<<Commenter’s Note (Rodney Cummings): This document provides text for the comments from Rodney Cummings on editor’s draft 802.1CB D0.3. This text is intended as a starting point for new content, with a focus on the technical description. If the comments are accepted in principle, the 802.1CB editor is welcome to change the text for grammar and clarity.>>

Comment A: Text for Sequencing sublayer

<<Commenter’s Note (Rodney Cummings): According to clauses 5 and 6, the “TSN encapsulation” sublayer manipulates the MSDU (packet) for the circuit_identifier parameter, and the “Sequence encapsulation” sublayer manipulates the MSDU for the sequence_number parameter. The “Sequencing” sublayer operates on circuit_identifier and sequence_number as input/output parameters, and it does not manipulate the MSDU. The benefit of this methodology is that the algorithms specified for the Sequencing sublayer can be used with any network technology, with no specific knowledge of the packet’s encoding.>>

<<Commenter’s Note (Rodney Cummings): This comment assumes the following addition to subclause 6.1.2, “Additional Seamless Redundancy service parameters”:

- ingress_port_num: This additional parameter specifies the port from which the packet was received. The ingress_port_num parameter is an Unsigned integer in the range 1 through 4095 (consistent with 802.1Q 12.3). The ingress_port_num is presumably the same as the 802.1AC service_access_point_identifier parameter of the M_UNITDATA.indication (prior to the network node’s relay). The Sequence Recovery sublayer is applied as the packet transmits, after the network node’s relay (e.g. Figure 6-5), and therefore the service_access_point_identifier parameter of M_UNITDATA.request provides the port for transmit. The additional ingress_port_num parameter provides the received port number to M_UNITDATA.request. The ingress_port_num parameter is required by the sequence recovery algorithms for diagnostic purposes, in order for network operators to identify the location of a fault.

>>

6.3 Sequencing sublayer

Unlike other sublayers in this standard, the sequencing sublayers are not truly peer entities, in that an M_UNITDATA.request passed down to the sequence generation sublayer does not necessarily match with the M_UNITDATA.indication passed up from the sequence recovery sublayer elsewhere in the network. Nevertheless, these sublayers are designed such that they can be applied using the same layering concepts as other IEEE 802.1 sublayers.

The sequencing sublayer exists above the sequence encapsulation sublayer.

6.3.1 Variables for sequence generation and sequence recovery

This subclause specifies variables that apply to both the sequence generation and sequence recovery sublayers.

In variable names, the prefix “oper” indicates a variable that is set from management or another higher layer entity.

In variable names, the prefix “diag” indicates a variable that is that used to diagnose streams of redundant traffic. In order for seamless redundancy to serve its intended purpose, it must be possible for an operator to identify when one or more redundant paths have stopped, and to replace the physical component that has failed. Diagnostic variables are read from management or another higher layer entity for this purpose.

<<Commenter’s Note (Rodney Cummings): Variables for capabilities (e.g. algorithms supported) will be specified in the Management clause (not here).>>

6.3.1.1 operReset

The operReset variable provides a mechanism for higher layer entities to reset the sublayer, such as when starting or stopping transmit of data on a circuit. This standard does not use special sequence number values (e.g. zero) to specify a reset, because such a value can be lost. Higher layer protocols can provide a more reliable mechanism for reset of the algorithm. The operReset variable is boolean. The sublayer resets when the value transitions from FALSE to TRUE. There is one operReset per circuit per port of a network node.

6.3.1.2 operSeqNumBits

The operSeqNumBits variable specifies the number of bits used for the sequence number in packets. Valid sequence numbers range from 0 through ($2^{operSeqNumBits}$ - 1), with an increment of value ($2^{operSeqNumBits}$ - 1) rolling over to value 0. The variable is an Unsigned integer value in the range 1 to (2^{28} - 1). The sequence number field within each packet is assumed to be at least 28 bits in size. There is one operSeqNumBits per circuit of a network node.

For a circuit using the intermittent-streams algorithm for sequence recovery (operAlgorithmID, 6.3.3.1.10), the default value for operSeqNumBits shall be 16.

For a circuit using the bulk-streams algorithm for sequence recovery (operAlgorithmID, 6.3.3.1.10), the default value for operSeqNumBits shall be 8.

<<Commenter’s Note (Rodney Cummings): An 8-bit number for the bulk algorithm requires a 256-bit discardHistory per stream. I assume that is a reasonable default. For the intermittent algorithm we only keep the expectedSeqNum per circuit (no discardHistory), so I assume the default can be larger.>>

6.3.2 Sequence generation

The sequence generation sublayer increments the sequence_number parameter to ensure that each data packet is unique for each circuit.

For each circuit, sequence generation shall be enabled in only one location in the network. The location can be within the host that transmits packets for the circuit. When the location is a network node, it is typically the network node at the edge of the network, closest to the host that transmits packets for the circuit.

When sequence generation is enabled in a network node, it shall be enabled on the ingress port for the circuit, prior to the network node’s relay. This ensures consistent numbering per circuit regardless of each relay’s operation for filtering or topology.

The sequence generation sublayer shall be enabled only for M_UNITDATA.request (upper to lower layer), not for M_UNITDATA.indication (lower to upper layer).

The sequence generation sublayer does not encode the sequence number field within the packet (MSDU) that passes through, because the sequence encapsulation sublayer performs that task. Therefore, the sequence generation sublayer obtains the current sequence number from a per-circuit variable (txSeqNum, 6.3.2.1.1), and not the sequence_number parameter passed from the upper layer.

6.3.2.1 Variables for sequence generation

This subclause specifies variables specific to the sequence generation sublayer.

6.3.2.1.1 txSeqNum

The txSeqNum variable specifies the sequence number that the sequence generation sublayer increments and passes down to the lower layer as the sequence_number parameter. The variable is an Unsigned integer value in the range 0 to ($2^{operSeqNumBits}$ - 1). There is one txSeqNum per circuit per egress port of a network node. The variable applies only to circuits for which the sequence generation sublayer is enabled.

6.3.2.2 Sequence generation algorithms

Packets whose circuit_identifier parameter indicate that the packet is not a circuit of
interest to the sequence generation sublayer are passed through without action.

Each algorithm is specified as pseudo code for two procedures.

For packets using a known circuit_identifer, the SequenceGenerationAlgorithm procedure is performed for each packet that passes from the upper layer to the sequence generation layer.

The SequenceGenerationReset procedure is performed when the circuit resets (operReset variable of 6.3.1.1 transitions TRUE).

Within the pseudo code, mathematical operations on sequence numbers assume an integer that is operSeqNumBits (6.3.1.2) in size. For example, if the number of bits for a sequence number is 8, then the expression ((255 + 3) < 5) evaluates to TRUE, because (2 < 5).

6.3.2.2.1 SequenceGenerationAlgorithm

```
txSeqNum = txSeqNum + 1;
   // Pass new value down as parameter to sequence encapsulation.
sequence_number = txSeqNum;
```

6.3.2.2.2 SequenceGenerationReset

```
txSeqNum = 0;
```

6.3.3 Sequence recovery

The sequence recovery sublayer evaluates the sequence_number parameter in order to discard duplicate packets for a given circuit_identifier. This effectively merges redundant traffic in the network.

Sequence recovery is typically enabled in multiple locations throughout the network, within any network node at which redundant packets can be received from multiple ports.

When sequence recovery is enabled in a network node, it shall be enabled on the egress port for the circuit, after the network node's relay. This ensures removal of duplicates from multiple ingress ports.

The sequence recovery sublayer shall be enabled only for M_UNITDATA.indication (lower to upper layer), not for M_UNITDATA.request (upper to lower layer).

Packets whose circuit_identifier parameter indicate that the packet is not a circuit of interest to the sequence recovery sublayer are passed through without action.

For packets using a known circuit_identifier, the variable operAlgorithmID (6.3.3.1.10) shall determine the algorithm that is used to discard duplicate packets.

<<Commenter's Note (Rodney Cummings): I assume that 802.1CB will specify algorithms in order to clarify the feasibility of the implementation, as well as provide consistent interoperability.>>

6.3.3.1 Variables for sequence recovery

This subclause specifies variables specific to the sequence recovery sublayer.

6.3.3.1.1 accept

The accept variable is boolean. For the packet passed as input from the lower layer, the accept variable determines whether the packet is accepted and passed to the upper layer (TRUE), or discarded by the algorithm (FALSE). When accept is FALSE, M_UNITDATA.indication is not invoked for the layer above sequence recovery.

6.3.3.1.2 diagNumAcceptInOrder

The diagNumAcceptInOrder variable specifies the number of packets accepted in-order by the algorithm for a specific circuit from a specific ingress port. In-order means that the packet's sequence number is one more than the previous packet, indicating no packet loss.

Each entry in the diagNumAcceptInOrder table is an Unsigned integer value in the range 0 through (2^{32} - 1). There is one diagNumAcceptInOrder table per circuit per egress port of a network node. The diagNumAcceptInOrder table is indexed using an ingress port number of the network node.

<<Commenter's Note (Rodney Cummings): For diagnostics, the network operator must be able to identify the specific link (LAN) that has failed, which implies that we must track status for each ingress port for each circuit's algorithm (which is per circuit per egress port).

I provide diagnostics through simple counters, under the assumption that protocols like MSRP can take this information as input, and provide more specific information like "redundant stream no longer received on ingress port N of bridge address Y".>>

6.3.3.1.3 diagNumAcceptOutOfOrder

The diagNumAcceptOutOfOrder variable specifies the number of packets accepted out-of-order by the algorithm for a specific circuit from a specific ingress port. Out-of-order means that the packet's sequence number is not one more than the previous packet, indicating loss of one or more packets.

Each entry in the diagNumAcceptOutOfOrder table is an Unsigned integer value in the range 0 through (2^{32} - 1). There is one diagNumAcceptOutOfOrder table per circuit per egress port of a network node. The diagNumAcceptOutOfOrder table is indexed using an ingress port number of the network node.

6.3.3.1.4 diagNumDiscard

The diagNumDiscard variable specifies the number of packets discarded by the algorithm for a specific circuit from a specific ingress port. The packet is discarded due to a duplicate sequence number.

Each entry in the diagNumDiscard table is an Unsigned integer value in the range 0 through (2^{32} - 1). There is one diagNumDiscard table per circuit per egress port of a network node. The diagNumDiscard table is indexed using an ingress port number of the network node.

6.3.3.1.5 diagNumError

The diagNumError variable specifies the number of erroneous packets discarded by the algorithm for a specific circuit from a specific ingress port.

This variable only applies to the bulk-streams sequence recovery algorithm. It is zero for all other algorithms. The bulk-streams algorithm maintains a history of past sequence numbers (discardHistory, 6.3.3.1.7). If a packet's sequence number precedes this history, this may indicate a problem beyond packet loss (e.g. faulty component), and therefore the error is counted separately from diagNumDiscard (6.3.3.1.4).

Each entry in the diagNumError table is an Unsigned integer value in the range 0 through (2^{32} - 1). There is one diagNumError table per circuit per egress port of a network node. The diagNumError table is indexed using an ingress port number of the network node.

6.3.3.1.6 diagNumLost

The diagNumLost variable specifies the number of packets lost for a specific circuit from a specific ingress port. A packet is counted as lost if its sequence number is not received on any ingress port.

The diagNumLost variable is an Unsigned integer value in the range 0 through (2^{32} - 1). There is one diagNumLost per circuit per egress port of a network node.

6.3.3.1.7 discardHistory

The discardHistory variable maintains a history that specifies whether a packet with an older than expected sequence number was accepted or discarded. This variable applies only to the bulk streams algorithm. The discardHistory variable represents a shift register, with one bit for each value from 0 to (operAcceptWindow - 1). For example, if operAcceptWindow is 128, the size of discardHistory is 128 bits (16 octets). There is one discardHistory per circuit per egress port of a network node.

6.3.3.1.8 expectedSeqNum

The expectedSeqNum variable specifies the next sequence number that is expected to be received for this circuit. The expectedSeqNum variable is an Unsigned integer value in the range 0 through ($2^{\text{operSeqNumBits}}$ - 1). There is one expectedSeqNum per circuit per egress port of a network node.

6.3.3.1.9 operAcceptWindow

The operAcceptWindow variable specifies the number of sequence numbers that shall be accepted for this circuit. This variable is typically set according to the maximum difference in the number of packets for this circuit that can be in transit along the longest redundant path as compared to the shortest redundant path. The variable is an Unsigned integer value in the range 1 through $((2^{operSeqNumBits} / 2) - 1)$. In order to account for rollover of the sequence number values within the algorithm, the limit is one less than one-half the total number of sequence number values. There is one operAcceptWindow per circuit per egress port of a network node.

6.3.3.1.10 operAlgorithmID

The operAlgorithmID variable identifies the algorithm used in this network node for sequence recovery. The variable is an Unsigned integer, with valid values specified in Table X-Y. All other values are reserved for future use by this standard. There is one operAlgorithmID per circuit per egress port of a network node.

<<Commenter's Note (Rodney Cummings): Scoping the algorithm to per-circuit allows for a mixture of use cases. For example, an industrial network can use a handful of bulk streams for measurements or A/V, and a larger number of intermittent streams for control data.>>

<<Commenter's Note (Rodney Cummings): In the draft, add a table with

value 1 for "intermittent streams"

value 2 for "bulk streams"

>>

6.3.3.2 Sequence recovery algorithms

Each algorithm is specified as pseudo code for two procedures.

The SequenceRecoveryAlgorithm procedure is performed for each packet that passes from the lower layer to the sequence recovery layer. If the SequenceRecoveryAlgorithm accepts the packet (accept variable of 6.3.3.1.1 is TRUE), the packet passes from the sequence recovery layer to the upper layer.

The SequenceRecoveryReset procedure is performed when the circuit resets (operReset variable of 6.3.1.1 transitions TRUE).

The following mathematical operators within the pseudo-code have a specific semantic:

- For sequence numbers, mathematical operations assume an integer that is operSeqNumBits (6.3.1.2) in size. For example, if the number of bits for a sequence number is 8, then the expression ((255 + 3) < 5) evaluates to TRUE, because (2 < 5).
- For diagnostic variables, increment operations cannot exceed the maximum values of variables (i.e. increment operations do not change the value if the maximum value is already reached).
- For discardHistory, the left shift operation ('<<') shifts values from higher to lower indices and resets the undefined values (with higher indices) to FALSE. For example, if discardHistory contains the value [TRUE@0, TRUE@1, FALSE@2], left shift by one bit changes the value to

[TRUE@0, FALSE@1, FALSE@2]. Since the lower indices of discardHistory represent the oldest sequence numbers, the left shift effectively forgets the past.
- For discardHistory, the procedure falseCountDiscardHistory(*n*) returns the number of FALSE values in the lowest (oldest) *n* indices of discardHistory.

<<Commenter's Note (Rodney Cummings): I am working under the assumption that the purpose of the standard is to clearly specify externally visible behavior, and not to dictate a specific implementation. Therefore, the pseudo code is not highly optimized, because it is not intended to be used literally as code for an implementation. This may be a topic for discussion.>>

6.3.3.2.1 Sequence recovery algorithm for intermittent streams

This algorithm is used for a circuit when operAlgorithmID is 1.

This algorithm is best suited to streams for which a single packet is in transit from talker to listeners at any moment in time (i.e. streams with a transmit rate slower than worst latency). The algorithm works to ensure that packets are delivered in order. Since the algorithm does not maintain a history of sequence numbers, the number of bits in the sequence number (operSeqNumBits, 6.3.1.2) can be large, which in turn allows for a larger difference in transit time along each redundant path (e.g. large ring topology).

6.3.3.2.1.1 SequenceRecoveryAlgorithm

```
if (sequence_number == expectedSeqNum) {
      // Packet is within the acceptance window, in expected order.
   accept = TRUE;
   diagNumAcceptInOrder[ingress_port_num]++;
      // Expect the next sequence number to be one more than current.
   expectedSeqNum = sequence_number + 1;

} else if ( (sequence_number - expectedSeqNum) >= operAcceptWindow) {
      // Packet is not within the acceptance window, so we discard.
   accept = FALSE;
   diagNumDiscard[ingress_port_num]++;

} else {
      // Packet is within the acceptance window, but the sequence number
      // is greater than expected.
   accept = TRUE;
   diagNumAcceptOutOfOrder[ingress_port_num]++;
      // We assume that packets from expected to current were lost.
   diagNumLost = diagNumLost + (sequence_number - expectedSeqNum);
      // Expect the next sequence number to be one more than current.
   expectedSeqNum = sequence_number + 1;
}
```

6.3.3.2.1.2 SequenceRecoveryReset

For the purposes of this procedure, NumPorts represents the number of ports in the network node. The variable “i” is a local integer variable.

```
expectedSeqNum = 0;
diagNumLost = 0;
for (i = 1; i <= NumPorts; i++)
   diagNumDiscard[i] = diagNumError[i] =
      diagNumAcceptInOrder[i] = diagNumAcceptOutOfOrder[i] = 0;
```

6.3.3.2.2 Sequence recovery algorithm for bulk streams

This algorithm is used for a circuit when operAlgorithmID is 2.

This algorithm is best suited to streams for which multiple packets are in transit from talker to listeners at any moment in time (i.e. streams with a transmit rate faster than worst latency). When one or more redundant paths are faulted or restored, the algorithm can deliver packets out of order, under the assumption that higher layer streaming protocols can re-order the packets. The algorithm maintains a history of sequence numbers (discardHistory, 6.3.3.1.7) for each circuit. Due to limitations in the size of this history, this algorithm may limit the difference in transit time along each redundant path (e.g. no large ring topologies).

6.3.3.2.2.1 SequenceRecoveryAlgorithm

```
if (sequence_number == expectedSeqNum) {
      // Packet is within the acceptance window, in expected order.
   accept = TRUE;
   diagNumAcceptInOrder[ingress_port_num]++;
   diagNumLost = diagNumLost + falseCountDiscardHistory(1);
      // Advance the discardHistory shift register by one.
   discardHistory = discardHistory << 1;
      // Expect the next sequence number to be one more than current.
   expectedSeqNum = sequence_number + 1;

} else if ( (expectedSeqNum – sequence_number) <= operAcceptWindow) {
      // The accept window applies to past numbers as well as future numbers.
      // For a packet in the past, we evaluate its history to decide whether to
      // accept or discard.
   if (discardHistory[operAcceptWindow - (expectedSeqNum - sequence_number)] )
      {
            // History for sequence_number is TRUE, so it was accepted
            // in the past, and therefore we discard this packet.
         accept = FALSE;
         diagNumDiscard[ingress_port_num]++;
      } else {
            // History for sequence_number is FALSE, so it was not accepted
            // in the past, and therefore we accept this packet (out of order),
            // and mark it as accepted in history.
         accept = TRUE;
         diagNumAcceptOutOfOrder[ingress_port_num]++;
         discardHistory[operAcceptWindow - (expectedSeqNum - sequence_number)] =
            TRUE;
      }

} else if ( (sequence_number – expectedSeqNum) >= operAcceptWindow) {
      // Packet is not within the acceptance window (error).
   accept = FALSE;
   diagNumError[ingress_port_num]++;

} else {
      // Packet is within the future acceptance window, but the sequence number
      // is greater than expected.
   accept = TRUE;
   diagNumAcceptOutOfOrder[ingress_port_num]++;
   diagNumLost = diagNumLost + falseCountDiscardHistory(
      (sequence_number – expectedSeqNum + 1) );

      // We are accepting a packet that is newer than the discardHistory,
      // so we need to advance the discardHistory shift register.
   discardHistory = discardHistory << (sequence_number – expectedSeqNum + 1);
      // Expect the next sequence number to be one more than current.
   expectedSeqNum = sequence_number + 1;
      // Mark the current packet as accepted in history.
   discardHistory[operAcceptWindow - 1] = TRUE;
}
```

6.3.3.2.2.2 SequenceRecoveryReset

For the purposes of this procedure, NumPorts represents the number of ports in the network node. The variable “i” is a local integer variable.

```
expectedSeqNum = 0;
discardHistory = 0; // all bits FALSE
diagNumLost = 0;
for (i = 1; i <= NumPorts; i++)
   diagNumDiscard[i] = diagNumError[i] =
      diagNumAcceptInOrder[i] = diagNumAcceptOutOfOrder[i] = 0;
```

~~6.3.4 Sequence number reset~~

<<Commenter’s Note (Rodney Cummings): This subclause exists in D0.3, but I remove it from this comment, because it is covered by the SequenceRecoveryReset algorithms specified above.>>

Comment B: Text for Sequence encapsulation sublayer

6.4 Sequence encapsulation sublayer

<<Editor’s note: This is the general description of the function of encoding sequence_number values without circuit_identifier values. At least one specific means will be provided (6.4.1).>>

The sequence encapsulation sublayer exists above the split/merge sublayer. If the split/merge sublayer does not exist, the sequence encapsulation sublayer exists above the circuit encoding sublayer.

6.4.1 Redundancy tag

IEEE 802.1Q-2011 clause 9 specifies the tagged frame format. A tag represents information that is used within the network by IEEE 802.1 standards, but not necessarily by the user applications that transmit and receive data frames. Within the Enhanced Internal Sublayer Service (EISS) described in subclause 6.9 of IEEE 802.1Q-2011, the tag is inserted in the data frame as it transmits to the network (M_UNITDATA.request), and the tag is removed as it is received from the network (M_UNITDATA.indication).

The sequence encapsulation sublayer inserts/removes a redundancy tag in the data frame in a manner similar to the IEEE 802.1Q tag. The redundancy tag contains the sequence_number parameter of this standard.

When M_UNITDATA.request is passed down from an upper sublayer to the sequence encapsulation sublayer, the redundancy tag is inserted into the MSDU (encapsulation), and the sequence_number parameter is encoded into the Tag Control Information (TCI, 6.4.1.3).

When M_UNITDATA.indication is passed up from a lower sublayer to the sequence encapsulation sublayer, the sequence_number is decoded from the TCI to be passed up as a parameter, and the redundancy tag is removed from the MSDU (decapsulation).

The representation and encoding of tag fields in this subclause uses the conventions specified in IEEE 802.1Q-2011, subclause 9.2.

6.4.1.1 Tag format

Each redundancy tag comprises the following sequential information elements:

a) Tag Protocol Identifier (TPID) (6.4.1.2);
b) Tag Control Information (TCI) (6.4.1.3);

The tag encoding function supports each sequence encapsulation instance by using an instance of the IEEE 802.1Q External Internal Sublayer Service (EISS) to transmit and receive frames, and encodes the above information in the first and subsequent octets of the MSDU that will accompany an EISS M_UNITDATA.request, immediately prior to encoding the sequence of octets that constitute the corresponding sequence encapsulation M_UNITDATA.request's MSDU. On reception the tag decoding function is selected by the TPID and decodes the TCI prior to issuing a sequence encapsulation M_UNITDATA.indication with an MSDU that comprises the subsequent octets.

<<Commenter's Note (Rodney Cummings): The preceding paragraph was adapted from IEEE 802.1Q-2011 9.3. Others who are more versed in 802.1 may be able to phrase this in a better manner.>>

As a consequence of the preceding specification, if an IEEE 802.1Q tag exists in the frame, the redundancy tag is encoded in the MSDU after the IEEE 802.1Q tag.

Tag Protocol Identifier (TPID) formats for the redundancy tag (i.e. EtherType encoding) shall follow the specifications of IEEE 802.1Q-2011 subclause 9.4.

<<My reference to 802.1Q-2011 9.4 may need to be updated to include 802.1Qbz, assuming 802.11 is no longer considered an LLC media.>>

Figure 7-X shows an example of the redundancy tag.

<<Commenter's Note (Rodney Cummings): Insert a Figure 7-X similar to slide 13 of:
http://www.ieee802.org/1/files/public/docs2014/cb-kiessling-CB-Layer2-Tag-0314-v01.pdf
Show the TPID and TCI fields of the redundancy tag. The TCI field is 32 bits, with the upper 4 bits reserved, and the lower 28 bits a sequence number.>>

6.4.1.2 Tag Protocol Identifier (TPID)

The Tag Protocol Identifier (TPID) of the redundancy tag shall use the following value:

xx-yy

<<Commenter's Note (Rodney Cummings): 802.1 will obtain an EtherType value for xx-yy from IEEE Registration Authority at the appropriate time.>>

6.4.1.3 Tag Control Information (TCI)

The Tag Control Information (TCI) consists of a single 32-bit value, with the first octet of the TCI containing the most significant bits (as per IEEE 802.1Q-2011 9.2).

The lower 28 bits of the TCI contain the sequence number. The upper 4 bits are reserved for future use by this standard, and shall be transmitted as zero and ignored on receipt.

Comment C: Text for Split/Merge sublayer

6.5 Split/Merge sublayer

The split/merge sublayer exists above the circuit encoding sublayer.

In IEEE 802.1 Time-Sensitive Networking (TSN) standards, each circuit (stream) within the network is identified by a group Destination MAC Address (DA), and a VLAN ID (VID). This corresponds to operCircuitFormat (6.x.1.1) value 1 of the circuit decoding sublayer. As frames for each TSN stream are forwarded from bridge to bridge, the DA/VID pair does not change. Since there is no requirement to change the circuit_identifier within a bridge, the split/merge sublayer is not needed for TSN streams.

<<Commenter's Note (Rodney Cummings): Subsequent edits will presumably add subclauses for other operCircuitFormat values such as MPLS pseudowire. We also might need a subclause for PRP-to-TSN interoperability, because that may need change the circuit_identifier (without a 'split' or 'merge').>>

Comment D: Text for new Circuit Encoding sublayer

<<Commenter's Note (Rodney Cummings): The AVB task group made the wise decision to standardize each feature as standalone as possible, with the term "AVB" used only in 802.1BA as a way to package all of the features together. This provided benefits that carried forward to the TSN task group. For example, since 802.1AS is a standalone document, 802.1Qbv is also standalone (i.e. uses any 1588 profile).

802.1CB should be written as a standalone document, without dependencies on other TSN features. For example:

- It should be possible to use 802.1CB with non-TSN traffic (not shaped/scheduled) to reduce packet loss.
- It should be possible to use the TSN encapsulation feature at the edge of the network, without using seamless redundancy. For example, A/V applications need the encapsulation concept to replace IP-based addresses with an 802.1 group destination address, and that feature is needed for a simple RSTP-only topology (without 802.1CB).

802.1CB requires a sublayer to encode/decode the circuit_identifier within a packet, in order for the Sequencing sublayer to handle the sequence_number on a per-circuit basis.

802.1CB does **not** require a sublayer that replaces one format of circuit_identifier within a packet with a different format of circuit_identifier within a packet. For example, the distinction between the "Null TSN encapsulation" (D0.3, 5.4), and the "Multicast MAC TSN encapsulation" (D0.3, 5.5) is irrelevant to the seamless redundancy layers (D0.3, 6.2), because for both of those encapsulations, the circuit_identifier represents DA/VID.

The TSN encapsulation layer is an important feature for TSN standards, but it does not belong in 802.1CB. It should be integrated into 802.1Q, or potentially in a distinct standard.

This comment provides text for a new Circuit Encoding sublayer that meets seamless redundancy requirements. Implementations may decide to merge this Circuit Encoding sublayer with the TSN Encapsulation sublayer, just as they would merge many of the layers shown in D0.3 6.6.

>>

6.x Circuit encoding sublayer

<<Commenter's Note (Rodney Cummings): This subclause would presumably be inserted between the existing 6.5 and 6.6, but I leave it as 6.x in this comment.>>

The sublayers for seamless redundancy require use of a circuit_identifier to uniquely identify each redundant flow of data. The sequence number increments uniquely for each circuit_identifier. The circuit encoding sublayer encodes/decodes the circuit_identifier parameter for various packet formats.

The circuit encoding sublayer exists above the media access for the link (e.g. IEEE 802.1Q EISS).

The additional sequence_number and ingress_port_number parameters (6.1.2) are not contained in the service interface for the circuit encoding sublayer (only circuit_identifier).

6.x.1 Variables for circuit encoding

This subclause specifies variables that apply to the circuit encoding sublayer.

In variable names, the prefix "oper" indicates a variable that is set from management or another higher layer entity.

6.x.1.1 operCircuitFormat

The operCircuitFormat variable identifies the format used on this port to encode and decode the circuit_identifier parameter. The variable is an Unsigned integer, with valid values specified in Table X-Y. All other values are reserved for future use by this standard. There is one operCircuitFormat per port of a network node.

<<Commenter's Note (Rodney Cummings): In the draft, add a table with

value 1 for "Destination MAC Address and VLAN ID"

value 2 for "Source MAC Address and VLAN ID

value 3 for "Source MAC Address and Destination MAC Address and VLAN ID"

value 4 for "IP octuple"

>>

For formats that use a VLAN ID, the VLAN ID for untagged frames and priority-tagged frames is determined according to IEEE 802.1Q-2011 rules (i.e. PVID).

<<Commenter's Note (Rodney Cummings): This initial list of formats originates from slide 22 of:

http://www.ieee802.org/1/files/public/docs2014/cb-kiessling-Details-0514-v01.pdf

I added the IP octuple of D0.3 subclause 5.6. We can change this list as we go along.

As written, this spec allows the format to be different for each port, such as to translate format 2 (PRP) to format 1 (AVB). Therefore, at some point in the future, we will need to specify the encode/decode of each format very precisely, so that the circuit_identifier is understood from one format to another.

>>

6.x.2 Circuit identifier encode/decode

When M_UNITDATA.request passes down to the circuit encoding sublayer, the circuit_identifier parameter is encoded into the packet, which then passes down to the lower layer.

When M_UNITDATA.indication passes up from the lower layer, the circuit identifier is decoded from the packet, and passed to the upper layer as the circuit_identifier parameter.

Comment E: Text for Annex A (PICS)

<<Commenter's Note (Rodney Cummings): We need to clearly specify the required features for an "802.1CB bridge" (original PAR scope), so that automotive and industrial standards can reference that feature in 802.1CB. This text assumes that we will be consistent with 802.1Q, and use distinct conformance for bridge, end-station, and other types of products (router, "IP host", etc).>>

I was not sure if we need conformance for "802.1CB end-station". A PRP-like dual-homed end-station can be implemented as a bridged end-station, in which case "802.1CB bridge" applies to the internal bridge component.>>

<<Commenter's Note (Rodney Cummings): Add the following subclauses to Annex A. I am using the table headings from the latest 802.1Q revision, since those provide a solid frame of reference. >>

A.2 PICS proforma for bridge implementations

A.2.1 Major capabilities

Item	Feature	Status	References	Support
BSEQ	Is the Sequencing sublayer implemented?	M	6.3	Yes [ ]
BSE	Is the Sequence Encapsulation sublayer implemented?	M	6.4	Yes [ ]
BSM	Is the Split/Merge sublayer implemented?	O	6.5	Yes [ ] No [ ]
BCE	Is the Circuit Encoding sublayer implemented?	M	6.x	Yes [ ]
MAX	State the number of circuits that can be supported on each port.	M		Number:

A.2.2 Sequencing

Item	Feature	Status	References	Support
BSEQ-1	Is the sequence generation sublayer implemented for every port?	M	6.3.2	Yes [ ]
BSEQ-2	Is the sequence recovery sublayer implemented?	M	6.3.3	Yes [ ]
BSEQ-3	Is the intermittent-streams algorithm implemented?	M	6.3.3.1.10, 6.3.3.2.1	Yes [ ]
BSEQ-4	For the intermittent-streams algorithm, is the default size for the sequence number 16 bits?	M	6.3.1.2	Yes [ ]
BSEQ-5	Is the bulk-streams algorithm implemented?	M	6.3.3.1.10, 6.3.3.2.2	Yes [ ]
BSEQ-6	For the bulk-streams algorithm, is the default size for the sequence number 8 bits?	M	6.3.1.2	Yes [ ]

A.2.3 Sequence Encapsulation

Item	Feature	Status	References	Support
BSE-1	Is the redundancy tag implemented?	M	6.4.1	Yes [ ]

A.2.4 Split/Merge

<<Commenter’s Note (Rodney Cummings): Do be determined… we may prohibit this (‘X’).

A.2.2 Circuit Encode

Item	Feature	Status	References	Support
BCE-1	Does operCircuitFormat support value 1 (DA/VID)?	M	6.x.1.1	Yes [ ]
BCE-2	Does operCircuitFormat support value 2 (SA/VID)?	M	6.x.1.1	Yes [ ]
BCE-3	Does operCircuitFormat support value 3 (SA/DA/VID)?	O	6.x.1.1	Yes [ ] No [ ]
BCE-3	Does operCircuitFormat support value 4 (IP octuple)?	O	6.x.1.1	Yes [ ] No [ ]