4.2.7.1 Common Constants and Types

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

```
const
    addressSize = ... ; {48 bits in compliance with 3.2.3}
    lengthOrTypeSize = 16; {in bits}
    clientDataSize = ...; {MAC client Data, see 4.2.2.2a)3)}
    padSize = ...; {in bits, = max (0, minFrameSize x addressSize
                        + lengthSize + clientDataSize + crcSize))}.
    dataSize = ...; {= clientDataSize + padSize}
    crcSize = 32;   {32 bit CRC = 4 octets}
    frameSize = ...; {= 2 x addressSize + lengthOrTypeSize + dataSize + crcSize,
                          see 4.2.2.2a)}
    minFrameSize = ... ;   {in bits, implementation-dependent, see 4.4}
    maxFrameSize = ... ;   {in octets, implementation-dependent, see 4.4}
    extendSize = ...; {in bits, implementation-dependent, see 4.4}
    extensionBit = ...; {a new type of non-data bit}
    minTypeValue = 1536; {minimum value of the Length/Type field for Type interpretation}
    maxValidFrame = maxFrameSize x addressSize + lengthOrTypeSize + crcSize) / 8;
                          {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 = ... ; {unit of time for collision handling, implementation-dependent, see 4.4}
    preambleSize = ... ; {in bits, physical-medium-dependent}
    sfdSize = 8; {8 bit start frame delimiter}
    headerSize = ...; {sum of preambleSize and sfdSize}
type
    Bit = 0..1;
    AddressValue = array [1..addressSize] of Bit;
    LengthOrTypeValue = array [1..lengthOrTypeSize] of Bit;
    DataValue = array [1..dataSize] of Bit;
    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)
```

end; {defines header for MAC frame}

4.2.7.2 Transmit state variables

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

const
interFrameSpacing = ... ; {minimum time between frames}
interFrameSpacingPart1 = ... ; {duration of first portion of interFrametiming. In range 0 up to 2/3 interFrameSpacing}
interFrameSpacingPart2 = ... ; {duration of remainder of interFrame. Equal to interFrameSpacing

attemptLimit = ... ; {Max number of times to attempt transmission}
backOffLimit = ... ; {Limit on number of times to back off }
BurstLength= ... ; {In Burst Mode, channel holding time limit}
jamSize = ... ; {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;
extension: 0..extendSize; {length of extension}
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}
halfDuplex: Boolean; {Indicates the desired mode. halfDuplex is a static variable; its value does not change between invocations of the Initialize procedure}
BurstMode: Boolean: {Enables the transmission of multiple packets in a single carrier event}
MyBurst : Boolean; {In BurstMode, the given station has acquired the medium and the burst timer MAC has not yet expired}
BurstStart : Boolean {In BurstMode, indicates that the first packet transmission is in progress}

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}
currentReceiveBit: 1..frameSize; {Position of current bit in incomingFrame}
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}
extendCount: 0..extendSize; {count of the extension bits at end of frame}
newBurst: Boolean; {In BurstMode, indicates whether this is the first frame of a burst}
FrameOver : Boolean; (In BurstMode, indicates the end of a frame within a burst}

4.2.7.4 Summary of interlayer interfaces

a) The interface to the LLC sublayer, defined in 4.3.2, is summarized below:

type
à TransmitStatus = (transmitDisabled, transmitOK, excessiveCollisionError);
{Result of TransmitFrame operation}
à 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}

function ReceiveFrame (
var destinationParam: AddressValue;
var sourceParam: AddressValue;
var lengthOrTypeParam: LengthOrTypeValue;
var dataParam: DataValue): 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 transmissions should defer}
transmitting: Boolean; {Indicates outgoing bits}
wasTransmitting: Boolean; {Indicates transmission in progress or just completed}
collisionDetect: Boolean; {Indicates medium contention}
procedure TransmitBit (bitParam: Bit); {Transmits one bit}
function ReceiveBit: Bit; {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: If in half-duplex operation the Initialize procedure waits for the medium to become idle, and then immediately starts the other processes, the Deference process will be unaware of the activity and hence will not generate the required interFrame gap. Thus there is a risk that the first frame transmission will violate the interFrame spacing requirement unless the Initialize procedure waits for a deference interval during startup.

If Layer Management is implemented, the Initialize procedure shall only be called as the result of the initializeMAC action (5.2.2.2.1).

```
procedure Initialize;
begin
    frameWaiting := false;
    deferring := false;
    newCollision := false;
    transmitting := false; {In interface to Physical Layer; see below}
    receiving := false;
    if BurstMode then
    begin
        MyBurst := false;
        newBurst := true
    end;
    while carrierSense do nothing;
    {Start execution of all processes; see NOTE above.}
end; {Initialize}
```

4.2.8 Frame transmission

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;
procedure TransmitDataEncap; ... {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 {assemble frame}
        view := fields;
        destinationField := destinationParam;
        sourceField := sourceParam;
        lengthOrTypeField: = lengthOrTypeParam;
        dataField := ComputePad (dataParam);
        fcsField := CRC32(outgoingFrame);
        view := bits
    end {assemble frame}
```

```
    with outgoingHeader do
    begin
        headerView: = headerFields;
        preamble: = ...; {* ô1010...10,õ LSB to MSB*}
        sfd: = ...; {* ô10101011,õ LSB to MSB*}
        headerView: = headerBits
    end
end; {TransmitDataEncap}
```

ComputePad appends an array of arbitrary bits to the LLCdataField 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}
```

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; {initialize}
    excessDefer := false;
    if BurstMode then  {Check to see if BurstTimer has expired}
        MyBurst := MyBurst and BurstTimer(BurstLength);
        frameWaiting := MyBurst {since BurstMode implies halfDuplex}
    end;
    while  (attempts<attemptLimit) and (not transmitSucceeding) do
    begin {loop}
        {If MyBurst is set, then go straight to transmission without checking deference. Otherwise . . .}
        if not (BurstMode and MyBurst) then
        begin
            if attempts>0 then BackOff;
            if halfDuplex then frameWaiting := true;
            while deferring do {defer to passing frame, if any*}
à               if halfDuplex then deferred := true; {or do nothing, without Layer Management}
            if  BurstMode then {this will be the first frame in a burst}
            begin
                StartBurstTimer;
                MyBurst := true;
                BurstStart := true
            end {Burst Mode starting a Burst}
        end; {not both BurstMode and MyBurst}
        lateCollisionError := false;
        if halfDuplex then frameWaiting := false;
```

*. The Deference process ensures that the variable deferring is not true for passing traffic in full-duplex mode.

```
            StartTransmit;
            if halfDuplex then
            begin
                while transmitting do WatchForCollision; {cancels MyBurst in a collision}
                if lateCollisionError then
                    lateCollisionCount := lateCollisionCount + 1;
                attempts := attempts + 1;
            end {half-duplex mode}
            else {full-duplex mode}
                while transmitting do nothing
        end; {loop}
        if transmitSucceeding then
        begin
            TransmitLinkMgmt := transmitOK;
            if BurstMode then
            begin
                BurstStart := false; {Canõt be the first packet anymore}
                MyBurst := BurstTimer(BurstLength) {Check to see if Burst has expired}
            end
        end
        else  TransmitLinkMgmt := excessiveCollisionError;
        LayerMgmtTransmitCounters;
            {update transmit and transmit error counters in 5.2.4.2}
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 Watch-ForCollision, once frame transmission has been initiated:

```
    procedure WatchForCollision;
    begin
        if transmitSucceeding and collisionDetect then
        begin
            if currentTransmitBit > (minFrameSize - headerSize + extendSize) then
                lateCollisionError := true;
            newCollision := true;
            transmitSucceeding := false;
            if BurstMode then
            begin
                MyBurst := false;
                if not BurstStart then
                    lateCollisionError := true {Every collision is late, unless it hits the first packet 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** ~~512 bit times~~ 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). An implementation may optionally elect to 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 := ...{uniformly distributed random integer r such that low
end; {Random}

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

procedure BackOff;
begin
if attempts = 1 *then* maxBackOff := 2
else if attempts
then maxBackOff := maxBackOff x 2;
Wait(slotTime x Random(0, maxBackOff))
end; {BackOff}

***procedure* StartBurstTimer;**
begin
{reset an independent realtime timer and start it timing}
***end*; {StartBurstTimer}**

***function* BurstTimer (**
begin
{return the value true if the specified number of microseconds have not elapsed since the most recent invocation of StartBurstTimer, otherwise return the value false}
***end*; {BurstTimer}**

The Deference process runs asynchronously to continuously compute the proper value for the variable deferring. **Note that, in the case of half-duplex burst mode, deferring remains true across the entire burst.**

process Deference;
begin
if halfDuplex *then cycle*{half-duplex loop}
while not carrierSense *do* nothing; {watch for carrier to appear}
deferring := true; {delay start of new transmissions}
wasTransmitting:=transmitting;
while carrierSense or transmitting *do*
wasTransmitting: = wasTransmitting or transmitting;

```
        if wasTransmitting then
            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)
            end;
        StartRealTimeDelay; {time out second part interframe gap}
        while RealTimeDelay(interFrameSpacingPart2) do  nothing;
        deferring: = false; {allow new transmissions to proceed}
        while frameWaiting do nothing {allow waiting transmission if any}
    end {half-duplex loop}
    else  cycle {full-duplex loop}
        while not transmitting do nothing; {wait for the start of a transmission}
        deferring := true; {inhibit future transmissions}
        while transmitting do nothing; {wait for the end of the current transmission}
        StartRealTimeDelay; {time out an interframe gap}
        while RealTimeDelay(interFrameSpacing) do nothing;
        deferring := false {donõt inhibit transmission}
    end {full-duplex loop}
end; {Deference}

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

function RealTimeDelay (
    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 Layerõs TransmitBit operation:

```
process BitTransmitter;
begin
    cycle {outer loop}
        if transmitting then
        begin {inner loop}
            if halfDuplex then extension := 0;
            PhysicalSignalEncap;  {Send preamble and start of frame delimiter}
            while transmitting do
            begin
                if halfDuplex and (currentTransmitBit > lastTransmitBit) then
                begin
                    transmitBit(extensionBit);
```

```
                extension := extension + 1
            end
            else
                TransmitBit(outgoingFrame[currentTransmitBit]);
                                {send next bit to Physical Layer}
            if newCollision then StartJam else NextBit
        end;
    end; {inner loop}
    else {not transmitting}
        if BurstMode and MyBurst then
        begin
            InterFrameSignalEncap; {continue extended carrier across a standard
                                interframe spacing}
            MyBurst := MyBurst and frameWaiting {End the burst unless another
                                frame is available}
        end
    end; {outer loop}
end; {BitTransmitter}

procedure PhysicalSignalEncap;
begin
    while currentTransmitBit
    begin
        TransmitBit(outgoingHeader[currentTransmitBit]);
                        {transmit header one bit at a time}
        currentTransmitBit := currentTransmitBit + 1
    end;
    if newCollision then StartJam else
    currentTransmitBit := 1
end; {PhysicalSignalEncap}

procedure InterFrameSignalEncap;
begin
        {transmit 96 bits of ExtensionBit}
end;

procedure NextBit;
begin
    currentTransmitBit := currentTransmitBit+1;
    transmitting := (currentTransmitBit
    if halfDuplex and not (BurstMode and not BurstStart) then {carrier extension may be required}
        transmitting := transmitting or (currentTransmitBit <= (minFrameSize + extendSize))
end; {NextBit}

procedure StartJam;
begin
    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

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

The procedure 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;
    function ReceiveDataDecap: ReceiveStatus; ... {nested function; see body below}
begin
    if receiveEnabled then
    repeat
        ReceiveLinkMgmt;
        ReceiveFrame := ReceiveDataDecap;
    until receiveSucceeding
    else
        ReceiveFrame := receiveDisabled
end; {ReceiveFrame}
```

If enabled, ReceiveFrame calls ReceiveLinkMgmt to receive the next valid frame, and then calls the internal procedure ReceiveDataDecap to return the frameõs fields to the MAC client if the frameõs 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;
à         var status : ReceiveStatus; {holds receive status information}
      begin
à         with incomingFrame do
à         begin
à             view := fields;
              receiveSucceeding := RecognizeAddress (incomingFrame, destinationField);
              receiveSucceeding := LayerMgmtRecognizeAddress (destinationField);
à             if receiveSucceeding then
              begin {disassemble frame}
                  destinationParam := destinationField;
                  sourceParam := sourceField;
                  lengthOrTypeParam: = lengthOrTypeField;
                  dataParam := RemovePad (lengthOrTypeField, dataField);
                  exceedsMaxLength := ...; {check to determine if receive frame size exceeds the
                                            maximum permitted frame size (maxFrameSize)}
                  if exceedsMaxLength then status := frameTooLong
                  else
                  if fcsField = CRC32 (incomingFrame) then
                  begin
à                     if validLength then ReceiveDataDecap: = receiveOK
à                     else status: = lengthError
                  end
                  else
                  begin
```

```
à                   if excessBits = 0 then ReceiveDataDecap := frameCheckError
à                   else status := alignmentError
                end;
                LayerMgmtReceiveCounters(status);
                    {update receive and receive error counters in 5.2.4.3}
                view: = bits
            end {disassemble frame}
à       end; {with incomingFrame}
à       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}
```

The function RemovePad strips any padding that was generated to meet the minFrameSize constraint, if possible. 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
            var dataParam:DataValue):DataValue;
    begin
            if lengthOrTypeParam then
            begin
                validLength:= true; {Donõt perform length checking for Type field interpretations}
                RemovePad := dataParam
            end
            else
            begin
                if lengthOrTypeParam 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 then
                        RemovePad:={truncate the dataParam (when present) to value
                                    represented by 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
```

```
            receiveSucceeding := true;
            StartReceive;
            while receiving do nothing; {wait for frame to finish arriving}
            excessBits := frameSize mod 8;
            frameSize := frameSize
            receiveSucceeding := receiveSucceeding and (frameSize >= minFrameSize)
                                            {reject collision fragments}
        until receiveSucceeding
    end; {ReceiveLinkMgmt}

    procedure StartReceive;
    begin
        currentReceiveBit := 1;
        receiving := true
    end; {StartReceive}
```

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

```
    process BitReceiver;
        var b : Bit;
    begin
        cycle {outer loop}
            if receiveEnabled then
            begin {receive next frame from physical medium}
                currentReceiveBit := 1; {moved here from StartReceive}
                extendCount := 0;
                FrameOver := false;
                PhysicalSignalDecap; {Skip idle and strip off preamble and sfd}
                while receiveDataValid and not FrameOver do
                 {inner loop to receive the rest of an incoming frame}
                begin
                    b := ReceiveBit; {next bit from physical medium}
                    if b = extensionBit then
                        if newBurst then {first frame may have needed carrier extension}
                            if (currentReceiveBit + extendCount) > (minFrameSize + extendSize) then
                            begin {extension is finished}
                                newBurst := false;
                                FrameOver := true
                            end
                            else {extension is not finished}
                                extendCount := extendCount + 1
                        else {remaining frames do not use carrier extension, this is interframe spacing}
                            FrameOver := true
                    else {next bit is not an extensionBit}
                    begin
                        if receiving then {append bit to frame}
                            incomingFrame[currentReceiveBit] := b;
                        newBurst := newBurst and
                             (currentReceiveBit + extendCount) < (minFrameSize + extendSize);
                        currentReceiveBit := currentReceiveBit + 1
                    end {not an extensionBit}
                end; {inner loop}
                receiving := false;
```

```
            frameSize := currentReceiveBit
            receiveSucceeding := not newBurst
        end {enabled}
    end {outer loop}
end; {BitReceiver}

procedure PhysicalSignalDecap;
begin
            {Receive one bit at a time from physical medium  until a valid sfd is detected,
    discard bits and return. In BurstMode, set newBurst := true when ReceiveDataValid
    is false, and treat an extensionBit like an idle if newBurst is false }
end; {PhysicalSignalDecap}
```