香港科技大学 ENTERPRIZE 战队

RM2024 超级电容控制器

The Hong Kong University of Science and Technology

完全开源

硬件方案&软件方案

ROBOMASTER

机甲大师超级对抗赛

技术方案

设计者：蒋益诚 戎羿

文档撰写：戎羿

目录

1 序言

在 RoboMaster 机甲大师赛事中，严苛的底盘功率限制规则迫使各参赛队伍从软件和硬件方面研究高效的底盘功率管理方案。早在 2018 年末，大连交通大学 TOE 战队，桂林电子科技大学 Evolution 战队与香港科技大学 ENTERPRIZE 战队等便相继开源了各具特色的超级电容方案。2020 年，官方裁判系统超级电容管理模块与相关限制规则加入，超级电容的使用更加规范化，给各战队带来了不小的挑战。

2021 赛季末大连理工凌_BUG 战队开源了他们基于双向 BUCK-BOOST 电路构建的超级电容控制模块，引发广泛关注。我队经过研发在 2023 年在其基础上完成了低成本，稳定可靠，大功率，高效率的数控超级电容方案。然而其并不完美，于是在 RM2024 中我队又改进得到了新一版的超级电容控制模块，在此开源硬件与软件方案，希望能和各位读者共同分享、交流学习。

RM 中的传承一直是一个很重的任务，尤其是对于如超级电容这种项目，涉及的方面较多、也很难定向培养新人。在这个背景下，这篇报告在笔者看来同时兼有教程的作用，覆盖更多的细节帮助新接触这一项目的同学们快速上手，因此篇幅可能会较长。

2 方案总览

2.1 继承 RM2023 开源

今年的超级电容设计总体继承了 ENTERPRIZE 在 RM2023 的超级电容方案开源，选用了相同的硬件拓扑与软件架构。

可以通过链接访问 RM2023 超级电容开源：

https://github.com/hkustenterprize/RM2023-SuperCapacitor

因此本文档会简要概括并略去部分方案原理分析。

相较 RM2023 开源的项目，本项目最大的特点便是体积缩小了非常多，且功率得到了提升。

2.2 充放电模块电路方案

自 2018 年超级电容方案诞生以来，其大致电路方案可分为以下 3 种

1）电源 ⇒ 恒功率充电模块 ⇒ 超级电容 || 底盘电机
2）电源 ⇒ 恒功率充电模块 ⇒ 超级电容 ⇒ 恒压放电模块 ⇔ 底盘电机
3）电源 || 底盘电机 ⇔ 双向可控功率模块 ⇔ 超级电容

本开源方案使用了第三种设计：底盘电机直接连接到电源，超级电容和一个双向可控功率模块共同构成一个可控的功率补偿系统，凭借高速的功率闭环控制动态对底盘功率进行削峰填谷，从而实现超级电容的能量缓冲。

该方案具有如下几个主要优点：

a）并联接入底盘电机母线，即使系统工作异常，只需要及时切断电路（关断 MOS 管/保险丝熔断等），底盘任可继续正常工作。
b）无论电容剩余能量如何，底盘电机的母线电压都能保持相对稳定，约等于电源电压，能够给电机提供良好的工作环境。
c）可以主动限制动能回收的电流，允许部分浪涌电流倒灌回电池，防止损坏超级电容组。
d）通过简单的代码逻辑，可在底盘电源切断时立即关断系统输出，在机器人阵亡后保存电容能量，同时避免违反规则。

该方案的显著缺点如下：

a）采用数控方案，对嵌入式程序设计，控制系统设计要求较高，稍有不慎即可能损伤硬件。
b）采用并联方式被动补偿功率，若底盘电控不加以合理限制，仍有超功率风险。

另外两种设计的分析可以参考 RM2023 超级电容开源报告，此处为了篇幅考虑不多分析。

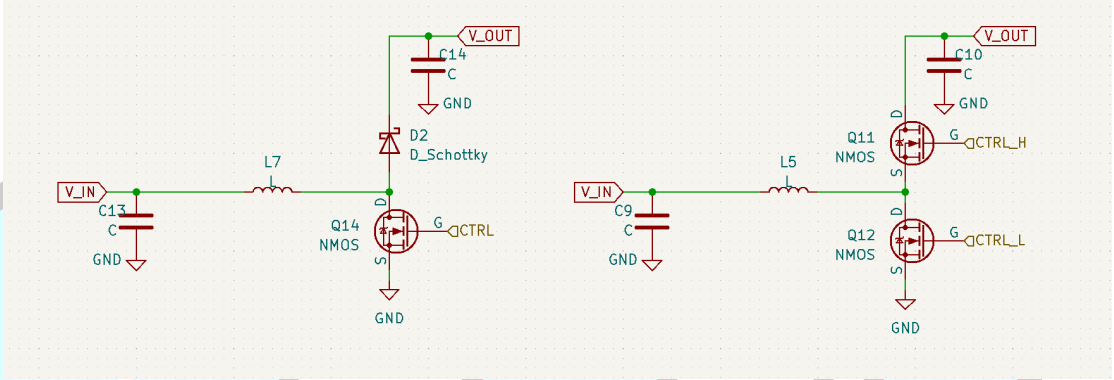
2.3 DC-DC 电路拓扑分析

前文提到，实现 RoboMaster 比赛中超级电容功率控制的关键在于负责充放电的双向可控功率模块。

常用的可控 DC-DC 拓扑有 Buck，Boost，四开关 Buck-Boost(FSBB)等。

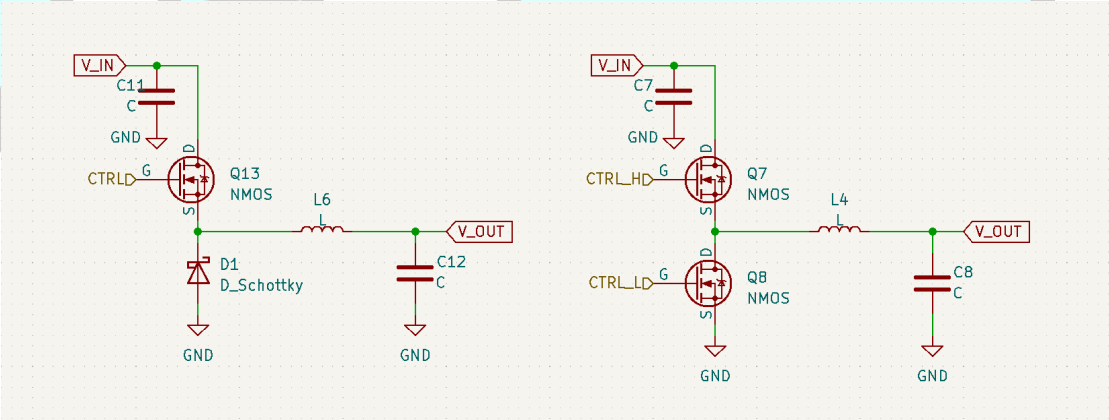
接下来我们进行不同拓扑的比较：

a）使用 Boost 拓扑为电容升压充电（从底盘网络->电容组为升压），而降压放电（从电容组->底盘的电流路径为降压）。



若底盘->电容的电流路径为升压，则当电容电压低于底盘电压时无法放电。由于比赛规则对电容组充电能量和电压有限制，这个方案会浪费很大一部分能量在电容组的低电压区(可用区间仅有 5V 左右)。故不适用。

b）使用 Buck 拓扑为电容降压充电（从底盘网络->电容为降压），而升压放电（从电容->底盘的电流路径为升压）。



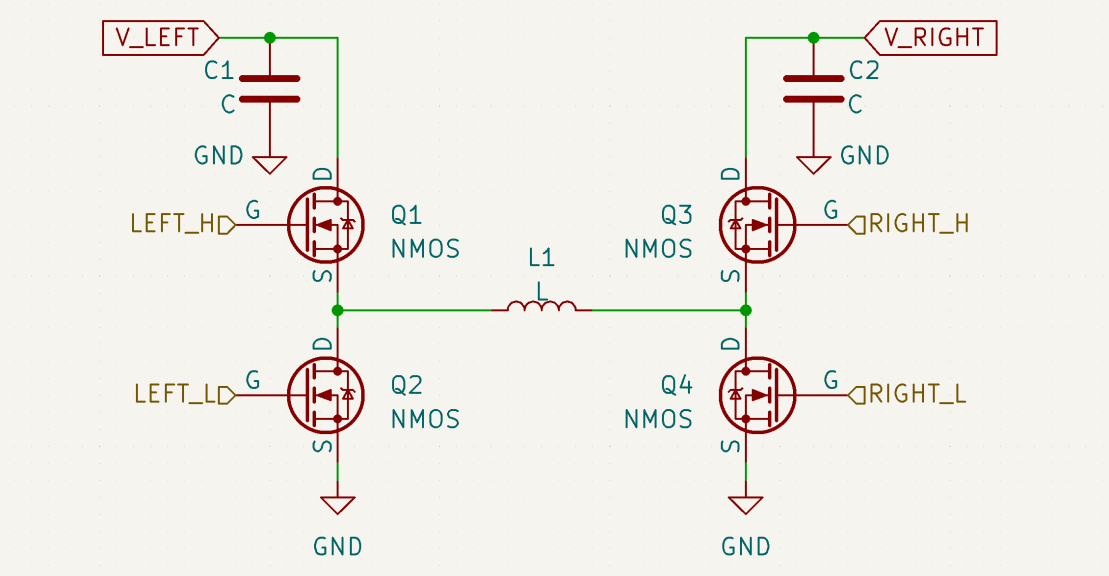
若底盘->电容的电流路径为降压，则电容最高充电电压为电池电压*0.9 左右；此方案的优点是拓扑结构简单，电容串数比较少；而缺点是由于最大电流限制，放电功率上限较低，同时也使得在相同功率下损耗较大。

另外，当底盘电压和电容组电压接近时，本拓扑很难控制电流流向（在电容电压高于底盘时，没法控制电流不回流）。一般在设计这种拓扑的超级电容控制器时都会额外多加

一部分 MOS 用于控制电流流向，并不能真正实现两开关 Buck 相对 FSBB 更节省物料的优点。

需要注意：示意图中同时显示了非同步 Buck/同步 Buck，实际中由于非同步 Buck 没办法反向升压，因此采用的都是右侧的同步 Buck。

c）使用 FSBB 拓扑为电容组升降压充电、放电。



作为 RM 比赛中超级电容控制器的主流方案，四开关 Buck-Boost 双向升降压电路可以不受电压限制地控制电流流向。

当左侧（A 侧）电压高于右侧（B 侧）电压时，100%常开 A 半桥，用 Buck 的形式控制 B 半桥；

当 B 侧电压高于 A 侧电压时，100%常开 B 半桥，用 Boost 的形式控制 A 半桥；

在需要的左电压与右电压相近时，由于最大占空比以及死区时间等限制，很难单独使用 Buck 或者 Boost 拓扑生成所需电压比。于是考虑用 Buck 控制策略控制左半桥，用 Boost 控制策略控制右半桥，此即为 buck-boost 电路。在任意时刻（半个周期内），电路可视为 Buck 电路或 Boost 电路。（此处省略了一些细节，如为了防止两边的下管同时导通，左右两边的上管占空比应相互制约，同时相位也应错开）此时，无论是使用叠加原理还是通过伏秒平衡定律，我们都可以得到如下结果：$\frac{V_B}{V_A} = \frac{\text{duty}_A}{\text{duty}_B}$ 。

具体时序可以参考 4.2.1 与 4.5.2。

关于伏秒平衡等推导，此处篇幅限制不详细说明，读者可以自行查阅资料。

2.4 功能总览

- 双向充放电
- 三端电流、两端电压监测
  - 裁判系统输入电压电流双向测定
  - 内部 DCDC 变换器裁判系统侧双向电流测定
  - 内部 DCDC 变换器电容侧双向电流测定
- 蜂鸣器报警/提示音
- CAN 通信
- 可接收 CAN 数据包更新当前裁判系统数据（缓冲能量、规则功率上限）
  - 对于收到的裁判系统缓冲能量可以闭环，实现最大化能量利用
- 可反馈 CAN 数据包输出当前状态
  - 估计电容剩余能量
  - 当前底盘电机实际功耗（用来给底盘电机功率控制系统反馈）

2.5 赛场表现

2.5.1 性能表现

我队步兵和英雄机器人在 RM2024 的赛场上可以实现在一级血量优先（底盘功率限制 40W）的性能配置下实现稳定飞坡，超级电容控制器可以稳定地提供坡上加速需要的功率。

配合我队嵌入式功率控制的“功率补偿”策略下，通过在低等级下使用电容电量补偿底盘电机功率到 80W，可以以较高的灵活性与加速度移动的同时，超级电容长时间放电。

2.5.2 稳定性

本项目的一大特点在于比较高的稳定性，赛场上从未出现过超级电容坏掉的情况。

且充足的报错机制可以方便 Debug，上下板分离的设计也可以在物料紧缺的情况下快速更换。

即使在调试过程中，各种保护机制（如短路保护、DCDC 变换器保护、过压保护等）也可以在出错的情况下及时关电、提示。

调试过程中除了在写保护期间与 Debug 硬件问题（在下文 3.4.3 有提到），几乎没有出现过软件导致硬件损坏的情况。

3 硬件设计

3.1 外观

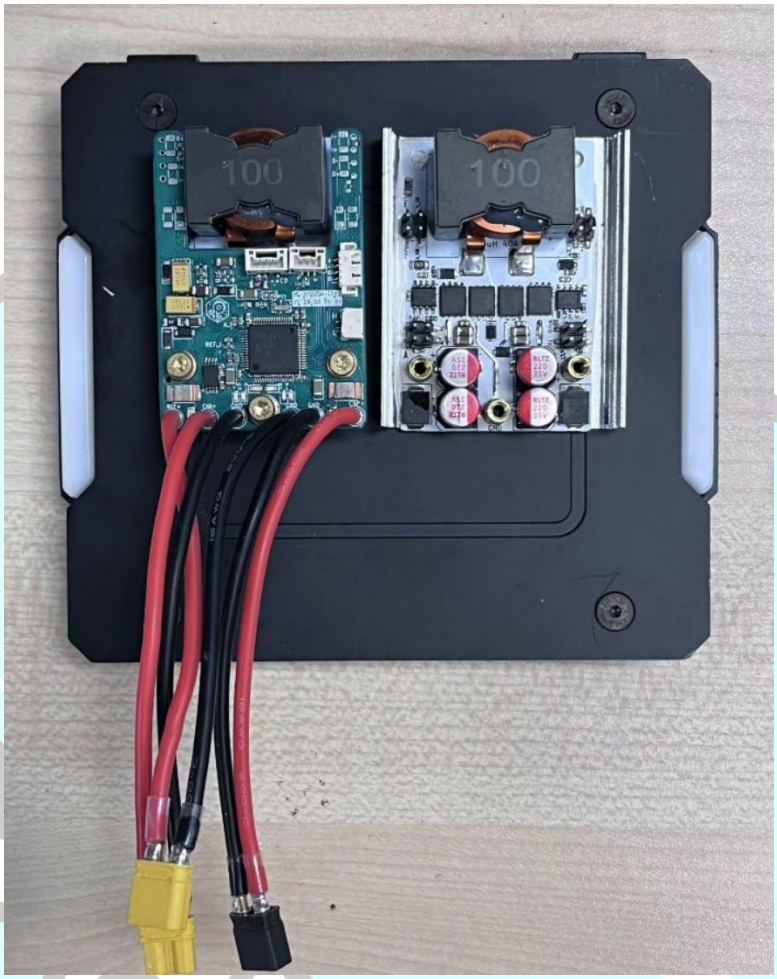
3.1.1 大小

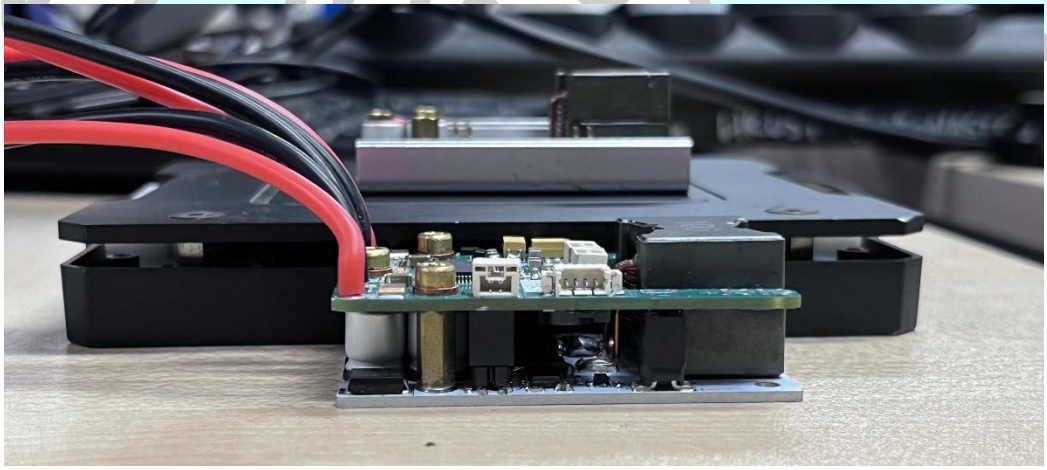
带外壳尺寸：

L=66mm

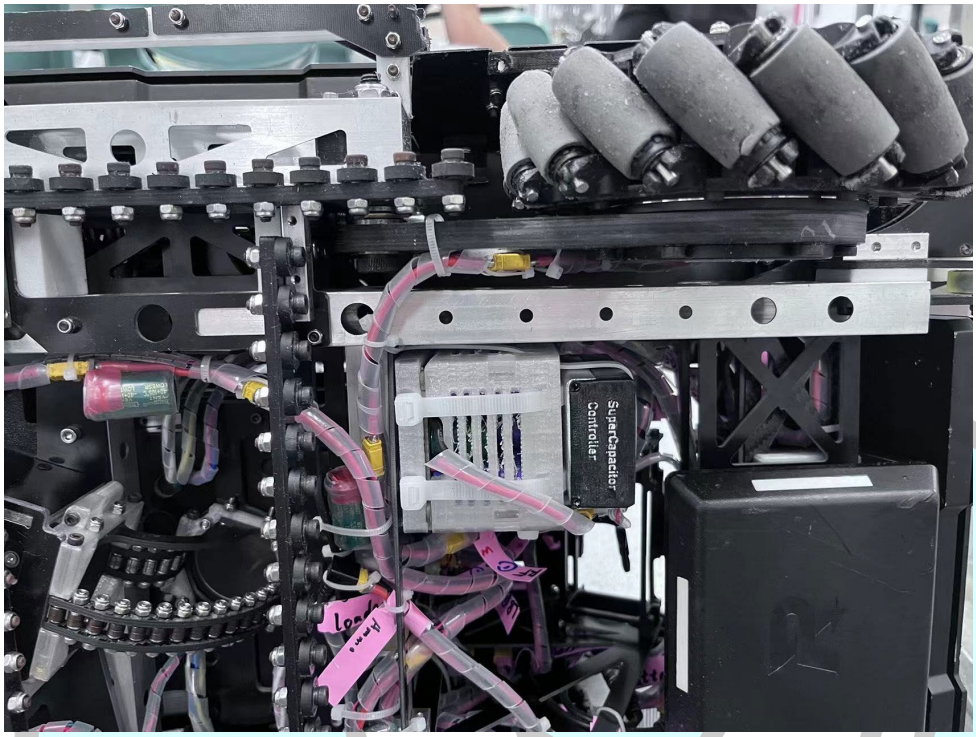
W=50mm

H=24mm

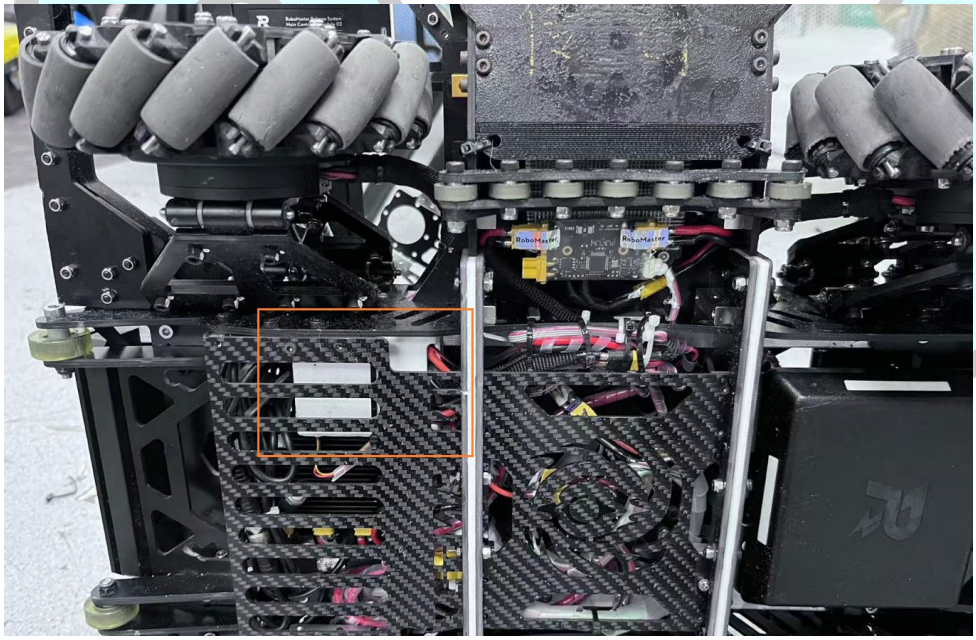




安装示范



如图为英雄机器人底盘上的超级电容控制器和电容组安装



如图为步兵机器人底盘上的超级电容控制器安装

模块是笔者见过的控制模块中体积较小者，且能够支持最高 24V*15A 的持续输入/输出，功率密度非常之高。

3.1.2 铝制下壳与打印上壳

此超级电容控制模块效率最高约为 97.5%。在处理大电流的情况下，不可避免地产生能量损耗，因而需要将热量传导至铝制部件来散热。

同时，铝基板 PCB 强度并不高，铜柱连接处与排针也是。为了保护机械强度，将下板粘合在半铝壳上。这样在碰撞、挤压之下也不会产生形变，也充当了散热组件

上壳使用 3D 打印，灵活性高，可以针对不同机器人设计做出不同模型。

能够实现全包裹，密封的设计可以防止异物进入，例如打磨产生的铝屑，碳粉，以及香港恶劣潮湿气候常见的水雾等。这对于超级电容这种大功率的模块的稳定性和安全性而言非常重要。

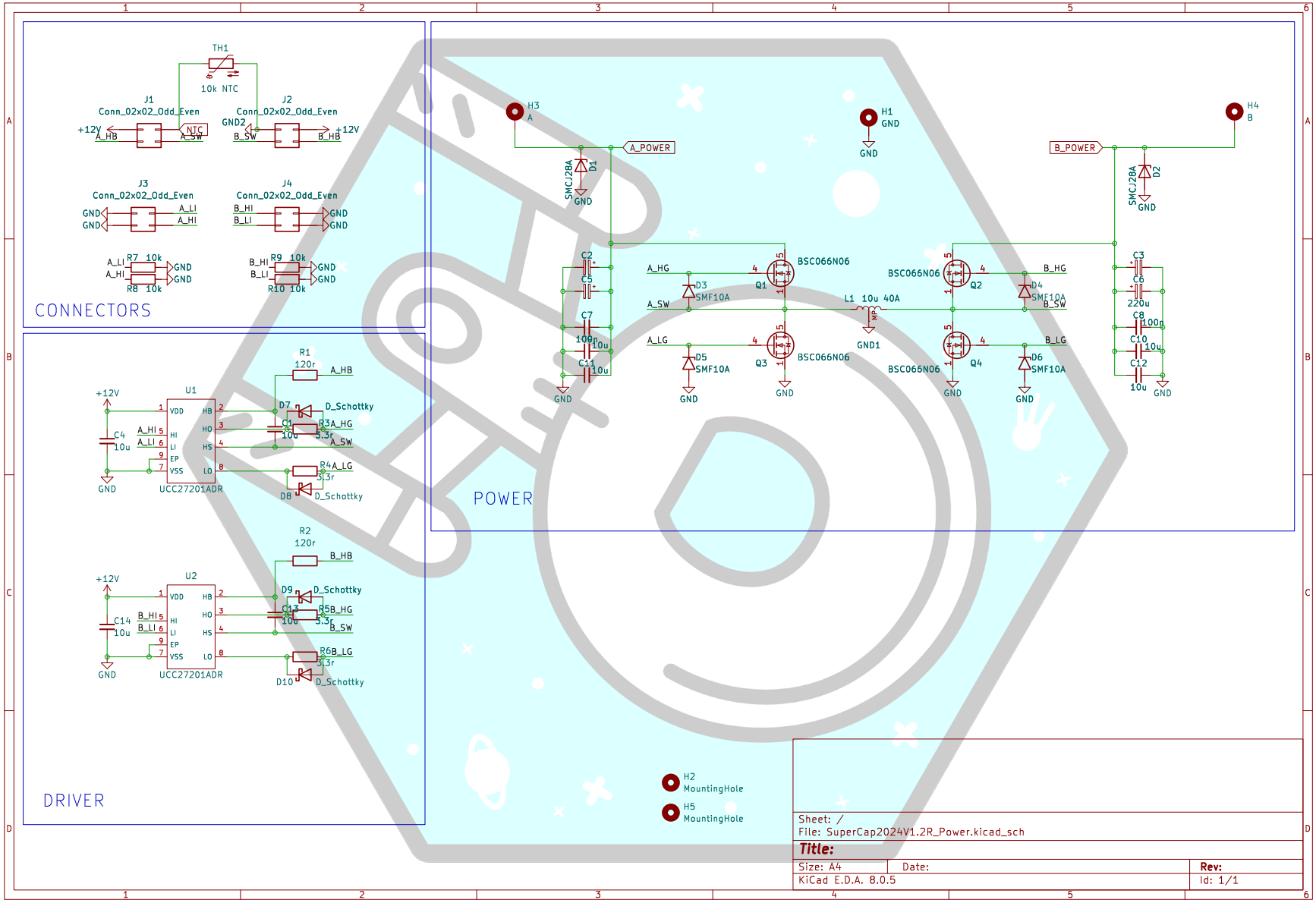
3.1.3 线路引出

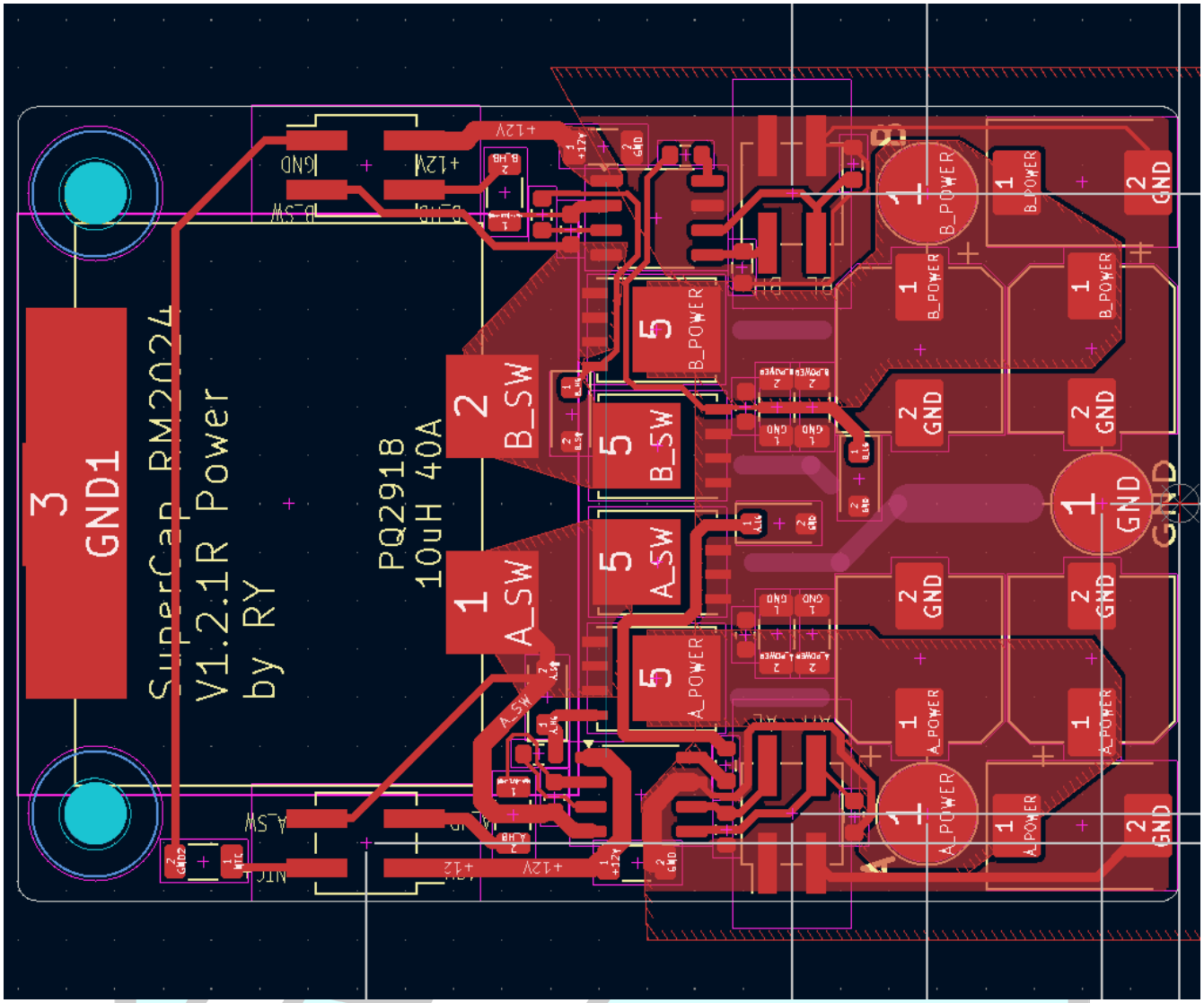
从 FR4 控制板的焊盘上接线引出。这样的设计能够减小 PCB 面积，且比较灵活，安装空间即使较小也能够方便地连接到外部电路，如底盘中心板、电容组等。

3.2 上下板方案

为了实现较好的体积表现与散热表现，使用上下板方案。既能兼顾铝基板的良好散热，也可以利用四层 FR4 板的灵活布线保证电源、信号等的可用性。

3.3 下板原理图与 PCB（下页）


CONNECTORS
DRIVER
POWER
TH1
10k NTC
J1
Conn_02x02_Odd_Even
J2
Conn_02x02_Odd_Even
J3
Conn_02x02_Odd_Even
J4
Conn_02x02_Odd_Even
+12V
GND
A_HB
A_SW
B_SW
B_HB
A_LI
A_HI
B_HI
B_LI
NTC
R7 10k
R8 10k
R9 10k
R10 10k
U1
U2
UCC27201ADR
R1
120r
R2
120r
R3 5.3r
R4 5.3r
R5 5.3r
R6 5.3r
C1 10u
C4 10u
C13 10u
C14 10u
D7
D8
D9
D10
D_Schottky
H3
A
H1
GND
H4
B
A_POWER
B_POWER
SMCJ28A
D1
D2
C2
C5
C7 100n
C11 10u
C3
C6 220u
C8 100n
C10 10u
C12 10u
A_HG
A_SW
A_LG
B_HG
B_SW
B_LG
D3 SMF10A
D4 SMF10A
D5 SMF10A
D6 SMF10A
Q1
Q2
Q3
Q4
BSC066N06
L1 10u 40A
GND1
H2
MountingHole
H5
MountingHole
Sheet: /
File: SuperCap2024V1.2R_Power.kicad_sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. 8.0.5
Id: 1/1



3.4 下板要点说明

下板即是功率板 Power Board ，材质为铝基板。带有 MOSFET、驱动电路、扁线功率电感、固态电容等。利用铝基板加强散热，使得超级电容控制模块可以持续输出 20A 以上大电流而保持不过热。

3.4.1 主功率回路

主功率回路是四个 MOS 组成的 Buck-Boost 电路，可以双向电流升降压。

使用 SMF10A TVS 来保护 MOS 的栅极受到可能的静电或振荡导致的击穿。SMF10A 的熔断电压为 20V，不超过 BSC066N06 的 Vgs 上限。

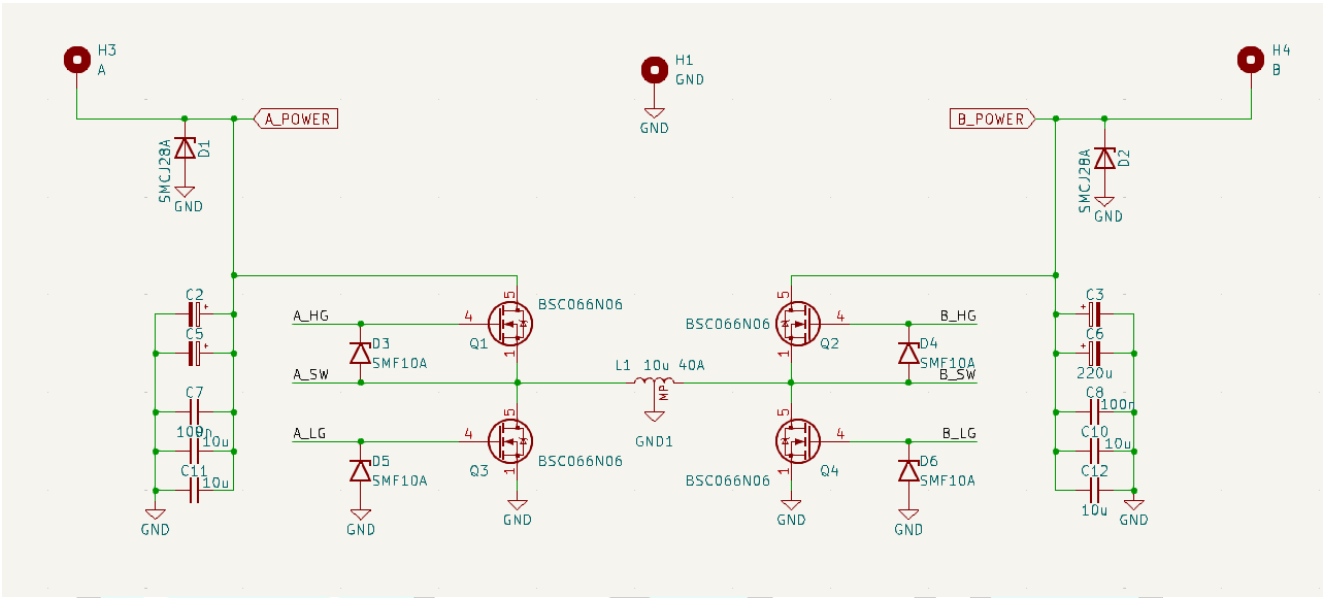
Layout 方面，采用了对称的 U 形设计，并且能够确保：

a) 高 dI/dt 回路可以尽可能小（如图，半桥的 VPP 和 GND 几乎贴在一起了）。近处使用 100nF 0603 的 MLCC 电容减小寄生电感对高频噪声的阻碍，远处再并联 10uF 1206 的 MLCC 提供较低频率噪声需要的电容值。这样可以最小化开关导致的 EMI 问题。
b) SW 面积尽可能小，减小耦合噪声

c）并联了贴片固态电容，能够在比较高的电压下接替 MLCC 来保持足够的滤波容值。且固态电容虽然成本较高，但是性能优越，不易烧毁、内阻很小。

d）大电流部分开窗上锡

至于 SW 节点的振荡，经测试正常开关的情况下振荡不大，不会影响其他部分。且用 RC Snubber 尝试之后发现并不能显著地优化振荡，因此取消了 RC 阻尼电路。



3.4.2 二极管

同时，也有尝试过在四个 MOS 上各并联肖特基二极管。为了面积考虑，使用 SR1060L 直插二极管直接搭棚并联于 MOS 上方。经过测试，能够比较显著地提升在轻载（大约 1A 电流下）的效率：从 97%提升至 97.3%。而重载下则没那么显著。

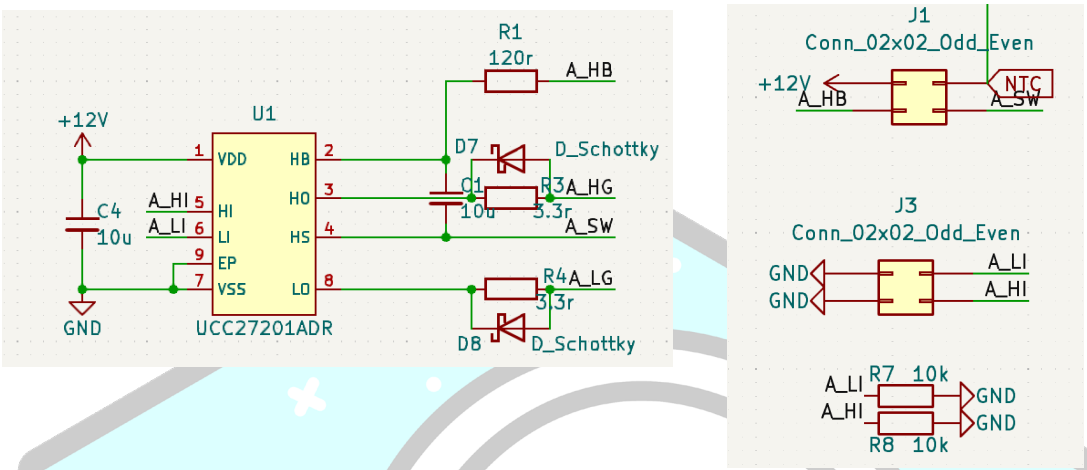
探究波形后发现，肖特基二极管的并联可以在死区较大的情况下让电流不经过 MOS 的体二极管而经过压降更小的肖特基二极管，因此提高效率。然而在重载下，一方面 MOS 的关断、开启波形变得更加陡峭，体现为死区更小；并且大电流下肖特基二极管的功耗损失也大。因此呈现如上结论。

最终综合表现与 PCB 面积、加工难度等，选择了不并联二极管。

（实际上市场上存在自带肖特基二极管的 MOS，但是种类较少，成本较高。各位若有兴趣可以尝试选用。）

3.4.3 MOS 驱动电路

MOS 的驱动也是很重要的一环。

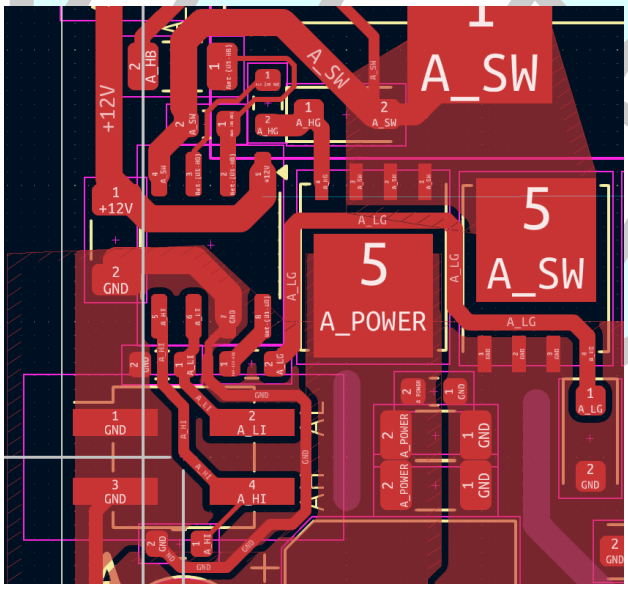


选用 TI 的 UCC272x1 系列芯片，能够提供较高的驱动电流且能够用自举电容来打开半桥上管。原理图比较简洁，理论上不需要额外的电路。

同时，为了满足“100%常开上管”的需求（为了提升单独 Buck/单独 Boost 时的效率），将驱动芯片的 HB 网络与开关电路的 SW 引脚引出。用排针连接到上板，在上板上接入隔离 12V DC-DC 模块，实现能够提供持续的电压差用来打开上管 MOS。隔离驱动需要的电流很小，因此接入一个电阻用来限流、防止静电等。

MOS 驱动芯片为 MOS 的栅极充放电，在高速的应用下电流很大、需要极小的寄生电感。因此这版功率电路的设计中将 MOS 驱动芯片也放置在了下板上，非常靠近 MOS。

虽然铝基板上的 Layout 比较困难，有些回路很难做到完美，但是总体带来的收益是较大的。



（例：一侧的 MOS 驱动电路）

Layout 方面优先将 BootStrap 电容与芯片尽可能靠近。而后让栅极到芯片的连线尽可能短（相对来说线路粗细反而关系不大）以减小振荡。

采用 1206 的 10uF 50V 去耦电容，因为担心 0603 封装的电容的寄生内阻太大、或受到 DC 偏置影响较大，导致芯片瞬间进入低压保护而出错。

使用 UCC27211 还是 UCC27201？——讲故事环节

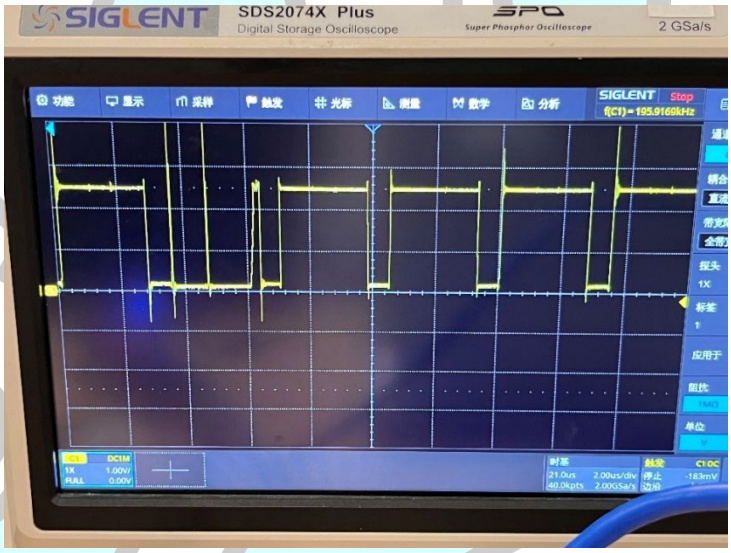
现实中使用的芯片型号为 UCC27201ADR。关于栅极驱动芯片的选择，笔者曾踩过好大一坑。

初始继承了去年的项目，继续使用实验室遗留的 UCC27211 芯片。在学长帮忙做了一版之后，调试一开始一切顺利，马上就到了能够上车的环节。

后来替换了新买的芯片，同时控制上也出现了问题，在特定方向、特定电压和特定占空比下电流闭环会跑飞，出现严重的振荡。

那个时候笔者排查重点在代码逻辑，然而并不能解决问题，每次都是看似好了之后又出现问题。因为硬件上多次更换过 MOS 与驱动芯片，且去年的项目用的是同一套，因此没有去想这方面。

最后终于怀疑到了 MOS 驱动了，经过测试发现在那个特定的环境下一侧 SW 的波形如图：



（图为 SW 网络的波形，出现了上管开不开的情况，且振荡很大）

又经过反复测试发现大概原因为 MOS 栅极出现了振荡。反复更换了上管的栅极电阻，发现当电阻为 15R 以上时很少见到这个问题。然而这个参数显然不符合 UCC27211 的 4A 驱动性能，因而怀疑了芯片的体质问题。

之前的芯片都是在某宝上购买，因为正版 UCC27211 很贵所以选择了 1~2 元的芯片。怀疑假芯片后，在某芯城上发现一批比较便宜的 UCC27201ADR 芯片（大约 2.5 元一片），经过测试之后可以在 3.3R 的栅极电阻下稳定运行而无问题。

最后得到的教训就是要多方面排查，且有时候不能贪小便宜。UCC27201 是早于 UCC27211 推出的版本，驱动电流是 3A 而非 4A，但是对于当前的应用已经足够（再高的速度可能会引起更大的 SW 振荡）。综合开发时间、成本控制等考虑，最后就选用了这个方案。而后不再有出现过类似的问题。

3.4.4 连接器

本项目的设计中，为了采样精确度考虑，电流采样电路设计在上板，因此需要实现上下板间大电流通过。使用了铜柱作为方案。

铜柱使用焊锡以贴片的形式焊接在下板上，具备一定的机械强度（当然使劲掰还是会碎，PCB 上铜毕竟附着强度有限），在外壳的保护下足以应付 RM 的工况。

上板通过铜制螺丝连接在下板上。为了防止出现振动脱落导致灾难性情况发生，在测试完毕装车前会上螺丝胶，只有加热的情况下才能把螺丝拧下来。

实际应用中没有出现这些问题，因此经过实测这个连接方案是较为可行的。

另外的非功率连接（如 MOS 驱动信号、NTC 反馈、隔离电源输出等）则使用贴片 2.54 排针排母实现。

3.4.5 电感选用

本方案设计最大电流为 25A. 由于负载底盘侧有很大的线路电阻，电容侧则是容性负载，因此对于输入输出纹波较不敏感。保证基本的纹波要求即可。

针对具体频率与电压电流所计算出电感取值的过程可以参考 RM2023 开源，本处篇幅限制不做推导。

在不超过饱和电流的前提下，电感取值越大，纹波电流越小，留出一定余量，实际中电感选择了 PQ2918 封装的 10uH 铁氧体扁线电感，标称最大电流为 40A，结合最大 25A 的软件电流限制是可行的。

铁氧体材质的磁芯优点是在高频开关下磁损失较小，缺点是若磁感应强度超过额定值，则电感值会立即大幅下降，导致电压变换器失控、损耗变大等，因此需要严格限流。

我们也考虑过其他材质的电感，如铁硅铝等。它们的特点正好相反，适合低频开关，但是电感值会缓慢下降，瞬间电流超过额定最大值也不会有事情。类似的选择还有羟基粉一体成型电感等，但是难以找到大额定电流的型号。

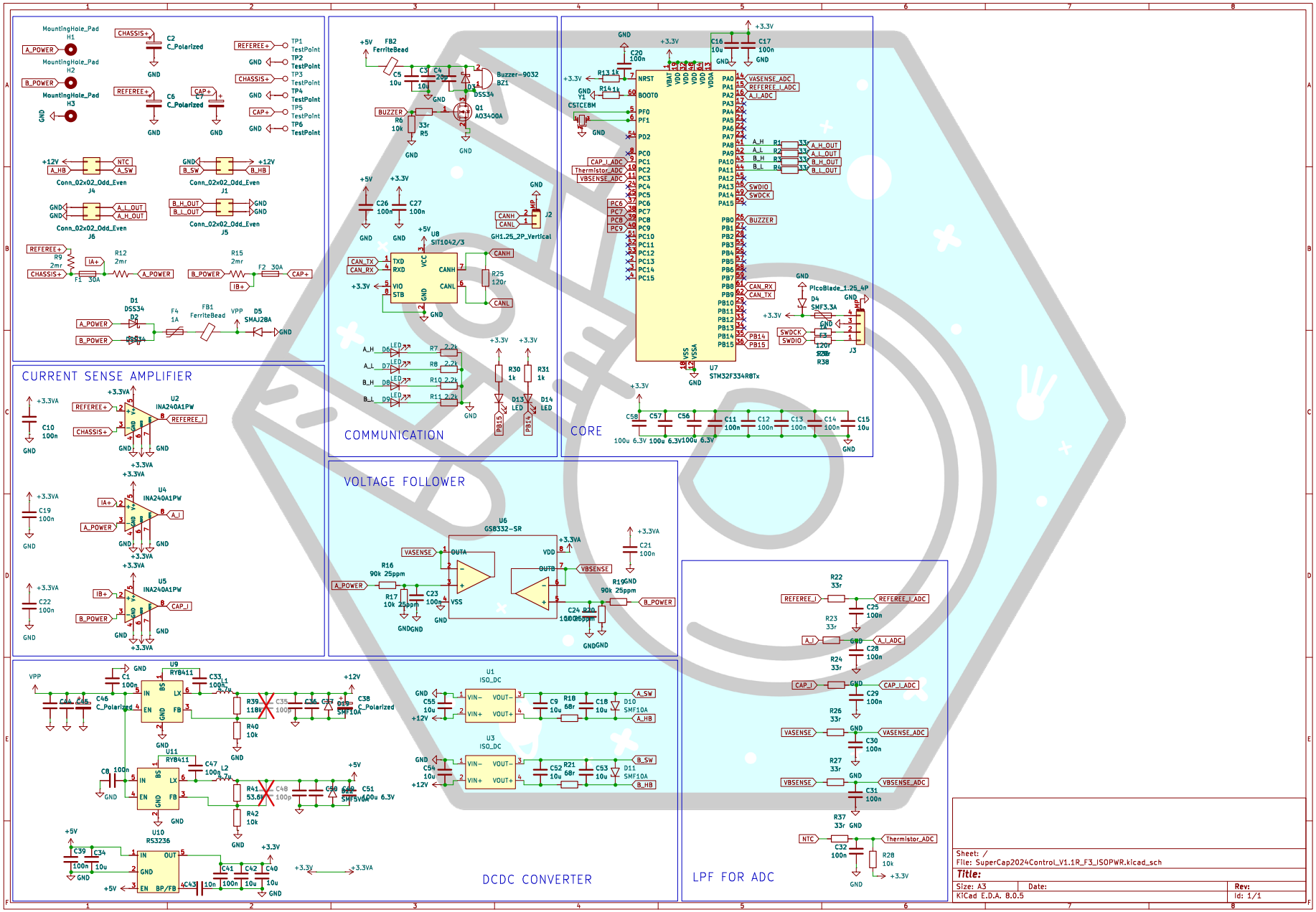
最后综合温升、饱和电流与成本考虑，还是选择了 PQ2918 封装的铁氧体贴片电感。经过测试能够承受 25A 的输出电流而正常控制、发热正常，符合选型要求。

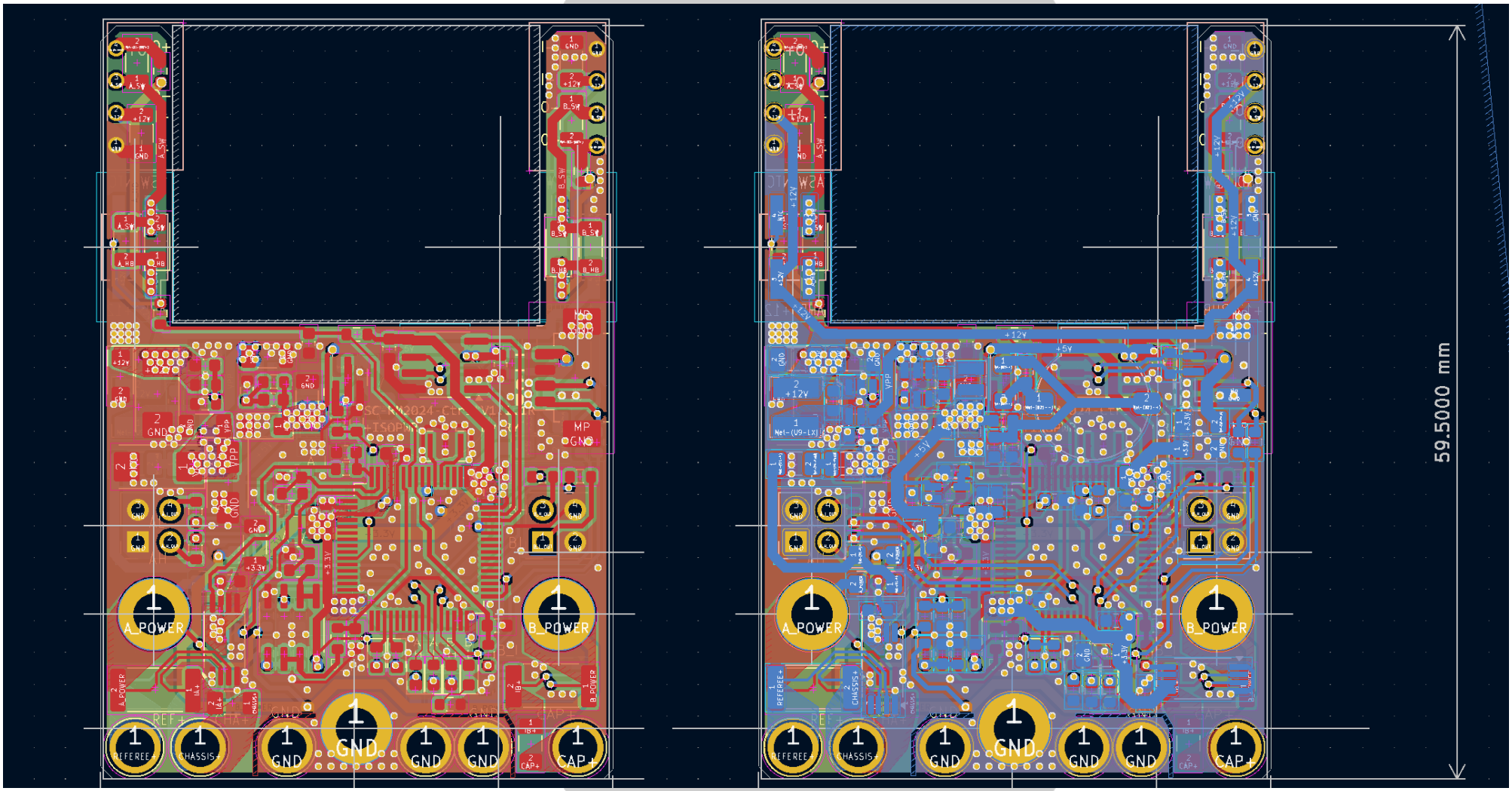
3.4.6 电容值计算

同样地，参照 RM2023 开源文件中的推导，我们选用每侧大约 450uF 的电容来滤波。

理想情况下，电容取值越大，纹波电压越小，留出一定余量，并考虑到 ESR 影响，实际应用中，我们使用 1 个 100nF 陶瓷电容，2 个耐压为 50V，额定容量为 10uF 的陶瓷电容与 2 个耐压 35V，容量为 220uF 的固态电容并联.

3.5 上板原理图与 PCB（下页）


CURRENT SENSE AMPLIFIER
COMMUNICATION
CORE
VOLTAGE FOLLOWER
DCDC CONVERTER
LPF FOR ADC
Sheet: /
File: SuperCap2024Control_V1.1R_F3_ISOPWR.kicad_sch
Title:
Size: A3
Date:
Rev:
KiCad E.D.A. 8.0.5
Id: 1/1


59.5000 mm
A_POWER
B_POWER
REFEREE+
CHASSIS+
GND
CAP+

3.6 上板要点说明

上板即是 Control Board ，材质为 FR4 四层板，负责控制，带有 MCU、信号调理电路、电流感应放大器电路、CAN 通信电路，以及相应的辅助供电电路等。

3.6.1 供电

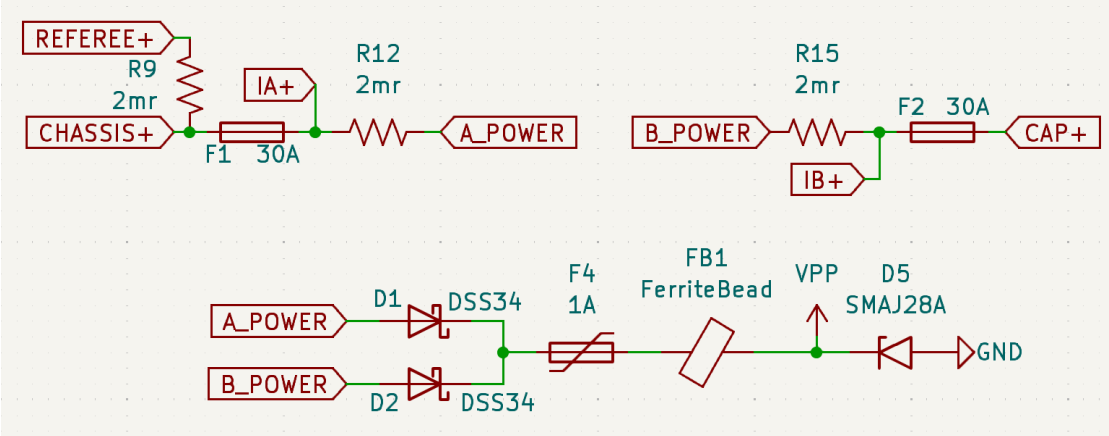
24V 转 10V 与 5V 采用 RY8411 作为供电芯片。RY8411 为同步 Buck，在 42V 耐压下能够提供最大 1A 的电流，效率较高，足够我们的使用场景。

与之 Pin2Pin 的 Buck 芯片有很多，例如 JW5026。经过测试，对于比较恶劣的输入电压环境（典型如热插拔高压供电线），RY8411 能够承受更高的电压尖峰而不烧毁。

RY8411 有轻载 PSM 模式，理论上纹波的表现相较于 FPWM 模式会差一些。然而一方面电路设计中使用了足够多、足够大的电容来滤波，且 STM32F334 在运行中的电流能够使其进入 CCM 模式中，因此实际测试中纹波表现并不差，5V 供电纹波大约为 110mV，12V 供电大约为 88mV。

为了应对功率电路可能出现的 EMI 干扰、以及插拔导致的电压尖峰，在 A 侧与 B 侧进入控制板后又经过了一次磁珠滤波。

3.6.2 电流路径



由上图说明，电流采样与保险丝均位于上板。采样分别有：

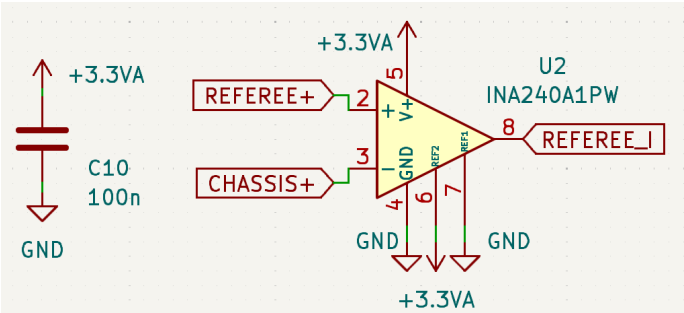
a） 自裁判系统电源管理模块输出->其他所有设备，称为 IReferee
b） DCDC 变换器 A 侧输入/输出电流，称为 IA
c） DCDC 变换器 B 侧输入/输出电流，称为 IB，同时也是给电容组充电的电流

有了这三路采样便足够实现我们需要的功能。

保险丝则分别设立于 A 与 B 侧输入/输出，特点是即使超级电容控制模块完全失效、保险丝断开，底盘依旧可以接收来自裁判系统电源的输入。确保了比赛中极端情况下机器人的基础功能可用性。

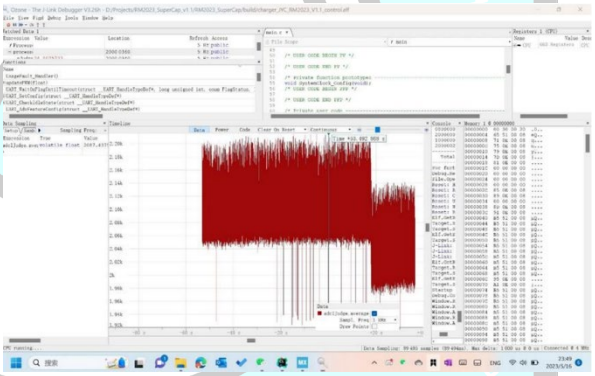
TVS 保护控制模块的输入电容与输入 DCDC 芯片，避免过压击穿。

3.6.3 电流采样放大器



去年的设计中使用了 INA186。其原因主要是当时难以买到 INA240（2022/23 年，INA186 价格大约为 3 元/片，INA240 则有可能达到几十元/片），从成本考虑上选择了 INA186。

今年则不一样，一方面 INA240 变得比 INA186 还便宜，同时 INA240 出色的 PWM 抑制功能相较 INA186 优势非常显著。且不需要 Bias 电源（因为 INA240 内置分压电阻，简单连接即可得到 1.65V 电压源）的特性大幅简化了设计。

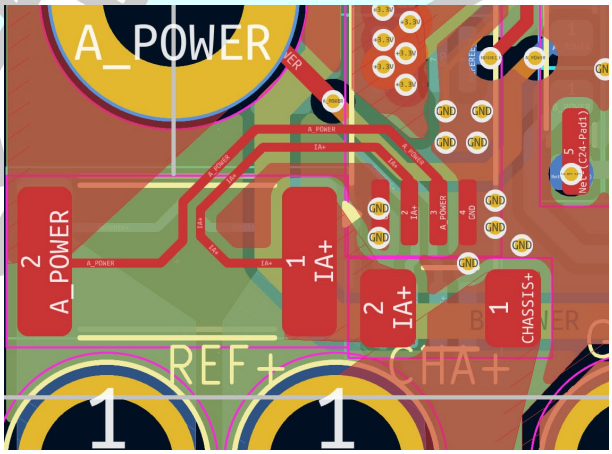


如图即是 INA186 放大后 ADC 采样得到的数据，未滤波时噪声严重。

而 INA240 即使不滤波，得到的数据也是比较平整的一条，且相较于大力滤波，数据的失真更少、能够更好更快地闭环控制。

细节上，选择了最低增益倍数的版本 INA240A1，不仅因为它的价格一般最低，且低增益的放大器一般具有更高的带宽。修改了采样电阻为 2mR，这样在 3.3V 的电源轨当中能够实现 ±40A 的电流量程。考虑到机器人底盘的瞬间爆发，这个大小之下的控制是有意义的。

Layout 方面，需要注意采样电阻的开尔文接法，即 ISense+与 ISense-两路信号不能与其他信号共用 PCB 路径。且两路信号的线路形成的面积尽可能小，为了防止可能的电磁干扰。



3.6.4 RC 滤波器

即为标准的 RC 低通滤波器。

一方面，需要将外部信号高于采样频率的部分去除，避免信号混叠；

另一方面，使用电容来满足 STM32 的 SAR-ADC 外设对外部信号的瞬时电流需求，避免因此导致的失真。（可见 4.3.2 章节）

选用了 33R 100nF 的组合，截止频率大约为 48kHz，小于采样频率 288kHz 的一半。

3.6.5 其他

使用 GS8552 实现对 A 侧、B 侧电压跟随，从而让 ADC 可以以较高的带宽对电压采样。

使用 SIT1042T/3 作为 CAN 收发芯片，电平可以兼容大疆电机的 CAN 网络。模块自带 120R 电阻，不需要额外配置终端电阻。

3.7 成本控制

这个项目并没有刻意去压低成本，不过依赖去年的项目（RM2023 开源超级电容）的基础上，仍然有很高的性价比。

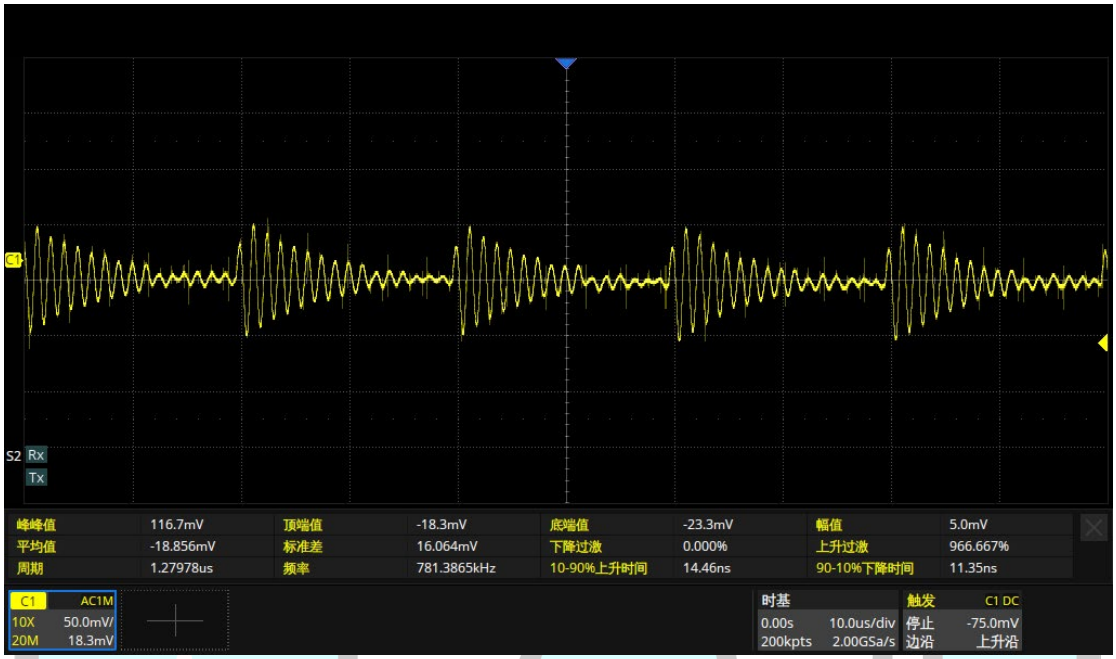
关键元件有：

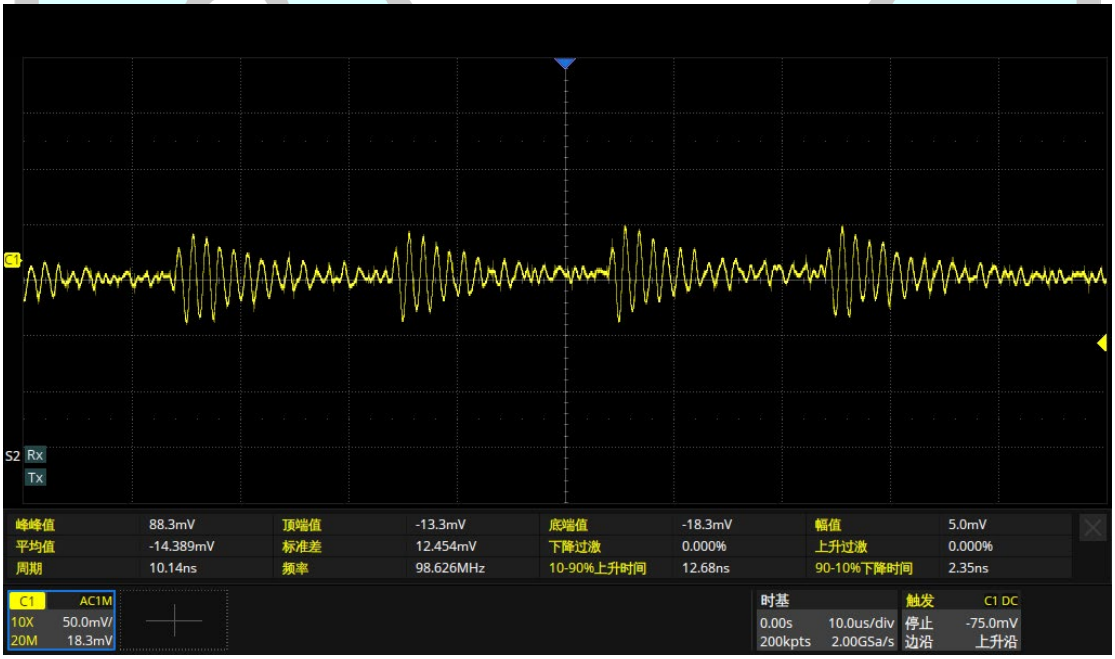
a）MCU-STM32F334R8，对比其他封装与型号的 MCU 来说，这款的成本是最低的。
b）MOSFET-BSC066N06，可以找到很多相对便宜的来源，大约 2 元一片，且性能优异。
c）MOS 驱动-UCC27201ADR，在芯城上可以得到约 2.5 元 1 片的价格。
d）电流采样放大器-INA240A1，现在经过降价后可以找到 2 元以内的。
e）PQ2918 扁线电感，有大约 10 元以内的来源。

总体上来说，这个项目内部的元件都以比较低的价格实现了应有的作用，而没有失去一些本该有的功能。成本控制方面比较好，大家可以以比较低的价格来复刻。

3.8 相关波形

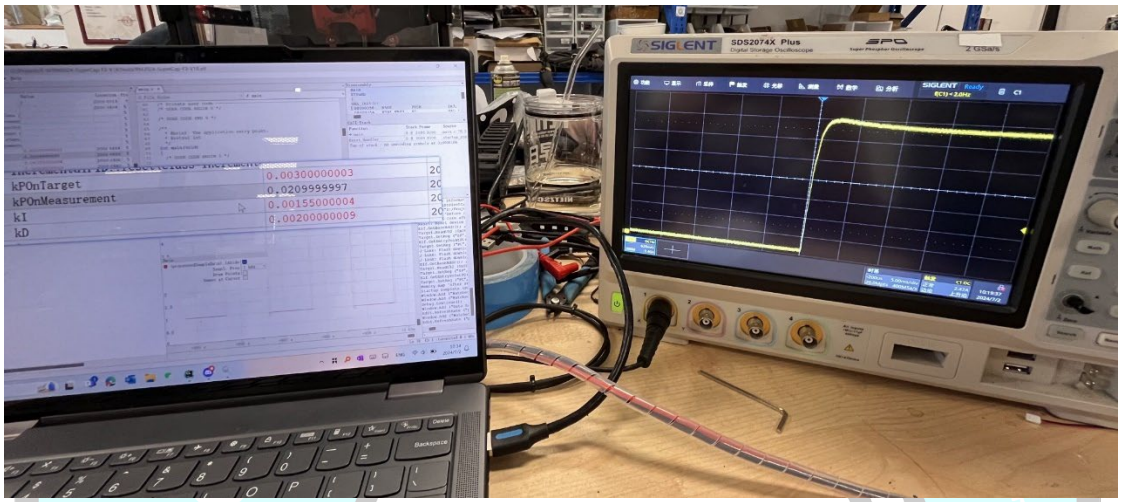
3.8.1 [5V 与 10V]辅助电源





3.8.2 电流控制

图为在某一 PID 参数下表现较好的突变控制电流的波形。



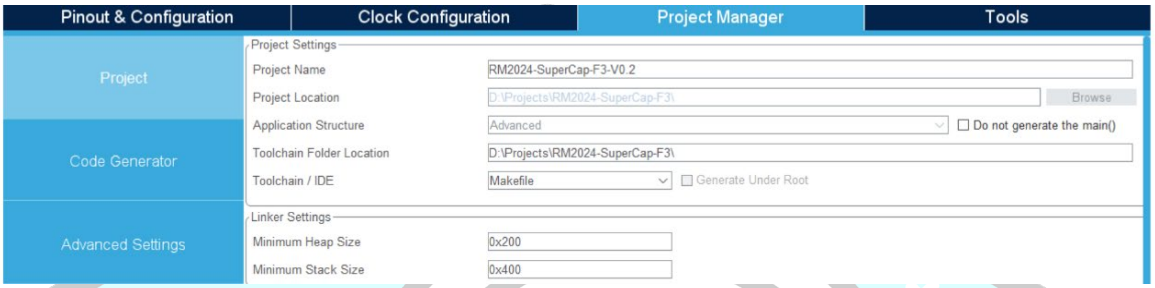
4 软件设计

4.1 开发环境与使用工具

4.1.1 环境

笔者使用的工具链（以及调试器、IDE 等），也基本是在 ENTERPRIZE 队里通用的惯例。

工具链与 IDE 使用了 STM32CubeMX + GCC + VSCode.



STM32CubeMX 能够帮助生成 Makefile 文件、LinkerScript 文件

（STM32F334R8Tx_FLASH.ld）以及初始化文件（startup_stm32f334x8），以及一系列需要的库文件。在生成后即是一个可以编译的项目。

在本项目中为了更好地实现功能，自定义了 Makefile 与 LinkerScript 。

4.1.2 C++兼容

本项目使用了 C++，因此通过使用队内通用的 mk 模板实现编译 C++。

代码入口仍然是 CubeMX 生成的 main.c。通过 extern method 的方式使 C++的入口代码被 main.c 中的主函数调用。

中断函数也使用 extern "C"方法在 C++文件中被调用，如图：

```
/**
 * @brief call back functions
 */
extern "C"
{
    void HRTIM1_Master_IRQHandler(void)
    {...}

    void TIM2_IRQHandler(void) {...}

}  // extern "C"
```

4.1.3 调试环境

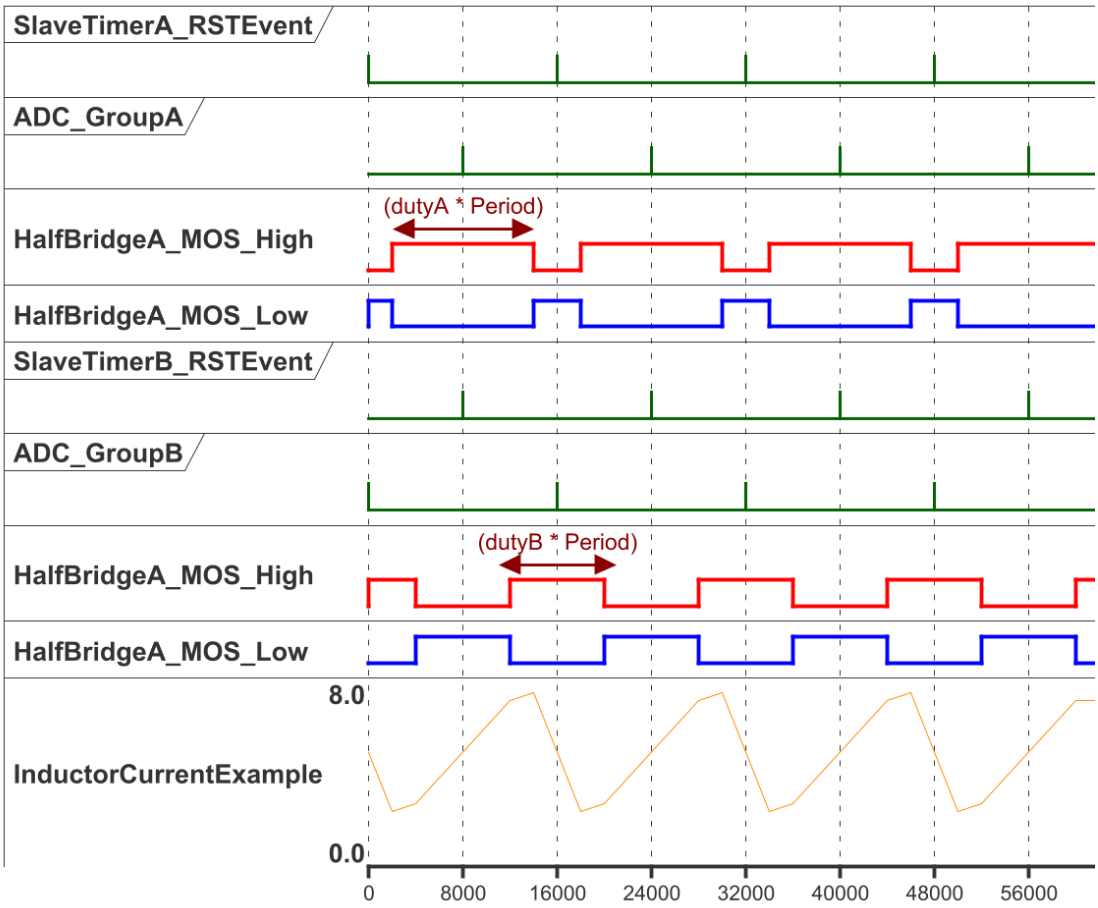
调试使用 Ozone + JLink 的组合，一个窗口内可以实现绘图、监视变量、断点等功能。

4.1.4 整体目录

见 readme.md 文件

4.2 时序

4.2.1 PWM 时序



以 A 半桥举例：令上管在 ACNT=ACMP1 时打开，ACMP3 时关闭。

假设 Period=16000，dutyA=0.75。

则此时 ACMP1 = Period/2 * (1 – dutyA) = 2000，

ACMP3 = Period/2 * (1 + dutyA) = 14000.

A 半桥在(14000-2000)/16000 = 75%的时间中上管打开，25%下管打开，符合 75%占空比。

B 半桥同理。

这样做的目的是使 A 半桥下管打开的时间与 B 半桥下管打开的时间错开，避免 GND 将功率电感短路。如上的操作中，只要确保 (dutyA + dutyB > 1) 即可。

同时，在电路实际工作过程中，MOSFET 的开关会给电压带来纹波进而产生电流纹波。为防止 MOSFET 开关瞬间引入电流采样共模干扰，或产生的电流纹波影响采样，电流采样的时机

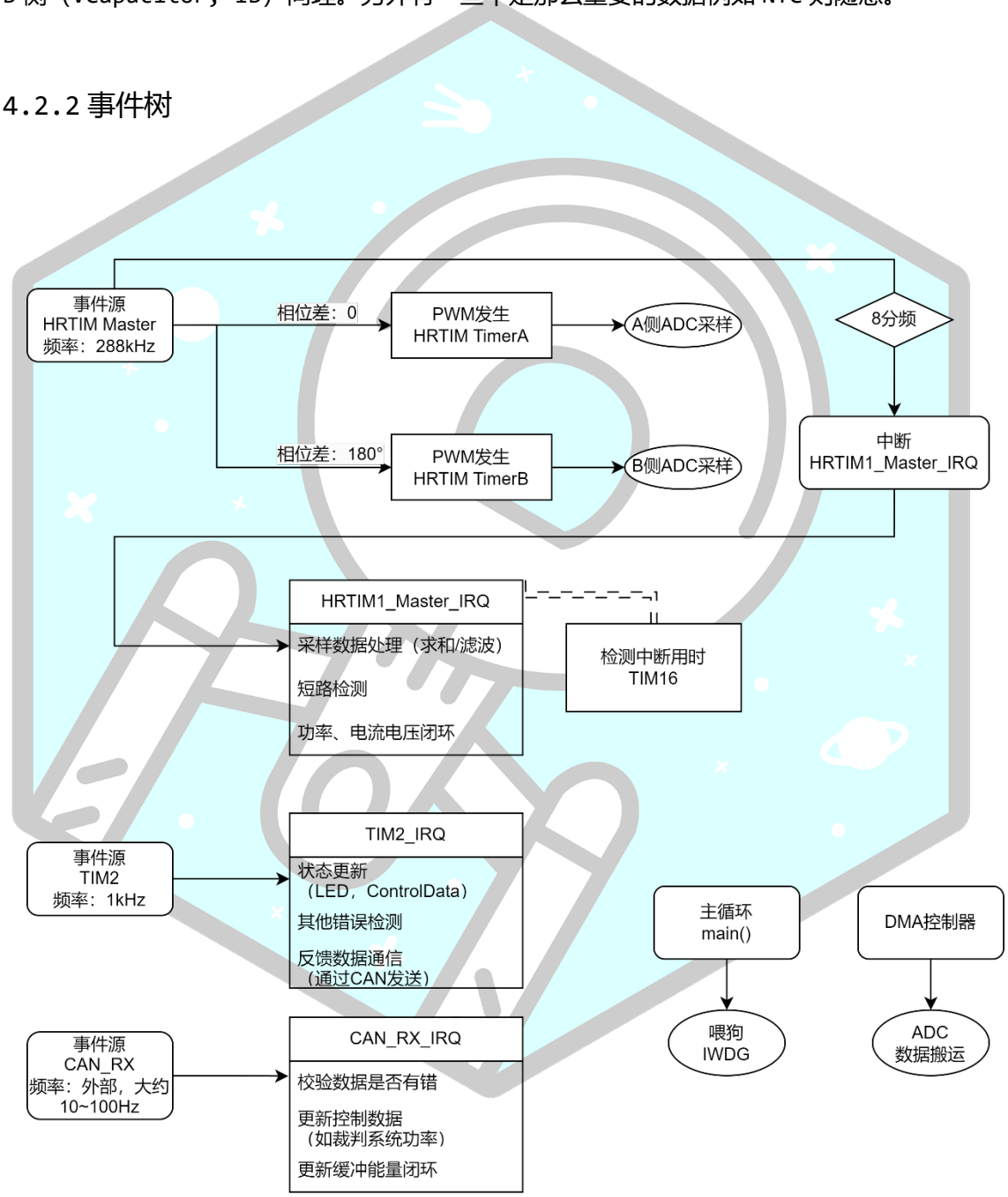
应当正好处于对应侧上管 PWM 波形的中点，此时瞬时电流值等于平均电流值，瞬时电压值等于平均电压值。

因此将 ADC 分为两组：

A 侧半桥的电压、电流（如 VChassis，IReferee，IA）由 TimerA 触发，在 ACNT = ACMP2 ≈ 8000 时采样。这样得到的电压与电流的干扰最小。

B 侧（VCapacitor，IB）同理。另外有一些不是那么重要的数据例如 NTC 则随意。

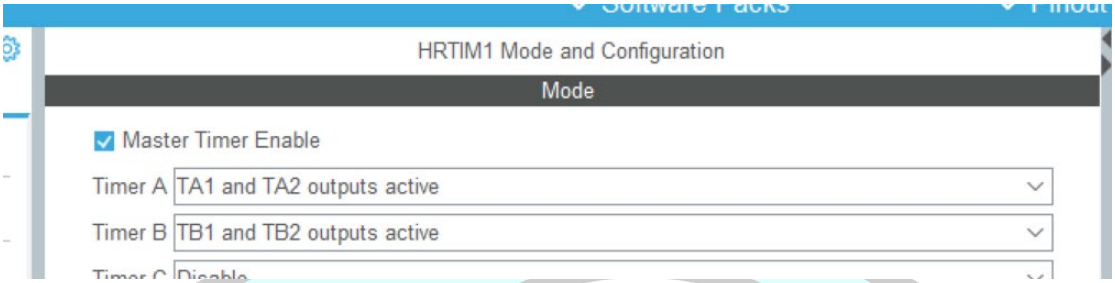
4.2.2 事件树



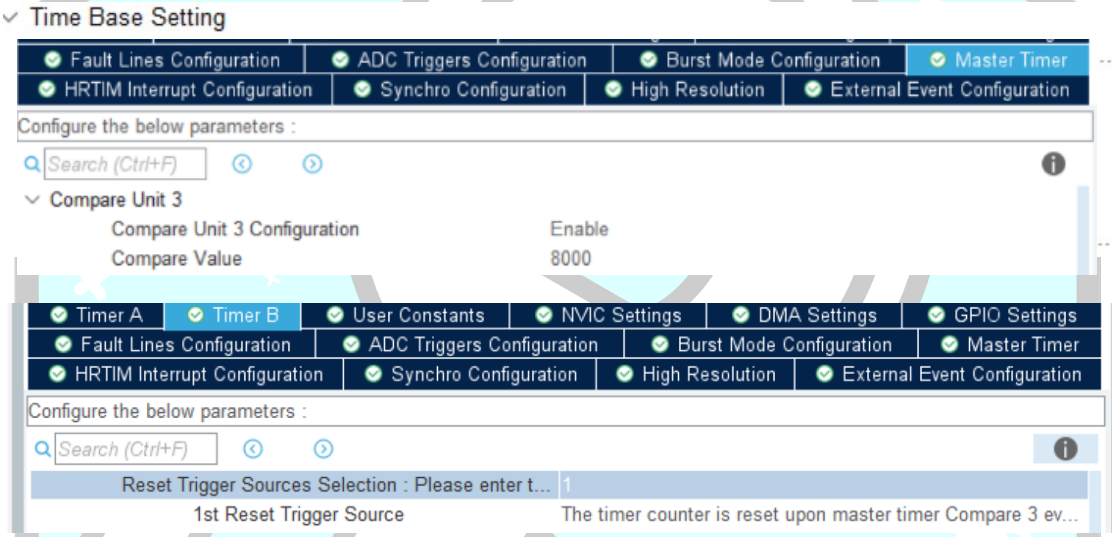
4.3 外设配置

数控 DCDC 非常依赖 MCU 的各种外设来实现时序和高频率的要求。此处简单说明以下配置的一些要点，详细可以参考开源 repo 中 ioc 文件。

4.3.1 HRTIM

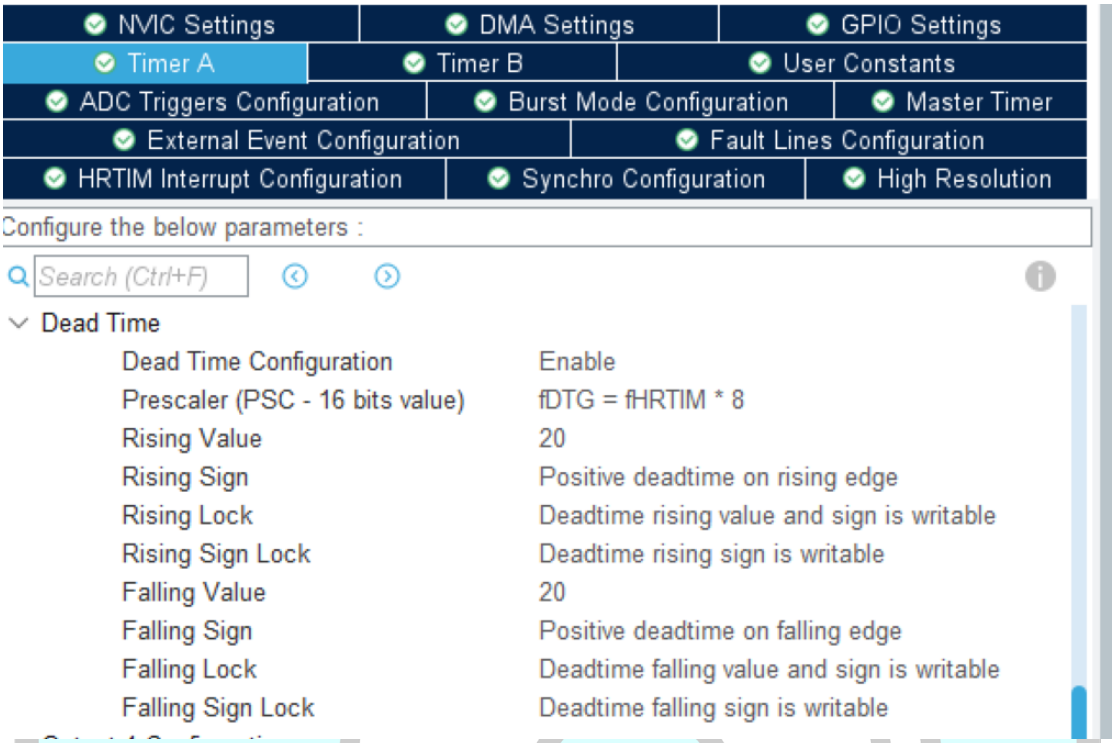


使用 TimerA 与 TimerB 分别控制 A 半桥与 B 半桥的上下管开关。



设定 PWM 频率 288kHz，八分频 34kHz 为*控制中断*，其中运行相应环路计算程序。

将 Timer B 的 Reset Trigger 设为 Master Timer 的一半比较(CMP3=8000)，从而使 Timer A 与 Timer B 有 180°相位差。



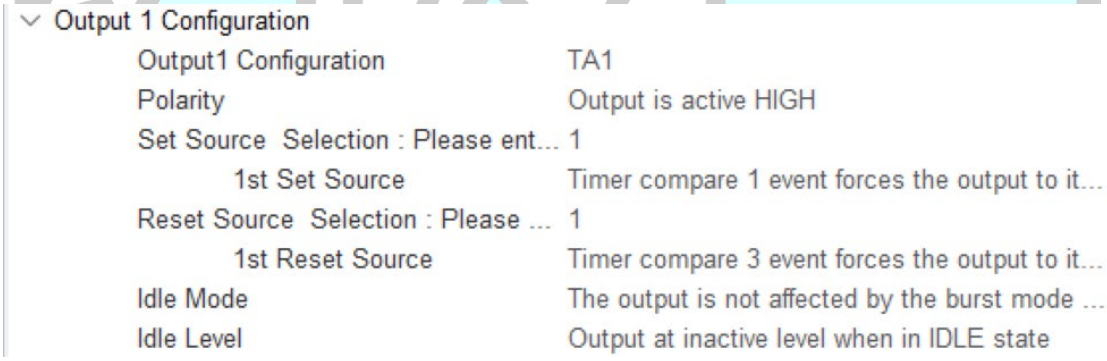
配置输出死区(Dead Time)。根据实际测试情况来调整。

(死区过大->更多依赖寄生二极管的续流，DC-DC 效率降低)

(死区过小->漏电增大，效率降低)

可以通过示波器上 SW 的波形来调整。一般来说寄生二极管续流阶段表现为 SW 电压的一段台阶（因为续流要经过 0.7V 的 MOS 寄生二极管），大约消除台阶即可。

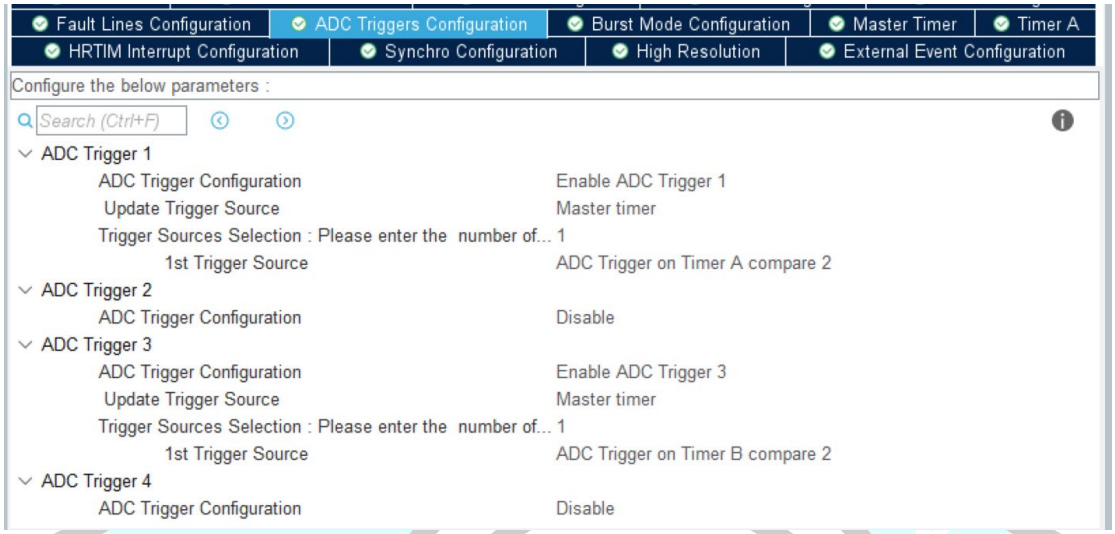
需要注意不同负载电流下死区不一样，因此需要避免死区太小导致某些情况下炸掉。

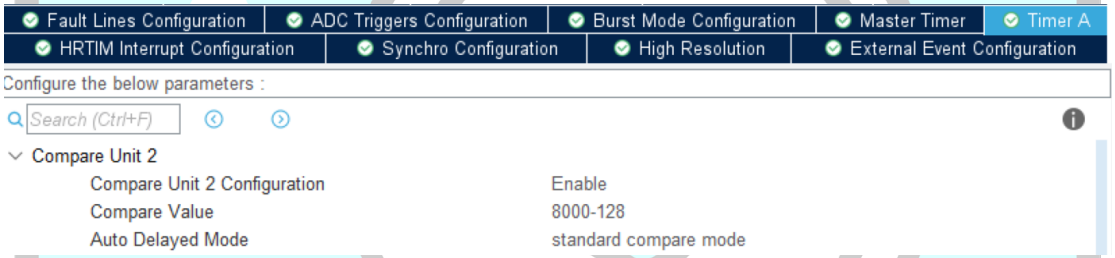


设置输出的置高(Set)与置低(Reset)比较器。

这里用 CMP1 与 CMP3 是因为这两个寄存器有“A specific no-latency update mode”，在更新寄存器的同一周期就可以更新 IO 输出，从而减小输出延迟。参考

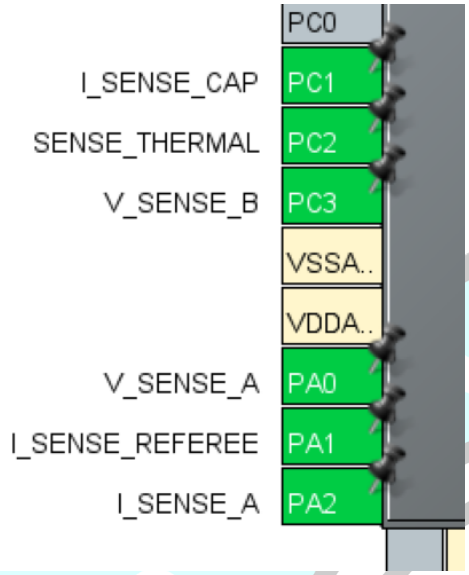
https://www.st.com/resource/en/reference_manual/rm0440-stm32g4-series-advanced-armbased-32bit-mcus-stmicroelectronics.pdf#27.3.12





配置 ADC Trigger，使用子 Timer 的触发，从而实现 ADC 采样时机正好在 PWM 输出的中间段（参考 4.2.1 中的时序图），最大程度降低干扰。

4.3.2 ADC



将同一边的采样分为同一个 ADC 外设中的不同 channel。

（ADC1 负责 VA、IRefere、IA 的采样，ADC2 负责 VB、IB、NTC 的采样。）

ADC_Settings	
Clock Prescaler	ADC Asynchronous clock mode
Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Scan Conversion Mode	Enabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Enabled
End Of Conversion Selection	End of single conversion
Overrun behaviour	Overrun data overwritten
Low Power Auto Wait	Disabled

打开 Scan Mode 与 DMA

ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Number Of Conversion	3
External Trigger Conversion Source	HRTimer Trigger Out 1 event
External Trigger Conversion Edge	Trigger detection on the rising edge
SequencerNbRanks	1
Rank	1
Rank	2
Rank	3

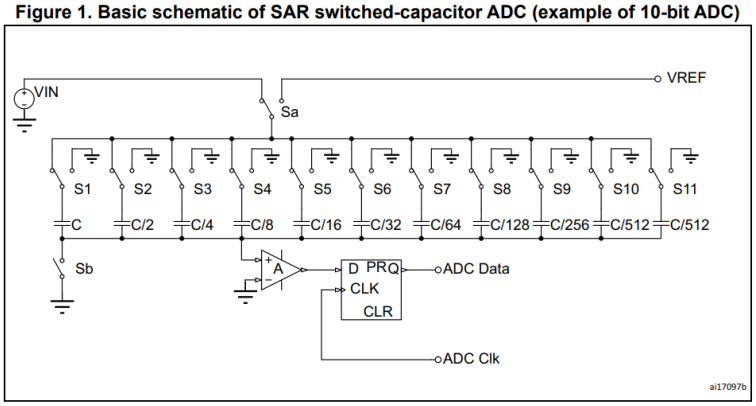
设置相应的 Trigger: HRTimer Trigger Out 1 event（对应 4.3.1 中的事件）

⌄	Rank	1
	Channel	Channel 3
	Sampling Time	7.5 Cycles
	Offset Number	No offset
	Offset	0

Sampling Time 可以酌情配置。这个参数控制了 ADC “Sample and Hold” 使用的时间。若时间太长会增大引入噪声的几率（如 MOS 开关噪声）

若时间太短会导致 ADC 内部 Hold 电容充放电时间太短导致采样误差出现。

Figure 1. Basic schematic of SAR switched-capacitor ADC (example of 10-bit ADC)



1. Basic ADC schematic with digital output.

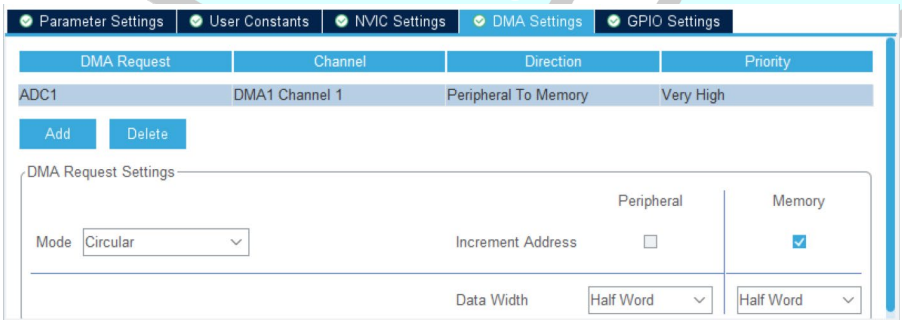
如图是 STM32 内部 SAR-ADC 的示意图，可见采样的瞬间实际上等效于为内部电容充放电。

参考：

https://www.st.com/resource/en/application_note/an2834-how-to-optimize-the-adc-accuracy-in-the-stm32-mcus-stmicroelectronics.pdf#2.1

Rank 的先后顺序可以任意，在硬件上配有 RC 滤波以及运放电压跟随器（见硬件设计片段）之后，基本不会有不同通道互相干扰的情况。

相对的，如果不经过电压跟随器，直接电阻分压就进入引脚采样，则会出现通道间干扰，例如 Rank2 本来采样对象的电压不变，然而数据却会由于 Rank1 读数的变化而变化。



最后为 ADC 设置 DMA，这样可以 a)同步数据传输 b)省下 CPU 时间增加性能

使用 DMA 后在代码中需要调用 HAL_ADC_Start_DMA()。

4.4 模拟数据获取

4.4.1 采样

```
static uint16_t adc1Buffer[ADC1_BUFFER_SIZE] = {};
static uint16_t adc2Buffer[ADC2_BUFFER_SIZE] = {};
```

使用 DMA 将 ADC 中的数据搬运到 buffer 中。

注意在 4.3.1 中 ADC 采样的时机相整点有提前 （CMP=8000-128），目的是为了让 DMA 能够在进入*控制中断*之前将最新的采样数据搬运到内存中。

这样本周期的运算可以直接采用本周期的数据，能减少不必要的延迟。

4.4.2 滤波

由 ADC 与 HRTIM 配置可见，输出 PWM 的每一个周期都会进行 ADC 采样(288kHz)，而控制运算要 8 周期才会进行一次(34kHz)。

因此将这 8 次的数据求和，实现平均滤波器。平均滤波器对随机噪声有较好的抑制，可以减少由于电磁干扰、信号串扰等导致的随机突变噪声。如下图：

```
void ADCSample::sumBuffer() {
    uint16_t tmp = 0;
    for (uint16_t i = 0; i < count; i++) {
        tmp += buffer[offset + i * step];
    }
    sum = tmp;
}
```

再使用一阶滤波将高频信号滤除，防止 PID 控制中 D 项因为高频噪声导致失控。

```
processedData.iASide = CALI_IA(rawADC_IA.sum) * CURRENT_LPF_ALPHA
    + processedData.iASide * (1 - CURRENT_LPF_ALPHA);
```

4.4.3 校准

由于不同硬件的差异（如 MCU 的 ADC 外设误差、采样电阻误差等）原因，ADC 计算得出的数据往往与真实数据之间存在误差，而且基本表现为线性误差。

```
...
#elif (HARDWARE_ID == 207) #elif (HARDWARE_ID == 206)

#define VA_GAIN 0.008127879f
#define VA_BIAS 0.014630182f

#define VB_GAIN 0.008121827f
#define VB_BIAS 0.044670051f

#define I_JUDGE_GAIN 0.020155485f
#define I_JUDGE_BIAS (-41.21019292f)

#define I_A_CONVERT_GAIN 0.020126509f
#define I_A_CONVERT_BIAS (-41.18257619f)

#define I_B_CONVERT_GAIN (-0.020207852f)
#define I_B_CONVERT_BIAS 41.29849885f
```

则每一个数据有对应的 bias 与 gain，如下：

校准函数（宏）举例：

使用 Excel 辅助校准的过程：

	A	B	C	D	E	F	G	H
1		real data		adc value			k	b
2	VASide	12	24	1474.6	2951		0.008128	0.01463
3	VBSide	12	28.8	1472	3540.5		0.008122	0.04467
4								
5	IAConvert	1	8	1996.5	1648.7		-0.02013	41.18258
6	Ijudge	1	8	1995	1647.7		-0.02016	41.21019
7	IBConvert	1	8	1994.2	1647.8		-0.02021	41.2985

校准数据储存以宏的形式储存在 Calibration.hpp 中，也通过宏来开启。

同时，校准后 HARDWARE_ID 参数会被写入 OptionBytes 当中，防止未来失误烧错代码。

每次上电时读取 OB 中的数据，若不一致则蜂鸣器报错。

```
#ifdef CALIBRATION_MODE
        while (HAL_FLASH_OB_Unlock() != HAL_OK)  // unlock flash option bytes
            __ASM volatile("bkpt ");

        while (HAL_FLASHEx_OBErase() != HAL_OK)  // erase option bytes
            __ASM volatile("bkpt ");

        FLASH_OBProgramInitTypeDef pOBInit;
        pOBInit.OptionType  = OPTIONBYTE_WRP | OPTIONBYTE_RDP | OPTIONBYTE_USER | OPTIONBYTE_DATA;
        pOBInit.WRPState    = OB_WRPSTATE_DISABLE;
        pOBInit.WRPPage     = 0xFFFFFFFF;
        pOBInit.RDPLevel    = OB_RDP_LEVEL_0;
        pOBInit.USERConfig  = OB_IWDG_SW | OB_STOP_NO_RST | OB_STDBY_NO_RST | OB_BOOT1_RESET | OB_VDDA_ANALOG_ON | OB_SRAM_PARITY_RESET;
        pOBInit.DATAAddress = OB_DATA_ADDRESS_DATA0;
        pOBInit.DATAData    = (uint8_t)HARDWARE_ID;       // set hardware id
```

经过校准之后，不同控制板的电流电压读数几乎一致。

编译时需要指定当前硬件 ID，详见 5.2 章节。

4.5 升降压与功率控制

4.5.1 控制电路升降压状态的关键变量

为了简化控制逻辑，将外部升降压电路使用一个参数控制：outputDuty，称为广义占空比。

定义：物理意义是理想情况下 $\text{outputDuty} = V_B / V_A$ （电压之比）

实现方式：由上文 Buck-Boost 电路的推导可以得出，若分别给左侧半桥（A 半桥，接在底盘/电池侧）与右侧半桥（B 半桥，接在电容组一侧）的占空比分别为 duty_A 与 duty_B，

则稳定平衡状态下，左右的电压关系为：

$\frac{V_B}{V_A} = \frac{\text{duty}_A}{\text{duty}_B}$.

因此通过生成一对 duty_A与 duty_B ，满足 $\text{duty}_A/\text{duty}_B = V_B/V_A = \text{outputDuty}$，并将其分别应用于 A 半桥与 B 半桥的输出 PWM 占空比中，即可实现用一个变量去控制升降压。

这一步的主要目的是为了将 PID 控制与占空比计算的逻辑分开，方便代码复用、逻辑分离。

需要注意这个电压关系只在变换器满足：

a）平均电流为零的平衡状态

b）MOS 瞬间开关、无内阻的理想情况

时成立。

因此开环控制 outputDuty 并不能精确控制电路的输出电流与电压，仍需要闭环控制。

4.5.2 Buck，Boost 与 Buck-Boost

由于硬件上支持 100%占空比的半桥上管开启（见 3.3），因此本模块可以运行在纯 Buck 或者纯 Boost 模式的 DCDC 中。

然而虽然上管可以完全开启，但是却不能做到接近 100%占空比（由于 MOS 开关需要时间，上管的每周期关闭时间要么为 0，要么为不太小的一段时间，大约>5%）。因此需要区分这三种变换器状态。

为了实现转换，在 $\text{outputDuty} < 0.8$ 时为 Buck 模式（A 半桥常高电平，B 半桥以outputDuty为占空比）；在 $\text{outputDuty} > 1.25$ 时为 Boost 模式（B 半桥常高电平，A 半桥以 $(1/\text{outputDuty})$ 为占空比。

当outputDuty在两者中间时，两个半桥分别以

$\text{duty}_A = (\text{outputDuty} + 1) * 0.4$, $\text{duty}_B = (1 / \text{outputDuty} + 1) * 0.4$ 为占空比，满足上文推导。

同时，由于状态切换不可避免带来跳变，使用迟滞来防止震荡。可参考 PowerManager.cpp 中的 updatePWM()函数。

4.5.3 电流，电压与功率闭环

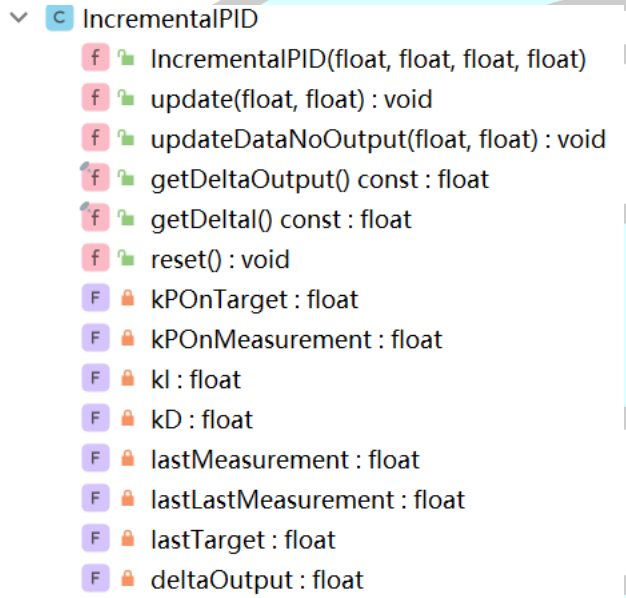
4.5.3.1整体策略

采用多环串联 PID：最外环为裁判系统缓冲能量闭环，而后功率闭环，最后同时闭环电压与电流。此处只介绍功率、电压电流闭环。

4.5.3.2PID 数据结构

采用增量式 PID，计算的输出是本次与上次输出的差值。

可见于 IncrementalPID.cpp 与 IncrementalPID.hpp



优点有：

a）单次运算量较小
b）方便逻辑部分管理累计值，便于实现积分上限、非线性化输出等功能
c）实现了部分基于 P on Measurement 的闭环，抑制过冲

为了性能考虑，每次运算可以选择单纯更新数据 updateDataNoOutput()，或者计算出结果 update()。

4.5.3.3外环功率环

```
float absIA = M_MIN(M_ABS(ProcessedSampleData::processedSampleData.iASide), 0.1f);
float absIB = M_MIN(M_ABS(ProcessedSampleData::processedSampleData.iBSide), 0.1f);
float actualIAtoIB = absIA / absIB;

pidPowerReferee.update(tempData.targetRefereePower, ProcessedSampleData::processedSampleData.pReferee);

tempData.targetAPower += pidPowerReferee.getDeltaOutput();
```

通过裁判系统侧 Referee_Power 采样得到的功率去修改目标 A_Power 值

4.5.3.4经过限制的电流环

这一步先输出 targetIA，然后输出占空比。

在这一阶段我们首先通过计算得到当前情况下 IA 的上限与下限。依据是底盘侧承受的平均电流与电容组能承受的平均电流。(电感电流的冗余已经通过开关频率计算得出，化为了两侧平均电流，因此不用多考虑)

计算方式：

底盘侧承受的平均电流直接转化为 targetIA 的 Max 与 Min；

电容组承受的平均电流则通过当前实际观测到的 IA 与 IB 之比来转化为 targetIA 的上下限。

经过这一步之后，我们就将上一步计算的 targetAPower 除以电压，再 Clamp 就得到 targetIA。

而后使用增量式 PID 控制器来输出 outputDuty 的差值，加上原来的值之后得到 iADuty.

4.5.3.5电流与电压双环竞争

大多数时候都只需要上文的闭环即可稳定输入功率与底盘侧电压。(因为底盘侧连接着电池，之间的线阻比较小，因此只要控制不让电流失控地流入、流出电池即可做到底盘电压自动稳定)

然而当电容组电压较高、即将充满电时，则需要另一个电压环来与上文提到的电流环竞争。否则，可能会出现电容超过耐压而击穿爆炸的情况。

我们同样通过 PID 计算得出需要的占空比 vBDuty.不同的是，我们主要控制 B 侧电压，则通过上文广义占空比 outputDuty 的定义可以得到，增量 PID 控制器输出的增量值应该除以 A 侧电压才能做到对 B 侧的线性化。具体做法可以参考代码。

为了性能考虑，只有当 B 侧电压到达满电的 90%时才开启电压环竞争。

得到了 iADuty 与 vBDuty 之后，我们选择占空比较小的一个值作为输出，即优先确保 B 侧电压不过高。

同时，当检测到电压环胜出的时候，此时功率闭环不会被满足，因此我们需要对功率环的积分消除，防止积分控制器过多地累加导致失控。

4.5.3.6前馈控制/缓启动

这个是针对零电流启动的“前馈”，即：

当整个控制环路新启动的时候，选择当前观测到的[B 侧电压除以 A 侧电压]即 VB/VA 作为 outputDuty 的值。从上文推导可以发现，观测值计算得到的这个初始 outputDuty 即满足零电流下的伏秒平衡，所以启动瞬间不会出现很大的电流。

实际测试发现上电瞬间波形是很平缓的，有效解决第一次开关的冲击问题。

4.6 错误处理

本项目的另一个特点在于非常完善的错误处理机制。对于数控的升降压变换器，控制上失误、调试太激进等都会导致严重的后果，令硬件损坏。因此该模块为了保障安全性，设计了数种故障模式，排查了几乎所有会导致灾难性后果的情况。

在 1kHz 的中断中检测大部分错误，在 34kHz 的中断中检测短路/过流保护。

4.6.1 低压保护

低压保护首先是为了防止模块供电电压过低导致 MOS 无法正常开关。因此在 A 侧与 B 侧电压最小值小于 12V(因为总供电是从两侧分别经过二极管后并联得到的)时，进入低压保护，防止 MOS 驱动工作不正常导致烧毁。此时进入 ERROR_UNDER_VOLTAGE 状态。

同时，规则要求底盘被裁判系统断电之后超级电容也不能输出，因此判断 A 侧电压是否小于 15V，若是则认为底盘被裁判系统断电了，此时进入 ERROR_NO_POWER_INPUT 状态。

4.6.2 过压保护

为了防止过压击穿 MOS 与供电、底盘电机等。

设计检测到电压越高，计数累加越大，直到回到正常电压或者触发过压保护后清除计数。

这样子做的目的是为了避免让动能回收导致的短暂母线电压升高触发保护。

可以参考 ErrorChecker.cpp 中的 checkOverVoltage()函数.

4.6.3 短路保护

这个保护的触发逻辑运行在 34kHz 高速中断中，检测到电流绝对值大于一定值，或者电流大而电压小(例如 I>15， V<4)则认为模块与相连电路存在短路，计数 5 次之后（持续短路约 150μs）即进入保护状态。

参考 updateShortCircuit()函数。

4.6.4 转换器错误保护

运行中的模块可能出现：虽然电压、电流等参数正常，但是 DCDC 转换器并不工作在正常情况下。

举例：若由于驱动 IC 损坏，某一半桥的下管无法打开，则这一侧只能降压无法升压。

这些情况对于稳定控制是致命的，因此使用 a)运行效率，即功率之比 b)实际升降压产生的的两侧电压之比，是否与理想情况下通过 outputDuty 参数计算得出的相符 来检测这类错误。

触发会进入 ERROR_BUCK_BOOST 保护，不再自动恢复。

4.6.5 电容组监测

当模块检测到超级电容故障，模块进入锁定状态，关断输出，且不会自动尝试重新启动，需要重置。

检测超级电容故障的逻辑如下：

当超级电容电压变化达到一定阈值时，若充入超级电容的电荷量过低，认为与超级电容之间的连线断开。 （有效电容值过低）

当充入超级电容的电荷量达到一定阈值时，若超级电容的电压变化过小，认为超级电容局部短路，过充，或漏电。

为防止电荷量累计漂移误差，若未处于故障状态，每隔一段时间会重置故障检测。

4.6.6 过温保护

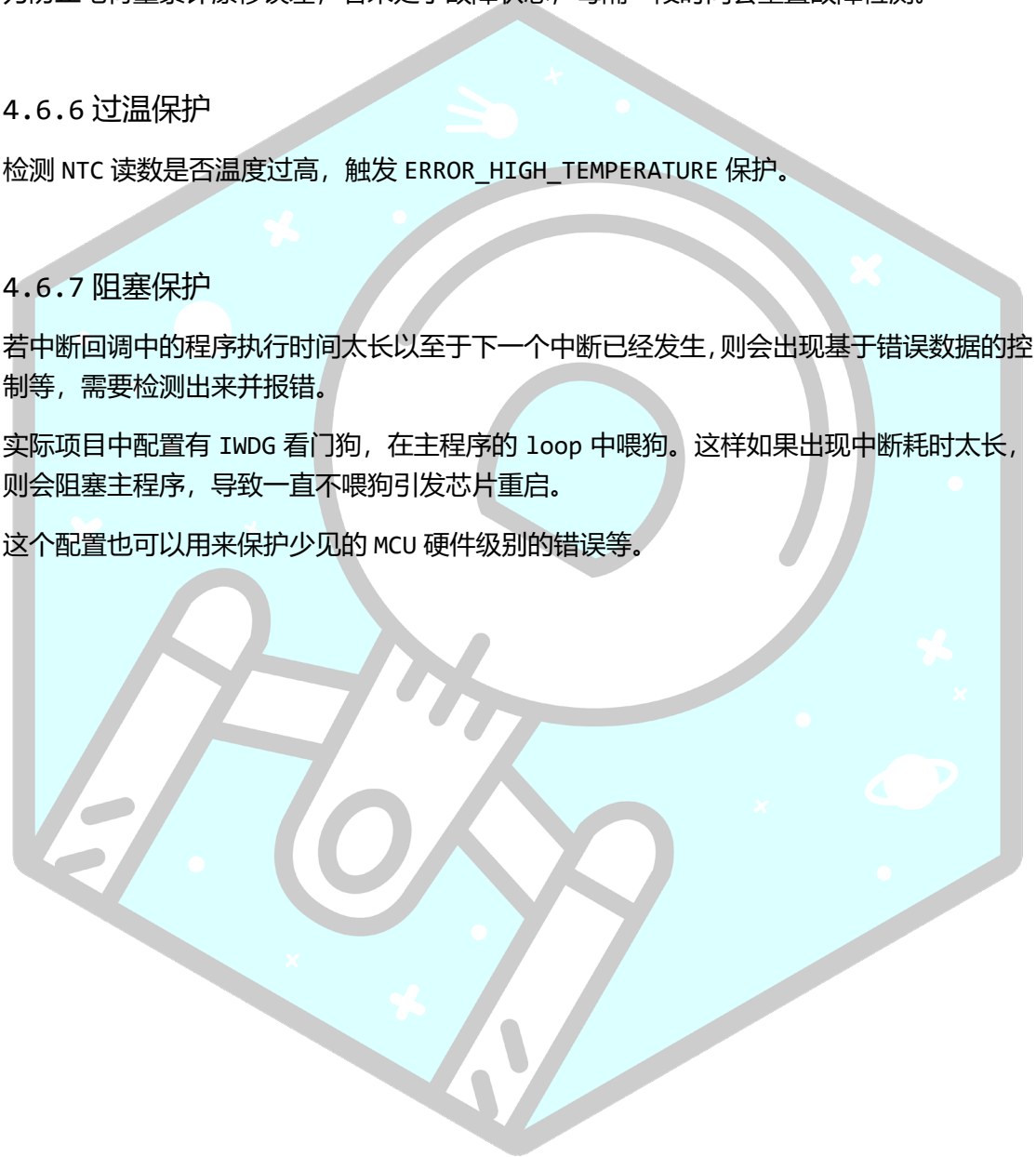
检测 NTC 读数是否温度过高，触发 ERROR_HIGH_TEMPERATURE 保护。

4.6.7 阻塞保护

若中断回调中的程序执行时间太长以至于下一个中断已经发生，则会出现基于错误数据的控制等，需要检测出来并报错。

实际项目中配置有 IWDG 看门狗，在主程序的 loop 中喂狗。这样如果出现中断耗时太长，则会阻塞主程序，导致一直不喂狗引发芯片重启。

这个配置也可以用来保护少见的 MCU 硬件级别的错误等。



4.7 通信

4.7.1 通信内容说明

在整车设计中，超级电容控制模块作为一个独立的模块，只会与主控板（底盘上搭载的开发板，可见于 Enterprize 在 RM2024 的 G4 主控板开源）交互。因此主要实现了基于 CAN 网络的通信。

硬件层面上，可以直接将 CAN 总线接入底盘电机所在的 CAN 总线，也可以单独接到另一个 CAN 网络。

电容控制模块以 1ms 为周期向主控板反馈电容剩余能量、底盘瞬时功率、瞬间最大输出功率等信息，辅助底盘控制板上的功率控制代码进行底盘电机功率控制。

大约每 10 到 100 毫秒接收一次来自底盘控制板的回报信息，包括来自裁判系统的剩余能量信息，来自裁判系统的功率限制信息，电容模式控制信息等。利用此信息对功率控制的误差进行修正（能量闭环），也能正确接收来自裁判系统的功率信息。

4.7.2 数据结构

```
struct CapacitorTx
{
    uint8_t enableDCDC : 1;
    uint8_t systemRestart : 1;
    uint8_t resv0 : 6;
    uint16_t feedbackJudgePowerLimit;
    uint16_t feedbackJudgePowerBuffer;
    uint8_t resv1[3];
} __attribute__((packed)) capacitorTx;
```

主控板->电容控制板

```
struct CapacitorRx
{
    uint8_t errorCode;
    float chassisPower;
    uint16_t chassisPowerLimit;
    uint8_t capEnergy;
} __attribute__((packed)) capacitorRx;
```

电容控制板->主控板

4.7.3 回调函数

当接收到从主控板发来的消息时，在回调函数中首先会搬运数据，将 RX 结构体中的控制数据搬运到 ControlData 数据体中，且都是原子操作避免出现脏数据。

同时，回调函数中还会执行缓冲能量 PID 闭环。可见于 updateEnergy()函数。

4.7.4 超时

若一段时间没收到来自主控板的消息，则认为当前的数据已经不可靠，会自动回到默认的功率参数（37 瓦）且停止缓冲能量闭环。

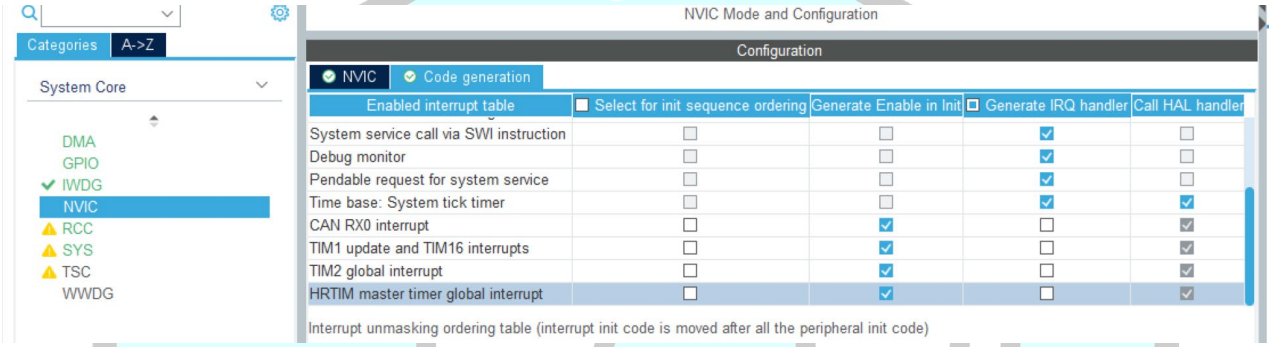
4.8 性能优化

4.8.1 DMA 搬运 ADC 数据

在 4.4.1 中说明，此处不重复写。

4.8.2 中断回调

对于一些回调操作，例如来自定时器的中断、CAN 中断等，从 STM32CubeMX 生成的代码框架中一般是先由用户注册函数，再由 Manager 去调用。这样虽然减少耦合方便复用，但是代码执行的效率会有损失。因此本项目中关键的中断回调都越过 Hal 库而直接引用用户的函数。



如此配置后，将用户的函数做出相应命名即可被中断调用。

举例：HRTIM 的中断函数，位于 PowerManager.cpp 中的 HRTIM1_Master_IRQHandler()

```
/**
 * @brief call back functions
 */
extern "C"
{
    void HRTIM1_Master_IRQHandler(void){...}

    void TIM2_IRQHandler(void){...}

}  // extern "C"

}  // namespace PowerManager
```

注意用户需要在回调函数中清除中断位，例如执行：

```
__HAL_HRTIM_MASTER_CLEAR_IT(&hhrtim1, HRTIM_MASTER_IT_MREP);
```

否则程序会卡住。

4.8.3 使用宏替代 Hal 库函数

普遍的来说，Hal 库函数会有安全检测等功能，效率不如调用相应的宏，或者直接写寄存器。

在一些频繁调用的程序中做出替代可以提升一定的性能。

举例：更新 PWM 占空比的函数会被每周期都调用，因此修改寄存器来实现高效率。

```
pwmDutyVariable.BCMP3 = (HRTIM_PERIOD / 2) * (1.0f + pwmDutyVariable.dutyB);

HRTIM1->sTimerxRegs[HRTIM_TIMERINDEX_TIMER_A].CMP1xR = pwmDutyVariable.ACMP1;
```

4.8.4 [G474]SRAM 与 CCMRAM 的使用

这段是针对 STM32G474 制作的电容控制代码。这一代今年有开发但是最终由于测试时间不足没有选择上场，但是其中的一些做法笔者认为值得分享给大家。

首先，一般 STM32 会默认将变量数据(data)放在 RAM 中，函数放在 FLASH 中。

把重要函数在开机时导入 RAM 可以提高其运行速度。因此进行如下设置：

```
138     /* Initialized data sections goes into RAM, load LMA copy after code */
139     .data :
140     {
141       . = ALIGN(4);
142       _sdata = .;        /* create a global symbol at data start */
143       *(.data)           /* .data sections */
144       *(.data*)          /* .data* sections */
145       *(.code_in_ram)
146
147       . = ALIGN(4);
148       _edata = .;        /* define a global symbol at data end */
149     } >CCMRAM AT> FLASH
150
```

加入了一行 `*(.code_in_ram)` 。这样之后，只要在重要函数的签名处加上

`__attribute__ ((section (".code_in_ram")))`

即可在开机的时候自动把函数放进内存中实现更快调用。

举例：

```
22  __attribute__ ((section (".code_in_ram"))) void IncrementalPID::update(float target, float measurement)
```

不过若函数已经被编译器展开则这个优化的用处不大。但是这个仍然是可行的优化之一。

再说说使用 CCMRAM 替代 SRAM 来加速。

STM32 一般含有一片 CCMRAM 区域，其中的数据相比普通的内存空间，被 CPU 读写的速度更快。

在 STM32F3 系列中，CCMRAM 很小，很难起到显著作用。而在 STM32G4 系列中，不仅 CCMRAM 很大（有 32KB），且新增了一系列功能。

Table 2. **CCM SRAM main features**

Feature/ products	STM32F303xB/C STM32F358xx	STM32F303x6/8 STM32F334xx STM32F328xx	STM32F303xD/E STM32F398xx	STM32G47xx STM32G84xx	STM32G431x STM32G441x
Size (Kbytes)	8	4	16	32	10
Mapping	0x1000 0000			0x1000 0000 and can be aliased at 0x2001 8000	0x1000 0000 and can be aliased at 0x2000 5800
Parity check	Yes				
Write protection	Yes, with 1-Kbyte page granularity				
Read protection	No			Yes	
Erase	No			Yes	
DMA access	No			• No if mapped at 0x1000 0000 • Yes if mapped at 0x2001 8000	• No if mapped at 0x1000 0000 • Yes if mapped at 0x2000 5800

G4 系列中，CCMRAM 可以通过内存总线访问，位于 0x20018000 之后的 32KB 空间。

因此我们通过修改 LinkerScript 可以做到将所有本来放置在 SRAM 中的数据改为放置在 CCMRAM 中。基于 CubeMX 生成的 STM32G474RBTx_FLASH.ld 修改：

```
/* Highest address of the user mode stack */
_estack = ORIGIN(CCMRAM) + LENGTH(CCMRAM);
/* Generate a link error if heap and stack do
_Min_Heap_Size = 0x200;      /* required amou
_Min_Stack_Size = 0x400; /* required amount o

/* Specify the memory areas */
MEMORY
{
  SRAM1    (xrw)    : ORIGIN = 0x20000000,
  SRAM2    (xrw)    : ORIGIN = 0x20014000,
  CCMRAM   (xrw)    : ORIGIN = 0x10000000,
  FLASH    (rx)      : ORIGIN = 0x8000000, LE
}
```

```
/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM);    /* end
/* Generate a link error if heap and stack don'
_Min_Heap_Size = 0x200;      /* required amount
_Min_Stack_Size = 0x400; /* required amount of

/* Specify the memory areas */
MEMORY
{
RAM (xrw)      : ORIGIN = 0x20000000, LENGTH =
FLASH (rx)      : ORIGIN = 0x8000000, LENGTH =
}

/* Define output sections */
```

这步修改告诉 CPU 将 0x10000000 之后的 CCMRAM 记作 MEMORY 的一部分从而可以调用。

```
    . = ALIGN(4);
    _edata = .;        /* define a global sym
  } >CCMRAM AT> FLASH


  /* Uninitialized data section */
  . = ALIGN(4);
  .bss :
  {
    /* This is used by the startup in order t
    _sbss = .;         /* define a global sym
    __bss_start__ = _sbss;
    *(.bss)
    *(.bss*)
    *(COMMON)

    . = ALIGN(4);
    _ebss = .;         /* define a global sym
    __bss_end__ = _ebss;
  } >CCMRAM
```

```
    . = ALIGN(4);
    _edata = .;        /* define a global symbo
  } >RAM AT> FLASH


  /* Uninitialized data section */
  . = ALIGN(4);
  .bss :
  {
    /* This is used by the startup in order to
    _sbss = .;         /* define a global symbo
    __bss_start__ = _sbss;
    *(.bss)
    *(.bss*)
    *(COMMON)

    . = ALIGN(4);
    _ebss = .;         /* define a global symbo
    __bss_end__ = _ebss;
  } >RAM
```

然后如此修改即可把原本放到 RAM 区域中的数据放到 CCMRAM 中。

然而这样的操作带来一个问题，即是 DMA 不能通过 0x10000000 的地址访问 CCMRAM。若直接配置如下，则会发现 Buffer 当中的数据无法被更新。

```
static uint16_t adc1Buffer[ADC1_BUFFER_SIZE] = {};
static uint16_t adc2Buffer[ADC2_BUFFER_SIZE] = {};

void ADCDMAStart()
{
    /* start ADC */
    HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc1Buffer, Length: ADC1_BUFFER_SIZE);
    HAL_ADC_Start_DMA(&hadc2, (uint32_t*)adc2Buffer, Length: ADC2_BUFFER_SIZE);
}
```

如果在 Ozone 里面观察会发现此时 adc1Buffer 等变量的地址位于 0x1000xxxx，说明确实被创建在了 CCMRAM 当中，且 DMA 无法访问。

应对这个问题，可以采用一种比较巧的做法，利用 STM32G4 的 CCMRAM 的 Bias 机制：

通过阅读手册发现，DMA 是可以访问位于 0x20018000 的 CCMRAM 内存的。但是在 LinkerScript 的配置之下已经把所有变量的地址都放在了 0x10000000，要想得到能被 DMA 访问的地址只能手动修改：

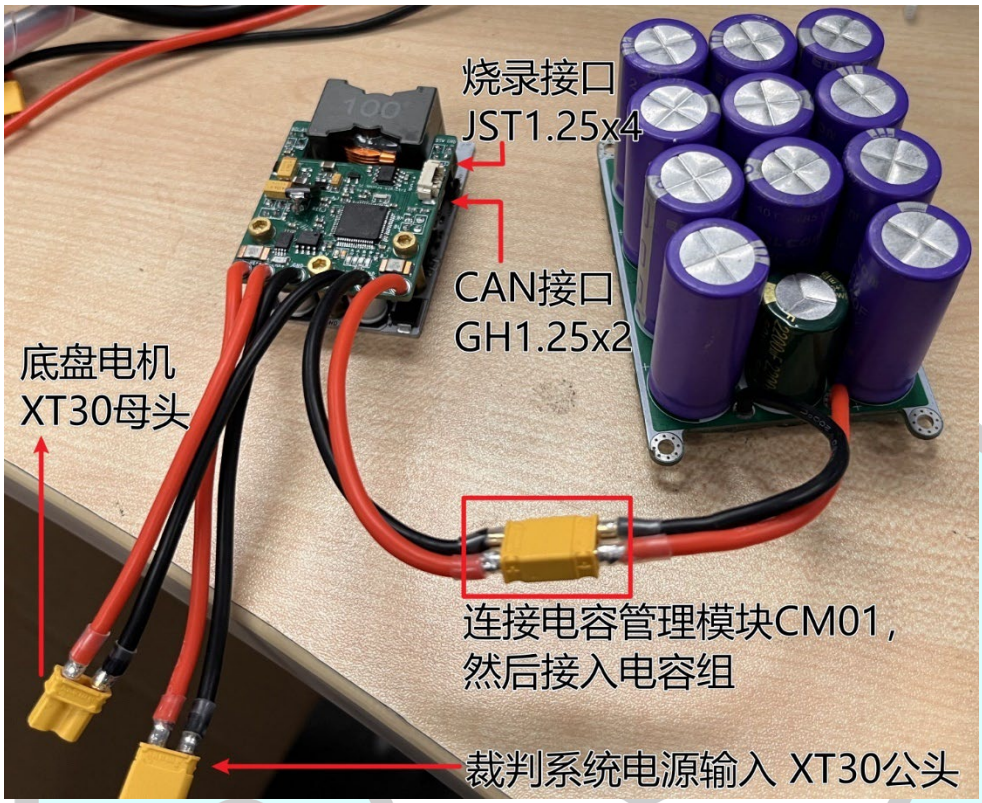
```
#ifdef USE_CCMRAM_ALIAS
#define CCMRAM_ALIAS_OFFSET (0x20018000 - 0x10000000)
#define CCMRAM_ALIAS_ADDRESS(pBuffer) (reinterpret_cast<uint32_t *>(reinterpret_cast<uint32_t>(pBuffer) + CCMRAM_ALIAS_OFFSET))
#else #ifdef USE_CCMRAM_ALIAS
    #define CCMRAM_ALIAS_ADDRESS(pBuffer) (pBuffer)
#endif #ifdef USE_CCMRAM_ALIAS #else

static uint16_t adc1Buffer[ADC1_BUFFER_SIZE] = {};
static uint16_t adc2Buffer[ADC2_BUFFER_SIZE] = {};

volatile static uint32_t *aliasADC1Buffer = CCMRAM_ALIAS_ADDRESS(adc1Buffer);
volatile static uint32_t *aliasADC2Buffer = CCMRAM_ALIAS_ADDRESS(adc2Buffer);
```

经过试验，这种方式是可行的，Buffer 中数据被 DMA 成功更新了。而且相比之前的，中断里面的总代码时间得到了优化大约 3%。

（可能会有人好奇如果直接把所有的数据都放到 0x20018000 地址会怎么样。功能都能正常运行，但是代码运行时间不降反增。推测原理是 CCMRAM 允许从内存总线访问，即是 0x20018000 的地址；然而如果 CPU 从这个地址访问则相当于绕了个大圈子，反而比普通内存更慢。）

5 使用方法

5.1 硬件接口&连接



5.2 软件环境&编译&烧录

本项目基于 GCC ARM GNU 工具链。

1. 安装 GCC 环境。以 Windows 系统下为例，下载 MSYS2 并安装，在系统环境变量中加入相关目录。
2. 安装 arm-none-eabi-gcc

然后即可执行编译操作。需要注意编译时需要指定 HARDWARE_ID，否则将会报错。

```
make -j HARDWARE_ID=xxx
```

得到 build/目录下 RM2024-SuperCap-F3-V1R.elf 文件即为编译成功。

将 elf 文件在 ozone 新建项目中打开，即可通过 JLink 烧录至 MCU 中。

6 未来优化方向

在 RM2024 赛季中，本开源所介绍的电容控制器替代掉了绝大多数旧版本电容控制器(2023 开源版本)， 并在调试和实战过程中表现稳定， 但是同时也在大量的测试与经验积累下发现一些问题，并作为未来改进方向：

6.1 硬件方面

出线方式： 本方案采用先用 AWG16 线材延长再焊接 XT30 接口的方案， 便于在狭小空间中走线； 但在维护过程中难免反复弯折线材， 使得焊板处的线格外脆弱， 并且线材更换起来可能会有不便。参考其他学校开源方案后计划研究 XT30 接口直接焊板的方案

走线与 Layout： 由于本项目时间规划的问题,以及作者本人的水平有限、PCB 限制比较多，因此 Layout 并不完全如人意，在很多地方的布局以及连接都有优化的空间。

元器件选用： 本代项目基本沿用了上一代的选型。有一些元器件例如功率电感直接采用了 PQ2918 扁线电感，然而经过别的队伍的启发发现可能存在更多的选择，例如线艺的一体成型电感、EPC 的羟基粉电感等具有优异的性能。同时，有一些元器件并没有完全发挥出全部能力，例如尝试不同的 MOS 与驱动的组合、寻找效率最高的开关等。这些都需要在来年继续研究。

外壳改进： 由于外壳设计考虑不足， 没有预留足够的安装孔位， 在安装时造成了一些不便。之后的设计可以结合机械设计，为不同的机器人量身定做外壳，使机械设计能够更加灵活小巧。

6.2 软件方面

本开源项目仍然使用了去年的 F334 方案。不过随着 G474 的价格下降，更高性能或许能够带来更高的闭环频率、响应速度，以至于有更好的冲击表现（例如动能回收）。这方面的探索正在进行中。实际上，笔者已经验证了 G474 制作的模块可以以 2 周期/中断来执行当前 8 周期/中断的程序，需要继续探究对性能的改善。

在信号调理方面，实际上本项目的一些处理（例如滤波）或许显得冗余。需要多组合尝试来选择最佳的信号处理程度。

在控制方面，本项目依靠的理论依据其实并不多，大多是以拟合的方式结合通用的 PID 算法来控制。这样的优点是开发较快速，缺点则是并不能发挥最佳的控制效果。随着之后更深入的研究，或许使用某些模型可以大幅度提升性能。

7 致谢

大约于 2018 年，大连交通大学 TOE 战队，桂林电子科技大学 Evolution 战队与香港科技大学 ENTERPRIZE 战队等便相继开源了各具特色的超级电容方案，推动了 RoboMaster 中各队对突破底盘功率限制的技术发展，也使得这一项目进入规则中。在此对各位拓荒的前辈们致以感谢。

2021 年，大连理工大学凌_BUG 战队首次开源了基于 Buck-Boost 的超级电容控制器。这种架构作为目前阶段（RM2024）中超级电容毋庸置疑的最优解，对本项目有极大的参考价值。

在超级电容的设计过程中，其他队伍的开源给予了我们莫大的帮助，在此感谢大连理工大学凌_BUG 战队、西交利物浦 GMaster 战队与仲恺奇点战队等极为优秀的开源。在赛季中也与来自华南农业大学 Taurus 战队、来自上海交通大学交龙战队的同学有过不少交流。

在 RM2024 赛季中，笔者由于水平有限，项目的起步显得有些艰难。感谢队内蒋益诚前辈的指导，他也是去年的超级电容项目的主导者。同时也要非常感谢队友们的传奇激励才让笔者能够顺利完成项目。

其实光有超级电容对于一个队伍来说是不够的，他的潜力需要搭配足够优秀的嵌入式功率控制系统和机械设计才能被有效发挥出来。因此需要感谢全体队友的付出才能让今年 ENTERPRIZE 战队有如此进步。2024 赛季已经进入尾声，愿 ENTERPRIZE 战队在 RM2025 继续前进，再创新高。

最后还是那句口号，初心高于胜负！

8 参考

i. RM0440 Reference manual - STM32G4 series advanced Arm®-based 32-bit MCUs. https://www.st.com/resource/en/reference_manual/rm0440-stm32g4-series-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

ii. 【RM2021-双向超级电容硬件开源】大连理工大学-凌 bug – https://bbs.robomaster.com/article/8435

iii. 【RM2023-数控超级电容方案开源】香港科技大学-ENTERPRIZE 战队 – https://bbs.robomaster.com/article/9452