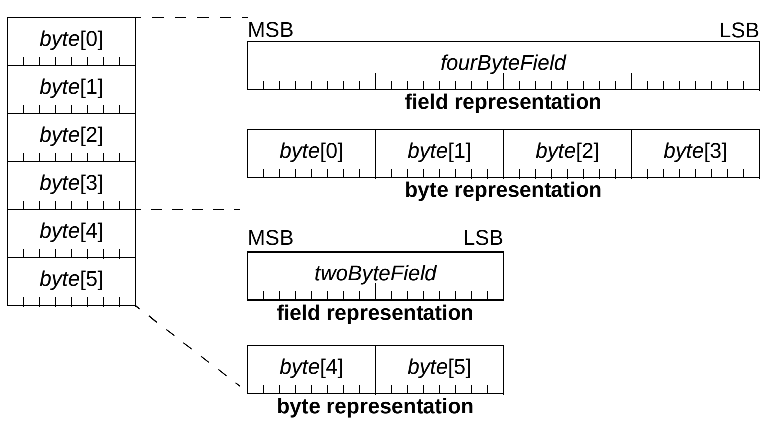


Figure 3.3—Multibyte field illustrations

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

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

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

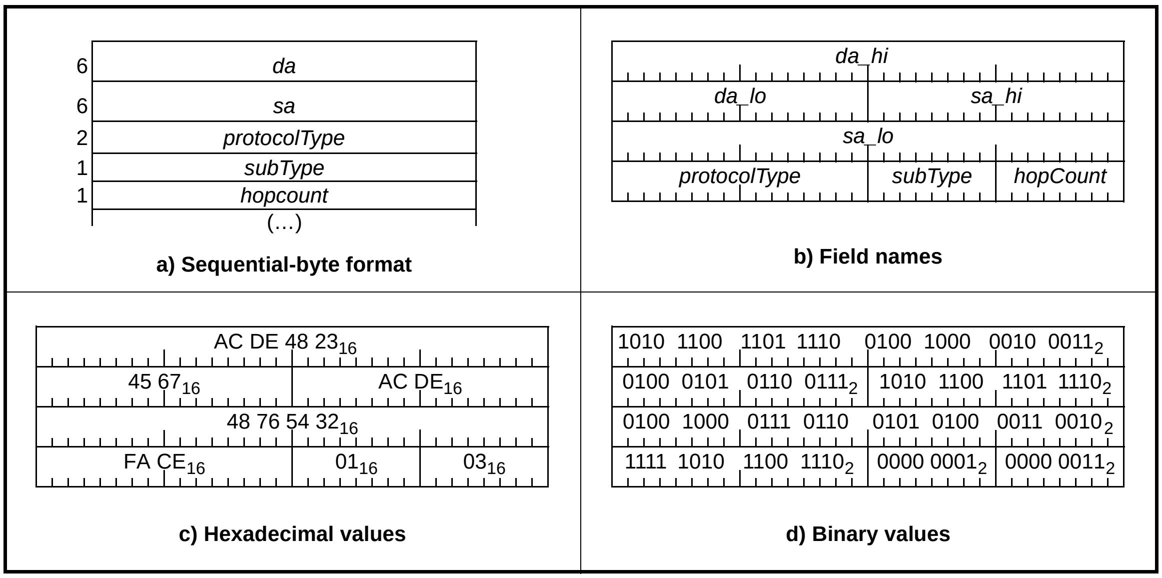


Figure 3.4—Illustration of fairness-frame structure

3.10 MAC address formats

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

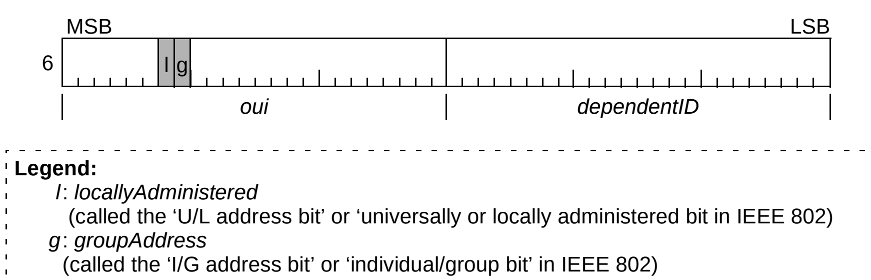


Figure 3.5—MAC address format

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

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

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

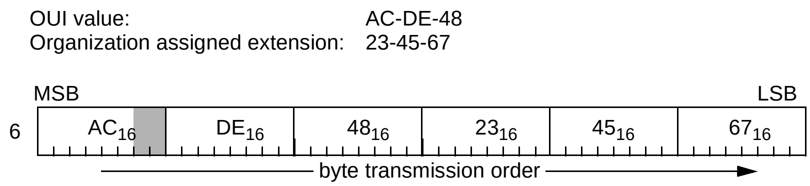


Figure 3.6—48-bit MAC address format

3.11 Informative notes

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

NOTE—This is an example of an informative note.

3.12 Conventions for C code used in state machines

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

4. Abbreviations and acronyms

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.

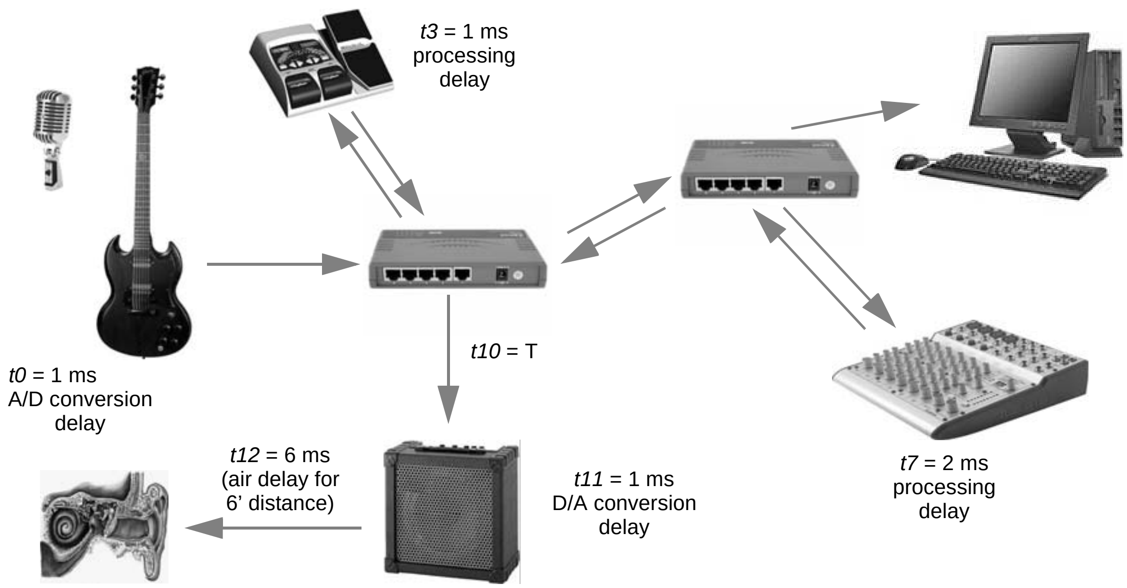


Figure 5.1—Garage jam session

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.

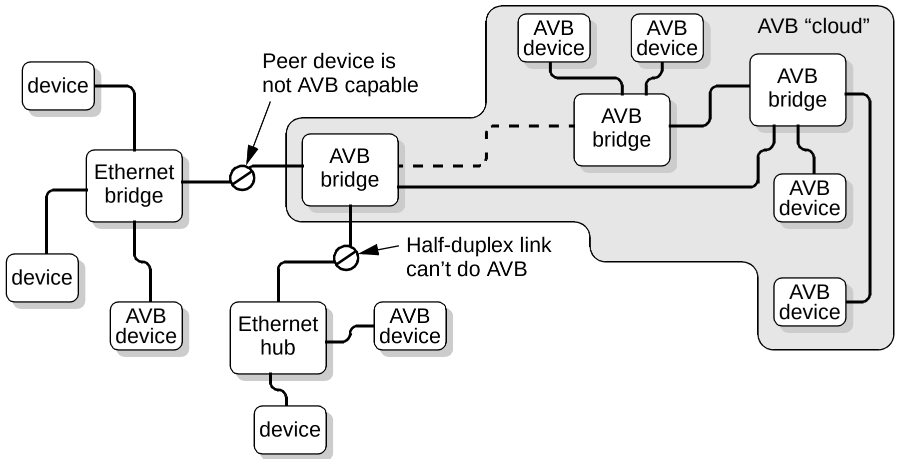


Figure 5.2—Possible looping topology

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 *grandTime* clocks.
   1) Offsets. Offset value adjustments eliminate immediate clock-value errors.
   2) Rates. Rate value adjustments reduce long-term clock-drift errors.
b) Simplicity is achieved by the following:
   1) Concurrence. Most configuration and adjustment operations are performed concurrently.
   2) Feed-forward. PLLs are unnecessary within bridges, but possible within applications.
   3) Frequent. Frequent (nominally 100 Hz) interchanges reduces needs for overly precise clocks.

5.3 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.

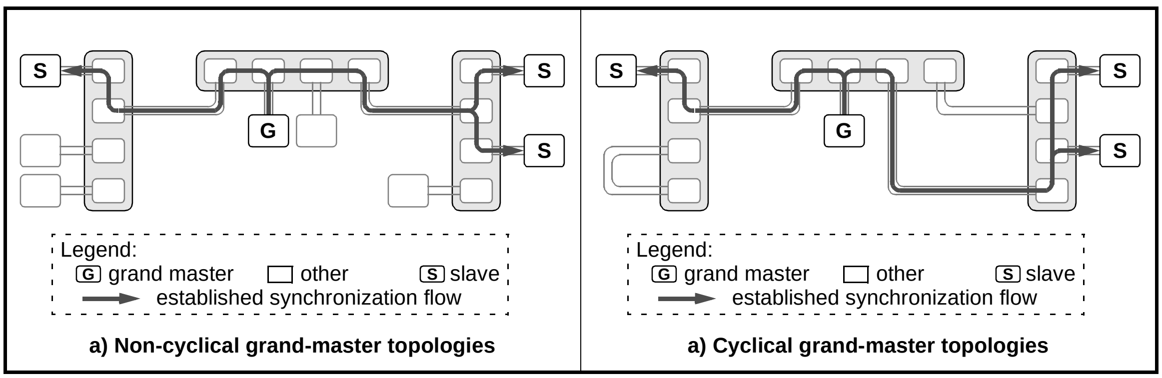


Figure 5.3—Timing information flows

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.

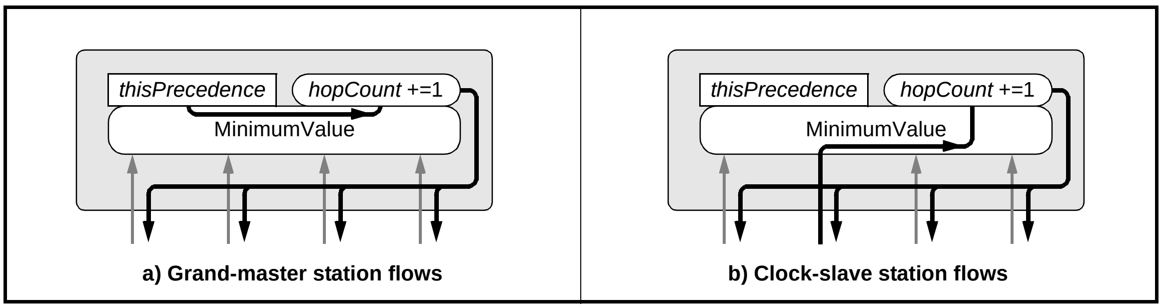


Figure 5.4—Grand-master precedence flows

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.

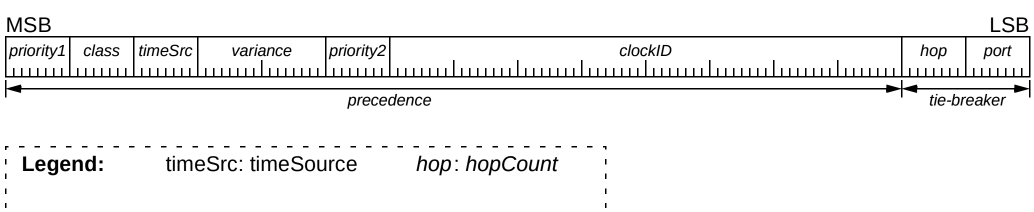
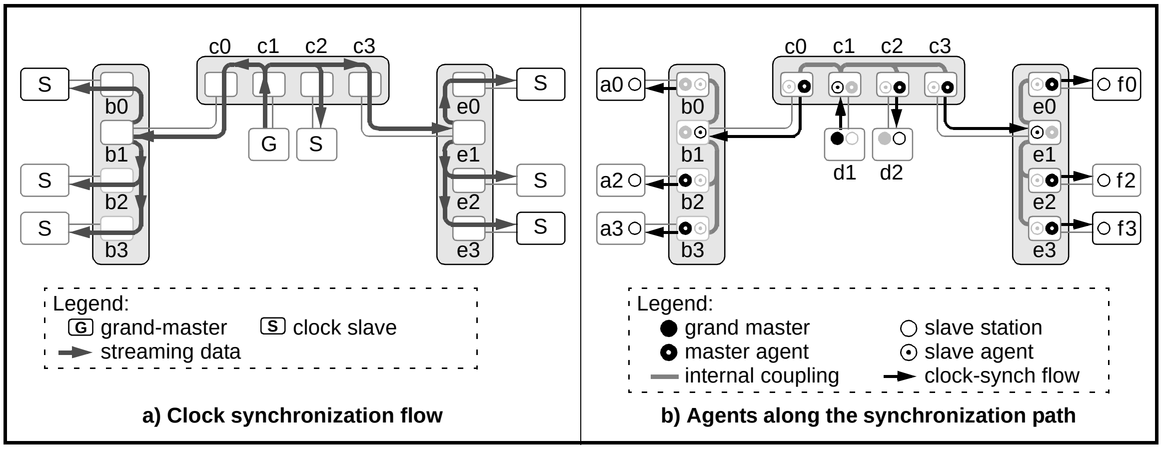


Figure 5.5—Grand-master selector

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.



a) Clock synchronization flow

b) Agents along the synchronization path

Figure 5.6—Hierarchical flows

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

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

5.4 Grand-master time distribution

The propagation of the grand-master time to other stations involves the transmission of *grandTime*, a value the represents the grand-master's time sampled at an instance in the past, as illustrated in Figure 5.7. Associated with the *grandTime* time is a *diffRate* that represents the rate differences between this station and the grand-master.

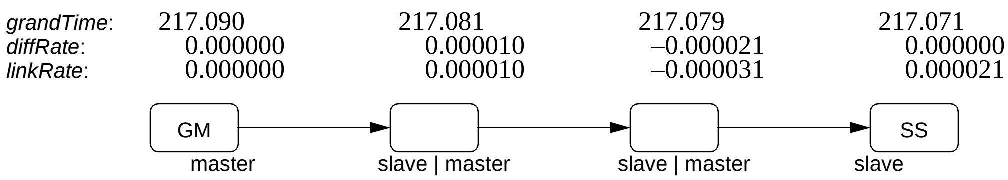


Figure 5.7—Grand-master time distribution

Each stations between the grand-master (GM) and the synchronized slave (SS) is responsible for computing *thisRate*, the rate difference between its free-running clock and the free-running clock of its master-side neighbor. The computation of these rate differences is done at a slow rate (1/10 of the timeSync-frame transmission rate), to improve the *linkRate* calculation accuracies.

The calculation of link delays between neighbors is based on the station's local clocks, as normalized by the computed *linkRate* values. Thus, the calibration of link delays is unaffected by the accuracy, dynamics, and transients associated with the *grandTime* distribution protocols**.**

Each station adds its *linkRate* value to its neighbor-supplied *diffRate* value, so that the cumulative *diffRate* value represents the rate difference between the station and its grand-master. This cumulative value is then used to accurately measure delays when the timeSync frame passes through the station.

5.5 Duplex-link time-synchronization operation

On some forms of duplex-link media, time-synchronization involves periodic not-necessarily synchronized packet transmissions between adjacent stations, as illustrated in Figure 5.8a. The transmitted frame contains the following information:

precedence—Specifies the grand-master precedence.
grandTime—An estimation of the grand-master time.
localTime—A sampling of the station-local time.
linkTime—Derived parameters from the neighbor, returned in a following cycle.

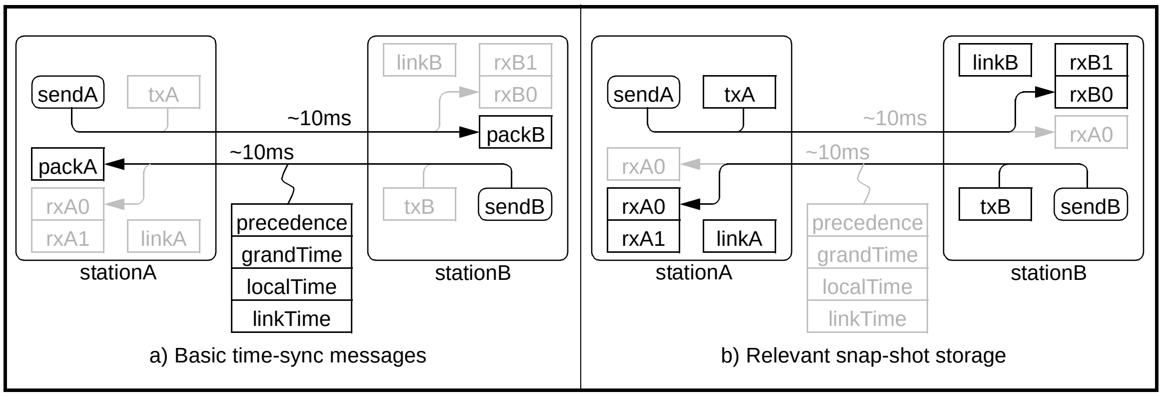


Figure 5.8—Timer snapshot locations

Snapshots are taken when packets are transmitted (illustrated as *txA* and *txB*) and received (illustrated as *rxA* and *rxB*), as illustrated in Figure 5.8b. The transmitted stopshot *txA* is placed into the next frame that is transmitted, as *packetA.localTime*, along with grand-master time *packetA.grandTime* 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.grandTime* 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 5.1. Similarly, *linkB* (the value transmitted from stationB to stationA) is specified by Equation 5.2.

$$linkA = rxA1 - packetFromB.localTime; \quad (5.1)$$
$$linkB = rxB1 - packetFromA.localTime; \quad (5.2)$$

The value of the intermediate span delay is readily derived from these values. At stationA and stationB, these computations are specified by Equation 5.3 and Equation 5.4, respectively.

$$cableDelayComputedAtA = (linkA + packetFromB.delta)/2; \quad (5.3)$$
$$cableDelayComputedAtB = (packetFromA.delta + linkB)/2; \quad (5.4)$$

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

5.6 Rate-difference adjustments

If the absence of rate adjustments, significant *grandTime* errors can accumulate between send-period updates, as illustrated in Figure 5.9. 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.

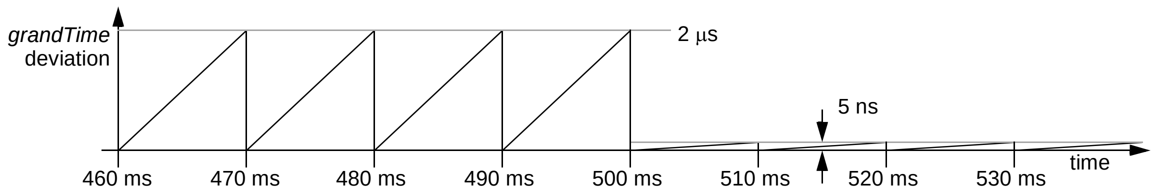


Figure 5.9—Rate-adjustment effects

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 *grandTime* 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.

5.7 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 effectively boundary clocks, since their compensation hardware (that accounts for by pass-through delays) is sufficient to provide a synchronized image of the grand-master time.

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

c) Interfaces that provide locally synchronized timers (such as 802.3-PON and wireless media) have no need to provide hardware shapshots of frame arrival and departure times.

d) 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.

e) 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. Duplex-link state machines

6.1 Overview

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

6.2 MAC-relay interface model

The time-synchronization service model assumes the presence of one or more time-synchronized AVB ports communicating with a MAC relay, as illustrated in Figure 6.1. A received MAC frame is associated with rxSync information, processed within the TimeSync state machine, and passed over the MAC relay. The preference of the relayed frame is determines whether the frame is dropped by the receiving TimeSync state machines or modified and queued for periodic transmission on the receiving PHY.

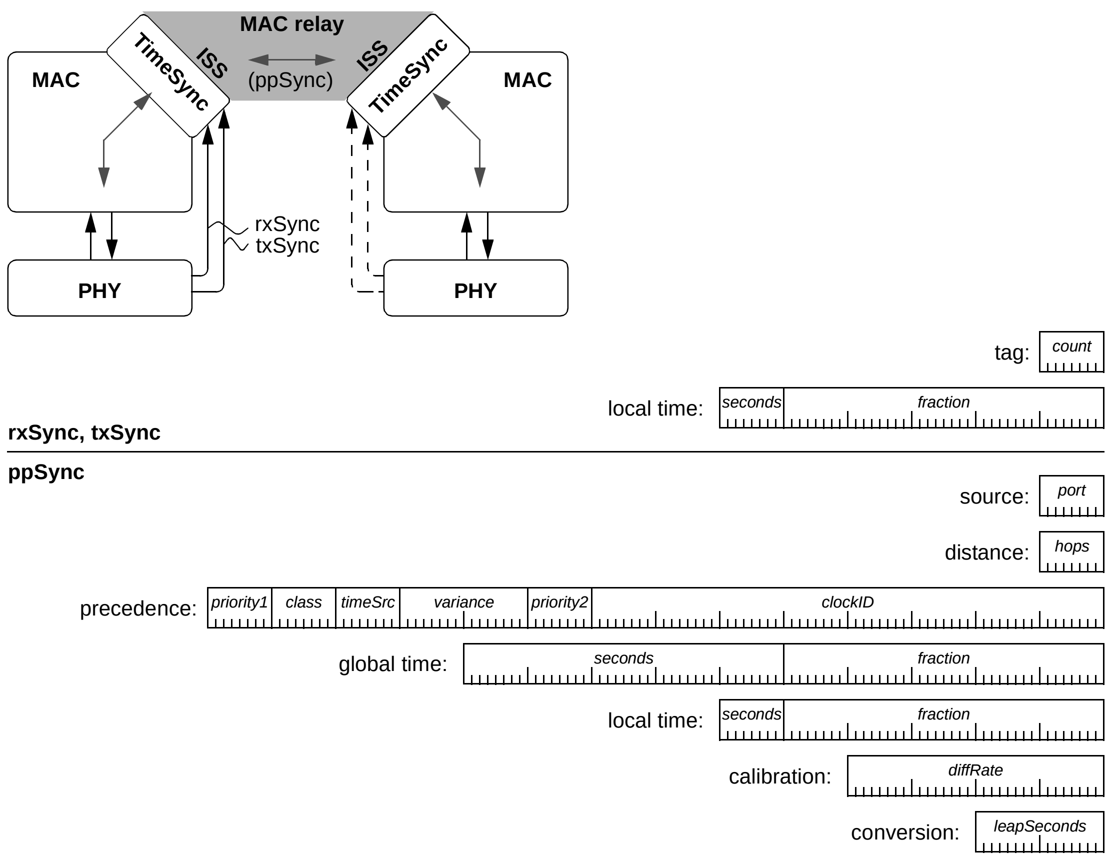


Figure 6.1—AVB service interface model

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

6.3 timeSync frame format

6.3.1 timeSync fields

Clock synchronization (timeSync) frames facilitate the synchronization of neighboring clock-master and clock-slave stations. The frame, which is normally sent at 10ms intervals, includes time-snapshot information and the identity of the network's clock master, as illustrated in Figure 6.2. 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	*frameCount*	— A (sequence number) count of time-sync frames
1	*hopCount*	— Hop count from the grand master
14	*precedence*	— Precedence for grand-master selection
10	*grandTime*	— Transmitter global-time snapshot (1 cycle delayed)
2	*leapSeconds*	— Additional seconds are introduced as time passes
6	*localTime*	— Transmitter local-time snapshot (1 cycle delayed)
4	*linkRate*	— Cumulative rate difference
6	*txTime*	— Opposing link's frame transmission time
6	*rxTime*	— Opposing link's frame reception time
4	*fcs*	— Frame check sequence

Figure 6.2—timeSync frame format

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

6.3.1.1 *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).

6.3.1.2 *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.

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

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

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

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

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

6.3.1.8 *precedence*: A 14-byte field that has specifies precedence in the grand-master selection protocols (see 6.3.3).

6.3.1.9 *grandTime*: An 80-bit field that specifies the grand-master synchronized time within the source station, when the previous timeSync frame was transmitted (see 6.3.5).

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

6.3.1.11 *localTime*: A 48-bit field that specifies the local free-running time within the source station, when the previous timeSync frame was transmitted (see 6.3.6).

6.3.2 *linkRate*: A 32-bit field that specifies the rate difference between the grand-master and local clock.

6.3.2.12 *txTime*: A 48-bit field that specifies when an opposing-link frame was transmitted (see 6.3.6).

6.3.2.13 *rxTime*: A 48-bit field that specifies when an opposing-link frame was received (see 6.3.6).

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

6.3.3 *precedence* subfields

The precedence field includes the concatenation of multiple fields that are used to establish precedence between grand-master candidates, as illustrated in Figure 6.3.

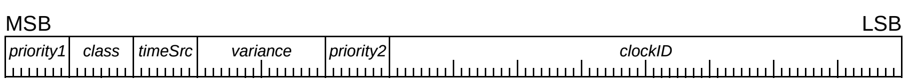


Figure 6.3—*precedence* subfields

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

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

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

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

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

6.3.3.6 *clockID*: A 64-bit globally-unique field that ensures a unique precedence value for each potential grand master, when {*priority1*, *class*, *variance*, *priority2*} fields happen to have the same value (see 6.3.4).

6.3.4 *clockID* subfields

The 64-bit *clockID* field is a unique identifier. For stations that have a uniquely assigned 48-bit *macAddress*, the 64-bit *clockID* field is derived from the 48-bit MAC address, as illustrated in Figure 6.4.

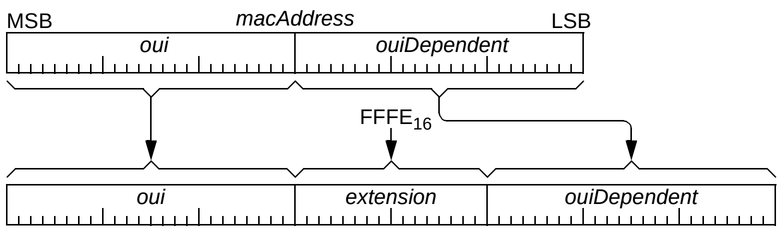


Figure 6.4—*clockID* format

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

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

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

6.3.5 Global-time subfield formats

Time-of-day values within a frame are based on seconds and fractions-of-second values, consistent with IETF specified NTP[B7] and SNTP[B8] protocols, as illustrated in Figure 6.5.

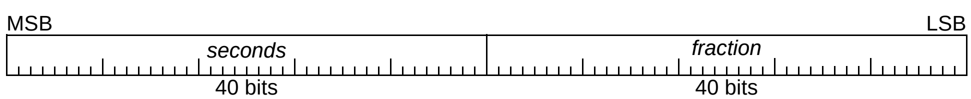


Figure 6.5—Global-time subfield format

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

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

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

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

6.3.6 Local time formats

The local-time values within a frame are based on a fractions-of-second value, as illustrated in Figure 6.6. The 40-bit *fraction* field specifies the time offset within the *second*, in units of 2^{-40} second.

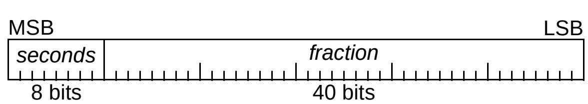


Figure 6.6—Local time format

6.4 ppSync frame format

The relayed ppSync (port-to-port clock-synchronization) frame is a variant of the timeSync frame, as illustrated in Figure 6.7. The gray boxes represent fields that are different from the received timeSync frame.

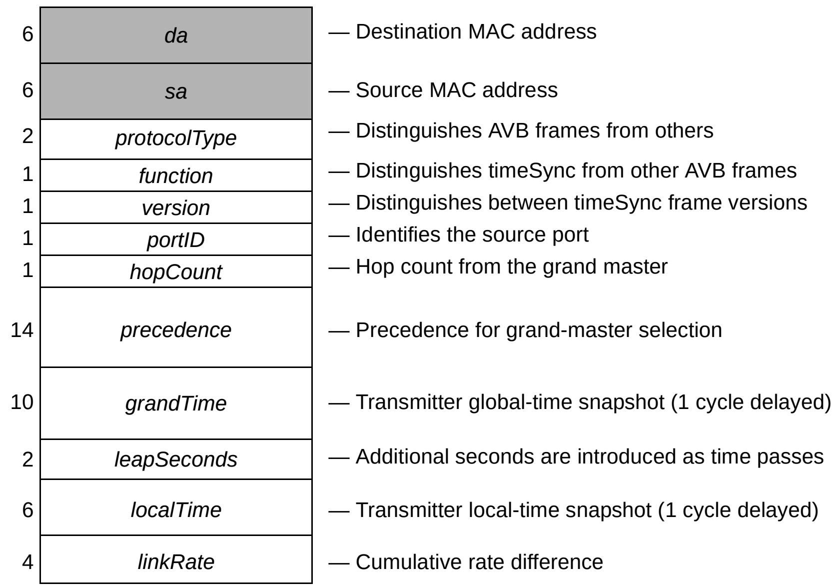


Figure 6.7—ppSync frame format

The 48-bit ***da*** (destination address), 48-bit ***sa*** (source address) field, 16-bit ***protocolType*,** 8-bit ***function***, 8-bit ***version***, 8-bit ***hopCount***, 14-byte ***precedence***, 80-bit ***grandTime***, 16-bit ***leapSeconds***, 48-bit ***localTime***, 4-byte ***linkRate***, and 32-bit ***fcs*** (frame check sequence) field are specified in 6.3.

6.4.1 *portID*: An 8-bit field that identifies the port that sourced the ppSync frame.

6.4.2 Clock-synchronization intervals

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

Clock synchronization involves the processing of periodic events. Multiple time periods are involved, as listed in Table 6.1. 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 6.1—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.

6.5 TimeSync state machine

6.5.1 Function

The TimeSync state machine is responsible for monitoring its port's rxSync/txSync indications, sending MAC-relay frames, and receiving MAC-relay frames. The sequencing of this state machine is specified by Table 6.2; details of the computations are specified by the C-code of Annex F.

6.5.2 TimeSync 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_ISS—The queue identifier associated with MAC frames sent into the relay.
    Q_RX_MAC—The queue identifier associated with the received MAC frames.
    Q_RX_SYNC—The queue identifier associated with rxSync, sent from the lower levels.
    Q_TX_ISS—The queue identifier associated with frames sent from the relay.
    Q_TX_MAC—The queue identifier associated with frames sent to the MAC.
    Q_TX_SYNC—The queue identifier associated with txSync, sent from the lower levels.
T10ms
  A constant the represents a 10 ms value.
T50ms
  A constant the represents a 50 ms value.
T100ms
  A constant the represents a 100 ms value.

6.5.3 TimeSync state machine variables

curentTime
  A shared value representing current time. There is one instance of this variable for each station. Within the state machines of this standard, this is assumed to have two components, as follows:
    seconds—An 8-bit unsigned value representing seconds.
    fraction—An 40-bit unsigned value representing portions of a second, in units of 2^{-40} second.
frame
  The contents of a MAC-supplied frame.
info
  A contents of a lower-level supplied time-synchronization poke indication, including the following:
    localTime—The value of *currentTime* associated with the last timeSync packet arrival.
    frameCount—The value of the like-named field within the last timeSync packet arrival.
port
  A data structure containing port-specific information comprising the following:
    rxFrame—The last received frame.
    rxFrameCount—The value of *frameCount* within the last received 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.
    rxLastTime—The *currentTime* value when syncFrame was received, used for timeouts.
    rxSyncFrame—The value of the previously observed timeSync frame.

    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.
    txLastTime—The value of *currentTime* when timeSync frame was enqueued, used for pacing.

6.5.4 TimeSync 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.
FrameToMac(*pPtr*, *frame*)
Transfers the frame to the MAC, as specified by the C code of Annex F.
MacToRelay(*pPtr*, *frame*, *ok*)
Depends on the value of ok, as specified by the C code of Annex F:
TRUE—Modifies and transfers the received frame to the MAC relay.
FALSE—Transfers a dummy timeout-indicated frame to the MAC relay.
RelayToFrame(*pPtr*, *frame*)
Copies a high-preference MAC-relay frame to port storage, as specified by the C code of Annex F. (Low preference MAC-relay frames are simply discarded.)
TimeSyncFrame(*frame*)
Checks the frame contents to identify timeSync frame.
TRUE—The frame is a timeSync frame.
FALSE—Otherwise.

6.5.5 TimeSync state machine table

The TimeSync state machine includes a media-dependent timeout, which effectively disconnects a clock-slave port in the absence of received timeSync frames, as illustrated in Table 6.2.

Table 6.2—TimeSync state machine table

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

START	(info = Dequeue(Q_RX_SYNC)) != NULL	1	port.rxSnapShot1 = port.rxSnapShot0;
			port.rxSnapShot0 = info.localTime;
			port.rxSnapCount = info.frameCount;
				PASS

	(frame = Dequeue(Q_RX_MAC)) != NULL	2	—	TEST
	(currentTime – port.txLastTime) > T10ms	3	port.txLastTime = currentTime;	SEND
	(currentTime – port.rxLastTime) > T50ms	4	Enqueue(Q_RX_ISS, MacToRelay(&port, port.rxFrame, LATE)); port.rxLastTime = currentTime;	START
	(frame = Dequeue(Q_TX_ISS)) != NULL	5	RelayToFrame(&port, frame);	
	(info = Dequeue(Q_TX_SYNC)) != NULL	6	port.txSnapShot = info.localTime; port.txSnapCount = info.frameCount;	
	—	7	—	
TEST	!TimeSyncFrame(frame)	8	Enqueue(Q_RX_ISS, frame);	START
	frame.hopCount == 255	9	—	
	frame.count != port.rxFrameCount+1	10	port.rxFrameCount = frame.count;	
	—	11	port.rxFrame = frame; port.rxFrameCount = frame.count;	PASS
PASS	port.rxFrame.frameCount != port.rxSnapCount	12	—	START
	—	13	Enqueue(Q_RX_ISS, MacToRelay(&port, port.rxFrame, !LATE)); port.rxLastTime = currentTime;	
SEND	port.txFrame.hopCount == LAST_HOP	14	—	START
	—	15	Enqueue(Q_TX_MAC, FrameToMac(&port, port.txFrame));	

Row 6.2-1: Update snapshot values on timeSync frame arrival.
Row 6.2-2: Initiate inspection of frames received from the lower-level MAC.
Row 6.2-3: Transmit periodic timeSync frames.
Row 6.2-4: The absence of timeSync frames generates a pseudo clock-master indication.
Row 6.2-5: Save frames received from other ports.
Row 6.2-6: Process time-snapshot information.
Row 6.2-7: Wait for the next change-of-state.

Row 6.2-8: The non-timeSync frames are passed through.
Row 6.2-10: Non-sequential frames are discarded.
Row 6.2-11: Sequential timeSync frames are processed.

Row 6.2-12: Inhibit processing when the frame and poke counts are different.
Row 6.2-13: Invoke common-entity processing when the frame and poke counts are the same.

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.

Annexes

Annex A

(informative)

Bibliography

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

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

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

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

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

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

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

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

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

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

Annex B

(informative)

Time-scale conversions

The synchronized value of *grandTime* (grand-master time) is based on the Precision Time Protocol (PTP). Time is measured in international seconds since the start of January 1, 1970 Greenwich Mean Time (GMT). Other representations of time can be readily derived from the values of *grandTime* and 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 = grandTime + offset;
GPS	global positioning satellite	2	–315 964 819	
UTC	Coordinated Universal Time	3	TBD	time = grandTime + 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.

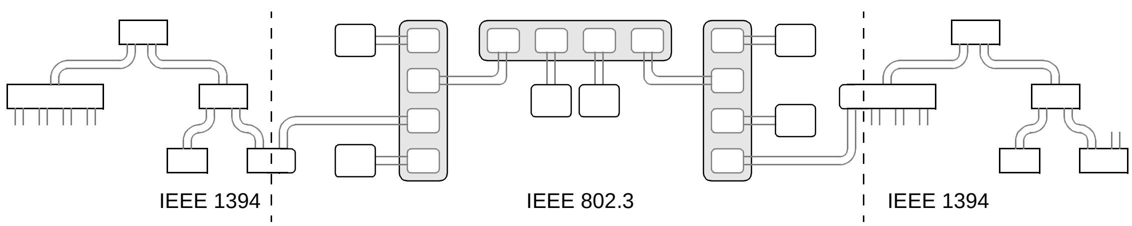


Figure C.1—IEEE 1394 leaf domains

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.

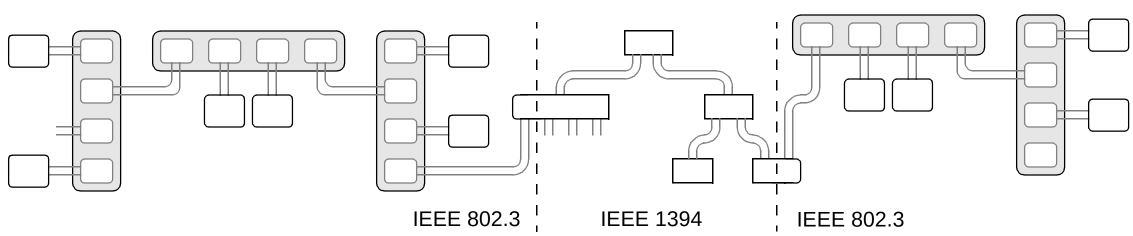


Figure C.2—IEEE 802.3 leaf domains

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.

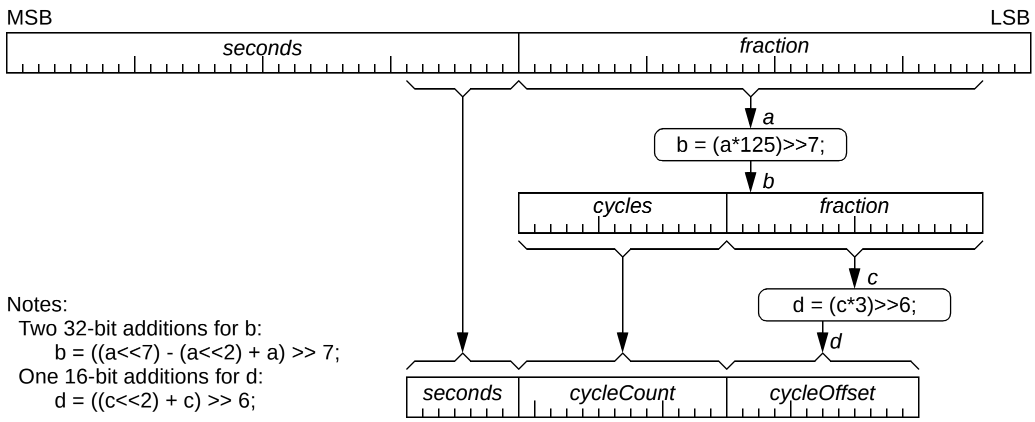


Figure C.3—Time-of-day format conversions

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:

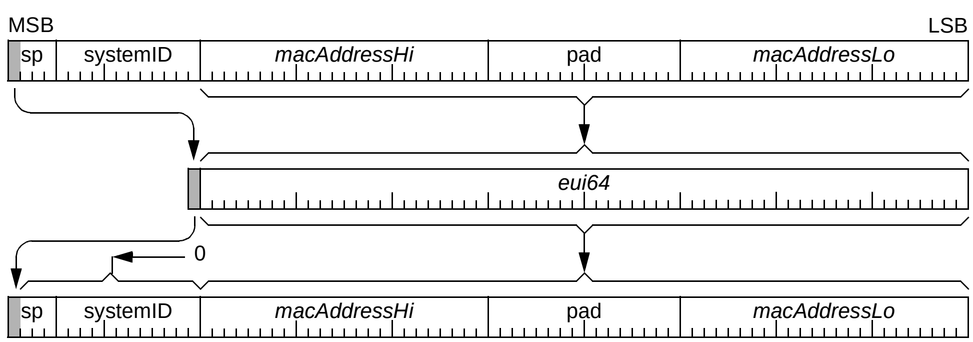


Figure C.4—Grand-master precedence mapping

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, various possible formats are described within this annex.

E.1 Possible time-of-day formats

E.1.1 Extended binary timer formats

The extended-binary timer format is used within this working paper and summarized herein. The 64-bit timer value consist of two components: a 40-bit *seconds* and 40-bit *fraction* fields, as illustrated in Figure 5.1.

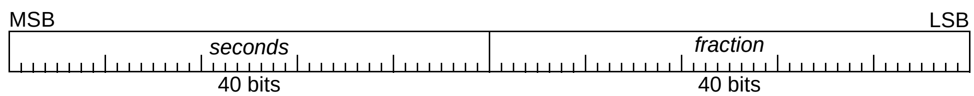


Figure 5.1—Global-time subfield format

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

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

Where:

seconds is the most significant component of the time value.
fraction is the less significant component of the time value.

E.1.2 IEEE 1394 timer format

An alternate “1394 timer” format consists of *secondCount*, *cycleCount*, and *cycleOffset* fields, as illustrated in Figure E.2. For such fields, the 12-bit *cycleOffset* field is updated at a 24.576MHz rate. The *cycleOffset* field goes to zero after 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.

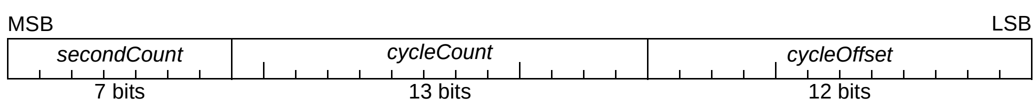


Figure E.2—IEEE 1394 timer format

E.1.3 IEEE 1588 timer format

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

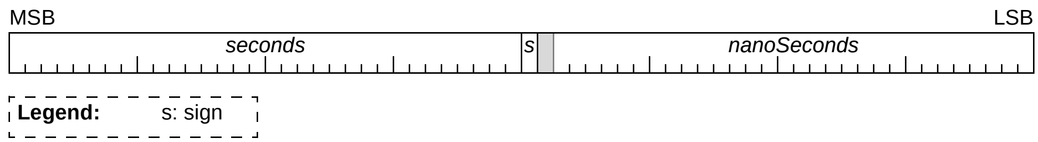


Figure E.3—IEEE 1588 timer format

E.1.4 EPON timer format

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

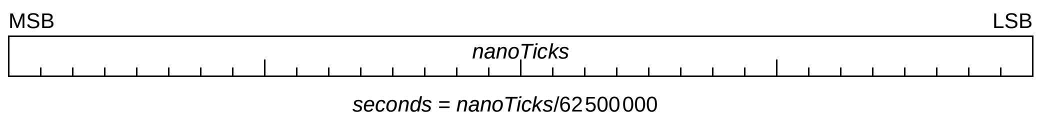


Figure E.4—EPON timer format

Annex F

(informative)

C-code illustrations

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

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

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

```
// *********************************************************************************************************************************
//                                                                                                 1         1         1         1
//       1         2         3         4         5         6         7         8         9         0         1         2         3
//345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012

// 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, y) ((x) >= y ? y-1 : ((x) <= -y ? -(y-1) : (x)))                               // Clip to |x| < y
#define LAST_HOP 255
#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 frameCount[1];                                                               // Transmit count (sequence number)
    uint8_t hopCount[1];                                                                 // Hop-count from the grand master
    uint8_t precedence[14];                                                              // Grand-master precedence
    uint8_t grandTime[10];                                                               // Grand-master time  (for last frame)
    uint8_t leapSeconds[2];                                                              // Leap seconds compensation
    uint8_t localTime[6];                                                                // Local-station time (for last frame)
    uint8_t linkRate[4];                                                                 // Grand-master rate difference
    uint8_t txTime[6];                                                                   // Apparent link delay
    uint8_t rxTime[6];                                                                   // Apparent link delay
    uint8_t fcs[4];                                                                      // CRC integrity check
} TimeSync;

typedef struct                                                                           // Port entity state
{
    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 rxSnapShot0;                          // This frame’s arrival time
    LocalTime rxSnapShot1;                          // Past frame’s arrival time
    LocalTime rxTxTime;                             // Opposing frame-transmit
    LocalTime rxRxTime;                             // Opposing frame-received
    TimeSync  rxSyncFrame;                          // Received timeSync frame.

    LocalTime rxThisLast;                           // The last local-time snapshot
    LocalTime rxThatLast;                           // The last neighbor-time snapshot

    int32_t   thisRate;                             // Rate difference from upstream
    int32_t   diffRate;                             // Rate difference from grand-master

    uint8_t   txFrameCount;                         // The timeSync frame count.
    LocalTime txSendTime;                           // TimeSync activation time
    LocalTime txSnapShot;                           // Transmit frame snapshot
    TimeSync  txSyncFrame;                          // To be transmitted timeSync.
    BigNumber bestPreference;                       // Grand-master preference
} 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

// Basic interface routines
TimeSync    MacToRelay(PortData *, TimeSync, Boolean ok);
void        RelayToFrame(PortData *, TimeSync);
TimeSync    FrameToMac(PortData *, TimeSync);

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

```
// Other routines
Precedence  FieldsToPrecedence(uint8_t, uint8_t, uint16_t, uint8_t, uint64_t);
BigNumber   FrameToValue(TimeSync *, uint8_t *, uint16_t, Boolean);
BigNumber   FormPreference(BigNumber, uint8_t, uint8_t);
BigNumber   LongToBig(LocalTime);
Port        PreferenceToPort(Preference);
Hops        PreferenceToHops(Preference);
TimeSync    PsTx(PortData *pPtr);
void        ValueToFrame(BigNumber, TimeSync *, uint8_t *, uint16_t);


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

TimeSync
MacToRelay(PortData *pPtr, TimeSync rxFrame, Boolean late)
{
    TimeSync result, *rxPtr, *txPtr;
    BigNumber grandTime, newTime;
    int64_t thisDelta, thatDelta, localTime, rxTxTime, rxRxTime;
    int64_t thisDelay, thatDelay, nextDelay, cableDelay;
    int32_t linkRate, diffRate;
    uint8_t hopCount;

    assert(pPtr != NULL);
    rxPtr = &rxFrame;
    txPtr = &result;
    result = rxFrame;
    if (late)
    {                                                                                   // A timeout is signaled
        LongToFrame(LAST_HOP, txPtr, hopCount);                                         // by out-of-range hopCount.
        return(result);
    }

    hopCount =  FieldToUnsigned(rxPtr, hopCount).lower;                                 // Hops from the GM station.
    grandTime = FieldToSigned(rxPtr,   grandTime);                                      // Frame transmission time.
    localTime = FieldToSigned(rxPtr,   localTime).lower;                                // Frame transmission time.
    linkRate =  FieldToSigned(rxPtr,   linkRate).lower;                                 // Cumulative rate differences.
    rxTxTime =  FieldToSigned(rxPtr,   txTime).lower;                                   // Opposing link delay.
    rxRxTime =  FieldToSigned(rxPtr,   rxTime).lower;                                   // Opposing link delay.

    assert(hopCount != 255);
    thisDelta = (pPtr->rxSnapShot1 - pPtr->rxThisLast);                                 // Wait a longer interval before
    if (thisDelta >= T100ms)                                                            // computing the rate difference.
    {
        thatDelta = localTime - pPtr->rxThatLast;                                       // Neighbor's timer changes
        pPtr->thisRate = DivideHi((thisDelta - thatDelta) * SCALE, thatDelta);          // Save rate difference.
        pPtr->rxThisLast = pPtr->rxSnapShot1;                                           // The local-time snapshot
        pPtr->rxThatLast = localTime;                                                   // The grand-master snapshot
    }
    diffRate = CLIP(pPtr->thisRate + linkRate, ((uint64_t)2 << 31));                    // Cumulative rate difference.
```

```
    thisDelay = (rxTxTime - pPtr->rxSnapShot1);                                     // Looped-response delay
    thatDelay = (localTime - rxRxTime);                                             // Remote-response delay
    nextDelay = thisDelay - (thatDelay + (MultiplyHi(thatDelay, diffRate) / SCALE));  // Computed cable delay
    cableDelay = MIN(0, nextDelay);                                                 // is never negative.
    grandTime = BigAddition(grandTime, LongToBig(cableDelay));                      // Adjust time with cable delay.
    hopCount = hopCount+1;                                                          // The GM distance.

    pPtr->rxTxTime = localTime;                                                     // Opposing-link transmit time
    pPtr->rxRxTime = pPtr->rxSnapShot1;                                             // Opposing-link received time
    LongToFrame(pPtr->portID,      txPtr, frameCount);                              // The frame source.
    LongToFrame(hopCount+1,        txPtr, hopCount);                                // The GM distance.
    LongToFrame(pPtr->rxSnapShot1, txPtr, localTime);                               // The snapshot localTime value.
    BigToFrame(newTime,            txPtr, grandTime);                               // The GM time (of last frame).
    LongToFrame(diffRate,          txPtr, linkRate);                                // The clock-rate difference.
    return(result);
}

void
RelayToFrame(PortData *pPtr, TimeSync rxFrame)
{
    Preference sentPreference, bestPreference;
    Precedence precedence;
    uint64_t sa;
    TimeSync *rxPtr;
    uint8_t hopCount, portID;
    Boolean best, none, same;

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

    portID =     FieldToUnsigned(rxPtr, frameCount).lower;                          // Source-port value.
    hopCount =   FieldToUnsigned(rxPtr, hopCount).lower;                            // Hop-count parameter.
    precedence = FieldToUnsigned(rxPtr, precedence);                                // GM precedence value.

    sentPreference = FormPreference(precedence, hopCount, sa);                      // Receive port precedence
    bestPreference = pPtr->bestPreference;
    same = (PreferenceToPort(bestPreference) == portID);                            // This was preferred port.
    best = (BigCompare(sentPreference, bestPreference) <= 0) && (hopCount != LAST_HOP);  // This port is preferred.
    none = (PreferenceToHops(bestPreference) == LAST_HOP);
    if (same || best || none)                                                       // Only the best are taken.
    {
        pPtr->txSyncFrame = rxFrame;                                                // Save the frame
        pPtr->bestPreference = sentPreference;                                      // Update the preference
    }
}

TimeSync
FrameToMac(PortData *pPtr, TimeSync rxFrame)
{
    TimeSync result, *rxPtr, *txPtr;
    GrandTime oldTime, newTime;
    uint64_t delayed, rxTime, txTime;
    int32_t diffRate;
    uint8_t frameCount;
```

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

    oldTime =  FieldToSigned(rxPtr, grandTime);                                   // GM synchronized time.
    rxTime =   FieldToSigned(rxPtr, localTime).lower;                             // Frame received time.
    diffRate = FieldToSigned(rxPtr, linkRate).lower;                              // Cumulative rate differences.

    txTime = pPtr->txSnapShot;                                                    // Frame transmitted time.
    frameCount = pPtr->txFrameCount;                                              // Frame sequence counter.
    delayed = (txTime - rxTime);                                                  // Time since observation
    delayed += MultiplyHi(delayed, diffRate) / SCALE;                             // Scaled rate difference
    newTime = BigAddition(oldTime, LongToBig(delayed));                           // Computed GM time

    LongToFrame(NEIGHBOR,           txPtr, da);                                   // Neighbor multicast address.
    LongToFrame(pPtr->macAddress,   txPtr, sa);                                   // This port's MAC address.
    LongToFrame(AVB_TYPE,           txPtr, protocolType);                         // The AVB protocol.
    LongToFrame(TIME_SYNC,          txPtr, function);                             // The timeSync frame in AVB.
    LongToFrame(VERSION_0,          txPtr, version);                              // This version number.
    LongToFrame(frameCount,         txPtr, frameCount);                           // The sequence number.
    BigToFrame(newTime,             txPtr, grandTime);                            // The grandTime at local txTime.
    LongToFrame(pPtr->rxTxTime,     txPtr, txTime);                               // Local time (of last frame).
    LongToFrame(pPtr->rxRxTime,     txPtr, rxTime);                               // Local time (of last frame).

    pPtr->txFrameCount = frameCount + 1;                                          // Increment for next frame.
    return(result);                                                               // Return frame for transmission.
}


// ************************************************************************************
// 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                                                     // x = (a *  b) >> 32,
MultiplyHi(uint64_t value2, int32_t value1)                 // for all {a,b} values.
{
    int64_t upper, lower;

    upper = (value2 >> 32) * value1;                        // Add the upper
    lower = ((value2 & (uint64_t)0XFFFFFFFF) * value1) >> 32; // to the lower
    return(upper + lower);                                  // for the result.
}

int64_t                                                     // x = (a << 32)/b, for
DivideHi(int64_t a, int64_t b)                              // for b < 2**48
{
    int64_t sum, rem;
    Boolean flip;

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

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