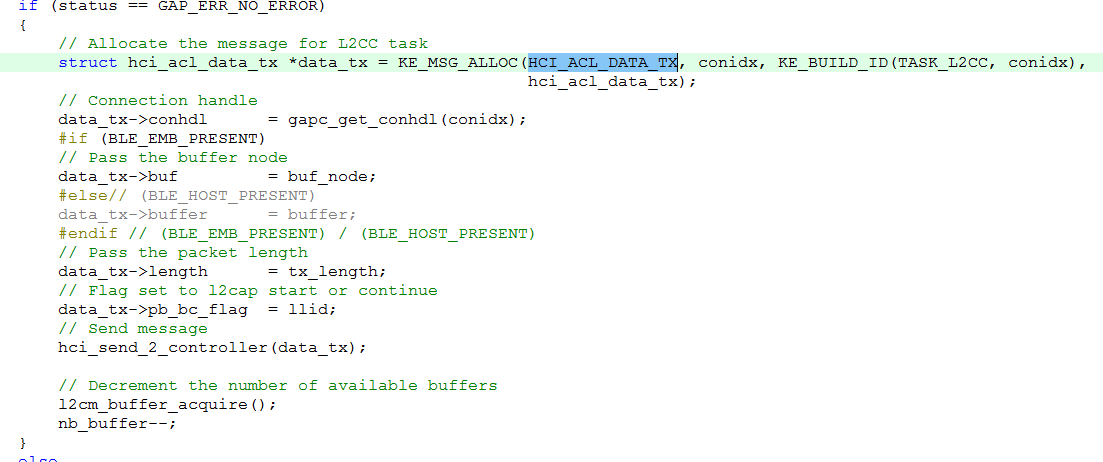
关于bk3432 对应RW IP 下发送数据在特定情况下 tx buff 没有free 的问题

1. 分析dev 端发送一个NOTIFY 数据过程为例
2. app 层 通过 KE\_MSG\_ALLOC\_DYN(GATTC\_SEND\_EVT\_CMD, 这个消息，operation 设置成GATTC\_NOTIFY，其它参数按照要求填入，该消息会发送给GATTC\_TASK 来处理。
3. GATTC\_TASK 会收到该消息，在 gattc\_send\_evt\_cmd\_handler 中调用处理 atts\_send\_event(conidx, param);
4. atts\_send\_event 中预处理之后通过 L2CC\_PDU\_SEND\_CMD 发送给L2CC\_TASK,并在l2cc\_pdu\_send\_cmd\_handler 中处理。
5. l2cc\_pdu\_send\_cmd\_handler 会把数据放到 l2cc\_env[conidx]->tx\_queue 的list中，接下来通过 l2cm\_tx\_status函数的 l2cm\_env.nb\_buffer\_avail来判断是否有buffer 来处理数据，有的话则 ke\_event\_set(KE\_EVENT\_L2CAP\_TX);
6. ke\_event\_callback\_set(KE\_EVENT\_L2CAP\_TX , &l2cm\_l2cap\_tx\_handler);
7. l2cm\_l2cap\_tx\_handler 中条件符合则调用l2cc\_data\_send
8. l2cc\_data\_send 函数，首先会把 要发送的数据从list 中取出来struct ke\_msg \*pkt = (struct ke\_msg \*) co\_list\_pick(&(env->tx\_queue));
9. 我们发送数据根据 IP 设计，需要一个DATA TX BUFF node, TX DESC node。
10. struct em\_buf\_node\* buf\_node = em\_buf\_tx\_alloc(); 获取到一个data tx buff。
11. buffer = (uint8\_t \*)(EM\_BASE\_ADDR + buf\_node->buf\_ptr); 拿到数据地址
12. 数据包装好之后



其中data\_tx->buf = buf\_node; 指向了我们分配的 data tx buff node。

l2cm\_buffer\_acquire 对l2cm\_env.nb\_buffer\_avail 进行处理，使得data tx buff node 个数和实际保持同步。该消息通过HCI\_ACL\_DATA\_TX 发送到control 层

12．llc\_task 中{HCI\_ACL\_DATA\_TX, (ke\_msg\_func\_t)llc\_hci\_acl\_data\_tx\_handler},

13. llc\_hci\_acl\_data\_tx\_handler-> lld\_pdu\_data\_send((void\*)param);

bool lld\_pdu\_data\_send(void \*param)

{

uint8\_t status = false;

GLOBAL\_INT\_DISABLE();

struct lld\_pdu\_data\_tx\_tag \*data\_tx = (struct lld\_pdu\_data\_tx\_tag\*)ke\_malloc(sizeof(struct lld\_pdu\_data\_tx\_tag), KE\_MEM\_ENV);

if(data\_tx)

{

struct hci\_acl\_data\_tx\* data\_to\_tx = (struct hci\_acl\_data\_tx\*)param;

// Get associated BLE event environment

struct lld\_evt\_tag \*evt = LLD\_EVT\_ENV\_ADDR\_GET(llc\_env[data\_to\_tx->conhdl]->elt);

// Get list of data packets ready for programming

struct co\_list \*list = &evt->tx\_acl\_rdy;

data\_tx->buf = data\_to\_tx->buf;

data\_tx->conhdl = data\_to\_tx->conhdl;

data\_tx->length = data\_to\_tx->length;

data\_tx->pb\_bc\_flag = data\_to\_tx->pb\_bc\_flag;

data\_tx->idx = BLE\_TX\_DESC\_CNT;

// Push the llcp pdu allocated at the end of the TX llcp pending list

co\_list\_push\_back(list, &data\_tx->hdr);

status = true;

}

GLOBAL\_INT\_RESTORE();

return (status);

}

该函数struct lld\_pdu\_data\_tx\_tag \*data\_tx = (struct lld\_pdu\_data\_tx\_tag\*)ke\_malloc(sizeof(struct lld\_pdu\_data\_tx\_tag), KE\_MEM\_ENV);分配了一个 lld\_pdu\_data\_tx\_tag \*data\_tx的内存，data\_tx->buf = data\_to\_tx->buf; 把我们的data tx buff node 指向 其 data-tx->buf. 并把data\_tx->idx = BLE\_TX\_DESC\_CNT; 把该idx 设置成 TX\_DESC\_CNT 的最大值。这里是一个关键位置，这时候的TX data buf 和TX DESC 还没有发生关联。之后该数据被 co\_list\_push\_back(list, &data\_tx->hdr); 放到tx\_acl\_rdy list。

14.在 lld\_evt\_schedule 中会处理 lld\_pdu\_tx\_prog(evt);

15. if (!co\_list\_is\_empty(acl\_rdy) && (send\_acl))

{

#if (BLE\_CENTRAL || BLE\_PERIPHERAL)

struct lld\_pdu\_data\_tx\_tag \*txacl = NULL;

if((evt->conhdl < BLE\_CONNECTION\_MAX) )

{

uint16\_t nb\_tx\_desc\_available = co\_list\_size(&em\_buf\_env.tx\_desc\_free); //获取可用的tx\_desc 个数。

uint8\_t total\_packet\_prog = 0;

uint8\_t phy = ble\_txrate\_getf(evt->conhdl);

bool encrypted = (LLC\_UTIL\_ENC\_STATE\_IS\_SET(evt->conhdl, LLC\_ENC\_TX) != 0)? true: false;

txacl = (struct lld\_pdu\_data\_tx\_tag \*)lld\_pdu\_pop\_front(acl\_rdy);

while(txacl)

{

uint8\_t nb\_packet\_prog = 0;

// Check if the transmit max effective time is respected

send\_acl = lld\_pdu\_send\_packet(evt, txacl, &nb\_packet\_prog, nb\_tx\_desc\_available, phy, encrypted);

// If the packet cannot be sent it should be fragmented

if(!send\_acl)

{

lld\_pdu\_push\_front(acl\_rdy,&txacl->hdr);

break;

}

total\_packet\_prog += nb\_packet\_prog;

nb\_tx\_desc\_available -= nb\_packet\_prog;

lld\_pdu\_push\_back(&evt->tx\_acl\_tofree, &txacl->hdr);// 数据已经取出，可以被free了。此处的 tx\_acl 只是ke\_malloc 出来的内存。（lld\_pdu\_data\_tx\_tag）是在lld\_pdu\_data\_send 中malloc到的

if(total\_packet\_prog > BLE\_NB\_MAX\_PACKET\_PROG)

{

//Like not all the packet has been moved in the programmed queue do not free the ready queue

send\_acl = false;

break;

}

txacl = (struct lld\_pdu\_data\_tx\_tag \*)lld\_pdu\_pop\_front(acl\_rdy);

}

}

else

#endif // (BLE\_CENTRAL || BLE\_PERIPHERAL)

{

co\_list\_merge(tx\_prog, acl\_rdy);

}

// Transmit effective time not respected, do not send the acl packet(s)

if(send\_acl != false)

{

// Reset ready list

co\_list\_init(acl\_rdy);

}

}

lld\_pdu\_send\_packet(evt, txacl, &nb\_packet\_prog, nb\_tx\_desc\_available, phy, encrypted);

在 该函数中 /\*\*

\* Phase 2 Get a new descriptor

\*/

struct em\_desc\_node\* desc\_node = (struct em\_desc\_node \*)em\_buf\_tx\_desc\_alloc();

获取到 tx desc node。最后 //Fill all the information before pushing the descriptor

desc\_node->buffer\_idx = temp\_buff\_idx\_val;

desc\_node->buffer\_ptr = temp\_data\_ptr\_val; 把我们前面的tx data buf node 和这里关联在一起。

之后 lld\_pdu\_data\_tx\_push 设置到硬件中，并把 tx desc node 发到 tx\_prog 的list中。

16. 在每次的evt 结束后 在lld\_pdu\_check中 txnode = (struct em\_desc\_node \*)co\_list\_pick(&evt->tx\_prog); 把tx\_node 从tx\_prog list 中拿出来，并判断数据发送成功后会if(em\_buf\_tx\_free(txnode))

{

//Buffer is sent, increase the tx counter

tx\_cnt++;

}并增加 tx\_cnt 计数值。llc\_common\_nb\_of\_pkt\_comp\_evt\_send 最后在l2cm\_env.nb\_buffer\_avail体现出来。

到此，一个完整的发送流程就这样结束了。Tx buff 和 tx desc 同步 释放掉。

二．异常时间点导致数据未发送出去

1.当app层发送数据到l2cc 在l2cc\_data\_send 函数中 struct em\_buf\_node\* buf\_node = em\_buf\_tx\_alloc(); 这个时候 只是分配tx data buf node。并只在 lld\_pdu\_data\_send 中把对应的idx 设置成了 data\_tx->idx = BLE\_TX\_DESC\_CNT;同时放到tx\_acl\_rdy list里面。

（注意，这个时候的 list 里面的内容是 struct lld\_pdu\_data\_tx\_tag 的格式）

2.在这个时候连接断开了，系统就会走相应的断开流程，会把buff 给释放掉。

3.ke\_event\_callback\_set(KE\_EVENT\_BLE\_EVT\_DELETE, &lld\_evt\_delete\_elt\_handler);

4. lld\_con\_stop -> lld\_evt\_delete\_elt\_push -> ke\_event\_set(KE\_EVENT\_BLE\_EVT\_DELETE); -> lld\_evt\_elt\_delete-> lld\_pdu\_tx\_flush(evt);

5. void lld\_pdu\_tx\_flush(struct lld\_evt\_tag \*evt)

{

#if (BLE\_CENTRAL || BLE\_PERIPHERAL)

uint8\_t nb\_of\_pkt\_flushed = 0;

// Free all memory allocated for the LLCP management

// Clear the pending LLCP

GLOBAL\_INT\_DISABLE();

struct co\_list \*list = &evt->tx\_llcp\_pdu\_rdy;

while(!co\_list\_is\_empty(list))

{

struct llcp\_pdu\_tag \*llcp\_elt = (struct llcp\_pdu\_tag \*)co\_list\_pop\_front(list);

ke\_free(llcp\_elt->ptr);

ke\_free(llcp\_elt);

nb\_of\_pkt\_flushed++;

}

// Flush TX data - Programmed and then Ready lists

nb\_of\_pkt\_flushed += lld\_pdu\_tx\_flush\_list(&evt->tx\_prog);

nb\_of\_pkt\_flushed += lld\_pdu\_tx\_flush\_list(&evt->tx\_acl\_rdy);

struct lld\_pdu\_data\_tx\_tag \*tx\_tofree;

while (!co\_list\_is\_empty(&evt->tx\_acl\_tofree))

{

tx\_tofree = (struct lld\_pdu\_data\_tx\_tag \*)co\_list\_pop\_front(&evt->tx\_acl\_tofree);

if (tx\_tofree)

{

ke\_free(tx\_tofree);

}

}

GLOBAL\_INT\_RESTORE();

// if the number of packet flushed is not NULL send a number of packets

if (nb\_of\_pkt\_flushed > 0)

{

llc\_common\_nb\_of\_pkt\_comp\_evt\_send(evt->conhdl, nb\_of\_pkt\_flushed);

}

#else

// Flush TX data - Programmed and then Ready lists

lld\_pdu\_tx\_flush\_list(&evt->tx\_prog);

lld\_pdu\_tx\_flush\_list(&evt->tx\_acl\_rdy);

#endif //(BLE\_CENTRAL || BLE\_PERIPHERAL)

}

🡪 lld\_pdu\_tx\_flush\_list

在 lld\_pdu\_tx\_flush\_list 中在处理 tx\_acl\_rdy list 时候，由于其数据还没有 和 tx desc 关联起来。

struct em\_desc\_node

{

struct co\_list\_hdr hdr;

/// Index of the buffer

uint16\_t idx;

/// EM Pointer of the buffer

uint16\_t buffer\_idx;

/// Buffer index

uint16\_t buffer\_ptr;

/// Logical Link Identifier

uint8\_t llid;

/// Data length

uint8\_t length;

};

/// ACL data TX element structure

struct lld\_pdu\_data\_tx\_tag

{

/// List element for chaining

struct co\_list\_hdr hdr;

/// index

uint16\_t idx;

/// connection handle

uint16\_t conhdl;

/// length of the data

uint16\_t length;

/// broadcast and packet boundary flag

uint8\_t pb\_bc\_flag;

#if (BLE\_EMB\_PRESENT)

/// Pointer to the first descriptor containing RX Data

struct em\_buf\_node \*buf;

#else // (BLE\_HOST\_PRESENT)

/// Pointer to the data buffer

uint8\_t\* buffer;

#endif // (BLE\_EMB\_PRESENT) / (BLE\_HOST\_PRESENT)

};

注意这两个结构体的 前两个是相同的，所以在处理tx\_acl\_rdy 时候，其取到的 idx 会等于BLE\_TX\_DESC\_CNT ，而原来的 代码在这里只是会简单的free（txnode），也就是free 掉在 lld\_pdu\_data\_send 中 struct lld\_pdu\_data\_tx\_tag \*data\_tx = (struct lld\_pdu\_data\_tx\_tag\*)ke\_malloc(sizeof(struct lld\_pdu\_data\_tx\_tag), KE\_MEM\_ENV); 分配的内存，而其 buff 指向的 tx data buff node 确没有处理，最终导致系统的 tx data buff node 越来越小。

三．解决方法

在 lld\_pdu\_tx\_flush\_list 中，如果取到txnode->idx 等于BLE\_TX\_DESC\_CNT，说明是还没有和tx desc node 关联的。因此 需要按照 struct lld\_pdu\_data\_tx\_tag 来解析从list 取出来的数据。并从 struct lld\_pdu\_data\_tx\_tag \*data\_tx = (struct lld\_pdu\_data\_tx\_tag \*)txnode\_tmp;

int idx = data\_tx->buf->idx;

找到所使用的tx data buff node 的 idx，调用 em\_buf\_tx\_buff\_free 放到em\_buf\_env.tx\_buff\_free list 里，同时tx\_cnt++,最终llc\_common\_nb\_of\_pkt\_comp\_evt\_send

使得 l2cm\_env.nb\_buffer\_avail 和 em\_buf\_env.tx\_buff\_free size 保持一致。

在if(txnode->idx >= BLE\_TX\_DESC\_CNT)

{

GLOBAL\_INT\_DISABLE();

struct lld\_pdu\_data\_tx\_tag \*data\_tx = (struct lld\_pdu\_data\_tx\_tag \*)txnode\_tmp;

int idx = data\_tx->buf->idx;

#if (BLE\_TX\_BUFFER\_CNT > 0)

em\_buf\_tx\_buff\_free(idx & BLE\_BUFFIDX\_MASK);

tx\_cnt++;

ke\_free(data\_tx);

#endif //(BLE\_TX\_BUFFER\_CNT > 0)

GLOBAL\_INT\_RESTORE();

}

附：发送流程

KE\_MSG\_ALLOC\_DYN(GATTC\_SEND\_EVT\_CMD, KE\_BUILD\_ID(TASK\_GATTC, 0),…..-> gattc\_send\_evt\_cmd\_handler->atts\_send\_event->atts\_send\_pdu->l2cc\_pdu\_send(pdu);🡪 KE\_MSG\_ALLOC\_DYN(L2CC\_PDU\_SEND\_CMD, KE\_BUILD\_ID(TASK\_L2CC, conidx),🡪 l2cc\_pdu\_send\_cmd\_handler🡪co\_list\_push\_back(&(l2cc\_env[conidx]->tx\_queue), &(msg->hdr));🡪 l2cm\_tx\_status(conidx, true);🡪 ke\_event\_set(KE\_EVENT\_L2CAP\_TX);🡪 l2cm\_l2cap\_tx\_handler🡪l2cc\_data\_send🡪 KE\_MSG\_ALLOC(HCI\_ACL\_DATA\_TX, conidx, KE\_BUILD\_ID(TASK\_L2CC, conidx),

hci\_acl\_data\_tx);🡪 llc\_hci\_acl\_data\_tx\_handler🡪lld\_pdu\_data\_send🡪co\_list\_push\_back(list, &data\_tx->hdr);( struct co\_list \*list = &evt->tx\_acl\_rdy;)🡪 lld\_evt\_schedule🡪lld\_pdu\_tx\_prog🡪lld\_pdu\_send\_packet🡪lld\_pdu\_data\_tx\_push🡪lld\_pdu\_push\_back(list, (struct co\_list\_hdr \*)txnode);( struct co\_list \*list = &evt->tx\_prog;)🡪 lld\_pdu\_push\_back(&evt->tx\_acl\_tofree, &txacl->hdr)

free

lld\_evt\_deffered\_elt\_handler--> lld\_pdu\_check--> txnode = (struct em\_desc\_node \*)co\_list\_pop\_front(&evt->tx\_prog);

if (!co\_list\_is\_empty(&evt->tx\_acl\_tofree))

{

tx\_tofree = (struct lld\_pdu\_data\_tx\_tag \*)co\_list\_pop\_front(&evt->tx\_acl\_tofree);

if (tx\_tofree)

{

ke\_free(tx\_tofree);

}

}

bool lld\_pdu\_check(struct lld\_evt\_tag \*evt)

{

// Get first deferred element

struct lld\_pdu\_rx\_info \*info = NULL;

struct lld\_pdu\_data\_tx\_tag \*tx\_tofree = NULL;

struct co\_list tmp\_pdu\_list;

co\_list\_init(&tmp\_pdu\_list);

// Get first programmed tx node

struct em\_desc\_node \*txnode;

uint16\_t conhdl = evt->conhdl;

uint8\_t tx\_cnt = 0;

bool elt\_deleted = false;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\* TX Check

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

GLOBAL\_INT\_DISABLE();

txnode = (struct em\_desc\_node \*)co\_list\_pick(&evt->tx\_prog);

GLOBAL\_INT\_RESTORE();

while (txnode)

{

// Check if packet has been transmitted and acknowledged

if (ble\_txdone\_getf(txnode->idx))

{

struct em\_desc\_node \*next;

GLOBAL\_INT\_DISABLE();

next = (struct em\_desc\_node \*)(txnode->hdr.next);

//Pop the acknowledged tx packet

txnode = (struct em\_desc\_node \*)co\_list\_pop\_front(&evt->tx\_prog);

if (!co\_list\_is\_empty(&evt->tx\_acl\_tofree))

{

tx\_tofree = (struct lld\_pdu\_data\_tx\_tag \*)co\_list\_pop\_front(&evt->tx\_acl\_tofree);

if (tx\_tofree)

{

ke\_free(tx\_tofree);

}

}

GLOBAL\_INT\_RESTORE();

// clear next pointer

ble\_nextptr\_setf(txnode->idx, 0);

#if (BLE\_CENTRAL || BLE\_PERIPHERAL)

/\*\*

\* LLCP tx confirmation

\*/

if(txnode->idx < BLE\_TX\_DESC\_CNTL)

{

struct llcp\_pdu\_tag \*llcp\_elt = (struct llcp\_pdu\_tag \*)txnode;

evt->evt.conn.tx\_prog\_pkt\_cnt--;

GLOBAL\_INT\_DISABLE();

if(co\_list\_is\_empty(&evt->tx\_llcp\_pdu\_rdy))

{

// no more LLCP packet to transmit

LLD\_EVT\_FLAG\_RESET(evt, WAITING\_TXPROG);

}

ke\_free(txnode);

GLOBAL\_INT\_RESTORE();

// inform that LLCP packet has been acknowledged

llc\_pdu\_llcp\_tx\_ack\_defer(conhdl, llcp\_elt->opcode);

}

/\*\*

\* ACL tx confirmation

\*/

else if (txnode->idx > DATA\_BASE\_INDEX)

{

// If the all the chunks have been acknowledged

if(em\_buf\_tx\_free(txnode))

{

//Buffer is sent, increase the tx counter

tx\_cnt++;

}

}

/\*\*

\* ADV,SCAN\_REQ and CONNECT\_REQ tx confirmation

\*/

else

#endif // (BLE\_CENTRAL || BLE\_PERIPHERAL)

{

tx\_cnt++;

}

txnode = next;

}

else

{

// Escape from the loop

break;

}

}

#if (BLE\_CENTRAL || BLE\_PERIPHERAL)

// check if it's a connection handle

if((conhdl != LLD\_ADV\_HDL) && (tx\_cnt > 0))

{

GLOBAL\_INT\_DISABLE();

ASSERT\_INFO(evt->evt.conn.tx\_prog\_pkt\_cnt >= tx\_cnt, conhdl, tx\_cnt);

evt->evt.conn.tx\_prog\_pkt\_cnt -= tx\_cnt;

GLOBAL\_INT\_RESTORE();

llc\_pdu\_acl\_tx\_ack\_defer(conhdl, tx\_cnt);

}

#endif // (BLE\_CENTRAL || BLE\_PERIPHERAL)

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\* RX Check

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

GLOBAL\_INT\_DISABLE();

info = (struct lld\_pdu\_rx\_info \*)co\_list\_pop\_front(&lld\_evt\_env.rx\_pkt\_deferred);

GLOBAL\_INT\_RESTORE();

// load all elements in rx queue

while (info != NULL)

{

// check if descriptor is for current connection handle

if(conhdl == info->conhdl)

{

#if (BLE\_CENTRAL || BLE\_PERIPHERAL)

// check if it's a connection handle

if(info->conhdl != LLD\_ADV\_HDL)

{

llc\_pdu\_defer(info->conhdl, info->status, info->rssi, info->channel, info->length);

}

else

#endif // (BLE\_CENTRAL || BLE\_PERIPHERAL)

#if (BLE\_OBSERVER || BLE\_CENTRAL || BLE\_PERIPHERAL)

{

elt\_deleted = llm\_pdu\_defer(info->status, info->rx\_hdl, tx\_cnt);

}

#endif // (BLE\_OBSERVER || BLE\_CENTRAL || BLE\_PERIPHERAL)

// free deferred structure

ke\_free(info);

}

else

{

// put element in temp list

co\_list\_push\_back(&tmp\_pdu\_list, &(info->hdr));

}

GLOBAL\_INT\_DISABLE();

info = (struct lld\_pdu\_rx\_info \*)co\_list\_pop\_front(&lld\_evt\_env.rx\_pkt\_deferred);

// no more elements, copy temp list to defered list

if(info == NULL)

{

if(!co\_list\_is\_empty(&tmp\_pdu\_list))

{

co\_list\_merge(&lld\_evt\_env.rx\_pkt\_deferred, &tmp\_pdu\_list);

}

}

GLOBAL\_INT\_RESTORE();

}

return(elt\_deleted);

}