电池log系统开发和维护指南

by 张鲜，zhangxian@zerozero.cn

1 目的

了解电池与测试平台的通信协议，完成固件升级开发和心跳包开发。**现以心跳包为最高优先级**。

2 思路

1. 固件升级：按钮触发，电池写入。即用户在上电falcon后，触发某按钮，电池执行写入操作，相当于重刷ipk，完成升级；
2. 心跳包：从log中观察心跳不稳现象，从代码找原因。

3 方法

固件升级：

1. 用户按钮触发，通知电池重启，即boot操作；
2. 电池请求接收固件；
3. 数据线自动传固件升级包到电池内。

心跳包：

1. 心跳不稳的现象是什么；
2. 搞清楚心跳程序逻辑，结合log看问题出哪儿了。

4 实施

4.1 基础概念

【1】字节数

假设一请求命令：$\mathrm{AA\ 01\ 00\ 0C\ 00\ 01\ 00\ 00\ 00\ 00\ 01\ 00\ 04\ 55}$

一个字节数指：XX，例如上面命令的AA就是一个字节数。

两个字节数指：XX XX，例如上面命令的 01 00就是两个字节数。

【2】不加返回码的数据包（Package）格式

$$\begin{aligned}\mathrm{Package} &= \mathrm{SOF} + \mathrm{Protocol} + \mathrm{ID} + \mathrm{Length} + \mathrm{Tag} + \\ &= \mathrm{Master\_Package\_ID} + \mathrm{Slave\_Package\_ID} + \mathrm{CMD\_ID} + \mathrm{Data} + \mathrm{CRC} + \mathrm{EOF}\end{aligned}$$

其中

域	字节数	详细描述
`SOF`	1	数据包的起始字节，固定为 0xAA，0x **代表16进制**
`Protocol ID`	2	协议的版本号
`Length`	2	数据包长度，包括整个数据包的所有字节 (From SOF to CRC-8) 注：LSB在前，MSB在后
`Tag`	1	标识码:标识协议通信的对象
`Master Package ID`	2	主机已发送的协议包的个数
`Slave Package ID`	2	从机发送过来的协议包的个数
`CMD ID`	1	命令码，表示具体的操作请求
`Data`	n(0~1011)	数据域，对请求或者应答内容的具体描述。应答包至少有1个字节，而且第一个字节固定为返回码，用于表示请求操作的执行结果。
`CRC-8`	1	数据包校验码，是前面所有字节（从`SOF`至Data）的`CRC-8`校验值。 注1：如果接收端发现数据包的校验码出错，会将数据包将直接扔掉，不做任何处理。
`EOF`	1	数据包结束字节，固定为`0x55`

【3】加返回码的数据包（Package）格式

$$\begin{aligned}\text{Package} &= \text{SOF} + \text{Protocol} + \text{ID} + \text{Length} + \text{Tag} + \text{Master\_Package\_ID} + \\ &= \text{Slave\_Package\_ID} + \text{CMD\_ID} + \text{Return\_Code} + \text{Data} + \text{CRC} + \text{EOF}\end{aligned}$$

【4】标识码 Tag

用来规定交互对象，如

Tag	说明	备注
0x01	智能电池与飞机端	飞机到电池
0x02	智能电池与充电器	电池到充电器
0x04	智能电池与测试平台	测试平台到电池
0x05	智能电池报警信息	电池到各个平台（充电器除外）

例如，要获取电池的动态和静态信息，显然是第一行 case；**固件升级是第三行case，即**0x04

【5】协议 Protocol

定义：双方完成通信所必须遵循的规则和约定。

例如：A按要求发送请求命令，该命令以规定的数据包格式发送（即【2】和【3】中所示）；B收到后返回数据，返回数据的格式可能不同于A发送的数据包格式。

示例：

A：AA 01 00 0C 00 01 00 00 00 00 02 00 4F 55

B：AA 01 00 25 00 01 1C 01 57 03 02 00 8F 0B E8 28 00 00 01 00 01 19 00 FF FF 9E 0D A1 0D AA 0D 00 00 00 00 00

注意：**除去AA和55剩下的就是代表数据的字节**，只不过传输出于高效目的，采用字节表示。由于协议在通信中属于最难的部分，下面给予重点讲解。

问题1：为什么是`0xAA`开头和`0x55`结尾呢？

答：方便数据传输过程中的异常检测。因为AA转为二进制展开为10101010，55展开为01010101，变成串行电平的话是一个占空比为50%的方波，这种方波在电路中最容易被分辨是否受干扰或者畸变，从而判断传输是否出现问题。

问题2：如何区分不同的协议？

答：主要**根据数据包里的** Tag **和** $\mathrm{CMD\_ID}$ **两项区分不同协议**，次要的是数据长度，校验码无法区分，它用来判断发送方和接收方数据包传输时是否接收一致。

4.2 通信协议

4.2.1 操作前提

实现通信协议的前提是要确保发送方和接收方之间的串口保持开通，否则无法初始化串口协议。

该操作需用户提供

- 串口地址：例如/dev/ttyHSL1，其中/dev是Linux系统下的一个挂载点；
- 波特率：衡量数据传送速率的指标。在信息传输通道中，携带数据信息的信号单元叫码元，每秒钟通过信道传输的码元数称为码元传输速率，简称波特率；
- 校验位：用于通信时的错误检测N为没有校验位，O为奇校验，E为偶检验；
- 数据位：数据的有效信息，位数一般是6位、7位或8位，传输数据时先传送字符的低位，后传送字符的高位，即低位（LSB）在前，高位（MSB）在后；
- 停止位：标志一个字符传送的结束。

示例：

```
serial_open(devpath, 115200, 'N', 8, 1)
```

其中：串口地址是devpath（即/dev/ttyHSL1），波特率是115200， N是无校验，8是数据位，停止位是1。

4.2.2 协议的数据结构

定义：不同作者写法不同，我司主要是标识符，线程池，数据链表，解析器，回调函数组成。

例如：

```
typedef struct {
  bool isrunning;  // running status of threads
  bool isupgrade;  // is the process of upgrading
  int dev_fd;
  /* threads */
  struct zz_thread *send_thread;
  struct zz_thread *poll_thread;
  struct zz_thread *recv_thread;
  /* signals */
  struct zz_signal_type *send_signal;
  struct zz_signal_type *recv_signal;
  /× 解析器 */
  bat_packet_parser_t parser;
  /* 线程池 */
  object_pool_t *packet_pool;       // memory poll for packets
  object_pool_t *packet_frag_pool;  // memory poll for fragments
  /* 数据链表 */
  struct zz_mutex *recv_lock;
  bat_packet_list_head_t recv_pkg_list;  // receive data fragment list
  struct zz_mutex *send_lock;
  bat_packet_list_head_t send_pkg_list;  // send data fragment list
  /* 回调函数 */
  void (*alarm_cb)(void *param);   // callback when alarm occurred
  void (*notice_cb)(void *param);  // callback when alarm occurred
} bat_protocol_t;
```

4.2.2 协议初始化

【1】分配协议数据的内存空间

定义：`static bat_protocol_t bat_protocol`，分配内存如下

```
memset(&bat_protocol, 0, sizeof(bat_protocol_t));
```

【2】初始化解析器

```
bat_packet_parser_init(&bat_protocol.parser)  //方法是memset
```

【3】初始化线程池

这个地方不懂

```
bat_protocol.packet_pool = object_pool_new(sizeof(bat_packet_t), 20, 1, 1);
bat_protocol.packet_frag_pool = object_pool_new(sizeof(bat_packet_fragment_t), 20, 10, 1);
```

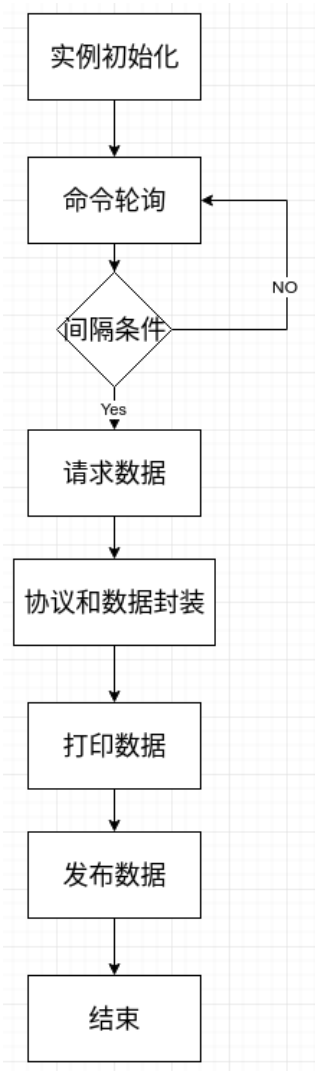
【4】初始化数据链表

```
bat_protocol.recv_pkg_list.num = 0;
INIT_LIST_HEAD(&bat_protocol.recv_pkg_list.list);
```

`INIT_LIST_HEAD`是个双向链表，利用库函数完成。

4.3 程序逻辑

基于4.1和4.2小节的概念和基本操作，下面是电池程序的整体实现逻辑（伪代码演示）。大体流程见下图



4.3.1 实例初始化

实例定义：服务器状态，通信协议，串口文件描述符，实时动态数据，电池静态数据，电池瞬间动态数据，更新标识符。

```
typedef struct {
    bat_server_state_t state;
    bat_protocol_t *bat_protocol;
    int bat_uart_fd;
    struct zz_bufresher *bat_dynamic_info;
    battery_static_info static_info;
    battery_dynamic_info instant_dynamic_info;
    int discharge_info_updated;
    int product_info_updated;
    int history_info_updated;
} bat_instance_t;
```

初始化过程就是对bat_instant_t声明变量（结构体型）的成员变量（同上面定义的）分配内存。

4.3.2 命令轮询

作用：两个，一个是电池检查客户端发来的命令，从而根据不同的命令准备不同的协议；另一个是因上一节初始化结束，所有动态信息和静态信息都有了各自的内存。当轮询到期后，为数据填充做准备。

参考：

[1] https://nanomsg.org/v1.1.2/nn_reqrep.html

[2] https://blog.csdn.net/youyou519/article/details/103026959 (nn_errno()函数返回值对应的错误提示)

[3] https://nanomsg.org/v0.1/nn_errno.3.html (为什么用nn_errno)

[4] https://blog.csdn.net/gyd0311/article/details/12944977?utm_source=blogxgwz2（如何打印nn_errno()返回值对应的错误提示）

函数：

```
int bat_command_service_pollwait(bat_instance_t, battery_static_info,battery_dynamic_info)
```

因4.3.1实例初始化完成，故开启命令轮训。其逻辑是请求和应答（返回数据）模式。

关键代码分析

【1】文件描述符 file descriptor, fd

```
fd = nn_socket(AF_SP, NN_REP);
```

NN_REP：表示主机和客户端的交互属于请求/应答模式[1]。因代码是batter_service的，所以电池是主机，收到请求后要发送replies；falcon部件与电池交互的设备是客户端，用来发电池发request。

【2】bind

```
int ret = nn_bind(fd, BATTERY_SRV_CMD_SOCKET_ADDR)
```

作用：把文件描述符fd和BATTERY_SRV_CMD_SOCKET_ADDR绑定起来，从而让fd可以监听后者所描述的ip地址和端口号。

注意：nn_strerror(nn_errno())是用来打印nn_errno()返回值对应的错误提示，一般用 %s.

【3】nn_poll 函数

背景：使用了nn_poll函数，意味着原作者掌握了高阶的C/S TCP模型的通信，好处是高效，不用写accept, connect, listen, recv或recvfrom等这种习惯性写法。

第一步：数据准备好（电池将外部数据从其I/O设备拷贝到内核缓冲区）

怎么做：用nn_poll函数设置文件描述符的功能和事件

```
struct nn_pollfd pfd;  //pfd是个数组
memset(&pfd, 0, sizeof(struct nn_pollfd));
pfd.fd = fd;
pfd.events = NN_POLLIN; //读事件
ret = nn_poll(&pfd, 1, BATTERY_SERVER_POLLTIME);
```

注意：

- 1：代表 pfd文件描述符数组只有一个，默认是监听描述符，用来监听所监听的描述符的读事件是否成功；
- ret：0 代表超时（BATTERY_SERVER_POLLTIME规定 事件长度）；> 代表成功；<0 代表失败。

第二步：电池将内核缓冲的数据拷贝到进程

```
int datalen;
char *recvbuf = NULL;
if (pfd.revents & NN_POLLIN) > 0 //才行
datalen = nn_recv(fd, &recvbuf, NN_MSG, 0)；
```

第三步：电池处理进程中的数据

```
battery::BatteryCommandAck commandack;
bat_command_service_proc(bat_inst, static_info, dynamic_info, recvbuf, datalen, commandack){
  battery::BatteryCommand command;
  battery::BatteryInfo battery_info;
  if (!command.ParseFromArray(recvbuf, recvsize)) > 0 才行 //此步指主机解析recvbuf
  commandack.set_id(command.id());
  switch (command.id()) {
    case battery::BATTERY_COMMAND_COULOMETER_UPGRADE:
    case battery::BATTERY_COMMAND_KEEPALIVE:
  }
  //如果上述OK
  commandack.set_rc(0);  //电池确认成功后设置 return code rc = 0
  bat_info_pb_fill(bat_inst, &battery_info, static_info, dynamic_info); //设置proto字段，生成
prototxt
  commandack.mutable_info()->CopyFrom(battery_info);
  ret = nn_send(fd, commandack.SerializeAsString().c_str(), commandack.ByteSize(),0)
}
```

注意：`commandack.mutable_info()->CopyFrom(battery_info);`函数有两层目的

1. pb_fill里已完成battery_info充填，传递给commandack，方便下次循环,数据因轮询，每隔10次没有更新时bat_command_service_proc程序执行；
2. 处理后，将commandack序列化，再发给电池，例如`nn_send(....)`函数所示

4.3.4 请求数据

背景：当轮询的次数（10次）到了，静态信息在 4.3.1 节实例初始化中发送过请求，因发送请求模式同动态信息，以动态信息请求为主要示例。动态信息实时变化，例如放电和告警信息，每隔10次需发送一次命令请求，电池接收并确认了才能根据请求的命令进行操作。

命令请求的含义：对于我们现有客户端而言，本质就是log或者将log中的数据以socket形式发布出来，供其他应用程序操作。

动态信息发送命令请求：

```
battery_discharge_info_request(bat_inst, 0)
bat_protocol_send_packet( device->bat_protocol, PKG_TAG_DRONE,
                 PKG_DRONE_CMD_DISCHARGE_DATA, NULL, 0, timeout )  //timeout = 0 返回值
```

`bat_protocol_send_packet` 返回值为下面说明发送成功

```
PKG_ACK_SUCCESS = 0x00  //十六进制0x00代表十进制0
```

发送的命令请求包含协议，协议操作见 4.3.5 节。

4.3.5 协议和数据封装

背景：当接收到请求命令时，电池会对该命令进行处理。

处理函数原型：

```
frame_ack_type_t bat_protocol_send_packet(bat_protocol_t *protocol, uint8_t tag,
                                          uint8_t cmdid, uint8_t *data,
                                          uint16_t len, uint16_t timeout)
```

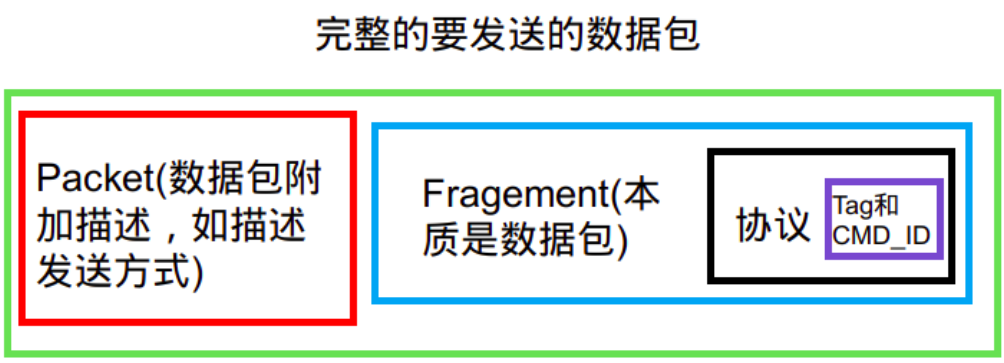
具体操作：

```
bat_packet_t *packet;           //声明一个双向链表，未初始化
bat_packet_fragment_t *frag;    //声明数据包（规定了帧格式），也是双向链表
int timer_cnt = 0;
frame_ack_type_t ret = PKG_ACK_UNDEFINED_ERROR;

#1 分配内存给线程池，并初始化双向链表
packet = bat_packet_new(){
  ppkg = object_pool_alloc(bat_protocol.packet_pool);
  memset(ppkg, 0, sizeof(bat_packet_t));
  INIT_LIST_HEAD(&ppkg->list);
  INIT_LIST_HEAD(&ppkg->frag_list.list);
  INIT_LIST_HEAD(&ppkg->ack_frag_list.list);
  ppkg->status = BAT_PACKET_STATUS_INIT;
  ppkg->iswaitack = true;
  ppkg->t_start = zz_time_in_usec() / 1000;
  ppkg->t_acktimeout = BAT_PKG_ACK_TIMEOUT;
}

#2 限制数据长度
len = len > BAT_PKG_DATA_MAX_SIZE ? BAT_PKG_DATA_MAX_SIZE : len;

#3 为数据包分配内存+填充4.1小节提到的数据包Package
frag = bat_packet_frag_new()
frag->frame.tag = tag;
frag->frame.cmd_id = cmdid;
frag->frame.length = BAT_PKG_HEADER_SIZE + len + BAT_PKG_CRC_SIZE;
memcpy(frag->frame.data, data, len);
frag->frame.crc8 = crc8((uint8_t *)&frag->frame,frag->frame.length - BAT_PKG_CRC_SIZE);

#4 相当于给数据包添加描述
bat_packet_add_frag(packet, frag){
    bat_packet_list_add(&packet->frag_list, &frag->list){
        bat_packet_list_add(&packet->frag_list, &frag->list){
            list_add(node, &head->list);
            head->num++;
        }
    }
}

#5 设置发送模式（阻塞还是非阻塞，但实际还没发送）
   //选项3.1：如果timeout == 0, 发送模式设置为立即发送
   bat_packet_send_list_add(packet);

   //选项3.2：如果time >0, ge
   bat_packet_send_list_add(packet);
   usleep(10 * 1000);  //ms in time unit
   packet->status == BAT_PACKET_STATUS_RECEIVED //判断电池是否接收到客户端发来的请求
   ret = bat_packet_ack_proc(protocol, packet); //数据包封装头部，设置为等待发送模式
```

小节：

- 在应用层对数据包的头部进行封装
- 按照数据包格式对包进行充填
- 根据timeout设置发送模式

备注：协议和数据包以及包头封装的关系如下图



4.3.6 打印数据

背景：当协议和发送模式设置好后，下面是获取数据并打印到Log中。

函数原型

```
tatic int bat_get_share_info(bat_instance_t *bat_inst,struct battery_static_info
*static_info, struct battery_dynamic_info *dynamic_info){
    get_battery_discharge_info(bat_inst, dynamic_info);
    get_battery_product_info(bat_inst, static_info);
}
```

以`get_battery_discharge_info`函数为例，操作是

```
bat_instance_t *temp = (bat_instance_t *)bat_handle;
zz_bufresher_get(temp->bat_dynamic_info, &temp->instant_dynamic_info);
# 以打印温度为例，其他略
ZZLOGD("temperature: %d", temp->instant_dynamic_info.discharge_info.temperature);
memcpy(info, &temp->instant_dynamic_info, sizeof(battery_dynamic_info));
```

即电池实时更新数据，然后打印，再恢复初始值。

4.3.7 发布数据

作用：相当于电池对客户端请求的应答。

在打印log成功后，发布数据供其他应用程序执行。至于先发布还是先打印Log取决于用户设置。具体函数见

```
bat_publisher_service_poll(bat_inst, &static_info, &dynamic_info){
    char sendbuf[BATTERY_NN_MSG_MAX_SIZE]; //声明即将要发送的数据缓冲
    fd = nn_socket(AF_SP, NN_PUB);
    nn_bind(fd, BATTERY_SRV_PUB_SOCKET_ADDR)
    bat_info_pb_fill(bat_inst, &battery_info, static_info, dynamic_info); //实时数据充填
    battery_info.SerializeToArray(sendbuf, BATTERY_NN_MSG_MAX_SIZE) //序列化数据
    ret = nn_send(fd, sendbuf, battery_info.ByteSize(), 0);  //发布数据
    battery_publish_info(static_info, dynamic_info);  //附带更新下其他数据并发布
}
```

小节：

- 准备套接字描述符并绑定发布地址；
- 发布前更新数据；
- 发布数据。

4.4 心跳包不稳

4.4.1 心跳包定义

心跳包：一种检测交互双方是否断线的方法。

4.4.2 心跳不稳现象

心跳不稳现象：

- 描述1：电池心跳开启，电池时发时不发，log记录时有时无；
- 描述2：电池心跳开启，电池一直发心跳log，正常情况下时间间隔为1秒，现在时而1秒，时而其他。

解决方案：

从两方面验证

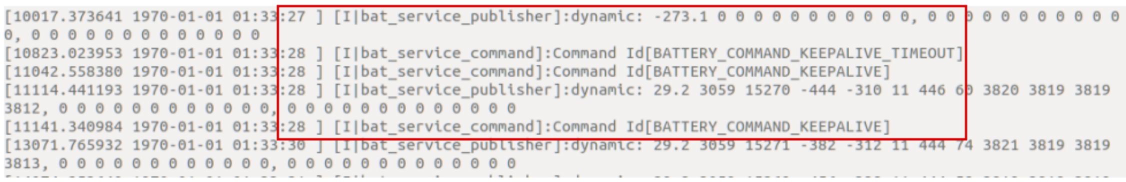
- 心跳是否开启；
- 时间间隔是否稳定。

4.4.3 实际`log`表现

根据小节 4.3 , 下载代码（已验证过，能为B2电池正确输出放电，告警，产品和历史信息）

http://git.zerozero.cn/Falcon/modules/battery_service/tree/zx/update_yl_battery_service

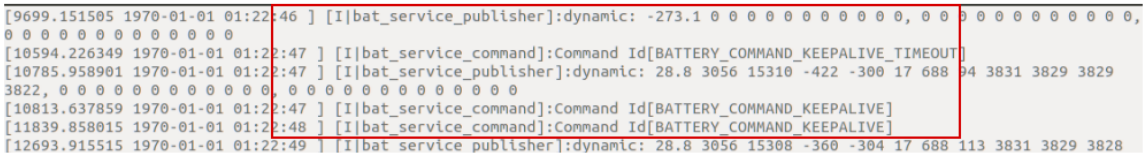
情况1：不做任何改动，将Falcon的repo编译，刷ipk，打印log，截图如下



情况2：在`bat_service.cpp`文件中，添加两行代码

```
EXPORT_C int bat_service(bat_instance_t *bat_inst) {
  int reqtimer = 0;
  int pubtimer = 0;
  int product_history_reqtimer = 0;
  struct battery_static_info static_info;
  struct battery_dynamic_info dynamic_info;

  /* init battery share info */
  memset(&static_info, 0, sizeof(static_info));
  memset(&dynamic_info, 0, sizeof(dynamic_info));

  /*
   * Now we can just publish results.  Note that there is no explicit
   * accept required.  We just start writing the information.
   */
  int battery_cycling_time = 0;
  bool is_battery_keepalive_on = true;                  //添加第一行代码
  battery_keepalive(bat_inst,is_battery_keepalive_on); //提加第二个代码
  while (true) {
    if (bat_inst->state == BAT_SERVER_STATE_POLL) {
      if (BATTERY_PUB_INFO_TIME <= (BATTERY_SERVER_POLLTIME * ++pubtimer)) {
        if (!bat_get_share_info(bat_inst, &static_info, &dynamic_info,
                                battery_cycling_time)) {
          bat_publisher_service_poll(bat_inst, &static_info, &dynamic_info);
```

其他不变，编译repo，刷ipk，打印log，截图如下



4.4.4 验证心跳开启

不论有无添加的代码，log都会出现1次 BATTERY_COMMAND_KEEPALIVE_TIMEOUT和两次 BATTERY_KEEPALIVE。

问题2：心跳开启还是没有开启呢？

答：从log上看，默认强制开启。因为添加的代码不管有无，均不影响。如果是这样，原因我认为在

bat_command_service_pollwait(...)函数里的recvbuf

```
bat_command_service_pollwait(...){  //bat_service.cpp发起轮询的地方
    ...
    datalen = nn_recv(fd, &recvbuf, NN_MSG, 0);  //根本原因
    ...
    bat_command_service_proc(...){
    if (!command.ParseFromArray(recvbuf, recvsize)) {
        ZZLOGE("Battery command data parse fail!");
        return;
    }
    commandack.set_id(command.id());
    switch (command.id()) {
        case battery::BATTERY_COMMAND_COULOMETER_UPGRADE:
            ZZLOGW("Command Id[BATTERY_COMMAND_COULOMETER_UPGRADE]");
            bat_inst->state = BAT_SERVER_STATE_UPGRADE;
            sleep(1);
            ret = battery_coulometer_upgrade(...);
            sleep(1);
            bat_inst->state = BAT_SERVER_STATE_POLL;
            break;
        case battery::BATTERY_COMMAND_KEEPALIVE: //间接原因
            ZZLOGI("Command Id[BATTERY_COMMAND_KEEPALIVE]");
            ret = battery_keepalive(bat_inst, command.keepalive_enable());
            break;
        case battery::BATTERY_COMMAND_KEEPALIVE_TIMEOUT:  //间接原因
            ZZLOGI("Command Id[BATTERY_COMMAND_KEEPALIVE_TIMEOUT]");
            ret = battery_keepalive_timeout(bat_inst, command.keepalive_timeout());
            break;
        case battery::BATTERY_COMMAND_REPORT_FLY_STATE:
            ZZLOGI("Command Id[BATTERY_COMMAND_REPORT_FLY_STATE]");
            ret = battery_report_fly_state(bat_inst, command.isflying());
            break;
            ...
        default:
            // default false: fail
            break;
        } //end of switch
    } //end of bat_command_service_proc(...)
} //end of bat_command_service_pollwait(...)
```

即 command第一次解析时，总有BATTERY_COMMAND_KEEPALIVE_TIMEOUT这个条件满足，且只满足一次；而这个条件会满足两次，故在代码添加那两行没用。而能让这个条件满足，就是`recvbuf`。

4.4.5 验证时间间隔

情况1和情况2红框中BATTERY_COMMAND_KEEPALIVE对应的的几个数字（冒号后单位为秒）有变化

情况1：无任何改动

```
:27 [bat_service publisher]
:28 [BATTERY_COMMAND_KEEPALIVE_TIMEOUT]
:28 [BATTERY_COMMAND_KEEPALIVE]
:28 [dynamic]
:28 [BATTERY_COMMAND_KEEPALIVE]
```

情况2：添加两行代码

```
:46 [bat_service publisher]
:47 [BATTERY_COMMAND_KEEPALIVE_TIMEOUT]
:47 [bat_service publisher]
:47 [BATTERY_COMMAND_KEEPALIVE]
:48 [BATTERY_COMMAND_KEEPALIVE]
```

上述心跳的间隔前者是$28-28=0$秒，后者是$48-47=1$秒，我认为和时间打印机制有关。1秒之内，或间隔一秒都是正确的。经和姚仑确认，心跳的间隔分钟级以内都是正常的。此外，从程序逻辑流程图和`bat_service.cpp`代码里来看，轮询时没有错误代码。

4.5 固件升级

4.5.1 基础概念

固件：英文名为 firmware，指的是具有软件功能的硬件。升级就是改善软件功能，拓宽硬件的使用范围和使用精度和效率。

库仑计：一种可编程数字电表，采用先进的微处理器进行智能控制，对输入的信号经过CPU运算处理后，输出当前电池电量的设备。

`MCU`：microcontroller unit 单片微型计算机(Single Chip Microcomputer)或者单片机，即缩小版的CPU。

升级：以现有的Falcon B2电池为例，固件升级只包含**`mcu`和库仑计两方面**的升级，取决于用户的选择。

4.5.2 客户端发送请求

Falcon的repo编译后，将`/build/modules/battery_service/tests/`目录下的可执行文件`test_command`推入到飞机的`/hover/tests/`目录。然后在飞机端执行升级命令来触发。

4.5.3 服务端回应请求

命令轮询：电池作为服务端，监听自身`BATTERY_SRV_CMD_SOCKET_ADDR`地址所收到的套接字描述符socket，其中就包含客户端发来的升级请求命令。

```
bat_command_service_pollwait(bat_inst, &static_info, &dynamic_info){
    struct nn_pollfd pfd;
    fd = nn_socket(AF_SP, NN_REP);
    nn_bind(fd, BATTERY_SRV_CMD_SOCKET_ADDR);  //关键地址
    memset(&pfd, 0, sizeof(struct nn_pollfd));
    pfd.fd = fd;
    pfd.events = NN_POLLIN;
    ret = nn_poll(&pfd, 1, BATTERY_SERVER_POLLTIME);
    char *recvbuf = NULL;
    datalen = nn_recv(fd, &recvbuf, NN_MSG, 0);
    battery::BatteryCommandAck commandack;
    bat_command_service_proc(bat_inst, static_info, dynamic_info, recvbuf,
                             datalen, commandack); //关键函数
    ret = nn_send(fd, commandack.SerializeAsString().c_str(), commandack.ByteSize(),0);
    nn_freemsg(recvbuf);
}
```

关键函数是`bat_command_service_proc(...)`，作用是根据不同命令进行相应的升级

```
bat_command_service_proc(...){
```

```
    battery::BatteryCommand command;
    command.ParseFromArray(recvbuf, recvsize);  //recvbuf通过zzrefresher实例化过
    commandack.set_id(command.id());
  switch (command.id()) {
    case battery::BATTERY_COMMAND_COULOMETER_UPGRADE:
      ...
      ret = battery_coulometer_upgrade(bat_inst,command.upgrade_file_path().c_str());
      ...
    case battery::BATTERY_COMMAND_MCU_UPGRADE:
      ...
      ret = battery_mcu_upgrade(bat_inst, command.upgrade_file_path().c_str());
      ...
    default:
      break;
  }
  if (ret) {
      ...
      bat_info_pb_fill(bat_inst, &battery_info, static_info, dynamic_info);
      commandack.mutable_info()->CopyFrom(battery_info);
    }
  } else {
    commandack.set_rc(-1);
  }
}
```

必备条件

根据`battery_coulometer_upgrade(bat_inst,command.upgrade_file_path().c_str());`和`battery_mcu_upgrade(bat_inst, command.upgrade_file_path().c_str());`函数可知，两个必备条件

1. 升级的类型：`mcu`还是`coulometer`；
2. 升级的路径：我理解为firmware file path；

4.5.4 升级的具体细节

当command_id()不论触发mcu还是coulometer的升级，都要走如下函数

```
battery_upgrade(bat_instance_t *device, int type, const char *filepath) {
    stat(filepath, &statbuf)； //检查用于固件升级的file在给定filepath内是否存在
    fd = open(filepath, O_RDONLY); //只读模式读取该file，内容保存在fd这个套接字中，内存中的特殊文件
    buf = (uint8_t *)mmap(NULL, total_size, PROT_READ, MAP_PRIVATE, fd, 0)； //从内核映射到内
存
    /* 进入boot模式(请求命令数据为00, 00, 0x37),请求过程不能超过2秒 */
    battery_send_upgrade_mode(device)；
     /* 发送固件升级包的信息，例如包个数，数据大小，类型，避免mcu和coulometer混淆 */
    battery_send_upgrade_info(device, type, total_size, pkg_num,
                              BAT_PKG_UPGRADE_MAX_DATA_SIZE)；
     /* 发送固件包(就是buftmp+封装头的所有数据) */
    battery_send_upgrade_pkg(device, buftmp,(total_size - i*BAT_PKG_UPGRADE_MAX_DATA_SIZE) >
                    BAT_PKG_UPGRADE_MAX_DATA_SIZE ? BAT_PKG_UPGRADE_MAX_DATA_SIZE
                  : (total_size - i * BAT_PKG_UPGRADE_MAX_DATA_SIZE), i + 1)
    /* 发送后停止映射，结束升级 */
    munmap(buf, statbuf.st_size)
}
```

4.5.5 实际操作

下载厂商提供的升级包。链接：https://zerozero.yuque.com/krief6/bh4ndq/ny558h；注意第一个bin文件用来修改B2电池的寄存器，临时弄的，不要，**只要第二个bin，即`ZZTDV44_2.1.zip`**。本地PC解压后放在/home/Downloads文件夹下，执行下面即可升级。

```
#1 推工具到/hover/tests/目录     //也可是自定义目录
cd repo_falcon
```

```
cd build
cd modules
cd battery_service
cd tests
adb push test_command /hover/tests

#2 推bin文件到工具目录
cd ~
cd Downloads                          //保存bin的地方
adb push b2.bin /hover/tests/         //推送工具目录

#3 执行升级命令
adb shell
cd hover/tests
./test_command 1 /hover/tests/ZZTDV44_2.1.bin        //1 coulometer;2 mcu
```

注意：

- 直接在PC的test_command路径下运行./test_command会失败，会一直提示zsh/bash format error；
- 在飞机上测试时，给错误的或不存在的地址，如`./test_command 1 /home/Downloads/b1`会提示升级Fail。也可以通过更改repo_falcon中/modules/battery_services/里的main.c中的log级别，即LEVEL_INFO。

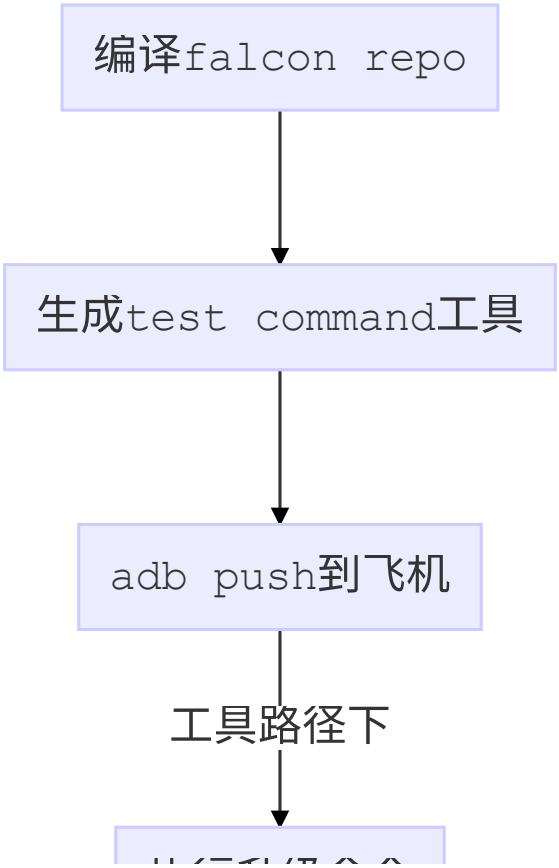
4.5.6 小结

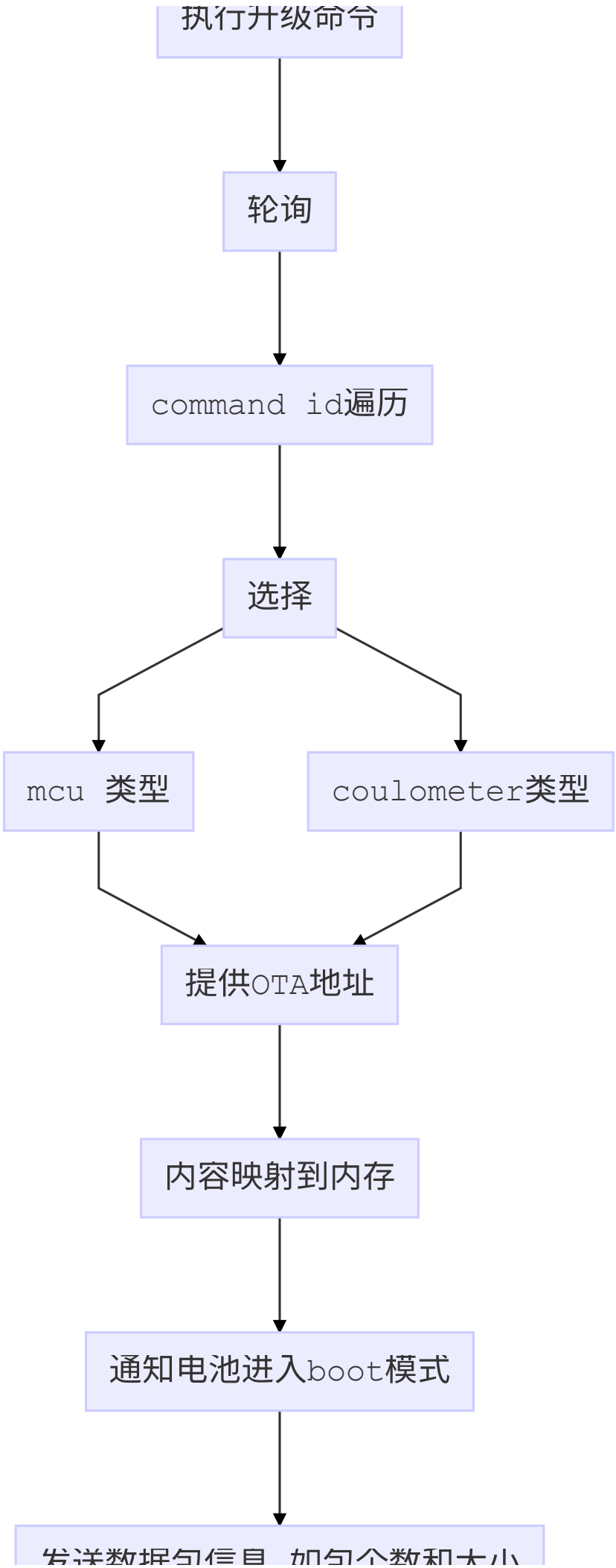
前提：必须提供升级命令所要求的固件升级包地址，即OTA file absolute path。所谓的OTA即Over The Air空中下载技术，是通过移动通信的空中接口对电池数据及应用进行远程管理的技术。**如果给出错误地址或错误的固件升级包，提示有限**。

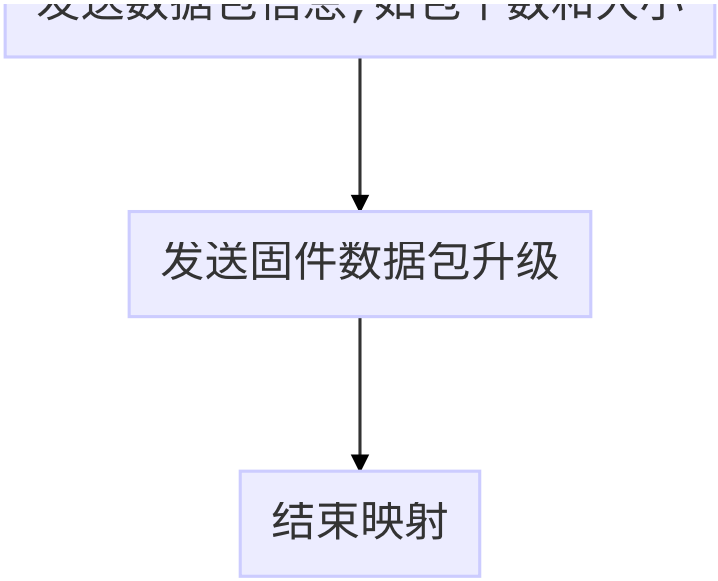
必选：coulometer还是mcu，**升级命令里1是前者，2是后者**；

触发：**必须将test_command工具和bin文件推入飞机**，进入飞机执行升级命令，注意bin在飞机中的路径。

流程：从编译【`falcon repo`到`执行升级命令`】属于客户端的操作；从【轮询到结尾】属于服务端的操作。




执行升级命令
轮询
command id遍历
选择
mcu 类型
coulometer类型
提供OTA地址
内容映射到内存
通知电池进入boot模式
发送数据包信息 如包个数和大小



5 串口通信原理

5.1 串口打开

前提：发送和接收方的串口要打开。

头文件：sys/stat.h和sys/types.h

函数：struct stat 结构体

用途：判断主机获取的串口号是否打开。

```
struct stat state1 = {(__dev_t) 0}; //结构体初始化
const string serialAddr_ = "/dev/ttyHSL1";
serial::Serial serial_;
//打开串口
serial_(SerialAddr_, 115200U, serial::Timeout::simpleTimeout(500)); //方式1
serial_open(devpath, 115200, 'N', 8, 1) //方式2
//判断串口地址是否存在
bool is_serial_addr_exists = stat(serialAddr_.c_str(),&stat1)!=0;
```

5.2 客户端发送请求命令（写入串口）

```
//不同功能，协议不同
enum frame_drone_cmd_e {
  PKG_DRONE_CMD_PRODUCT_DATA = 0x01,
  PKG_DRONE_CMD_DISCHARGE_DATA,
  PKG_DRONE_CMD_CERTIFICATION_MSG,
  PKG_DRONE_CMD_CERTIFICATION_DIG,
  PKG_DRONE_CMD_HISTORY_MSG,
  PKG_DRONE_CMD_CONTROL_MSG,
  PKG_DRONE_CMD_FIRMWARE_UPDATE,
  PKG_DRONE_CMD_MODIFY_AUTH_KEY,
  PKG_DRONE_CMD_END
};

//tag码
typedef enum {
```

```
  PKG_TAG_DRONE = 0x01,
  PKG_TAG_CHARGER,
  PKG_TAG_WIRELESS_CHARGER,
  PKG_TAG_TESTER,
  PKG_TAG_ALARM,
  PKG_TAG_END
} frame_tag_t;
//数据帧细节
static constexpr uint8_t BAT_PKG_SOF = 0xAA;
static constexpr uint8_t BAT_PKG_EOF = 0x55;
static constexpr int BAT_PKG_PROTOCOL_ID = 0x0001;
static constexpr int BAT_PKG_HEADER_SIZE = 11;
static constexpr int BAT_PKG_CRC_SIZE = 1;
static constexpr int BAT_PKG_EOF_SIZE = 1;
static constexpr int BAT_PKG_UPGRADE_MAX_DATA_SIZE = 128;
typedef struct {
  uint8_t sof;
  uint16_t protocol_id;
  uint16_t length;
  uint8_t tag;
  uint16_t master_package_id;
  uint16_t slave_package_id;
  uint8_t cmd_id;
  uint8_t data[BAT_PKG_DATA_MAX_SIZE];
  uint8_t crc8;
  uint8_t eof;
} __attribute__((packed)) bat_frame_t;

void WriteBattery(uint8_t cmdid, uint8_t *data, uint16_t len) {
    if (len > BAT_PKG_DATA_MAX_SIZE) {
      len = BAT_PKG_DATA_MAX_SIZE;
    }

    bat_frame_t frame;
    static uint32_t index = 0;
    frame.sof = BAT_PKG_SOF;
    frame.protocol_id = BAT_PKG_PROTOCOL_ID;
    frame.length = BAT_PKG_HEADER_SIZE + len + BAT_PKG_CRC_SIZE;
    frame.tag = PKG_TAG_DRONE;
    frame.master_package_id = index++;
    frame.slave_package_id = 0;
    frame.cmd_id = cmdid; //PKG_DRONE_CMD_PRODUCT_DATA
    if (len > 0) {
      memcpy(frame.data, data, len);
    }
    frame.data[len] = crc8((uint8_t *) &frame, frame.length - BAT_PKG_CRC_SIZE);
    frame.data[len + 1] = BAT_PKG_EOF;
    serial_.write((uint8_t *) &frame, frame.length + BAT_PKG_EOF_SIZE);
  }
```

5.3 服务端读取命令（读取串口）

```
typedef std::deque<uint8_t> mtRecvBuffer;  //队列方式读取
static constexpr int recv_buffer_size = 1152;
bool ReadBattery(BatteryServerPimpl::mtRecvBuffer &recv_buf) {
    uint8_t buf[recv_buffer_size];
    size_t bytes_read = serial_.read(buf, recv_buffer_size);
    if (bytes_read == 0) {
      zz_log_string_writer(ZZLOG_LEVEL_ERROR, FNL, "[ReadBattery]Bytes read time out!");
      return false;
    }
    if (bytes_read > recv_buffer_size) {
```

```
      zz_log_string_writer(ZZLOG_LEVEL_ERROR, FNL, "[ReadBattery]Bytes read error: %d",
bytes_read);
      return false;
    }
    for (size_t i = 0; i < bytes_read; i++) {
      recv_buf.push_back(buf[i]);
    }
    return true;
  }
//如果上述读取串口数据成功，下面去掉封装头 AA
auto it_aa = std::find(recv_buf.begin(), recv_buf.end(), 0xAA);
if (it_aa != recv_buf.begin()) {
    printf("[Parser]Erasing %d:", it_aa - recv_buf.begin());
    DumpBuf(recv_buf.begin(), std::next(it_aa));
    printf("\n");
    recv_buf.erase(recv_buf.begin(), it_aa);
}
if (it_aa == recv_buf.end()) {
    continue;
}
//删除数据帧的末尾55，将中间的数据协议放到frame中去
// Find AA-CRC8-55 part. If found, parse it.
// Loop until no AA or 55 can be found.
      while (true) {
        auto it_55 = it_aa;
        bool is_no_55_left = false;
        while (true) {
          usleep(10000);
          it_55 = std::find(std::next(it_55), recv_buf.end(), 0x55);
          if (it_55 == recv_buf.end()) {
            is_no_55_left = true;
            break;
          }
          if (it_55 - recv_buf.begin() <= 1) {
            continue;
          }
          if (!CrcCheck(it_aa, std::prev(it_55))) {
            continue;
          }
          bat_frame_t frame = ParseBuf(recv_buf.begin(), std::next(it_55));
          mutex_frame_queue_.lock();
          frame_queue_.push(frame);  //开启线程做
          mutex_frame_queue_.unlock();
          recv_buf.erase(recv_buf.begin(), std::next(it_55));
          break;
        }
        if (is_no_55_left) {
          break;
        }
        it_aa = std::find(recv_buf.begin(), recv_buf.end(), 0xAA);
        if (it_aa == recv_buf.end()) {
          break;
        }
      }
```

5.4 服务端数据处理（回应命令）

根据tag码和cmd_id处理，打印信息。

```
void FrameProcessor() {
    while (running) {
        bat_frame_t frame;
```

```
        mutex_frame_queue_.lock();
        if (frame_queue_.empty()) {
            mutex_frame_queue_.unlock();
            usleep(10000);
            continue;
        }
        frame = frame_queue_.front();
        frame_queue_.pop();
        mutex_frame_queue_.unlock();
        switch (frame.tag) {
            case PKG_TAG_DRONE: {
            switch (frame.cmd_id) {
            case PKG_DRONE_CMD_PRODUCT_DATA: {
                ProductInfoCallback(frame);
                break;
                }
            case PKG_DRONE_CMD_DISCHARGE_DATA: {
                DischargeInfoCallback(frame);
                break;
                }
            default:
                break;
            }
          break;
        }
        default:
          break;
      }
    }
}
```