6. Frame formats

6.1 ClassA frames

6.1.1 ClassA frame fields

A classA frame differs from other frames in the format of its multicast *da* (destination address), as illustrated in Figure 6.1.

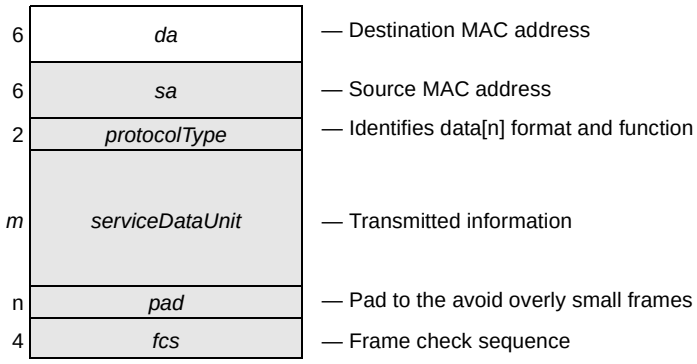


Figure 6.1—ClassA frame formats

6.1.1.1 *da*: A 6-byte (destination address) field that specifies a multicast address associated with the stream.

6.1.1.2 *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 3.11) as specified in 9.2 of IEEE Std 802-2001.

6.1.1.3 *protocolType*: A 16-bit field contained within the payload. When the value of *protocolType* is greater than or equal to 1536 (600_{16}) the *protocolType* field indicates the nature of the MAC client protocol (type interpretation), selecting from values designated by the IEEE Type Field Register. When less than 1536 (0_{16}–$5FF_{16}$), the *protocolType* is interpreted as the length of the frame (length interpretation). The length and type interpretations of this field are mutually exclusive.

6.1.1.4 *serviceDataUnit*: An m-byte field the contains the service data unit provided by the client.

6.1.1.5 *pad*: If the sum of the other field lengths is less than 64 bytes, then the number of zero-valued *pad* bytes are sufficient to make a 64-byte frame. Otherwise, the *pad* field is not present.

6.1.1.6 *fcs*: A 4-byte (frame check sequence) field whose 32-bit CRC covers the frame's content.

6.2 timeSync frame format

6.2.1 timeSync fields

Clock synchronization (timeSync) frames facilitate the synchronization of neighboring clock span-master and clock span-slave stations. The frame, which is normally sent once each isochronous cycle, includes time-snapshot information and the identity of the network's clock master, as illustrated in 6.2. The gray boxes represent physical layer encapsulation fields that are common across all Ethernet frames.

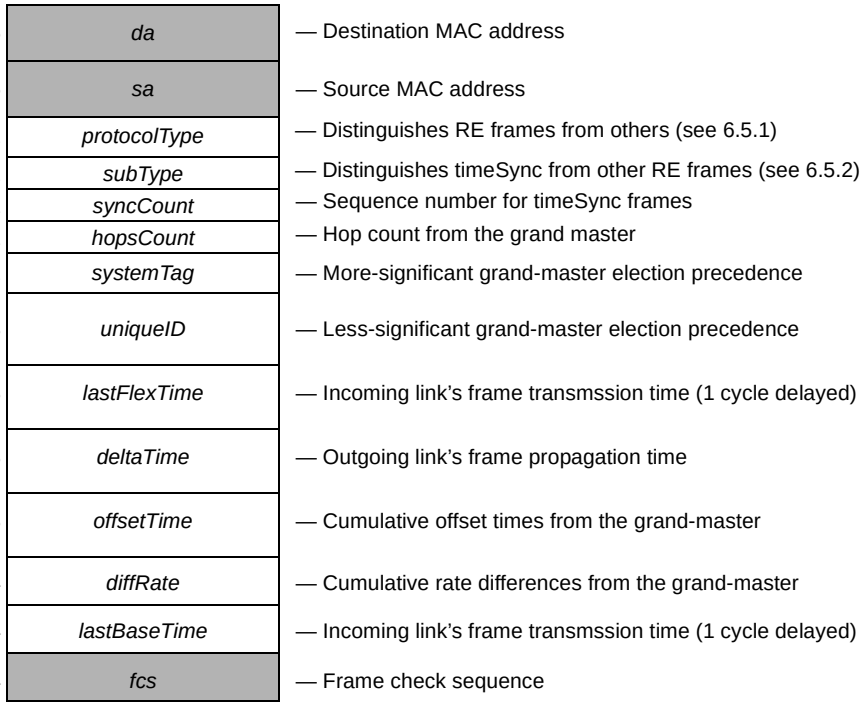


Figure 6.2—timeSync frame format

6.2.1.1 *da*: A 48-bit (destination address) field that specifies the station(s) for which the frame is intended. The *da* field contains either an individual or a group 48-bit MAC address (see 3.11), as specified in 9.2 of IEEE Std 802-2001.

6.2.1.2 *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 3.11), as specified in 9.2 of IEEE Std 802-2001.

6.2.1.3 *protocolType*: A 16-bit field contained within the payload that identifies the format and function of the following fields (see 6.5.1).

6.2.1.4 *subType*: A 16-bit field that identifies the format and function of the following fields (see 6.5.2).

6.2.1.5 *syncCount*: An 8-bit field that is incremented on each timeSync frame transmission.

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

6.2.1.7 *systemTag*: A 16-bit field that has highest precedence in the grand-master selection protocols.

6.2.1.8 *uniqueID*: A 64-bit field that specifies the precedence of the grand clock master, specified in 6.2.3.

6.2.1.9 *lastFlexTime*: A 64-bit field that specifies the time within the source station when the previous timeSync frame was transmitted. The format of this field is specified in 6.2.4.

6.2.1.10 *deltaTime*: A 64-bit field that specifies the differences between timeSync receive and transmit times, as measured on the opposing link. The format of this field is specified in 6.2.4.

6.2.1.11 *offsetTime*: A 64-bit field that specifies the offset time within the source station. The format of this field is specified in 6.2.4.

6.2.1.12 *diffRate*: A 32-bit field that specifies the *diffRate* value within the source station.

6.2.1.13 *lastBaseTime*: A 32-bit field that specifies the *timer1* value within the source station when the previous timeSync frame was transmitted.

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

6.2.2 *systemTag* subfields

The format of the 16-bit *systemTag* field is based on the format of the spanning tree protocol precedence value, as illustrated in Figure 6.3.

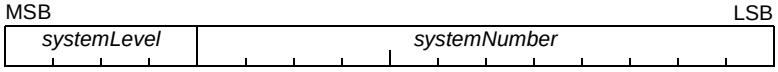


Figure 6.3—*systemTag* subfields

6.2.2.1 *systemLevel*: A 4-bit field that comprise a settable priority component that permits the relative priority of bridges to be managed.

6.2.2.2 *systemNumber*: A 12-bit field that comprise a locally assigned system identifier extension. (The term *systemID* is equivalent to ‘system ID’, as specified within IEEE Std 802.1D-2004.)

6.2.3 *uniqueID* fields

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

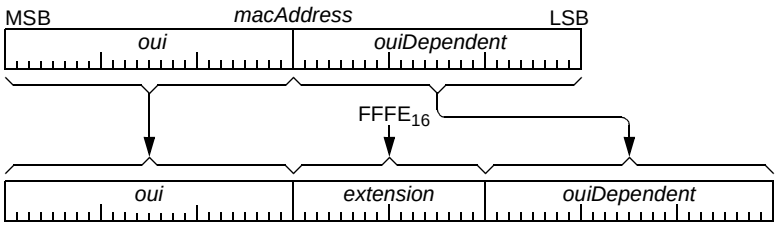


Figure 6.4—*uniqueID* format

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

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

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

6.2.4 Time field formats

Time-of-day values within a frame are specified by 64-bit values, consistent with IETF specified NTP[B8] and SNTP[B9] protocols. These 64-bit values consist of two components: a 32-bit *seconds* and 32-bit *fraction* fields, as illustrated in Figure 6.5.

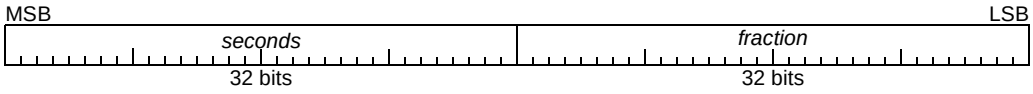


Figure 6.5—Complete seconds timer format

6.2.4.1 *seconds:* A 32-bit field that specifies time in seconds.

6.2.4.2 *fraction*: A 32-bit field that specified time offset within the second, in units of 2^{-32} second.

The concatenation of 32-bit *seconds* and 32-bit *fraction* field specifies a 64-bit *time* value, as specified by Equation 6.1.

$$time = seconds + (fraction / 2^{32}) \quad (6.1)$$

Where:

seconds is the most significant component of the time value (see Figure 6.5).
fraction is the less significant component of the time value (see Figure 6.5).

6.3 Subscription frame

6.3.1 Subscription frame fields

Subscription frames contain channel-acquisition information, as illustrated in Figure 6.6.

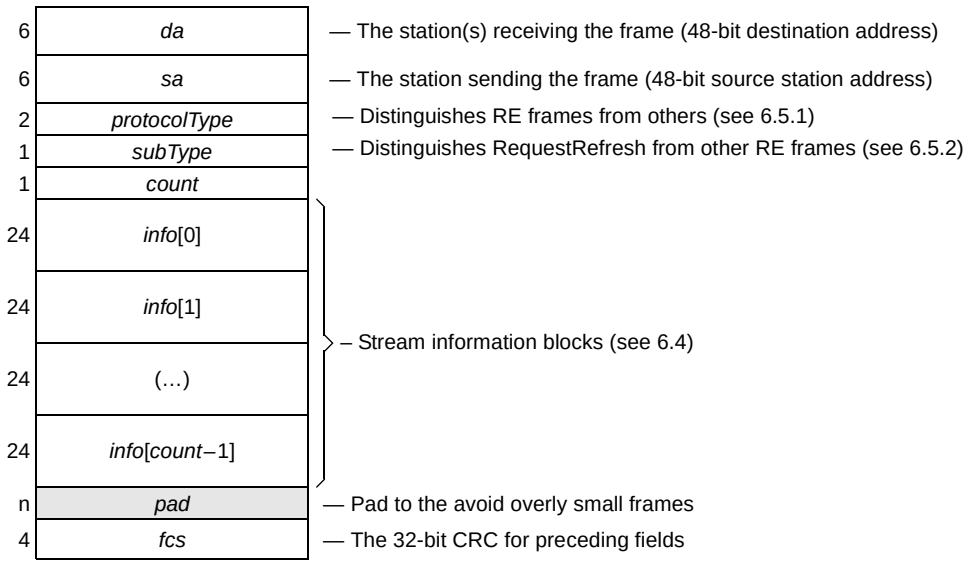


Figure 6.6—Subscription frame format

6.3.1.1 *da*: A 6-byte (destination address) field that normally specifies the destination address for the frame transmission, with unicast and multicast forms.

6.3.1.2 *sa*: A 6-byte (source address) field that normally specifies the source address for the frame transmission. If a bridge is present between the frame and its associated listener, the *sa* value identifies the bridge.

6.3.1.3 *protocolType*: A 2-byte field that normally specifies the frame length, or the format and function of the following fields (excluding the 4-byte *fcs* field). This RE assigned value distinguishes its frame formats from others (see 6.5.1).

6.3.1.4 *subType*: A 1-byte field that distinguishes the ResponseError frame from other frames defined within this working paper.

6.3.1.5 *count*: A 1-byte field that specifies the number of elements within the following *info*-block array.

6.3.1.6 *info*: A 24-byte array element that provides listener subscription information (see 6.4).

6.3.1.7 *pad*: If the sum of the other field lengths is less than 64 bytes, then the number of zero-valued *pad* bytes are sufficient to make a 64-byte frame. Otherwise, the *pad* field is not present.

6.3.1.8 *fcs*: The 4-byte (frame check sequence) field whose 32-bit CRC covers the frame's content. For RE content frames, the standard definition applies.

6.4 Common *info* field format

Many frame transports an array of one or more *info*[ ] fields, whose content is illustrated in Figure 6.7.

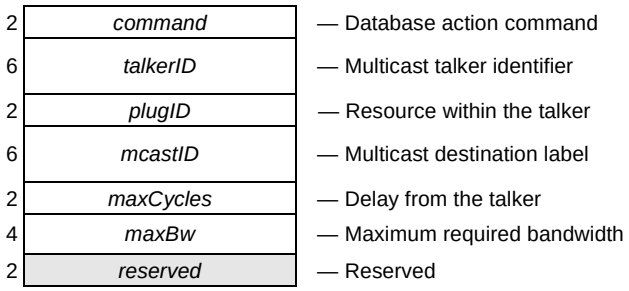


Figure 6.7—Common *info* field format

6.4.1 *command*: A 2-byte field that differentiates between database-update actions.

6.4.2 *talkerID*: A 6-byte field that identifies the stream's talker.

6.4.3 *plugID*: A 16-bit field that specifies the plug identifier within the talker.

The concatenation of the 48-bit *talkerID* and 16-bit *plugID* fields forms a 64-bit *streamID* that uniquely identifies the classA multicast stream.

6.4.4 *mcastID*: A 6-byte (multicast identifier) field that routes frames betwee the talker and audience.

6.4.5 *maxCycles*: A 2-byte field that is updated by bridges, as the RequestRefresh flows from the talker to the listener, allowing the maximum number of delay cycles between the talker and listener stations to be known to the talker.

6.4.6 *maxBw*: A 4-byte field that specifies the level of negotiated classA bandwidth, measured in bytes of per-cycle content.

6.4.7 *reserved*: A 2-byte zero-valued field that is ignored.

6.5 Unique identifier values

6.5.1 *protocolType* identifier

NOTE—The following *protocolType*-assignment text will ultimately be updated with assigned values.

The timeSync (see 6.2) and subscription (see 6.3) frames are distinguished from other frames by their 16-bit distinct *protocolType* value, as illustrated in Figure 6.8. The following 1-byte *subType* field further distinguishes between these uses (see 6.5.2).

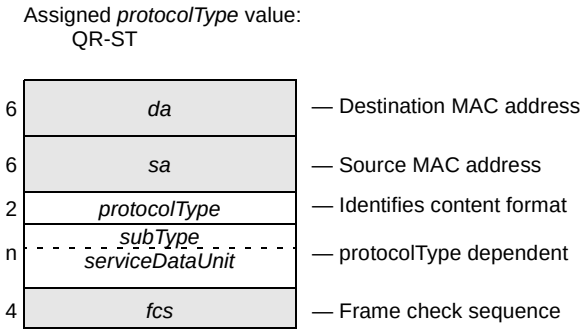


Figure 6.8—*protocolType* field value

6.5.2 *subType* identifier

Distinct *subType* identifiers distinguish between RE frame types, as specified by Table 6.1.

Table 6.1—Assigned *subType* identifiers

Value	Name	Row	See	Description
TBD	CLOCK_SYNC	1	6.2	Demarcates boundaries between isochronous cycles.
192-255	E1394	2	C.2.2	Encapsulated IEEE 1394 packet (or portion of 1394 packet)

7. Timer synchronization

7.1 Synchronized time-of-day timers

7.1.1 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) Point-to-point transmit/receive duplex connections are provided.

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

7.1.2 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 A/V 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.

7.1.3 Strategies

Strategies used to meet these objectives include the following:

a) Precision is achieved by calibrating and adjusting *timeOfDay* 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) Symmetric. Clock-master/clock-slave computations are similar (only slave results are saved).
   4) Periodic. Messages are sent periodically, rather than in timely response to other requests.
   5) Frequent. Frequent (every 10 ms) interchanges reduces needs for precise clocks.

c) Balanced functionality.

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

7.1.4 Timer synchronization services

Clock synchronization involves the transmission and reception of timeSync frames interchanged between adjacent-span stations, using the state machines defined within this clause. When considered as a whole, these provide the following services:

a) Election. The grand clock master is elected from among the grand-clock-master capable stations.

b) Isolation. Timeouts identify the boundaries, beyond which RE services are not supported.

c) Clock-sync. Clock-slave stations are synchronized to the grand master station's time reference.

7.1.5 Grand-master selection

7.1.5.1 Grand-master selection

Clock synchronization involves streaming of clock-synchronization information from a grand-master timer to one or more slave timers. Although primarily intended for non-cyclical physical topologies (see Figure 7.1a), the synchronization protocols also function correctly on cyclical physical topologies (see Figure 7.1b), by activating only a non-cyclical subset of the physical topology.

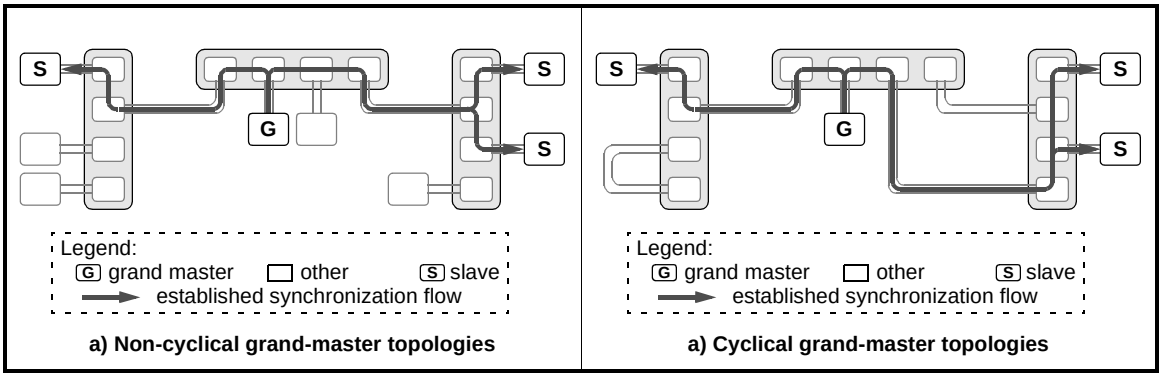


Figure 7.1—Timer synchronization flows

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 RE-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.

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.

7.1.5.2 Communicated preference values

The grand-master station observes that its precedence is better than values received from its neighbors, as illustrated in Figure 7.2a. 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 7.2b. To avoid cyclical behaviors, a *hopsCount* value is associated with preference values and is incremented before the best-precedence value is communicated to others.

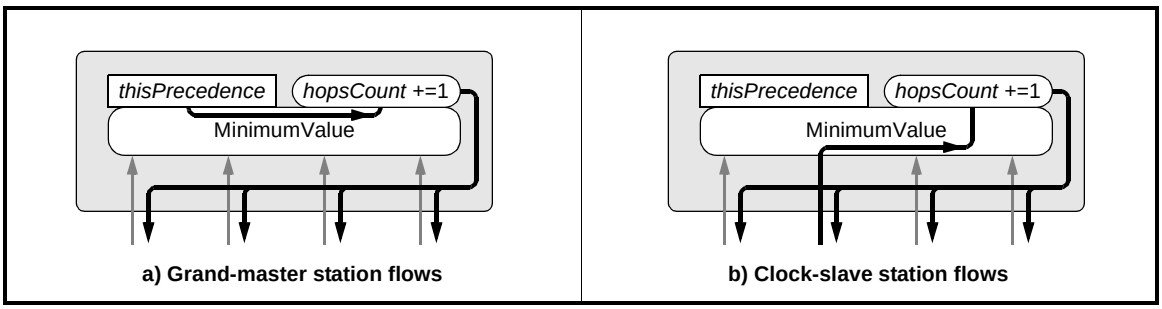


Figure 7.2—Grand-master precedence flows

The grand-master selection precedence includes multiple components, listed and described below (see 7.1.8). The *portTag* value is only needed within a bridge and is therefore not transmitted between stations.

a) *systemTag*. A changeable value that is associated with each grand-master capable station. This value can specify grand-master preferences (e.g., a home gateway may be preferred).
b) *uniqueID*. A unique value associated with each station, typically based on its MAC address. This value is used as a tie breaker, when two contenders have identical *systemTag* values.
c) *hopsCount*. A value that is incremented when passing through stations. This is the tie breaker, when two ports receive identical *systemTag*:*uniqueID* values.
d) *portTag*. A changeable value that is associated with each port on a grand-master capable station. This is the tie breaker, when two ports receive identical *systemTag*:*uniqueID*:*hopsCount* values.

7.1.6 Clock-synchronization agents

Clock-synchronization information conceptually flows from a grand-master station to clock-slave stations, as illustrated in Figure 7.3a. A more detailed illustration shows pairs of synchronized clock-master and clock-slave components, as illustrated in Figure 7.3b.

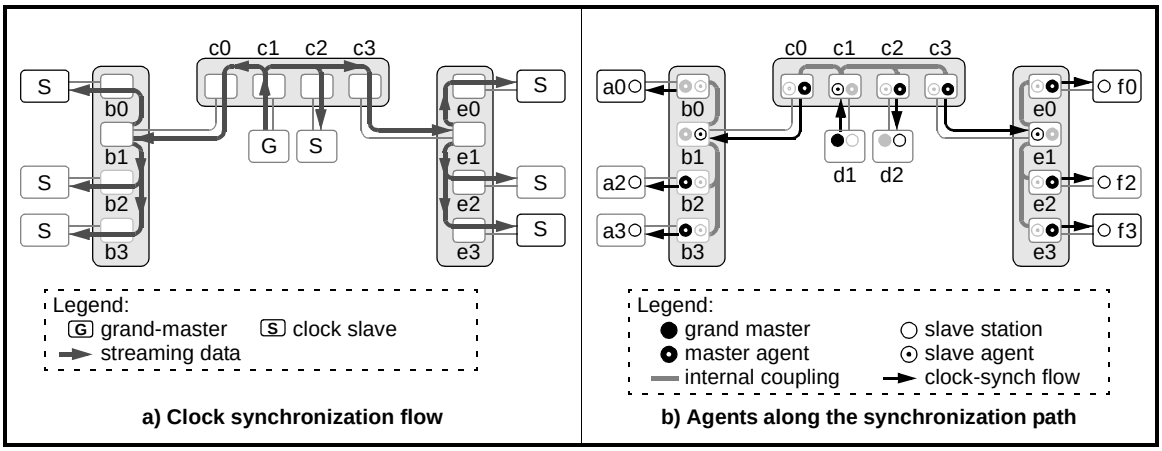


Figure 7.3—Hierarchical flows

7.1.7 Clock-synchronized pairs

Each bridge port provides clock-master and clock-slave agents, although both are never simultaneously active. External communications (see 7.3b) synchronize clock-slaves to clock-masters, as listed in Table 7.1.

Table 7.1—External clock-synchronization pairs

Grand master	Clock master agent	Clock slave agent	Clock slave	Type of synchronization
d1	–	c1	–	Station-to-bridge
–	c0	b1	–	Bridge-to-bridge
–	c3	e1	–	
–	b0	–	a0	Bridge-to-station
–	b2	–	a2	
–	b3	–	a3	
–	c2	–	d2	
–	e0	–	f0	
–	e2	–	f2	
–	e3	–	f3	

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. However, bridge-internal port-to-port synchronization protocols are implementation-dependent and beyond the scope of this working paper.

Within a clock-slave, precise time synchronization involves adjustments of timer offset and rate values. The adjustments of the timer's offset is called offset synchronization (see 7.1.12); the adjustments of the timer's rate is called rate synchronization (see 7.1.13). Both involve calibration of local clock-master/clock-slave differences and the propagation of cumulative differences in the clock-slave direction, as described by the C code of Annex J.

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

7.1.8 Grand-master precedence

Grand-master precedence is based on the concatenation of multiple fields, as illustrated in Figure 7.4. The *portTag* value is used within bridges, but is not transmitted between stations.

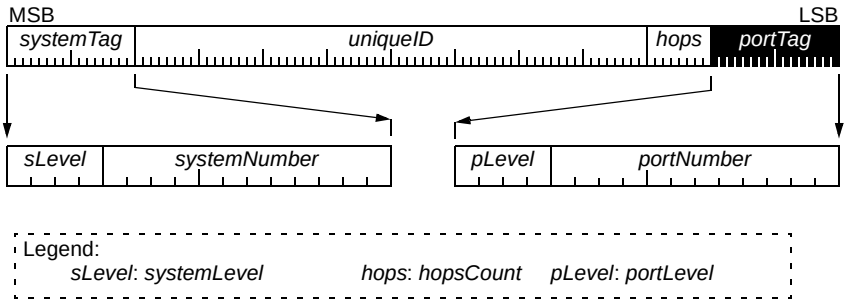


Figure 7.4—Grand-master precedence

This format is similar to the format of the spanning-tree precedence value, except that a larger *uniqueID* is provided for compatibility with interconnects based on 64-bit station identifiers.

7.1.9 Synchronization principles

Timer synchronization is based on the concept of free-running local times (*localD*, *localE*, and *localF*) with compensating offset values (*offsetD*, *offsetE*, and *offsetF*), as illustrated in Figure 7.5. Updates involve changes to the offset values, not the free-running local timer values. In this example, we assume that: StationE is synchronized to its adjacent StationD; StationF is synchronized to its adjacent StationE. As a result, StationF is indirectly synchronized to StationD (through StationE).

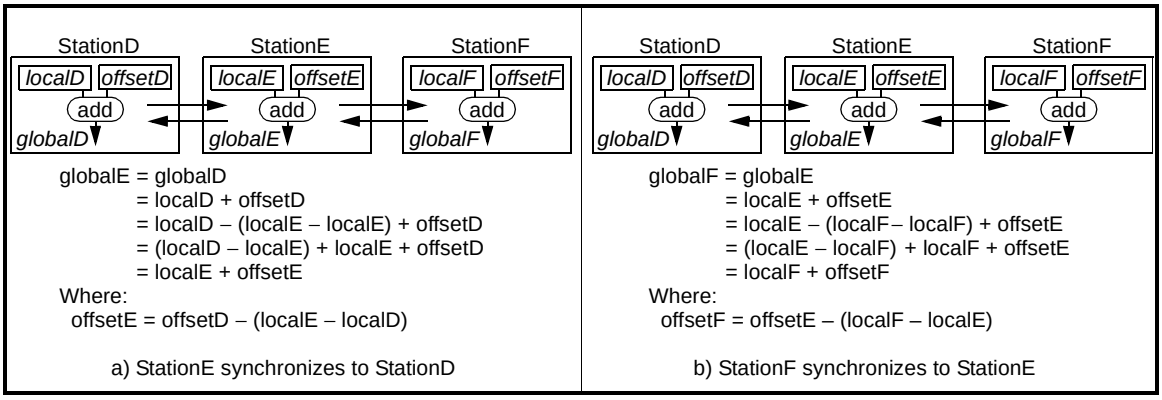


Figure 7.5—Time synchronization principles

The formulation of the *offsetE* value begins the assumption that the *globalE* and *globalD* times are identical. Addition of (*localE–localE*) and regrouping of terms leads to the formulation of the desired *offsetE* value, based on *offsetD* and (*localE–localD*) time difference values, as illustrated in Figure 7.5-a. Synchronization is thus possible using periodic transfers of *offsetD* values and computations of the (*localE–localD*) difference.

The formulation of the *offsetF* value begins the assumption that the *globalF* and *globalE* times are the identical. Addition of (*localF–localF*) and regrouping of terms leads to the formulation of the desired *offsetF* value, based on *offsetE* and (*localF–localE*) time difference values, as illustrated in Figure 7.5-b. Synchronization is thus possible using periodic transfers of *offsetE* values and computations of the (*localF–localE*) difference.

In concept, the *offsetE* value is adjusted first; its adjusted value is then used to compute the desired *offsetF* value. In actuality, the periodic computations of *offsetE* and *offsetF* values are performed concurrently.

7.1.10 Timer snapshot locations

Mandatory jitter-error accuracies are sufficiently loose to allow transmit/receive snapshot circuits to be located with the MAC, as illustrated in Figure 7.6a. Vendors may elect to further reduce timing jitter by latching the receive/transmit times within the PHY, where the uncertain FIFO latencies can be best avoided, as illustrated in Figure 7.6b.

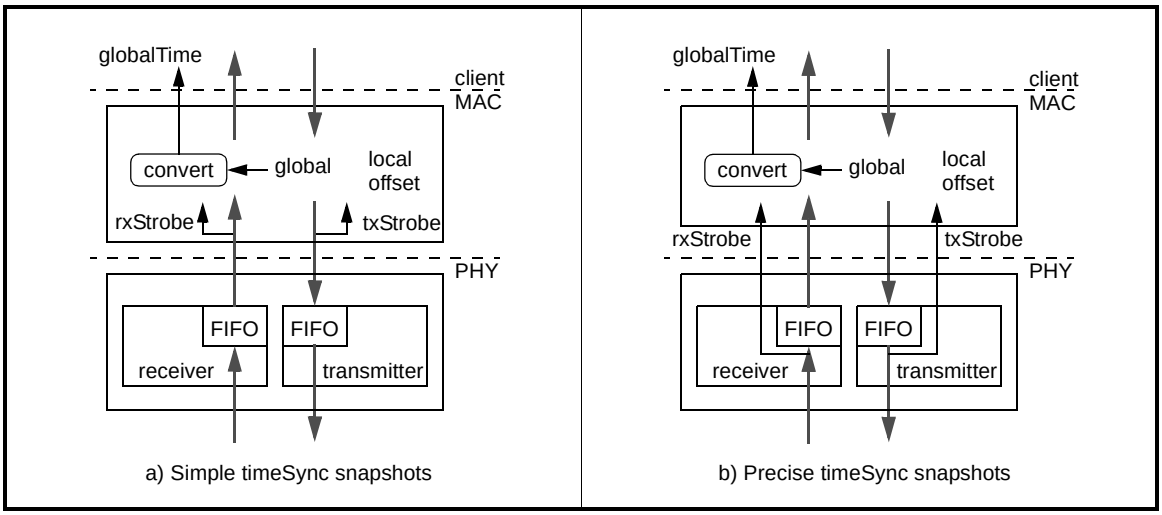


Figure 7.6—Timer snapshot locations

7.1.11 Clock-synchronization updates

7.1.11.1 Clock-synchronization intervals

Clock synchronization involves the processing of periodic events. Three distinct time periods are involved, as listed in Table 7.2. The clock-period events trigger the update of free-running timer values; the period affects the timer-synchronization accuracy and is therefore constrained to be small.

Table 7.2—Clock-synchronization intervals

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

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

The slow-period events trigger the computation of timer-rate differences. The timer-rate differences are computed over two slow-period intervals, but recomputed every slow-period interval. The larger 100 ms (as

opposed to 10 ms) computation interval is intended to reduce errors associated with sampling of clock-period-quantized slow-period-sized time intervals.

7.1.12 Offset synchronization

7.1.12.1 Offset synchronization adjustments

Offset synchronization involves a subset of the time-synchronization components, as illustrated by white-colored boxes in Figure 7.9. Each clock consists of a progressing *timeOfDay* value, whose offset and rate are periodically adjusted. The free-running *flexTimer* timer is never reset; synchronization of stationE (with respect to stationD) is accomplished by adjustments to the *flexOffset* and *flexRate* values within stationE.

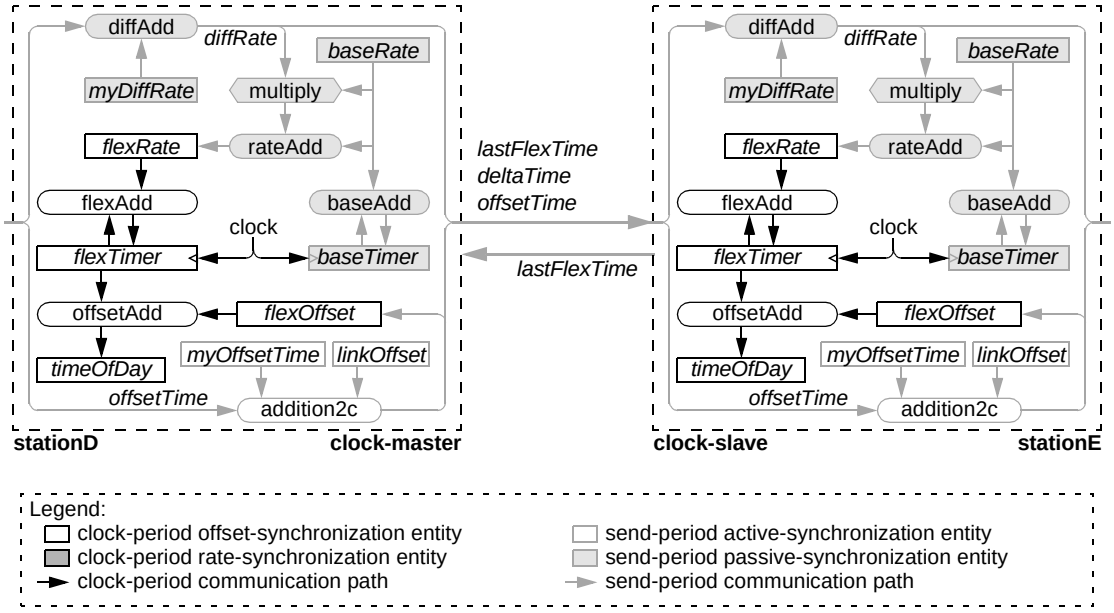


Figure 7.7—Offset synchronization adjustments

The offset-synchronization protocols interchange parameters periodically, possibly every 10 ms. The *lastFlexTime*, *deltaTime*, and *offsetTime* values are sent periodically from the clock-master to the clock-slave. The *lastFlexTime* is sent periodically from the clock-slave to the clock-master, providing information necessary for the clock-master to produce a *deltaTime* value for the clock-slave.

The offset-compensation protocols for stationE adjust its *myOffsetTime* value so that the instantaneous values of *stationE.timeOfDay* and *stationD.timerOfDay* are the same. Computations are performed on clockStrobe reception and clockStrobe transmission.

As an option, an additional *linkOffset* value is available to manually compensate for mismatched transmit-link/receive-link duplex-cable delays on the clock-master side. The *linkOffset* value is expected be manually set when the cable mismatch is known through other mechanisms, such as specialized cable-characterization equipment.

The station's *offsetTime* value is constructed by adding the received *clockStrobe.offsetTime*, local *myOffsetTime*, and local *linkOffset* values. This revised *clockStrobe.offsetTime* value is used within each station and is passed to the downstream neighbor (when such a neighbor is present).

7.1.12.2 Cascaded offsets

The concept of cascaded offset values can be better understood by considering a simple 3-bridge example, as illustrated in Figure 7.8. The slave-agent in bridgeB is synchronized to its neighbor grand-master via timeSync frames sent on the connecting bidirectional span. Within bridgeB, the clock-slave agent passes the time directly to the clock-master agent. The slave-agent in bridgeC is synchronized to its neighbor clock-master via timeSync frames sent on the connecting bidirectional span. Other ports are similarly synchronized, thus synchronizing the right-most clock-slave station to the left-most grand-master station.

Parameter					
name	grand-master	bridgeB	bridgeC	bridgeD	clock-slave
number	1	2	3	4	5
flexTimer	100	500	-300	200	400
myOffsetTime	10	-400	800	-500	-200
flexOffset	10	-390	410	-90	-290
timeOfDay	110				

Representing:
myOffsetTime[k+1] = flexTimer[k]–flexTimer[k+1];
flexOffset[k+1] = flexOffset[k]+myOffsetTime[k+1];
timeOfDay[k] = flexTimer[k] + flexOffset[k];

Figure 7.8—Cascaded offsets (a possible scenario)

To simplify this illustration, consider only the seconds portion of the *flexTimer* value within each station or bridge. These values may differ dramatically, based (perhaps) on the power-cycling or topology formation sequence. Thus, the grand-master could have a *flexTimer* value of 100 while its bridgeB neighbor has a *flexTimer* value of 500.

The *myOffsetTime* value within bridgeB will converges to the value of –400, representing the differences between grand-master and bridgeB *flexTimer* values. The *flexOffset* value received from the grand-master is added to this *myOffsetTime* value, so that bridgeB's *flexOffset* becomes –390. The *flexTimer* and *flexOffset* values are added, to yield a resultant bridgeB *timeOfDay* value of 110, properly synchronized to the identical grand-master's value.

Similarly, bridgeC is synchronized to bridgeB, bridgeD to bridgeC, and the clock-slave to bridgeD.

7.1.13 Rate synchronization

7.1.13.1 Rate synchronization adjustments

Rate synchronization involves a subset of the time-synchronization components, as illustrated by white-colored boxes in Figure 7.9. The free-running *baseTimer* timer facilitate the determination of rate differences between the clock-master and clock-slave stations.

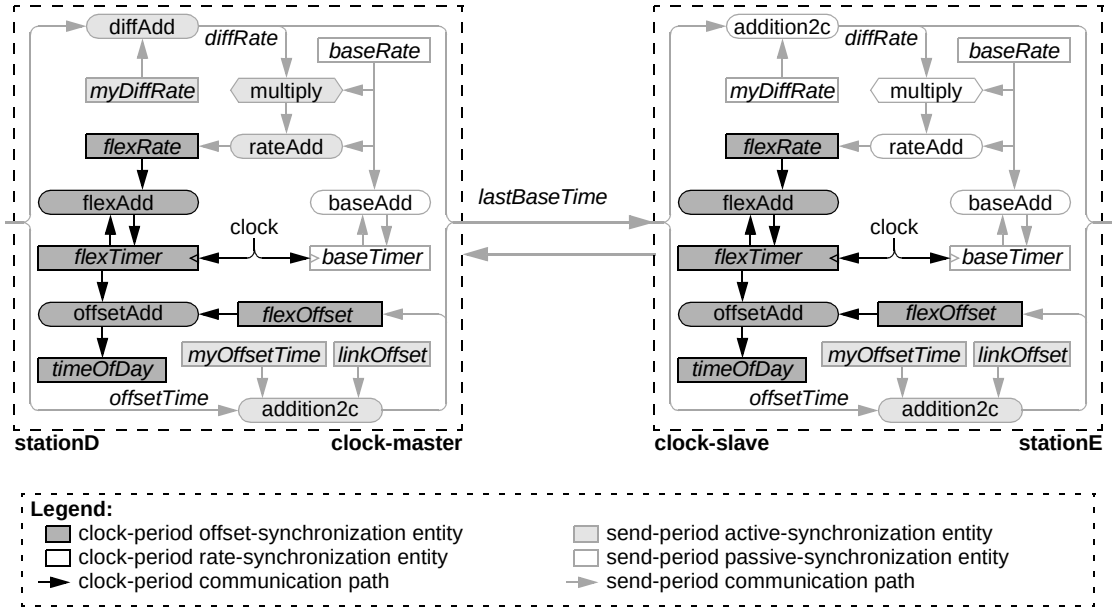


Figure 7.9—Rate synchronization adjustments

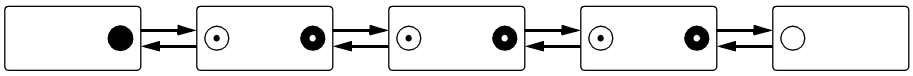
The rate-synchronization protocols interchange parameters periodically, but less frequently than the offset-synchronization protocols, possibly every 100 ms. The *lastBaseTime* value is sent periodically from the clock-master to the clock-slave. Nothing is returned from the clock-slave station.

The rate-compensation protocols for stationE adjust its *myDiffRate* value to accommodate for differences between the *stationD.baseTimer* and *stationE.baseTimer* rates. Computations are performed on clockStrobe reception and clockStrobe transmission.

The station's *diffRate* value is constructed by adding the received *clockStrobe.diffRate* and local *myDiffRate* values. This revised *clockStrobe.diffRate* value is used within each station and is passed to the clock-slave side neighboring station (if present).

7.1.13.2 Cascaded rate differences

The concept of cascaded rate values can be better understood by considering a simple 3-bridge example, as illustrated in Figure 7.10. Within this figure, the *myDiffRateN* and *diffRateN* represent parts-per-million (PPM) normalized values of *myDiffRate* and *diffRate* respectively.

Parameter					
name	grand-master	bridgeB	bridgeC	bridgeD	clock-slave
number	1	2	3	4	5
crystal deviation	+10 PPM	+100 PPM	–100 PPM	–75 PPM	+75 PPM
myDiffRateN	0 PPM	–90 PPM	200 PPM	–25 PPM	–150 PPM
diffRateN	0 PPM	–90 PPM	110 PPM	+85 PPM	–65 PPM
flexTimer deviation	10 PPM				

Representing:
myDiffRateN[k+1] = flexRate[k]–flexRate[k+1];
diffRate[k+1] = diffRate[k]+myDiffRate[k+1];
flexTimerDeviation[k] = crystalDeviation[k] + diffRate[k];

Figure 7.10—Cascaded rate differences (a possible scenario)

The slave-agent in bridgeB is synchronized to its neighbor grand-master via timeSync frames sent on the connecting bidirectional span. Within bridgeB, the clock-slave agent passes the time directly to the clock-master agent. The slave-agent in bridgeC is synchronized to its neighbor clock-master via timeSync frames sent on the connecting bidirectional span. Other ports are similarly synchronized, thus synchronizing the right-most clock-slave station to the left-most grand-master station.

To simplify this illustration, consider only the parts-per-million (PPM) normalized rate values within each station or bridge. These values may differ significant, based (perhaps) on the nominal value or ambient temperature. Thus, the grand-master could have a crystal deviation of +10 while its bridgeB neighbor has a crystal deviation of +100.

The *myDiffRate* value within bridgeB will converges to the value of –90 PPM, representing the differences between grand-master and bridgeB crystal accuracies. The *diffRate* value received from the grand-master is added to the *myDiffRate* value, so that bridgeB's *diffRate* becomes –90 PPM. The *diffRate* and crystal deviation values are additive, yielding a resultant bridgeB *flexTimer* deviation of 10 PPM, properly synchronized to the identical grand-master's value.

Similarly, the rate of bridgeC is synchronized to bridgeB, bridgeD to bridgeC, and the clock-slave to bridgeD.

7.1.14 Rate-difference effects

If the absence of rate adjustments, significant *timeOfDay* errors can accumulate between send-period updates, as illustrated on the left side of Figure 7.11. The 2 ms 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.

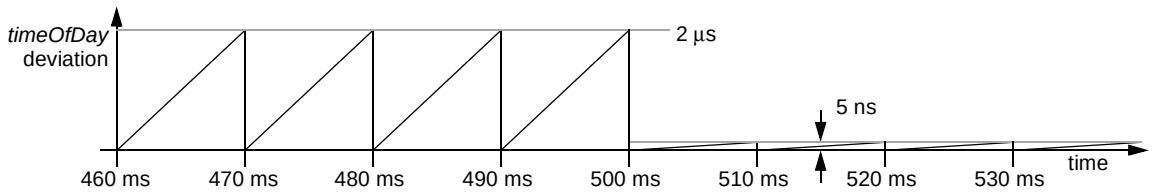


Figure 7.11—Rate-adjustment effects

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 *timeOfDay* 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.

7.1.15 *baseTimer* functionality

The external formats within timeSync frames assumes the presence of *baseTimer*-related values. A direct-mapped hardware implementation involves a clocked *baseTimer* register and a precision adder, as illustrated in Figure 7.12a. Within this figure, the shaded bytes represent values that can safely be hardwired to zero with insignificant loss of accuracy.

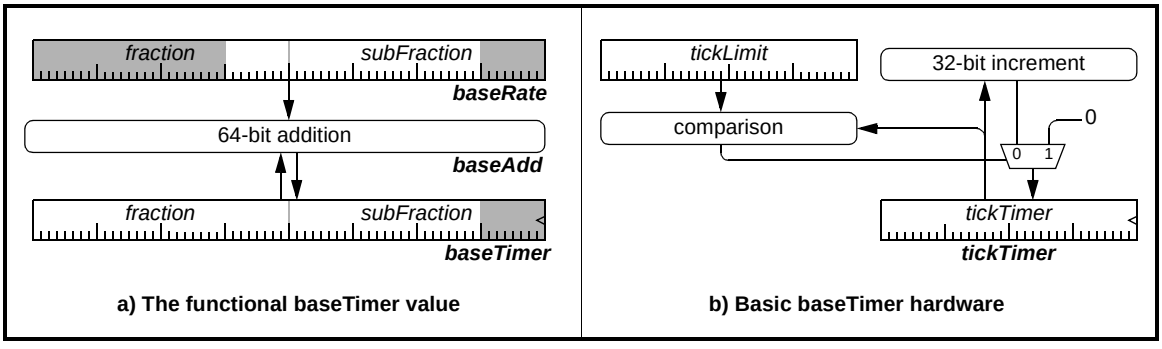


Figure 7.12—*baseTimer* implementation examples

A simpler hardware implementations is to periodically increment a tick timer at an implementation-dependent rate, as illustrated in Figure 7.12a. Although this timer is improperly scaled and insufficiently large, firmware can perform the multiplications and additions necessary to provide the normalized external values.

7.1.16 *flexTimer* functionality

The selection of the best time-of-day format is oftentimes complicated by the desire to equate the clock format granularity with the granularity of the implementation's 'natural' clock frequency. Unfortunately, the 'natural' frequency within a multimodal {1394, 802-100Mb/s, 802.3 1Gb/s} implementation is uncertain, and may vary based on vendors and/or implementation technologies.

The difficulties of selecting a 'natural' clock-frequency can be avoided by realizing that any clock with sufficiently fine resolution is acceptable. Flexibility involves using the most-convenient clock-tick value, but adjusting the timer advance rate associated with each clock-tick occurrence.

The same mechanism easily supports both near-arbitrary clocking rates and fine-grained rate-adjustments, needed to support timer-synchronization protocols, as illustrated in Figure 7.13. Within this figure, the shaded bytes represent values that can safely be hardwired to zero with insignificant loss of accuracy.

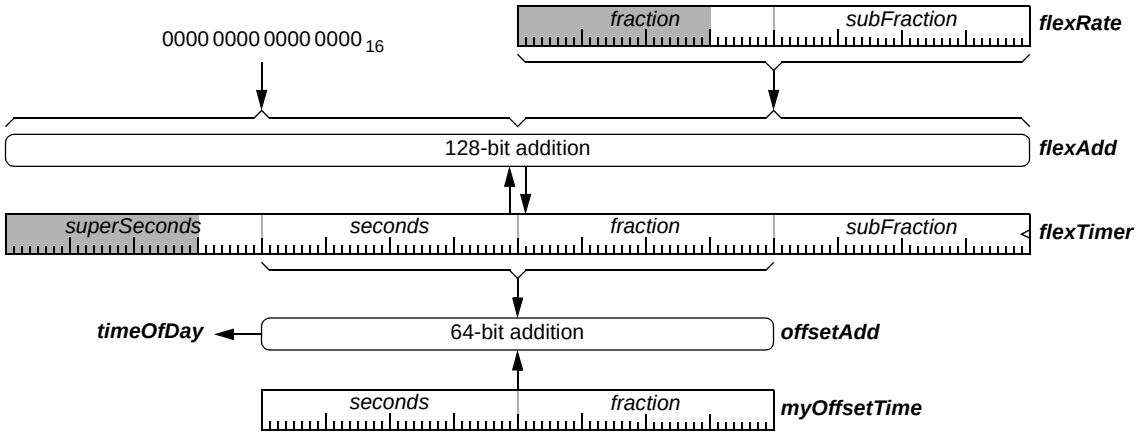


Figure 7.13—*flexTimer* implementation example

This illustration is not intended to constrain implementations, but to illustrate how the system's clock and timer formats can be optimized independently. This allows the *timeOfDay* timer format to be based on arithmetic convenience, timing precision, and years-before-overflow characteristics (see Annex E).

Although a direct hardware implementation is possible, a simpler solution is to utilize a single *tickTimer* register (see 7.1.15). Since the external communication rates are low, firmware conversions between *tickTimer* and *flexTimer* values are expected to be feasible.

7.2 Terminology and variables

7.2.1 Common state machine definitions

The following state machine inputs are used multiple times within this clause.

NULL
: Indicates the absence of a value and (by design) cannot be confused with a valid value.

queue values
: Enumerated values used to specify shared queue structures.
  - Q_RX_FRAME—Identifies the queue that supplies received frames.
  - Q_RX_SYNC—Identifies the queue where received timeSync frames are saved.
  - Q_RX_ELSE—Identifies the queue where received non-timeSync frames are saved.
  - Q_TX_FRAME—The identifier associated with the transmitted timeSync frames.

SCALE
: A multiplicative scalar for converting between the internal *core.diffRate* fraction and the external *frame.diffRate* integer values.
  Value: 2^{44}

7.2.2 Common state machine variables

One instance of each variable specified in this clause exists in each port, unless otherwise noted.

currentTime
: A value representing the current time.

core
: The state associated with the common portion of a bridge:
  - *clockDeviation*—Nominal frequency deviation of crystal-stabilized clock, in PPM.
    Maximim: +100
    Minimum: –100
  - *clockFrequency*—The nominal frequency of the synchronized timer.
    clockFrequency > 50 MHz
  - *diffRate*—The cumulative rate difference from the grand-master station.
  - *flexOffset*—The cumulative value difference from the grand-master station.
  - *linkOffset*—The specified delay differences between transmit and receive links.
  - *myDiffRate*—The measured rate difference between the local and remote ports.
  - *myOffsetTime*—The measured time-offset difference between the local and remote ports.
  - *counter*—A running count that is incremented approximately once every 10 ms.
  - *offsetTicks*—A *tickTimer* snapshot taken at the start of each 10 ms interval.
  - *precedence*—The currently observed grand-master precedence value.
  - *rateCount*—A running count that is incremented approximately once every 100 ms.
  - *slaveCount*—A snapshot of *offsetCount*, received from a grand-master slave port.
  - *slavePort*—The slave port identifier associated with an upstream grand-master station.
  - *systemTag*—The most-significant portion of this station’s grand-master precedence.
  - *tickExtra*—The software-maintained seconds-overflow value associated with *core.tickTimer.*
  - *tickLimit*—The maximum value of *core.tickTimer* before a seconds overflow occurs.
  - *tickOffset*—A scalar parameter for the conversion between *tickTimer* and *flexTimer* values.
  - *tickRate*—A scalar parameter for the conversion between *tickTimer* and *flexTimer* values.
  - *tickTime*—A snapshot of the *currentTime* value at the beginning of each *clockPeriod* interval.
  - *tickTimer*—A running count that is incremented at the end of each *clockPeriod* interval.
  - *uniqueID*—The more-significant portion of this station’s grand-master precedence

frame
The contents of a timeSync frame, including the following components (see 6.2.1):
da, *sa*, *protocolType*, *subType*, *hopsCount*, *syncCount*, *cycleCount*, *systemTag*, *uniqueID*, *lastFlexTime*, *deltaTime*, *offsetTime*, *diffRate*, *lastBaseTime*, *fcs*.
See 6.2.1.
rxtickTimer—The value of *core.tickTimer*, sampled when this frame was received.
port
The state associated with each of the ports on a bridge:
counter—An accounting value whose comparison to *core.counter* triggers processing.
portPri—A priority differentiator for the port.
portID—A unique identifier for the port.
propTime—An average cable propagation delay measurement for the attached link.
rateCount—An accounting value whose comparison to *core.rateCount* triggers processing.
rxFlexTime0—The current value of *flexTime* when the frame is received.
rxFlexTime—The current of *rxFlexTime0* when the frame is received.
rxTickTime0—The current value of *tickTime* when the frame is received.
rxTickTime —The current value of *rxTickTime0* when the frame is received.
syncCount—The last observed *frame.syncCount* value, saved for consistency checks.
timeSyncAllowed—Indicates when a timeSync frame is enabled for transmission.
txtickTimer—The saved *core.tickTimer* value, when timeSync was last transmitted.
tickTimer
See 7.2.2.

7.2.3 Common state machine routines

Best(*test*, *base*)
Indicates whether the *test* is smaller than the *base* precedence, as defined by Equation 7.1.

```
(test.hi < base.hi || (test.hi == base.hi && test.lo <= base.lo))
```
(7.1)

Dequeue(*queue*)
Returns the next available frame from the specified queue.
frame—The next available frame.
NULL—No frame available.
Enqueue(*queue*, *frame*)
Places the frame at the tail of the specified queue.
FlexTimer(*tickTime*, *tickRate*, *tickOffset*)
Computes the effective *flexTimer* value based on the provided inputs, as defined by Equation xx.

```
(tickTime * tickRate * (1.0 + tickDiff) + tickOffset)
```
(7.2)

Min(*value1*, *value2*)
Returns the numerically smaller of two values.
Precedence(*sys*, *uid*, *hops*, *tag*)
Forms a 128-bit precedence value from its component fields, as defined by Equation 7.1.

```
Precedence(sys, uid, hops, pri, pid)
{
    doubleInt value;

    value.hi = (sys << 24) | (uid >> 40);
    value.lo = (uid << 24) | (hops << 16) | (pri << 12) | pid;
    return(value);
}
```
(7.3)

QueueEmpty(*queue*)
Indicates when the queue has emptied.
TRUE—The queue has emptied.
FALSE—(Otherwise.)

7.3 Clock synchronization state machines

7.3.1 TickTimer state machine

7.3.1.1 TickTimer state machine definitions

The following definitions are used within this subclause:

NULL
A constant that indicates the absence of a slave-port identifier.
TICKS_PER_SEC
The nominal number of tick periods in every cycle.

7.3.1.2 TickTimer state machine variables

Only one instance of each state-machine variable exists in each bridge.

core
currentTime
See 7.2.2.
precedence
A computed grand-master precedence value, based on this station's parameters.

7.3.1.3 TickTimer state machine routines

The following routines are used within this subclause:

Best(*test, base*)
Precedence(*sys, uid, hops, tag*)
See 7.2.3.

7.3.1.4 TickTimer state table

Each bridge supplies only one TickTimer state machine.

The TickTimer state machine provides the timer values for other state machines, as specified in Table 7.3. Only the START-1, BUMP-1, and BUMP-2 rows (shaded gray) are expected to require hardware support; the remaining rows are expected to be easily mapped to firmware. The notation used in the state table is described in 3.4.

Table 7.3—TickTimer state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	delta = core.clockDeviation / 1000000, (currentTime – core.tickTime) >= 1.0 / (core.clockFrequency * (1.0 + delta))	1	core.tickTime = currentTime;	BUMP

	(core.tickTimer – core.offsetTicks) >=
	0.010 * TICKS_PER_SEC
		2	core.offsetTicks = core.tickTimer;
			core.counter += 1;
			preference =
			Precedence(core.systemTag,
			core.uniqueID, 0, 0, 0);
				PLUS

	—	3	—	START
BUMP	core.tickTimer >= core.tickLimit	1	core.tickTimer = 0;	START
	—	2	core.tickTimer += 1;	
PLUS	core.tickTimer < core.lastTimer	1	core.tickExtra += core.tickLimit + 1; core.lastTimer = core.tickTimer;	NEXT
	—	2	—	
NEXT	(core.counter – core.slaveCount) >= 5	1	core.slavePort = NULL;	NEAR
	—	2	—	

NEAR	Best(preference, core.precedence)
	|| core.slavePort == NULL
		1	core.slavePort = NULL;
			core.precedence = precedence;
			core.slaveCount = core.counter;
				FINAL

	—	2	—	

Row START-1: The *tickTimer* register is incremented once every *clockPeriod* interval.
(The *tickTimer* register is the timer used to snapshot all arrival and departure times.)
Row START-2: The *counter* register is incremented approximately once every 10 ms interval.
(Changes in the *counter* register triggers transmissions of periodic *timerSync* frames.)
Row START-3: Otherwise, no updates are performed.

Row BUMP-1: The *tickTimer* register overflows after the core.tickLimit value is reached.
Row BUMP-2: Otherwise, the *tickTimer* register is incremented.

Row PLUS-1: The *tickTimer* register seconds-carry eventually propagates into the *core.tickExtra* field.
Row PLUS-2: Otherwise, no seconds-carry is required.

Row NEXT-1: If the slave port receives no heartbeats, the slave-port identifier is released.
Row NEXT-2: Otherwise, no updates are performed.

Row NEAR-1: If this station has the best preference, or no slave port exists, this station has precedence.
Row NEAR-2: Otherwise, the cumulative preference value remains unchanged.

7.3.2 TimerRxLatch state machine

7.3.2.1 TimerRxLatch state machine definitions

The following definitions are used within this subclause:

Q_RX_ELSE
Q_RX_FRAME
Q_RX_SYNC
See 7.2.1.

7.3.2.2 TimerRxLatch state machine variables

The following variables are used within this subclause:

core
frame
See 7.2.2.

7.3.2.3 TimerRxLatch state machine routines

The following routines are used within this subclause:

Dequeue(*queue*)
Enqueue(*queue*, *frame*)
See 7.2.3.

7.3.2.4 TimerRxLatch state table

The TimerRxLatch state machine associates a time stamp within incoming frames, as specified in Table 7.4. The notation used in the state table is described in 3.4.

Table 7.4—TimerRxLatch state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(frame = Dequeue(Q_RX_FRAME)) != NULL	1	—	FINAL
	—	2	—	START
FINAL	frame.protocolType == TIME_SYNC	1	frame.rxTickTimer = core.tickTimer; Enqueue(Q_RX_SYNC, frame);	START
	—	2	Enqueue(Q_RX_ELSE, frame);	

Row START-1: A new frame has arrived and is available for processing.
Row START-2: Wait for a new frame to arrive.

Row FINAL-1: The timeSync frames are immediately time-stamped and enqueued for special processing.
Row FINAL-2: Other frames are processed in their normal fashion.

7.3.3 TimerTxLatch state machine

7.3.4 TimerRxLatch state machine

7.3.4.1 TimerRxLatch state machine definitions

The following definitions are used within this subclause:

Q_TX_FRAME
Q_TX_SYNC
See 7.2.1.

7.3.4.2 TimerRxLatch state machine variables

The following variables are used within this subclause:

core
frame
port
See 7.2.2.

7.3.4.3 TimerTxLatch state machine routines

The following routines are used within this subclause:

Dequeue(*queue*)
Enqueue(*queue, frame*)
QueueEmpty(*queue*)
See 7.2.3.

7.3.4.4 TimerTxLatch state table

The TimerTxLatch state machine associates a time stamp within incoming frames, as specified in Table 7.5. The notation used in the state table is described in 3.4.

Table 7.5—TimerTxLatch state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(QueueEmpty(Q_TX_FRAME)) && port.timeSyncAllowed	1	—	FINAL
	—	2	—	START
FINAL	(frame = Dequeue(Q_TX_SYNC)) != NULL	1	port.txTickTimer = core.tickTimer; Enqueue(Q_TX_FRAME, frame);	START
	—	2	—	

Row START-1: A new frame has arrived and is available for processing.
Row START-2: Wait for a new frame to arrive.

Row FINAL-1: The timeSync frames are immediately time-stamped and saved for special processing.
Row FINAL-2: Other frames are processed in their normal fashion.

7.3.5 TimerRxCompute state machine

7.3.5.1 TimerRxCompute state machine definitions

The following definitions are used within this subclause:

Q_TX_SYNC
See 7.2.1.

7.3.5.2 TimerRxCompute state machine variables

The following variables are used within this subclause:

core
: See 7.2.2.

count
: A saved copy of the last received frame's *syncCount* field.

diffRate
: The copy of the *core.diffRate* value.

diffRate0
: The value of *core.diffRate*, before the decay computation is performed.

diffRate1
: The value of *core.diffRate*, after the decay computation is performed.

formed
: A computed grand-master precedence value, based on the frame-supplied parameters.

frame
: See 7.2.2.

myTickDelta
: The difference between *tickTime* arrival times within sampled frames.

pastCount
: A saved copy of the previous *count* value.

port
: See 7.2.2.

rxBaseDelta
: The difference between *lastBaseTime* values within sampled frames.

rxBaseTime1
: The *lastBaseTime* value observed in a previously sampled frame.

rxPrecedence
: A copy of the grand-master precedence from the last received frame.

rxTickDelta
: A temporary value representing time-snapshot differences on the receiving link.

rxTickTime1
: A *tickTime* value observed in a previously sampled frame.

tickTimer
: A copy of the frame-supplied *rxtickTimer* field.

txDeltaTime
: A temporary value representing time-snapshot differences on the transmitting link.

7.3.5.3 TimerRxCompute state machine routines

The following routines are used within this subclause:

Dequeue(*queue*)

FlexTime(*tickTime, tickRate, tickDiff, tickOffset*)
: See 7.2.3.

7.3.5.4 TimerRxCompute state table

The TimerRxCompute state machine processes received timeSync frames while participating in the grand-master selection protocol, as specified in Table 7.6. The notation used in the state table is described in 3.4.

Table 7.6—TimerRxCompute state table

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

START	(frame = Dequeue(Q_RX_SYNC)) != NULL && port.timeSyncAllowed	1	pastCount = port.syncCount;
			port.syncCount = count = frame.syncCount;
			tickTime = frame.rxTickTimer + core.tickExtra;
				TIME

	—	2	—	START
TIME	frame.rxTickTimer < core.lastTimer	1	tickTime += core.tickLimit;	PREP
	—	2	—	

PREP	—	1	port.rxFlexTime = port.rxFlexTime0;
			port.rxFlexTime0 = FlexTime(tickTimer, core.tickRate, core.diffRate, core.tickOffset);
			
			rxTickTime = port.rxTickTime0;
			port.rxTickTime0 = tickTimer;
				FIRST

FIRST	count != (pastCount + 1) % 256;	1	—	FINAL

	—	2	port.rxDeltaTime = port.rxFlexTime1 – frame.lastFlexTime;
			txDeltaTime = frame.deltaTime;
			
			port.propTime = (txDeltaTime + port.rxDeltaTime)/2;
			
			rxPrecedence = Precedence(frame.systemTag, frame.uniqueID, frame.hopsCount, port.portPri, port.portID);
				FAST

FAST	!Best(rxPrecedence, core.precedence) && core.slavePort != port.portID	1	—	START

	—	2	core.slavePort = port.portID;
			core.precedence = core.compare;
			
			core.myOffsetTime = (txDeltaTime – rxDeltaTime)/2;
			core.flexOffset = core.linkOffset + frame.offsetTime + core.myOffsetTime;
				NEAR

Table 7.6—TimerRxCompute state table

Current		Row	Next	
state	**condition**		**action**	**state**
SLOW	port.counter >= core.counter + 50	1	—	FINAL
	port.counter >= core.counter + 10	2	rxBaseDelta = (frame.lastBaseTime – port.rxBaseTime1); rxTickDelta = (rxTickTime = port.rxTickTime1); myTickDelta = FlexRate(rxTickTime, core.baseRate, 0); diffRate0 = core.myDiffRate; diffRate = (rxTickDelta – myTickDelta) / myTickDelta; core.myDiffRate = diffRate = Clip(diffRate, 1.0 / 4096); diffRate1 = diffRate + (frame.diffRate / SCALE); core.diffRate = diffRate1 = Clip(diffRate1, 1.0 / 4096); core.tickOffset –= FlexTime(core.tickTimer, core.tickRate, diffRate1 – diffRate0, 0);	
	—	3	—	START

FINAL	—	1	port.counter = core.counter;
			
			port.rxBaseTime1 = frame.lastBaseTime;
			port.rxTickTime1 = rxTickTime;
				START

Row START-1: A new frame has arrived for processing.
Save the frame's *syncCount* field, to facilitate detection of missing timeSync frames.
Row START-2: Wait for a new timeSync frame to arrive.

Row TIME-1: If the sampled time has overflowed, an additional overflow amount is added.
Row TIME-2: Otherwise, only the cumulative overflow amount is added.

Row PREP-1: If the sampled time has overflowed, an additional overflow amount is added.
Save the current *flexTime* value, to facilitate computation of:
myOffsetTime—The time difference between neighbor and local *flexTimer* values.
linkOffset—The time difference attributed to link propagation delay.
Save the current *tickTime* value, to facilitate computation of *myDiffRate* values

Row FIRST-1: Discard non-consecutive timeSync frames, since previously saved values are incorrect.
Row FIRST-2: Process consecutive timeSync frames to maintain time synchronization:
Compute *rxDeltaTime* and *txDeltaTime* values.
Compute the cable propagation delays, based on *rxDeltaTime* and *txDeltaTime* values.
Compute the grand-master precedence of the frame, based on frame-supplied values.

Row FAST-1: Skip further processing if this is the primary clock-slave receive port, based on:
The frame-supplied precedence is less than this station's observed precedence.

This port has not been identified as the clock-slave port.
Row FAST-2: The clock-slave port updates appropriate clock-synchronization values.
This port is identified as the clock-slave port and the frame's precedence is saved.
The residual rate differential (if any) decays towards zero with an approximate 1-second time constant.
The core's *flexTimer* offset values are computed and updated.

Row SLOW-1: An excessive lower-rate interval is processed as an error.
Row SLOW-2: Further computations are performed at the lower-rate interval.
The current lower-rate interval index is saved, so that processing only occurs once per interval.
The neighbor's *baseTime* difference is computed.
The station's *baseTime* difference is computed and normalized to the interchange format.
Compute the local (from the neighbor) and cumulative (from the grand-master) rate differences.
This station's timer is adjusted to operate at the observed rate of the grand master.
Row SLOW-3: Otherwise, no rate differences are calculated.

Row FINAL-1: The lower-rate interval is concluded.
Update the counter to facilitate measurement of the next lower-rate interval duration.
Save the observed time snapshot values.

7.3.6 TimerTxCompute state machine

7.3.6.1 TimerTxCompute state machine definitions

The following definitions are used within this subclause:

NULL
: A constant that indicates the absence of a slave-port identifier.

Q_TX_FRAME
: See 7.2.1.

7.3.6.2 TimerTxCompute state machine variables

The following variables are used within this subclause:

core

diffRate0
: The value of core.diffRate, before the decay computation is performed.

diffRate1
: The value of core.diffRate, after the decay computation is performed.

frame

port
: See 7.2.2.

tickTime
: A extended version of core.timerTicks that accounts for per-second overflows.

7.3.6.3 TimerTxCompute state machine routines

The following routines are used within this subclause:

Enqueue(*queue, frame*)

FlexTime(*tickTime, tickRate, tickDiff, tickOffset*)

QueueEmpty(*queue*)
: See 7.2.3.

7.3.6.4 TimerTxCompute state table

The TimerTxCompute state machine provides the arguments for the timeSync frame, as specified in Table 7.7. The notation used in the state table is described in 3.4.

Table 7.7—TimerTxCompute state table

Current		Row	Next	
state	**condition**		**action**	**state**
START	(QueueEmpty(Q_TX_FRAME)) && port.counter != core.counter	1	port.counter = core.counter; tickTime = port.txTickTime +core.tickExtra;	TIME
	—	2	—	START
TIME	port.tickTimer < core.lastTimer	1	tickTime += core.tickLimit;	PREP
	—	2	—	

PREP	core.slavePort == NULL	1	diffRate0 = core.diffRate;
			diffRate1 = diffRate0 – (diffRate0 / 256);
			core.diffRate = diffRate1;
			
			core.tickOffset –=
			FlexTime(tickTime, core.tickRate,
			newDiffRate – oldDiffRate, 0);
			
			frame.offsetTime = core.myOffsetTime;
				FINAL

	—	2	frame.offsetTime = core.flexOffset;	

FINAL	—	1	frame.lastFlexTime =
			FlexTime(tickTime, core.tickRate,
			core.diffRate, core.tickOffset);
			
			frame.deltaTime = port.rxDeltaTime;
			
			frame.diffRate = core.diffRate * SCALE;
			
			frame.lastBaseTime =
			FlexTime(tickTime, core.tickRate, 0, 0);
			Enqueue(Q_TX_FRAME, frame)
				SEND

Row START-1: When space is available and the time has arrived, transmit the next timeSync frame.
Row START-2: Wait until the next timeSync transmission time.

Row TIME-1: If the sampled time has overflowed, an additional overflow amount is added.
Row TIME-2: Otherwise, only the cumulative overflow amount is added.

Row PREP-1: The grand-master station processing is distinct.
The inherited rate differences decays towards zero.
Adjust the *tickOffset* value to compensate for rate-change induced offsets.
Transmit an *offsetTime* value that corresponds to the core's present value.
Row PREP-2: Transmit an *offsetTime* value that corresponds to the cumulative value.

Row FINAL-1: When space is available and the time has arrived, transmit the next timeSync frame.
The frame's *lastFlexFrame* field is based on the previously sampled *tickTime* value.
The frame's *deltaTime* field is based on the port's receiver-computed time-difference value.
The frame's *diffRate* field is based on the receiver-computed rate-difference value.
The frame's *lastBaseTime* field is based on the previously sampled *tickTime* value.

7.4 Protocol comparison

A comparison of the Residential Ethernet Study Group (RE-SG) and IEEE 1588 proposals is summarized in Table 7.8.

Table 7.8—Protocol comparison

Properties		Row	Descriptinos	
state			RE-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.
RE-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.
RE-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.
RE-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.
RE-SG: There are no distinct initialization phases.
1588: There are multiple initialization phases (to be provided).

Row 5: Arbitrary interconnect topologies should be supported.
RE-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.
RE-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.
RE-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.
RE-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.
RE-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.
RE-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.
RE-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.
RE-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.
RE-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 faciliate accurate synchronization.
RE-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 suffiently large to avoid overflows within one's lifetime.
RE-SG: The 64-bit time-of-day timer overflows once every 136 years.
1588: (To be provided).