P802.1CBec — Sequence Generation and Vector Recovery — Pseudocode

Lisa Maile, TU/e Eindhoven University of Technology

July 13, 2026

This pseudocode summarizes the current discussion and is a first step towards detailing the new proposals for CBec. All aspects are currently under discussion and not final.

There are two sequence number spaces: the cyclic *normal* space $[0, GenSeqSpace - 1]$ and the linear *InitSeqNumSpace* $[InitSeqStart, GenSeqSpace - 1]$, used directly after a reset of the sequence generation function, with

$$InitSeqStart = GenSeqSpace - InitSeqSpaceLength,$$

$$InitSeqSpaceLength = ValidWindowLength + 2 \cdot frerSeqRcvyHistoryLength.$$

Packets in the InitSeqNumSpace carry $InitSeqNumFlag = 1$; the first *InitStartFlagLength* (to be discussed, but $<$ *frerSeqRcvyHistoryLength*) of them additionally carry $InitStartFlag = 1$.

The sequence recovery function maintains two recovery instances R, NORMAL and INIT. Each instance owns one full set of recovery variables, written $R.RecovSeqNum$, $R.SequenceHistory$, $R.TakeAny$, $R.SuppressLostUntilSeen$, $R.RemainingTicks$. For the NORMAL instance these are the variables defined in IEEE Std 802.1CB-2017 (7.4.3.2) plus the new *SuppressLostUntilSeen*; for the INIT instance they carry an "Init" prefix, i.e., *InitRecovSeqNum*, *InitSequenceHistory*, *InitTakeAny*, *InitSuppressLostUntilSeen*, *InitRemainingTicks*.

We will also need a new counter, to count both resets and timeouts separatly (*frerCpsSeqRcvyResets* plus *frerCpsSeqRcvyTimeouts*).

A few things not yet looked at: R-TAG encoding of the two new flag bits (Clause 7.8/7.9); interaction of the Latent error detection function (7.4.4) with two parallel recovery instances.

Sequence generation

SequenceGenerationReset is called whenever the BEGIN event occurs or the value True is written to the *frerSeqRcvyReset* managed object. If the InitSeqNumSpace is enabled, generation (re-)enters the linear space at *InitSeqStart* instead of restarting at 0; the legacy behavior is kept for implementations with the InitSeqNumSpace disabled.

We have to think what happens if *InitSeqNumSpaceEnabled* was *True* and is then changed to *False*.

Algorithm 1 SequenceGenerationReset — called on BEGIN or *frerSeqRcvyReset* = True

```
1: function SequenceGenerationReset( )
2:     if InitSeqNumSpaceEnabled then
3:         InitGenSeqNum ← InitSeqStart
4:         UseInitSeqNum ← true                    ▷ enter (or re-enter) the InitSeqNumSpace
5:     else
6:         GenSeqNum ← 0;  UseInitSeqNum ← false                    ▷ legacy behavior
7:     end if
8:     frerCpsSeqGenResets ← frerCpsSeqGenResets + 1
9: end function
```

SequenceGenerationAlgorithm is called whenever the DATA_REQUEST event occurs. Depending on *UseInitSeqNum*, the sequence_number is drawn from the linear InitSeqNumSpace (setting the packet flags accordingly) or from the normal cyclic space. When the linear space is exhausted, generation continues seamlessly at 0 in the normal space.

Algorithm 2 SequenceGenerationAlgorithm — called on DATA_REQUEST

```
1: function SequenceGenerationAlgorithm(packet)
2:     if UseInitSeqNum then
3:         packet.sequence_number ← InitGenSeqNum
4:         packet.InitSeqNumFlag ← 1
5:         packet.InitStartFlag ← 1 for the first InitStartFlagLength packets after reset, else 0
6:         if InitGenSeqNum = GenSeqSpace − 1 then          ▷ InitSeqNumSpace exhausted
7:             GenSeqNum ← 0;  UseInitSeqNum ← false
8:         else
9:             InitGenSeqNum ← InitGenSeqNum + 1
10:        end if
11:    else
12:        packet.sequence_number ← GenSeqNum
13:        packet.InitSeqNumFlag ← 0;  packet.InitStartFlag ← 0
14:        GenSeqNum ← (GenSeqNum + 1) mod GenSeqSpace
15:    end if
16:    SEND_DATA
17: end function
```

Sequence recovery

VectorRecoveryAlgorithm is called whenever the DATA_INDICATION event occurs. It selects the recovery instance based on the packet's *InitSeqNumFlag*: packets of the InitSeqNumSpace are handled by the Init instance, all other packets by the Normal instance. A packet carrying *InitStartFlag* while no Init instance is active indicates a reset of the sequence generation function and starts a new Init instance. A second reset through *InitStartFlag* is not detectable while *InitActive* = true. Update from discussions on 13 July 2026: We want to allow for a new reset once the new sequence number space moved "far enough" ahead (not yet in the code). Once the Init instance has reached the end of the linear space and the new sequence enters the normal cyclic space, the Normal instance is handed over to the new sequence.

Update from discussions on 13 July 2026: What happens when *InitSeqNumSpaceEnabled* is *false* is not yet sure. This also relates to the question which parts of the algorithm should become

optional. Currently, the code will completely ignore all flags. In the future, we want to use the flags to trigger the reset faster. What is not yet sure: Do we always switch to INIT if we receive such a flag, or do we just clear NORMAL?

Algorithm 3 VectorRecoveryAlgorithm — called on DATA_INDICATION

```
1:  function VectorRecoveryAlgorithm(packet)
2:      if packet.InitSeqNumFlag = 1 and InitSeqNumSpaceEnabled then
3:          if packet.InitStartFlag = 1 and InitActive = false then
4:              StartInitRecoveryInstance( )                    ▷ reset of SeqGen detected
5:          end if
6:          ApplyVectorRecovery(packet, Init)
7:          if InitActive and Init.RecovSeqNum = RecovSeqSpace − 1 then
8:              InitCompleted ← true                            ▷ linear space exhausted at receiver
9:          end if
10:     else                                  ▷ normal-space packet, or InitSeqNumSpace disabled
11:         if InitActive and InitCompleted then
12:             HandoverToNormalSpace( )
13:         end if
14:         ApplyVectorRecovery(packet, Normal)
15:     end if
16: end function
```

StartInitRecoveryInstance initializes the INIT instance after a detected reset of the sequence generation function. Since no packets exist before *InitSeqStart*, the instance is anchored at *InitSeqStart*− 1 with *TakeAny* disabled; *SuppressLostUntilSeen* prevents the freshly inserted 0s from being counted as losses. The NORMAL instance is left untouched, so the old (slow-path) sequence can be processed in parallel. *TakeAny* should be discussed. It seems to be not needed for operation (false), but could add resilience for temporary outages.

Algorithm 4 StartInitRecoveryInstance — reset of SeqGen detected

```
1:  function StartInitRecoveryInstance( )
2:      InitActive ← true;  InitCompleted ← false
3:      Init.RecovSeqNum ← InitSeqStart − 1   ▷ fixed anchor: no packets exist before InitSeqStart
4:      Init.SequenceHistory ← all 0
5:      Init.TakeAny ← false
6:      Init.SuppressLostUntilSeen ← true          ▷ artificial 0s must not be counted as losses
7:      restart Init.RemainingTicks
8:  end function
```

HandoverToNormalSpace is executed when the first packet of the new sequence arrives in the normal cyclic space after the INIT instance has completed. The state belonging to the old sequence is discarded and the NORMAL instance is re-anchored so that sequence_number 0 is expected next.

Currently, HANDOVERTONORMALSPACE discards the old NORMAL instance without counting the 0s remaining in its *SequenceHistory* as *frerCpsSeqRcvyLostPackets*. We have to decide whether these should be evaluated (analogous to the timeout behavior) or discarded silently, given that the old sequence was superseded by a generator reset?

We cannot set *InitActive* ← false here because there could still arrive packets from the INIT space. Update from discussions on 13 July 2026: We should use INIT.*RemainingTicks* for this (not

yet in the code).

Algorithm 5 HandoverToNormalSpace — new sequence enters the normal cyclic space

```
1: function HandoverToNormalSpace( )
2:     Discard the old state of the Normal instance.
3:     Normal.RecovSeqNum ← RecovSeqSpace − 1          ▷ next expected sequence_number is 0
4:     Normal.SequenceHistory ← all 0
5:     Normal.TakeAny ← false
6:     Normal.SuppressLostUntilSeen ← true      ▷ the new artificial 0s must not be counted as losses
7:     InitCompleted ← false
8: end function
```

SequenceRecoveryTimeout is called when $R.RemainingTicks$ reaches 0 for instance R, i.e., no packet has been accepted by that instance for *frerSeqRcvyResetMSec* milliseconds. So currently, the timeout value is shared by both instances, but we plan for individual R.RemainingTicks to trigger their R.TakeAny. In contrast to the current standard, a timeout no longer resets the instance: losses are evaluated immediately, the history is cleared, and *TakeAny* lets the next received packet re-anchor the instance.

Algorithm 6 SequenceRecoveryTimeout — called when $R.RemainingTicks$ reaches 0 for instance R

```
1: function SequenceRecoveryTimeout(R)
2:     if R.SuppressLostUntilSeen = false then
3:         frerCpsSeqRcvyLostPackets ← frerCpsSeqRcvyLostPackets + number of 0s in R.SequenceHistory
                                          ▷ losses are reported at timeout, not only when shifted out later
4:     end if
5:     R.SequenceHistory ← all 0
6:     R.TakeAny ← true          ▷ R.RecovSeqNum is kept; next packet re-anchors the instance
7:     R.SuppressLostUntilSeen ← true
8:     frerCpsSeqRcvyTimeouts ← frerCpsSeqRcvyTimeouts + 1
9: end function
```

SequenceRecoveryReset is called whenever the BEGIN event occurs or the value True is written to the *frerSeqRcvyReset* managed object (so not triggered by the reception of a flag). Both recovery instances are reset to the initial state with *SuppressLostUntilSeen* additionally set so that the fresh 0s are not counted as losses.

We might want to revent processing of packets for a while after this has been triggered.

Algorithm 7 SequenceRecoveryReset — called on BEGIN or *frerSeqRcvyReset* = True (*not* on timeout)

```
1: function SequenceRecoveryReset( )
2:     InitActive ← false;  InitCompleted ← false
3:     for R ∈ {Normal, Init} do
4:         R.RecovSeqNum ← RecovSeqSpace − 1
5:         R.SequenceHistory ← all 0
6:         R.TakeAny ← true;  R.SuppressLostUntilSeen ← true
7:     end for
8:     frerCpsSeqRcvyResets ← frerCpsSeqRcvyResets + 1
9: end function
```

ApplyVectorRecovery applies the Vector Recovery rules of IEEE Std 802.1CB-2017 (7.4.3.4) to a packet using the variables of the selected instance R. Note that the valid window (*ValidWindowLength*) may be longer than the history (*frerSeqRcvyHistoryLength*) (this was a new proposal).

Algorithm 8 ApplyVectorRecovery — Vector Recovery rules on instance R

```
1: function ApplyVectorRecovery(packet, R)
2:     s ← packet.sequence_number
3:     if R.TakeAny then                          ▷ first packet after timeout or SeqRcvy reset
4:         R.TakeAny ← false
5:         R.SequenceHistory[0] ← 1;  R.RecovSeqNum ← s
6:         Accept(packet, R)
7:         return
8:     end if
9:     d ← offset(s, R.RecovSeqNum)               ▷ signed distance, modulo RecovSeqSpace
10:    if −frerSeqRcvyHistoryLength < d ≤ 0 and R.SequenceHistory[−d] = 0 then
11:        R.SequenceHistory[−d] ← 1                          ▷ late packet, not yet seen
12:        Accept(packet, R)
13:    else if −frerSeqRcvyHistoryLength < d ≤ 0 then
14:        Discard packet as duplicate.
15:    else if 0 < d ≤ ValidWindowLength then     ▷ packet ahead, within the valid window
16:        ShiftSequenceHistory(R, 0)  (d − 1) times
17:        ShiftSequenceHistory(R, 1)
18:        R.RecovSeqNum ← s
19:        Accept(packet, R)
20:    else
21:        Discard packet as rogue (outside the valid window).
22:    end if
23: end function

24: function Accept(packet, R)
25:    restart R.RemainingTicks; update passed-packet counters; PRESENT_DATA
26: end function
```

ShiftSequenceHistory advances *R.SequenceHistory* by one position, as in IEEE Std 802.1CB-2017 (7.4.3.6), but with loss counting made aware of artificial 0s: while *R.SuppressLostUntilSeen* is

set, 0s that were inserted by a timeout, a reset, or a handover are shifted out without being counted. Once a real packet (a 1) has traversed the whole history, normal loss counting resumes.

Algorithm 9 ShiftSequenceHistory — loss counting aware of artificial 0s

```
1: function ShiftSequenceHistory(R, newZeroValue)
2:     if R.SuppressLostUntilSeen then
3:         if R.SequenceHistory[frerSeqRcvyHistoryLength − 1] = 1 then
4:             R.SuppressLostUntilSeen ← false        ▷ a real packet has traversed the whole history
5:         end if
                   ▷ 0s shifted out here are artificial (timeout/reset/handover) and are not counted
6:     else if R.SequenceHistory[frerSeqRcvyHistoryLength − 1] = 0 then
7:         frerCpsSeqRcvyLostPackets ← frerCpsSeqRcvyLostPackets + 1      ▷ normal loss counting
8:     end if
9:     shift(R.SequenceHistory); R.SequenceHistory[0] ← newZeroValue
10: end function
```