

```

        handle_net_change();
        netState = WATCH; // go to WATCH state
        watch_timer.start(1,25 sec);

// net change complete, this node is not moderator, no significant changes
event: NUT_timer.count <= DLL_SM_current.gb_start
condition: ( tMinus == 1
            && DLL_SM_pending.myaddr != 0
            && DLL_SM_pending.myaddr == DLL_SM_current.myaddr
            || SM_listen_only != SM_listen_only_req
            && !SM_listen_only)
            && netState != MOD
destination: rxMod0
action:
    handle_net_change();

// net change complete, this node is moderator, no major changes, and can remain
moderator
event: NUT_timer.count <= DLL_SM_current.gb_start
condition: ( tMinus == 1
            && DLL_SM_pending.myaddr != 0
            && DLL_SM_pending.myaddr == DLL_SM_current.myaddr
            || SM_listen_only != SM_listen_only_req
            && !SM_listen_only)
            && netState == MOD
            && SM_mod_enable == TRUE
            && min(DLL_SM_pending.myaddr, qlowman) == DLL_SM_pending.myaddr
destination: txMod0
action:
    handle_net_change();

// net change complete, this node is moderator, no major changes, but can't stay
moderator
event: NUT_timer.count <= DLL_SM_current.gb_start
condition: ( tMinus == 1
            && DLL_SM_pending.myaddr != 0
            && DLL_SM_pending.myaddr == DLL_SM_current.myaddr
            || SM_listen_only != SM_listen_only_req
            && !SM_listen_only)
            && netState == MOD
            && (SM_mod_enable == FALSE
            || min(DLL_SM_pending.myaddr, qlowman) != DLL_SM_pending.myaddr)
destination: rxMod0
action:
    handle_net_change();
    netState = NOTMOD;
    modcount = 0;
    lowcount = LOWCOUNTINIT;

//
// wait for moderator start
//
state: rxMod0

// got it
event: ph_frame_indication == TRUE
destination: rxMod1

// timed out waiting for moderator
event: NUT_timer.count <= 20 usec
destination: waitTone
action:
    missed_moderator();
    housekeeping();

//
// receive the moderator
//
state: rxMod1

// bad moderator
event: ph_frame_indication == FALSE
destination: badMod
action:
    DLL_event(DLL_EV_badFrame, macSrce);

// get MAC source ID
event: RX_dataReady == TRUE
destination: rxMod2
action:
    macSrce = RX_rxData;
    nqlowman = min(nqlowman, macSrce);

//
// continue receiving moderator
//
state: rxMod2

```

```

// bad DLPDU
event:    ph_frame_indication == FALSE
destination: badMod
action:
    DLL_event(DLL_EV_badFrame,macSrce);

// bad moderator DLPDU if it's null
event:    RX_endMAC
destination: badMod
action:
    DLL_event(DLL_EV_badFrame,macSrce);

// got an Lpacket, save the Lpacket for later, and go check the rest of the DLPDU
event:    RX_receivedLpacket
destination: rxMod3

//
// handle the end of a moderator DLPDU
//
state: rxMod3

// bad DLPDU
event:    ph_frame_indication == FALSE
destination: badMod
action:
    DLL_event(DLL_EV_badFrame,macSrce);

// bad DLPDU
event:    RX_abort
destination: badMod
action:
    DLL_event(DLL_EV_rxAbort);

// bad DLPDU if another Lpacket is in the moderator DLPDU
event:    RX_receivedLpacket
destination: badMod
action:
    DLL_event(DLL_EV_badFrame,macSrce);

// bad DLPDU
event:    RX_endMAC
condition: !RX_FCSOK
destination: badMod
action:
    DLL_event(DLL_EV_badFrame,macSrce);

// dupnode!
event:    RX_endMAC
condition: RX_FCSOK && macSrce == DLL_SM_current.myaddr
destination: endFrame
action:
    netState = DUPNODE;
    if(!dupflag)
    {
        dupflag = TRUE;
        DLL_event(DLL_EV_dupNode);
    }
    gen_timer.start(5,2 usec);
    // start hold timer after line is quiet
    holdoff_timer.start((DLL_SM_current.blanking+1) * 1,6 usec);

// good DLPDU, but bad header
event:    RX_endMAC
condition: RX_FCSOK
    && macSrce != DLL_SM_current.myaddr
    && (    RX_Lpacket.size != MODERATOR_SIZE
        || RX_Lpacket.ctl != MODERATOR_CTL
        || RX_Lpacket.service != MODERATOR_TAG
        || RX_Lpacket.dest != 0xFF)
destination: badMod

// moderator, but from an incorrect address (not lowman)
event:    RX_endMAC
condition: RX_FCSOK
    && macSrce != DLL_SM_current.myaddr
    && RX_Lpacket.size == MODERATOR_SIZE
    && RX_Lpacket.ctl == MODERATOR_CTL
    && RX_Lpacket.service == MODERATOR_TAG
    && RX_Lpacket.dest == 0xFF
    && macSrce > qlowman
destination: endFrame
action:
    DLL_event(DLL_EV_invalidModAddress);
    gen_timer.start(5,2 usec);
    // start hold timer after line is quiet
    holdoff_timer.start((DLL_SM_current.blanking+1) * 1,6 usec);

// this is a moderator from a supernode,

```

```

// or this node is at 0xFF so that all others look like a supernode
// adopt all the parameters from the moderator
event:    RX_endMAC
condition: RX_FCSOK
    && macSrce != DLL_SM_current.myaddr
    && RX_Lpacket.size == MODERATOR_SIZE
    && RX_Lpacket.ct1 == MODERATOR_CTL
    && RX_Lpacket.service == MODERATOR_TAG
    && RX_Lpacket.dest == 0xFF
    && macSrce <= qlowman
    && (macSrce == 0 || DLL_SM_current.myaddr == 0xFF)
destination: waitTone
action:
    qlowman = min(qlowman, macSrce);
    if (macSrce == 0 && currentMod != 0) DLL_event(DLL_EV_supernode);
    currentMod = macSrce;
    DLL_currentMod_indication(currentMod);
    DLL_SM_current.NUT_length = moderator.NUT_length;
    DLL_SM_current.gb_prestart = moderator.gb_prestart;
    DLL_SM_current.gb_start = moderator.gb_start;
    DLL_SM_current.gb_center = moderator.gb_center;
    DLL_SM_current.modulus = moderator.modulus;
    DLL_SM_current.blanking = moderator.blanking;
    DLL_SM_current.slotTime = moderator.slotTime;
    DLL_SM_current.smax = moderator.smax;
    DLL_SM_current.umax = moderator.umax;
    processModerator(moderator);
    housekeeping();

// good moderator received, everything matches local copy
// resynchronize and continue
event:    RX_endMAC
condition: RX_FCSOK
    && macSrce != DLL_SM_current.myaddr
    && RX_Lpacket.size == MODERATOR_SIZE
    && RX_Lpacket.ct1 == MODERATOR_CTL
    && RX_Lpacket.service == MODERATOR_TAG
    && RX_Lpacket.dest == 0xFF
    && macSrce <= qlowman
    && !(macSrce == 0 || DLL_SM_current.myaddr == 0xFF)
    && moderator.NUT_length == DLL_SM_current.NUT_length
    && moderator.gb_prestart == DLL_SM_current.gb_prestart
    && moderator.gb_start == DLL_SM_current.gb_start
    && moderator.gb_center == DLL_SM_current.gb_center
    && moderator.modulus == DLL_SM_current.modulus
    && moderator.blanking == DLL_SM_current.blanking
    && moderator.slotTime == DLL_SM_current.slotTime
    && moderator.smax == DLL_SM_current.smax
    && moderator.umax == DLL_SM_current.umax
destination: waitTone
action:
    qlowman = min(qlowman, macSrce);
    currentMod = macSrce;
    DLL_currentMod_indication(currentMod);
    processModerator(moderator);
    housekeeping();

// The moderator doesn't match, so this node is a rogue.
// Since in the guardband, go to rxmod for recovery
event:    RX_endMAC
condition: RX_FCSOK
    && macSrce != DLL_SM_current.myaddr
    && RX_Lpacket.size == MODERATOR_SIZE
    && RX_Lpacket.ct1 == MODERATOR_CTL
    && RX_Lpacket.service == MODERATOR_TAG
    && RX_Lpacket.dest == 0xFF
    && macSrce <= qlowman
    && !(macSrce == 0 || DLL_SM_current.myaddr == 0xFF)
    && !(
        moderator.NUT_length == DLL_SM_current.NUT_length
        && moderator.gb_prestart == DLL_SM_current.gb_prestart
        && moderator.gb_start == DLL_SM_current.gb_start
        && moderator.gb_center == DLL_SM_current.gb_center
        && moderator.modulus == DLL_SM_current.modulus
        && moderator.blanking == DLL_SM_current.blanking
        && moderator.slotTime == DLL_SM_current.slotTime
        && moderator.smax == DLL_SM_current.smax
        && moderator.umax == DLL_SM_current.umax
    )
    && in_guardband == TRUE
destination: rxMod0
action:
    qlowman = min(qlowman, macSrce);
    currentMod = macSrce;
    DLL_currentMod_indication(currentMod);
    netState = ROGUE;
    DLL_event(DLL_EV_rogue);

// The moderator doesn't match, so this node is a rogue.
// Since currently not in the guardband, go to endframe for recovery.

```

```

event:    RX_endMAC
condition: RX_FCSOK
    && macSrce != DLL_SM_current.myaddr
    && RX_Lpacket.size == MODERATOR_SIZE
    && RX_Lpacket.ctl == MODERATOR_CTL
    && RX_Lpacket.service == MODERATOR_TAG
    && RX_Lpacket.dest == 0xFF
    && macSrce <= qlowman
    && !(macSrce == 0 || DLL_SM_current.myaddr == 0xFF)
    && !(
        moderator.NUT_length == DLL_SM_current.NUT_length
        && moderator.gb_prestart == DLL_SM_current.gb_prestart
        && moderator.gb_start == DLL_SM_current.gb_start
        && moderator.gb_center == DLL_SM_current.gb_center
        && moderator.modulus == DLL_SM_current.modulus
        && moderator.blanking == DLL_SM_current.blanking
        && moderator.slotTime == DLL_SM_current.slotTime
        && moderator.smax == DLL_SM_current.smax
        && moderator.umax == DLL_SM_current.umax)
    && in_guardband == FALSE
destination: endFrame
action:
    qlowman = min(qlowman, macSrce);
    currentMod = macSrce;
    DLL_currentMod_indication(currentMod);
    netState = ROGUE;
    DLL_event(DLL_EV_rogue);
    gen_timer.start(5,2 usec);
    // start hold timer after line is quiet
    holdoff_timer.start((DLL_SM_current.blanking+1) * 1,6 usec);

//
// come here to handle a bad moderator DLPDU
// wait for link to become quiet, or until tone generation is due
//
state: badMod

    // This state is reached when this node identifies a moderator DLPDU,
    // but the DLPDU is subsequently reported bad so the state transfers
    // back to endFrame if this node is not in the guardband.
event:    TRUE
condition: in_guardband == FALSE
destination: endFrame
action:
    DLL_event(DLL_EV_badFrame,macSrce);
    gen_timer.start(5,2 usec);
    // start hold timer after line is quiet
    holdoff_timer.start((DLL_SM_current.blanking+1) * 1,6 usec);

// the guardband should be ended NOW
event:    NUT_timer.count <= 30 usec
destination: waitTone
action:
    missed_moderator();
    housekeeping();

// else wait until the link is quiet
event:    ph_lock_indication == FALSE
destination: rxMod0

//
// send the moderator, wait for guardband center
//
state: txMod0

event:    NUT_timer.count <= DLL_SM_current.gb_center
destination: txMod1
action:
    TX_sendHeader(DLL_SM_current.myaddr);    // send header

//
// send more
//
state: txMod1

event:    TX_headerComplete
destination: txMod2
action:
    moderator.size = MODERATOR_SIZE;
    moderator.ctl = MODERATOR_CTL;
    moderator.service = MODERATOR_TAG;
    moderator.dest = 0xff;
    moderator.NUT_length = DLL_SM_current.NUT_length;
    moderator.smax = DLL_SM_current.smax;
    moderator.umax = DLL_SM_current.umax;
    moderator.slotTime = DLL_SM_current.slotTime;
    moderator.blanking = DLL_SM_current.blanking;
    moderator.gb_start = DLL_SM_current.gb_start;
    moderator.gb_center = DLL_SM_current.gb_center;

```

```

        moderator.usr = usr;
        moderator.interval_count = interval_count;
        moderator.modulus = DLL_SM_current.modulus;
        moderator.tMinus = tMinus;
        moderator.gb_prestart = DLL_SM_current.gb_prestart;
        moderator.spare = 0;

        TX_sendLpacket(moderator);          // send the Lpacket

//
// finish up
//
state: txMod2

    event:      TX_LpacketComplete
    destination: txMod3
    action:
        TX_sendTrailer();

//
// at completion, transferring to waiting for tone
//
state: txMod3

    event:      TX_trailerComplete
    destination: waitTone
    action:
        currentMod = DLL_SM_current.myaddr;
        DLL_currentMod_indication(currentMod);
        housekeeping();

//
// wait for tone
//
state: waitTone

    // end of this NUT, and transferring to off-line
    event:      NUT_timer.count == 0
    condition:   SM_online == FALSE
    destination: offline
    action:
        DLL_tone_indication(interval_count);
        DLL_online_confirm(SM_online);

    // end of this NUT, start of another
    event:      NUT_timer.count == 0
    condition:   SM_online_req == TRUE
    destination: waitSlotZero
    action:
        DLL_tone_indication(interval_count);
        NUT_timer.restart();
        scheduled = TRUE;
        in_guardband = FALSE;

        // housekeeping

        // If this node detects line activity, but not moderators, attempt to recover.
        // Maybe problem is that the local node is performing housekeeping (and is
        // therefore deaf) during the time the moderator DLPDU is transmitted.

        if ((netState != MOD || netState != NOTMOD)
            && qlowman != 255
            && ph_frame_indication == TRUE)
        {
            deafcount++;
        }
        else
        {
            deafcount = 0;
        }

        gen_timer.start(20 usec); // start timer for start of scheduled time

//
// wait for start of scheduled
//
state: waitSlotZero

    // start a new scheduled token pass
    event:      gen_timer.expired()
    destination: gap
    action:
        itr = -1;
        if (DLL_SM_current.myaddr == 0) // supernode doesn't try to detect lowman
        {
            nqlowman = 0;

```

```

        qlowman = 0;
    }
    else // else initialize the detector
    {
        nqlowman = 255;
        qlowman = 255;
    }
    slot_timer.restart();
    gen_timer.start(0,6 usec);

```

9.3 TxLLC

The TxLLC (transmit LLC) shall receive and buffer Lpackets from the upper layers. It shall pick the next Lpacket to be transmitted based on

- the order in which Lpackets were queued;
- attributes of the Lpacket;
- information provided by the Access Control Machine (ACM).

The TxLLC shall present the selection Lpacket to the ACM for transmission.

```

// DLL TX LLC State Machine Description

// This state machine accepts transmit requests from the DLS-user,
// queues and prioritizes them, and submits one Lpacket at a time
// to the ACM based on parameters received from the ACM.

////////////////////////////////////
//
// type and constant definitions
//
typedef enum {FALSE=0, TRUE=1} BOOL;
typedef void *IDENTIFIER;

typedef enum { M_0,
               M_1,
               M_ND_plus,
               M_ND_minus } M_SYMBOL;

// These are the three transmit priorities.
// hard assignments are so that the priority can be
// used as an index into an array of FIFOs
// note that HIGH and LOW are unscheduled

typedef enum { SCHEDULED=0,
               HIGH=1,
               LOW=2 } PRIORITY;

typedef enum { OK,
               TXABORT,
               FLUSHED } TXSTATUS; // describes the result of a transmit request.

//
// Lpacket class
//
class Lpacket
{
public:

    // return the size octet of the current Lpacket
    USINT size;

    // return the ctl octet of the current Lpacket
    USINT ctl;

// Lpacket constants: masks for ctl octet

#define FIXEDSCREEN 1
#define TAGPAD 2
#define DATAPAD 4

    // return the value of the tag pad bit
    int tag_pad(void)
    {
        return (ctl&TAGPAD) >>1;
    }

    // return the value of the data pad bit
    int data_pad(void)
    {

```

```

        return (ctl&DATAPAD) >>2;
    }

    // return the number of octets in the Lpacket
    int wire_size(void)
    {
        return size*2 - tag_pad() - data_pad();
    }

    // store next octet to the Lpacket
    void put_octet(USINT data);

    // get an octet from the Lpacket
    USINT &operator[](int index);

    Lpacket(void *p)          // constructor
    {
    }

    Lpacket(int size)          // constructor
    {
    }
};

//
// superclass of Lpacket that defines an Lpacket to be transmitted
//
class txLpacket : public Lpacket
{
public:

    IDENTIFIER id;
    BOOL fixed;

    // constructors

    txLpacket(void *p) : Lpacket(p)
    {
    }

    txLpacket(int size) : Lpacket(size) // size is size of PDU buffer
    {
        // (Lpacket plus pads, in octets)
    }
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// interface to DLS-user
//
void DLL_xmit_fixed_request(
    IDENTIFIER id,
    USINT      Lpacket[],
    UINT       size,
    PRIORITY   priority,
    USINT      service,
    USINT      destID);

extern void DLL_xmit_fixed_confirm (IDENTIFIER id, TXSTATUS status);

void DLL_xmit_generic_request(
    IDENTIFIER id,
    USINT      Lpacket[],
    UINT       size,
    PRIORITY   priority,
    USINT      tag[3]);

extern void DLL_xmit_generic_confirm (IDENTIFIER id, TXSTATUS status);

void DLL_flush-requests-by-QoS_request (PRIORITY priority);

extern void DLL_flush-requests-by-QoS_confirm (PRIORITY priority);

DLL_flush_single_request( PRIORITY priority, IDENTIFIER xmit_id );

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// interface to ACM
//
txLpacket *pickLpacket(BOOL scheduled, int octetsLeft); // pick an Lpacket to send next
void LpacketSent(TXSTATUS status);                     // the Lpacket was sent

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// interface to station management
//

```

```

void SM_powerup(void);          // input indication: powerup has occurred

////////////////////////////////////
//
// a class to represent message FIFOs
//
class FIFO
{
public:

    void put(txLpacket lp);      // add an Lpacket to the FIFO
    txLpacket &get(void);        // remove an Lpacket from the FIFO
    txLpacket &peek(void);       // look at the first Lpacket in the FIFO
    void flush(void);           // delete all Lpackets in the FIFO
    void flush1(IDENTIFIER id);  // delete one Lpacket in the FIFO, given it's ID
    BOOL has_data();            // true if the FIFO is not empty
};

FIFO fifo[3];                  // at least three priority levels shall be provided

////////////////////////////////////
//
// TxLLC implementation
//

//
// powerup initialization
//
void SM_powerup(void)
{
    fifo[SCHEDULED].flush();
    fifo[HIGH].flush();
    fifo[LOW].flush();
}

//
// flush a particular FIFO
//
void DLL_flush-requests-by-QoS-request (PRIORITY priority)
{
    fifo[priority].flush();
    DLL_flush-requests-by-QoS_confirm(priority);
}

//
// flush a particular Lpacket
//
void DLL_flush_single_request (PRIORITY priority, IDENTIFIER id)
{
    fifo[priority].flush1(id);
}

//
// requests from DLS-user to submit Lpackets for transmission
//
void DLL_xmit_fixed_request(
    IDENTIFIER id,
    USINT      Lpacket[],
    UINT       size,
    PRIORITY   priority,
    USINT      service,
    USINT      destID)
{
    int tmp;

    tmp = size + 4 + (size&1);    // size of DLSdu plus 4 octet fixed Lpacket header
                                // plus optional data pad

    txLpacket &lp(new txLpacket(tmp)); // create a new Lpacket buffer

    // build the PDU (Lpacket)
    lp[0] = tmp/2;                // size, in words
    lp[1] = 0x01 | ((size&1)<<2); // ctl octet, plus data pad bit, if needed
    lp[2] = service;              // fixed tag service
    lp[3] = destID;               // destination MAC ID
    memcpy(&lp[4], Lpacket, size); // copy the rest of the data
                                // there may be a pad octet sent after the DLSdu
    lp.id = id;                   // store the user's id (for the confirmation)
    lp.fixed = TRUE;              // store type of Lpacket (fixed)
    fifo[priority].put(lp);       // queue the Lpacket
}

void DLL_xmit_generic_request(
    IDENTIFIER id,
    USINT      Lpacket[],
    UINT       size,

```



```

        PRIORITY    priority,
        USINT       tag[3])
{
    int tmp;
    int pad;

    tmp = size + 6 + (size&1);    // octet size of DLSdu plus 6 octet GEN Lpacket header
                                // plus data pad
    pad = 0;

    txLpacket &lp(new txLpacket(tmp)); // create a new Lpacket buffer

    // build the PDU (Lpacket)
    lp[0] = tmp/2;                // size, in words
    lp[1] = 0x12 | ((size&1)<<2); // control octet,
                                // plus maybe a data pad bit,

    lp[2] = 0;                    // tag pad octet (affects memory image of Lpacket)
    lp[3] = tag[0];               // generic tag
    lp[4] = tag[1];               // generic tag
    lp[5] = tag[2];               // generic tag
    memcpy(&lp[6], Lpacket, size); // copy the rest of the data
                                // there may be pad octet after the DLSdu

    lp.id = id;                   // store the user's identifier (for the confirm)
    lp.fixed = FALSE;             // store type of Lpacket (generic)

    fifo[priority].put(lp);       // queue the new Lpacket
}

static txLpacket *picked = 0;    // remember which Lpacket was picked

//
// called by ACM to pick the next Lpacket to be sent out
//
txLpacket *pickLpacket(BOOL scheduled, int octetsLeft)
{
    int wordsleft = (octetsLeft+1)/2; // wordsleft is rounded up, accepting that in
    some                               // cases this results in an Lpacket not being sent

    if(scheduled)                   // if scheduled, only look at scheduled FIFO
    {
        // if there is anything in the FIFO and it fits in the time left
        if(fifo[SCHEDULED].has_data()
            && fifo[SCHEDULED].peek().size <= wordsleft)
        {
            picked = &fifo[SCHEDULED].get(); // then pick it to be sent
        }
        else picked = 0;             // no Lpacket available
    }
    else                             // if unscheduled -- look at all FIFOs in order
    {
        // if there is anything in the FIFO and it fits in the time left
        if(fifo[SCHEDULED].has_data()
            && fifo[SCHEDULED].peek().size <= wordsleft)
        {
            picked = &fifo[SCHEDULED].get(); // then pick it to be sent
        }
        else if(fifo[HIGH].has_data() // ditto for HIGH
            && fifo[HIGH].peek().size <= wordsleft)
        {
            picked = &fifo[HIGH].get();
        }
        else if(fifo[LOW].has_data() // ditto for LOW
            && fifo[LOW].peek().size <= wordsleft)
        {
            picked = &fifo[LOW].get();
        }
        else picked = 0;             // no Lpacket available
    }

    return picked;
}

//
// confirmation from the ACM that the picked Lpacket was sent (or possibly aborted)
//
void LpacketSent(TXSTATUS status)
{
    if(picked->fixed)                // if the Lpacket was fixed
    {
        DLL_xmit_fixed_confirm (picked->id, status); // generate a fixed confirm
    }
    else                             // else
    {

```

```

        DLL_xmit_generic_confirm (picked->id, status); // generate a generic confirm
    }

    delete picked; // delete temp data
    picked = 0;
}

```

9.4 RxLLC

The RxLLC (receive LLC) shall buffer Lpackets from the Receive Machine (RxM) as they are assembled. When the RxM indicates that a good DLPDU has concluded, all buffered Lpackets shall be presented to the upper layers in the order received. If the RxM indicates a bad DLPDU has concluded, all buffered Lpackets shall be discarded.

```

// DLL RX LLC State Machine Description

// This state machine gets Lpackets from the RxM,
// and DLPDU status from the deserializer.
// It handles quarantining, and passes quarantined
// Lpackets up to the DLS-user

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// generic type and constant definitions
//
typedef enum {FALSE=0, TRUE=1} BOOL;

//
// Lpacket class
//
class Lpacket
{
public:

    // the size octet of the current Lpacket
    USINT size;

    // the ctl octet of the current Lpacket
    USINT ctl;

// Lpacket constants: masks for ctl octet

#define FIXEDSCREEN 1
#define TAGPAD 2
#define DATAPAD 4

    // return the value of the tag pad bit
    int tag_pad(void)
    {
        return (ctl&TAGPAD) >>1;
    }

    // return the value of the data pad bit
    int data_pad(void)
    {
        return (ctl&DATAPAD) >>2;
    }

    // return the number of octets in the Lpacket
    int wire_size(void)
    {
        return size*2 - tag_pad() - data_pad();
    }

    // store next octet to the Lpacket
    void put_octet(USINT data);

    USINT &operator[](int index);

    // init for a new Lpacket
    void init(void);

    Lpacket(void *p) {} // constructor
};

```

```

class rxLpacket : public Lpacket
{
public:
    USINT    source;           // the MAC ID of the node that sent this Lpacket
    int      memory_size;      // size of the Lpacket in memory
    BOOL     fixed;           // true if fixed , false if generic

    rxLpacket(void *p) : Lpacket(p) {}
};

/////////////////////////////////////////////////////////////////
//
// a class to represent message FIFOs
//
class FIFO
{
public:

    void put(rxLpacket lp);      // add an Lpacket to the fifo
    rxLpacket &get(void);        // remove an Lpacket from the fifo
    void flush(void);           // delete all Lpackets in the FIFO
    BOOL has_data();            // true if the fifo is not empty
};

/////////////////////////////////////////////////////////////////
//
// a class to represent generic tags
//
class gen_tag
{
    USINT data[3];
public:
    // constructor
    gen_tag(USINT first, USINT second, USINT third)
    {
        data[0] = first;
        data[1] = second;
        data[2] = third;
    }

    USINT &operator[](int index)
    {
        return data[index];
    }
};

/////////////////////////////////////////////////////////////////
//
// internal variables
//

FIFO fifo;

/////////////////////////////////////////////////////////////////
//
// interface to PhL
//
BOOL ph_lock_indication; // optimistic DLPDU detect (clock recovery is tracking)
BOOL ph_frame_indication; // pessimistic DLPDU detect (Manchester valid since the SD)

/////////////////////////////////////////////////////////////////
//
// interface to Deserializer
//
extern BOOL RX_endMAC;      // the end delimiter has been detected
extern BOOL RX_FCSOK;       // FCS on last DLPDU was OK
extern BOOL RX_abort;       // abort received on last DLPDU

/////////////////////////////////////////////////////////////////
//
// interface to RxM
//
BOOL RX_receivedLpacket; // indication: an Lpacket has been received
rxLpacket RX_Lpacket(0); // data: the Lpacket most recently received

/////////////////////////////////////////////////////////////////
//

```