# DVJ Comments for

**Information Technology—**
**Telecommunications and information exchange between systems—**
**Local and metropolitan area networks—**
**Specific requirements—**

# Part 17: Resilient packet ring access method & physical layer specifications

Cross references:

| | |
|---|---|
| Topology constraints: | 1.2.1 |
| Ringlet topologies: | 1.2.2 |
| Spatial reuse: | 1.2.3 |
| Traffic classes: | 1.2.4 |
| Attachments: | 1.3.1 |
| Reverse flow control | 1.4.1 |
| Cut-through&store-forward | 1.6 |
| Conformance | 3.1 |
| Numerical notation | 3.3 |
| Field names | 3.4 |
| Bit numbering | 3.5 |
| C notation | 3.6 |
| *ClassABC | 5.2 |
| MAC interface | 6.1 |
| Framing | 8.1 |
| Formats | 8.2 |
| Fairness | 9 |
| WrapAndSteer | 10.1 |
| PingSequences | 11.1.1 |
| ProvisionedBandwidth | 11.3 |
| BandwidthSurveys | 11.4 |
| CRC stomping | 11.2.1 |
| CRC protected TTL | 11.2.2 |
| CRC calculations | Annex H |
| Time-of-day | Annex I |

# Table of contents

# List of figures

# List of tables

# 1. Overview

## 1.1 Scope and purpose

## 1.2 RPR topologies

### 1.2.1 RPR topology constraints

RPR protocols are highly scalable, in the sense that 1-to-63 stations can be supported on a small or large (up to 2,000 kilometer) duplex ring, as illustrated in Figure 1.1. The RPR protocols are designed to be physical medium independent and speeds beyond 40Gbs are expected. Bandwidth allocation protocols rely on the ringlet-nature of an RPR topology to control transmissions of concurrent "on-the-wire" packets.

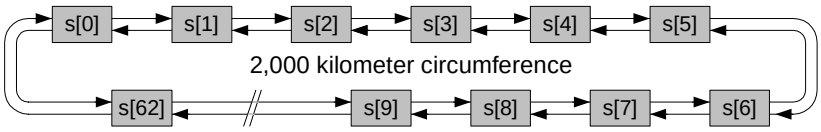



**Figure 1.1—RPR size constraints**

RPR protocols are based on the use of duplex links, so that each ring normally consists of counter-rotating ringlets. Each of these ringlets operates in a largely independent fashion, although flow-control information for one ringlet is placed on the opposing ringlet.

### 1.2.2 Ring topologies

RPR is targeted for cable-ring topologies and maximizes bandwidth capabilities through the use of full-duplex cabling, as illustrated in Figure 1.2. The full-duplex cable infrastructure normally allows concurrent transmissions on the clockwise and counter-clockwise rings, as illustrated in the left of Figure 1.2. The client may choose to send data in either direction, based on shortest-distance, available bandwidth, or higher link capacity criteria.

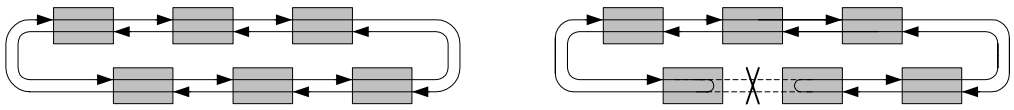



**Figure 1.2—Ring topologies**

After a link failure, communication continues (but at a reduced rate) over the remaining ring, as illustrated in the right of Figure 1.2. The clients are expected to direct data (to the right or left) based solely on the relative physical location of the destination. (This is called steering). Although a full ring is present, steering inhibits utilization of the loop-back paths within the terminal stations.

Similar performance enhancing techniques have also been used on serial-copper SSA, serial-fiber FDDI (Fiber Distributed Data Interface), and parallel-copper SCX interconnects.

### 1.2.3 Spatial reuse

Spatial reuse is a concept used on rings to increase the aggregate bandwidth beyond the capacity of an individual link. Spatial reuse occurs when concurrent data transfers occupy non-overlapping portions of a ringlet, as illustrated in the left side of Figure 1.3. In this example, traffic can be sent between stations *s[0]*-and-*s[2]* without affecting the bandwidth available between stations *s[4]*-and-*s[5]*.

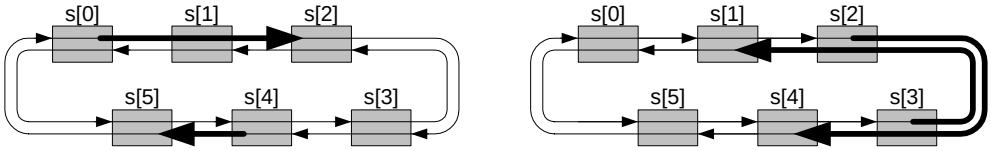



**Figure 1.3—Concurrent data transfers**

The counter-rotating nature of the ringlets also supports concurrent transfers on overlapping segments, as illustrated on the right side of Figure 1.3. In this example, traffic is being sent between stations *s[2]*-and-*s[4]* while opposing traffic is being sent between stations *s[3]*-and-*s[1]*. Concurrent transfer are possible, even though the traffic occupies opposing runs on the bidirectional link between stations *s[2]*-and-*s[3]*.

### 1.2.4 Traffic classes

RPR supports three classes of client traffic, although one class is partitioned into two distinct subclasses on the ring, as illustrated in Figure 1.4 and listed below. Each station is required to police its class-A and class-B traffic to avoid exceeding its provisioned limits.

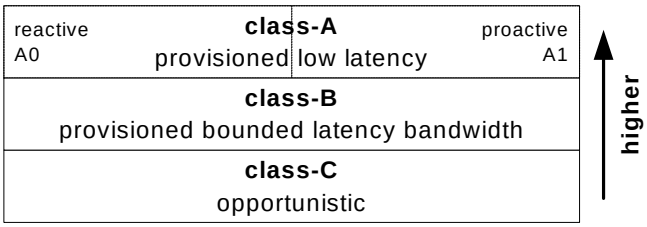



**Figure 1.4—Traffic classes**

1) Class-A: Provisioned low-jitter bandwidth, as needed for interactive audio/video traffic.
   Jitter bounds are based on the number of stations times the maximum transfer unit.
   a) Subclass-A0 (reactive). An efficient dynamic bandwidth allocation, wherein the unused class-A bandwidth is available to class-B and class-C traffic.
   b) Subclass-A1 (proactive). A less efficient static bandwidth allocation, wherein the unused bandwidth is unavailable to class-B and class-C traffic.
2) Class-B: Provisioned bounded-jitter bandwidth, as needed for streaming video.
   Jitter bounds are based on the ringlet-circulation time.
3) Class-C: Weighted assignment of unprovisioned or unused higher-priority bandwidth, as needed to support bursty and/or best-effort traffic.

There is no delivery-jitter distinction between subclass-A0 and subclass-A1 traffic, since the MAC is responsible for partitioning the overall class-A bandwidths. The use of efficient subclass-A0 bandwidth is preferred, but the levels of available class-A0 bandwidth are restricted by the ratio of transit-buffer and station-to-station-link delays.

## 1.3 MAC components

### 1.3.1 Ringlet attachment points

The MAC interface definition provides interfaces to a pair of attachment points, as illustrated in Figure 1.5. The attachment interface receives data and FIFO fill-level indications from the client; each attachment transfers data-frame and flow-control information.

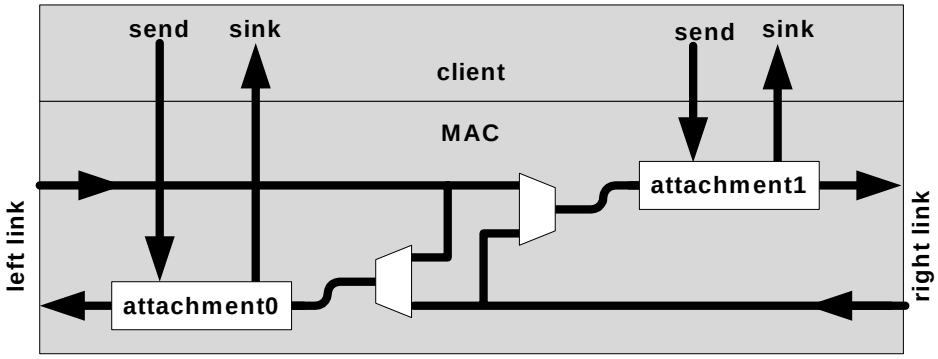



**Figure 1.5—MAC interface signals**

The data paths are expected to flow through two multiplexers, whose controls are semistable in that they are not changed on a packet-by-packet basis, but are available to loop-back frames when failed links are reached.

This concept of an electronically-switched station is not new; a similar capability is provided by stations attached to Serial Bus. Although Serial Bus supports N-port attachments, a 2-port design is sufficient to support the common topologies and simplifies the hardware design.

### 1.3.2 RPR packet buffers

Packet transfers involve transmit queues (where packets are placed for MAC-layer processing), receive queues (when packets are placed for client-layer processing), and transit FIFOs, as illustrated in Figure 1.6. The purpose of the transitA buffer is to hold class-A packets that arrive during this station's transmissions; the purpose of the transitBC buffer is to hold class-B or class-C packets that arrive during class-A transmissions.

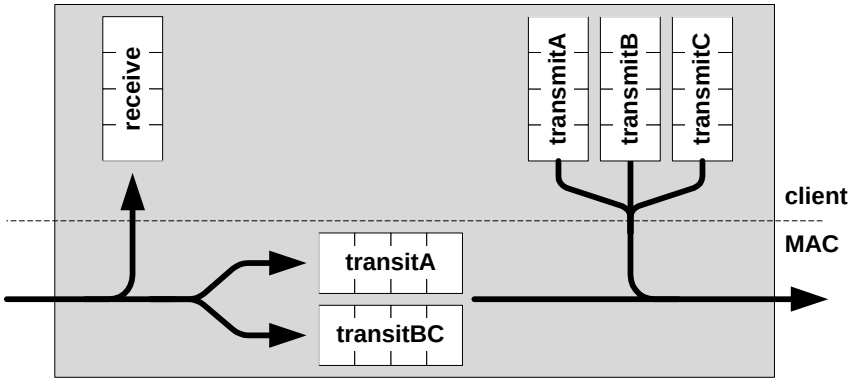



**Figure 1.6—RPR attachment queues**

Multiple transmit queues are provided, for class-A, class-B, and class-C traffic respectively. These transmit buffers are located in the client, which reduce the cost of the MAC while providing flexibility for vendor-dependent just-in-time scheduling protocols.

## 1.4 Flow control

For a shared ring topology in which the RPR MAC is used, each ring segment carries both local client traffic and the traffic from other upstream clients. Unless the upstream clients control their access rates, their traffic can consume unfair portions of ringlet-segment bandwidth, creating congestion and hence blocking the local client from gaining access to the media.

For the higher-priority class-A and class-B traffic, RPR employs a rate-limiting mechanism to prevent long-term congestion conditions. The rate limiting applies to the source traffic, so that sufficient link capacity exists to support the long-term provisioned per-link traffic.

For the lower-priority class-C traffic, a more opportunistic flow-control protocol is used, to maximize bandwidth utilization under dynamically changing conditions.

### 1.4.1 Flow-control signals

For efficiency, arbitration signals and data packets normally flow in opposite directions, as illustrated in Figure 1.7. Clockwise data transmissions (solid lines) are coupled to counterclockwise arbitration signals (dotted lines), as illustrated in the left half of Figure 1.7. Counterclockwise data transmissions (solid lines) are coupled to clockwise arbitration signals (dotted lines), as illustrated in the right half of Figure 1.7.

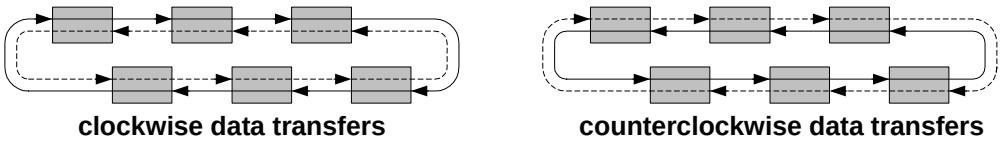



**Figure 1.7—Opposing data and flow-control flows**

The flow-control signals are encapsulated and sent as small class-A data packets. These packets are stripped at their downstream neighbor; their contents are merged with that station's indications and sent further upstream.

### 1.4.2 Proactive class-A scheduling

Proactive scheduling allocates static amounts of provisioned traffic bandwidths, leaving the unprovisioned bandwidth for opportunistic uses. Proactive class-A flow control involves periodic interleaving of idles and class-A0 packets within the data stream, sufficient to sustain the provisioned class-A0 traffic on the most heavily provisioned link.

Since the average rate of class-A0 traffic is prenegotiated and maintained throughout the ringlet, there is no need to dynamically throttle upstream stations based on the depth of the *transitBC* FIFO. However, a modest (several times the maximum frame size) *transitBC* FIFO is sufficient to sustain class-A transmissions during brief pauses in incoming class-A traffic.

The selection between proactive and reactive scheduling is automatic and performed by the MAC, based on the bandwidth requirements, *transitBC* depths, and link lengths. For small bandwidths and/or distances, the more efficient reactive scheduling is used; otherwise, a less efficient proactive scheduling is used. Both mechanisms involve throttling the opportunistic traffic when the progress of provisioned traffic is threatened, due to changes in offered load or load distributions.

### 1.4.3 Virtual output queues

The client of an RPR MAC may send traffic to multiple destinations traversing multiple ring segments. If the MAC does not allow an independent access rate per destination, it is possible that the MAC sets the access rate low to satisfy the bandwidth allocated by one remote congested destination and severely limits the access rates to nearby uncongested destinations.

Thus, destination insensitive congestion management protocols can cause head-of-line blocking: a frame destined to an uncongested destination is forced behind a head-of-line frame whose destination is congested. Until the head-of-line frame is removed from the FIFO, all following frames are blocked.

A well-known solution to this head-of-line blocking problem is a virtual output queue implementation, wherein the client maintains a dedicated transmit-queue for each destination. With a per-destination queue, a frame for one destination is no longer blocked by another frame for a different destination, hence eliminating head-of-line blocking completely.

In order to allow the client to maximize the spatial reuse property of the ring, the RPR MAC implements independent access rate control for each ring segment, allowing the RPR client to provide virtual-output-queue implementations. To support virtual-output-queue implementations, the RPR MAC provides range information (number of allowed hops to the destination) for each of the congestion-level indications sent between the MAC and client.

## 1.5 Station addressing

### 1.5.1 Broadcast addressing

Global broadcast packets are expected to be sent in the shortest *Sa0*-to-*Sa4* and *Sa0*-to-*Sa3* directions, as illustrated in Figure 1.8. The bridge sets the *flood* bit in the RPR header and the multicast bit within the destination MAC address. This facilitates the flooding of broadcasts when the destination RPR station address is unknown.

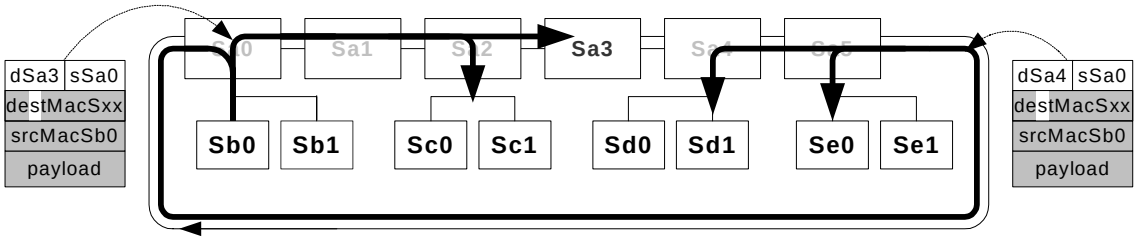



**Figure 1.8—Broadcast Ethernet) frames**

NOTE —The aforementioned broadcast routing technique assumes that station *Sa0* sends unicast transactions over the shortest path. Broadcasts with unknown destinations and unicasts with known directions follow the same paths, so that directed and broadcast packets are not unintentionally reordered.

### 1.5.2 Local unicast transmissions

The RPR addressing protocol is based on the use of unique destination and source station addresses, each of which identify one-and-only-one of the directly attached stations. A local unicast packet is sent directly from station *Sa0* to itself, based on the station addresses within the RPR header, as illustrated in Figure 1.9.

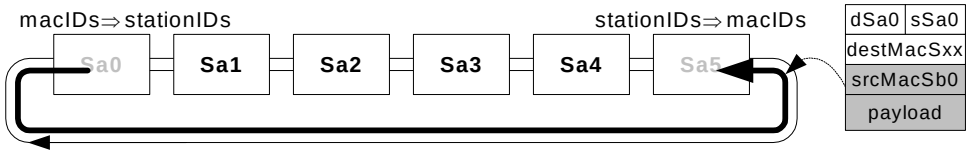



**Figure 1.9—Local unicast transmission**

### 1.5.3 Flood addressing

Remote frames have *destinationMacAddresses* values that differ from the local *macAddresses*. For efficiency and reduced latency, these frames are flooded concurrently in both directions, as illustrated in Figure 1.10. To ensure frame processing within all bridges, the RPR header has a *flood* bit to differentiate between flooded frames and global unicast frames.

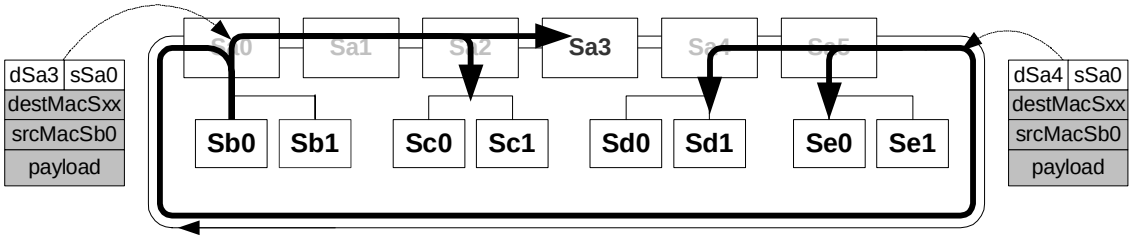



**Figure 1.10—Flooded bridged frames**

### 1.5.4 Remote addressing

A remote unicast frame is sent over the shortest path, from station *Sa0*-to-*Sa5*, based on local station addresses within the RPR header, as illustrated in Figure 1.11. The bridge includes the ring-local destination and source station identifiers, allowing the frame to be stripped at the appropriate bridge.

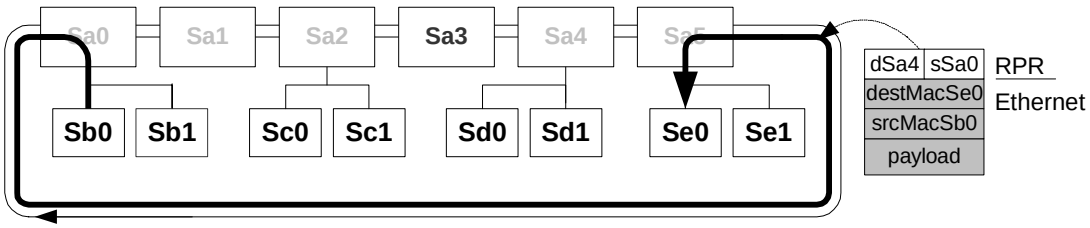



**Figure 1.11—Counterclockwise remote addressing**

Other remote unicast frames are sent in the opposing direction over the shortest path, from station *Sa0*-to-*Sa2*, based on local station addresses within the RPR header, as illustrated in Figure 1.12. Again, the ring-local destination and source station identifiers allow the frame to be stripped at the appropriate bridge.

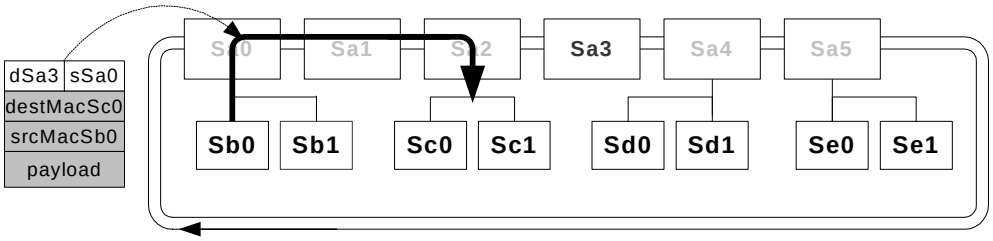


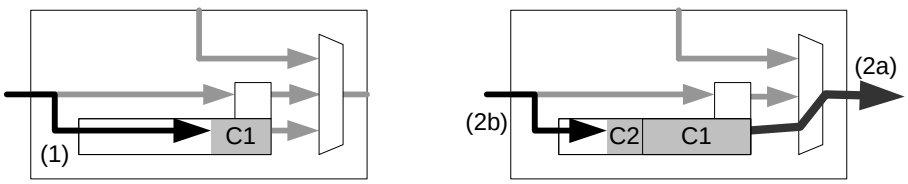

**Figure 1.12—Clockwise remote addressing**

## 1.6 Queuing options

The terms cut-through and store-and-forward describe options for the processing of pass-through traffic. Implementations use either protocol; store-and-forward is simpler but cut-through has the possibility of improved performance.

### 1.6.1 Store-and-forward

Store-and-forward processing delays packet forwarding until after the final portion of the packet has been received, as illustrated by the Figure 1.13 sequence. A packet cannot be retransmitted (1) before the trailing portion of the packet has been received. Once a full packet is available, that packet can be retransmitted (2a) while the following packet (2b) is being received.

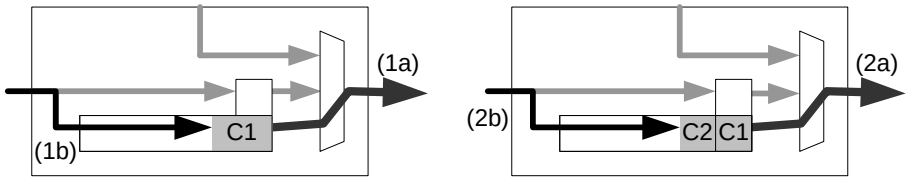



**1) Receiving C1** **2) TransmitC1; receive C2**

**Figure 1.13—Store&forward flows**

### 1.6.2 Cut through

Cut-through processing allows packet forwarding before the final portion of the packet has been received, as illustrated by the Figure 1.14 sequence. The leading portion of a packet is retransmitted (1a) while the trailing portion of the packet (1b) is being received. The retransmission (2a) of the cut-through packet continues while new packets (2b) are received.




**1) Transmit C1; receive C1** **2) Transmit C1; receive C2**

**Figure 1.14—Cut-through flows**

## 2. References

The following standards contain provisions, which through reference in this document, constitute provisions of this standard. 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 standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

[R1] ANSI/ISO 9899-1990, Programming Language—C.[1,2]

1 Replaces ANSI X3.159-1989.

2 ISO documents are available from ISO Central Secretariat, 1 rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzer-land/Suisse; and from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036-8002, USA

# 3. Terms and definitions

## 3.1 Conformance levels

**3.1.1 expected:** A key word used to describe the behavior of the hardware or software in the design models *assumed* by this Specification. Other hardware and software design models may also be implemented.

**3.1.2 ignored, ign:** A term used to describe the fields within registers or frames, whose zero or last-written values shall be ignored.

**3.1.3 may:** A key word that indicates flexibility of choice with *no implied preference*.

**3.1.4 shall:** A key word indicating a mandatory requirement. Designers are *required* to implement all such mandatory requirements.

**3.1.5 should:** A key word indicating flexibility of choice with a strongly preferred alternative. Equivalent to the phrase *is recommended*.

**3.1.6 reserved fields:** A set of bits within a data structure that are defined in this specification as reserved, and are not otherwise used. Implementations of this specification shall zero these fields. Future revisions of this specification, however, may define their usage.

**3.1.7 reserved values:** A set of values for a field that are defined in this specification as reserved, and are not otherwise used. Implementations of this specification shall not generate these values for the field. Future revisions of this specification, however, may define their usage.

## 3.2 Glossary of terms

A large number of network and interconnect-related technical terms are used in this document. These terms are defined below:

NOTE—The following terms are proposed by the author of this draft:

**3.2.1 aligned:** A term which refers to the constraints placed on the address of the data; the address is constrained to be a multiple of the data format size.

**3.2.2 big endian:** A term used to describe the arithmetic significance of addressed data-bytes within a multibyte register. Within a big-endian register or register set, the data byte with the largest address is the least significant.

**3.2.3 byte:** An 8-bit entity. In other standards, this is also called an octet.

**3.2.4 class-A:** Data traffic for which the transmission bandwidth is provisioned and low latency is ensured by assignment of the maximum effective transmission priority.

**3.2.5 class-B:** Data traffic for which the transmission bandwidth is provisioned and latency is bounded by assignment of the high effective transmission priority.

**3.2.6 class-C:** Data traffic for which the transmission bandwidth is unprovisioned; this traffic class has no minimum bandwidth or maximum latency guarantees.

**3.2.7 doublet:** A data format or data type that is 2 bytes in size.

**3.2.8 hexlet:** A data format or data type that is 16 bytes in size.

**3.2.9 octlet:** A data format or data type that is 8 bytes in size. Not to be confused with an octet, which has been commonly used to describe 8 bits of data. In this document, the term byte, rather than octet, is used to describe 8 bits of data.

**3.2.10 quadlet:** A data format or data type that is 4 bytes in size.

## 3.3 Numerical values

Decimal, hexadecimal, and binary numbers are used within this document. 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 "$11010_2$".

## 3.4 Field names

This document describes values that are packetized or located in MAC-resident registers. For clarity, names of these values have an italic font and contain the context as well as field names, as illustrated in Table 3.1.

**Table 3.1—Names of registers and fields**

| Name | Description |
|---|---|
| thisState.levelAB | A register within the MAC |
| informState.accounts[n].rateB.c | A congestion indication transported within a packet |

Note that run-together names (like "*thisState*") are preferred because they are more compact than under-score-separated names (like "*this_state*"). The use of multiword names with spaces (like "This State" is avoided, to avoid confusion between commonly used capitalized key words and the capitalized word used at the start of each sentence.

## 3.5 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 illustrated as 4-byte groups, as illustrated in Figure 3.1. For multibyte objects, the first-through-last data bytes are the most-through-least significant respectively.

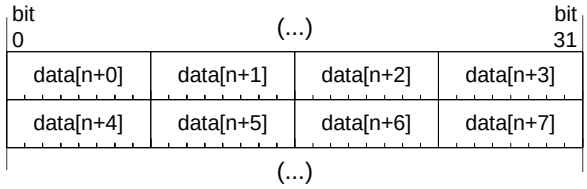



**Figure 3.1—Byte and bit ordering**

The data-byte transmission order is left-to-right within each cycle and top-to-bottom between cycles, as is consistent with the flow of English language documentation. For consistency, bits and bytes are numbered in the same fashion.

## 3.6 C code notation

The behavior of data-transfer command execution is frequently specified by C code, such as Equation 1.1. To differentiate such code from textual descriptions, such C code listings are formatted using a fixed-width Courier font. Similar C-code segments are included within some figures.

```
// Return maximum of a and b values
Max(a,b) {
  if (a<b)
    return(LT);
  if (a>b)
    return(GT);
  return(EQ);
}
```

**1.1**

Since the meaning of many C code operators are not obvious to the casual reader, their meanings are summarized in Table 3.2.

**Table 3.2—C code expressions**

| Expression | Description |
|---|---|
| `~i` | Bitwise complement of integer *i* |
| `i^j` | Bitwise EXOR of integers *i* and *j* |
| `i&j` | Bitwise AND of integers *i* and *j* |
| `i<<j` | Left shift of bits in *i* by value of *j* |
| `i*j` | Arithmetic bmultiplication of integers *i* and *j* |
| `!i` | Logical negation of Boolean value *i* |
| `i&&j` | Logical AND of Boolean *i* and *j* values |
| `i\|\|j` | Logical OR of Boolean *i* and *j* values |
| `i^= j` | Equivalent to: `i= i^j`. |
| `i==j` | Equality test, true if *i* equals *j* |
| `i!=j` | Equality test, true if *i* does not equal *j* |
| `i<j` | Inequality test, true if *i* is less than *j* |
| `i>j` | Inequality test, true if *i* is greater than *j* |

# 4. Acronyms and abbreviations

# 5. Media reference and service model

## 5.1 Scope

## 5.2 Overview of MAC services

### 5.2.1 Class-A service

The MAC provides a class-A service with guaranteed bandwidth and low jitter specifications. This class is intended to allow the client to implement a synchronous traffic class. The MAC is responsible for policing class-A traffic to ensure that provisioned service parameters are not violated; therefore class-A traffic need not be shaped by the client.

The MAC provides mechanisms for provisioning class-A traffic (see clause 11.3) and ensures that the provisioned bandwidths never exceed link capacities. Since the levels of sustainable class-A traffic are limited, requests for class-A allotments may sometimes be rejected. This forces release of other provisioned bandwidths, to avoid continued bandwidth-request rejections, is beyond the scope of this RPR specification

The service access point provides an indication to the MAC client of the status of the underlying channel, indicating where there are policing constraints enforced by the MAC and traffic over this path cannot currently be accepted.

### 5.2.2 Class-B service

The MAC provides a class-B service with guaranteed bandwidth and bounded delays specifications. This class is intended to allow the client to implement a guaranteed traffic class (GTC). As is true for class-A traffic, the MAC is responsible for policing class-B traffic to ensure that provisioned service parameters are not violated; therefore class-B traffic need not be shaped by the client.

The service access point provides an indication to the MAC client of the status of the underlying channel, indicating where there is dynamic backpressure from the media and traffic over this path cannot currently be accepted.

### 5.2.3 Class-C service

The class-C service is provided to implement a best effort traffic class. The class-C traffic passes through the lower-priority transmit-path FIFO so that, once accepted, the bounded delays for class-B and class-C traffic are the same. The MAC is responsible for enforcing weighted fairness, therefore class-C traffic need not be shaped by the client. The allocation of fairness weights is beyond the scope of this specification.

The service access point provides an indication to the MAC client of the status of the underlying channel, indicating where there is dynamic backpressure from the media and traffic over this path cannot currently be accepted.

# 6. MAC data paths

## 6.1 MAC interface functionality

The MAC interface definition provides interfaces to a pair of attachment points, as illustrated in Figure 6.1. The attachment interface receives data and FIFO fill-level indications from the client; the attachment transmits data and transmit-permission information.

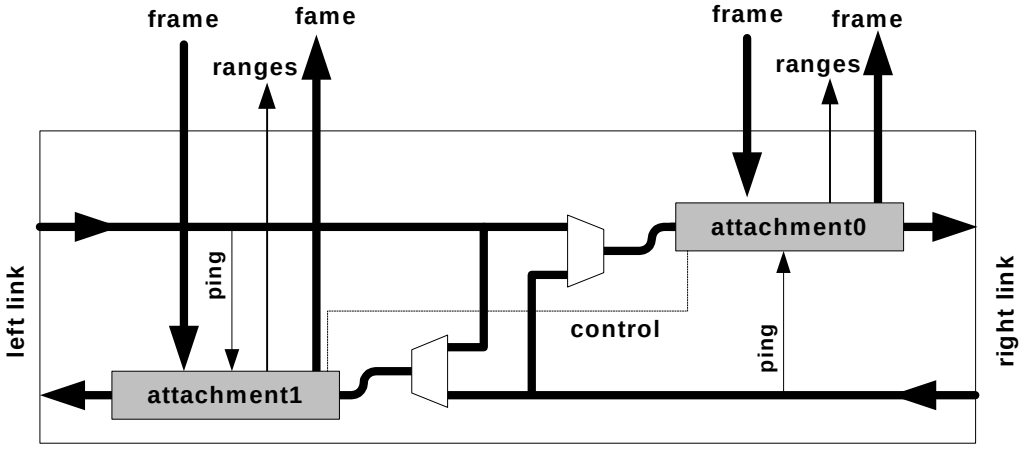



**Figure 6.1—MAC interface signals**

The data paths are expected to flow through two multiplexers, whose controls are semistable in that they change during protection events, rather than on a packet-by-packet basis.

The client is responsible for selecting the interface over which the data packet is sent. Deferring this decision to the client, rather than the MAC, provides flexibility to select the attach point based on a wide range of client-managed parameters, including queue levels and priorities.

## 6.2 Incoming frame processing

The processing of incoming frames depends on the validity of the packet and its contents. If the ***headerCrc*** is invalid, the frame shall be immediately discarded. Otherwise, the *timeToLive*, *destinationStationID, sourceStationID*, *destinationMacAddress*, and *sourceMacAddress* fields affect the frame processing, as specified in Table 6.1 and following row-by-row descriptions.

**Table 6.1—Incoming frame processing**

| TTL | destination StationID | source StationID | flood | destination MacAddress | source MacAddress | Row | Action | Route |
|---|---|---|---|---|---|---|---|---|
| — | — | — | — | — | — | 6.1.1 | discard* | strip |
| ≥1 | — | — | — | matchMac | anyMac | 6.1.2 | accepted | |
| | matchID | — | — | NONE | NONE | 6.1.3 | mapped | |
| | | — | — | multicastMac | anyMac | 6.1.4 | multicheck | |
| | | — | — | unicastMac | anyMac | 6.1.5 | unicheck# | |
| — | — | matchID | — | — | — | 6.1.6 | discard* | |
| — | — | — | — | anyMac | matchMac | 6.1.7 | | |
| ≤1 | — | — | — | — | — | 6.1.8 | | |
| >1 | — | — | — | multicastMac | anyMac | 6.1.9 | multicheck | pass |
| | — | — | 1 | unicastMac | anyMac | 6.1.10 | unicheck# | |
| | — | — | — | — | — | 6.1.11 | ignored | |

Notes:
* Error condition should be logged
# Non-bridge stations discard these frames

**Row 6.1.1:** A corrupted frame, with an invalid header-CRC is invalid, shall be discarded.

**Row 6.1.2:** Any frame with matching destinationMacAddress is stripped and copied.
**Row 6.1.3:** A *destinationStationID*-matching frame is stripped; a mapped MAC addresses is assigned.
**Row 6.1.4:** A *destinationStationID*-matching multicast is stripped & checked for multicast matches.
**Row 6.1.5:** A *destinationStationID*-matching unicast frame is stripped.
Bridges check these frames for possible forwarding to remote locations; nonbridges discard these frames.

**Row 6.1.6:** A sourceID-matching frame is stripped&discarded at its source station; an error is logged.
**Row 6.1.7:** A sourceID-matching frame is stripped&discarded at its source MAC; an error is logged.
**Row 6.1.8 :** A will-become-zero *timeToLive* field is stripped&discarded; an error is logged.

**Row 6.1.9:** A different-ID multicast frame is copied and checked for multicast matches.
The time-to-live field is decremented as the packet passes through the station.
**Row 6.1.10:** A different-ID unicast frame is copied by bridges and checked for unicast matches.
Bridges check these frames for possible forwarding to remote locations; nonbridges discard these frames.
The time-to-live field is decremented as the packet passes through the station.
**Row 6.1.11:** The time-to-live field is decremented as the packet passes through the station.

The integrity of the payload-CRC value has no effect on its routing decision, but affects error logging and stomped-CRC processing, as further described in 11.2.1.

# 7. MAC client interface

# 8. Frame formats

## 8.1 Packet framing

The physical layer is expected to provide first-byte and final-byte framing of packets, as illustrated in Figure 8.1.

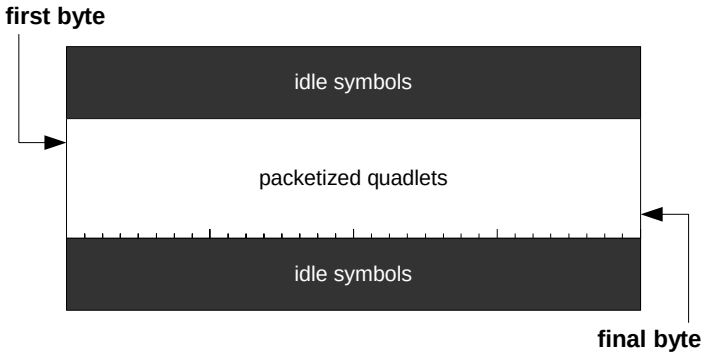



**Figure 8.1—Packet framing**

The physical layer may also provide other services, including the following:

1) Rate matching. Insertion and/or deletion of between-packet symbols, as necessary to compensate for drifts between transmit and receive clocks. MAC-level support (see Annex B) is possible when this capability is not supplied by the PHY.
2) Timer synchronization. Primitives for maintaining accurate clock synchronization between attached clock-master and clock-slave station. MAC-level support (see I.1) is possible when this capability is not supplied by the PHY.
3) Fault monitoring. Primitives for maintaining accurate clock synchronization between attached clock-master and clock-slave station (see Annex I).

## 8.2 Frame formats

### 8.2.1 Complete header frames

A frame consists of header and optional payload components, both of which are CRC protected, as illustrated in Figure 8.2

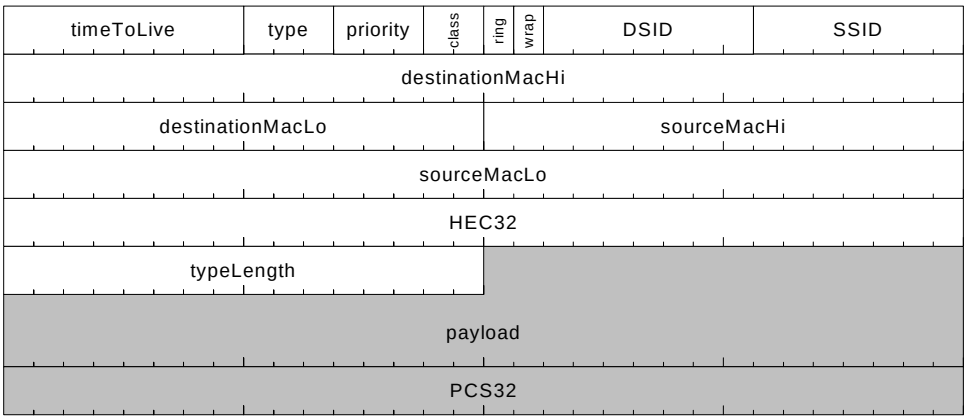



**Figure 8.2—Packet header format**

The 8-bit ***timeToLive*** field is initially set to the all one's value of 255; this value is decremented when passing through each station. Stations shall strip and discard frames received with a *timeToLive* value of 0.

NOTE —The behavior of the timeToLive field is slightly different but simpler than the behavior of a like-named field within IP packets. The intent is to simplify frame decoding operations, by decoupling the frame-discard decision from the address-match conditions.

NOTE —The *timeToLive* value of zero is illegal and (in the absence of an error) would never be observed.

The 3-bit ***type*** field specifies the frame type (format and function), as specified in Table 8.1.

**Table 8.1—*type* field values**

| Value | Name | Row | Description |
|---|---|---|---|
| 0 | DISCOVERY | 8.1.1 | Discovery and flow-control information |
| 1 | SURVEY | 8.1.2 | Provisioning survey information |
| 2 | PING_REQ | 8.1.3 | Ping request |
| 3 | PING_RES | 8.1.4 | Ping response |
| 4-5 | — | 8.1.5 | Reserved |
| 6 | FLOOD | 8.1.6 | Flooded data |
| 7 | DIRECT | 8.1.7 | Directed data |

NOTE —These format values and their meanings are subject to change as needed to identify distinct 802.17 type values.

The 3-bit ***priority*** field value is generated by the source client and transported though the MAC for the convenience of the destination client. The intent is to facilitate prioritized destination-client processing without mandating processing of data-payload fields.

The 2-bit ***class*** field values specify the class of RPR traffic, as specified in Table 8.2. The CLASS_A0 and CLASS_A1 values identify the proactive and reactive class-A traffic respectively. The CLASS_A label identifies reactive class-A traffic; the CLASS_BC identifies lower-class CLASS_B and CLASS_C traffic.

**Table 8.2—class field values**

| Value | Name | Description |
|---|---|---|
| 0 | CLASS_A0 | Proactive class-A traffic |
| 1 | CLASS_A1 | Reactive class-A traffic |
| 2 | CLASS_B | Class-B traffic |
| 3 | CLASS_C | Class-C traffic |

The ***ring*** bit values of 0 and 1 indicate the packet was sourced on ring-0 and ring-1 respectively. The ***wrap*** bit values of 0 and 1 indicate the packet shall be discarded or sustained at wrap points.

The 7-bit ***DSID*** field and the 7-bit ***SSID*** field identify the destination and source stations respectively.

NOTE —The persistent ***DSID*** and ***SSID*** identifiers are unique on each ring; they are assigned during the discovery process (see xx).

The 32-bit ***HEC32*** value is CRC value that protects the aforementioned header parameters. A standard 32-bit CRC protocol is specified; see Annex H for details.

### 8.2.2 Compact frames

A compact frame consists of header and optional payload components, both of which are CRC protected, as illustrated in 8.3.

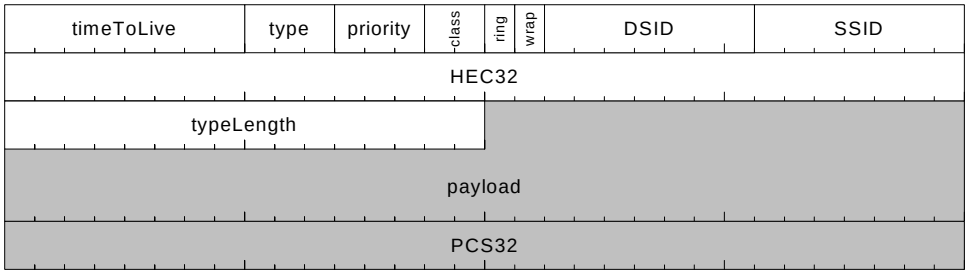



**Figure 8.3—Packet header format**

The 8-bit ***timeToLive*** field, the 3-bit ***type*** field, the 3-bit ***priority*** field, the 2-bit ***class*** field, the ***ring*** bit, the ***wrap*** bit, the 7-bit ***DSID*** field, and the 7-bit ***SSID*** field are specified in 8.2.2.

NOTE —The type field differentiates between compact and complete frame formats.

The 32-bit ***HEC32*** value is CRC value that protects the aforementioned header parameters. A standard 32-bit CRC protocol is specified; see Annex H for details.

## 8.3 Payload formats

### 8.3.1 Ethernet payloads

Ethernet payloads start with a 16-bit ***typeLength*** field that specifies the format and function of the remaining fields, as illustrated in Figure 8.4.

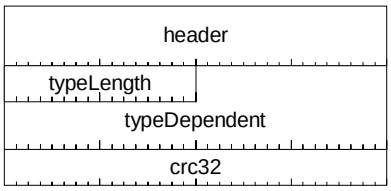



**Figure 8.4—Ethernet payloads**

NOTE —By convention, the lower values (les than $1536_{10}$ or $600_{16}$) specify the length of an Ethernet frame. The larger values specify the payload format, based on numerical-value assignments registered with the IEEE/RAC.

### 8.3.2 Ping payloads

NOTE —The ping format is subject to change based on requirements identified while refining other clauses.

The ping frames consist of header and payload components, both of which are CRC protected, as illustrated in Figure 8.5. Distinct RPR frame-type codes differentiate these ping frames from other RPR frame types.

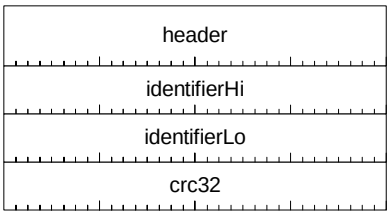



**Figure 8.5—Ping frame formats**

The 32-bit ***identifierHi*** and ***identifierLo*** fields are copied from the request into the response, with the intent of being used by the client to affiliate the returned response with the transmitted request. The format of these fields is implementation dependent and beyond the scope of this standard.

### 8.3.3 Discovery payloads

(see John Lemon minutes of Discovery&Protection task force)

### 8.3.4 Survey payloads

NOTE —The survey format is subject to change based on requirements identified while refining other clauses.

For survey frames, an array of provisioned-level information represents an accumulation of range-dependent information, as illustrated in Figure 8.6. The ***info[0]*** through ***info[total-1]*** entries communicate provisioned resource levels on each of the attached links. Fields within each of these *info[n]* components are described below.

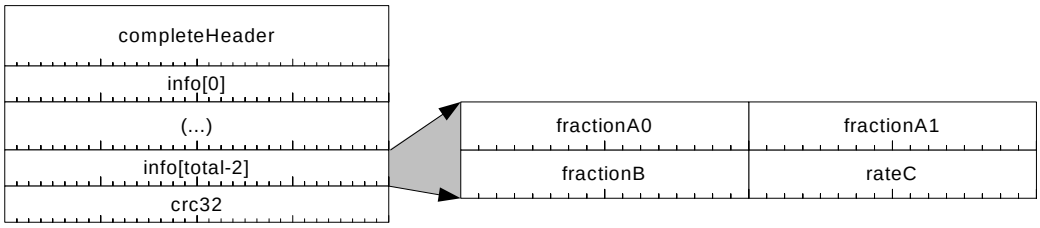



**Figure 8.6—Communicated congestion information**

The 16-bit *rprSurvey* value equals **RPR_SURVEY0** during the primary survey phase, when cumulative provisioned levels are accumulated. The 16-bit *rprSurvey* value equals **RPR_SURVEY1** during the secondary survey phase, when cumulative provisioned levels are distributed.

The 16-bit ***rateA0***, 16-bit ***rateA1***, and 16-bit ***rateB*** values correspond to the provisioned level of class-A0, class-A1, and class-B traffic respectively. The 16-bit ***rateC*** values correspond to the provisioned weight of class-C traffic.

# 9. Media access control (flow control)

## 9.1 Flow control overview

### 9.1.1 Traffic classes

Flow control protocols are based on the policing of offered traffic, based on the class of the traffic and the provisioned bandwidths. Higher level protocols are expected to further partition bandwidth restrictions of flows from within a station, based on per-flow service level agreements (SLAs) maintained within that station. However, the use of within-the-station per-flow restrictions is beyond the scope of this standard.

Flow control protocols involve limiting transmissions of class-A, class-B, and class-C traffic. The objectives are to achieve desired bandwidth partitioning without compromising bandwidth efficiencies or spatial-reuse opportunities. Strategies for achieving these goals are listed below:

1) Class-A. Two interoperable options are used to support class-A traffic:
   a) Reactive. Acquiring dynamic class-A0 bandwidth involves sending a congestion indication (a transitBC-depth indication) to throttle excessive upstream transmissions.
   b) Proactive. Preserving static class-A1 bandwidth involves sustaining a nominal class-A1/idle transmission rate to the (possibly needy) downstream station.

   Stations can exhibit both efficient reactive and sufficient proactive behaviors, depending on their transit-queue sizes; large transitBC FIFOs are required to support large levels of class-A0 traffic.
2) Class-B. The class-B transmissions are throttled to prenegotiated levels, while deferring to higher-priority class-A transmissions.
3) Class-C. The class-C transmissions are throttled to weighted fairness levels, while deferring to higher-priority class-A and class-B transmissions.

Different congestion management protocols are applied to class-A, class-B, and class-C traffic, as each traffic class has a distinct set of performance requirements. The class-A traffic is the most demanding, with strict bandwidth and latency guarantees. The class-C traffic has no guarantees, but attempts to ensure the residual best-effort bandwidth is efficiently and fairly allocated.

The arbitration indications are level-sensitive signals, rather than tokens or edge-sensitive values, making the protocols robust. Most importantly, from a simplicity perspective, these arbitration indications are fault tolerant, in that special fault-recovery protocols are unnecessary.

Sophisticated clients are expected to maintain a topology table, to assist in identifying the station locations (measured in hop counts) based on their unique station identifiers. This information is sufficient to select frames for few-hop transmissions, while avoiding selection of blocked many-hop transmissions.

The arbitration indications flow in the reverse direction, with respect to the data-frame flows, starting from stations currently requesting their share of the bus bandwidth. The reverse-flow direction allows inactive stations to delay forwarding of arbitration indications while filling of their *transitBC* transit buffer generates idles; stations which cannot generate idles quickly forward arbitration indications to throttle upstream stations.

## 9.2 Flow control components

Flow control involves a cooperative effort between MAC-resident components and client-managed queues, as illustrated in Figure 9.1. Whenever possible, the transmission queues are located in the client, as opposed to the MAC, so that packets can be conveniently resorted, inserted, or deleted while awaiting transmission. The client-level queue management protocols are application dependent and beyond the scope of this standard, although standard queue-gating indications are passed across the MAC-to-client interface.

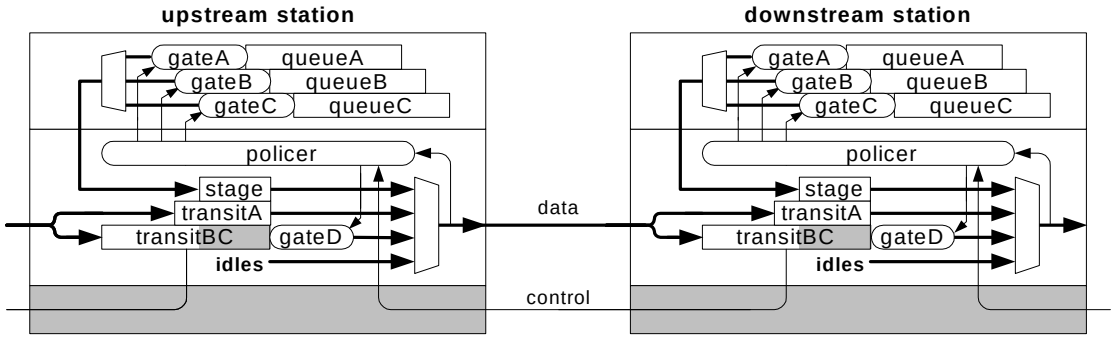



**Figure 9.1—Flow control components**

The client is expected to provide ***queueA***, ***queueB***, and ***queueC*** storage for holding class-A, class-B, and class-C traffic respectively. These queues have ***gateA***, ***gateB***, and ***gateC*** functions, which inhibits client-to-MAC packet transmissions based on the MAC provided flow-control signals; a similar ***gateD*** component limits the transitBC retransmissions based on such flow-control signals. These gate components are have the following responsibilities:

1) The ***gateA*** components polices class-A *queueA* traffic, to ensure that transmission rate *ra* (class-A transmission rate) never exceeds the provisioned *rateA* traffic rate.
2) The ***gateB*** component polices *sendB* transmission rates, blocking class-B transmissions when:
   a) A class-A transmission is enabled.
   b) A congestion condition is sensed in this or the downstream station.
   c) The class-B transmission rate would exceed its provisioned limit.
3) The ***gateC*** component polices *sendC* transmission rates, blocking class-C transmissions when:
   a) A class-A or class-B transmission is enabled.
   b) A congestion condition is sensed in this or the downstream station.
   c) The class-C transmission rate would exceed its provisioned limit.
4) The ***gateD*** component polices *transitBC* retransmission rates, so that:
   a) Dynamic idle bandwidth is provided for provisioned class-A0&class-A1 traffic.
   b) Static idle bandwidth is provided to sustain provisioned class-A1 traffic.

The ***stage*** storage (typically 2 MTUs) holds ready-to-send packets in high-speed MAC-resident buffers, to ensure its reliable delivery independent of the client-to-MAC bandwidth and flow-control latencies.

The ***transitA*** storage (a few MTUs) holds class-A traffic that arrives and cannot be immediately retransmitted, due to a previously committed transmission or a retransmission of another frame.

The ***transitBC*** storage (many MTUs) holds class-B and class-C traffic that arrives and cannot be immediately retransmitted, due to a class-A transmission, a previously committed transmission, or a retransmission of another frame. Although large transitBC FIFOs are not required, larger sizes increase the levels of available class-A0 bandwidths.

The MAC/client interface supplies flow-control indications to client-queue gates, as illustrated in Figure 9.1. These control indications include the following:

1) The ***rangeA0*** and ***rangeA1*** indications allow the gateA component to limit class-A transmissions.
2) The ***rangeB*** indications allow the gateB component to limit class-B transmissions.
3) The ***rangeC*** indications allow the gateC component to limit class-C transmissions.

## 9.3 Transmit selections

Within the MAC, the enabled-buffer selection is based on the need to support lossless class-A transmissions, as specified in Table 9.1.

**Table 9.1—Transmit selections**

| conditions | | | Row | select value | Description |
|---|---|---|---|---|---|
| transitA | stage | transitBC | | | |
| not empty | — | — | 9.1.1 | TRANSIT_A | Always prepare for class-A |
| empty | not empty | — | 9.1.2 | STAGE | Always transmit client-supplied traffic |
| empty | empty | not empty | 9.1.3 | TRANSIT_BC | Throttled retransmissions |
| empty | empty | empty | 9.1.4 | IDLES | Sustaining idles for downstream station |

**Row 9.1.1:** The class-A transit FIFO is emptied first, to enable further class-A transmissions.
**Row 9.1.2:** The stage buffer has precedence over lower-class transit-FIFO retransmissions; prioritization and policing policies are applied when the frame staged and never reapplied thereafter.

**Row 9.1.3:** The *transitBC* FIFO retransmissions continue deffered transitBC retransmissions.
**Row 9.1.4:** Transmit *idles* when no transmissions or retransmissions are possible.

## 9.4 Stop conditions

The *stopA*, *stopB*, and *stopC* indications block all class-B, class-B, and class-C transmissions, independent of the hop-count distance of the traffic. Each stop condition is true if any subcondition is true; these subconditions are specified in Table 9.2.

**Table 9.2—stopCondition values**

| Description | Conditions | Row | Result |
|---|---|---|---|
| Stopped when staging buffer is nearly full | stageNearFull | 9.2.1 | stopA=1 |
| Class-A transmissions have precedence | stopA | 9.2.2 | stopB=1 |
| Idles needed to sustain downstream class-A0/A1 bandwidth | creditD>0 | 9.2.3 | |
| Idles needed to sustain downstream class-A1 bandwidth | creditD1>0 | 9.2.4 | |
| Retransmissions needed to bound transitBC delays | creditBC<0 | 9.2.5 | |
| Class-B transmissions have precedence | stopB | 9.2.6 | stopC=1 |
| Class-C transmission rates have been exceeded | creditC>0 | 9.2.7 | |

## 9.5 MAC-to-client range indications

Transmissions are policed to ensure that the consumed bandwidths never exceed the provisioned rates for each hop taken by the frame. Policing determines the range values passed to the client, for use as transmit permissions, as specified in Table 9.3 (*stopCondition* values within this tables are specified in Table 9.2.)

**Table 9.3—Range indication values**

| condition1 | Condition2 for n<=N | Row | Result | Description |
|---|---|---|---|---|
| stopA=0 | creditA0[n] >= 0 | 9.3.1 | rangeA0=n | Class-A permissions are range-dependent |
| | creditA1[n] >= 0 | 9.3.2 | rangeA1=n | |
| stopA=1 | — | 9.3.3 | rangeA0=0 | Class-A permissions are nullified |
| | | 9.3.4 | rangeA1=0 | |
| stopB=0 | creditB[n] >= 0 | 9.3.5 | rangeB=n | Class-B permissions are range-dependent |
| stopB=1 | — | 9.3.6 | rangeB=0 | Class-B permissions are nullified |
| stopC=0 | (limitC[n]-countC[n]) > 0 | 9.3.7 | rangeC=n | Class-C permissions are range-dependent |
| stopC=1 | — | 9.3.8 | rangeC=0 | Class-B permissions are nullified |

**Row 9.3.1: Row 9.3.2: Row 9.3.5: Row 9.3.7:** Transmissions are limited to positive-credit hops.
**Row 9.3.3: Row 9.3.4: Row 9.3.6: Row 9.3.8:** Transmissions are sometimes blocked.

## 9.6 Rate shaping

### 9.6.1 Rate-shaping updates

Rate limiting components are used throughout the MAC. Depending on the component, the *sendRate*, *sendSize*, *waitSize*, *hiSide*, or *loSide* values may be specified in terms of static rate-provisioned or dynamic load-dependent parameters. The rate-limit is a classic leaky-bucket protocol, illustrated in Figure 9.2, described in this subclause and formally specified by the *CreditUpdate(…)* routine within Annex K.

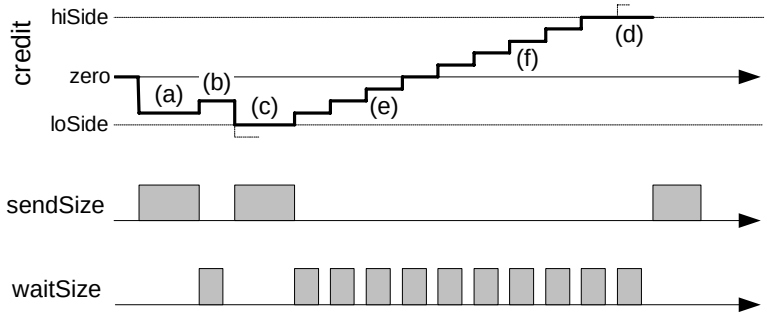



**Figure 9.2—Rate-limiting class-A traffic**

Changes in credit are affected by observed *sendSize* and *waitSize* counts. The credit value is decreased (a) by *sendSize* when a nonzero *sendSize* value is observed. The sendSize is typically associated with a transmit-frame length.

The credit value is increased by *sendRate*×*waitSize* when a nonzero *waitSize* value is observed. The *waitSize* prameter is typically associated with competing frame retransmission, a passing of time, or an idle frame-equivalent transmission.

The *loSide* parameter restricts (c) the *credit* value's negative excursion. The *hiSide* parameter restricts (d) the *credit* value's positive excursion.

While the credit value remains negative (e), *sendSize* affiliated transmissions are typically inhibited; when the credit value becomes positive (f), *sendSize* affiliated transmissions are typically re-enabled.

### 9.6.2 Rate-shaping parameters

Flow-control protocols utilize multiple rate-shaping components, as described in Table 9.4. Some of the more-complicated parameter derivations are derived in Table 9.5; an italics font is used to clearly identify these entries with Table 9.5 dependencies.

**Table 9.4—Rate shaper parameters**

| rate | sendSize | waitSize | hiSide | loSide | Row | Update |
|---|---|---|---|---|---|---|
| rateA0[n] | moveA0 | waitT | *hiSideA0* | –DTU | 9.4.1 | creditA0[n] |
| rateA1[n] | moveA1 | waitT | *hiSideA1* | –DTU | 9.4.2 | creditA1[n] |
| rateB[n] | moveB | waitT | *hiSideB* | –DTU | 9.4.3 | creditB[n] |
| scaleC | -na- | *waitSizeD* | limitC[n]+DTU | lowerC[n] | 9.4.4 | countC[n] |
| ratingA | sendI+sendA0+sendA1 | sendB+sendC | *hiSideD* | –DTU | 9.4.5 | creditD |
| ratingA1 | sendI+sendA1 | sendA0+sendB+sendC | *hiSideD1* | –DTU | 9.4.6 | creditD1 |
| rateBC | moveB+moveC | sendBC | *hiSideBC* | *loSideBC* | 9.4.7 | creditBC |
| rateC | moveC | waitT | DTU | –DTU | 9.4.8 | creditC |

The *rateA0[i]*, *rateA1[i]*, and *rateB[i]* values represent the provisioned class-A0, class-A1, and class-B transfer rates over segment *i*. The *ratingA* and *ratingA1* values represent the provisioned levels of ringlet bandwidth, for class-A (class-A0 plus class-A1) and class-A1 respectively. The *rateBC* value represents the current rate limit for the client-to-MAC lower-class (class-B or class-C) transfers. The *rateC* value represents the upper-limit rate for class-C traffic (the effective rate of class-C traffic is also limited by fairness constraints, as described in 9.8).

The *moveA0*, *moveA1*, *moveB*, and *moveC* represent the sizes of class A0, A1, B, and C client-to-MAC transfers respectively. The *sendA0*, *sendA1*, *sendB*, *sendC*, and *sendI* values repesent the sizes of class A0, A1 B, and C transfers respectively. The *waitT* values represents the time since the previous credit-value update.

The DTU value equals (1+*n*)*MTU, where n is the number of additional MTUs that may be sent by the client after the stop indication is provided across the MAC-to-client interface. The value of *hops* represents the number of active stations attached the ringlet.

**Row 9.4.1:** The ***creditA0[i]*** value enforces provisioned ringlet-bound class-A0 bandwidths.
**Row 9.4.2:** The ***creditA1[i]*** value enforces provisioned ringlet-bound class-A1 bandwidths.
**Row 9.4.3:** The ***creditB[i]*** value enforces conformance to provisioned ringlet-bound class-B bandwidths.

**Row 9.4.4:** The ***countC[n]*** values account for the cumulative class-C transmissions, expressed as a scaled bytes-transmitted value. The maximum *countC[n]* value is based on the choke-point supplied *limit[n]* value, with a DTU overrating to account for client-to-MAC flow-control signaling delays. The minimum *countC[n]* value is based on the observed *lower[n]* value, which represents a ringlet-loop-delayed *limit[n]* value from the past.

**Row 9.4.5:** The ***creditD*** value selectively blocks class-B/C transmissions and retransmissions to sustain necessary downstream reactive class-A (cumulative class-A0 plus class-A1) bandwidths.

**Row 9.4.6:** The ***creditD1*** value selectively blocks class-B/C transmissions and retransmissions to sustain necessary downstream proactive class-A0 bandwidths.

**Row 9.4.7:** The ***creditBC*** value selectively blocks sendB/sendC transmissions in favor of transitBC retransmissions. Selective transmission blocking also bounds the worst-case pass-through time, by enforcing interleaved retransmissions during high transmission-rate intervals.

To facilitate a smooth transition between these extremes, the class-B/C transmissions are rate limited, where the rate is dependent on the transitBC queue depth, as illustrated in Figure 9.3. The intent is to gradually reduce class-B and class-C transmission rates, as the transitBC FIFO fills to a half-full condition. Illustrative code is provided in Annex K, subroutine *DepthToRateBC(uInt4 depth)*.

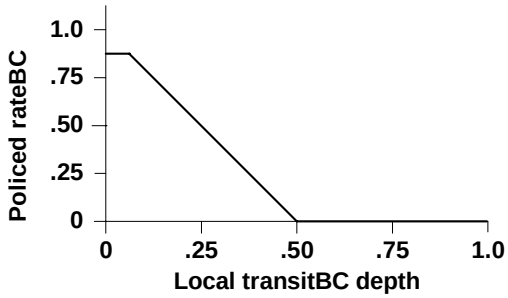



**Figure 9.3—Depth dependent *rateBC* transmission ratios**

**Row 9.4.8:** The ***creditC*** value enforces conformance to rate-limited class-C bandwidths. The maximum class-C transmission limit is intended to improve others' observed the ring behavior during changes between unloaded-and-loaded conditions. Weighted fairness between class-C active stations is handled by a distinct mechanism (see 9.7 for details).

NOTE —The *creditC* value sets a rate limit on class-C bandwidths and is optional, in the sense that setting rateC to the full link bandwidth has the effect of nullifying the *creditC* bandwidth-limiting effects.

### 9.6.3 Rate-shaping parameters

Flow-control protocols utilize multiple rate-shaping components, as described in Table 9.4.

**Table 9.5—Setting rate-shaping parameters**

| Variable | Condition | Row | Value | Description |
|---|---|---|---|---|
| hiSideA0 | — | 9.5.1 | (hops*MTU*ratingA0)/ONE+DTU | Worst-case class-A0 credits |
| hiSideA1 | — | 9.5.2 | (hops*MTU*ratingA1)/ONE+DTU | Worst-case class-A1 credits |
| hiSideB | — | 9.5.3 | 2*(loopDelay+MTU*hops)+DTU | Worst-case class-B credits |
| waitSizeC | depthTransitBC < 0.25 | 9.5.4 | waitT | Uncongested potential rate |
| | depthTransitBC >= 0.25 | 9.5.5 | moveC | Throttled transmission rate |
| hiSideD | assist==0 | 9.5.6 | 0 | Clear when not assisting |
| | assist==1 | 9.5.7 | DTU | Round-trip signaling delays |
| hiSideD1 | — | 9.5.8 | (2*MTU*hops*rateA1)/ONE+DTU | Round-trip signaling delays |
| hiSideBC | staged==0 | 9.5.9 | 0 | Clear when not transmitting |
| | staged==1 | 9.5.10 | DTU | Round-trip signaling delays |
| loSideBC | queued==0 | 9.5.11 | 0 | Clear if not retransmitting |
| | queued==1 | 9.5.12 | –DTU | Round-trip signaling delays |

**Row 9.5.1:** The ***hiSideA0*** value includes effects of jitter and client-flow-control delays.
**Row 9.5.2:** The ***hiSideA1*** value includes effects of jitter and client-flow-control delays.

**Row 9.5.3:** The ***hiSideB*** value includes effects of control-flow, jitter, and client-flow-control delays. The *linkDelays* value represents the best-case ring-circumference delay, due to speed-of-light delays in station-to-station cabling or active circuitry within the station, excluding variable delays associated with nonempty transit queues.

**Row 9.5.4:** When uncongested (transitBC FIFO is less than ¼ full) the *countC[n]* value is adjusted as though every transmissions was a class-C frame, to avoid blocking upstream class-C transmissions.
**Row 9.5.5:** When congested (transitBC FIFO is at least ¼ full) the *countC[n]* value is adjusted by the sizes of class-C frames.

**Row 9.5.6: Row 9.5.7:** For best class-A0 bandwidth utilization, positive credits (that normally inhibit class-B/C transmissions) are discarded when *assist* is zero. The assist-value specification is formalized by Figure 9.4, where *assist* is nonzero within the shaded-grey area under the graphed function. The intent is to provide assistance when the downstream-station congestion exceeds local congestion, but to delay that assistance for small levels of downstream congestion. Illustrative code is provided in Annex K, subroutine *DepthsToAssist(uInt4 thisDepth, uInt4 thatDepth).*

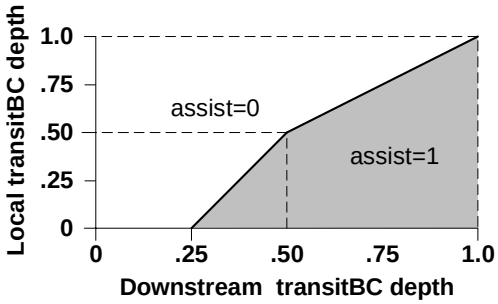



**Figure 9.4—Class-A reactive threshold**

**Row 9.5.8:** The high credit limit is designed to compensate for transient effects due coincidental multiple-station back-to-back class-A frame transmissions.

**Row 9.5.9:** Positive transmit credits are discarded when nothing is available to be transmitted.
**Row 9.5.10:** Credit limit accounts for round-trip MAC-to-client-to-MAC flow-control signaling delays .

**Row 9.5.11:** Negative transmit credits are discarded when nothing is available to be retransmitted.
**Row 9.5.12:** Credit limit accounts for round-trip MAC-to-client-to-MAC flow-control signaling delays .

## 9.7 Fair class-C transmissions

Management of class-C traffic involves blocking transmissions in preference of class-A and class-B traffic. Unblocked class-C rates are also policed to ensure conformance with published choke-point rates.

Each station continuously publishes a *limitC* value to upstream stations, communicating the run-rate of transmitted class-C traffic. Stations compare the received *limitC* value to their affiliated *creditC* value; transmissions stop when *creditC* exceeds the received *limitC* value. The intent is to let congested nodes pace the transmissions of upstream greedy stations.

The weighting factor determines the scale factor for the leaky bucket, as illustrated in Figure 9.3. The intent is to approximate a *1/x* function with a an easy-to-compute function. Illustrative code is provided in Annex K, subroutine *WeightToScaleC(uInt4 weightC).*

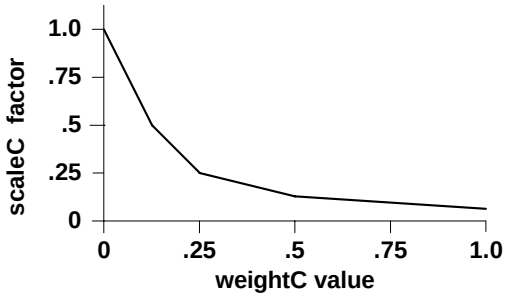



**Figure 9.5—Scale factor conversion**

When uncongested, each station continuously updates its creditC, as though every transmissions was a class-C frame, to avoid unnecessarily blocking upstream class-C transmissions. When congested, each station updates creditC based on its own class-C transmissions, to throttle greedy upstream neighbors. The transitBC FIFO depth provides the congestion indication: the node is congested when at-least ¼ full.

## 9.8 Policing state

### 9.8.1 Rate policing

Each station maintains per-ringlet rate-shaping state, to stop class-B and classC traffic as necessary to sustain class-A traffic. Only one copy of this rate-shaping state is necessary, as illustrated in Figure 9.6. Within this figure, the rate variables are shaded white and the rate constants are shaded grey.

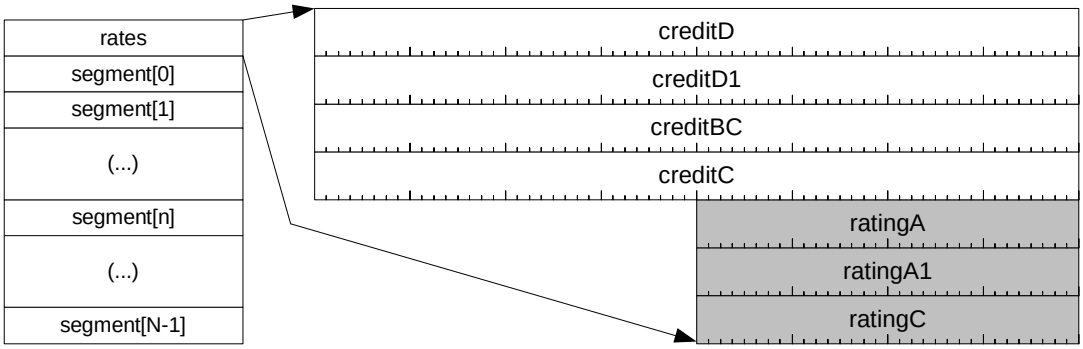



**Figure 9.6—Ratio-policing state**

The signed 64-bit ***creditD*** value selectively blocks class-B and class-C transmissions and retransmissions, as necessary to sustain class-A0/A1 traffic. The signed 64-bit ***creditD1*** value selectively blocks class-B and class-C transmissions and retransmissions, as necessary to sustain class-A1 traffic. The signed 64-bit ***creditBC*** value selectively blocks class-B and class-C transmissions, as necessary to avoid *transitBC* overflow.

The signed 64-bit ***creditBC*** value selectively blocks class-B and class-C transmissions, as necessary to avoid transitBC starvation. The signed 64-bit ***creditC*** value selectively blocks class-C transmissions, as necessary to limit maximum class-C transmission rates.

The unsigned 32-bit ***ratingA0*** value identifies the provisioned class-A0 rating of the ringlet; this effects updating *creditD* values. The unsigned 32-bit ***ratingA1*** value identifies the provisioned class-A1 rating of the ringlet; this effects updating *creditD1* values. The unsigned 32-bit ***ratingC*** value identifies the class-C rate-limit of the station; this effects updating creditC values.

### 9.8.2 Send policing

Each station maintains per-segment rate-shaping state, to stop class-A, class-A1, and class-B and traffic as necessary to remain within provisioned bandwidths. A copy of this rate-shaping state is maintained for each segment of the ringlet, as illustrated in Figure 9.7. Within this figure, the rate variables are shaded white and the rate constants are shaded grey.

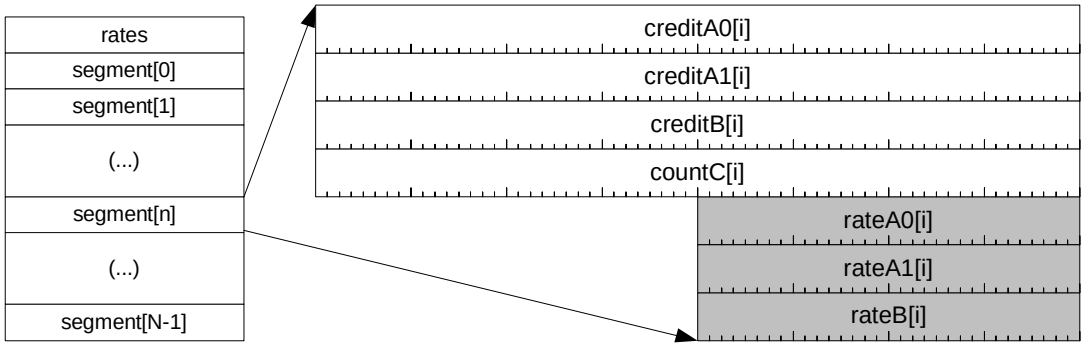



**Figure 9.7—Send-policing state**

The police state involves maintenance of N-1 accounts, where N is the number of segments assigned to the ring. The *n*'th account corresponds to a link located between the source and destination, where *n* is the number of ring-segments located between source and this link. Each account has several components, described in the remainder of this subclause.

The signed 64-bit ***creditA0[i]*** value enables or disables class-A0 traffic to segment *i*, for positive and negative values respectively. The signed 64-bit ***creditA1[k]*** value enables or disables class-A1 traffic to segment i, for positive and negative values respectively. The signed 64-bit ***creditB*** value enables or disables class-B traffic on segment *i*, for positive and negative values respectively. The wrapping 64-bit ***countC*** value enables or disables class-C traffic on this transmit link, when this station is observed to be behind or ahead of the observed choke-point *limit[i]* value.

The unsigned 32-bit ***rateA0[i]*** value specifies this source's provisioned class-A0 bandwidth over segment *i*. The unsigned 32-bit ***rateA1[i]*** values specifies this source's provisioned class-A1 bandwidth over segment *i*. The unsigned 32-bit ***rateB[i]*** value specifies this source's provisioned class-B bandwidth over this link. These update parameters are semistable, in that they remain constant until provisioned bandwidths change.

## MAC fairness

# Topology discovery

# 10. Protection

NOTE —This clause is preliminary; further specification details are required.

## 10.1 Severed link effects

### 10.1.1 Link failures

A link failure effects the routing (1) of packets that would normally pass over the affected link. A station may elect to wrap quickly, returning packets (2a) on the opposing run rather than discarding them at the filed link, as illustrated in the left of Figure 10.1.

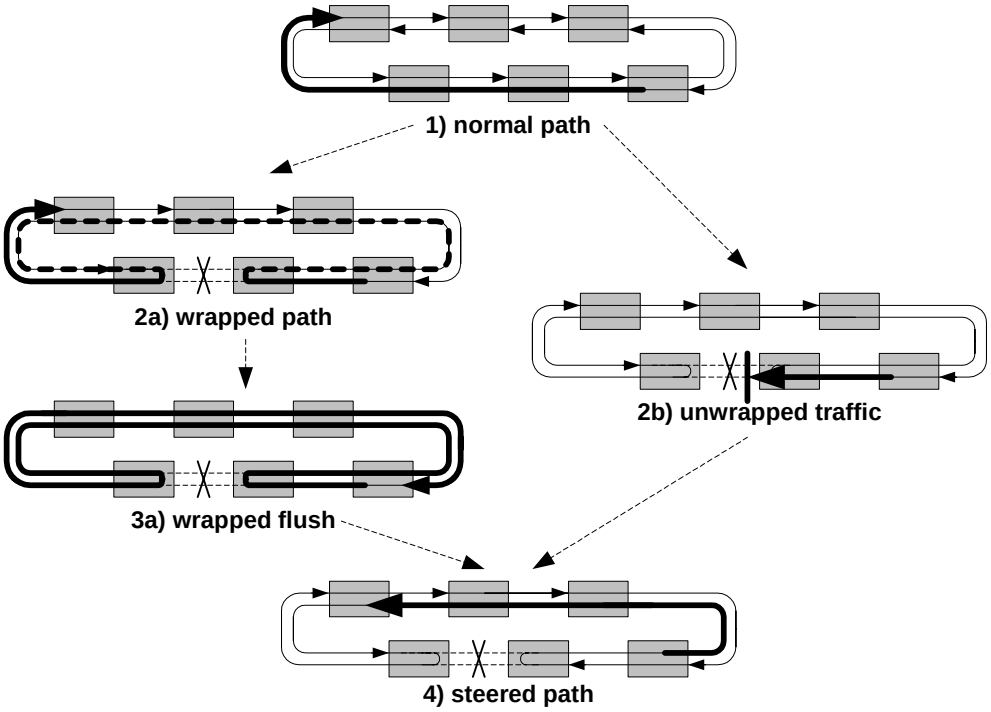



**Figure 10.1—Protection steering**

Wrapping minimizes packet loss during the protection event, but consumes excessive link bandwidth until steering is enabled thereafter. The time delay required to perform flushing (3a) also limits the usefulness of wrapping for latency-sensitive class-A traffic, since time-sensitive traffic cannot be sent during the flush operation.

Wrapping is expected to be a transient state, as steering is more efficient and part of the failed-link recover protocols (which are ultimately invoked when the failed link operation is restored). To maintain packet ordering when switching between wrapping and steering modes, outstanding traffic must be flushed before the change occurs. Packet flushing involves sending a non-class-A packet to one's self, along the wrapping path. All packets are known to be delivered, and switching between wrapping and steering is therefore safe, when this packets returns to its source.

If only steering is employed, some traffic will continue to be lost (2b) until intermediate stations become aware of the ring failure, and begin transmitting traffic on both ringlets. However, because the data packets are discarded at the failed link, no flush operation is required before steering (4) is invoked.

### 10.1.2 Link recovery

A severed link (1) is expected to be recovered, whereupon dual ringlet operations (2) become possible. Efficient utilization of these ringlets involves flushing outstanding traffic (3) before redirecting traffic (4) in the preferred direction, as illustrated in Figure 10.2.

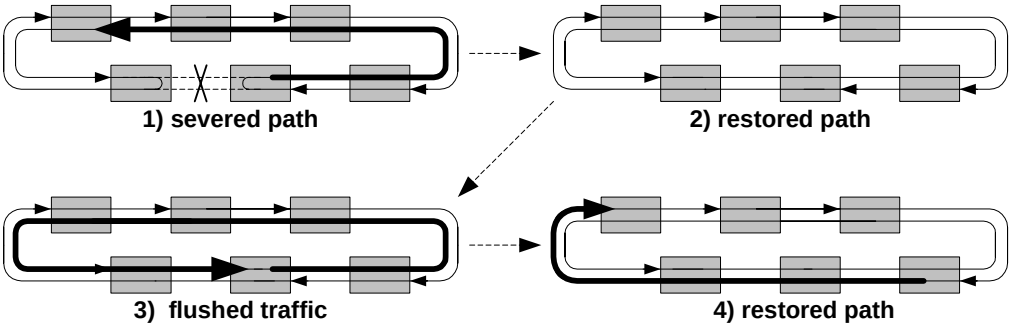



**Figure 10.2—Protection steering**

Link recovery from a wrapped mode is not supported. Instead, wrapped rings are converted to steered rings (see 10.1.1), whereupon the aforementioned link-recovery techniques can be used.

## Ringlet selection

Separate client-to-MAC interfaces are provided for each ringlet, as described in 6.1. The ringlet selection is based on which data path is used, not the content of the packet provided for transmission.

# 11. OAMP

This clause specifies the OAM&P (operations, administration, maintenance, and provisioning) functionalities for RPR stations.

## 11.1 RPR ping transaction

### 11.1.1 Ping sequences

The RPR client may request an echo operation (called a ping) to a specified destination with the intent of checking the reachability of an RPR station. This ping request generates a ping response, and that response returns on the other ring. The client has the option of sending the frame using class-A, class-B, or class-C delivery services.

A ping is a form of split-response transaction. The pin request transfers information from the requester to the responder, as illustrated in the top half of Figure 11.1. When received, the pin-request information is placed into a hold buffer, illustrated as a white square within the responder station.

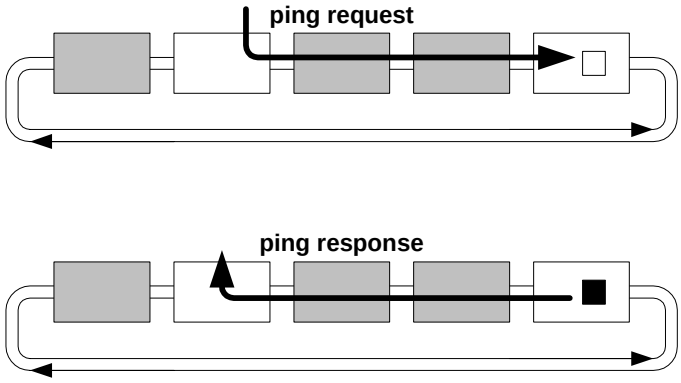



**Figure 11.1—Ping transaction sequence**

Processing of the ping-request generates a ping-response, illustrated as a black square within the responder node of Figure 11.1. The ping response is returned on the opposing run, to avoid possible link failures or protection wraps. A small amount of field transfers are necessary to convert the ping-request to a ping-response, as specified in 11.1.2.

A second ping request may be received before the first ping response has been returned. Although stations could provide additional buffers to account for such overloads, any finite sized buffer could be insufficient. Instead, only the highest priority pin is retained, where the frame with the largest sourceStationID has precedence.

TBD—Consider the support of an on-the-same-ringlet ping option.

### 11.1.2 Ping conversions

The contents of the ping-response leader is derived from the ping-request leader, as illustrated in Figure 11.2. The ***wrap*** and ***flood*** bits of the response shall be zero.

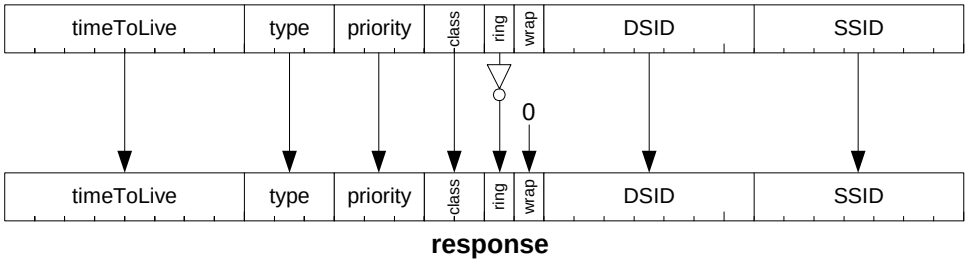



**Figure 11.2—Ping-frame leader contents**

## 11.2 CRC processing

### 11.2.1 Data CRC stomping

Cut-through frame processing allows frame payload retransmissions to begin before the frame's CRC has been verified. Store-and-forward processing may confirm a valid header-CRC but detect an invalid data-CRC. In both cases, the verified header continues circulating and the payload is marked invalid.

With this invalidation strategy, a payload transmission error causes an error to be logged and the frame's CRC set to a well defined "stomped" value. That stomped value is also an invalid CRC value, but further logging of the error condition is inhibited. These erroneous-CRC processing steps are illustrated in Figure 11.4.

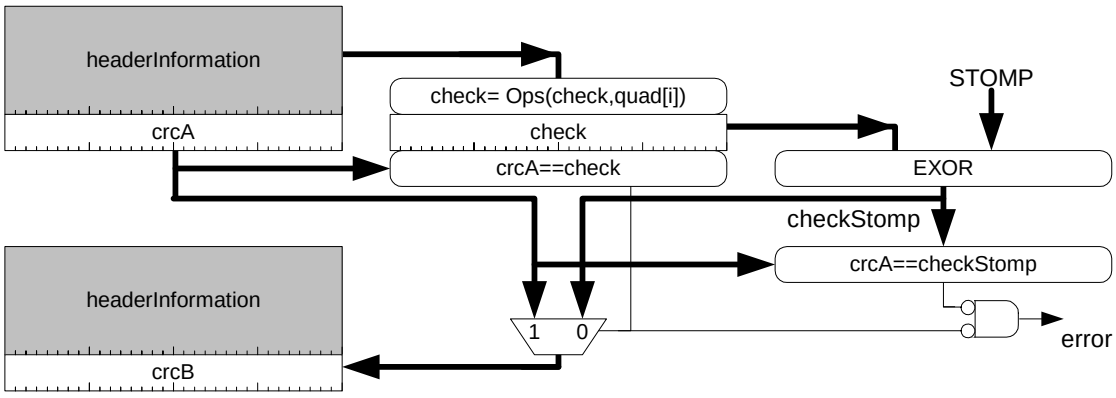



**Figure 11.3—Data CRC stomping**

A new CRC value, called check, is computed based on the frame's contents. The *checkStomp* value is computed by EXOR'ing the check value with a STOMP value (STOMP is a 32-bit constant). If the frame's *crcA* differs from the computed *check* value, the revised *crcB* value is set to the *checkStomp* value. An error condition is flagged if the frame's CRC value is incorrect (*crcA*!=*check*) and the error has apparently not been previously flagged (*crcA*!=*checkStomp*).

### 11.2.2 Protected time-to-live adjustments

The time-to-live field is normally decremented when frames pass through stations, so that corrupted frames can be discarded when the destination station is no longer present or is incorrectly identified. Decrementing the TTL field involves adjusting the following CRC field, as illustrated in Figure 11.4.

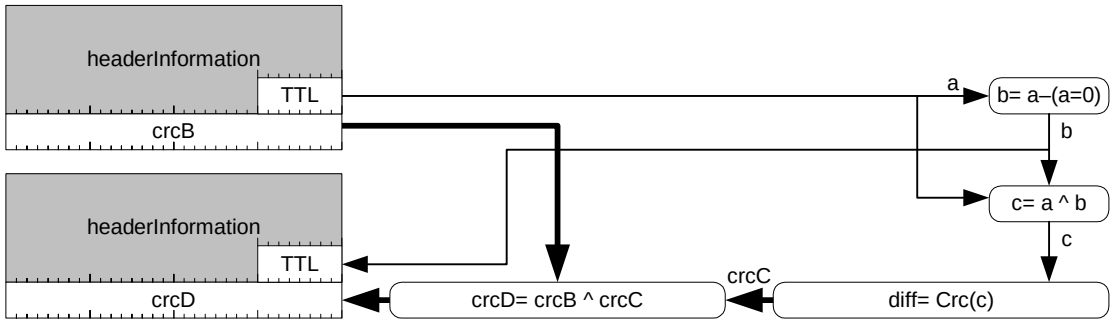



**Figure 11.4—Protected time-to-live adjustments**

An incremental update of the CRC shall be used to maintain CRC coverage when the TTL field is adjusted. This involves computing the new TTL field value *b* and the difference *c* between new and old TTL values. The difference value *c* generates an incremental CRC value *crcC*, which is EXOR'd with the old *crcB* value to generate the new *crcD* value. The data is never left unprotected: an error in *crcB* or the computation of internal CRC values will (nearly) always be reflected as an error in check value *crcD*.

## 11.3 Provisioned bandwidth

Provisioned bandwidth allocation involves computation of rate-related parameters. Rate parameters are associated with each class (A0, A1, B0, and B1) and are provisioned independently. For any specific class, illustrative definitions of provisioned parameters are presented within Figure 11.5.

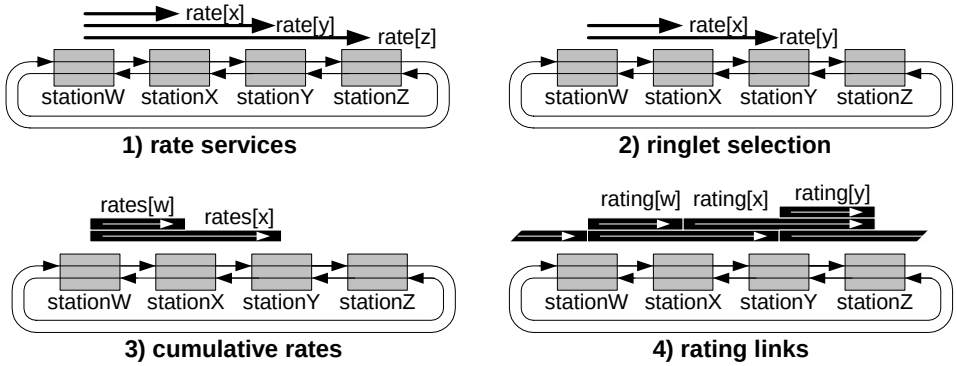



**Figure 11.5—Provisioned rate parameters**

Provisioning starts (1) with the application layer desires, to sustain different traffic rates between one node and others. Routing decisions (typically shortest path) determine which ringlet is assigned, so that portions of these rates are assigned to each ringlet. Within stationW, for example, *rate[x]* and *rate[y]* provisions are assigned (2) to the clockwise ringlet; *rate[z]* is assigned to the counterclockwise ringlet (not illustrated).

Provisioning is based (3) on a per-hop basis, based on the sum of the rate values over that hop. As examples when sending from stationW: credits over the stationW-to-stationX hop are adjusted by *rates[w*], where *rates[w]= rate[x*]+*rate[y]*; credits on the stationX-to-stationY hop are adjusted by *rates[x*], where *rates[x]= rate[y]*.

Summing of the rates, over each of the transmitting stations, yields (4) a per-hop rating. As examples: *rating[w]* is the sum of all rates associated with stationW transmissions, since no overlapping transmissions are provisioned; *rating[y]* is the sum of rates associated with stationX and stationY transmissions.

The provisioning of class-A0 traffic is based on the rated capacity of the ringlet, the maximum *rating[k]* value, where *k* is iterated over all of the attached links. If this example were applied to class-A0 traffic, then the ringlet's *rated* value equals *rating[y]*. Other stations are required to sustain this rate of class-A0 or idle (reserved for class-A0) traffic.

## 11.4 Bandwidth surveys

Bandwidth provisioning involves negotiation for guaranteed transmission bandwidths, over specified source-to-destination paths, such that the cumulative provisioned bandwidths remains below the capacity of any link located between the source and destination. Bandwidth provisioning is parameterized by the class of traffic being partitioned, with the constraint that the cumulative class-A and class-B traffic never exceeds the bandwidth guaranteed by the link.

Bandwidth surveys involve computing *ratingA0[k]*, *ratingA1[k]*, *ratingB0[k]*, and *ratingC[k]* parameters (see 11.3 for functional definitions) associated with the transmission link of station *k*. These ratings are used as follows:

1) Below capacity. Consistency checks involving *rating[k]= ratingA0[k]+ratingA1[k]+ratingB0[k]*:
   a) Consistent. Before any provisioning changes, *rating[k]* is less than the link capacity.
   b) Capacity. Before any tentative provisioning changes, *rating[k]* remains below link capacity.
2) Rated capacity. Ringlet *ratedA0* and *ratedA1* capacities are available to flow-control protocols

TBD—Define behaviors if (before the provisioning change) inconsistent *ratings* are discovered.

### 11.4.1 Bandwidth accounts

Provisioned communication between source and destination stations requires allocation of link-bandwidth resources affiliated with one or more intermediate hops, as illustrated in Figure 11.6. This provisioning is performed in a distributed fashion: each station has provisioned-bandwidth accounts (one entry for each distance) and special survey messages are provided for providing per-link provisioned-bandwidth summaries.

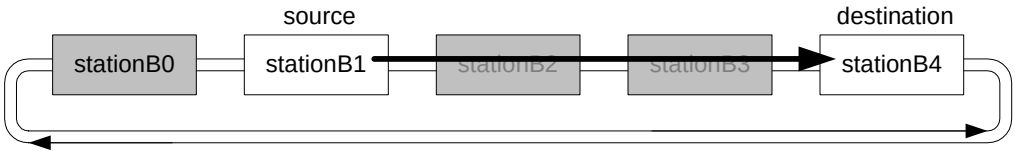



**Figure 11.6—Provisioned-bandwidth segments**

Each station keeps accounts of its provisioned class-A and class-B resource allocations, on a per-hop basis. This requires an array of storage entries, where each *rates[n]* value specifies the bandwidth provisioned for communication through *n* stations. Each entry consists of two values, corresponding to the fractional link bandwidth provisioned for class-A and class-B traffic, as illustrated in Figure 11.7.

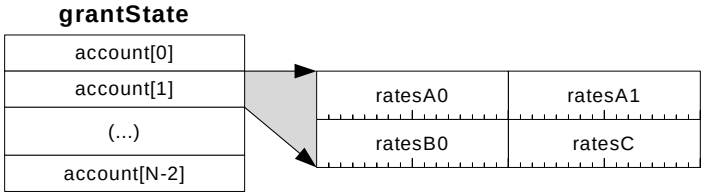



**Figure 11.7—Provisioned accounts**

The 16-bit ***ratesA0*** value is the amount of proactive class-A0 traffic provisioned to this path. The 16-bit ***ratesA1*** value is the amount of reactive class-A1 traffic provisioned over this path.

The 16-bit ***ratesB*** value is the amount of guaranteed class-B traffic provisioned over this path. The 16-bit ***ratesC*** value is the weighting of opportunistic class-C traffic over this path.

NOTE —The provisioned bandwidth numbers decrease in a monotonic fashion: the value of *rates[n+1]* is no more than *rates[n]*. This monotonic relationship is based on the pipelined nature of the traffic: all traffic passing over *n+i* hops also passes through *n* hops.

### 11.4.2 Bandwidth surveys

Each station is responsible for updating its provisioned-bandwidths accounts. These accounts can be reduced without conferring with others. However, these accounts cannot be increased without a bandwidth-survey, to verify availability of the desired bandwidths. A survey of bandwidth accounts (by *station[1]*, for example) involves sending of a bandwidth survey message through others, as illustrated in Figure 11.8.

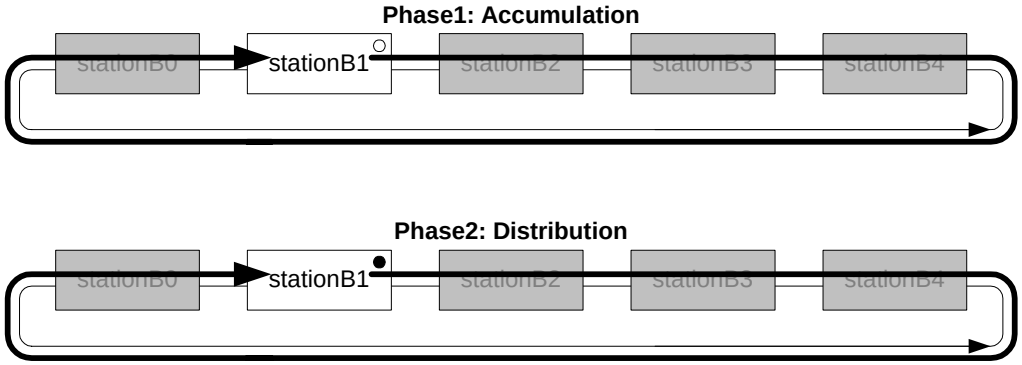
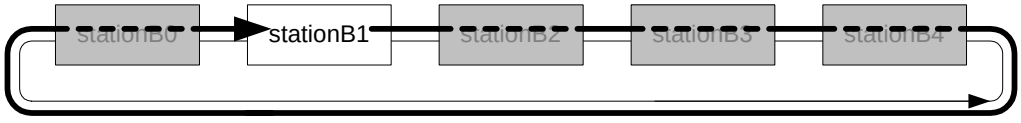



**Figure 11.8—Bandwidth surveys**

Rather than checking for availability of specific links and bandwidths, the bandwidth-check message determines the available bandwidth on a link-by-link basis, allowing the requester to make the most intelligent decisions on how that bandwidth should be allocated among multiple (possibly prioritized) subclients.

### 11.4.3 Survey-message content

Bandwidth survey messages are sent from one station to itself, but modified by all intermediate stations, as illustrated in Figure 11.9. The initial message summarizes the bandwidth accounts of the requester, listed in order of the link's distance from the source.




| | | | | | |
|---|---|---|---|---|---|
| bw_4+ | bw_0+ | bw_1 | bw_2+ | bw_3+ | bw_4+ |
| bw_0+ | bw_1+ | bw_2 | bw_3+ | bw_4+ | bw_0+ |
| bw_1+ | bw_2+ | bw_3 | bw_4+ | bw_0+ | bw_1+ |
| bw_2+ | bw_3+ | bw_4 | bw_0+ | bw_1+ | bw_2+ |
| bw_3+ | bw_4+ | bw_0 | bw_1+ | bw_2+ | bw_3+ |

**Figure 11.9—Bandwidth survey messages**

### 11.4.4 Survey message processing

To simplify the protocols, each of the intermediate stations has the same behavior, as illustrated in Figure 11.10. The ordering of the incoming entries is first rotated. These rotated values are then added to the station-provided values, either in parallel (as illustrated) or in sequential operations (not illustrated). The cumulative effect of these actions, when performed by all stations, is the return of an accurate bandwidth survey to the requesting station, in this example, *station[0]*.

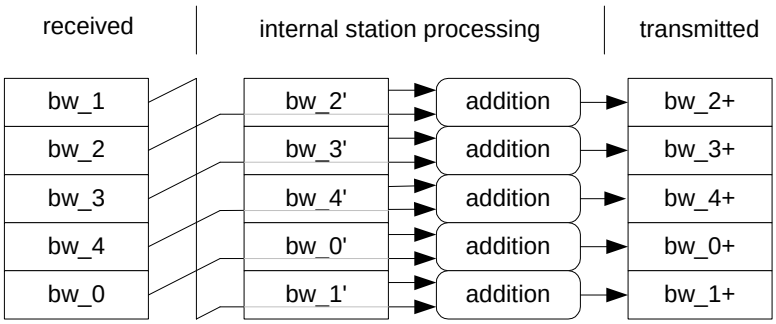



**Figure 11.10—Survey message processing**

### 11.4.5 Survey conflicts

Multiple requesters could attempt to simultaneously sample and allocate additional bandwidths. To avoid the data inconsistencies that could be caused by such conflicts, these provisioned-bandwidth messages are serialized. Conflict resolution and serialization are handled by the same mechanism: conflicts are always resolved in favor of the requester with the highest MAC address.

NOTE —The precedence of provisioning messages is based on the need to resolve circular conflicts, such as multiple stations generating messages concurrently. The intent is to break the circular deadlock by assigning asymmetric precedence values. Higher level protocols, rather than hardware-based precedence rules, are expected to resolve conflicts after attempts to overprovision available link bandwidths.

The survey state sequence involves aborting lower precedence surveys in favor of the higher precedence surveys, as specified in Table 11.1.

**Table 11.1—Survey state transitions**

| oldState | condition | Row | actions | newState |
|---|---|---|---|---|
| IDLE | SurveyRequest | 11.1.1 | — | SEND |
| SEND | — | 11.1.2 | SendSurveyA() | SENT |
| | | 11.1.3 | sample= timer | |
| SENT | peekTransitA && passID>thisID | 11.1.4 | TossTransitA() | — |
| | peekTransitA && passID<thisID | 11.1.5 | — | WAITA |
| | peekTransitA && passID==thisID | 11.1.6 | good= AdjustRates(i) | TEST |
| TEST | good= 0 | 11.1.7 | Complete(FAULT) | IDLE |
| | good= 1 | 11.1.8 | i^= 1 | WAITB |
| | | 11.1.9 | sample= timer | |
| | | 11.1.10 | ClearNeeds() | |
| | | 11.1.11 | SendSurveyB() | |
| WAITA | peekPassB | 11.1.12 | — | SEND |
| | (timer-sample)>TIMEOUT | 11.1.13 | | |
| WAITB | peekPassB && passID==thisID | 11.1.14 | Complete(GOOD) | IDLE |
| | (timer-sample)>TIMEOUT | 11.1.15 | — | SEND |

**Row 11.1.1:** Receipt of the SurveyRequest event triggers the initial state transition.
**Row 11.1.2:** Send the pass-A survey sequence message.
**Row 11.1.3:** Start the timeout timer, in case the pass-A survey message never returns.

**Row 11.1.4:** A higher precedence accumulation message aborts this survey sequence.
**Row 11.1.5:** A lower precedence accumulation message is tossed, thereby aborting that survey sequence.
**Row 11.1.6:** When the accumulation message circulates and returns, the bandwidths are checked.
A speculative adjustment is performed; the adjustment status is saved in an internal *good* value.

**Row 11.1.7:** A failure status is provided if the sampled rates plus the desired rates exceed the capacity.
**Row 11.1.8:** The change is committed is the sampled rates plus the desired rates remain below capacity.
**Row 11.1.9:** A timeout for the phase-B message is started.
**Row 11.1.10:** The requested rate changes are cleared to zero when they have taken effect.
**Row 11.1.11:** The phaseB survey-sequence message distributes the revised cumulative link rates.

**Row 11.1.12:** A pass-through phaseB survey-sequence message triggers a repeat of the survey sequence, since completion of that higher-precedence survey confirms the failure of this lower-precedence survey.
**Row 11.1.13:** A lengthy waiting-for-phaseB timeout triggers a repeat of the survey sequence.

**Row 11.1.14:** A success status is provided when the phaseB survey-sequence message returns.
**Row 11.1.15:** A lengthy waiting-for-phaseB timeout triggers a repeat of the survey sequence.
Although the revised bandwidths are committed, repeating the survey sequence robustly distributes revised rates in the presence of other concurrent surveys.

# 12. LME

# Annexes

## Annex A: Bibliography (informative)

The following publications are recommended as background material for understanding the objectives behind this standard:

[B1] IEEE Std 1596-1992, Scalable Coherent Interface.[3]

[B2] IEEE Std 1394-1995, High Performance Serial Bus.[4]

[3] ANSI/IEEE publications are available from the Institute of Electrical and Electronics Engineers, Service Center, 445 Hoes Lane, P. O. Box 1331, Piscataway, NJ 08855-1331, USA.

## Annex B: Transmit clock synchronization (normative)

## Annex C: 10G Ethernet PHY (normative)

## Annex D: SONET PHY (normative)

# Annex E: Physical MAC client interface (normative)

## E.1 Interface topologies

Both split and unified MAC implementation models are supported, as illustrated in the left and right sides of Figure E.1 respectively. Assuming 40Gbs data paths, 10Gbs status paths, and a signal-pin capacity of 1Gbs, this implies 250 and 340 signal pins for the split-MAC and unified-MAC implementations respectively.

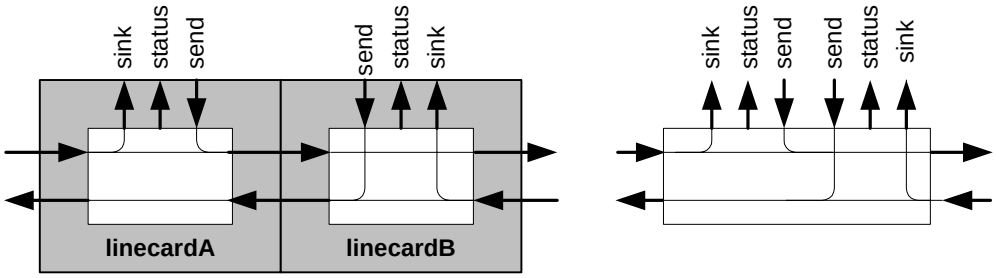



**Figure E.1—MAC partitioning models**

These examples illustrate the need for a high-speed, low-power, low-cost communication pipe between components. This subannex describes LiteLink, an implementation of a 10Gbs byte lane that meets these objectives. LiteLink is a byte-wide pseudo-differential signaling scheme based on a parellel-signal DC-free signal coding, developed by Cypress for connecting high-speed networking components.

Key properties of the link, when compared to existing parallel data-transfer standards include: increased speed, supply-independent voltages, and pseudo differential signaling.

# Annex F: MIB (normative)

# Annex G: 802 LAN bridging (normative)

## G.1 Bridging overview

All types of IEEE 802 Local Area Networks (or LANs) can be interconnected using MAC bridges. Each individual LAN consists of devices attached to the LAN having the same MAC type. The bridged LAN created allows for the inter-connection of stations attached to separate LANs as if they were attached to a single LAN, although they are in fact attached to separate LANs. A transparent MAC bridge operates below the MAC service boundary, and is transparent to protocols operating above this boundary, in the logical link control (LLC) sublayer or network layer (ISO/IEC 7498-1: 1994 1 ). The presence of one or more MAC bridges can lead to differences in the quality of service (QOS) provided by the MAC sublayer; it is only because of such differences that MAC bridge operation may not be fully transparent.

A bridged LAN can provide for

1) The interconnection of stations attached to LANs of different MAC types;
2) An effective increase in the physical extent, the number of permissible attachments, or the total performance of a LAN;
3) Partitioning of the physical LAN for administrative or maintenance reasons.

The MAC bridge standard IEEE Std 802.1D-1990 (subsequently republished as ISO/IEC 10038:1993 [IEEE Std 802.1D, 1993 Edition]) specifies an architecture and protocol for the interconnection of IEEE 802 LANs below the MAC service boundary. Within this context, the RPR network defines a ring topology forming a broadcast media where specific access control mechanisms are employed by the MAC in order to achieve frame delivery and spatial reuse on the ring media. The RPR MAC entity shall provide optional functions within the MAC which optimize bridging of 802 traffic across the ring medium in order to maintain spatial reuse of unicast traffic, as illustrated in bridging reference model of Figure G.1.

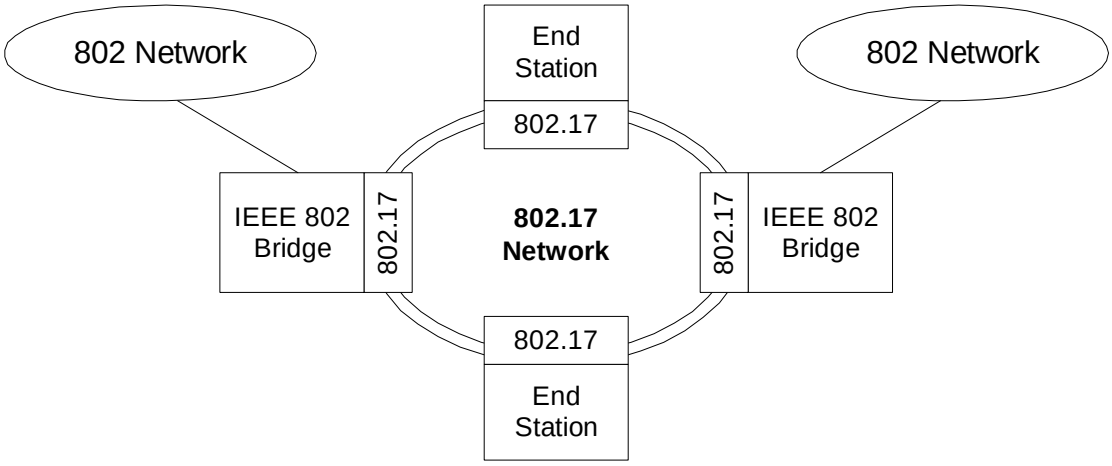



**Figure G.1—Bridging reference model for an 802.17 network**

In order to support transparent bridging of 802 traffic and maintain the spatial reuse property of the ring, the RPR MAC service interface performs the following functions : simple mapping of 802 traffic to the RPR frame format, transport of 802 traffic across the RPR physical medium and delivery of 802 traffic to either the MAC relay or intended 802.17 client at the RPR MAC service interface. The mapping function performed by the RPR MAC service interface shall conform to the interface between a MAC entity and MAC relay, and preserve the filtering services and other requirements for bridged LANs as specified in

ISO/IEC 10038 [IEEE Std 802.1D, 1998 Edition], and ISO/IEC [IEEE Std 802.1q, 1998 Edition]. These services include:

1) Maintaining the bridge architecture;
2) Maintaining the nature of filtering services in bridged LANs;
3) Maintaining the extensions specified by IEEE P802.1Q to allow MAC bridges to support the definition and management of virtual LANS (VLANs);
4) Maintaining the provision of filtering services that support the dynamic definition and establishment of groups in a LAN environment, and the filtering of frames by Bridges such that frames addressed to a given group are forwarded only on those LAN segments that are required in order to reach the members of that group;
5) Supporting the registration protocol that is required in order to provide dynamic multicast filtering services;
6) Supporting management services and operations that are required in order to support administration of dynamic multicast filtering services;
7) Maintaining the provision of expedited traffic capabilities, to support the transmission of time-critical information in a LAN environment;
8) Maintaining the concept of traffic classes and the effect on the operation of the forwarding process of supporting multiple traffic classes in bridges;
9) Maintaining the spanning tree algorithm and protocol;
10) Maintaining the generic attribute registration protocol (GARP);
11) Maintaining the GARP multicast registration protocol (GMRP);

## G.2 Architectural model of a bridge

The RPR MAC conforms to the architectural model of a bridge as defined by IEEE 802.1D. The component LANs are interconnected by means of MAC bridges; each port of a MAC bridge connects to a single LAN. Figure G.2 illustrates the architecture of such a bridge.

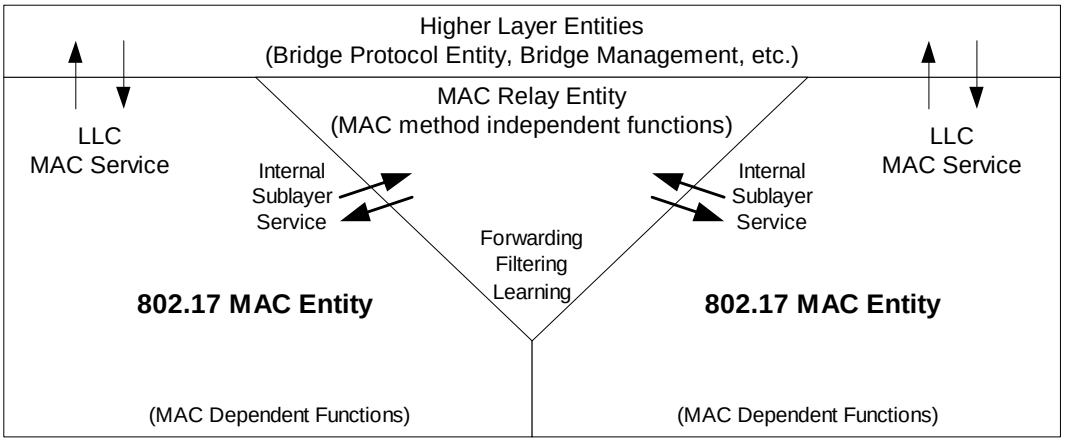



**Figure G.2—Bridge architecture model**

A bridge consists of:

1) A MAC relay entity that interconnects the bridge's ports;
2) At least two ports;
3) Higher layer entities, including at least a bridge protocol entity.

### G.2.2 MAC relay entity

The MAC relay entity handles the MAC method independent functions of relaying frames between bridge ports, filtering frames, and learning filtering information. It uses the internal sublayer service provided by the separate MAC entities for each port. Frames are relayed between ports attached to different LANs.

### G.2.3 Ports

Each bridge port transmits and receives frames to and from the LAN to which it is attached. An individual MAC entity permanently associated with the port provides the internal sublayer service used for frame transmission and reception. The MAC entity handles all the MAC method dependent functions (MAC protocol and procedures) as specified in the relevant standard for that IEEE 802 LAN MAC technology.

### G.2.4 Higher layer entities

The bridge protocol entity handles calculation and configuration of bridged LAN topology.

The bridge protocol entity and other higher layer protocol users, such as bridge management (7.1.3) and GARP application entities including GARP participants (Clause 12), make use of logical link control procedures. These procedures are provided separately for each port, and use the MAC service provided by the individual MAC Entities.

## G.3 RPR MAC bridging reference model

The MAC reference model is illustrated in Figure G.3. The RPR MAC consists of a MAC entity which provides the media access control functions to the pair of ringlets (ringlet0/ringlet1) comprising the RPR ring. The pass-through function within the RPR MAC entity processes frames which are intended for other RPR stations on the ring. The pass-through function takes frames from the receive side of the ringlet and presents them to the transmit side of the ringlet. Traffic received from either ringlet_0 or ringlet_1, intended for this RPR station, is passed up to the RPR internal sublayer service which in turn passes ingress traffic to the 802 MAC relay entity. The 802 MAC relay performs forwarding, filtering, learning functions between this RPR interface and other 802 type interfaces within the bridge. Traffic from the 802 MAC Relay destined to the ring is presented to the RPR internal sublayer service which in turn determines whether to transmit the traffic on either ringlet_0, ringlet_1, or in some cases both. The RPR internal sublayer service performs the mapping between client MAC addresses provided by the 802 MAC relay, and RPR station addresses in the RPR frame header.

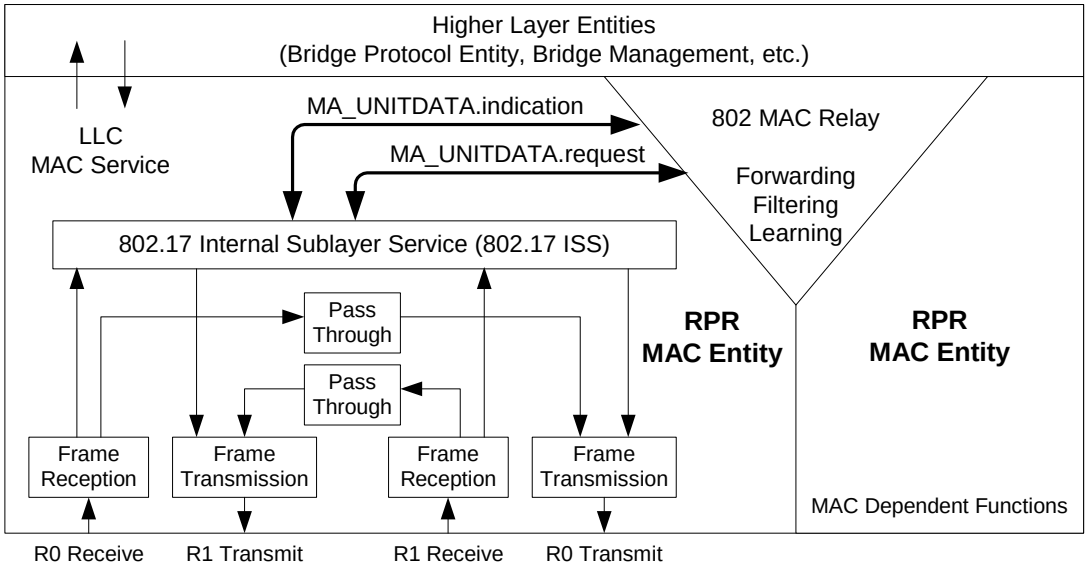



**Figure G.3—RPR MAC reference model**

The RPR MAC entity appears as a single interface to the 802 MAC relay. This means the RPR ring media and the collection of stations which attach to the ring appears to the 802 MAC relay as a single loop free broadcast media. The RPR MAC ensures that a frame is delivered to the intended RPR station (in the case of a known unicast) or is delivered to all stations (in the cast of a multicast, broadcast or unknown frame). The RPR MAC also ensures that only a single copy of a frame is delivered to the RPR MAC internal sublayer service within each station. RPR MAC procedures ensure that duplicate copies of a frame are not transferred to the RPR internal sublayer service (ISS). This includes scenarios where the ring is in a normal operating configuration, or frames are being wrapped or steered during a ring failure. Since the RPR ring behaves as a loop free broadcast media, spanning tree protocol is not required for networks where a collection of 802 bridges attach to a single RPR ring and do not create a loop via another network connection. Spanning tree protocol can be enabled over an RPR ring for the purpose of maintaining a loop free bridged network topology when 802 bridges attach to an RPR ring and are multiply interconnected via another RPR ring or 802 type network. The RPR MAC entity provides LLC services to support the bridge protocol entity and other higher layer protocol users.

The RPR MAC entity appends RPR source/destination station identifier's (DSID, SSID) to the RPR frame for the purpose of performing destination and source stripping of frames from the ring. Destination stripping allows a frame to be stripped from the ring when arriving at the intended destination without having to traverse the entire ring. Subsequent spans, following the span where the packet was stripped, can be reused by other stations for transmitting new traffic onto the ring thereby providing spatial reuse of the ring. Source stripping ensures that a frame which traverses the entire ring is not read a second time by stations, thus maintaining a loop free behavior. RPR destination station ID is appended to the RPR frame via a mapping function as part of the 802.17 MAC Entity support of the ISS. This mapping function maps the 802 *destinationMacAddress* to the RPR *destinationStationID* in the RPR frame header. The RPR ISS also appends the RPR *sourceStationID* in the RPR frame header with the transmitter's source station identifier.

## G.4 Model of operation

The model of operation is simply a basis for describing the functionality of the MAC bridge. It is in no way intended to constrain real implementations of a MAC bridge; these may adopt any internal model of operation compatible with the externally visible behavior that this standard specifies. Conformance of equipment to this standard is purely in respect of observable protocol.

### G.4.1 802.17 Support of the Internal Sublayer Service

The following figure illustrates the mapping of the MA-UNITDATA.request / MA-UNITDATA.indication primitives to the 802.17 frame format.

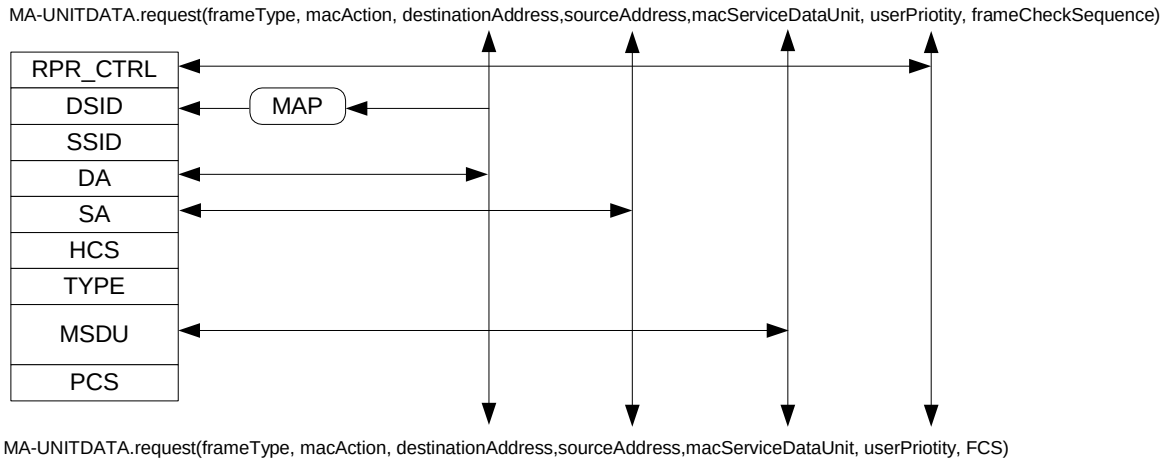



**Figure G.4—Mapping of MA-UNITDATA primitives to 802.17 frame format**

On receipt of an M_UNITDATA.request primitive, the local MAC Entity performs Transmit Data Encapsulation, assembling a frame using the parameters supplied as specified below. On receipt of a MAC frame by Receive Media Access Management, the MAC frame is passed to Receive Data Decapsulation, which validates the FCS and disassembles the frame, as specified below, into the parameters that are supplied with an M_UNITDATA.indication primitive.

The ***frameType*** parameter takes only the value user_data_frame and is not explicitly encoded in MAC frames. The ***macAction*** parameter takes only the value request_with_no_response and is not explicitly encoded in MAC frames.

The ***destinationAddress*** parameter is encoded in the *destinationMacAddress* field of the MAC frame (see **Error! Reference source not found.**). The ***sourceAddress*** parameter is encoded in the *sourceMacAddress* field of the MAC frame (see **Error! Reference source not found.**).

The number of octets in the ***macServiceDataUnit*** parameter is encoded in the length field of the MAC frame (IEEE Std 802.17 ?.?.?), and the octets of data are encoded in the data field (see **Error! Reference source not found.**).

The ***userPriority*** parameter provided in a data request primitive is encoded in corresponding priority bits of the RPR control header of the transmitted frame. The *userPriority* parameter provided in a data indication primitive takes the value of the corresponding priority bits of the RPR control header of the received frame.

The headerCheckSequence (***HCS***) of the MAC frame is computed as a function of the *destinationStationID*, *sourceStationID*, *destinationMacAddress*, *sourceMacAddress*, and RPR header control fields of the transmitted frame.

The *payloadCheckSequence* (***PCS***) of the MAC frame is re-computed as a function of the MacServiceDataUnit (see xx).

The ***frameCheckSequence*** parameter in the MA_UNITDATA.request is defined as an unspecified value, signaling the underlying 802.17 MAC to regenerate the frame FCS. The FCS in the

MA_UNITDATA.indication is set to either valid or invalid based on whether the FCS of the receive frame is valid/invalid.

NOTE —IEEE Std 802.3, 1998 Edition, describes the use of either a Length or an Ethernet protocol type in its frame format; however, the text of this subclause has yet to be revised to describe the use of Ethernet protocol types.

### G.4.2 Frame transmission

The individual MAC entity associated with each bridge port transmits frames submitted to it by the MAC relay entity.

Relayed frames for transmission by the forwarding process are submitted to the RPR ISS. The M_UNITDATA.request primitive associated with such frames conveys the values of the client MAC source and destination address fields received in the corresponding M_UNITDATA.indication primitive.

LLC protocol data units (PDUs) are submitted by LLC as a user of the MAC service provided by the bridge port. Frames transmitted to convey such PDUs carry the individual client MAC address of the port in the source address field. All LLC PDUs are submitted to the RPR ISS. The RPR ISS in turn performs the same client MAC destination address to RPR *destinationStationID* and *destinationAddress* mapping as described for frames submitted to the RPR ISS from the MAC relay entity.

Each frame is transmitted subject to the following procedure associated with the RPR MAC technology. The values of the frameType and macAction parameters of the corresponding M_UNIT-DATA. request primitive shall be user_data_frame and request_with_no_response, respectively (6.5).

The client MAC destination address is used by the RPR ISS mapping function to determine the RPR *destinationStationID* (DSID) and *destinationAddress* used in the RPR frame header of the transmitted frame.

1) If the client MAC destination address is found in the RPR ISS mapping table, the associated RPR *destinationStationID* and *ringletID* are extracted from the table; these provide for destination stripping of the unicast frame and shortest-path routing. This station's *sourceStationID* is included in the header. The RPR *destinationMacAddress* and *sourceMacAddress* fields are copies of the client MAC *destinationAddress* and *sourceAddress* fields respectively.
2) If the client MAC destination address is not found in the mapping table, two frames are created. Within these frames, this station's *sourceStationID* is included in the header. The RPR *destinationMacAddress* and *sourceMacAddress* fields are copies of the client MAC *destinationAddress* and *sourceAddress* fields respectively. Other parameters are different within each of these frames, as follows:
    a) The *destinationStationID* is set to identify bridge0 and ringletID is set to 0 (bridge0 may be any bridge station located on ring0).
    b) The *destinationStationID* is set to identify bridge1 and ringletID is set to 1 (bridge1 shall be the last bridge before bridge0).

    There are several acceptable degenerate cases where only one frame is sent, as follows:
    The frame is sent on ring0 and *destinationStationID* equals *sourceStationID*.
    The frame is sent on ring1 and *destinationStationID* equals *sourceStationID*.
    The frame is sent on ring0 and *destinationStationID* identifies the last reachable bridge.
    The frame is sent on ring1 and *destinationStationID* identifies the last reachable bridge.
3) All broadcast and multicast type frames set the *flood* bit in the RPR header. The frame contents are otherwise the same as specified in (2).

The RPR *sourceStationID* (SSID) in the transmitted frame shall always be set to the transmitting station's source station ID. This parameter is used to invoke source stripping at the receiver, which allows the

receiver to learn the association of *sourceStationID* with client MAC *sourceAddress* in received frames. This knowledge should then be used to efficiently direct the expected unicast response frames to the client.

Frames transmitted following a request by the LLC user of the MAC service provided by the bridge port shall also be submitted to the MAC relay entity.

NOTE —Maintaining ordering sometimes mandates flushing of in-flight packets during protection events; see Clause 0 for details.

### G.4.3 Frame reception

The individual MAC entity associated with each bridge port examines all frames received on the RPR ringlet to which it is attached. The RPR *destinationStationID* and *sourceStationID* affect where the packet is stripped; the *destinationMacAddress* and *sourceMacAddress* affect how packets are processed; see 6.1 for details.

All error-free received frames are passed to the RPR ISS give rise to M_UNITDATA indication primitives which shall be handled as follows:

NOTE —A frame that is in error, as defined by the relevant MAC specification, is discarded by the MAC entity without giving rise to any M_UNITDATA indication; see 6.4.

The receiving station's receive procedure updates its mapping table with the client MAC source address, its associated VID (if available), and the RPR *sourceStationID* address from the RPR frame header. The RPR ISS provides the M_UNITDATA indication primitive, *frameType* and *macAction* parameter values of user_data_frame and request_with_no_response respectively to the learning and forwarding processes in the MAC relay entity.

Frames with other values of *frameType* and *macAction* parameters (e.g., request_with_response and response frames), shall not be submitted to the forwarding process. They may be submitted to the learning process.

Frames with a *frameType* of user_data_frame and addressed to the bridge port as an end station shall be submitted to LLC. Such frames carry either the individual MAC address of the port or a group address associated with the port (7.12) in the destination address field. Frames submitted to LLC can also be submitted to the learning and forwarding processes, as specified above.

Frames addressed to a bridge port as an end station, and relayed to that bridge port from other bridge ports in the same bridge by the forwarding process, shall also be submitted to LLC.

No other frames shall be submitted to LLC.

# Annex H: CRC calculations (informative)

## H.1 Cyclic redundancy check (CRC)

### H.1.1 Algorithmic definition

There is a 32-bit check symbol at the end of the packet header and payload. For good error coverage, a cyclic redundancy code (CRC) is used. The CRC efficiently detects errors but does not correct errors. Error recovery is performed at a higher level.

The CRCs that is used is the same CRC used by IEEE 802 LANs and FDDI. The CRC uses the generator polynomial of equation xx. Which is performed on the most significant bits first:

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$$

### H.1.2 Serial CRC calculation

The serial implementation of the CRC-32 polynomial, as applied to the most- through least-significant bits, is specified by the C-code calculations of Table H.1 and the hardware implementation illustrated in figurexx.

**Table H.1—Serial CRC-32 computations**

```
// c00-through-c31 are the most- through least-significant bits of check.
// d00 is the input value, sum is an intermediate value.
Sum= c00^d00;
c00= c01;         c01= c02;         c02= c03;         c03= c04;
c04= c05;         c05= c06^sum;     c06= c07;         c07= c08;
c08= c09^sum;     c09= c10^sum;     c10= c11;         c11= c12;
c12= c13;         c13= c12;         c14= c15;         c15= c16^sum;
c16= c17;         c17= c18;         c18= c19;         c19= c20^sum;
c20= c21^sum;     c21= c22^sum;     c22= c23;         d23= c24^sum;
c24= c25^sum;     c25= c26;         c26= c27^sum;     c27= c28^sum;
c28= c29;         c29= c30^sum;     c30= c31^sum;     c31= sum;
```

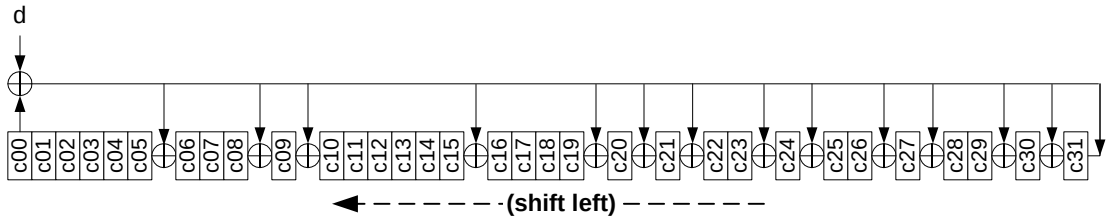



**Figure H.1—Serial crc32 reference model**

The CRC calculation has several somtimes subtle characteristics, in addition to the basic polynomial-based CRC calculations, that could produce non-standard results if implemented differently. For the benefit of the casual reader, and to reduce the possibility of creating non-standard implementations, these characteristics are summarized below:

1) Startup. The CRC calculations start with an all-ones crcSum value.
2) Complete. The computed CRC value is complemented before being appended to the packet.

## H.2 Arranged-CRC calculations

Several interconnects send bits in a most- through least-significant ordering. For such interconnects, the CRC calculation is based on this ordering assumption and described in the remainder of this subclause.

NOTE —If the CRC becomes a MAC level definition, this would be the proposed algorithm. If the CRC becomes a PHY level definition, then this would be one of the physical layer definitions.

### H.2.1 Arranged ExorSum calculations

The generation and checking of 32-bit CRC values, optimized for bit-sequential transmission, is illustrated in Figure H.2.

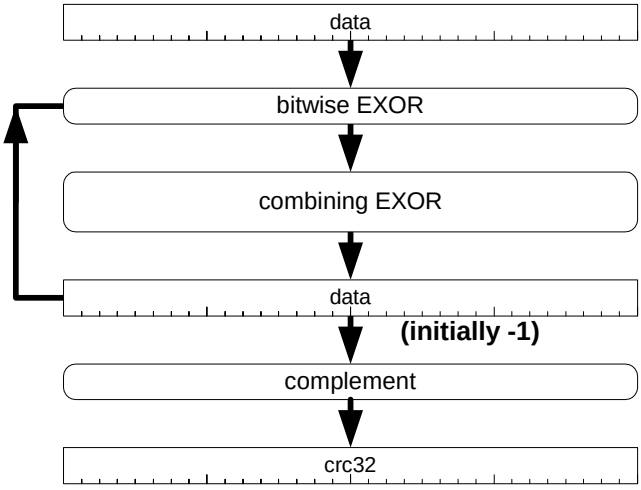



**Figure H.2—Arranged ExorSum calculations**

The CRC-generation code of B.1.2 can be called to generate CRC-computation tables, using the C program documented in Annex E. This program supports the creation of tables for performing parallel CRC checks, where 1, 2, 4, 8, 16, or 32 data bits are processed in parallel. Computer generation of the CRC-table text, rather than their values, minimized the possibility of introducing errors in the documentation process.

### H.2.2 Arranged ExorSum32 equations

Although the CRC is specified as a bit-serial computation, the CRC value can be computed in parallel. This is important for RPR, because CRCs have to be checked and regenerated at full RPR speed. Parallelizing the serial specification, to process 32 data bits in parallel, generates the equations shown in Table H.2.

**Table H.2—Arranged ExorSumCrc32 equations**

```
// C00-through-c31 are the most- through least-significant bits of check.
// d00-through-d31 are the most- through least-significant bits of input.
// "a".."t""A".." " are intermediate bit values.
a= c00^d00;   b= c01^d01;   c= c02^d02;   d= c03^d03;
e= c04^d04;   f= c05^d05;   g= c06^d06;   h= c07^d07;
j= c08^d08;   k= c09^d09;   m= c10^d10;   n= c11^d11;
p= c12^d12;   r= c13^d13;   s= c14^d14;   t= c15^d15;
A= c16^d16;   B= c17^d17;   C= c18^d18;   D= c19^d19;
E= c20^d20;   F= c21^d21;   G= c22^d22;   H= c23^d23;
J= c24^d24;   K= c25^d25;   M= c26^d26;   N= c27^d27;
P= c28^d28;   R= c29^d29;   S= c30^d30;   T= c31^d31;
//                      1                   2                   3
//   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
//   a b c d e f g h j k m n p r s t A B C D E F G H J K M N P R S T
c00= a^b^c^d^e^  g^h^j^             A^      E^  G^H^    M          ;
c01=   b^c^d^e^f^  h^j^k^             B^      F^  H^J^    N        ;
c02= a^  c^d^e^f^g^  j^k^m^             C^      G^  J^K^    P      ;
c03=   b^  d^e^f^g^h^  k^m^n^             D^      H^  K^M^    R    ;
c04=     c^  e^f^g^h^j^  m^n^p^             E^      J^  M^N^    S  ;
c05= a^    d^  f^g^h^j^k^  n^p^r^             F^      K^  N^P^    T;
c06= a^  c^d^          k^m^  p^r^s^  A^      E^    H^        P^R    ;
c07=   b^  d^e^          m^n^  r^s^t^  B^      F^    J^        R^S  ;
c08= a^  c^  e^f^          n^p^  s^t^A^  C^      G^    K^        S^T;
c09= a^  c^  e^f^  h^j^      p^r^  t^  B^  D^E^  G^                T;
c10= a^  c^  e^f^  h^  k^      r^s^      C^    F^G^      M          ;
c11=   b^  d^  f^g^  j^  m^      s^t^      D^    G^H^      N        ;
c12=     c^  e^  g^h^  k^  n^      t^A^      E^    H^J^      P      ;
c13= a^    d^  f^  h^j^  m^  p^      A^B^      F^    J^K^      R    ;
c14= a^b^    e^  g^  j^k^  n^  r^      B^C^      G^    K^M^      S  ;
c15=   b^c^    f^  h^  k^m^  p^  s^      C^D^      H^    M^N^      T;
c16=   b^    e^    h^    m^n^  r^  t^A^    D^    G^H^J^  M^N^P      ;
c17=     c^    f^    j^    n^p^  s^  A^B^    E^    H^J^K^  N^P^R    ;
c18= a^    d^    g^    k^    p^r^  t^  B^C^    F^    J^K^M^  P^R^S  ;
c19= a^b^    e^    h^    m^    r^s^  A^  C^D^    G^    K^M^N^  R^S^T;
c20= a^    d^e^f^g^h^      n^    s^t^A^B^  D^    G^        N^P^  S^T;
c21= a^  c^d^  f^            p^    t^  B^C^      G^      M^  P^R^  T;
c22=     c^        h^j^        r^        C^D^E^  G^      M^N^  R^S  ;
c23= a^    d^        j^k^        s^        D^E^F^  H^      N^P^  S^T;
c24=     c^d^    g^h^j^k^m^        t^A^        F^  H^J^  M^  P^R^  T;
c25=   b^c^      g^    k^m^n^          B^    E^    H^J^K^M^N^  R^S  ;
c26=     c^d^      h^    m^n^p^          C^    F^    J^K^M^N^P^  S^T;
c27= a^b^c^      g^h^      n^p^r^    A^    D^E^    H^  K^  N^P^R^  T;
c28= a^      e^  g^          p^r^s^  A^B^      F^G^H^J^      P^R^S  ;
c29= a^b^      f^  h^          r^s^t^  B^C^      G^H^J^K^      R^S^T;
c30=       d^e^    h^            s^t^    C^D^E^  G^  J^K^        S^T;
c31= a^b^c^d^  f^g^h^              t^      D^  F^G^    K^          T;
```

### H.2.3 Arranged ExorSumCrc16 equations

Although the CRC computation is expected to be performed in a 32-bit parallel fashion, less logic is required if 16 data bits can be processed at twice the 32-bit-symbol clock rate. Parallelizing the serial specification, to process 16 data bits in parallel, generates the equations shown in Table H.3.

**Table H.3—Arranged ExorSumCrc16 equations**

```
// C00-through-c31 are the most- through least-significant bits of check.
// d00-through-d15 are the most- through least-significant bits of input.
// "a".."t""A".." " are intermediate bit values.
a= c00^d00;   b= c01^d01;   c= c02^d02;   d= c03^d03;
e= c04^d04;   f= c05^d05;   g= c06^d06;   h= c07^d07;
j= c08^d08;   k= c09^d09;   m= c10^d10;   n= c11^d11;
p= c12^d12;   r= c13^d13;   s= c14^d14;   t= c15^d15;
A= c16;       B= c17;       C= c18;       D= c19;
E= c20;       F= c21;       G= c22;       H= c23;
J= c24;       K= c25;       M= c26;       N= c27;
P= c28;       R= c29;       S= c30;       T= c31;
//                       1                   2                   3
//   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
//   a b c d e f g h j k m n p r s t A B C D E F G H J K M N P R S T
c00= a^      e^  g^h^    m^          A                               ;
c01=   b^      f^  h^j^    n^          B                             ;
c02=     c^      g^  j^k^    p^          C                           ;
c03=       d^      h^  k^m^    r^          D                         ;
c04=         e^      j^  m^n^    s^          E                       ;
c05=           f^      k^  n^p^    t^          F                     ;
c06= a^      e^    h^        p^r^              G                   ;
c07=   b^      f^    j^        r^s^              H                 ;
c08= a^  c^      g^    k^        s^t^              J               ;
c09=   b^  d^e^  g^                t^              K             ;
c10=     c^    f^g^      m^                        M           ;
c11=       d^    g^h^      n^                        N         ;
c12= a^      e^    h^j^      p^                        P       ;
c13= a^b^      f^    j^k^      r^                        R     ;
c14=   b^c^      g^    k^m^      s^                        S   ;
c15=     c^d^      h^    m^n^      t^                        T;
c16= a^    d^    g^h^j^  m^n^p                               ;
c17= a^b^    e^    h^j^k^  n^p^r                             ;
c18=   b^c^    f^    j^k^m^  p^r^s                           ;
c19= a^  c^d^    g^    k^m^n^  r^s^t                         ;
c20= a^b^  d^    g^        n^p^  s^t                         ;
c21=   b^c^      g^      m^  p^r^  t                         ;
c22=     c^d^e^  g^      m^n^  r^s                           ;
c23=       d^e^f^  h^      n^p^  s^t                         ;
c24= a^        f^  h^j^  m^  p^r^  t                         ;
c25=   b^    e^    h^j^k^m^n^  r^s                           ;
c26=     c^    f^    j^k^m^n^p^  s^t                         ;
c27= a^    d^e^    h^  k^  n^p^r^  t                         ;
c28= a^b^      f^g^h^j^      p^r^s                           ;
c29=   b^c^      g^h^j^k^      r^s^t                         ;
c30=     c^d^e^  g^  j^k^        s^t                         ;
c31=       d^  f^g^    k^          t                         ;
```

### H.2.4 Arranged ExorSumCrc8 equations

The CRC computation logic can be further reduced if 8 data bits can be processed at four times the 32-bit-symbol clock rate. Parallelizing the serial specification, to process 8 data bits in parallel, generates the equations shown in Table H.4.

**Table H.4—Arranged ExorSumCrc8 equations**

```
// c00-through-c31 are the most- through least-significant bits of check.
// d00-through-d07 are the most- through least-significant bits of input.
// "a".."t""A.." " are intermediate bit values.
a= c00^d00;   b= c01^d01;   c= c02^d02;   d= c03^d03;
e= c04^d04;   f= c05^d05;   g= c06^d06;   h= c07^d07;
j= c08;       k= c09;       m= c10;       n= c11;
p= c12;       r= c13;       s= c14;       t= c15;
A= c16;       B= c17;       C= c18;       D= c19;
E= c20;       F= c21;       G= c22;       H= c23;
J= c24;       K= c25;       M= c26;       N= c27;
P= c28;       R= c29;       S= c30;       T= c31;
//   00                  10                  20                  30
//   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
//   a b c d e f g h j k m n p r s t A B C D E F G H J K M N P R S T
c00=     c^          j                                             ;
c01= a^    d^          k                                           ;
c02= a^b^    e^          m                                         ;
c03=   b^c^    f^          n                                       ;
c04= a^  c^d^    g^          p                                     ;
c05=   b^  d^e^    h^          r                                   ;
c06=         e^f^                s                               ;
c07= a^        f^g^                t                             ;
c08=   b^        g^h^                A                           ;
c09=               h^                  B                         ;
c10=     c^                              C                       ;
c11=       d^                              D                     ;
c12= a^      e^                              E                   ;
c13= a^b^      f^                              F                 ;
c14=   b^c^      g^                              G               ;
c15=     c^d^      h^                              H             ;
c16= a^  c^d^e^                                    J            ;
c17= a^b^  d^e^f^                                    K          ;
c18= a^b^c^  e^f^g^                                    M        ;
c19=   b^c^d^  f^g^h^                                    N       ;
c20=       d^e^  g^h^                                      P     ;
c21=     c^  e^f^  h^                                        R   ;
c22=     c^d^  f^g^                                            S ;
c23=       d^e^  g^h^                                            T;
c24= a^  c^  e^f^  h                                             ;
c25= a^b^c^d^  f^g                                               ;
c26= a^b^c^d^e^  g^h                                             ;
c27=   b^  d^e^f^  h                                             ;
c28= a^      e^f^g                                               ;
c29= a^b^      f^g^h                                             ;
c30= a^b^        g^h                                             ;
c31=   b^          h                                             ;
```

## H.3 Exchanged CRC calculations

Several interconnects send bits in a least- through most-significant bit ordering. For such interconnects, the CRC calculation is based on this ordering assumption and described in the remainder of this subclause.

NOTE —If the CRC becomes a MAC level definition, this proposed algorithm would be abandoned. If the CRC becomes a PHY level definition, then this would be one of the two physical layer definitions.

### H.3.1 Exchanged ExorSum calculations

The generation and checking of 32-bit CRC values, optimized for bit-reversed transmission, is illustrated in Figure H.3. The complexity of the implementation is smaller than at first seems, since the bitSwap operations involve not circuitry and can be eliminated by encorporating their functionality within the combining-EXOR circuitry.

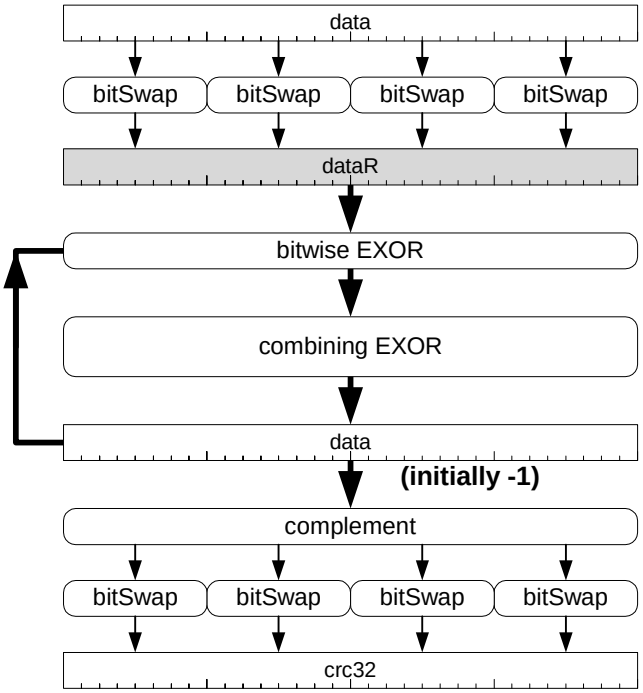



**Figure H.3—exchangedExorSum calculations**

The CRC-generation code of B.1.2 can be called to generate CRC-computation tables, using the C program documented in Annex E. This program supports the creation of tables for performing parallel CRC checks, where 1, 2, 4, 8, 16, or 32 data bits are processed in parallel. Computer generation of the CRC-table text, rather than their values, minimized the possibility of introducing errors in the documentation process.

### H.3.2 Exchanged ExorSumCrc32 equations

Although the CRC is specified as a bit-serial computation, the CRC value can be computed in parallel. This is important for RPR, because CRCs have to be checked and regenerated at full RPR speed. Combining the bit-within-byte reversals and parallelizing the serial specification, to process 32 data bits in parallel, generates the equations shown in table B.1.

**Table H.5—Exchanged ExorSumCrc32 equations**

```
// C00-through-c31 are the most- through least-significant bits of check.
// d00-through-d31 are the most- through least-significant bits of input.
// "a".."t""A".." " are intermediate bit values.
a= c00^d00;   b= c01^d01;   c= c02^d02;   d= c03^d03;
e= c04^d04;   f= c05^d05;   g= c06^d06;   h= c07^d07;
j= c08^d08;   k= c09^d09;   m= c10^d10;   n= c11^d11;
p= c12^d12;   r= c13^d13;   s= c14^d14;   t= c15^d15;
A= c16^d16;   B= c17^d17;   C= c18^d18;   D= c19^d19;
E= c20^d20;   F= c21^d21;   G= c22^d22;   H= c23^d23;
J= c24^d24;   K= c25^d25;   M= c26^d26;   N= c27^d27;
P= c28^d28;   R= c29^d29;   S= c30^d30;   T= c31^d31;
//   00                  10                  20                  30
//   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
//   a b c d e f g h j k m n p r s t A B C D E F G H J K M N P R S T
c00=       d^e^  g^  j^k^m^  p^r^        C^      G^    K^M^        T;
c01=         e^f^  h^  k^m^n^  r^s^  A^    D^      H^    M^N        ;
c02= a^b^c^  e^    h^    m^n^p^  s^t^    C^          J^    N^P^  S  ;
c03= a^b^c^d^  f^          n^p^r^  t^      D^          K^    P^R^  T;
c04= a^b^c^d^e^  g^          p^r^s^  A^      E^          M^    R^S  ;
c05=   b^c^d^e^f^  h^          r^s^t^  B^      F^          N^    S^T;
c06= a^  c^d^e^f^g^              s^t^A^  C^      G^          P^    T;
c07= a^b^  d^e^f^g^h^              t^A^B^  D^      H^          R    ;
c08= a^  c^    f^g^    k^  n^  r^s^  A^      E^F^    J^      P^R    ;
c09=   b^  d^    g^h^    m^  p^  s^t^  B^      F^G^    K^      R^S  ;
c10= a^  c^  e^    h^      n^  r^  t^    C^      G^H^    M^      S^T;
c11= a^b^  d^  f^    j^      p^  s^  A^    D^      H^      N^      T;
c12=   b^c^  e^  g^  j^k^      r^  t^A^B^    E^              P      ;
c13= a^  c^d^  f^  h^  k^m^      s^    B^C^    F^              R    ;
c14= a^  c^d^  f^  h^j^  m^n^      t^  B^  D^E^  G^  J            ;
c15=     c^d^  f^  h^j^k^  n^p^        B^      F^  H^J^K^        S  ;
c16=         e^    h^  k^        s^t^A^  C^D^E^      J^K^  N^P      ;
c17= a^        f^        m^          t^  B^  D^E^F^      K^M^  P^R    ;
c18=     c^  e^f^  h^j^    n^          B^      F^G^  J^  M^N^  R    ;
c19= a^b^c^d^e^    h^j^k^    p^        B^    E^  G^H^J^K^  N^P      ;
c20= a^    d^    g^h^  k^m^    r^      B^    E^F^  H^J^K^M^  P^R^S  ;
c21=   b^    e^    h^j^  m^n^    s^      C^    F^G^    K^M^N^  R^S^T;
c22=     c^    f^      k^  n^p^    t^A^    D^    G^H^    M^N^P^  S^T;
c23= a^    d^    g^  j^  m^  p^r^    A^B^    E^    H^      N^P^R^  T;
c24= a^b^c^  e^f^g^h^j^              B^C^  E^      J^        S  ;
c25= a^    d^e^      j^k^              B^  D^E^F^    J^K^        S^T;
c26= a^  c^      g^h^j^k^m^          A^B^      F^G^  J^K^M^      S^T;
c27=   b^  d^      h^  k^m^n^        A^B^C^      G^H^  K^M^N^      T;
c28= a^b^      f^g^h^    m^n^p^      A^    D^E^    H^J^  M^N^P^  S  ;
c29= a^      e^f^          n^p^r^        C^    F^    J^K^  N^P^R^S^T;
c30=   b^      f^g^          p^r^s^  A^    D^    G^    K^M^  P^R^S^T;
c31= a^b^    e^f^    j^        r^s^t^A^  C^        H^J^  M^N^  R^  T;
```

### H.3.3 Exchanged ExorSumCrc16 equations

Although the CRC computation is expected to be performed in a 32-bit parallel fashion, less logic is required if 16 data bits can be processed at twice the 32-bit-symbol clock rate. Combining the bit-within-byte reversals and parallelizing the serial specification, to process 16 data bits in parallel, generates the equations shown in table B.2.

**Table H.6—Exchanged ExorSumCrc16 equations**

```
// C00-through-c31 are the most- through least-significant bits of check.
// d00-through-d15 are the most- through least-significant bits of input.
// "a".."t""A".." " are intermediate bit values.
a= c00^d00;   b= c01^d01;   c= c02^d02;   d= c03^d03;
e= c04^d04;   f= c05^d05;   g= c06^d06;   h= c07^d07;
j= c08^d08;   k= c09^d09;   m= c10^d10;   n= c11^d11;
p= c12^d12;   r= c13^d13;   s= c14^d14;   t= c15^d15;
A= c16;       B= c17;       C= c18;       D= c19;
E= c20;       F= c21;       G= c22;       H= c23;
J= c24;       K= c25;       M= c26;       N= c27;
P= c28;       R= c29;       S= c30;       T= c31;
//   00                  10                  20                  30
//   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
//   a b c d e f g h j k m n p r s t A B C D E F G H J K M N P R S T
c00=     c^      g^    k^m^        t^A                               ;
c01= a^    d^      h^    m^n^          B                             ;
c02=     c^          j^    n^p^  s^      C                           ;
c03=       d^          k^    p^r^  t^      D                         ;
c04= a^      e^          m^    r^s^          E                       ;
c05=   b^      f^          n^    s^t^          F                     ;
c06= a^  c^      g^          p^    t^            G                   ;
c07= a^b^  d^      h^          r^                  H                 ;
c08= a^      e^f^    j^      p^r^                    J               ;
c09=   b^      f^g^    k^      r^s^                    K             ;
c10=     c^      g^h^    m^      s^t^                    M           ;
c11= a^    d^      h^      n^      t^                      N         ;
c12= a^b^    e^              p^                            P       ;
c13=   b^c^    f^              r^                            R     ;
c14=   b^  d^e^  g^  j^                                      S   ;
c15=   b^      f^  h^j^k^        s^                            T;
c16= a^  c^d^e^      j^k^  n^p                                   ;
c17=   b^  d^e^f^      k^m^  p^r                                 ;
c18=   b^      f^g^  j^  m^n^  r                                 ;
c19=   b^    e^  g^h^j^k^  n^p                                   ;
c20=   b^    e^f^  h^j^k^m^  p^r^s                               ;
c21=     c^    f^g^    k^m^n^  r^s^t                             ;
c22= a^    d^    g^h^    m^n^p^  s^t                             ;
c23= a^b^    e^    h^      n^p^r^  t                             ;
c24=   b^c^  e^      j^          s                               ;
c25=   b^  d^e^f^    j^k^        s^t                             ;
c26= a^b^      f^g^  j^k^m^      s^t                             ;
c27= a^b^c^      g^h^  k^m^n^      t                             ;
c28= a^    d^e^    h^j^  m^n^p^  s                               ;
c29=     c^    f^    j^k^  n^p^r^s^t                             ;
c30= a^    d^    g^    k^m^  p^r^s^t                             ;
c31= a^  c^        h^j^  m^n^  r^  t                             ;
```

### H.3.4 Exchanged ExorSumCrc8 equations

The CRC computation logic can be further reduced if 8 data bits can be processed at four times the 32-bit-symbol clock rate. Combining the bit-within-byte reversals and parallelizing the serial specification, to process 8 data bits in parallel, generates the equations shown in table B.3.

**Table H.7—Exchanged ExorSumCrc8 equations**

```
// c00-through-c31 are the most- through least-significant bits of check.
// d00-through-d07 are the most- through least-significant bits of input.
// "a".."t""A".." " are intermediate bit values.
a= c00^d00;   b= c01^d01;   c= c02^d02;   d= c03^d03;
e= c04^d04;   f= c05^d05;   g= c06^d06;   h= c07^d07;
j= c08;       k= c09;       m= c10;       n= c11;
p= c12;       r= c13;       s= c14;       t= c15;
A= c16;       B= c17;       C= c18;       D= c19;
E= c20;       F= c21;       G= c22;       H= c23;
J= c24;       K= c25;       M= c26;       N= c27;
P= c28;       R= c29;       S= c30;       T= c31;
//   00                  10                  20                  30
//   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
//   a b c d e f g h j k m n p r s t A B C D E F G H J K M N P R S T
c00=   b^c^        h^j                                             ;
c01=     c^d^          k                                           ;
c02= a^    d^e^  g^      m                                         ;
c03=   b^    e^f^  h^      n                                       ;
c04=     c^    f^g^          p                                     ;
c05=       d^    g^h^          r                                   ;
c06=         e^    h^            s                                 ;
c07=           f^                  t                               ;
c08= a^      e^f^                    A                             ;
c09=   b^      f^g^                    B                           ;
c10=     c^      g^h^                    C                         ;
c11=       d^      h^                      D                       ;
c12=         e^                              E                     ;
c13=           f^                              F                   ;
c14= a^                                          G                 ;
c15= a^b^        g^                                H               ;
c16= a^b^  d^e^                                      J             ;
c17=   b^c^  e^f^                                      K           ;
c18= a^  c^d^  f^                                        M         ;
c19= a^b^  d^e^                                            N       ;
c20= a^b^c^  e^f^g^                                          P     ;
c21=   b^c^d^  f^g^h^                                          R   ;
c22=     c^d^e^  g^h^                                            S  ;
c23=       d^e^f^  h^                                              T;
c24= a^          g                                                 ;
c25= a^b^        g^h                                               ;
c26= a^b^c^      g^h                                               ;
c27=   b^c^d^      h                                               ;
c28= a^  c^d^e^  g                                                 ;
c29= a^b^  d^e^f^g^h                                               ;
c30=   b^c^  e^f^g^h                                               ;
c31= a^  c^d^  f^  h                                               ;
```

# Annex I: Time-of-day distribution (normative)

## I.1 Time-of-day synchronization

Time-of-day synchronization involves the tight synchronization of timers maintained on clock-master and clock-slave stations. The intent is to provide uniform "stratum" clocks, to enable synchronization of source and destination devices, to avoid data slips or gaps during the distribution and/or presentation of real-time information, such as telephony traffic.

Some physical layers, such as SONET provide stratum clock services. This subannex describes how these services may be provided at higher layers within other less-supportive physical layers.

### I.1.1 Time-of-day calibration

With bidirectional cables, the clockSync transmissions can account for the constant cable-induced delays, by measuring round-trip cable delays. Using such techniques, the accuracy of these wallclock synchronization protocols is dependent on the delay differences between incoming and outgoing links, not the overall delay of either. Implementation of these wallclock synchronization protocols involves monitoring the arrival and departure time of specialized class-A frames, called clockSync frames, as described in this subclause.

The root station is responsible for generation of clockSync frames. All stations (root as well as nonroot) are responsible for measuring the clockSync propagation time through themselves. Clock deviations are sampled in cycle N and calibrations are performed in cycle N+1. Clock sampling involves through-station delays measurements and sampling of the station's *clockTime* value at its transmitter, as illustrated in Figure I.1.

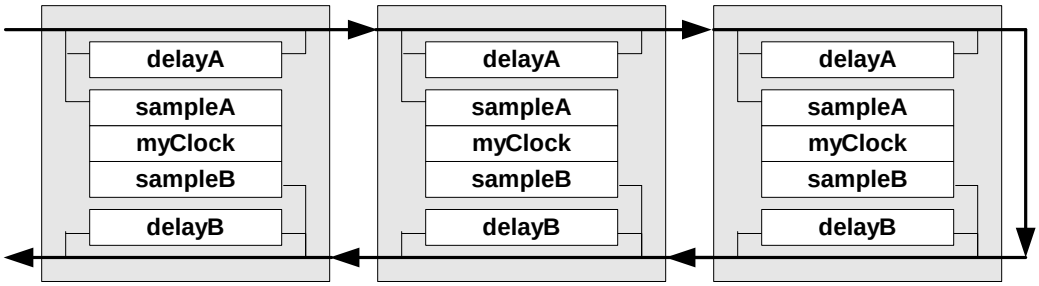



**Figure I.1—Clock and delay measurements**

The behavior on synchronization protocols is as follows:

1) Deviation. The station computes its clock deviation, as follows:
   *timeSink= (sampleA+sampleB+delayB)*/2;
   *timeDiff= timeSend–timeSink*;
2) Core. The core sends the average of the observed right-side times, as follows:
   *timeSendOut= timeSendIn+*(*delayA–delayB*)/2

### I.1.2 Time-of-day adjustments

The wallclock measurements in cycle N are used to adjust the clock-slave wallclock values in cycle N+1, as specified in equation 2. Initial synchronization involves setting the station's *clockTime* value, to minimize the clock-value lock-up delays. Maintaining synchronization involves clock-rate adjustments, to avoid *clockTime* discontinuities.

```
#define THRESHOLD ONE_SECOND/8000       // Adjust after 8KHz interval
#define TICK (CLOCK_NOMINAL/5000)       // 200PPM overcomes 100PPM
inaccuracy
delta= timeSink-timeSend;
if (Magnitude(delta)>(THRESHOLD/2))
  clockTime+= delta;
else
  clockRate+= difference>0 ? TICK:-TICK;
```

# Annex J: Background information (informative)

## Annex K: C code illustrations

```
//                                                                                                                   1         1         1         1
//       1         2         3         4         5         6         7         8         9         0         1         2         3
//23456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
//
//  Basic frame format:
//
//                        class --+  +-- ring
//                priority --+    |  | +-- wrap
//                           |    |  | |
//                           v    v  v v
//  +-------------------------------------------------------------+
//  |  timeToLive   |type | --- | - |-|-|    DSID     |    SSID     |
//  +-'-'-'-'-'-'-'-+-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-|
//  |                        destinationMacHi                     |
//  +-'-'-'-'-'-'-'-+-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-|
//  |       destinationMacLo        |          sourceMacHi          |
//  +-'-'-'-'-'-'-'-+-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+
//  |                          sourceMacHi                        |
//  +-'-'-'-'-'-'-'-+-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-|
//  |                             HEC32                           |
//  +-'-'-'-'-'-'-'-+-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+
//  |          typeLength           |                             |
//  +-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+                             |
//  |                                                             |
//  |                           payload                           |
//  |                                                             |
//  +-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+
//  |                             PCS32                           |
//  +-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+-'-'-'-'-'-'-'-+

// TBD:
//   class-A0/class-A1 transparency to the client?
//   Discarding of MAC-transmission excesses
//   Coarse accounting granularity
#define HOPS 64                                         // Worst-case number of hops
#define BYTES_PER_TICK 16                               // A coarse-grained accounting value
#define MTU (1536/BYTES_PER_TICK)                       // An estimated maximum-frame size
#define DTU (MTU+4*MTU)                                 // MTU plus MAC-to-client turn-around delay
#define MINIMUM(x,y) (((x)-(y))>0 ? (x):(y))            // Modulo-arithmetic-compatible maximum
#define MAXIMUM(x,y) (((x)-(y))<0 ? (x):(y))            // Modulo-arithmetic-compatible minimum
#define ONE (((uInt8)1)<<32)                            // Scalar for {integer:32, fraction:32} values
```

```
typedef unsigned char uInt1;           // assuming 'char' is an 8-bit value
typedef unsigned long uInt4;           // assuming 'long' is a 32-bit value
typedef unsigned long long uInt8;      // assuming 'long long' is a 64-bit value
typedef signed long sInt4;             // assuming 'long' is a 32-bit value
typedef signed long long sInt8;        // assuming 'long long' is a 64-bit value
typedef struct {
    //
    // Hop-count insensitive information
    sInt8 creditD;                     // Sustaining class-A idles
    sInt8 creditD1;                    // Sustaining class-A1 idles
    sInt8 creditBC;                    // Ensuring transitBC progress
    sInt8 creditC;                     // Limiting class-A transmit bandwidth
    uInt4 ratingA0;                    // Ringlet rating for class-A0 traffic
    uInt4 ratingA1;                    // Ringlet rating for class-A1 traffic
    uInt4 ratingC;                     // Maximum class-C transmit bandwidth
    uInt4 weightC;                     // Weighting for class-C traffic
    //
    // Hop-count sensitive information
    sInt8 creditA0[HOPS];              // Hop-count based class-A0 transmission credits
    sInt8 creditA1[HOPS];              // Hop-count based class-A1 transmission credits
    sInt8 creditB[HOPS];               // Hop-count based class-C transmission credits
    sInt8 countC[HOPS];                // Hop-count based class-C transmission counts
    uInt4 rateA0[HOPS];                // Hop-count based class-A0 transmission rates
    uInt4 rateA1[HOPS];                // Hop-count based class-A1 transmission rates
    uInt4 rateB[HOPS];                 // Hop-count based class-B transmission rates
    //
    // Information derived from downstream-sourced flow-control frame
    uInt4 sensedDepth;                 // Sense indication of downstream transitBC depth
    uInt4 limitC[HOPS];                // Other stations published countC run-rates
    uInt4 lowerC[HOPS];                // Other stations previously published countC
    //
    // Information asserted across the MAC-to-client interface, for flow-control purposes
    uInt1 rangeA0;                     // Client indication of class-A0 permission range
    uInt1 rangeA1;                     // Client indication of class-A1 permission range
    uInt1 rangeB;                      // Client indication of class-B permission range
    uInt1 rangeC;                      // Client indication of class-C permission range
    //
    // Performance parameters derived during the discovery process
    uInt1 hops;                        // Number of unwrapped hops
    uInt4 loopDelay;                   // Ringlet-circulation time, unwrapped and unloaded
} Station;

void  RangeAdjustments(Station *);
void  CreditAdjustments(Station *, int, int, int, int, int, int, int, int, int, int, int);
void  CreditLimits(int, uInt8 *, uInt4 *, uInt4, uInt4, sInt4, sInt4);
int   CreditUpdate(uInt8 *, uInt4, uInt4, uInt4, sInt4, sInt4);
int   DepthsToAssist(uInt4, uInt4);
uInt4 DepthToRateBC(uInt4);
uInt4 WeightToScaleC(uInt4);
```

```
uInt4 DepthTransitBC(Station *);        // Implementation-specific transitBC depth measurement
int   DepthStage(Station *);            // Implementation-specific stage-buffer nearly full
int   Staged(Station *);                // One or more stage-buffer entries are ready to send
int   Queued(Station *);                // One or more transitBC FIFO entries are ready to send

// Called to update credits
void
CreditAdjustments(Station *context,
 int waitT,                             // Time duration since last called
 int moveA0,                            // Class-A0 client-to-MAC transfer size
 int moveA1,                            // Class-A1 client-to-MAC transfer size
 int moveB,                             // Class-B client-to-MAC transfer size
 int moveC,                             // Class-C client-to-MAC transfer size
 int sendBC,                            // Lower-class transitBC retransmission size
 int sendA0,                            // Class-A0 output-transmission size
 int sendA1,                            // Class-A1 output-transmission size
 int sendB,                             // Class-B output-transmission size
 int sendC,                             // Class-C output-transmission size
 int sendI)                             // Udle-frame equivalent output-transmission size
{   uInt8 least;
    int assist, n, hops= context->hops;
    uInt4 depthTransitBC, rateBC, rateC;
    uInt4 hiSideA0, hiSideA1, hiSideB, hiSideD, hiSideD1, hiSideBC, loSideBC, scaleC, waitSizeC;
    //
    depthTransitBC= DepthTransitBC(context);
    hiSideA0= ((MTU*context->ratingA0)/ONE)+DTU;
    CreditLimits(HOPS, context->creditA0, context->rateA0, moveA0, waitT, hiSideA0, -DTU);
    //
    hiSideA1= ((MTU*context->ratingA1)/ONE)+DTU;
    CreditLimits(HOPS, context->creditA1, context->rateA1, moveA1, waitT, hiSideA1, -DTU);
    //
    hiSideB= 2*(context->loopDelay+hops*MTU)+DTU;
    CreditLimits(HOPS, context->creditB, context->rateB, moveB, waitT, hiSideB, -DTU);
    //
    scaleC= WeightToScaleC(context->weightC);
    waitSizeC= depthTransitBC>=(ONE/4) ? moveC : waitT;
    for (n=0; n<context->hops; n= n+1)
        CreditUpdate(context->countC+n, scaleC, 0, waitSizeC, context->limitC[n]+DTU, context->lowerC[n]);
    //
    hiSideD= DepthsToAssist(depthTransitBC, context->sensedDepth)!=0 ? DTU : 0;
    CreditUpdate(&(context->creditD), context->ratingA0+context->ratingA1, sendA0+sendA1+sendI, sendB+sendC, hiSideD, -DTU);
    //
    hiSideD1= ((2*hops*context->ratingA1)/ONE)+DTU;
    CreditUpdate(&(context->creditD1), context->ratingA1, sendA1+sendI, sendA0+sendB+sendC, hiSideD1, -DTU);
    //
    rateBC= DepthToRateBC(depthTransitBC);
    hiSideBC= Staged(context) ? DTU : 0;
    loSideBC= Queued(context) ? DTU : 0;
    CreditUpdate(&(context->creditBC), rateBC, moveB+moveC, sendBC, hiSideBC, -loSideBC);
    //
    CreditUpdate(&(context->creditC), context->ratingC, moveC, waitT, DTU, -DTU);
}
```

```
// Called to update range values
void
RangeAdjustments(Station *context)
{   int n, stopA, stopB, stopC;
    //
    stopA= DepthStage(context)>(ONE/2);
    stopB= stopA;
    stopB|= (context->creditD>0);
    stopB|= (context->creditD1>0);
    stopB|= (context->creditBC<0);
    stopC= stopB;
    stopC|= (context->creditC>0);
    //
    for (n=0; stopA==0 && n<HOPS && context->creditA0[n]>=0; n+= 1);
    context->rangeA0= n;
    //
    for (n=0; stopA==0 && n<HOPS && context->creditA1[n]>=0; n+= 1);
    context->rangeA1= n;
    //
    for (n=0; stopB==0 && n<HOPS && context->creditB[n]>=0; n+= 1);
    context->rangeB= n;
    //
    for (n=0; stopC==0 && n<HOPS && (context->limitC[n]-(context->countC[n]/ONE))>0; n+= 1);
    context->rangeC= n;
    return;
}

// Called to update CreditUpdate() over multiple hop counts
// Excessive longer-hop credits are also discarded
void
CreditLimits(int count, uInt8 *creditPtr, uInt4 *sendRate, uInt4 sendSize, uInt4 waitSize, sInt4 hiSide, sInt4 loSide)
{   uInt8 least;
    int i;
    //
    CreditUpdate(creditPtr, sendRate[0], sendSize, waitSize, hiSide, loSide);
    least= creditPtr[0];
    for (i=1; i<count; i= i+1) {
        CreditUpdate(creditPtr+i, sendRate[i], sendSize, waitSize, hiSide, loSide);
        least= MINIMUM(least, creditPtr[0]);
        creditPtr[i]= MINIMUM(creditPtr[i], least+DTU);
    }
}
```

```
// Arguments for leaky-bucket adjustments:
//   creditPtr - pointer to credit value
//   sendRate  - transmission rate limitation
//   sendSize  - size of rate-limited frame
//   waitSize  - size of other frames or idles
//   hiSide    - maximum credit value limit
//   loSide    - minimum credit value limit
// Return a value of 1 when low threshold is reached
CreditUpdate(uInt8 *creditPtr, uInt4 sendRate, uInt4 sendSize, uInt4 waitSize, sInt4 hiSide, sInt4 loSide)
{   uInt8 credits, hiLevel, loLevel;
    //
    credits= creditPtr[0];
    hiLevel= hiSide*ONE;
    loLevel= loSide*ONE;
    credits= credits+(waitSize*sendRate)-(sendSize*ONE);
    credits= MINIMUM(credits, hiLevel);
    credits= MAXIMUM(credits, loLevel);
    return(credits!= loLevel);
}

uInt4
DepthToRateBC(uInt4 depth)
{   uInt4 rate;

    rate= ONE-(2*depth);
    rate= MAXIMUM(rate, (ONE*7)/8);
    rate= MINIMUM(rate, 0);
    return(rate);
}

int
DepthsToAssist(uInt4 thisDepth, uInt4 thatDepth)
{
    if (thisDepth<=(ONE/4))
        return(0);
    if (thisDepth<(ONE/2))
        return(thisDepth<(2*thatDepth-ONE/2));
    return(thisDepth<thatDepth);
 }

uInt4
WeightToScaleC(uInt4 weight)
{
    if (weight>=(ONE/2))
        return((3*ONE)/16-weight/8);
    if (weight>=(ONE/4))
        return((3*ONE)/8-weight/2);
    if (weight>=(ONE/8))
        return((3*ONE)/4-2*weight);
    return(ONE-4*weight);
}
```

```
void PrintTable(int, int);
uInt4 GenerateCrc(int, uInt4 *, int);
int ValidateCrc(int, uInt4 *, int);
uInt4 CalculateCrc(int, uInt4 *, int);
uInt4 CrcBits(uInt4, uInt4, int);
uInt4 BitReverse(uInt4);
void Error(char *);
void assert(int);
//
#define MSB32  ((unsigned)1<<31)
#define ONES32 0XFFFFFFFF
#define CRC_COMPUTE ((uInt4)0X04C11DB7)
#define CRC_RESULTS ((uInt4)0XC704DD7B)
//
int
main(int argc, char **argv) {
    int i;
    int size=32, reverse=1, table=1;
    int setRev= 0, setHow=0;
    char *argPtr;

    // Command line specifies number of bits computed in parallel
    for (i= 1; i<argc; i+= 1) {
        argPtr= argv[i];
        if (*argPtr!='-')
            Error("Illegal argument, use: -n -r -tdd -c");
        argPtr+= 1;
        switch (*argPtr) {
        case 'n':
        case 'r':
            if (setRev)
                Error("Mutually exclusive options: -n -r");
            reverse= (*argPtr=='r');
            setRev= 1;
            break;
        case 't':
            if (setHow)
                Error("Mutually exclusive options: -tdd -c");
            size = atoi(argPtr+1);
            if (size != 1 && size != 2 && size != 4 &&
             size != 8 && size != 16 && size != 32)
                Error("Incorrect width; -t1 -t2 -t4 -t8 -t16 or -t32\n");
            table= 1;
            break;

        case 'c':
            if (setHow)
                Error("Mutually exclusive options: -wdd -c");
            table= 0;
            setHow= 1;
            break;
```

```
        default:
            Error("Arguments: -n -r -t[1,2,4,8,16,32] -c\n");
            break;
        }
    }
    assert(table);
    PrintTable(reverse, size);
    return(0);
}

void
Error(char *string)
{
    printf(string);
    exit(1);
}

void
assert(int test)
{
    if (test==0)
        exit(1);
}
```

```
char keys[]= {'a','b','c','d','e','f','g','h','j','k','m','n','p','r','s','t'};
void
PrintTable(int reverse, int size)
{
    uInt4 last, next, select, mask, sum;
    int i, j, numb;
    for (i=0; i<32; i+= 1) {
        /* Calculate contributing-input values */
        select= 1 << (31-i);
        mask= reverse ? BitReverse(select) : select;
        printf("c%02d= ", i);
        for (j= sum= 0; j < 32; j+= 1) {
            last= 1<<(31-j);
            next = CrcBits(last, (uInt4)0, size);
            if (next&mask)
                sum|= last;
        }
        for (j= 0; j<32; j+= 1) {
            select= 1<<(31-j);
            mask= reverse ? BitReverse(select) : select;
            numb= j<16 ? keys[j] : keys[j-16]+'A'-'a';
            if ((sum & mask) != 0) {
                sum&= ~mask;
                printf("%c", numb);
                if (j != 31)
                printf(sum!=0 ? "^" : " ");
            } else {
                printf(j!= 31 ? "  " : " ");
            }
        }
      printf(";\n");
    }
}

// Generate the CRC in packet containing "sizeInQuads" quadlet data values
uInt4
GenerateCrc(int reverse, uInt4 *inputs, int sizeInQuads)
{
    uInt4 crcSum;

    assert(sizeInQuads >= 1);                   // Packet size including CRC

    crcSum = CalculateCrc(reverse, inputs, sizeInQuads - 1);
    // compute CRC on just the data
    return (~crcSum);
}

// Validate the CRC for a packet containing "size" quadlet data values
int
ValidateCrc(int reverse, uInt4 *inputs, int sizeInQuads)
{
    uInt4 crcSum, check;
```

```
    assert(sizeInQuads >= 1);                 // Packet size including CRC

    crcSum = CalculateCrc(reverse, inputs, sizeInQuads);
    /* compute CRC on the data * and * the received CRC */
    check = reverse ? BitReverse(crcSum) : crcSum;
    return (check != CRC_RESULTS);
}

// The GenerateCrc() function points to protected values,
// it checks these values and return a final 32 - bit result
uInt4
CalculateCrc(int reverse, uInt4 *inputs, int sizeInQuads)
{
    uInt4 inQuad, crcSum, sum;
    int i;

    // The crcSum value is initialized to all ones
    crcSum= (uInt4) 0XFFFFFFFF;

    // Process each of the quadlets covered by the CRC value
    for (i = 0; i < sizeInQuads; i += 1) {
        inQuad= reverse ? inputs[i] : BitReverse(inputs[i]);
        crcSum= CrcBits(crcSum, inQuad, 32);
    }
    sum= reverse ? BitReverse(crcSum) : crcSum;
    return (sum);
}

uInt4
CrcBits(uInt4 last, uInt4 input, int size)
{
    uInt4 crcSum, newMask;
    int i, oldBit, newBit, sumBit;

    // Process each of the bits within the input quadlet value
    crcSum= last;
    for (newMask = MSB32, i= 0; i < size; newMask >>= 1, i+= 1) {
        newBit = ((input & newMask) != 0);      // The next input bit
        oldBit = ((crcSum & MSB32) != 0);       // and MSB of crcSum
        sumBit = oldBit ^ newBit;               // are EXOR'd together

        // Shift the old crcSum left and exclusive - OR the new newBit values
        crcSum = ((crcSum << 1) & ONES32) ^ (sumBit ? CRC_COMPUTE : 0);
    }
    return(crcSum);
}
```

```
// Reverse the order of bits within bytes
uInt4
BitReverse(uInt4 old)
{
    uInt4 new, oldMask, newMask;
    int i, j;

    for (i= new= 0; i < 4; i+= 1) {
        for (j= 0; j < 8; j+= 1) {
          oldMask= 1 << (8*i + 7 - j);
          newMask= 1 << (8*i + j);
          new|= (old & oldMask) ? newMask : 0;
        }
    }
    return(new);
}
```