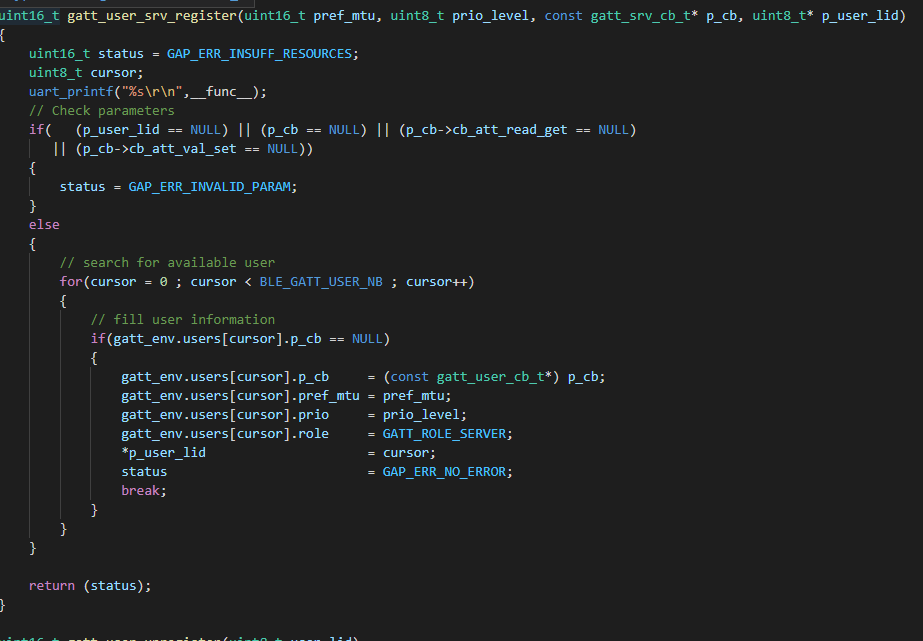
Bk3436 rx 数据之后的代码执行流程

本文以BK3436 rw 5.2版本的协议栈来分析rx 数据的代码执行流程。分析其数据是如何通过底层上报到APP层的。

此处以fee0 的服务说明数据流程，手机端通过fee2 cmd 向dev 发送了一些数据。

一、协议栈初始话时候，app 通过 GAPM\_PROFILE\_TASK\_ADD\_CMD 向协议栈注册了 fee0 的server，协议栈调用 fee0s\_init 来初始化一些cb和 添加db 到gatt svc list 中去。

在该函数中有两个关键的函数 1. gatt\_user\_srv\_register 2. gatt\_db\_svc16\_add



该函数 gatt\_env.users[cursor].p\_cb     = (const gatt\_user\_cb\_t\*) p\_cb; // p\_cb 就是 fee0s\_cb

\_\_STATIC const gatt\_srv\_cb\_t fee0s\_cb =

{

        .cb\_event\_sent    = fee0s\_cb\_event\_sent,

        .cb\_att\_read\_get  = fee0s\_cb\_att\_read\_get,

        .cb\_att\_event\_get = NULL,

        .cb\_att\_info\_get  = NULL,

        .cb\_att\_val\_set   = fee0s\_cb\_att\_val\_set,

};

而 gatt\_db\_svc16\_add 则把 fee0s的服务添加到了gatt srv 的list 中 并获取到了 srv 的start handle。

二、当手机发起连接之后，协议栈开始连接之后的一些初始化。

1. l2cap\_create

2. l2cap\_sig\_create

3.l2cap\_chan\_fix\_register(conidx, L2CAP\_CID\_LE\_SIGNALING, L2CAP\_LE\_MTU\_MIN, &(l2cap\_sig\_chan\_cb), &(p\_con->sig.chan\_lid));

4. gatt\_create

5. gatt\_bearer\_create

6.l2cap\_chan\_fix\_register(conidx, L2CAP\_CID\_ATTRIBUTE, L2CAP\_LE\_MTU\_MIN,

                                             &gatt\_bearer\_fix\_chan\_cb, &(chan\_lid));

7. gapc\_con\_create

8.gapc\_smp\_create

9.l2cap\_chan\_fix\_register(conidx, L2CAP\_CID\_SECURITY, L2CAP\_SMP\_MTU, &gapc\_smp\_chan\_cb, &(p\_con->smp.chan\_lid));

通过l2cap\_chan\_fix\_register 函数，分别对CID是 4 CID\_ATTRIBUTE，5 LE\_SIGNALING ，6 CID\_SECURITY 的 channel 分配注册了其cb 函数。

10.当完成连接以及初始化完成之后（srv discover 此处忽略该流程，其流程和接收到手机数据很类似，可自行分析），手机端向fee2 发送了 0x86，0x58 的数据。

11.连接上之后 llc\_start —> lld\_con\_start🡪 evt->cb\_start            = &lld\_con\_evt\_start\_cbk;-🡪

prog\_par.frm\_cbk        = &lld\_con\_frm\_cbk;

lld\_con\_frm\_cbk 是evt 结束之后的cb，在收到数据后 会最终调用 lld\_con\_rx

12.收到数据后 通过解析 rxllid 来区分是LLID\_CNTL还是 ACL 数据，这里我们是ACL的数据 则

KE\_MSG\_ALLOC(LLD\_ACL\_RX\_IND, KE\_BUILD\_ID(TASK\_LLC, link\_id), TASK\_NONE, lld\_acl\_rx\_ind);

                msg->em\_buf   = rxdataptr;

                msg->data\_len = rxlength;

                msg->llid     = rxllid;

                ke\_msg\_send(msg);

把该数据通过LLD\_ACL\_RX\_IND 发送给TASK\_LLC,到 lld\_acl\_rx\_ind\_handler函数执行。

13. lld\_acl\_rx\_ind\_handler 函数里又通过 HCI\_ACL\_DATA 将消息 hci\_send\_2\_host

到l2cap\_hci\_acl\_data\_handler

14. l2cap\_hci\_acl\_data\_handler 🡪 l2cap\_chan\_ll\_rx\_acl 当一包完整的数据收完之后，就 通过p\_chan->p\_cb->cb\_sdu\_rx 去处理数据，此处的cb\_sdu\_rx 则是我们之前在l2cap\_chan\_fix\_register 中注册的 gatt\_bearer\_fix\_chan\_cb🡪 gatt\_bearer\_sdu\_rx\_cb 、、

15. gatt\_bearer\_sdu\_rx\_cb 对数据进行处理后，gapc\_sdt\_prepare(&(p\_sdu\_meta->defer), conidx, GAPC\_SDT\_GATT\_BEARER);gapc\_sdt\_defer🡪 co\_djob\_reg(&(gapc\_env.sdt\_job));

16. void co\_djob\_reg(co\_djob\_t\* p\_djob)

{

    // check if job is not already present in the queue

    if(p\_djob->hdr.next == NULL)

    {

        // Set event only if no job present in queue

        if (co\_list\_is\_empty(&(djob\_env.job\_queue)))

        {

            ke\_event\_set(KE\_EVENT\_DJOB);

        }

        // Add a critical section as function can be called under interrupt context

        GLOBAL\_INT\_DISABLE();

        // Put element at end of the list

        co\_list\_push\_back(&(djob\_env.job\_queue), &(p\_djob->hdr));

        GLOBAL\_INT\_RESTORE();

    }

}

17. void co\_djob\_init(uint8\_t init\_type)

{

    switch (init\_type)

    {

        case RWIP\_INIT:

        {

            // Register BLE delayed job kernel event

            ke\_event\_callback\_set(KE\_EVENT\_DJOB, &co\_djob\_evt\_handler);

        }

        // no break;

        case RWIP\_1ST\_RST:

        case RWIP\_RST:

        {

            // remove all delayed job pending

            co\_list\_init(&(djob\_env.job\_queue));

        } break;

        default: { /\* Do nothing \*/ } break;

    }

}

18. \_\_STATIC void co\_djob\_evt\_handler(void)

{

    DBG\_FUNC\_ENTER(co\_djob\_evt\_handler);

    co\_djob\_t \*p\_djob;

    // Add a critical section as list can be accessed under interrupt context

    GLOBAL\_INT\_DISABLE();

    p\_djob = (co\_djob\_t \*)co\_list\_pop\_front(&(djob\_env.job\_queue));

    // Check if another job must be executed

    if (co\_list\_is\_empty(&(djob\_env.job\_queue)))

    {

        // Clear event

        ke\_event\_clear(KE\_EVENT\_DJOB);

    }

    GLOBAL\_INT\_RESTORE();

    if (p\_djob != NULL)

    {

        p\_djob->hdr.next = NULL;

        p\_djob->cb(p\_djob->p\_env);//此处调用处理函数

    }

    DBG\_FUNC\_EXIT(co\_djob\_evt\_handler);

}

19. void gapc\_sdt\_init(uint8\_t init\_type)

{

    switch (init\_type)

    {

        case RWIP\_RST:

        case RWIP\_1ST\_RST:

        {

            // initialize Djob and native timer

            co\_djob\_prepare(&(gapc\_env.sdt\_job), gapc\_sdt\_djob\_handler, NULL);

            co\_time\_timer\_init(&(gapc\_env.sdt\_timer), gapc\_sdt\_timer\_handler, NULL);

        } break;

        default:  { /\* Do nothing \*/ } break;

    }

}

18. const gapc\_sdt\_handler\_t gapc\_sdt\_handlers[GAPC\_SDT\_MAX] =

{

    [GAPC\_SDT\_SMP]         = { gapc\_smp\_timeout\_handler,             NULL                          },

    #if (BLE\_L2CAP)

    [GAPC\_SDT\_L2CAP]       = { l2cap\_sig\_trans\_timeout\_handler,      l2cap\_sig\_req\_start\_handler   },

    #endif // (BLE\_L2CAP)

    #if (BLE\_GATT)

    [GAPC\_SDT\_GATT\_PROC]   = { gatt\_proc\_trans\_timeout\_cb,           gatt\_proc\_continue\_defer\_cb   },

    [GAPC\_SDT\_GATT\_BEARER] = { gatt\_bearer\_eatt\_estab\_timer\_handler, gatt\_bearer\_sdu\_rx\_bg\_handler },

    #endif // (BLE\_GATT)

};

19.最后调用到 gatt\_bearer\_sdu\_rx\_bg\_handler –->gatt\_proc\_handler\_create 因为我们的OPCODE 是 L2CAP\_ATT\_WR\_CMD\_OPCODE== 0x52 ，所以 🡪 gatt\_srv\_write\_proc\_create

{// Allocate procedure

    status = gatt\_proc\_handler\_alloc(conidx, proc\_id, sizeof(gatt\_srv\_write\_proc\_t),

                                     (gatt\_proc\_cb) gatt\_srv\_write\_proc\_continue, pp\_proc);

    \*p\_pdu\_hdl\_cb = (gatt\_proc\_pdu\_handler\_cb) gatt\_srv\_write\_l2cap\_att\_wr\_req\_handler;

}

在这里面 给 gatt\_proc\_handler\_create给 cb\_pdu\_handler 指向了 gatt\_srv\_write\_l2cap\_att\_wr\_req\_handler ，并在之后 proc\_status = cb\_pdu\_handler(conidx, (gatt\_proc\_t\*) p\_proc\_handler, p\_pdu, p\_sdu, p\_bearer->mtu);

调用了它。

在 处理完gatt\_srv\_write\_l2cap\_att\_wr\_req\_handler之后，程序继续往下执行了gatt\_proc\_continue(conidx, p\_proc, GATT\_PROC\_PDU\_RX, proc\_status); 🡪gatt\_proc\_continue\_exe(conidx, p\_proc, state, status);🡪  p\_proc->cb\_continue(conidx, p\_proc, state, status);此处的cb\_continue 就是gatt\_srv\_write\_proc\_create中的 gatt\_srv\_write\_proc\_continue

20.在gatt\_srv\_write\_proc\_continue 中通过 gatt\_db\_att\_get

p\_proc->hdl 拿到了p\_svc 指针内容

21. 之后gatt\_user\_t\* p\_user = gatt\_user\_get(p\_proc->svc\_user\_lid);拿到之前fee0s 初始化时候的 信息，之后  p\_user->p\_cb->srv.cb\_att\_val\_set(conidx, p\_proc->hdr.user\_lid, p\_proc->hdr.token, p\_proc->hdl,  p\_proc->offset, p\_proc->p\_buf); 此处的 cb\_att\_val\_set 指向的是

\_\_STATIC const gatt\_srv\_cb\_t fee0s\_cb =

{

        .cb\_event\_sent    = fee0s\_cb\_event\_sent,

        .cb\_att\_read\_get  = fee0s\_cb\_att\_read\_get,

        .cb\_att\_event\_get = NULL,

        .cb\_att\_info\_get  = NULL,

        .cb\_att\_val\_set   = fee0s\_cb\_att\_val\_set,

};

22.调用 fee0s\_cb\_att\_val\_set 就到了我们的profile 中间层，进一步把消息通过cb\_att\_data\_upd

/// Default Message handle

\_\_STATIC const fee0s\_cb\_t fee0s\_msg\_cb =

{

    .cb\_fee0s\_value\_upd\_cmp = fee0s\_cb\_value\_upd\_cmp,

    .cb\_att\_data\_upd      = fee0s\_cb\_att\_data\_upd,

    .cb\_bond\_data\_upd = fee0s\_cb\_bond\_data\_upd

};

fee0s\_cb\_att\_data\_upd 函数则是把消息通过 FEE0S\_FEE2\_WRITER\_CMD\_IND 通知给到app层。

23.app 处理数据后，调用了gatt\_proc\_continue(conidx, (gatt\_proc\_t\*) p\_proc, GATT\_PROC\_USER\_CFM, status);

🡪 gatt\_proc\_continue\_exe(conidx, p\_proc, state, status);又到gatt\_srv\_write\_proc\_continue

case GATT\_PROC\_USER\_CFM:

gatt\_bearer\_rx\_continue(conidx, p\_proc->bearer\_lid);

finished = true;

最后

if(finished)

    {

        // check if buffer must be released

        if(p\_proc->p\_buf != NULL)

        {

            co\_buf\_release(p\_proc->p\_buf);

        }

        // Pop Procedure

        gatt\_proc\_pop(conidx, (gatt\_proc\_t\*) p\_proc, true);

    }

到此处，一个完整的rx 流程结束。