IEEE 802.3ae™-2002
(Amendment to IEEE Std 802.3™2002)

Corrections to

IEEE Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements

Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications

Amendment: Media Access Control (MAC) Parameters, Physical Layers, and Management Parameters for 10 Gb/s Operation

Sponsor

LAN/MAN Standards Committee

of the

IEEE Computer Society

Correction Sheet
Issued 20 July 2004

4.2.7 – 4.2.9 is incorrect and should be replaced with the following 4.2.7 – 4.2.9:

4.2.7 Global declarations

Change 4.2.7 and its related subclauses to read as follows:

This subclause provides detailed formal specifications for the CSMA/CD MAC sublayer. It is a specification of generic features and parameters to be used in systems implementing this media access method. Subclause 4.4 provides values for these sets of parameters for recommended implementations of this media access mechanism.

4.2.7.1 Common constants, types, and variables

The following declarations of constants, types and variables are used by the frame transmission and reception sections of each CSMA/CD sublayer:

const
~~addressSize = ... ; {48 bits in compliance with 3.2.3)~~
addressSize = 48; {In bits, in compliance with 3.2.3}
lengthOrTypeSize = 16; {~~i~~In bits}
~~clientDataSize = ...; {in bits, MAC client Data, see 4.2.2.2, (1)(c)}~~
clientDataSize = ...; {In bits, size of MAC client data; see 4.2.2.2, a) 3)}
padSize = ...; {~~i~~In bits, = max (0, minFrameSize ─ (2 x addressSize + lengthOrTypeSize + clientDataSize + crcSize))}
dataSize = ...; {~~i~~In bits, = clientDataSize + padSize}
crcSize = 32; {In bits, 32-bit CRC~~= 4 octets~~}
~~frameSize = ...; {in bits, = 2 x addressSize + lengthOrTypeSize + dataSize + crcSize, see 4.2.2.2, (1)}~~
frameSize = ...; {In bits, = 2 x addressSize + lengthOrTypeSize + dataSize + crcSize; see 4.2.2.2, a)}
minFrameSize = ...~~-~~; {~~i~~In bits, implementation-dependent, see 4.4}
maxUntaggedFrameSize = ...~~-~~; {~~i~~In octets, implementation-dependent, see 4.4}
qTagPrefixSize = 4; {~~i~~In octets, length of QTag Prefix, see 3.5}
extend = ...; {Boolean, true if (slotTime ─ minFrameSize) > 0, false otherwise}
extensionBit = ...; {~~a~~A nondata value which is used for carrier extension and interframe during bursts}
extensionErrorBit = ...; {~~a~~A nondata value which is used to jam during carrier extension}
minTypeValue = 1536; {~~m~~Minimum value of the Length/Type field for Type interpretation}
maxValidFrame = maxUntaggedFrameSize ─ (2 x addressSize + lengthOrTypeSize + crcSize) / 8; {~~i~~In octets, the maximum length of the MAC client data field. This constant is defined for editorial convenience, as a function of other constants}
slotTime = ...~~-~~; {In bit times, unit of time for collision handling, implementation-dependent, see 4.4}
~~preambleSize = ...; {56 bits, see 4.2.5}~~
preambleSize = 56; {In bits, see 4.2.5}
sfdSize = 8; {~~8 bit~~In bits, start frame delimiter}
~~headerSize = ...; {64 bits, sum of preambleSize and sfdSize}~~
headerSize = 64; {In bits, sum of preambleSize and sfdSize}
type
Bit = ~~0..1~~(0, 1);
PhysicalBit = ~~0,1, extensionBit, extensionErrorBit~~(0, 1, extensionBit, extensionErrorBit); {~~b~~Bits transmitted to the Physical Layer can be either 0, 1, extensionBit or extensionErrorBit. Bits received from the Physical Layer can be either 0, 1 or extensionBit~~.~~}
AddressValue = *array* [1..addressSize] *of* Bit;
LengthOrTypeValue = *array* [1..lengthOrTypeSize] *of* Bit;

```
    DataValue = array [1..dataSize] of Bit; {Contains the portion of the frame that starts with the first bit
                                            following the Length/Type field and ends with the last bit
                                            prior to the FCS field. For VLAN Tagged frames, this value
                                            includes the Tag Control Information field and the original
                                            MAC client Length/Type field. See 3.5}
    CRCValue = array [1..crcSize] of Bit;
    PreambleValue = array [1..preambleSize] of Bit;
    SfdValue = array [1..sfdSize] of Bit;
    ViewPoint = (fields, bits); {Two ways to view the contents of a frame}
    HeaderViewPoint = (headerFields, headerBits);
    Frame = record {Format of Media Access frame}
      case view: ViewPoint of
        fields: (
          destinationField: AddressValue;
          sourceField: AddressValue;
          lengthOrTypeField: LengthOrTypeValue;
          dataField: DataValue;
          fcsField: CRCValue);
        bits: (contents: array [1..frameSize] of Bit)
      end; {Frame}

    Header = record {Format of preamble and start frame delimiter}
      case headerView-: HeaderViewPoint of
        headerFields-: (
          preamble-: PreambleValue;
          sfd-: SfdValue);
        headerBits : (
          headerContents : array [1..headerSize] of Bit)
        headerBits: (headerContents: array [1..headerSize] of Bit)
      end; {dDefines header for MAC frame}
var
    halfDuplex: Boolean; {Indicates the desired mode of operation. halfDuplex is a static variable; its value
                          shall only be changed by the invocation of the Initialize procedure}
                          its value does not change between invocations of the Initialize procedure}
```

4.2.7.2 Transmit state variables

The following items are specific to frame transmission. (See also 4.4.)

```
const
    interFrameSpacing = ...-; {In bit times, minimum time gap between frames. Equal to interFrameGap,
                               see 4.4}
    interFrameSpacingPart1 = ...-; {In bit times, duration of the first portion of interFrameSpacing. In the
                                    range of 0 up to 2/3 of interFrameSpacing}
    interFrameSpacingPart2 = ...-; {In bit times, duration of the remainder of interFrameSpacing. Equal to
                                    interFrameSpacing – interFrameSpacingPart1}
    interFrameSize = ... ; {in bits, length of interframe fill during a burst. Equal to interFrameGap
                            divided by the bit period}
    ifsStretchRatio = ...; {In bits, determines the number of bits in a frame that require one octet of
                            interFrameSpacing extension, when ifsStretchMode is enabled; implementation
                            dependent, see 4.4}
    attemptLimit = ...-; {Max number of times to attempt transmission}
    backOffLimit = ...-; {Limit on number of times to back off}
```

burstLimit= ...~~-~~; {~~in bits: Limit~~In bits, limit for initiation of frame transmission in Burst Mode, implementation dependent, see 4.4}
jamSize = ...~~-~~; {~~in bits:~~In bits, the value depends upon medium and collision detect implementation}
var
outgoingFrame: Frame; {The frame to be transmitted}
outgoingHeader: Header;
currentTransmitBit, lastTransmitBit: 1..frameSize; {Positions of current and last outgoing bits in outgoingFrame}
lastHeaderBit: 1..headerSize;
deferring: Boolean; {Implies any pending transmission must wait for the medium to clear}
frameWaiting: Boolean; {Indicates that outgoingFrame is deferring}
attempts: 0..attemptLimit; {Number of transmission attempts on outgoingFrame}
newCollision: Boolean; {Indicates that a collision has occurred but has not yet been jammed}
transmitSucceeding: Boolean; {Running indicator of whether transmission is succeeding}
burstMode: Boolean~~:~~; {Indicates the desired mode of operation, and enables the transmission of multiple frames in a single carrier event. burstMode is a static variable; its ~~value does not change between invocations of the Initialize procedure.}~~ value shall only be changed by the invocation of the Initialize procedure}
bursting: Boolean; {In burstMode, the given station has acquired the medium and the burst timer has not yet expired}
burstStart: Boolean; {In burstMode, indicates that the first frame transmission is in progress}
extendError: Boolean; {Indicates a collision occurred while sending extension bits}
ifsStretchMode: Boolean; {Indicates the desired mode of operation, and enables the lowering of the average data rate of the MAC sublayer (with frame granularity), using extension of the minimum interFrameSpacing. ifsStretchMode is a static variable; its value shall only be changed by the invocation of the Initialize procedure}
ifsStretchCount: 0..ifsStretchRatio; {In bits, a running counter that counts the number of bits during a frame's transmission that are to be considered for the minimum interFrameSpacing extension, while operating in ifsStretchMode}
ifsStretchSize: 0..(((maxUntaggedFrameSize + qTagPrefixSize) x 8 + headerSize + interFrameSpacing + ifsStretchRatio – 1) *div* ifsStretchRatio);
{In octets, a running counter that counts the integer number of octets that are to be added to the minimum interFrameSpacing, while operating in ifsStretchMode}

4.2.7.3 Receive state variables

The following items are specific to frame reception. (See also 4.4.)

var
incomingFrame: Frame; {The frame being received}
receiving: Boolean; {Indicates that a frame reception is in progress}
excessBits: 0..7; {Count of excess trailing bits beyond octet boundary}
receiveSucceeding: Boolean; {Running indicator of whether reception is succeeding}
validLength: Boolean; {Indicator of whether received frame has a length error}
exceedsMaxLength: Boolean; {Indicator of whether received frame has a length longer than the maximum permitted length}
extending: Boolean; {Indicates whether the current frame is subject to carrier extension}
extensionOK: Boolean; {Indicates whether any bit errors were found in the extension part of a frame, which is not checked by the CRC}
passReceiveFCSMode: Boolean; {Indicates the desired mode of operation, and enables passing of the frame check sequence field of all received frames from the MAC sublayer to the MAC client. passReceiveFCSMode is a static variable}

4.2.7.4 Summary of interlayer interfaces

a) The interface to the MAC client, defined in 4.3.2, is summarized below:

```
type
    TransmitStatus = (transmitDisabled, transmitOK, excessiveCollisionError, lateCollisionErrorStatus);
                     {Result of TransmitFrame operation, reporting of lateCollisionErrorStatus is
                     optional for MACs operating at speeds at or below 100Mb/s}
    ReceiveStatus = (receiveDisabled, receiveOK, frameTooLong, frameCheckError, lengthError,
                     alignmentError); {Result of ReceiveFrame operation}
function TransmitFrame (
    destinationParam: AddressValue;
    sourceParam: AddressValue;
    lengthOrTypeParam: LengthOrTypeValue;
    ~~dataParam: DataValue): TransmitStatus;  {Transmits one frame}~~
    dataParam: DataValue;
    fcsParamValue: CRCValue;
    fcsParamPresent: Bit): TransmitStatus;  {Transmits one frame}
function ReceiveFrame (
    var destinationParam: AddressValue;
    var sourceParam: AddressValue;
    var lengthOrTypeParam: LengthOrTypeValue;
    ~~var dataParam: DataValue): ReceiveStatus; {Receives one frame}~~
    var dataParam: DataValue;
    var fcsParamValue: CRCValue;
    var fcsParamPresent: Bit): ReceiveStatus; {Receives one frame}
```

b) The interface to the Physical Layer, defined in 4.3.3, is summarized in the following:

```
var
    receiveDataValid: Boolean; {Indicates incoming bits}
    carrierSense: Boolean; {In half duplex mode, indicates that transmission should defer}
    transmitting: Boolean; {Indicates outgoing bits}
    collisionDetect: Boolean; {Indicates medium contention}
procedure TransmitBit (bitParam: PhysicalBit); {Transmits one bit}
function ReceiveBit: PhysicalBit; {Receives one bit}
procedure Wait (bitTimes: integer); {Waits for indicated number of bit times}
```

4.2.7.5 State variable initialization

The procedure Initialize must be run when the MAC sublayer begins operation, before any of the processes begin execution. Initialize sets certain crucial shared state variables to their initial values. (All other global variables are appropriately reinitialized before each use.) Initialize then waits for the medium to be idle, and starts operation of the various processes.

NOTE—Care should be taken to ensure that the time from the completion of the Initialize process to when the first packet transmission begins is at least an interFrameGap.

If Layer Management is implemented, the Initialize procedure shall only be called as the result of the initializeMAC action (30.3.1.2.1).

```
procedure Initialize;
~~begin~~begin
    frameWaiting := false;
```

```
    deferring := false;
    newCollision := false;
    transmitting := false; {~~t~~An interface to Physical Layer; see below}
    receiving := false;
    halfDuplex := ...; {~~t~~True for half duplex operation, false for full duplex operation. For operation at
                        speeds above 1000 Mb/s, halfDuplex shall always be false}~~halfDuplex is a static
                        variable; its value does not change between invocations of the Initialize~~
    bursting := false;
    burstMode := ...; {~~t~~True for half duplex operation at~~ speeds above 100 Mb/s~~ an operating speed of 1000
                        Mb/s, when multiple frames' transmission in a single carrier event is desired and
                        supported, false otherwise}~~. burstMode is a static variable; its value does not change
                        between invocations of the Initialize procedure}~~
    extending := extend ~~and~~and halfDuplex;
    ifsStretchMode := ...; {True for operating speeds above 1000 Mb/s when lowering the average data rate
                            of the MAC sublayer (with frame granularity) is desired and supported, false
                            otherwise}
    ifsStretchCount := 0;
    ifsStretchSize := 0;
    passReceiveFCSMode := ...; {True when enabling the passing of the frame check sequence of all
                                received frames from the MAC sublayer to the MAC client is desired and
                                supported, false otherwise}
    ~~while~~ carrierSense ~~or~~ receiveDataValid ~~do~~ nothing~~
    if halfDuplex then while carrierSense or receiveDataValid do nothing
    else while receiveDataValid do nothing
    {Start execution of all processes}
end; {Initialize}
```

4.2.8 Frame transmission

Change subclause 4.2.8 to read as follows:

The algorithms in this subclause define MAC sublayer frame transmission. The function TransmitFrame implements the frame transmission operation provided to the MAC client:

```
function TransmitFrame (
    destinationParam: AddressValue;
    sourceParam: AddressValue;
    lengthOrTypeParam: LengthOrTypeValue;
    ~~dataParam: DataValue): TransmitStatus;~~
    dataParam: DataValue;
    fcsParamValue: CRCValue;
    fcsParamPresent: Bit): TransmitStatus;
procedure TransmitDataEncap; ~~... {n~~{Nested procedure; see body below}
begin
    if transmitEnabled then
        begin
            TransmitDataEncap;
            TransmitFrame := TransmitLinkMgmt
        end
    else TransmitFrame := transmitDisabled
end; {TransmitFrame}
```

If transmission is enabled, TransmitFrame calls the internal procedure TransmitDataEncap to construct the frame. Next, TransmitLinkMgmt is called to perform the actual transmission. The TransmitStatus returned indicates the success or failure of the transmission attempt.

TransmitDataEncap builds the frame and places the 32-bit CRC in the frame check sequence field:

```
procedure TransmitDataEncap;
begin
   with outgoingFrame do
      begin {~~a~~Assemble frame}
         view := fields;
         destinationField := destinationParam;
         sourceField := sourceParam;
         lengthOrTypeField := lengthOrTypeParam;
         ~~dataField := ComputePad (dataParam);~~
         ~~fcsField := CRC32(outgoingFrame);~~
         if fcsParamPresent then
            begin
               dataField := dataParam; {No need to generate pad if the FCS is passed from MAC client}
               fcsField := fcsParamValue {Use the FCS passed from MAC client}
            end
         else
            begin
               dataField := ComputePad(dataParam);
               fcsField := CRC32(outgoingFrame)
            end;
         view := bits
      end {~~a~~Assemble frame}
   with outgoingHeader do
      begin
         headerView := headerFields;
         preamble := ...; {* '1010...10,' LSB to MSB*}
         sfd := ...; {* '10101011,' LSB to MSB*}
         headerView := headerBits
      end
end; {TransmitDataEncap}
```

If the MAC client chooses to generate the frame check sequence field for the frame, it passes this field to the MAC sublayer via the fcsParamValue parameter. If the fcsParamPresent parameter is true, TransmitDataEncap uses the fcsParamValue parameter as the frame check sequence field for the frame. Such a frame shall not require any padding, since it is the responsibility of the MAC client to ensure that the frame meets the minFrameSize constraint. If the fcsParamPresent parameter is false, the fcsParamValue parameter is unspecified.TransmitDataEncap first calls the ComputePad function, followed by a call to the CRC32 function to generate the padding (if necessary) and the frame check sequence field for the frame internally to the MAC sublayer.

ComputePad appends an array of arbitrary bits to the MAC client data to pad the frame to the minimum frame size~~.~~:

```
~~function ComputePad(~~
      ~~var dataParam: DataValue): DataValue;~~
~~begin~~
   ~~ComputePad := {Append an array of size padSize of arbitrary bits to the MAC client dataField}~~
~~end;~~ ~~{ComputePadParam}~~

function ComputePad(var dataParam: DataValue): DataValue;
begin
   ComputePad := {Append an array of size padSize of arbitrary bits to the MAC client dataField}
end; {ComputePad}
```

TransmitLinkMgmt attempts to transmit the frame. In half duplex mode, it first defers to any passing traffic. In half duplex mode, if a collision occurs, transmission is terminated properly and retransmission is scheduled following a suitable backoff interval:

```
function TransmitLinkMgmt: TransmitStatus;
begin
   attempts := 0;
   transmitSucceeding := false;
   lateCollisionCount := 0;
   deferred := false; {~~i~~Initialize}
   excessDefer := false;
   while  (attempts < attemptLimit) and (not transmitSucceeding)
                                          and (not extend or lateCollisionCount = 0) do
                                  {~~n~~No retransmission after late collision if operating at ~~> 100~~ 1000 Mb/s}
      begin {~~l~~Loop}
         if bursting then {~~t~~This is a burst continuation}
            frameWaiting := true {~~s~~Start transmission without checking deference}
         else {~~n~~Non bursting case, or first frame of a burst}
            begin
               if attempts > 0 then BackOff;
               ~~if halfDuplex then frameWaiting := true;~~
               frameWaiting := true;
               while deferring do {~~d~~Defer to passing frame, if any[1]}
                  if halfDuplex then deferred := true;
                  ~~begin~~
                     ~~nothing;~~
‡                    ~~if halfDuplex then deferred := true~~
                  ~~end;~~
               burstStart := true;
               if burstMode then bursting := true
            end;
         lateCollisionError := false;
         StartTransmit;
         frameWaiting := false;
         if halfDuplex then
         begin
            ~~frameWaiting := false;~~
```

1. The Deference process ensures that the reception of traffic does not cause deferring to be true when in full duplex mode. Deferring is used in full duplex mode to enforce the minimum interpacket gap spacing.

```
            while transmitting do WatchForCollision;
            if lateCollisionError then lateCollisionCount := lateCollisionCount + 1;
               ~~lateCollisionCount := lateCollisionCount + 1;~~
            attempts := attempts + 1
         end {~~h~~Half duplex mode}
         else while transmitting do nothing {~~f~~Full duplex mode}
      end; {~~l~~Loop}
   LayerMgmtTransmitCounters; {~~u~~Update transmit and transmit error counters in 5.2.4.2}
   if transmitSucceeding then
      begin
         if burstMode then burstStart := false; {Can't be the first frame anymore}
         TransmitLinkMgmt := transmitOK
      end
   else if (extend and lateCollisionCount > 0) then TransmitLinkMgmt := lateCollisionErrorStatus;
      ~~TransmitLinkMgmt := lateCollisionErrorStatus;~~
   else  TransmitLinkMgmt := excessiveCollisionError
end; {TransmitLinkMgmt}
```

Each time a frame transmission attempt is initiated, StartTransmit is called to alert the BitTransmitter process that bit transmission should begin:

```
procedure StartTransmit;
begin
   currentTransmitBit := 1;
   lastTransmitBit := frameSize;
   transmitSucceeding := true;
   transmitting := true;
   lastHeaderBit~~: =~~ := headerSize
end; {StartTransmit}
```

In half duplex mode, TransmitLinkMgmt monitors the medium for contention by repeatedly calling WatchForCollision, once frame transmission has been initiated:

```
procedure WatchForCollision;
begin
   if transmitSucceeding and collisionDetect then
      begin
         if currentTransmitBit > (slotTime – headerSize) then lateCollisionError := true;
            ~~lateCollisionError := true;~~
         newCollision := true;
         transmitSucceeding := false;
         if burstMode then
         begin
            bursting := false;
            if not burstStart then
               lateCollisionError := true {Every collision is late, unless it hits the first frame in a burst}
         end
      end
end; {WatchForCollision}
```

WatchForCollision, upon detecting a collision, updates newCollision to ensure proper jamming by the BitTransmitter process. The current transmit bit number is checked to see if this is a late collision. If the collision occurs later than a collision window of slotTime bits into the packet, it is considered as evidence of a late collision. The point at which the collision is received is determined by the network media propagation time and the delay time through a station and, as such, is implementation-dependent (see 4.1.2.2). While operating at speeds of 100 Mb/s or lower, an implementation may optionally elect to end retransmission attempts after a late collision is detected. While operating at ~~speeds above 100 Mb/s~~ the speed of 1000 Mb/s, an implementation shall end retransmission attempts after a late collision is detected.

After transmission of the jam has been completed, if TransmitLinkMgmt determines that another attempt should be made, BackOff is called to schedule the next attempt to retransmit the frame.

```
function Random (low, high: integer): integer;
begin
   Random := ...{~~u~~Uniformly distributed random integer r, such that low ≤ r < high}
end; {Random}
```

BackOff performs the truncated binary exponential backoff computation and then waits for the selected multiple of the slot time~~.~~:

```
var maxBackOff: 2..1024; {Working variable of BackOff}
procedure BackOff;
begin
   if attempts = 1 then maxBackOff := 2
   else if attempts ≤ backOffLimit then maxBackOff := maxBackOff x 2;
   Wait(slotTime x Random(0, maxBackOff))
end; {BackOff}
```

BurstTimer is a process that does nothing unless the bursting variable is true. When bursting is true, BurstTimer increments burstCounter until the burstLimit limit is reached, whereupon BurstTimer assigns the value false to bursting~~.~~:

```
~~process BurstTimer;~~
   ~~var burstCounter: integer;~~
~~begin~~
   ~~cycle~~
      ~~while not bursting do nothing; {wait for a burst}~~
      ~~burstCounter := 0;~~
      ~~while bursting and (burstCounter < burstLimit) do~~
         ~~begin~~
            ~~Wait(1);~~
            ~~burstCounter := burstCounter + 1~~
         ~~end;~~
      ~~bursting := false~~
   ~~end {burstMode cycle}~~
~~end; {BurstTimer}~~

process BurstTimer;
begin
   cycle
      while not bursting do nothing; {Wait for a burst}
      Wait(burstLimit);
      bursting := false
   end {burstMode cycle}
end; {BurstTimer}
```

The Deference process runs asynchronously to continuously compute the proper value for the variable deferring. In the case of half duplex burst mode, deferring remains true throughout the entire burst. Interframe spacing may be used to lower the average data rate of a MAC at operating speeds above 1000 Mb/s in the full duplex mode, when it is necessary to adapt it to the data rate of a WAN-based physical layer. When interframe stretching is enabled, deferring remains true throughout the entire extended interframe gap, which includes the sum of interFrameSpacing and the interframe extension as determined by the BitTransmitter:

```
process Deference;
   var realTimeCounter: integer; wasTransmitting: Boolean;
begin
   if halfDuplex then cycle{hHalf duplex loop}
      while not carrierSense do nothing; {wWatch for carrier to appear}
      deferring := true;  {dDelay start of new transmissions}
      wasTransmitting := transmitting;
      while carrierSense or transmitting do wasTransmitting := wasTransmitting or transmitting;
         wasTransmitting: = wasTransmitting or transmitting;
      if wasTransmitting then Wait(interFrameSpacingPart1) {Time out first part of interframe gap}
         begin
            StartRealTimeDelay; {time out first part interframe gap}
            while RealTimeDelay(interFrameSpacingPart1) do nothing
         end
      else
         begin
            StartRealTimeDelay;
            repeat
            while carrierSense do StartRealTimeDelay
            until not RealTimeDelay(interFrameSpacingPart1)
            realTimeCounter := interFrameSpacingPart1;
            repeat
               while carrierSense do realTimeCounter := interFrameSpacingPart1;
               Wait(1);
               realTimeCounter := realTimeCounter – 1
            until (realTimeCounter = 0)
         end;
      StartRealTimeDelay; {time out second part interframe gap}
      while RealTimeDelay(interFrameSpacingPart2) do nothing;
      Wait(interFrameSpacingPart2); {Time out second part of interframe gap}
      deferring: = := false; {aAllow new transmissions to proceed}
      while frameWaiting do nothing {aAllow waiting transmission, if any}
   end {hHalf duplex loop}
   else cycle {fFull duplex loop}
      while not transmitting do nothing; {wWait for the start of a transmission}
      deferring := true; {iInhibit future transmissions}
      while transmitting do nothing; {wWait for the end of the current transmission}
      StartRealTimeDelay; {time out an interframe gap}
      while RealTimeDelay(interFrameSpacing) do nothing;
      Wait(interFrameSpacing + ifsStretchSize x 8); {Time out entire interframe gap and IFS extension}
      if not frameWaiting then {Don't roll over the remainder into the next frame}
         begin
            Wait(8);
            ifsStretchCount := 0
```

```
            end
        deferring := false {dDon't inhibit transmission}
    end {fFull duplex loop}
end; {Deference}
```

If the ifsStretchMode is enabled, the Deference process continues to enforce interframe spacing for an additional number of bit times, after the completion of timing the interFrameSpacing. The additional number of bit times is reflected by the variable ifsStretchSize. If the variable ifsStretchCount is less than ifsStretchRatio and the next frame is ready for transmission (variable frameWaiting is true), the Deference process enforces interframe spacing only for the integer number of octets, as indicated by ifsStretchSize, and saves ifsStretchCount for the next frame's transmission. If the next frame is not ready for transmission (variable frameWaiting is false), then the Deference process initializes the ifsStretchCount variable to zero.

```
procedure StartRealTimeDelay
    begin
        {reset the realtime timer and start it timing}
    end; {StartRealTimeDelay}

function RealTimeDelay (µsec:real): Boolean;
    begin
        {return the value true if the specified number of microseconds have
        not elapsed since the most recent invocation of StartRealTimeDelay,
        otherwise return the value false}
    end; {RealTimeDelay}
```

The BitTransmitter process runs asynchronously, transmitting bits at a rate determined by the Physical Layer's TransmitBit operation:

```
process BitTransmitter;
begin
    cycle {oOuter loop}
        if transmitting then
            begin {iInner loop}
                extendError := false;
                if ifsStretchMode then {Calculate the counter values}
                    begin
                        ifsStretchSize := (ifsStretchCount + headerSize + frameSize + interFrameSpacing) div
                                                                    ifsStretchRatio; {Extension of the interframe spacing}
                        ifsStretchCount := (ifsStretchCount + headerSize + frameSize + interFrameSpacing)
                                mod ifsStretchRatio {Remainder to carry over into the next frame's transmission}
                    end;
                PhysicalSignalEncap;  {Send preamble and start of frame delimiter}
                while transmitting do
                    begin
                        if (currentTransmitBit > lastTransmitBit) then TransmitBit(extensionBit)
                        else if extendError then TransmitBit(extensionErrorBit) {Jam in extension}
                            if extendError then
                                TransmitBit(extensionErrorBit) {jam in extension}
                        else TransmitBit(outgoingFrame[currentTransmitBit]);
                                TransmitBit(outgoingFrame[currentTransmitBit]);
                        if newCollision then StartJam else NextBit
                    end;
                if bursting then
                    begin
```

```
                    InterFrameSignal;
                    if extendError then
                        if transmitting then transmitting := false~~ {TransmitFrame may have been
                                                                called during InterFrameSignal}~~
                                                {TransmitFrame may have been called during InterFrameSignal}
                        else IncLargeCounter(lateCollision);
                                                {Count late collisions which were missed by TransmitLinkMgmt}
                            ~~IncLargeCounter(lateCollision); {count late collisions which
                                    were missed by TransmitLinkMgmt}~~
                        bursting := bursting and (frameWaiting ~~or~~or transmitting)
                    end
                end {~~i~~Inner loop}
            end {~~o~~Outer loop}
        end; {BitTransmitter}
```

The bits transmitted to the physical layer can take one of four values: data zero (0), data one (1), extensionBit (EXTEND), or extensionErrorBit (EXTEND_ERROR). The values extensionBit and extensionErrorBit are not transmitted between the first preamble bit of a frame and the last data bit of a frame under any circumstances. The BitTransmitter calls the procedure TransmitBit with bitParam = extensionBit only when it is necessary to perform carrier extension on a frame after all of the data bits of a frame have been transmitted. The BitTransmitter calls the procedure TransmitBit with bitParam = extensionErrorBit only when it is necessary to jam during carrier extension.

```
    procedure PhysicalSignalEncap;
    begin
        while currentTransmitBit ≤ lastHeaderBit do
            begin
                TransmitBit(outgoingHeader[currentTransmitBit]); {~~t~~Transmit header one bit at a time}
                currentTransmitBit := currentTransmitBit + 1
            end;
        if newCollision then StartJam else currentTransmitBit := 1
        ~~currentTransmitBit := 1~~
    end; {PhysicalSignalEncap}
```

The procedure InterFrameSignal fills the interframe interval between the frames of a burst with extensionBits. InterFrameSignal also monitors the variable collisionDetect during the interframe interval between the frames of a burst, and will end a burst if a collision occurs during the interframe interval. The procedural model is defined such that a MAC operating in the burstMode will emit an extraneous sequence of interFrameSize extensionBits in the event that there are no additional frames ready for transmission after InterFrameSignal returns. Implementations may be able to avoid sending this extraneous sequence of extensionBits if they have access to information (such as the occupancy of a transmit queue) that is not assumed to be available to the procedural model.

```
    procedure InterFrameSignal;
        var interFrameCount, interFrameTotal-: integer;
    begin
        interFrameCount := 0;
        ~~interFrameTotal := interFrameSize;~~
        interFrameTotal := interFrameSpacing;
        while interFrameCount < interFrameTotal do
            begin
                if not extendError then TransmitBit(extensionBit)
                    ~~TransmitBit(extensionBit)~~
                else TransmitBit(extensionErrorBit);
                    ~~TransmitBit(extensionErrorBit);~~
                interFrameCount := interFrameCount + 1;
```

```
            if collisionDetect and not extendError then
            begin
                bursting := false;
                extendError := true;
                interFrameCount := 0;
                interFrameTotal := jamSize
            end
        end
    end; {InterFrameSignal}

    procedure NextBit;
    begin
        currentTransmitBit := currentTransmitBit + 1;
        if halfDuplex and burstStart and transmitSucceeding then {eCarrier extension may be required}
            transmitting := (currentTransmitBit ≤ max(lastTransmitBit, slotTime))
        else transmitting := (currentTransmitBit ≤ lastTransmitBit)
            transmitting := (currentTransmitBit ≤ lastTransmitBit)
    end; {NextBit}

    procedure StartJam;
    begin
        extendError := currentTransmitBit > lastTransmitBit;
        currentTransmitBit := 1;
        lastTransmitBit := jamSize;
        newCollision := false
    end; {StartJam}
```

BitTransmitter, upon detecting a new collision, immediately enforces it by calling StartJam to initiate the transmission of the jam. The jam should contain a sufficient number of bits of arbitrary data so that it is assured that both communicating stations detect the collision. (StartJam uses the first set of bits of the frame up to jamSize, merely to simplify this program.)

4.2.9 Frame reception

Change subclause 4.2.9 to read as follows:

The algorithms in this subclause define CSMA/CD Media Access sublayer frame reception.

The function ReceiveFrame implements the frame reception operation provided to the MAC client:

```
    function ReceiveFrame (
        var destinationParam: AddressValue;
        var sourceParam: AddressValue;
        var lengthOrTypeParam: LengthOrTypeValue;
        var dataParam: DataValue): ReceiveStatus;
        var dataParam: DataValue;
        var fcsParamValue: CRCValue;
        var fcsParamPresent: Bit): ReceiveStatus;
    function ReceiveDataDecap: ReceiveStatus; ... {nNested function; see body below}
    begin
        if receiveEnabled then
            repeat
                ReceiveLinkMgmt;
                ReceiveFrame := ReceiveDataDecap;
```

```
        until receiveSucceeding
      else ReceiveFrame := receiveDisabled
        ~~ReceiveFrame := receiveDisabled~~
  end; {ReceiveFrame}
```

If enabled, ReceiveFrame calls ReceiveLinkMgmt to receive the next valid frame, and then calls the internal function ReceiveDataDecap to return the frame's fields to the MAC client if the frame's address indicates that it should do so. The returned ReceiveStatus indicates the presence or absence of detected transmission errors in the frame.

```
  function ReceiveDataDecap: ReceiveStatus;
‡   var status-: ReceiveStatus; {~~h~~Holds receive status information}
  begin
‡   with incomingFrame do
‡     begin
‡       view := fields;
        ~~receiveSucceeding := RecognizeAddress (incomingFrame, destinationField);~~
‡       receiveSucceeding := LayerMgmtRecognizeAddress-(destinationField);
        if receiveSucceeding then
          begin {~~d~~Disassemble frame}
            destinationParam := destinationField;
            sourceParam := sourceField;
            lengthOrTypeParam~~: =~~ := lengthOrTypeField;
            dataParam := RemovePad-(lengthOrTypeField, dataField);
            fcsParamValue := fcsField;
            fcsParamPresent := passReceiveFCSMode;
            exceedsMaxLength := ...; ~~(~~{Check to determine if receive frame size exceeds the maximum
                                    permitted frame size. MAC implementations may use either
                                    maxUntaggedFrameSize or (maxUntaggedFrameSize +
                                    qTagPrefixSize) for the maximum permitted frame size,
                                    either as a constant or as a function of whether the frame being
                                    received is a basic or tagged frame (see 3.2, 3.5). In
                                    implementations that treat this as a constant, it is recommended
                                    that the larger value be used. The use of the smaller value
                                    in this case may result in valid tagged frames exceeding the
                                    maximum permitted frame size.~~}
            if exceedsMaxLength then status := frameTooLong
            else if fcsField = CRC32(incomingFrame) and extensionOK then
‡               if validLength then status := receiveOK else status := lengthError
‡           else if excessBits = 0 or not extensionOK then status := frameCheckError
‡           else status := alignmentError;
              ~~if fcsField = CRC32(incomingFrame) and extensionOK then~~
                ~~begin~~
‡                 ~~if validLength then status: = receiveOK~~
‡                 ~~else status: = lengthError~~
                ~~end~~
              ~~else~~
                ~~begin~~
‡                 ~~if excessBits = 0 or not extensionOK then status:= frameCheckError~~
‡                 ~~else status := alignmentError~~
                ~~end;~~
‡           LayerMgmtReceiveCounters(status); {Update receive counters in 5.2.4.3}
              ~~{update receive and receive error counters in 5.2.4.3}~~
            view~~: =~~ := bits
```

```
                end {dDisassemble frame}
‡           end; {wWith incomingFrame}
‡       ReceiveDataDecap := status
    end; {ReceiveDataDecap}

    function RecognizeAddress (address: AddressValue): Boolean;
    begin
        RecognizeAddress := ...; {Returns true for the set of physical, broadcast,
                                  and multicast-group addresses corresponding
                                  to this station}
    end; {RecognizeAddress}

    function LayerMgmtRecognizeAddress(address: AddressValue): Boolean;
    begin
        if {promiscuous receive enabled} then LayerMgmtRecognizeAddress := true;
        if address = ... {MAC station address} then LayerMgmtRecognizeAddress := true;
        if address = ... {Broadcast address} then LayerMgmtRecognizeAddress := true;
        if address = ... {One of the addresses on the multicast list and multicast reception is enabled} then
                                                              LayerMgmtRecognizeAddress := true;
        LayerMgmtRecognizeAddress := false
    end; {LayerMgmtRecognizeAddress}
```

The function RemovePad strips any padding that was generated to meet the minFrameSize constraint, if possible. When the MAC sublayer operates in the mode that enables passing of the frame check sequence field of all received frames to the MAC client (passReceiveFCSMode variable is true), it shall not strip the padding and it shall leave the data field of the frame intact. Length checking is provided for Length interpretations of the Length/Type field. For Length/Type field values in the range between maxValidFrame and minTypeValue, the behavior of the RemovePad function is unspecified.:

```
    function RemovePad(var lengthOrTypeParam: LengthOrTypeValue; dataParam: DataValue): DataValue;
        var lengthOrTypeParam:LengthOrTypeValue; dataParam:DataValue):DataValue;
    begin
        if lengthOrTypeParam ≥ minTypeValue then
            begin
                validLength := true; {Don't perform length checking for Type field interpretations}
                RemovePad := dataParam
            end
        else if lengthOrTypeParam ≤ maxValidFrame then
            begin
                if lengthOrTypeParam ≤ maxValidFrame then
                begin
                validLength := {For length interpretations of the Length/Type field, check to determine if value
                                represented by Length/Type field matches the received clientDataSize};
                if validLength and not passReceiveFCSMode then
                    RemovePad := {tTruncate the dataParam (when present) to the value represented by the
                                  lengthOrTypeParam (in octets) and return the result}
                else RemovePad := dataParam
                end
            end
    end; {RemovePad}
```

ReceiveLinkMgmt attempts repeatedly to receive the bits of a frame, discarding any fragments from collisions by comparing them to the minimum valid frame size:

```
procedure ReceiveLinkMgmt;
begin
   repeat
      StartReceive;
      while receiving do nothing; {~~w~~Wait for frame to finish arriving}
      excessBits := frameSize mod 8;
      frameSize := frameSize – excessBits; {~~t~~Truncate to octet boundary}
      receiveSucceeding := receiveSucceeding and (frameSize ≥ minFrameSize)
                                                          {~~r~~Reject collision fragments}
   until receiveSucceeding
end; {ReceiveLinkMgmt}

procedure StartReceive;
begin
   receiveSucceeding := true;
   receiving := true
end; {StartReceive}
```

The BitReceiver process runs asynchronously, receiving bits from the medium at the rate determined by the Physical Layer's ReceiveBit operation, partitioning them into frames, and optionally receiving them:

```
process BitReceiver;
   var   b-: PhysicalBit;
         incomingFrameSize: integer; {~~c~~Count of all bits received in frame including extension}
         frameFinished: Boolean;
         enableBitReceiver: Boolean;
         currentReceiveBit: 1..frameSize; {Position of current bit in incomingFrame}
begin
   cycle {~~o~~Outer loop}
      if receiveEnabled then
         begin {~~r~~Receive next frame from physical layer}
            currentReceiveBit := 1;
            incomingFrameSize := 0;
            frameFinished := false;
            enableBitReceiver := receiving;
            PhysicalSignalDecap; {Skip idle and extension, strip off preamble and sfd}
            if enableBitReceiver then extensionOK := true;
            while receiveDataValid and not frameFinished do
                                          ~~{inner loop to receive the rest of an incoming frame}~~
               begin {Inner loop to receive the rest of an incoming frame}
                  b := ReceiveBit; {~~n~~Next bit from physical medium}
                  incomingFrameSize := incomingFrameSize + 1;
                  if b = 0 or b = 1 then {~~n~~Normal case}
                     if enableBitReceiver then {~~a~~Append to frame}
                     begin
                        if incomingFrameSize > currentReceiveBit then extensionOK := false;
                                          {Errors in the extension get mapped to data bits on input}
                        incomingFrame[currentReceiveBit] := b;
                        currentReceiveBit := currentReceiveBit + 1
                     end
                  else if not extending then frameFinished := true; {b must be an extensionBit}
                     ~~if not extending then frameFinished := true;~~
                  if incomingFrameSize ≥ slotTime then extending := false
               end; {~~i~~Inner loop}
```

```
            if enableBitReceiver then
               begin
                  frameSize := currentReceiveBit – 1;
                  receiveSucceeding := not extending;
                  receiving := false
               end
         end {~~e~~Enabled}
      end {~~o~~Outer loop}
   end; {BitReceiver}
```

The bits received from the physical layer can take one of three values: data zero (0), data one (1), or extensionBit (EXTEND). The value extensionBit will not occur between the first preamble bit of a frame and the last data bit of a frame in normal circumstances. Extension bits are counted by the BitReceiver but are not appended to the incoming frame. The BitReceiver checks whether the bit received from the physical layer is a data bit or an extensionBit before appending it to the incoming frame. Thus, the array of bits in incomingFrame will only contain data bits. The underlying Reconciliation Sublayer (RS) maps incoming EXTEND_ERROR bits to normal data bits. Thus, the reception of additional data bits after the frame extension has started is an indication that the frame should be discarded.

```
   procedure PhysicalSignalDecap;
   begin
      {Receive one bit at a time from physical medium until a valid sfd is detected, discard bits and return~~.~~}
   end; {PhysicalSignalDecap}
```

The process SetExtending controls the extending variable, which determines whether a received frame must be at least slotTime bits in length or merely minFrameSize bits in length to be considered valid by the BitReceiver. SetExtending sets the extending variable to true whenever receiveDataValid is de-asserted, while in half duplex mode at ~~operating speeds above 100 Mb/s.~~ an operating speed of 1000 Mb/s:

```
   process SetExtending;
   begin
      ~~cycle~~cycle {~~l~~Loop forever}
         while receiveDataValid do nothing;
         extending := extend and halfDuplex
      end {~~l~~Loop}
   end; {SetExtending}
```