A2508基于IMAP的PDU功率分配软件概要设计

**发放范围**：XXXX软件开发部、项目组、评审组

**本模板责任部门**： 中研

|  |  |  |  |
| --- | --- | --- | --- |
| **项目名称** | 基于IMAP的PCU功率分配软件概要设计 | 项目代码 | A2508 |
| 拟制人 | YBA40320 | 日期 | 2025-09-22 |
| 审核人 |  | 日期 |  |
| 批准人 |  | 日期 |  |

**修订记录：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 版本 | 日期 | 修改内容及理由 | 拟制人 | 审核人 | 批准人 |
| V1.0 | 2025-09-22 | 初审文稿 | YBA40320 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**目录**

[1. 概述 ４](#_Toc209380739)

[1.1软件背景 ４](#_Toc209380740)

[1.2软件范围 ４](#_Toc209380741)

[1.3开发环境 ４](#_Toc209380742)

[1.4命名规范 ４](#_Toc209380743)

[1.5术语和缩写词 ４](#_Toc209380744)

[2. 软件工程依赖及基本功能 ６](#_Toc209380745)

[3. 软件架构设计 ７](#_Toc209380746)

[3.1软件体系结构描述 ７](#_Toc209380747)

[3.1.1 功能模块描述 ７](#_Toc209380748)

[3.1.2 程序入口IO描述 ８](#_Toc209380749)

[3.2软件架构技术决策分析 ８](#_Toc209380750)

[4. 软件主要的数据结构 ９](#_Toc209380751)

[4.1 双向内核式链表节点结构体定义 ９](#_Toc209380752)

[4.2 接触器可变数组结构体定义 ９](#_Toc209380753)

[4.3 功率节点数据结构体定义 １０](#_Toc209380754)

[4.4 功率节点可变数组结构体定义 １０](#_Toc209380755)

[4.5 充电枪头数据结构体定义（或叫汇流点、充电枪位） １０](#_Toc209380756)

[4.6 充电枪头可变数组结构体定义 １０](#_Toc209380757)

[4.7 拓扑参数序列化键值对结构体定义 １１](#_Toc209380758)

[4.8 可变长键值对序列集合结构体定义 １１](#_Toc209380759)

[4.9 枪接铜排接触器联合体定义 １１](#_Toc209380760)

[5. 软件总体方案 １２](#_Toc209380761)

[5.1系统功能模块划分 １２](#_Toc209380762)

[5.2系统功能模块关系 １２](#_Toc209380763)

[5.3系统主要功能模块设计 １２](#_Toc209380764)

[5.3.1 拓扑结构参数序列化/ FR01\_TopoParam\_Srlize １２](#_Toc209380765)

[5.3.2 拓扑构建/FR02\_ Toponomy\_Constructing １４](#_Toc209380766)

[5.3.3 功率分配策略/FR03\_Allocation\_Tactic １７](#_Toc209380767)

[5.3.4 调试可视/FR04\_Vislize\_Dbg １８](#_Toc209380768)

[5.3.5 可追溯日志/ FR05\_TracableLog １９](#_Toc209380769)

[6. 软件安全策略 １９](#_Toc209380770)

[6.1统一管理外部变量 ２０](#_Toc209380771)

[6.2专有RAM ２１](#_Toc209380772)

[6.3关键数据保护机制 ２２](#_Toc209380773)

[6.4专用堆内存分配 ２４](#_Toc209380774)

[7. 接口定义 ２４](#_Toc209380775)

[7.1硬件接口 ２５](#_Toc209380776)

[7.2软件接口 ２５](#_Toc209380777)

[8. 软件设计计划表 ２５](#_Toc209380778)

[9. 附录 ２６](#_Toc209380779)

# 概述

## 1.1软件背景

随着电动汽车产业的迅猛发展，直流充电场站作为关键基础设施，其运行效率和智能化水平直接关系到用户体验和运营效益。功率分配系统作为直流充电场的"大脑"，通过智能调度算法和柔性配电技术，对有限功率资源进行动态调配，已成为提升场站整体效能的关键技术手段。

传统直流充电桩采用固定功率分配模式，常导致功率模块利用不均衡——部分模块超负荷运行而部分模块处于闲置状态。现代功率分配系统通过实时监测各充电枪的功率需求、车辆电池SOC（荷电状态）以及各功率模块的工作状态，运用动态调配算法实现功率的按需分配。这种智能化分配策略可使设备利用率提升15%以上，显著缩短投资回报周期。

目前主流功率分配矩阵有三种实现方式，各自具有不同特点

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ​拓扑类型 | 实现原理​ | 优点​ | 缺点​ | ​适用场景 |
| ​半矩阵 | 使用无极性直流接触器实现模块分配 | 模块调用灵活，维护更换方便 | 成本较高，体积偏大 | 大型公共充电站 |
| ​全矩阵 | 使用板载继电器进行功率分配 | 整桩体积小，成本较低 | 受继电器性能限制，电流一般小于150A | 中小功率场站 |
| 环形 | 半矩阵/全矩阵的简化版本 | 成本低，满足大部分场景 | 极端条件下可能有枪无法调用其 | 普通停车场和社区充电站 |

本软件内嵌于运行在ISMU-X的PCU软件中，通过拓扑参数自动识别拓扑类型并生成对应数据结构；根据CCU提供的桩侧功率需求信息自动计算最优功率分配方案，并调用接口执行功率调配动作。

## 1.2软件范围

具体软件范围及交付件如下表所示

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 设备名称 | 软件名称 | 交付件 | 说明 |
| 1 | 功率柜IMSU-X | PDU\_CORE.a | 源码及.a静态库工程 | 全新开发 |
| 2 | \ | PDU\_DEMO | 静态库演示及测试example例程 | 全新开发 |
| 3 | \ | PDU 功率分配静态库使用说明.md  功率分配静态库例程使用手册.pdf | 软件使用说明及主要函数介绍；  例程应用手册 | 全新开发 |

## 1.3开发环境

编译工具 ：IAR Embedded Workbench IDE

编译器版本 ：9.30.1

语言 ：C语言

CPU ：STM32F407ZGT6

操作系统 ：FreeRTOS

## 1.4命名规范

软件开发各参数变量命名规则和接口函数命名规范参照我司《C语言编码规范》。

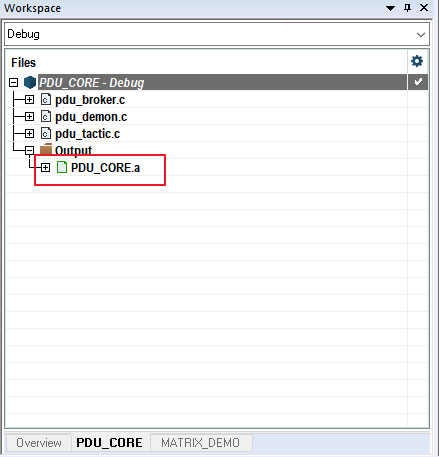
## 1.5术语和缩写词

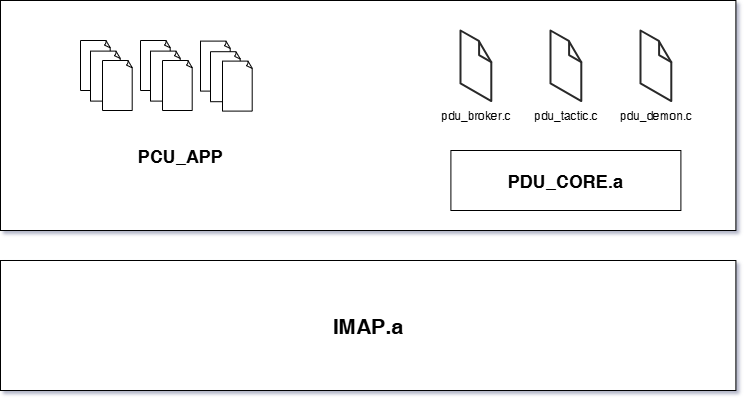
表二 术语与缩写词

|  |  |
| --- | --- |
| **术语/缩写词** | **说明** |
| IMAP（Intelligence Monitor Abstract Platform） | 智能监控抽象平台：面向智能电力设备的模块化开发平台，专注于高可靠性、实时性和可扩展性，支持充电桩等复杂电力系统的快速开发。 |
| PCU（Power Control Unit） | 功率控制单元：充电桩核心控制器，负责功率转换、分配及保护。 |
| CCU（Charge Control Unit） | 充电控制单元：终端核心控制器，负责车桩充电流程的控制。 |
| PC-Upper（ PowerCube Upper） | 远程信息控制单元：平台、APP与充电桩之间的数据信息交互中心。 |
| Dispenser-Upper（Charger Dispenser Upper Controller） | 充电终端计费单元：作为终端PC的上位机，同时负责刷卡、电表、灯板读取或控制 |
| MTU（Maintanence Unit） | 维护单元：功率柜与终端之间的一道物理隔层。 |
| PDU（Power Distribution Unit） | 功能分配单元：用于功率分配接触器的DO、DI板。 |
| ACAmbientCU（AC Ambient Collect Uint） | 交流环境采集单元：交流电表数据采集、交流环境采集等。 |
| ACContactCU（AC Contact Control Uint） | 交流接触器控制单元:负责多交流（>2）接触器的控制与反馈采集。 |
| PCAmbientCU（Power Cube Ambient Control Unit） | 电源柜环境控制单元：负责温度采集、水泵、风机控制。 |
| EV（Electric Vehicle） | 电动汽车 |
| EVCC（Electric Vehicle Communication Controller） | 电动汽车通信控制器：与桩进行通信交互的控制引导器 |
| EVSE（Electric Vehicle Supply Equipment） | 电动汽车供应设备：指给车充电的供能设备 |
| Charging Station Manufacturers | 充电站制造商 |
| CPO（Charge Point Operator） | 充电点运营商 |
| OCPP （Open Charge Point Protocol） | 开放式充电点协议 |
| DFX（design for debug/test/maintenance） | 调试、测试、维护设计：在软件开发过程中便考虑到了软件调试和后续维护的方便性。 |

# 软件工程依赖及基本功能

本项目作为PCU应用软件引用的静态库，负责接收并解析CCU侧发送的功率需求并合理分配功率资源，向PCU颁发功率设置命令。PCU应用软件工程中包含该静态库，运行于IMAP平台之上。本项目软件工程在本文档中简称为功率分配静态库。





* 功率分配静态库的程序运行逻辑类似某种介于CCU和PCU之间的中间件，作为充电桩主程序与功率执行单元之间的桥梁，通过分配算法和实时功率监测，实现对功率柜有限功率资源的最优化分配，确保系统在高负载情况下仍能保持高效、稳定运行，并最大化充电模块的整体利用效率。

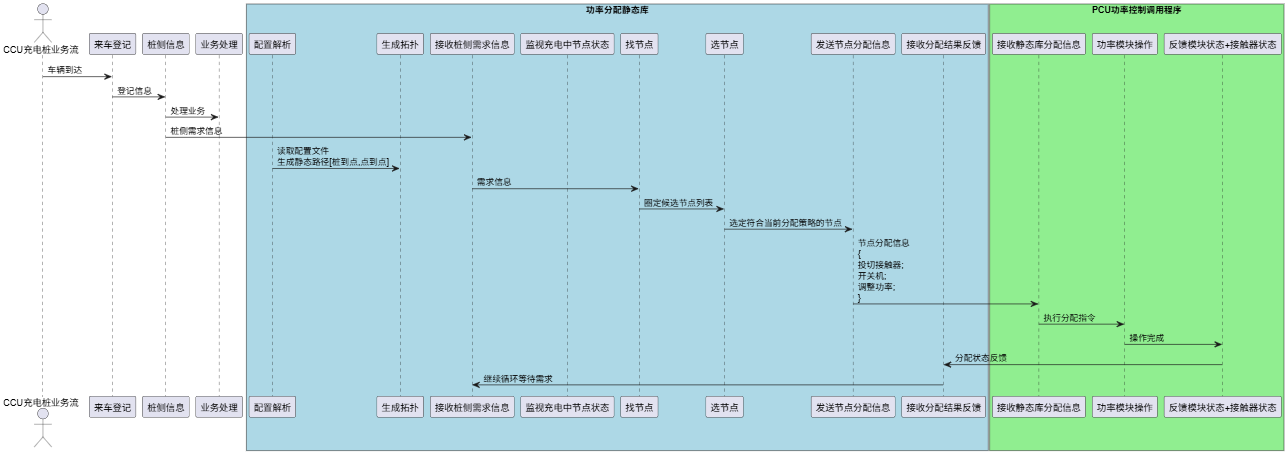


图2 软件运行序列图

# 软件架构设计

## 3.1软件体系结构描述

### 3.1.1 功能模块描述

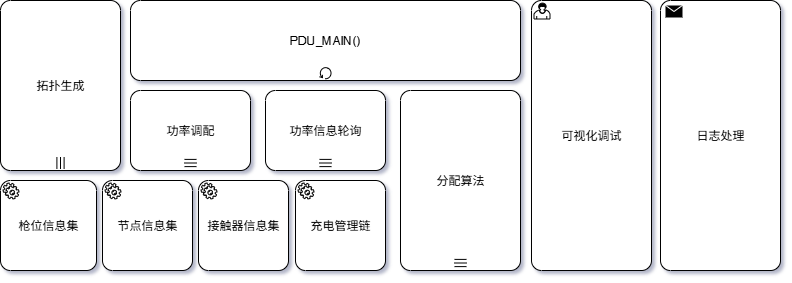
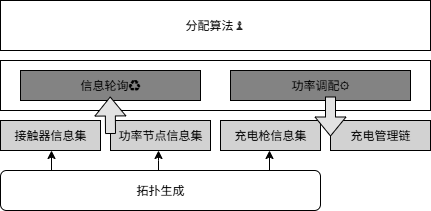


图3 软件体系结构



|  |  |  |  |
| --- | --- | --- | --- |
| 拓扑生成 | | 抽象物理电气连接关系，在软件层面构建①功率节点②接触器③充电枪 三个信息集合 | **数据对象**：   1. 由所有充电枪头信息   组成的集合   1. 由所有功率节点信息组成的集合 2. 由所有接触器信息组成的集合 3. 充电枪→电气连接节点的双向链表 4. 充电枪→正在向该枪输出功率的节点的单向链表 |
|  | 功率调配 | **时序控制**​：管理指令的执行顺序，例如先预合节点验证绝缘再启用功率设置，先降功率再切换等，避免产生浪涌电流或短路。  ​**安全校验**​：在执行前进行安全校验，防止设置参数超出安全限值（如：超功率、过压、过流） |
| 功率信息轮询 | 从每个功率模块读取：​输出电流、输出电压、当前功率、运行状态（启用/禁用/故障/待机）等；从充电枪读取车辆需求功率、连接状态；为分配算法和可视化调试提供决策依据和数据展示来源 |
| 分配算法 | **输入**​：自拓扑生成的约束 + 功率信息轮询的实时数据  **资源评估​**：计算出与需求充电枪连接的功率节点链上可用的总功率和每个功率模块的可用容量。  ​**策略执行​**：根据预设策略，（如：平均分配、优先级分配、抢占式分配、基于效率的分配等）进行数学计算，为每个可以选择的功率节点评估符合当前分配策略的分数，最后决定“将哪个或哪些功率模块分配给哪个充电枪，分配多少功率”  **输出**：生成一个详细的**调配指令集**（绝缘测试节点，功率分配预充节点，节点分配电流、电压、吸合接触器的序号），下发给功率调配模块。 | |
| 可视化调试 | | ​**状态监控**​：以RTT终端方式展示系统拓扑图形、每个模块和充电枪的实时数据（功率、电压、电流、状态）  ​​**策略模拟**​：在产品不具备实操条件下，可以进行离线模拟验证软件分配结果。 | |
| 日志处理 | | ​**操作日志​**：记录所有发给PCU的控制指令（哪个节点哪个开关、在什么时候、设置了什么参数）。  ​**策略日志​**：记录所有关键策略事件（分配算法轨迹，决策逻辑过程） | |

### 3.1.2 程序入口IO描述

功率分配静态库使用简约接口设计，对于PCU来说静态库仅显式提供两个回调函数

|  |  |
| --- | --- |
| 函数声明 | 功能说明 |
| **PDU\_STA FSM\_mainEntry\_PDU(PDU\_CMD cmd)** | 向PCU提供一个被周期调用的函数入口，输入PCU对功率分配静态库的指令，返回当前功率分配静态库的状态 |
| **void register\_to\_pdu(void)** | 将PCU提供的全局变量按约定名称注册进静态库，具体参考7.2软件接口列表 |

函数入口根据输入命令参数做出相应操作

输入命令

返回状态

**FSM\_mainEntry\_PDU**

【初始化】上电后或upper下发新参数时

【未就绪】通常为拓扑参数为构建成功

【运行中】静态库入口函数正常运行

【旁路】暂时不启用功率分配静态库

【采】通过回调接口输入序号查询对应的节点、接触器、充电枪

【控】通过回调接口

调配功率模块

【故障中】数据库错误；无法分配功率

【运行】正常执行功率分配逻辑

【旁路中】入口函数直接返回

【调试】在RTT终端打印调试信息

## 3.2软件架构技术决策分析

**优势：**

1. **功率分配静态库编译时根据icf预设参数与应用程序静态内存隔离**
2. **代码间集约管理注册互通变量，避免“全局变量迷宫”**
3. **数据结构为软件运行后动态生成，通过参数设置实现一个程序可以适配多种拓扑结构**

**风险：**

1. **需要兼容多种拓扑结构，势必会导致软件维护灵活性降低**
2. **软件前期验证功率分配算法准确性只能依靠demo的沙盘推演，无法验证实时性和与IMAP平台的结合度是否良好。**

# 软件主要的数据结构

|  |  |  |
| --- | --- | --- |
| 参数类型 | 结构体名称 | 内容含义 |
| 双向内核式链表节点 | **Proto\_listObj** | 基本等同于接触器数据 |
| 接触器可变数组 | **Alloc\_contactorArray** | 根据拓扑参数创建容纳接触器的数据库 |
| 功率节点数据 | **Alloc\_nodeObj** | 描述功率节点，并记录分配数据 |
| 功率节点可变数组 | **Alloc\_nodeArray** | 根据拓扑参数创建容纳功率节点的数据库 |
| 充电枪头数据 | **Alloc\_plugObj** | 描述枪头，并记录功率需求 |
| 充电枪头可变数组 | **Alloc\_plugArray** | 根据拓扑参数创建容纳枪头的数据库 |
| 拓扑参数序列化键值对 | **KeyValue\_Pair** | {“名称”，数值} |
| 可变长键值对序列集合 | **KeyValue\_Array** | {{“名称”，数值}，{“名称”，数值}，{“名称”，数值}} |
| 枪接铜排连接器联合体 | **Combo\_Koinon** | 枪号->铜排上任一个接触器编号 |

### 4.1 双向内核式链表节点结构体定义

typedef *struct* **Proto\_listObj**

{

***uint32\_t*** id;

*bool* is\_contactee;

*struct* **Proto\_listObj** \*next\_linker;

*struct* **Proto\_listObj** \*prev\_linker;

} **ListObj**, \***ListRef**;

### 4.2 接触器可变数组结构体定义

typedef *struct*

{

***uint32\_t*** front\_canary; // absolutely required to be top element

***size\_t*** length;

***size\_t*** forks\_num;

**ListObj** obj\_array*[]*;

} **Alloc\_contactorArray**;

### 4.3 功率节点数据结构体定义

*Struct*  **Alloc\_nodeObj**

{

***uint32\_t*** id;

*float* value\_Iset;

*float* value\_Vset;

***uint32\_t*** adaptness;

*bool* energon;

*struct* **Alloc\_nodeObj** \*next\_charger;

    //...

};

### 4.4 功率节点可变数组结构体定义

typedef *struct*

{

***uint32\_t*** front\_canary; // absolutely required to be top element

***size\_t*** length;

***size\_t*** pools\_num;

***size\_t*** forks\_num;

*struct* **Alloc\_nodeObj** obj\_array*[]*;

} **Alloc\_nodeArray**;

### 4.5 充电枪头数据结构体定义（或叫汇流点、充电枪位）

*struct* **Alloc\_plugObj**

{

***uint32\_t*** id;

*bool* energon;

*float* demand;

    //...

*struct* **Alloc\_nodeObj** \*next\_charger;

**ListObj** koinon;

};

### 4.6 充电枪头可变数组结构体定义

typedef *struct*

{

***uint32\_t*** front\_canary; // absolutely required to be top element

***size\_t*** length;

**CRITERION** criterion;

*struct* **Alloc\_plugObj** obj\_array*[]*;

} **Alloc\_plugArray**;

### 4.7 拓扑参数序列化键值对结构体定义

typedef *struct*

{

***uint32\_t*** value;

*char* key[KEYSTRLEN];

} **KeyValue\_Pair**;

### 4.8 可变长键值对序列集合结构体定义

typedef *struct*

{

***size\_t*** count;

***uint16\_t*** crc;

**KeyValue\_Pair** pairs*[]*;

} **KeyValue\_Array**;

### 4.9 枪接铜排接触器联合体定义

typedef *union*

{

***uint32\_t*** all;

*struct*

    {

***uint32\_t*** contactor : 24;

***uint32\_t*** plug : 8;

    } id\_of;

} **Combo\_Koinon**;

|  |  |  |
| --- | --- | --- |
| 参数类型 | 结构体名称 | 内容含义 |
| 功率需求信息结构体 | **PlugInfoInst, \*PlugInfoPtr** | 重命名PCU中St\_ChargingGunInfo |
| 功率供给信息结构体 | **NodeInfoInst, \*NodeInfoPtr** | 重命名PCU中St\_PwrNodeInfo |
| 接触器信息结构体 | **ContactorinfoInst, \*ContactorinfoPtr** | 重命名PCU中St\_RouteInfo |

    typedef St\_ChargingGunInfo PlugInfoInst, \*PlugInfoPtr;

typedef St\_PwrNodeInfo NodeInfoInst, \*NodeInfoPtr;

typedef St\_RouteInfo ContactorinfoInst, \*ContactorinfoPtr;

# 软件总体方案

## 5.1系统功能模块划分

| **功能需求** | **功能需求概述** | **备注** |
| --- | --- | --- |
| **拓扑结构参数序列化FR01\_TopoParam\_Srlize** | 将拓扑参数形成一个序号无关、不定长的键值对序列，供功率分配静态库读取生成拓扑结构 |  |
| **拓扑构建**  **FR02\_ Toponomy\_Constructing** | 发现和抽象物理硬件连接关系，在软件层面构建一个反应枪-点、点-点电气连接的数据结构。 |  |
| **功率分配策略FR03\_Allocation\_Tactic** | 作为系统的​“大脑”​，是功率分配策略的核心实现。它根据当前系统状态和既定策略，计算出最优的功率分配方案。 |  |
| **调试可视**  **FR04\_Vislize\_Dbg** | 为开发人员提供了一种调试手段​，用于功率分配的离线推演，可以监控、干预和理解系统运行状态，缩短项目研发周期。 |  |
| **可追溯日志**  **FR05\_TracableLog** | 。。。 |  |

## 5.2系统功能模块关系

序列化拓扑参数

拓扑构建

可追溯日志

调试可视化

功率分配策略

## 5.3系统主要功能模块设计

### 5.3.1 拓扑结构参数序列化/ FR01\_TopoParam\_Srlize

■拓扑参数可以描述物理硬件连接关系，在软件层面构建一个虚拟的功率网络模型

●**确定容量**​：由功率池个数、功率节点个数和连接器个数来推算矩阵容量，以及每个每个功率池的节点数和每个节点的连接器数。

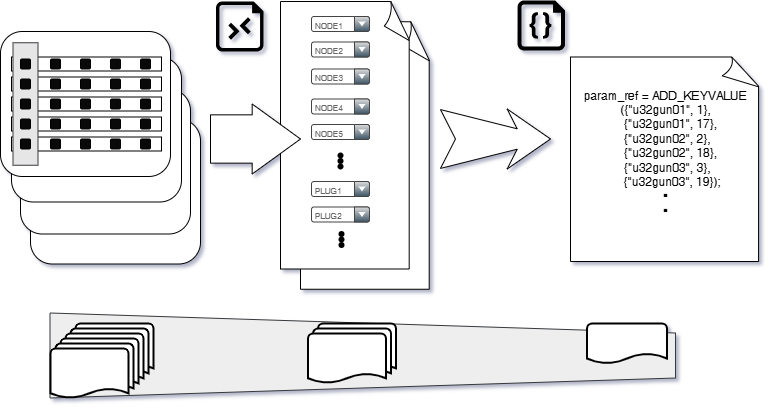
●**池内点-点映射**​：确定一个功率池内一条铜排上连接的所有功率节点之间的电气连接。

●**桩-点互联**​：确定哪个或哪些功率模块可以被分配到哪个充电枪。这是分配算法的物理约束基础。例如，识别出“功率模块01到06是充电枪1的电气连接模块，而功率模块01到06也通过另外一组接触器共享给充电枪2，可通过两组接触器切换给枪1或枪2”。

●**生成元数据**​：产生描述这种结构的键值对，并生成一个不定长无序的数据集合供程序解析。例如："u32gun01", 1}, {"u32gun01", 17}, {"u32gun02", 2}, {"u32gun02", 18}, {"u32gun03", 3}

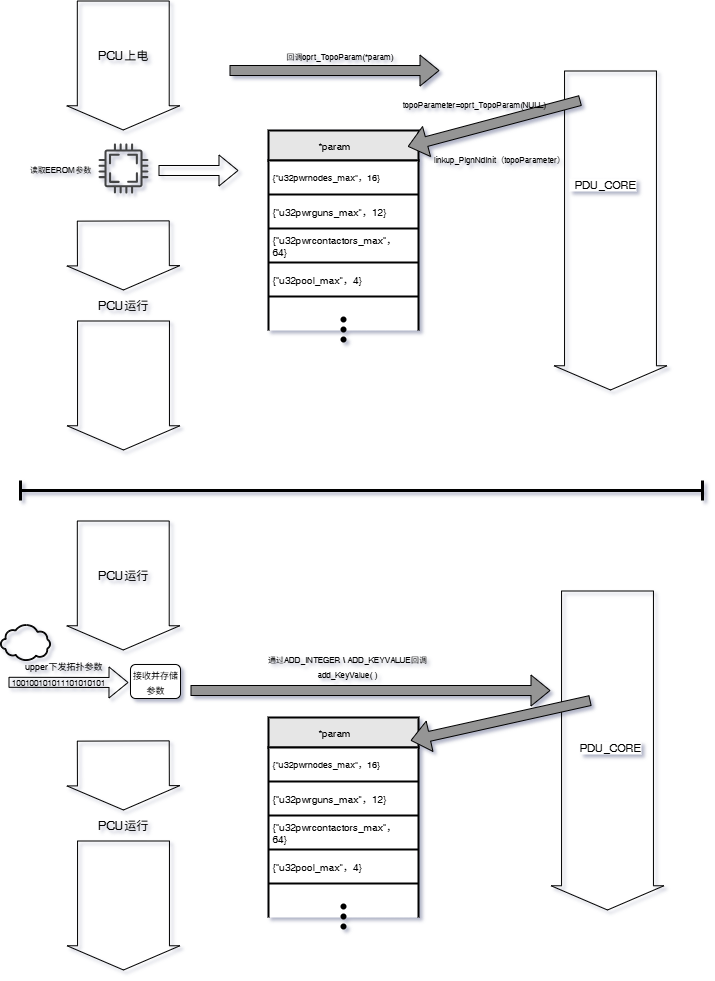
●**重要性**​：这是整个系统运行的**基石**。如果拓扑识别错误，后续的分配和调配指令会导致功率无法正确送达，甚至引发安全事故。

■拓扑参数由upper通过通信规约下发给PCU，通过PCU运行**功率分配静态库**提供的参数序列化函数将参数形成**功率分配静态库**可分解析构的键值对序列，并由PCU存入FLASH。



●**精简参数**​：如上图，序列化的参数未必完全等同于upper通信规约，而是一个精简子集。按照信息量排列，物理电气结构 > upper规约设置点表 > 序列化参数。精简后的参数通常包括如下：

|  |  |
| --- | --- |
| 含义 | 伪代码 |
| 功率节点总数 | u32pwrnodes\_max |
| 接触器总数 | u32pwrcontactors\_max |
| 功率池个数 | u32pool\_max |
| 充电枪个数 | u32pwrguns\_max |
| 充电枪与接触器对应关系 | {"u32gun01", 1},  充电枪1与1号接触器相连 |
| {"u32gun01", 17}  充电枪1同时跨接17号接触器 |
| {"u32gun07", 0}  充电枪7枪位空置，不与任何接触器连接 |
| 铜排限流参数 | （可选） |
| … | … |



### 5.3.2 拓扑构建/FR02\_ Toponomy\_Constructing

内建拓扑数据的宗旨是为了在功率需求发生时方便对功率提供对象的遍历，对于全矩阵结构来说遍历最终是在遍历枪与节点的连接，不管一个枪与多个节点连接，一个节点与多个枪连接，拓扑数据都会如实反映。

拓扑数据结构在软件上的最佳实现方法就是对数据单元的内核式链表互联；功率节点之间以及充电桩枪头与功率节点（模块）之间的物理电气连接，被抽象为数据单元之间的链表结构；每个设备对应一个数据对象，对象之间通过指针或引用形成链式关系，从而反映实际的电气拓扑

■软件对数据维护的日常操作即：

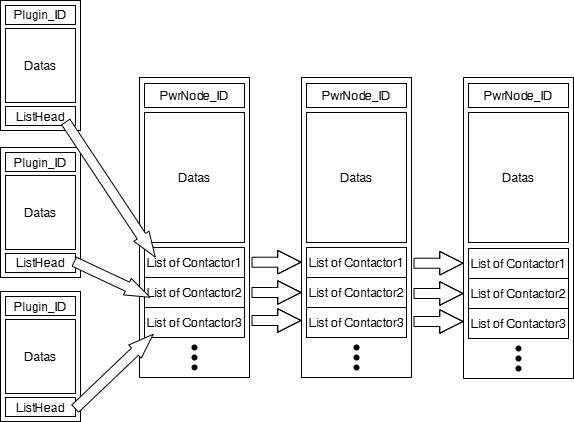
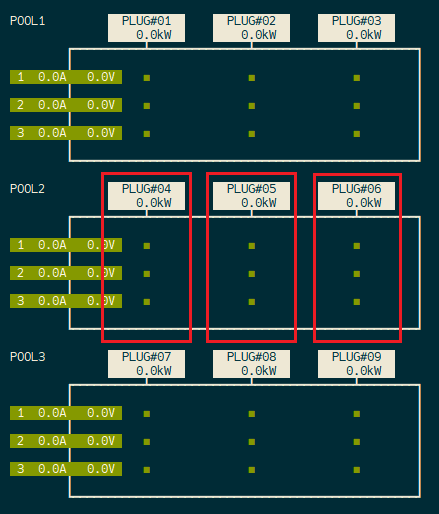
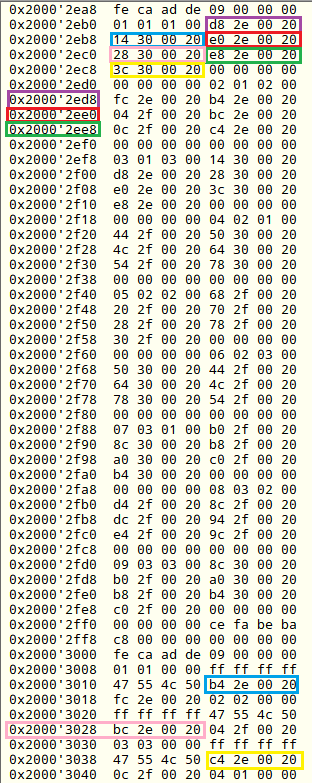
●通过链表节点侵入式访问数据结构内容（如由一个功率节点获取当前节点的额定功率）

●通过链表节点前推回溯其他节点（如通过枪头寻找下一个符合分配条件的功率节点）

●通过链表节遍历追溯链表头（如从一个功率节点找到与该节点连接的充电桩）

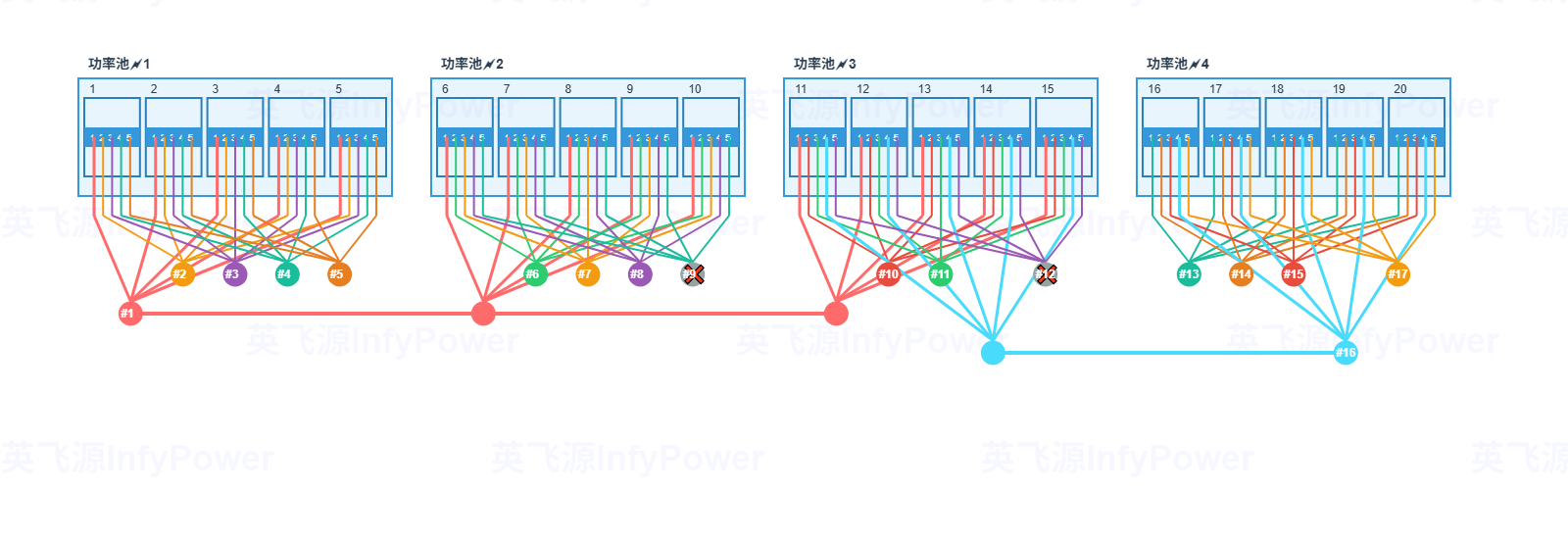
●通过链表头遍历所有链表节点（如根据枪头找到所有与其电气连接的功率节点）

●对链表进行节点插入和解除（如响应upper下发设置重构电气拓扑）

 ****

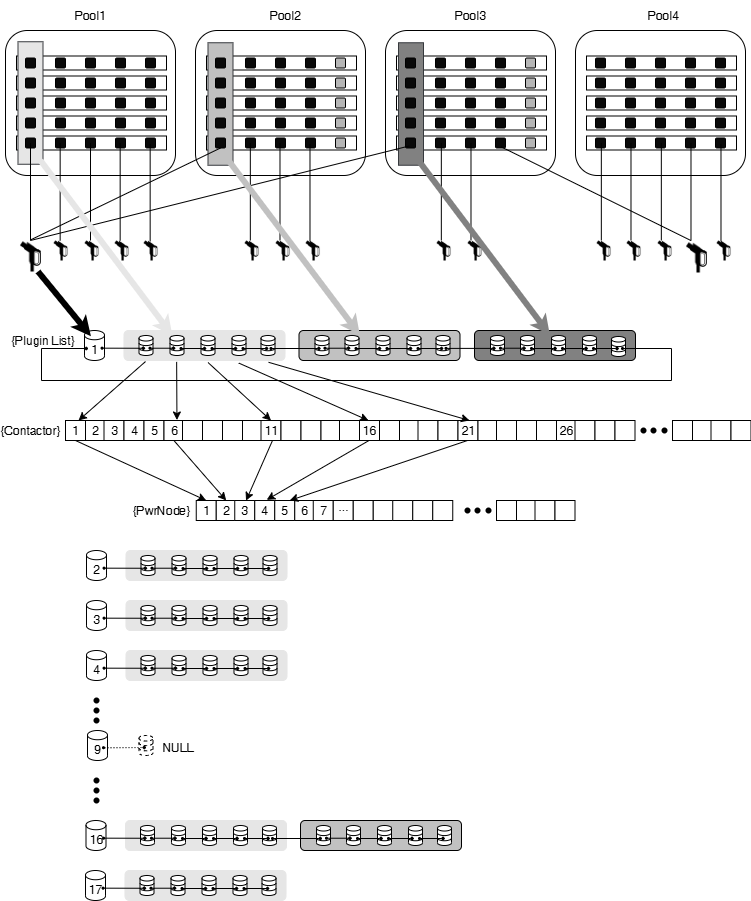
■充电枪与功率节点的电气联接在程序中被抽象成一个以充电枪数据单元为链表头的双向链表，链表节点为与该枪头电气连接的接触器。

以下图为例：工程配备4个功率池，每个功率池5个功率节点，每个功率节点可以连接5个接触器。



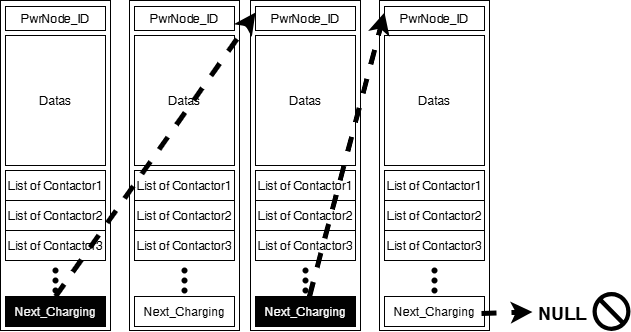
●其中1号液冷超充枪跨接了三组铜排共计连接15个功率节点，16号枪跨接两组铜排共计10个功率节点。在数据结构中，对应1号充电枪的双向链表体现为Plugin1←→Contactor1←→Contactor6←→Contactor11←→Contactor16←→Contactor21←→Contactor26←→Contactor31←→Contactor36←→Contactor41←→Contactor46←→Contactor51←→Contactor56←→Contactor61←→Contactor66←→Contactor71。当1号充电枪接收到CCU侧功率请求时，程序会在这个链表上寻找最优匹配功率需求的节点。

由于矩阵的容量已知，一个接触器只从属于一个功率节点，所以通过接触器编号就可以反推节点编号。所以1号充电枪连接的功率节点实际上是Plugin1←→Node1←→Node2←→Node3←→Node4←→Node5←→Node6←→Node7←→Node8←→Node9←→Node10←→Node11←→Node12←→Node13←→Node14←→Node15。



■枪头的充电操作在软件中体现为通过普通链表对功率节点的串联

●注意充电链表是相对描述电气拓扑关系的内核式链表更简单快捷的另一种链表



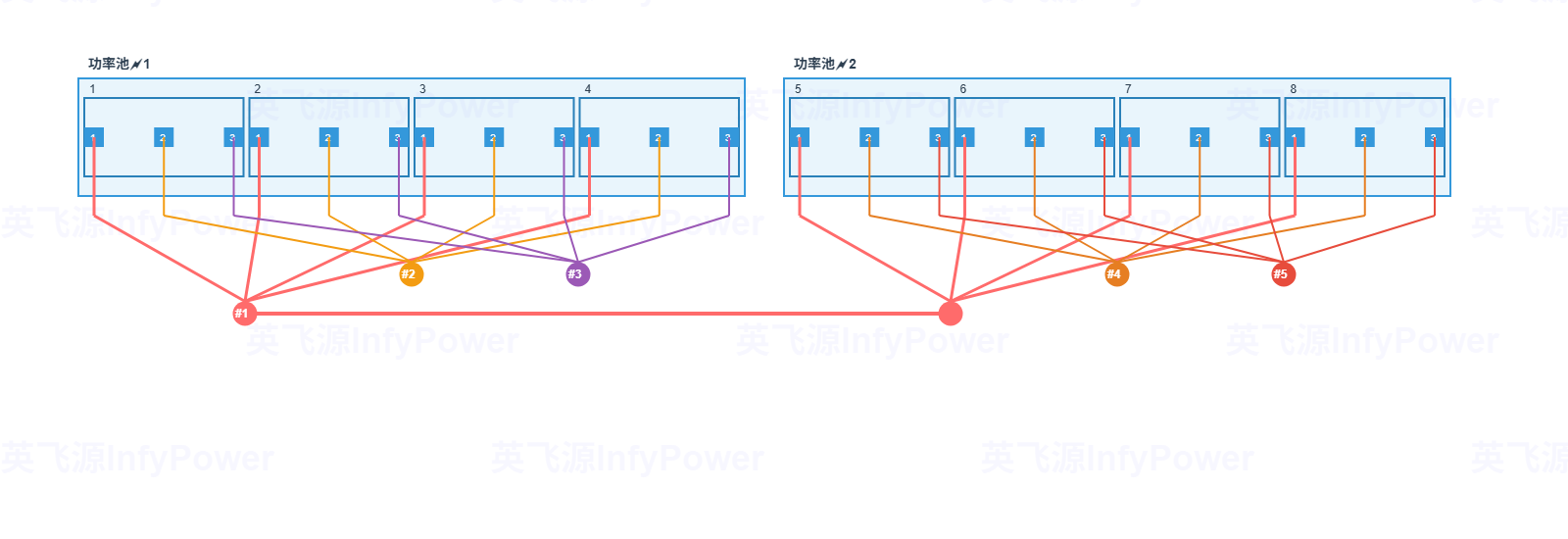
枪头通过该链表实现对功率节点的并解列操作，包括如下：

●通过插入链表节点实现对功率节点并列

●通过解除链表节点实现对功率节点解列

●通过遍历链表实现对在充功率节点的查询，从而进行下一步逻辑决策

以下图为例,工程配置两个功率池，每个功率池4个功率节点，每个节点连接3个接触器



软件运行一段时间后，#1~#5每个充电枪都先后收到了功率需求，于是形成5个充电操作链表：

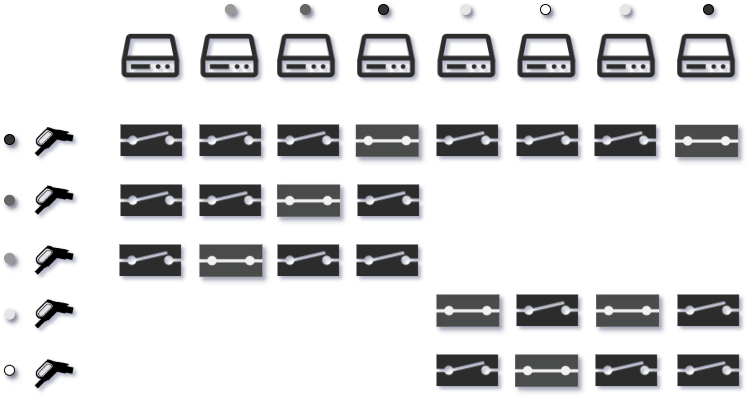
●Plugin1→Node4→Node8

●Plugin2→Node3

●Plugin3→Node2

●Plugin4→Node5→Node7

●Plugin5→Node6



### 5.3.3 功率分配策略/FR03\_Allocation\_Tactic

■**策略的参考维度**

●**等健康度**

**动态轮巡**​：健康度是通过每个功率模块的历史累计工作时间和接触器动作次数做一个加权得分，接触器权重随铜排接触器个数增加。优先调用历史工作量更少的模块和动作次数更少的接触器，实现真正的“劳逸均衡”。

**老化预测与主动均衡**​：目前尚未形成清晰思路。

●**最大效率**

**效率寻优**​：每个功率模块在不同负载率（如10%-100%）下的效率是不同的，通常存在一个**高效点**​。策略应实时查询所有模块的“效率-MAP”图，优先让更多模块工作在各自的高效点附近，从而提升**系统整体效率**，而非单个模块最大功率。

●**车优先级**

* + **用户等级**​：VIP用户VS.普通用户。
  + ​**订单类型**​：预约订单（需保证功率和时间）> 即时订单。
  + ​**车辆需求**​：高速车辆（800V）> 常压车辆（400V）；低SOC（急需补能）> 高SOC。
  + **抢占与补偿机制**​：高优先级车可以抢占低优先级车部分功率配额，但功率分配静态库应记录这种“不公平”行为，并尽可能在后续有可用配额时给予补偿。

●**经济运行**

**电网电价联动**：在TCU告知**谷电价**时段时，可适当提高功率分配上限，鼓励充电，降低运营成本；在峰电价时段，可适当限功率，或通过CCU引导用户进入“慢充”模式。

**网侧用电调整：**响应网侧用电要求，比如用电高峰时段根据台区能源路由器的有序用电要求限制功率。

功率因数调整​：暂不考虑。

●**铜排限流**

**如果存在超充枪铜排限流，在给充电枪配额功率节点时要尽量分散在不同的功率池。**

*typedef enum*

*{*

*CRITERION\_NONE = 0,*

*CRITERION\_PRIOR,          // 车优先级*

*CRITERION\_MIN\_COST,       // 经济运行*

*CRITERION\_MAX\_POWER,      // 最大需求*

*CRITERION\_EQU\_SANITY,     // 均衡健康度*

*CRITERION\_LMT\_CAPACITY,   // 单铜排限电流容量*

*CRITERION\_MAX\_EFFICIENCY, // 模块效率*

*} CRITERION;*

在程序中，分配策略为以上考虑维度的逻辑组合结果。

### 5.3.4 调试可视/FR04\_Vislize\_Dbg

​ ■**数据采集与发送（PCU目标端）​​：**

●在RTT机制中开通一个特殊通道（如terminal01），不与PCU程序正常printf定向打印冲突。

●在功率分配系统的不同关键时刻（如功率分配算法执行后、设备状态更新时）调用RTT API（如 SEGGER\_RTT\_Write）打印数据或刷新图形到终端。

●对于CCU桩侧功率需求的异步事件，采用非阻塞模式写入以避免影响系统实时性（目前demo暂采用CLI模拟CCU来车）。

​ ■**数据接收与解析（主机端）​​：**

●使用 ​J-Link RTT Viewer、配套脚本或自定义主机软件连接目标系统，读取RTT打印数据。

​ 双向交互实现​：

●主机端监听下行缓冲区，目标系统定期读取并解析来自主机的命令，执行相应的参数修改或控制操作。

**■双向交互实现​：**

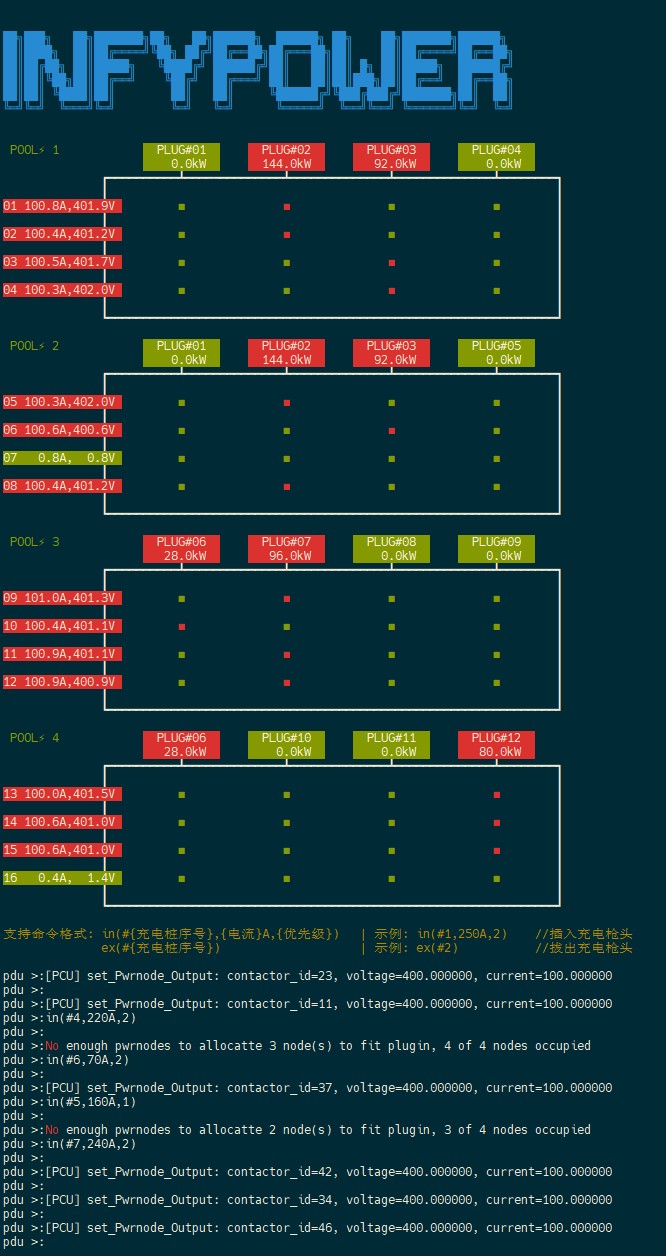
●目标机端监听下行缓冲区，读取并解析来自主机的命令，执行相应的参数修改或控制操作。

目前demo由CLI命令行解析器实现。在系统硬件条件未满足（比如工装输出功率无法匹配、无条件实现车辆充电）时可以由命令输入代替工况运行，验证软件工作逻辑是否正确。

●在系统运行期间，也可以借助可视化调试交互来监视当前设备功率分配运行工况，辅助定位问题原因。

●在软件自测过程中可以通过主流终端软件（SecureCRT、Moba Xterminal、Xshell）软件中集成的宏操作功能来通过命令输入实现准自动化测试。

交互视图如下所示：



### 5.3.5 可追溯日志/ FR05\_TracableLog

**。。。**

# 软件安全策略

由于功率分配静态库担负PCU软件核心计算任务，一旦出现软件安全问题会导致整个软件无法正常使用，所以在软件设计时增加以下几个安全策略，确保静态库的健壮性。

## 6.1统一管理外部变量

■静态库程序与外部工程程序间只保留一个注册函数统一管理接口（函数，数组，变量，指针等）

这样做有如下优点：

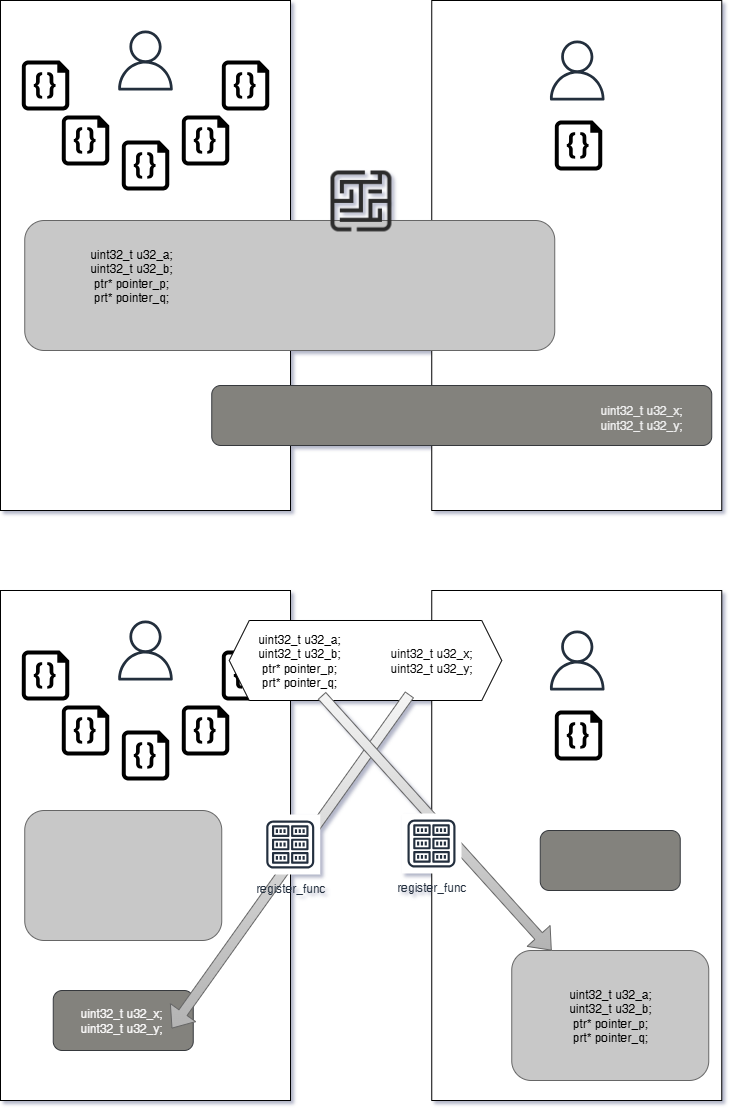
**1. 避免 extern 声明**：静态库和外部工程代码间无需知道对方符号的存在，通过名称查询即可，降低耦合，提高可维护性，静态库只是在自己作用域内拷贝了一个需要extern变量的指针引用副本

**2. 控制暴露范围**：只有通过 EXPORT\_COPYOUT\_OF( ) 注册过的符号才能被静态库获取，注册的对象仍在各自作用域内，作用域范围不改变

**3. 灵活性高**：符号名称是字符串，可动态查询（后续可根据配置文件决定需要哪些符号）

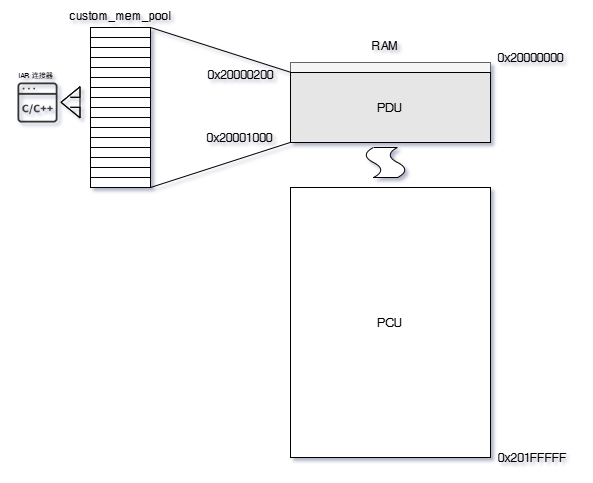
**4. 可注册动态值**：通过extern声明方式只可以对编译期间的显式固定变量有效，一些软件运行后才进入生命周期的变量（比如通过malloc申请的地址、运行期间可变的回调函数入口）则可以通过注册函数管理

**5. 可多次调用**：通过extern声明方式只能在链接器链接时一次性生效，注册函数则可以反复调用，在合适的位置和时机将接口对象告知静态库



## 6.2专有RAM

■在IAR编译环境下设置一块专有RAM区域给功率分配静态库，与PCU程序使用的RAM区域隔离，在编译静态链接库的布局文件.icf中划定一段地址，0x20000000 ~ 0x200000000+512用于PDU存放全局变量和静态变量；0x20000000+512 ~ 0x20000000+512+4\*1024为PDU的“Heap”，从连接器的角度来看这个Heap被一个4K的uint8\_t数组塞满。用于存放gpNodesArray、gpPlugsArray、gpContactorsArray三个弹性数组。这样也方便实现静态类聚（Static Clustering），即有利于连接器将同一模块的全局和静态变量分配到内存相邻地址，减少内存访问开销。



define symbol \_\_ICFEDIT\_region\_CUSTOM\_RAM\_start\_\_ = 0x20000000;

define symbol \_\_ICFEDIT\_region\_CUSTOM\_RAM\_length\_\_ = 0x000000200; // 512 B

define symbol \_\_ICFEDIT\_region\_CUSTOM\_HEAP\_length\_\_ = 0x00001000; // 4kB

define symbol \_\_ICFEDIT\_region\_ROM\_start\_\_ = 0x08000000;

define symbol \_\_ICFEDIT\_region\_ROM\_end\_\_ = 0x080FFFFF;

define symbol \_\_ICFEDIT\_region\_RAM\_start\_\_ = \_\_ICFEDIT\_region\_CUSTOM\_RAM\_start\_\_ + \_\_ICFEDIT\_region\_CUSTOM\_RAM\_length\_\_ + \_\_ICFEDIT\_region\_CUSTOM\_HEAP\_length\_\_;

define symbol \_\_ICFEDIT\_region\_RAM\_end\_\_ = 0x2001FFFF;

define symbol \_\_ICFEDIT\_region\_CCM\_start\_\_ = 0x10000000;

define symbol \_\_ICFEDIT\_region\_CCM\_end\_\_ = 0x1000FFFF;

define symbol \_\_ICFEDIT\_size\_cstack\_\_ = 0x1000;

define symbol \_\_ICFEDIT\_size\_heap\_\_ = 0x2000;

define symbol \_\_ICFEDIT\_size\_ccm\_heap\_\_ = 0x800;

define symbol \_\_PDU\_CORE\_RAM\_start\_\_ = \_\_ICFEDIT\_region\_CUSTOM\_RAM\_start\_\_;

define symbol \_\_PDU\_CORE\_RAM\_end\_\_ = \_\_ICFEDIT\_region\_CUSTOM\_RAM\_start\_\_ + \_\_ICFEDIT\_region\_CUSTOM\_RAM\_length\_\_;

define symbol \_\_PDU\_CORE\_RAM\_size\_\_ = \_\_ICFEDIT\_region\_CUSTOM\_RAM\_length\_\_;

define symbol \_\_PDU\_CORE\_HEAP\_start\_\_ = \_\_PDU\_CORE\_RAM\_end\_\_;

define symbol \_\_PDU\_CORE\_HEAP\_end\_\_ = \_\_PDU\_CORE\_HEAP\_start\_\_ + \_\_ICFEDIT\_region\_CUSTOM\_HEAP\_length\_\_;

define symbol \_\_PDU\_CORE\_HEAP\_size\_\_ = \_\_ICFEDIT\_region\_CUSTOM\_HEAP\_length\_\_;

define memory mem with size = 4G;

define region ROM\_region = mem:[from \_\_ICFEDIT\_region\_ROM\_start\_\_ to \_\_ICFEDIT\_region\_ROM\_end\_\_];

define region RAM\_region = mem:[from \_\_ICFEDIT\_region\_RAM\_start\_\_ to \_\_ICFEDIT\_region\_RAM\_end\_\_];

define region CCM\_region = mem:[from \_\_ICFEDIT\_region\_CCM\_start\_\_ to \_\_ICFEDIT\_region\_CCM\_end\_\_];

define region PDU\_CORE\_RAM\_region = mem:[from \_\_PDU\_CORE\_RAM\_start\_\_ to \_\_PDU\_CORE\_RAM\_end\_\_];

define region PDU\_CORE\_HEAP\_region = mem:[from \_\_PDU\_CORE\_HEAP\_start\_\_ to \_\_PDU\_CORE\_HEAP\_end\_\_];

define block CSTACK with alignment = 8, size = \_\_ICFEDIT\_size\_cstack\_\_ { };

define block HEAP with alignment = 8, size = \_\_ICFEDIT\_size\_heap\_\_ { };

define block CCM\_HEAP with alignment = 8, size = \_\_ICFEDIT\_size\_ccm\_heap\_\_ { };

place in PDU\_CORE\_RAM\_region { section .pdu\_ram\_section };

place in PDU\_CORE\_HEAP\_region { section .pdu\_heap\_section };

place at address mem:\_\_ICFEDIT\_region\_ROM\_start\_\_ { readonly section .intvec };

keep { section .pdu\_ram\_section };

place in ROM\_region { readonly };

place in RAM\_region { readwrite,

section .data,

section .bss,

section .non\_init };

place in RAM\_region { block CSTACK, block HEAP };

place in CCM\_region { block CCM\_HEAP,

section .ccm\_data };

define symbol \_\_VECTOR\_TABLE = \_\_ICFEDIT\_region\_ROM\_start\_\_;

export symbol \_\_PDU\_CORE\_RAM\_start\_\_;

export symbol \_\_PDU\_CORE\_RAM\_end\_\_;

export symbol \_\_PDU\_CORE\_RAM\_size\_\_;

export symbol \_\_PDU\_CORE\_HEAP\_start\_\_;

export symbol \_\_PDU\_CORE\_HEAP\_end\_\_;

export symbol \_\_PDU\_CORE\_HEAP\_size\_\_;

## 6.3关键数据保护机制

■为防止专有RAM内数据被其他内存区数据破坏（本方案中专有内存区置于0x20000000，则主要防止上溢），在几个关键数据结构体中的前后各置一个保护值。功率分配静态库在正常运行过程中每次进入主逻辑函数前会检查关键数据是否安全，如保护值被“破坏”，则说明被保护的关键数据被“污染”，功率分配静态库立即停止运行，并告知PCU当前静态库不可用。



typedef *struct*

{

*uint32\_t* **front\_canary;** // absolutely required to be top element

***size\_t*** length;

**CRITERION** criterion;

*struct* **Alloc\_plugObj** obj\_array*[]*;

} **Alloc\_plugArray**;

*#define* IS\_FRONT\_CANARY\_INTACT(ptr, type) \

    ((ptr) ? (((*const* type \*)(ptr))->front\_canary == (FRONT\_MAGICWORD)) : false)

*#define* GET\_REAR\_CANARY\_PTR(ptr, type) \

    ((*uint32\_t* \*)((*char* \*)(ptr) + sizeof(type) + (ptr)->length \* sizeof(((type \*)0)->obj\_array[0])))

*#define* IS\_REAR\_CANARY\_INTACT(ptr, type) \

    ((ptr) ? (\*GET\_REAR\_CANARY\_PTR(ptr, type) == (REAR\_MAGICWORD)) : false)

bool hear\_Canaries\_Twittering(void)

{

    if (!IS\_FRONT\_CANARY\_INTACT(gpNodesArray, Alloc\_nodeArray))

    {

        printf("gpNodesArray Front canary corrupted!\r\n");

        return false;

    }

    if (!IS\_REAR\_CANARY\_INTACT(gpNodesArray, Alloc\_nodeArray))

    {

        printf("gpNodesArray Rear canary corrupted!\r\n");

        return false;

    }

    if (!IS\_FRONT\_CANARY\_INTACT(gpPlugsArray, Alloc\_plugArray))

    {

        printf("gpPlugsArray Front canary corrupted!\r\n");

        return false;

    }

    if (!IS\_REAR\_CANARY\_INTACT(gpPlugsArray, Alloc\_plugArray))

    {

        printf("gpPlugsArray Rear canary corrupted!\r\n");

        return false;

    }

    if (!IS\_FRONT\_CANARY\_INTACT(gpContactorsArray, Alloc\_contactorArray))

    {

        printf("gpContactorsArray Front canary corrupted!\r\n");

        return false;

    }

    if (!IS\_REAR\_CANARY\_INTACT(gpContactorsArray, Alloc\_contactorArray))

    {

        printf("gpContactorsArray Rear canary corrupted!\r\n");

        return false;

    }

    return true;

}

## 6.4专用堆内存分配

■由于充电枪信息集、功率节点信息集、接触器信息集是在一块“专有Heap”上开辟内存空间创建的弹性数组，所以PCU软件调用的IMAP平台通用malloc()、calloc()、free()等函数不再适用，需要针对“专有Heap”定制一个简易的内存分配函数。该函数不需要内存分配策略，仅实现能将返回内存地址定位到“专有Heap”中即可。

*void* \*pdu\_calloc(*size\_t* size)

{

*static* *size\_t* custom\_mem\_offset IN\_PDU\_RAM\_SECTION = 0;

*if* (!size)

    {

        custom\_mem\_offset = 0;

*return* NULL;

    }

    size = (size + 3) & ~3; // allign

*if* (custom\_mem\_offset + size > CUSTOM\_MEM\_POOL\_SIZE)

    {

*return* NULL;

    }

*void* \*ptr = (*void* \*)(custom\_mem\_pool + custom\_mem\_offset);

    custom\_mem\_offset += size;

    memset(ptr, 0, size);

*return* ptr;

}

# 接口定义

## 7.1硬件接口

本软件集成于PCU软件中，硬件接口在IMAP平台定义。

## 7.2软件接口

通过注册函数将PCU软件中以下函数地址导入功率分配静态库中

|  |  |
| --- | --- |
| typedef NodeInfoPtr (\*func\_subscribe\_pwrnode)(*uint8\_t*); | 通过编号获取功率节点信息 |
| typedef PlugInfoPtr (\*func\_subscribe\_pwrplug)(*uint8\_t*); | 通过编号获取充电枪信息 |
| typedefContactorinfoPtr (\*func\_subscribe\_contactor)(*uint8\_t*); | 通过编号获取接触器信息 |
| typedef *void* (\*func\_set\_pwrnode)(*uint8\_t*, *float*, *float*); | 通过接触器编号预设功率节点电压、电流 |
| typedef void (\*func\_alarm)(…); | 告警上报 |
| typedef void (\*func\_trace\_log)(char\*); | 打印输出回调 |
| … | … |

具体细则参看《PDU功率分配静态库使用说明.md》。

# 软件设计计划表

时间计划表

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **任务** | **开始时间** | **结束时间** |
| **第一阶段** | | | |
| 1 | 概要设计 | 2025.08.01 | 2025.08.30 |
| 2 | 编码（拓扑构建） | 2025.09.01 | 2025.09.16 |
| 3 | 编码（分配策略） | 2025.09.17 | 2025.10.15 |
| 编码（其他功能） | 2025.09.17 | 2025.10.20 |
| 4 | 调试 | 2025.10.20 | 2025.11.30 |

# 附录