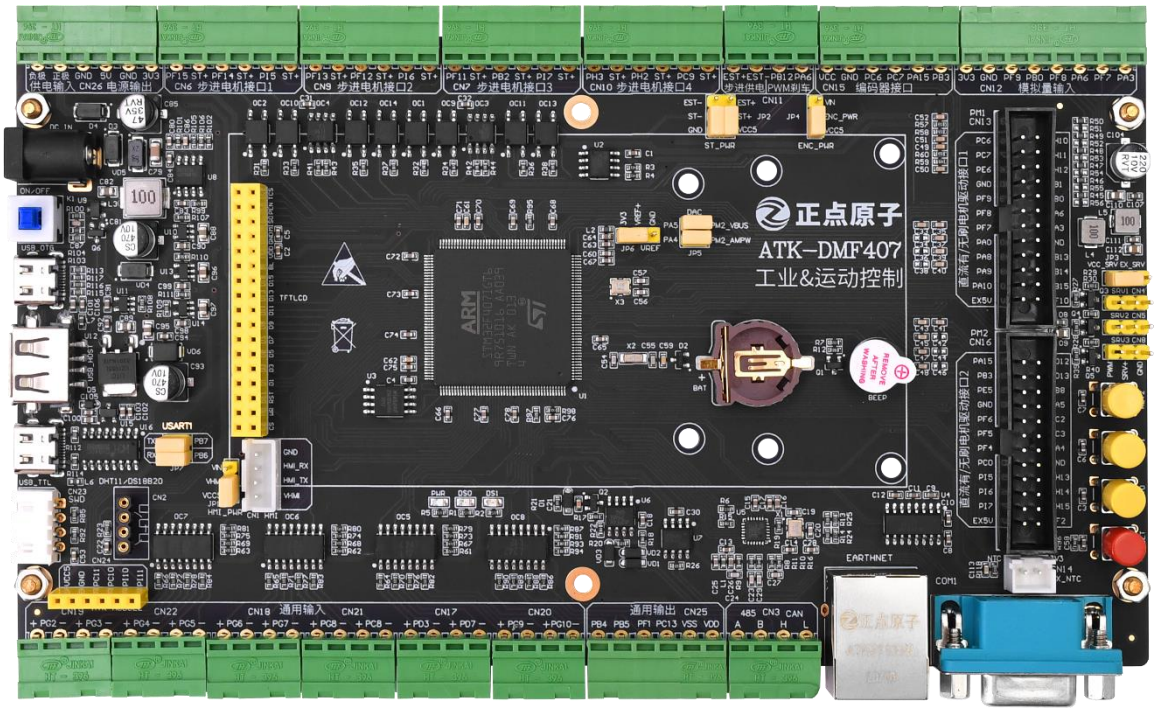
DMF407 电机控制专题教程 V1.0

－正点原子 DMF407 开发板电机专题教程



修订历史：

版本	日期	修改内容
V1.0	2022/06/07	第一次发布

正点原子公司名称 ：广州市星翼电子科技有限公司
原子哥在线教学平台：www.yuanzige.com
开源电子网 / 论坛 ：www.openedv.com
正点原子官方网站 ：www.alientek.com
正点原子淘宝店铺 ：https://openedv.taobao.com
正点原子 B 站视频 ：https://space.bilibili.com/394620890

电话：020-38271790 传真：020-36773971

请下载原子哥 APP，数千讲视频免费学习，更快更流畅。
请关注正点原子公众号，资料发布更新我们会通知。

扫码下载“原子哥”APP

扫码关注正点原子公众号

内容简介.....14
第一篇 基础篇.....15
第 1 章 本书的学习方法.....16
1.1 本书学习顺序.....16
1.2 本书参考资料.....16
1.3 本书编写规范.....16
1.4 本书代码规范.....17
1.5 例程资源说明.....17
1.6 学习资料查找.....18
1.7 给初学者的建议.....20
第 2 章 硬件平台简介.....22
2.1 配套开发板.....22
2.2 配套驱动器.....23
2.3 配套电机.....25
第 3 章 电机基础知识.....29
3.1 电机的概念.....29
3.2 电机的分类.....29
3.2.1 直流有刷电机.....29
3.2.2 直流无刷电机.....30
3.2.3 舵机.....30
3.2.4 步进电机.....31
3.2.5 伺服电机.....31
3.3 各类电机的特点.....31
3.4 电机常用名词.....32
第 4 章 电机控制相关的 STM32 外设.....33
4.1 电机控制与 STM32 定时器的关系.....33
4.2 基本定时器.....33
4.2.1 基本定时器简介.....33
4.2.2 硬件设计.....37
4.2.3 程序设计.....38
4.2.4 下载验证.....42
4.3 高级定时器.....43
4.3.1 高级定时器简介.....43

4.3.2 高级定时器中断实验 46
4.3.3 高级定时器 PWM 输出实验 52
4.3.4 高级定时器互补输出带死区刹车控制实验 62
4.3.5 高级定时器输入捕获实验 73
4.3.6 高级定时器 PWM 输入实验 84
4.3.7 高级定时器输出比较模式实验 94
4.3.8 高级定时器输出指定个数 PWM 实验 102
4.4 ADC 111
4.4.1 ADC 简介 111
4.4.2 单通道 ADC 采集实验 118
4.4.3 单通道 ADC 采集（DMA 读取）实验 130
4.4.4 多通道 ADC 采集（DMA 读取）实验 139
4.4.5 内部温度传感器实验 146
4.5 DAC 150
4.5.1 DAC 简介 150
4.5.2 DAC 输出实验 152
4.5.3 DAC 输出三角波实验 161
4.5.4 DAC 输出正弦波实验 166
第二篇 电机专题篇 172
第 5 章 直流有刷电机 173
5.1 直流有刷电机简介 173
5.2 直流有刷电机原理 174
5.3 直流有刷驱动板 176
5.4 直流有刷电机基础驱动实验 182
5.4.1 直流有刷电机基础驱动原理 182
5.4.2 硬件设计 182
5.4.3 程序设计 184
5.4.4 下载验证 191
第 6 章 直流有刷驱动板电压温度电流采集 192
6.1 电压温度电流采集原理 192
6.2 硬件设计 194
1. 例程功能 194
2. 硬件资源 194

3. 原理图 195
6.3 程序设计 195
ADC、DMA 配置步骤 196
6.3.1 程序流程图 197
6.3.2 程序解析 197
6.4 下载验证 204
第 7 章 直流有刷电机编码器测速 205
7.1 编码器简介 205
7.2 编码器测速原理 208
7.3 编码器测速实现 210
7.3.1 TIM2/TIM3/TIM4/TIM5 寄存器 211
7.3.2 硬件设计 213
7.3.3 程序设计 214
7.3.4 下载验证 224
第 8 章 PID 算法 225
8.1 自动控制系统 225
8.2 PID 算法入门 226
8.2.1 比例环节（ Proportion） 226
8.2.2 积分环节（Integral） 228
8.2.3 微分环节（Differential） 229
8.2.4 PID 算法离散公式 230
8.3 积分饱和问题 231
8.4 PID 算法代码实现 231
第 9 章 PID 参数整定 234
9.1 采样周期选择 234
9.2 PID 参数整定方法 234
9.2.1 试凑法 235
9.2.2 临界比例法 235
9.2.3 一般调节法 235
9.2.4 实际调参演示 236
9.3 正点原子 PID 上位机介绍 237
9.3.1 上位机通信协议 237
9.3.2 上位机使用方法 238

9.3.3 下位机协议代码实现 242
第 10 章 直流有刷电机速度环控制实现 246
10.1 速度环 PID 控制原理 246
10.2 硬件设计 246
1. 例程功能 246
2. 硬件资源 246
3. 原理图 247
10.3 程序设计 247
速度环 PID 控制的配置步骤 247
10.3.1 程序流程图 248
10.3.2 程序解析 248
10.4 下载验证 254
第 11 章 直流有刷电机电流环控制实现 255
11.1 电流环 PID 控制原理 255
11.2 硬件设计 255
1. 例程功能 255
2. 硬件资源 255
3. 原理图 256
11.3 程序设计 256
电流环 PID 控制的配置步骤 256
11.3.1 程序流程图 257
11.3.2 程序解析 257
11.4 下载验证 263
第 12 章 直流有刷电机位置环控制实现 265
12.1 位置环 PID 控制原理 265
12.2 硬件设计 265
1. 例程功能 265
2. 硬件资源 265
3. 原理图 266
12.3 程序设计 266
位置环 PID 控制的配置步骤 266
12.3.1 程序流程图 267
12.3.2 程序解析 267

12.4 下载验证 272
第 13 章 直流有刷电机三环控制实现 274
13.1 三环 PID 控制原理 274
13.2 硬件设计 274
1. 例程功能 274
2. 硬件资源 275
3. 原理图 275
13.3 程序设计 275
三环 PID 控制的配置步骤 275
13.3.1 程序流程图 276
13.3.2 程序解析 277
13.4 下载验证 283
第 14 章 舵机控制 285
14.1 舵机的简介以及分类 285
14.2 舵机的结构 285
14.3 舵机的工作原理以及控制原理 286
14.4 舵机的参数介绍 288
14.5 舵机的控制实现 288
14.5.1 硬件设计 288
14.5.2 程序设计 290
14.5.3 下载验证 295
第 15 章 步进电机 297
15.1 步进电机的简介 297
15.2 步进电机的分类 297
15.3 步进电机的工作原理 298
15.3.1 单极性步进电机驱动原理 299
15.3.2 双极性步进电机驱动原理 300
15.3.3 细分驱动原理 302
15.4 步进电机的指标参数 303
15.4.1 静态指标术语 303
15.4.2 动态指标术语 303
15.5 步进电机驱动器介绍 303
15.5.1 正点原子步进电机细分驱动器介绍 304

15.5.2 驱动器控制接口 305
15.5.3 步进电机驱动器实物接线 308
15.6 步进电机旋转控制实现 312
15.6.1 硬件设计 312
15.6.2 程序设计 314
15.6.3 下载验证 326
15.7 步进电机定位控制实现 326
15.7.1 硬件设计 326
15.7.2 程序设计 327
15.7.3 下载验证 335
第 16 章 步进电机梯形加减速 337
16.1 梯形加减速的简介 337
16.2 梯形加减速的原理分析 338
16.2.1 确定未知量和已知量 338
16.2.2 三角形运动变化的加减速阶段所需脉冲数 339
16.2.3 梯形运动变化的加减速阶段所需脉冲数 341
16.2.4 脉冲时间间隔的精确计算 342
16.3 梯形加减速整体流程 345
16.3.1 不同速度段的处理方法 345
16.4 梯形加减速的控制实现 346
16.4.1 硬件设计 346
16.4.2 程序设计 347
16.4.3 下载验证 353
第 17 章 步进电机 S 形加减速 355
17.1 S 形曲线加减速的简介 355
17.2 S 形曲线加减速的的原理分析 355
17.2.1 确定已知量和未知量 357
17.2.2 获取加加速度段的步数 S1 358
17.2.3 获取减加速度段的步数 S2 359
17.2.4 确定下一个脉冲的周期 359
17.3 S 型曲线加减速的控制实现 360
17.3.1 硬件设计 360
17.3.2 程序设计 360

17.3.3 下载验证 366
第 18 章 步进电机直线插补 368
18.1 插补的简介 368
18.2 插补算法 368
18.3 逐点比较法直线插补原理 369
18.3.1 逐点比较法整体流程图 369
18.3.2 偏差判别 369
18.3.3 坐标进给 370
18.3.4 偏差计算 370
18.3.5 终点判别 370
18.4 任意象限直线插补的原理 371
18.5 任意象限直线插补实验 371
18.5.1 硬件设计 371
18.5.2 程序设计 372
18.5.3 下载验证 379
第 19 章 步进电机圆弧插补 380
19.1 圆弧插补的简介 380
19.2 逐点比较法圆弧插补原理 380
19.2.1 偏差判别 380
19.2.2 坐标进给 381
19.2.3 偏差计算 381
19.2.4 终点判别 381
19.3 任意象限圆弧插补原理 382
19.4 任意象限圆弧插补实验 383
19.4.1 硬件设计 383
19.4.2 程序设计 383
19.4.3 下载验证 391
第 20 章 步进电机闭环系统（位置环） 392
20.1 步进电机闭环系统组成 392
20.2 步进电机闭环系统原理 392
20.3 步进电机位置闭环控制实现（增量式/位置式） 393
20.3.1 硬件设计 393
20.3.2 程序设计 393

20.3.3 下载验证 401
第 21 章 直流无刷电机 403
21.1 直流无刷电机的简介 403
21.1.1 直流无刷电机与直流有刷电机的特点 403
21.1.2 直流无刷电机的分类 404
21.1.3 BLDC 和 PMSM 的区别 405
21.1.4 直流无刷电机的主要参数介绍 405
21.2 直流无刷电机的基本控制原理 406
21.3 直流无刷电机的霍尔传感器控制基本原理 411
21.4 直流无刷电机的驱动硬件平台简介 414
21.4.1 店铺 BLDC 的技术参数 414
21.4.2 无刷电机驱动板介绍 414
第 22 章 有感方波驱动 420
22.1 硬件设计 420
1、例程功能 420
2、硬件资源 420
3、原理图 420
22.2 程序设计 421
22.2.1 无刷电机基本控制的配置步骤 421
22.2.2 程序流程图 422
22.2.3 程序解析 422
22.3 下载验证 431
第 23 章 无刷电机电压温度电流采集 432
23.1 驱动板硬件电路原理分析 432
23.1.1 三相电流采集电路分析 432
23.1.2 电源电压采集电路分析 433
23.1.3 温度采集电路分析 433
23.2 硬件设计 434
1、例程功能 434
2、硬件资源 434
3、原理图 435
23.3 程序设计 436
23.3.1 无刷电机参数采集的配置步骤 436

23.3.2 程序流程图 437
23.3.3 程序解析 437
23.4 下载验证 447
第 24 章 无刷电机有感方波闭环控制 448
24.1 速度闭环控制 448
24.1.1 无刷有感测速原理 448
24.1.2 硬件设计 449
24.1.3 程序设计 451
24.1.4 下载验证 460
24.2 速度&电流双闭环控制 462
24.2.1 硬件设计 462
24.2.2 程序设计 464
24.2.3 下载验证 473
第 25 章 直流无刷电机无感控制 474
25.1 直流无刷电机的无感驱动介绍 474
25.1.1 反电动势法控制 BLDC 电机的原理 474
25.1.2 反电动势的检测方法 475
25.1.3 比较器模式采样过零信号 475
25.1.4 闭环的建立 476
25.1.5 BLDC 方波启动技术 477
25.2 硬件设计 478
1、例程功能 478
2、硬件资源 479
3、原理图 479
25.3 程序设计 480
25.3.1 无刷电机无感控制的配置步骤 480
25.3.2 程序流程图 481
25.3.3 程序解析 481
25.4 下载验证 486
第 26 章 无刷电机无感方波闭环控制 487
26.1 无感速度闭环控制 487
26.1.1 无刷无感测速原理 487
26.1.2 硬件设计 487

26.1.3 程序设计 488
26.1.4 下载验证 493
26.2 无感速度+电流双闭环控制 495
26.2.1 硬件设计 495
26.2.2 程序设计 497
26.2.3 下载验证 504
第三篇 FOC 篇 506
第 27 章 FOC 507
27.1 FOC 原理介绍 507
27.1.1 FOC 框图介绍 508
27.1.2 FOC 坐标变换 508
27.1.3 FOC 的目的 512
27.1.4 SVPWM 解析 513
27.2 FOC 硬件平台 520
27.2.1 驱动板硬件电路介绍 520
27.2.2 电机参数介绍 522
27.3 FOC 开发工具 522
27.3.1 Motor Control Workbench 简介 523
27.3.2 Motor Control Workbench 获取与安装 523
27.3.3 STM32CubeMX 的获取与安装 524
27.4 FOC 例程创建 525
27.5 FOC 下载验证 539
27.6 ST MCSDK 库源码介绍 544
27.7 FOC 例程分析 547

内容简介

本书将由浅入深，带领大家学习使用 STM32F407 来控制电机，为您开启 STM32F407 的电机控制学习之旅。

本书总共分为三篇：

1， 基础篇，主要介绍介绍电机的基础入门知识，包括电机的概念、原理、特点以及电机控制相关的 STM32 外设等。必须好好学习并掌握；

2， 电机专题篇，主要是介绍各类电机的控制，包括直流有刷电机、舵机、步进电机、BLDC 等。必须好好学习并掌握；

3， FOC 篇，主要介绍 PMSM 的 FOC 控制，包括：FOC 理论知识、环境安装、FOC 相关例程等。这些例程难度相对高一点，大家一定要沉下心来，戒骄戒躁，循序渐进的学习。对于一些较难的知识点，我们只需要知道怎么用就可以了，并不需要详细研究具体实现过程，比如 FOC 算法库的实现原理；

本书为正点原子 DMF407 电机开发板的配套教程，在开发板配套的光盘里面，有详细原理图以及所有实例的完整代码，这些代码都有详细的注释，所有源码都经过我们严格测试，不会有任何警告和错误，另外，源码有我们生成好的 hex 文件，大家只需要通过仿真器下载到开发板即可看到实验现象，亲自体验实验过程。

本书并没有详细介绍 DMF407 电机开发板的硬件，硬件部分内容我们单独编写了一个文档，请参考：《DMF407 硬件参考手册.pdf》进行学习。

本书不仅非常适合广大学生和电子爱好者学习 STM32F407 以及电机控制，其大量的实验以及详细的解说，也是公司产品开发的不二参考。

第一篇 基础篇

万事开头难，如果打好了基础，后面学习就事半功倍了！本篇将详细介绍电机的基础知识，包括：电机的概念、分类、结构、工作原理、电机控制相关的 STM32 外设介绍等部分。学好了这些基础知识，在后面的例程学习部分，将会有非常大的帮助，能极大的提高大家的学习效率。

如果您是初学者，建议好好学习本并理解这些知识，手脑并用，不要漏过任何内容，一遍学不会的可以多学几遍，总之这些知识点都要掌握。

本篇将分为如下章节：

1，本书的学习方法
2，硬件平台简介
3，电机基础知识
4，电机控制相关的 STM32 外设

第 1 章 本书的学习方法

为了让大家更好的学习和使用本书，本章将给大家介绍一下本书的学习方法，包括：本书的学习顺序、编写规范、代码规范、资料查找、学习建议等内容。

本章将分为如下几个小节：

1.1，本书学习顺序
1.2，本书参考资料
1.3，本书编写规范
1.4，本书代码规范
1.5，例程资源说明
1.6，学习资料查找
1.7，给初学者的建议

1.1 本书学习顺序

为了让大家更好的学习和使用本书，我们做了以下几点考虑：

1，坚持循序渐进的思路编写，从基础到入门，从简单到复杂。

2，将知识进行分类介绍，简化学习过程，包括：基础篇、电机专题篇、FOC 篇。

3，将硬件介绍独立成一个文档(《DMF407 硬件参考手册.pdf》)，本书着重介绍软件知识。

因此，大家在学习本书的时候，我们建议：先通读一遍《DMF407 硬件参考手册.pdf》，对开发板的硬件资源有个大概了解，然后从基础篇开始，再到电机专题篇，最后学习 FOC 篇，循序渐进，逐一攻克。

对初学者来说，尤其要按照以上顺序学习，不要跳跃式学习，因为我们书本的知识都是一环扣一环的，如果前面的知识没学好，后面的知识学起来就会很困难。

对于已经学过 STM32 以及电机的朋友来说，就可以跳跃式学习了，当然如有不懂，也得翻阅前面的知识点进行巩固。

1.2 本书参考资料

本书的主要参考资料有以下几份文档：

《STM32F4xx 参考手册_V4（中文版）》

《STM32F407xx 数据手册_V4（英文版）》

前者是 ST 官方针对 STM32 的一份通用参考资料，重点介绍 STM32 内部资源及使用，寄存器描述等，内容翔实，但是没有实例，读者只能根据自己对书本的理解来编写相关代码。

后者是专门介绍 STM32 芯片硬件设计方面的书，其内容包括：芯片的引脚功能、外设复用情况、电气特性等。所以，在学习 STM32 的时候，必须结合这份资料来看。

另外，由于 STM32F4 参考手册的中文版是 V4 版本的，而最新的 STM32F4 参考手册已经是 V6 的了，所以大家在遇到一些有问题/矛盾的地方，可以参考最新的英文版参考手册：

《STM32F4xx 参考手册_V6（英文版）》

这几份参考文档都在我们提供的光盘资料里面可以找到，路径：A 盘→8，STM32 参考资料 文件夹下可以找到。

1.3 本书编写规范

本书通过数十个例程，给大家详细介绍如何使用 STM32 控制电机，按难易程度以及知识结构，我们将本书分为三个篇章：基础篇、电机专题篇和 FOC 篇。

基础篇，共 4 章，主要是电机基础知识介绍，包括电机的概念、分类、结构、工作原理、电机控制相关的 STM32 外设介绍等，这些章节在结构上没有共性，但是互相有关联，有一个集成的关系在里面，即：必须先学了前面的知识，才好学习后面的知识点。

电机专题篇和 FOC 篇，共 23 章，详细介绍了多种电机的控制原理，并且还介绍了一些非常实用的程序代码。这部分内容占了本书的绝大部分篇幅，而且这些章节在结构上都比较有共性，一般分为 4 个部分，如下：

1，控制原理介绍
2，硬件设计
3，程序设计
4， 下载验证

控制原理介绍，简单介绍具体章节所要用到的电机控制原理，让大家对所用电机的控制原理有一个基本了解，方便后面的程序设计。

硬件设计，包括具体章节的实验具体功能说明、所用到的硬件资源及原理图连接方式，从而知道要做什么？ 需要用到哪些 IO 口？ 是怎么接线的？ 方便程序设计的时候编写驱动代码。

程序设计，一般包括：驱动介绍、配置步骤、程序流程图、关键代码分析、main 函数讲解等部分。一点点介绍程序代码是怎么来的，注意事项等，从而学会整个代码。

下载验证，属于实践环节，在完成程序设计后，教大家如何下载并验证我们的例程是否正确？ 完成一个闭环过程。

1.4 本书代码规范

为了方便大家编写高质量代码，我们对本书的代码风格进行了统一，详细的代码规范说明文档，见光盘：A 盘→1，入门资料→《嵌入式单片机 C 代码规范与风格.pdf》，初学者务必好好学习一下这个文档。

总结几个规范的关键点：

1，所有函数/变量名字非特殊情况，一般使用小写字母；
2，注释风格使用 doxgen 风格，除屏蔽外，一律使用 /* */ 方式进行注释；
3，TAB 键统一使用 4 个空格对齐，不使用默认的方式进行对齐；
4，每两个函数之间，一般有且只有一个空行；
5，相对独立的程序块之间，使用一个空行隔开；
6，全局变量命名一般用 g_开头，全局指针命名一般用 p_开头；
7，if、for、while、do、case、switch、default 等语句单独占一行，一般无论有多少行执行语句，都要用加括号：{}。

1.5 例程资源说明

DMF407 电机开发板提供的电机专题例程多达 73 个（基于 HAL 库）。我们提供的这些例程，基本都是原创，拥有非常详细的注释，代码风格统一、循序渐进，非常适合初学者入门。

DMF407 电机开发板的例程列表如表 1.5.1 所示：

编号	实验名字	编号	实验名字
1-1	基本定时器中断实验	6-7	直流有刷电机-位置环 PID 控制实验
2-1	通用定时器中断实验	6-8	直流有刷电机-速度环+位置环双环控
2-2	通用定时器 PWM 输出实验	6-9	直流有刷电机- 电流环+位置环双环控制实验
2-3	通用定时器输入捕获实验	6-10	直流有刷电机- 电流环+速度环双环控制实验
2-4	通用定时器 PWM 输入实验	6-11	直流有刷电机-电流环+速度环+位置环三环控制实验
2-5	通用定时器输出比较实验	7	舵机实验
2-6	通用定时器编码器接口模式实验（驱动接口 1）	8-1	步进电机基础驱动（PWM 模式）

2-7	通用定时器编码器接口模式实验（驱动接口 2）	8-2	步进电机基础驱动（翻转模式）
3-1	高级定时器中断实验	8-3	步进电机变速控制（PWM 模式）
3-2	高级定时器 PWM 输出实验	8-4	步进电机变速控制（翻转模式）
3-3	高级定时器互补输出带死区刹车实验	8-5	步进电机定位控制（PWM 模式）
3-4	高级定时器输入捕获实验	8-6	步进电机定位控制（翻转模式）
3-5	高级定时器 PWM 输入实验	8-7	步进电机梯形加减速运动
3-6	高级定时器比较输出实验	8-8	步进电机 S 型加减速运动
3-7	高级定时器输出指定个数 PWM 实验	8-9	步进电机任意象限直线插补运动
4-1	单通道 ADC 采集实验	8-10	步进电机任意象限圆弧插补运动
4-2	单通道 ADC 采集（使用 DMA）实验	8-11	步进电机任意象限直线圆弧插补运动
4-3	多通道 ADC 采集实验	8-12	步进电机闭环控制（位置环）
4-4	多通道 ADC 采集（使用 DMA）实验	9-1	BLDC 基础驱动
4-5	内部温度传感器实验	9-2	BLDC 基础驱动 2 路
4-6	定时器触发 ADC 采集实验	9-3	BLDC 电源电压-温度-三相电流采集
5-1	DAC 输出实验	9-4	BLDC 的速度环 PID 控制实验
5-2	DAC 输出三角波实验	9-5	BLDC 双环 PID 控制-速度环+电流环
5-3	DAC 输出正弦波实验	9-6	BLDC 方波无感 6 步换向
6-1	直流有刷电机-基础驱动实验	9-7	BLDC 方波无感过零检测控制
6-2	直流有刷电机-基础驱动（2 路）实验	9-8	BLDC 方波无感 PID 速度环控制
6-3	直流有刷电机-电压、电流、温度检测实验	9-9	BLDC 方波无感 PID 速度环+电流环控制
6-4	直流有刷电机-编码器测速实验	10	PMSM_FOC 专题实验（15 个）
6-5	直流有刷电机-速度环 PID 控制实验	11	电机综合测试实验
6-6	直流有刷电机-电流环 PID 控制实验		

表 1.5.1 DMF407 电机开发板例程表

从上表可以看出，正点原子 DMF407 电机开发板的例程基本上涵盖了电机控制相关的 STM32 外设，并且在电机控制方面实现了很多有价值的例程。

从上表可知，例程安排是循序渐进的，首先从最基础的基本定时器中断开始，然后一步步深入，从简单到复杂，有利于大家的学习和掌握。所以，正点原子 DMF407 电机开发板是非常适合初学者的。当然，对于想深入学习 STM32 以及电机控制的朋友，正点原子 DMF407 电机开发板也绝对是一个不错的选择。

1.6 学习资料查找

学习资料包括三个方面：

1，ST 官方的学习资料

ST 官方资料有两个网址：www.stmcu.org 和 www.st.com 。

www.stmcu.org 是 ST 中文社区，里面的资料全部由 ST 中国区的人负责更新和整理，包含了所有 ST 公司的 MCU 资料，比如：STM32F4 最新的芯片文档（参考手册、数据手册、勘误手册、编程手册等）、软件资源（固件库、配置工具、PC 软件等）、硬件资源（各种官方评估板）等，如图 1.6.1 所示：

图 1.6.1 STM32F4 相关资料（stmcu.org）

www.st.com 是 ST 官网，ST 最新最全的资料，一般都是放在该网站，ST 中文社区的资料，一般都是从 ST 官网搬过来的，所以如果你想找最新的 STM32 官方资料，应该在 ST 官网找。对于初学者，一般从 ST 中文社区获取 ST 官方资料就可以了。ST 官网的 STM32F4 资料页面如图 1.6.2 所示（注意：默认是英文语言，需要在网页右上角设置成中文）：

图 1.6.2 STM32F407 官网资料（st.com）

① ，STM32F407 的硬件相关资源，在：产品→微控制器→STM32 ARM Cortex 32 位微控制器→STM32 高性能 MCU→STM32F4 系列→STM32F407 →文件路径下面可以找到。

② ，STM32F407 的软件相关资源，在：工具与软件→嵌入式软件→微控制器软件→STM32 微控制器软件 路径下面可以找到。

2，正点原子的学习资料

正点原子给大家提供的学习资料，都放在正点原子文档中心，大家可以在文档中心下载所有正点原子最新最全的学习资料，文档中心地址：www.openedv.com/docs/index.html ，如图 1.6.3 所示：

图 1.6.3 正点原子文档中心

在文档中心下面，我们可以找到正点原子所有开发板、模块、产品等的详细资料下载地址。

3，正点原子论坛

正点原子论坛，即开源电子网：www.openedv.com/forum.php ，该论坛从 2010 年成立至今，已有十余年时间，数十万注册用户，STM32 相关帖子数量有 20 多万，每天数百人互动，是一个非常好的 STM32 学习交流平台。

在学习过程中，我们难免会遇到一些问题，有任何问题，大家都可以先去开源电子网搜索一下，基本上你能遇到的问题，我们论坛都有人问过了，所以可以很方便的找到一些参考解决方法。如果实在找不到，你也可以在论坛提问，每天原子哥都会在上面给大家做解答。

不过，大家在论坛发帖的时候，建议先阅读一下提问的智慧，缕清思路，不要乱问一通，提高提问质量，提问的智慧阅读地址：www.openedv.com/thread-7245-1-1.html 。

1.7 给初学者的建议

对于学习 STM32 以及电机控制，这里我给大家提以下三点建议：

1，准备一款合适的开发板（强烈建议配仿真器）

任何实验，我们都需要验证，最好的验证方式就是在开发板上面实际跑起来，然后通过仿真器仿真调试，查看具体的执行过程。仿真器可以加深印象，还可以方便的查找 bug，所以学习 STM32 以及电机控制必备：一个开发板 + 一个仿真器 + 一个驱动器 + 所学习的电机。

另外，开发板在精不在多，学好一款，基本上就够用了。

2，两本参考资料，即《STM32F4xx 参考手册_V4（中文版）》和《STM32F407xx 数据手册_V4（英文版）》。

这两个手册在本章 1.2 节有过介绍，对于我们学习电机控制相关的 STM32 外设和了解 STM32 硬件资源非常有帮助，是学习 STM32 以及电机控制的必备资料，因此初学者尤其要多看这两个手册。

3，戒骄戒躁，勤思敏行。

学习 STM32 以及电机控制千万不能浮躁，更加不能骄傲，初学者学习 STM32 以及电机控制会遇到很多问题和难点，这个时候千万不能浮躁，不要带情绪，一定要静下心来，缕清思路，逐一攻克。

我就曾经遇到一个问题半个月都没解决的情况，但是这半个月我尝试并掌握了很多解决问题的方法，最终解决问题的时候，其实带来的收获远远大于问题本身。所以不要遇到问题就认怂，就想问别人，问老师，先尝试自己解决一下，比如花个十天半个月去解决一个问题，我相信你也会有很多收获。

学习我们教程的时候，要多思考，多想想为什么要这么写？有没有其他更好的办法？然后，自己去验证，去实践。这里非常重要的一点是要多实践，一定要自己动手写代码，然后再下载到开发板验证，不要只是看看视频，看看例程就算完了，要能做到举一反三，如果自己不实践，不动手写代码，是很难真正学会的。

最后，C 语言是学习 STM32 的必备知识，所以如果 C 语言不过关的朋友，得先好好学习一下 C 语言基础，否则学起来会比较吃力。

第 2 章 硬件平台简介

本章我们主要来介绍本教程所配套的硬件平台，包括：开发板、驱动器、电机等。大家可以根据自己的需求来选择相应的硬件。

本章分为如下几个小节：

2.1 配套开发板
2.2 配套驱动器
2.3 配套电机

2.1 配套开发板

本教程所配套的开发板为 ATK-DMF407 电机开发板，此开发板可用于多种电机的控制，包括：直流有刷电机、直流无刷电机、舵机、步进电机、PMSM 等。若您使用的是其他开发板，则可以参考或借鉴本教程的内容。ATK-DMF407 电机开发板的外观如图 2.1.1 所示：

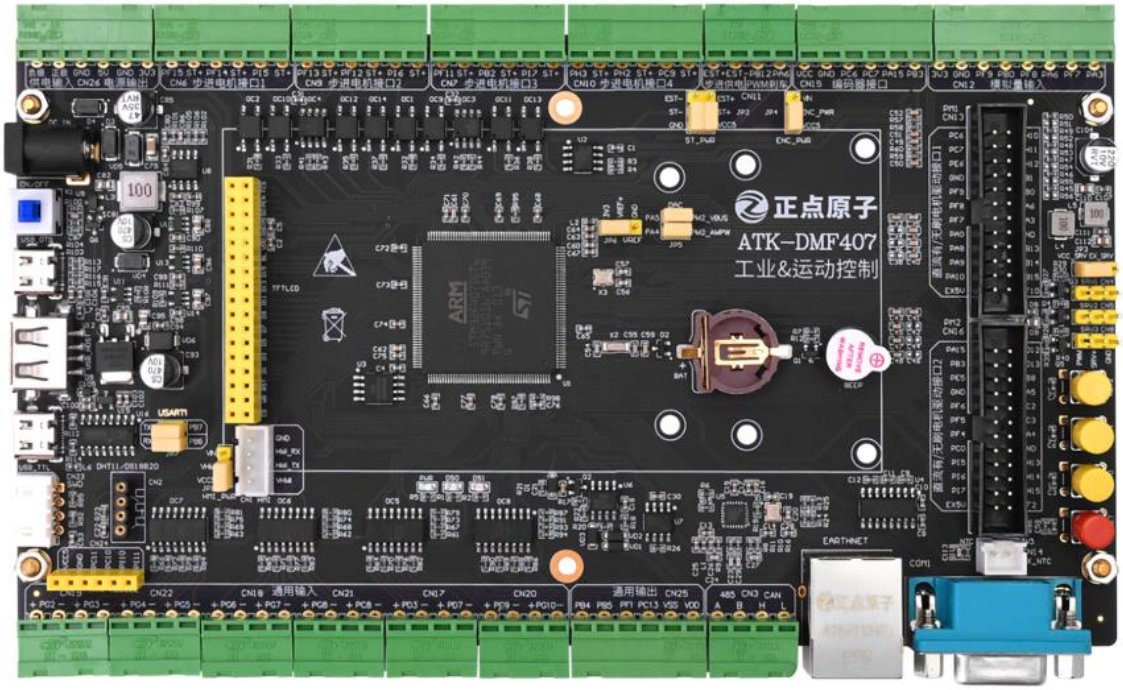


图 2.1.1 ATK-DMF407 电机开发板

ATK-DMF407 电机开发板的硬件参数如表 2.1.1 所示：

项目	说明
产品型号	ATK- DMF407 V1.0
CPU	STM32F407IGT6，LQFP176
外形尺寸	110mm*196.5mm
工作电压	5V（USB）、DC6V~24V（DC005）
工作电流	100mA~140mA（@5V）
工作温度	-40℃~+85℃

表 2.1.1 ATK-DMF407 电机开发板硬件参数

注意：由于 ATK-DMF407 电机开发板的硬件资源介绍篇幅过长，这里单独分出了硬件参考手册文档，大家可以在我们提供的光盘资料里面查找，路径：A 盘→《DMF407 硬件参考手册.pdf》。这一本手册对于学习电机控制非常重要，大家至少要通读一遍！

2.2 配套驱动器

由于 STM32 的 IO 输出能力非常有限，并不能直接驱动大功率电机，所以需要有一个驱动器对输出信号进行放大或者转换，这样才可以正常驱动电机。大家需要注意，不同类型的电机，驱动方式是不一样的，所以对应的驱动器也是有区别的，切记不要混用！下面我们来介绍几款正点原子研发的电机驱动器：

1）步进电机驱动器

本教程所配套的步进电机驱动器为 ATK-2MD5050 步进电机驱动器，此驱动器可用于驱动常用的 42mm、57mm、86mm 两相混合式步进电机，其外观如图 2.2.1 所示：

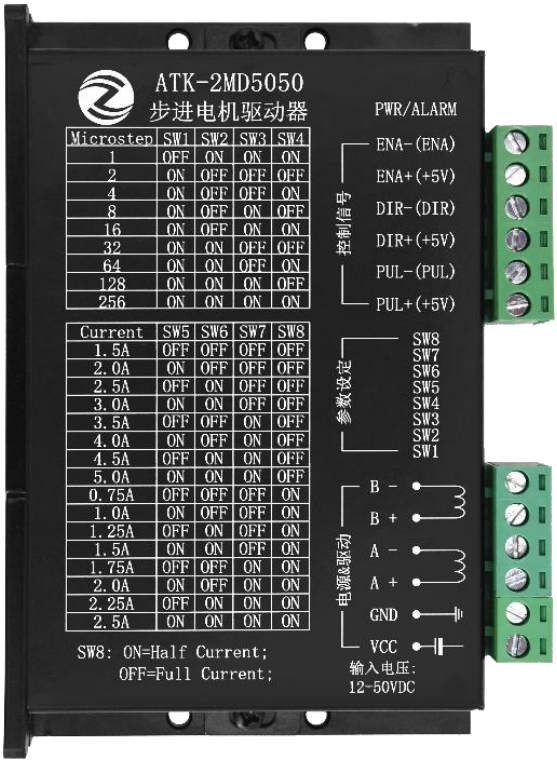


图 2.2.1 步进电机驱动器

ATK-2MD5050 步进电机驱动器的硬件参数如表 2.2.1 所示：

项目	说明
输入电压	DC 12~50V
驱动方式	双极恒流 PWM 驱动输出
输出电流	全流：1.5~5.0A，8 档可调，分辨率 0.5A 半流：0.75~2.5A，8 档可调，分辨率 0.25A
温度	工作温度：-10℃~60℃；存储温度-40℃~80℃
使用场合	避免粉尘、油雾以及腐蚀性气体;避开过热潮湿环境
冷却方式	自然冷却
安装尺寸	118mm*76mm*33mm（长*宽*高）
重 量	250 克

表 2.2.1 ATK-2MD5050 步进电机驱动器硬件参数

注意：由于 ATK-2MD5050 步进电机驱动器的产品介绍篇幅过长，这里单独分出了用户手册文档，大家可以在我们提供的光盘资料里面查找，路径：A 盘→9，增值资料→2，正点原子电机驱动器资料→《ATK-2MD5050 步进电机驱动器用户手册.pdf》。这一本手册对于学习步进电机控制非常重要，大家在学习步进电机章节之前，一定要先看这本手册！

2）直流有刷驱动板

本教程配套的直流有刷驱动板为 ATK-PD6010D 直流有刷驱动板，其外观如图 2.2.2 所示：

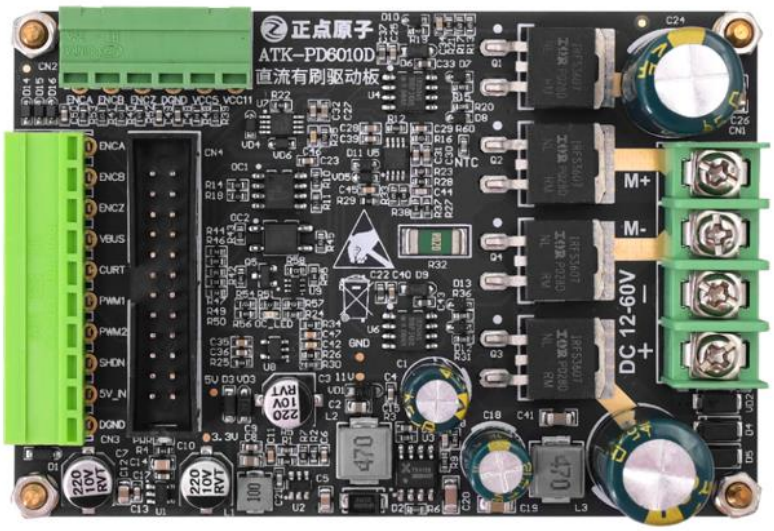


图 2.2.2 直流有刷驱动板

ATK-PD6010D 直流有刷驱动板的硬件参数如表 2.2.2 所示：

项目	说明
工作电压	DC 12~60V
最大输出电流	10 A
最大驱动功率	600 W
尺寸	95mm*65mm（长*宽）

表 2.2.2 ATK-PD6010D 直流有刷驱动板硬件参数

注意：由于 ATK-PD6010D 直流有刷驱动板的产品介绍篇幅过长，这里单独分出了用户手册文档，大家可以在我们提供的光盘资料里面查找，路径：A 盘→9，增值资料→2，正点原子电机驱动器资料→《ATK-PD6010D 直流有刷驱动板用户手册.pdf》。这一本手册对于学习直流有刷电机控制非常重要，大家在学习直流有刷电机章节之前，一定要先看这本手册！

3）直流无刷驱动板

本教程配套的直流无刷驱动板为 ATK-PD6010B 直流无刷驱动板，其外观如图 2.2.3 所示：

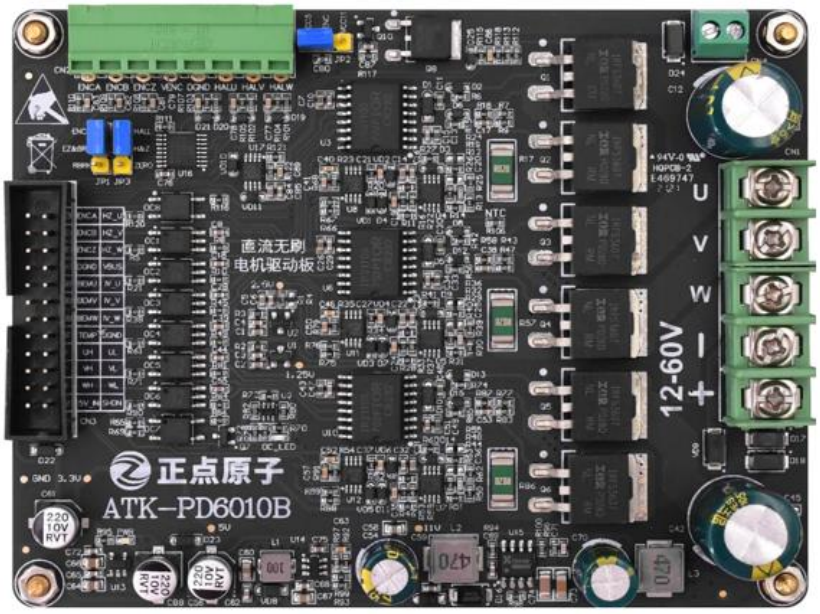


图 2.2.3 直流无刷驱动板

ATK-PD6010B 直流无刷驱动板的硬件参数如表 2.2.3 所示：

项目	说明
工作电压	DC12~60V
最大输出电流	10A
最大驱动功率	600W
尺寸	115mm*90mm（长*宽）

表 2.2.3 ATK-PD6010B 直流无刷驱动板硬件参数

注意：由于 ATK-PD6010B 直流无刷驱动板的产品介绍篇幅过长，这里单独分出了用户手册文档，大家可以在我们提供的光盘资料里面查找，路径：A 盘→9，增值资料→2，正点原子电机驱动器资料→《ATK-PD6010B 直流无刷驱动板用户手册.pdf》。这一本手册对于学习直流无刷电机控制非常重要，大家在学习直流无刷电机章节之前，一定要先看这本手册！

2.3 配套电机

1）直流有刷减速电机

本教程配套的直流有刷电机为 ATK-JGB37-520E 直流有刷减速电机，其外观如图 2.3.1 所示：

图 2.3.1 直流有刷减速电机

ATK-JGB37-520E 直流有刷减速电机的硬件参数如表 2.3.1 所示：

项目	说明
工作电压	DC 12V
电流	0.2~0.5A
空载转速	333RPM（峰值）
编码器功能	有（增量式）
可否调速	可以
可否正反转	可以
减速比	30:1

表 2.3.1 ATK-JGB37-520E 直流有刷减速电机硬件参数

2）直流无刷电机（BLDC）

本教程配套的直流无刷电机为 ATK-BL57H95D20E 直流无刷电机，其外观如图 2.3.2 所示：

图 2.3.2 直流无刷电机（BLDC）

ATK-BL57H95D20E 直流无刷电机的硬件参数如表 2.3.2 所示：

项目	说明	项目	说明
磁级数	4	峰值扭矩	4.2 N·m
相数	3	峰值电流	19.8 A
额定电压	24V DC	力矩常数	0.063 N·m/A
额定转速	3000 RPM	反电势	6.6 V/kRPM
额定扭矩	0.33 N·m	线电阻	0.4Ω
额定电流	6.6 A	线电感	0.8 mH
输出功率	105W	转动惯量	173 g·cm²

表 2.3.2 ATK-BL57H95D20E 直流无刷电机硬件参数

3）永磁同步电机（PMSM）

本教程配套的永磁同步电机为 42JSF630AS-1000 永磁同步电机，其外观如图 2.3.3 所示：

图 2.3.3 永磁同步电机

42JSF630AS-1000 永磁同步电机的硬件参数如表 2.3.3 所示：

项目	说明	项目	说明
磁级数	8	线电感	0.59 mH
相数	3	编码器线数	1000

额定电压	24V DC	霍尔安装角度	120°
额定转速	3000 RPM	输出功率	64W
额定电流	4 A	额定扭矩	0.2 N • m
反电势	4.3 V/kRPM	机身长度	97mm
线电阻	1.02 Ohm	机座尺寸	42*42mm

表 2.3.3 42JSF630AS-1000 永磁同步电机硬件参数

4）步进电机

本教程配套的步进电机为 ATK-S42H40D20 步进电机，其外观如图 2.3.4 所示：

图 2.3.4 步进电机

ATK-S42H40D20 步进电机的硬件参数如表 2.3.4 所示：

项目	说明
型号	ATK-S42H40D20
尺寸	长宽高：42*42*40（不含电机轴）
扭矩	0.46N.m
额定电流	1.7A
出线方式	两相四线（黑：A+ 红：A- 蓝：B- 绿：B+）
线长	30cm
相电阻	2Ω
相电感	3.2mH

表 2.3.4 ATK-S42H40D20 步进电机硬件参数

5）舵机

本教程配套的舵机为 MG996R 舵机，其外观如图 2.3.5 所示：

图 2.3.5 舵机

MG996R 舵机的硬件参数如表 2.3.5 所示：

项目	说明
型号	MG996R 舵机
工作电压	DC 4.8~6V
空载电流	4.8V 供电：0.22A；6V 供电：0.25A
最大扭矩	4.8V 供电：11kg • cm；6V 供电：13 kg • cm
堵转电流	2~2.2A

表 2.3.5 MG996R 舵机硬件参数

第 3 章 电机基础知识

本章，我们主要介绍电机的一些基础知识，让大家对电机有一个初步的了解，为后面电机控制的学习做铺垫，方便后面的学习。本章内容大家第一次看的时候了解一个大概即可，后面需要用到这方面知识的时候可以再回过头来针对性地学习。

本章分为如下几个小节：

3.1 电机的概念
3.2 电机的分类
3.3 各类电机的特点
3.4 电机常用名词

3.1 电机的概念

电机是一种可以实现电能与机械能之间相互转换的设备，其可分为电动机和发电机两种，前者能够把电能转换成机械能，而后者则能把机械能转换成电能。大家需要注意，在电机专题教程中，如果没有特殊说明，我们所说的电机都是电动机。

3.2 电机的分类

电机的种类繁多，其可按工作电源、用途、转子结构、运转速度等进行分类。这里我们以用途为例，对常见的电机进行分类，具体的框图如图 3.2.1 所示：

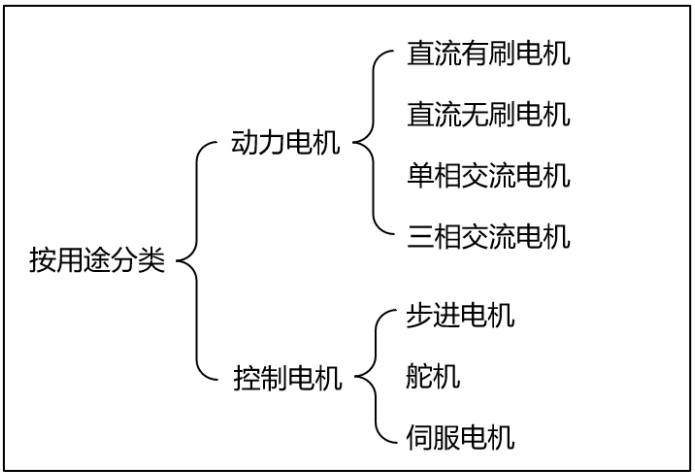


图 3.2.1 电机按用途分类

我们下面对几种常用的电机进行简单的介绍，大家只需要有一个大概的了解即可，而更详细的内容将会在对应的电机专题中展开。

3.2.1 直流有刷电机

直流有刷电机（BDC）是一种内含电刷装置，可以将直流电能转换成机械能的电动机。在允许的范围之内，只需要调整电压，即可调整它的转速，因此这类电机拥有良好的调速性能。直流有刷电机实物图如图 3.2.1.1 所示：

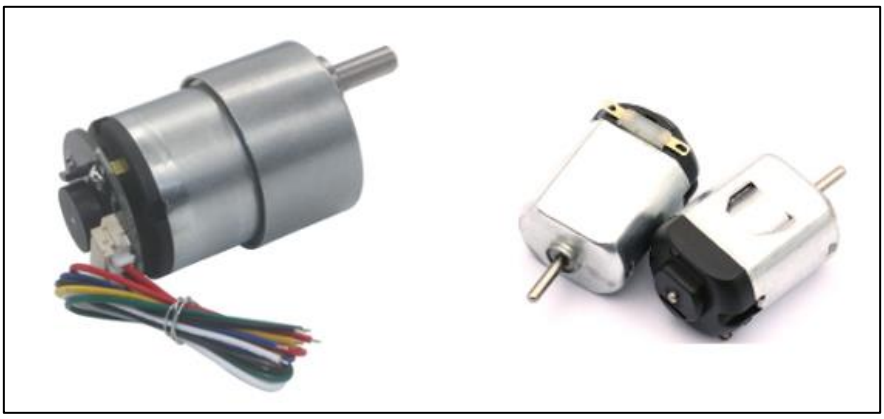


图 3.2.1.1 直流有刷电机

3.2.2 直流无刷电机

无刷电机是指无电刷和换向器（或集电环）的电机，又称无换向器电机，它由电动机主体和驱动器组成，是一种典型的机电一体化产品。早在十九纪，就已经出现了无刷形式的电机，即交流鼠笼式异步电动机，但是，其有着许多无法克服的缺陷，以至于人们不断地探索更好的电机技术。直到上世纪中叶，晶体管诞生了，晶体管换向电路代替了传统的电刷与换向器结构，这种新型无刷电机克服了第一代无刷电机的缺陷。直流无刷电机实物图如图 3.2.2.1 所示：

图 3.2.2.1 直流无刷电机

3.2.3 舵机

舵机是一种“特殊”的直流电机，其构造不仅只有电机部分，而且还自带一个控制器，所以舵机本身就是一个闭环的控制系统（伺服系统）。当我们给舵机发送指令信号时，它可以将指令信号变换为输出轴的角度并保持住，即便是负载发生了变化，它也能够进行自动调节。舵机的实物图如图 3.2.3.1 所示：

图 3.2.3.1 SG90 舵机

3.2.4 步进电机

步进电机是一种将电脉冲信号转变为角位移或线位移的开环控制电机，是现代数字程序控制系统中的主要执行元件，应用极为广泛。在非超载的情况下，步进电机的转速、停止的位置只取决于脉冲信号的频率和脉冲个数，不受负载变化的影响。步进电机的实物图如图 3.2.4.1 所示：

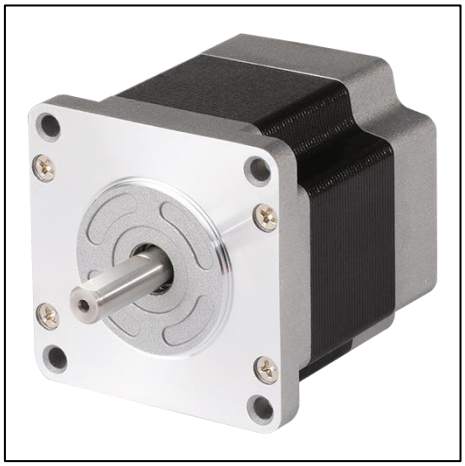


图 3.2.4.1 步进电机

3.2.5 伺服电机

伺服电机是指在伺服系统中控制机械元件运转的电动机，它搭配上驱动器，形成一个完整的伺服系统，就能够精准地控制速度、位置等。伺服电机具有线性度高、起动转矩大、运行范围较广、无自转现象等显著特点，遗憾的是，就目前来说，一整套伺服系统的成本是相对较高的。伺服电机和控制器的实物图如图 3.2.5.1 所示：

图 3.2.5.1 伺服电机及驱动器

3.3 各类电机的特点

下面我们总结一下几种常见电机的特点，具体见表 3.3.1：

电机类别	特点
直流有刷电机	驱动简单，成本低廉，但是噪音大，故障较多，寿命也相对较短。
直流无刷电机	噪音低，运转流畅，高速，寿命更长，但是控制较为复杂。

舵机	自成闭环控制系统，角度控制优秀，价格低，但是带负载的能力较弱。
步进电机	结构简单，驱动方便，没有积累误差，不过因为其可能存在丢步现象，常用于对精度要求不高的场合。
伺服电机	速度、位置控制精确，可低速运行，效率高，寿命长，可以说是目前最为可靠的一类电机。但是价格较高。

表 3.3.1 常见电机的特点

3.4 电机常用名词

为了让大家可以更好地学习电机控制，搭建更健全的电机知识架构，下面我们先学习一些电机常用名词：

① 转速：电机输出轴的速度，一般单位是 r/min（转每分钟），也常用 RPM 表示。
② 输出扭矩：表示了电机输出轴可以输出力的大小，单位一般为 N·m 或者 kg·cm。
③ 工作电压：电机正常工作的电压。
④ 额定电流：电机正常工作时的电流。
⑤ 堵转电流：电机负载过大，驱动力不足，发生堵转时的电流。

第 4 章 电机控制相关的 STM32 外设

本章我们主要来学习电机控制相关的 STM32 外设，包括基本定时器、高级定时器、ADC、DAC 等，这一章节内容是电机控制的基础，大家一定要掌握。

本章分为如下几个小节：

4.1 电机控制与 STM32 定时器的关系
4.2 基本定时器
4.3 高级定时器
4.4 ADC
4.5 DAC

4.1 电机控制与 STM32 定时器的关系

电机的控制与 STM32 定时器有着密不可分的关系，举两个例子：1、在直流有刷电机的控制中，我们常用脉冲宽度调制技术（PWM）来控制电压的大小，以此改变直流有刷电机的转速。2、对于步进电机而言，接收的脉冲个数决定了它的旋转位置，脉冲频率决定了它的旋转速度。

从上面的例子中可以得知，电机的控制本质上就是**脉冲的控制**，因此，我们引入了 STM32 定时器，以便对脉冲信号实现更高效的控制。

接下来，我们就开始学习 STM32 定时器的功能，包括定时器中断、PWM 输出、互补输出带死区刹车、输入捕获，等等。

4.2 基本定时器

STM32F407 有众多的定时器，其中包括 2 个基本定时器（TIM6 和 TIM7）、10 个通用定时器（TIM2~TIM5，TIM9~ TIM14）、2 个高级控制定时器（TIM1 和 TIM8），这些定时器彼此完全独立，不共享任何资源。本小节我们学习如何使用 STM32F407 的基本定时器中断。我们将使用 TIM6 的定时器中断来控制 LED1 的翻转，在主函数用 LED0 的翻转来提示程序正在运行。

4.2.1 基本定时器简介

STM32F407 有两个基本定时器 TIM6 和 TIM7，它们的功能完全相同，资源是完全独立的，可以同时使用。其主要特性如下：16 位自动重载递增计数器，16 位可编程预分频器，预分频系数 1~65536，用于对计数器时钟频率进行分频，还可以触发 DAC 的同步电路，以及生成中断/DMA 请求。

4.2.1.1 基本定时器框图

下面先来学习基本定时器框图，通过学习基本定时器框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。

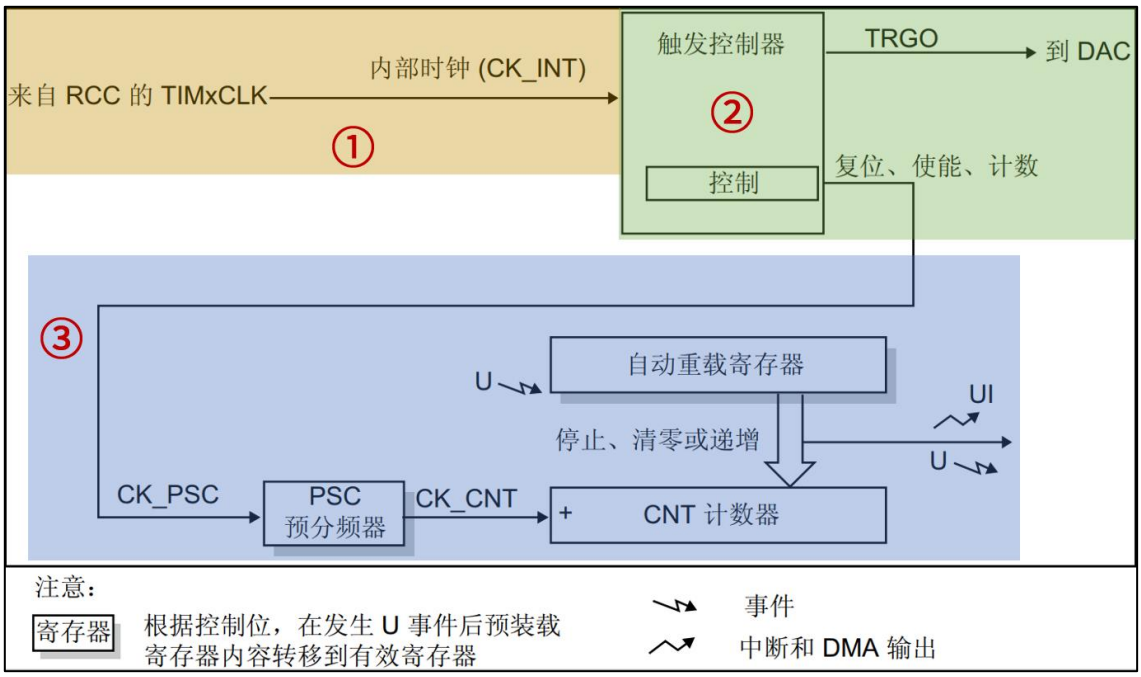


图 4.2.1.1.1 基本定时器框图

① 时钟源

定时器的核心就是计算器，要实现计数功能，首先要给它一个时钟源。基本定时器时钟挂载在 APB1 总线，所以它的时钟来自于 APB1 总线，但是基本定时器时钟不是直接由 APB1 总线直接提供，而是先经过一个倍频器。当 APB1 的预分频器系数为 1 时，这个倍频器系数为 1，即定时器的时钟频率等于 APB1 的频率；当 APB1 的预分频器系数≥2 分频时，这个倍频器系数就为 2，即定时器的时钟频率等于 APB1 频率的两倍。我们在 sys_stm32_clock_init 时钟设置函数已经设置 APB1 总线时钟频率为 42M,，APB1 总线的预分频器分频系数是 4，所以挂载在 APB1 总线的定时器时钟频率为 84Mhz。

② 控制器

控制器除了控制定时器复位、使能、计数等功能之外，还可以用于触发 DAC 转换。

③ 时基单元

时基单元包括：计数器寄存器(TIMx_CNT)、预分频器寄存器(TIMx_PSC)、自动重载寄存器(TIMx_ARR) 。基本定时器的这三个寄存器都是 16 位有效数字，即可设置值范围是 0~65535。

时基单元中的预分频器 PSC，它有一个输入和一个输出。输入 CK_PSC 来源于控制器部分，实际上就是来自于内部时钟（CK_INT），即 2 倍的 APB1 总线时钟频率（84MHz）。输出 CK_CNT 是分频后的时钟，它是计数器实际的计数时钟，通过设置预分频器寄存器(TIMx_PSC)的值可以得到不同频率 CK_CNT，计算公式如下：

$$f_{CK_CNT} = f_{CK_PSC} / (PSC[15:0]+1)$$

上式中，PSC[15:0]是写入预分频器寄存器(TIMx_PSC)的值。

另外：预分频器寄存器(TIMx_PSC)可以在运行过程中修改它的数值，新的预分频数值将在下一个更新事件时起作用。因为更新事件发生时，会把 TIMx_PSC 寄存器值更新到其影子寄存器中，这才会起作用。

什么是影子寄存器？从框图上看，可以看到图 4.2.1.1.1 中的预分频器 PSC 后面有一个影子，自动重载寄存器也有个影子，这就表示这些寄存器有影子寄存器。影子寄存器是一个实际起作用的寄存器，不可直接访问。举个例子：我们可以把预分频系数写入预分频器寄存器(TIMx_PSC)，但是预分频器寄存器只是起到缓存数据的作用，只有等到更新事件发生时，预分频器寄存器的值才会被自动写入其影子寄存器中，这时才真正起作用。

自动重载寄存器及其影子寄存器的作用和上述同理。不同点在于自动重载寄存器是否具有缓冲作用还受到 ARPE 位的控制，当该位置 0 时，ARR 寄存器不进行缓冲，我们写入新的 ARR 值时，该值会马上被写入 ARR 影子寄存器中，从而直接生效；当该位置 1 时，ARR 寄

存器进行缓冲，我们写入新的 ARR 值时，该值不会马上被写入 ARR 影子寄存器中，而是要等到更新事件发生才会被写入 ARR 影子寄存器，这时才生效。预分频器寄存器则没有这样相关的控制位，这就是它们不同点。

值得注意的是，更新事件的产生有两种情况，一是由软件产生，将 TIMx_EGR 寄存器的位 UG 置 1，产生更新事件后，硬件会自动将 UG 位清零。二是由硬件产生，满足以下条件即可：计数器的值等于自动重装载寄存器影子寄存器的值。下面来讨论一下硬件更新事件。

基本定时器的计数器（CNT）是一个递增的计数器，当寄存器（TIMx_CR1）的 CEN 位置 1，即使能定时器，每来一个 CK_CNT 脉冲，TIMx_CNT 的值就会递增加 1。当 TIMx_CNT 值与 TIMx_ARR 的设定值相等时，TIMx_CNT 的值就会被自动清零并且会生成更新事件（如果开启相应的功能，就会产生 DMA 请求、产生中断信号或者触发 DAC 同步电路），然后下一个 CK_CNT 脉冲到来，TIMx_CNT 的值就会递增加 1，如此循环。在此过程中，TIMx_CNT 等于 TIMx_ARR 时，我们称之为定时器溢出，因为是递增计数，故而又称为定时器上溢。定时器溢出就伴随着更新事件的发生。

由上述可知，我们只要设置预分频寄存器和自动重载寄存器的值就可以控制定时器更新事件发生的时间。自动重载寄存器(TIMx_ARR)是用于存放一个与计数器作比较的值，当计数器的值等于自动重载寄存器的值时就会生成更新事件，硬件自动置位相关更新事件的标志位，如：更新中断标志位。

下面举个例子来学习如何设置预分频寄存器和自动重载寄存器的值来得到我们想要的定时器上溢事件发生的时间周期。比如我们需要一个 500ms 周期的定时器中断，一般思路是先设置预分频寄存器，然后才是自动重载寄存器。考虑到我们设置的 CK_INT 为 84MHZ，我们把预分频系数设置为 8400，即写入预分频寄存器的值为 8399，那么 f_{CK_CNT}=84MHZ/8400=10KHZ。这样就得到计数器的计数频率为 10KHZ，即计数器 1 秒钟可以计 10000 个数。我们需要 500ms 的中断周期，所以我们让计数器计数 5000 个数就能满足要求，即需要设置自动重载寄存器的值为 4999，另外还要把定时器更新中断使能位 UIE 置 1，CEN 位也要置 1。

4.2.1.2 TIM6/TIM7 寄存器

下面介绍 TIM6/TIM7 的几个重要的寄存器，具体如下：

- **控制寄存器 1（TIMx_CR1）**

TIM6/TIM7 的控制寄存器 1 描述如图 4.2.1.2.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								ARPE	Reserved			OPM	URS	UDIS	CEN
								rw				rw	rw	rw	rw

位 7 **ARPE**：自动重载预装载使能 (Auto-reload preload enable)

0：TIMx_ARR 寄存器不进行缓冲。

位 0 **CEN**：计数器使能 (Counter enable)

0：禁止计数器
1：使能计数器

注意：只有事先通过软件将 CEN 位置 1，才可以使用门控模式。而触发模式可通过硬件自动将 CEN 位置 1。

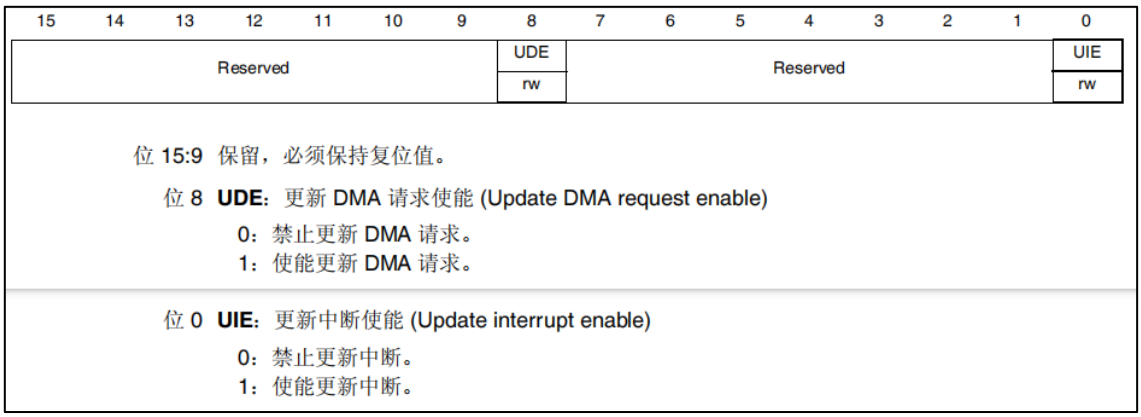
在单脉冲模式下，当发生更新事件时会自动将 CEN 位清零。

图 4.2.1.2.1 TIMx_CR1 寄存器

该寄存器，我们需要注意的是：位 0（CEN）用于使能或者禁止计数器，该位置 1 计数器开始工作，置 0 则停止。还有位 7（APRE）用于控制自动重载寄存器 ARR 是否具有缓冲作用，如果 ARPE 位置 1，ARR 起缓冲作用，即只有在更新事件发生时才会把 ARR 的值写入其影子寄存器里；如果 ARPE 位置 0，那么修改自动重载寄存器的值时，该值会马上被写入其影子寄存器中，从而立即生效。

- **DMA/中断使能寄存器（TIMx_DIER）**

TIM6/TIM7 的 DMA/中断使能寄存器描述如图 4.2.1.2.2 所示：



位 15:9 保留，必须保持复位值。

位 8 **UDE**：更新 DMA 请求使能 (Update DMA request enable)

0：禁止更新 DMA 请求。

1：使能更新 DMA 请求。

位 0 **UIE**：更新中断使能 (Update interrupt enable)

0：禁止更新中断。

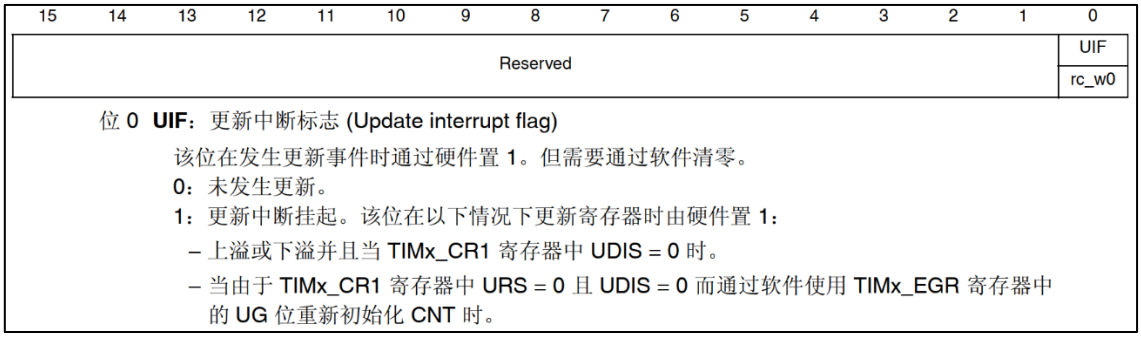
1：使能更新中断。

图 4.2.1.2.2 TIMx_DIER 寄存器

该寄存器位 0（UIE）用于使能或者禁止更新中断，因为本实验用到了定时器的更新中断，所以配置时需要把该位置 1。位 8（UDE）用于使能或者禁止更新 DMA 请求，本实验暂不使用。

- **状态寄存器（TIMx_SR）**

TIM6/TIM7 的状态寄存器描述如图 4.2.1.2.3 所示：



位 0 **UIF**：更新中断标志 (Update interrupt flag)

该位在发生更新事件时通过硬件置 1。但需要通过软件清零。

0：未发生更新。

1：更新中断挂起。该位在以下情况下更新寄存器时由硬件置 1：

– 上溢或下溢并且当 TIMx_CR1 寄存器中 UDIS = 0 时。

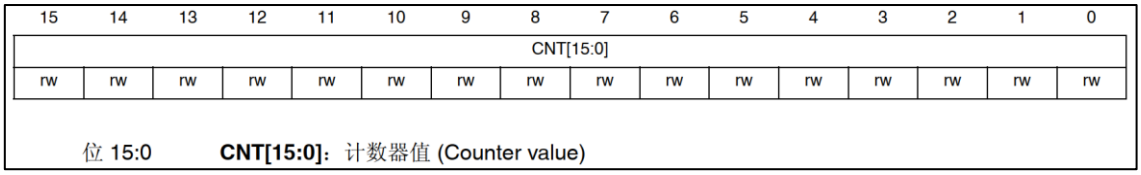
– 当由于 TIMx_CR1 寄存器中 URS = 0 且 UDIS = 0 而通过软件使用 TIMx_EGR 寄存器中的 UG 位重新初始化 CNT 时。

图 4.2.1.2.3 TIMx_SR 寄存器

该寄存器位 0（UIF）是中断更新的标志位，当发生中断时由硬件置 1，然后就会执行到中断服务函数，但是需要软件去清零，所以我们必须在中断服务函数里把该位清零。如果中断到来后，不清零该位，那么系统就会一直进入中断服务函数，这显然不是我们想要的。

- **计数器寄存器（TIMx_CNT）**

TIM6/TIM7 的计数器寄存器描述如图 4.2.1.2.4 所示：



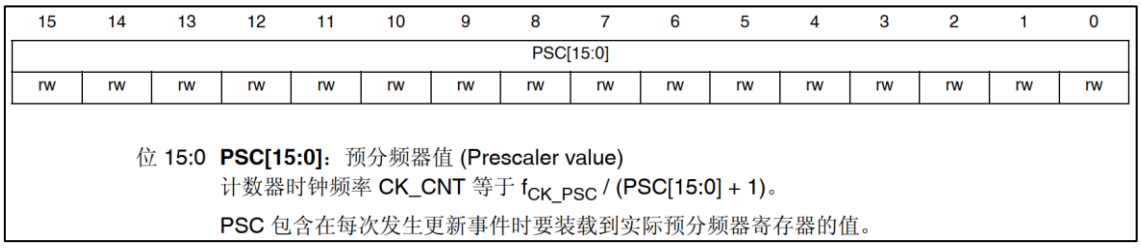
位 15:0 **CNT[15:0]**：计数器值 (Counter value)

图 4.2.1.2.4 TIMx_CNT 寄存器

该寄存器位[15:0]就是计数器的实时的计数值。

- **预分频寄存器（TIMx_PSC）**

TIM6/TIM7 的预分频寄存器描述如图 4.2.1.2.5 所示：



位 15:0 **PSC[15:0]**：预分频器值 (Prescaler value)

计数器时钟频率 CK_CNT 等于 f_{CK_PSC} / (PSC[15:0] + 1)。

PSC 包含在每次发生更新事件时要装载到实际预分频器寄存器的值。

图 4.2.1.2.5 TIMx_PSC 寄存器

该寄存器是 TIM6/TIM7 的预分频寄存器，比如我们要 8400 分频，就往该寄存器写入 8399。

注意这是 16 位的寄存器，写入的数值范围是 0 到 65535，分频系数范围：1 到 65536。

- **自动重载寄存器（TIMx_ARR）**

TIM6/TIM7 的自动重载寄存器描述如图 4.2.1.2.6 所示：

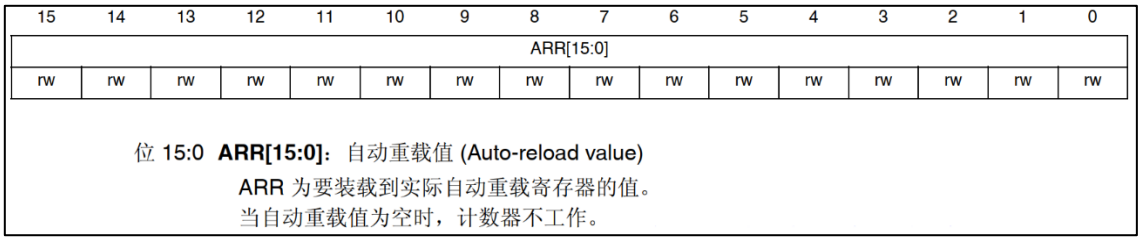


图 4.2.1.2.6 TIMx_ARR 寄存器

该寄存器可以由 APRE 位设置是否进行缓冲。计数器的值会和 ARR 寄存器影子寄存器进行比较，当两者相等，定时器就会溢出，从而发生更新事件，如果打开更新中断，还会发生更新中断。

4.2.1.3 基本定时器中断应用

本实验，我们主要配置定时器产生周期性溢出，从而在定时器更新中断中做周期性的操作，如周期性翻转 LED 灯。假设计数器计数模式为递增计数模式，那么实现周期性更新中断原理示意图如下所示：

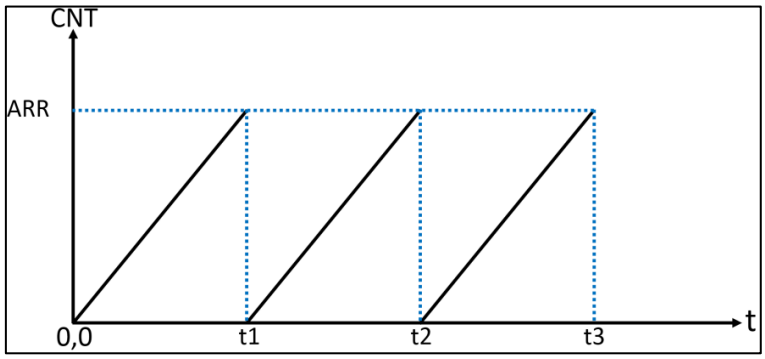


图 4.2.1.3.1 基本定时器中断示意图

如图 4.2.1.3.1 所示，CNT 计数器从 0 开始计数，当 CNT 的值和 ARR 相等时（t1），产生一个更新中断，然后 CNT 复位（清零），然后继续递增计数，依次循环。图中的 t1、t2、t3 就是定时器更新中断产生的时刻。

通过修改 ARR 的值，可以改变定时时间。另外，通过修改 PSC 的值，使用不同的计数频率（改变图中 CNT 的斜率），也可以改变定时时间。

4.2.2 硬件设计

1. 例程功能

LED0 用来指示程序运行，每 200ms 翻转一次。我们在更新中断中，将 LED1 的状态取反。LED1 用于指示定时器发生更新事件的频率，500ms 取反一次。

2. 硬件资源

1）LED 灯

LED0 - PE0

LED1 - PE1

2）定时器 6

3. 原理图

定时器属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们通过 LED1 来指示 STM32F407 的定时器进入中断情况。

4.2.3 程序设计

4.2.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在 stm32f4xx_hal_tim.c 和 stm32f4xx_hal_tim_ex.c 文件（及它们的头文件）中。

1. HAL_TIM_Base_Init 函数

定时器的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef *htim);
```

- **函数描述：**

  用于初始化定时器。

- **函数形参：**

  形参 1 是 TIM_HandleTypeDef 类型结构体指针变量，其定义如下：

```
typedef struct
{
  TIM_TypeDef                 *Instance;                /* 外设寄存器基地址 */
  TIM_Base_InitTypeDef         Init;                    /* 定时器初始化结构体*/
  HAL_TIM_ActiveChannel        Channel;                 /* 定时器通道 */
  DMA_HandleTypeDef           *hdma[7];                 /* DMA 管理结构体 */
  HAL_LockTypeDef              Lock;                    /* 锁定资源 */
  __IO HAL_TIM_StateTypeDef    State;                   /* 定时器状态 */
  __IO HAL_TIM_ChannelStateTypeDef    ChannelState[4];  /* 定时器通道状态 */
  __IO HAL_TIM_ChannelStateTypeDef    ChannelNState[4]; /* 定时器互补通道状态 */
  __IO HAL_TIM_DMABurstStateTypeDef   DMABurstState;    /* DMA 溢出状态 */
}TIM_HandleTypeDef;
```

1）Instance：指向定时器寄存器基地址。
2）Init：定时器初始化结构体，用于配置定时器的相关参数。
3）Channel：定时器的通道选择，基本定时器没有该功能。
4）hdma[7]：用于配置定时器的 DMA 请求。
5）Lock：ADC 锁资源。
6）State：定时器工作状态。
7）ChannelState：定时器通道状态。
8）ChannelNState：定时器互补通道状态。
9）DMABurstState：DMA 溢出状态。

我们主要看 TIM_Base_InitTypeDef 这个结构体类型定义：

```
typedef struct
{
  uint32_t Prescaler;           /* 预分频系数 */
  uint32_t CounterMode;         /* 计数模式 */
  uint32_t Period;              /* 自动重载值 ARR */
  uint32_t ClockDivision;       /* 时钟分频因子 */
  uint32_t RepetitionCounter;   /* 重复计数器 */
  uint32_t AutoReloadPreload;   /* 自动重载预装载使能 */
} TIM_Base_InitTypeDef;
```

1）Prescaler：预分频系数，即写入预分频寄存器的值，范围 0 到 65535。
2）CounterMode：计数器计数模式，这里基本定时器只能向上计数。
3）Period：自动重载值，即写入自动重载寄存器的值，范围 0 到 65535。
4）ClockDivision：时钟分频因子，也就是定时器时钟频率 CK_INT 与数字滤波器所使用的采样时钟之间的分频比，基本定时器没有此功能。

5）RepetitionCounter：设置重复计数器寄存器的值，用在高级定时器中。

6）AutoReloadPreload：自动重载预装载使能，即控制寄存器 1 (TIMx_CR1)的 ARPE 位。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

2. HAL_TIM_Base_Start_IT 函数

HAL_TIM_Base_Start_IT 函数是更新定时器中断和使能定时器的函数。其声明如下：

```
HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim);
```

- **函数描述：**

  该函数调用了__HAL_TIM_ENABLE_IT 和__HAL_TIM_ENABLE 两个函数宏定义，分别是更新定时器中断和使能定时器的宏定义。

- **函数形参：**

  形参 1 是 TIM_HandleTypeDef 结构体指针类型的定时器句柄。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

- **注意事项：**

  下面分别列出单独使能/关闭定时器中断和使能/关闭定时器方法：

```
__HAL_TIM_ENABLE_IT(htim, TIM_IT_UPDATE);      /* 使能句柄指定的定时器更新中断 */
__HAL_TIM_DISABLE_IT (htim, TIM_IT_UPDATE);    /* 关闭句柄指定的定时器更新中断 */
__HAL_TIM_ENABLE(htim);                        /* 使能句柄 htim 指定的定时器 */
__HAL_TIM_DISABLE(htim);                       /* 关闭句柄 htim 指定的定时器 */
```

定时器中断配置步骤

1）使能定时器时钟

HAL 中定时器使能是通过宏定义标识符来实现对相关寄存器操作的，方法如下：

```
__HAL_RCC_TIMx_CLK_ENABLE();                   /* x=1~14 */
```

2）初始化定时器参数，设置自动重装值，分频系数，计数方式等

定时器的初始化参数是通过定时器初始化函数 HAL_TIM_Base_Init 实现的。

注意：该函数会调用：HAL_TIM_Base_MspInit 函数，我们可以通过后者存放定时器时钟和中断等初始化的代码。

3）使能定时器更新中断，开启定时器计数，配置定时器中断优先级

通过 HAL_TIM_Base_Start_IT 函数使能定时器更新中断和开启定时器计数。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断，通过 HAL_NVIC_SetPriority 函数设置中断优先级。

4）编写中断服务函数

定时器中断服务函数为：TIMx_IRQHandler，当发生中断的时候，程序就会执行中断服务函数。HAL 库为了使用方便，提供了一个定时器中断通用处理函数 HAL_TIM_IRQHandler，该函数又会调用 HAL_TIM_PeriodElapsedCallback 等一些回调函数，需要用户根据中断类型选择重定义对应的中断回调函数来处理中断程序。

4.2.3.2 程序流程图

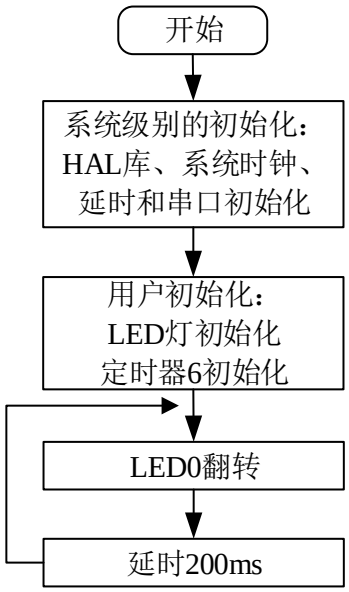


图 4.2.3.2.1 基本定时器中断实验程序流程图

4.2.3.3 程序解析

1. 基本定时器中断驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。基本定时器驱动代码包括两个文件：btim.c 和 btim.h。

首先看 btim.h 头文件的几个宏定义：

```
/* 基本定时器 定义 */
/* TIMX 中断定义
 * 默认是针对 TIM6/TIM7.
 * 注意：通过修改这 4 个宏定义,可以支持 TIM1~TIM8 任意一个定时器.
 */
#define BTIM_TIMX_INT                TIM6
#define BTIM_TIMX_INT_IRQn           TIM6_DAC_IRQn
#define BTIM_TIMX_INT_IRQHandler     TIM6_DAC_IRQHandler
/* TIM6 时钟使能 */
#define BTIM_TIMX_INT_CLK_ENABLE()  do{ __HAL_RCC_TIM6_CLK_ENABLE(); }while(0)
```

通过修改这 4 个宏定义，可以支持 TIM1~TIM8 任意一个定时器。

下面我们解析 btim.c 的程序，先看定时器的初始化函数，其定义如下：

```
/**
 * @brief       基本定时器 TIMX 定时中断初始化函数
 * @note
 *              基本定时器的时钟来自 APB1,当 PPRE1 ≥ 2 分频的时候
 *              基本定时器的时钟为 APB1 时钟的 2 倍, 而 APB1 为 42M, 所以定时器时钟 = 84Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值。
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void btim_timx_int_init(uint16_t arr, uint16_t psc)
{
    timx_handler.Instance = BTIM_TIMX_INT;                      /* 定时器 X */
    timx_handler.Init.Prescaler = psc;                          /* 设置预分频器 */
```

```
    timx_handler.Init.CounterMode = TIM_COUNTERMODE_UP;     /* 向上计数器 */
    timx_handler.Init.Period = arr;                         /* 自动装载值 */
    HAL_TIM_Base_Init(&timx_handler);

    HAL_TIM_Base_Start_IT(&timx_handler);     /* 使能定时器 x 和定时器 x 更新中断 */
}
```

btim_timx_int_init 函数用来初始化定时器，我们可以通过修改宏定义 BTIM_TIMX_INT 来初始化 TIM1~TIM14 中的任意一个，本章我们是初始化基本定时器 6。该函数的 2 个形参：arr 设置自动重载寄存器(TIMx_ARR)，psc 设置预分频器寄存器(TIMx_PSC)。HAL_TIM_Base_Init 函数初始化定时器后，再调用 HAL_TIM_Base_Start_IT 函数使能定时器和更新定时器中断。

因为我们在 sys_stm32_clock_init 函数里面已经初始化 APB1 的时钟为 HCLK 的 4 分频，所以 APB1 的时钟为 42M，而从 STM32F407 的内部时钟树图得知：当 APB1 的时钟分频数为 1 的时候，TIM2~7 的时钟为 APB1 的时钟，而如果 APB1 的时钟分频数不为 1，那么 TIM2~7 的时钟频率将为 APB1 时钟的两倍。因此，TIM6 的时钟为 84M，再根据我们设计的 arr 和 psc 的值，就可以计算中断时间了。计算公式如下：

$$Tout= ((arr+1)*(psc+1))/Tclk$$

其中：

Tout：定时器溢出时间（单位为 s）。

Tclk：定时器的时钟源频率（单位为 MHz）。

arr：自动重装寄存器（TIMx_ARR）的值。

psc：预分频器寄存器（TIMx_PSC）的值

定时器底层驱动初始化函数如下：

```
/**
 * @brief       定时器底册驱动，开启时钟，设置中断优先级
                此函数会被 HAL_TIM_Base_Init()函数调用
 * @param       htim:定时器句柄
 * @retval      无
 */
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == BTIM_TIMX_INT)
    {
        BTIM_TIMX_INT_CLK_ENABLE();                         /* 使能 TIMx 时钟 */
        /* 设置中断优先级，抢占优先级 1，子优先级 3 */
        HAL_NVIC_SetPriority(BTIM_TIMX_INT_IRQn, 1, 3);
        HAL_NVIC_EnableIRQ(BTIM_TIMX_INT_IRQn);             /* 开启 ITMx 中断 */
    }
}
```

HAL_TIM_Base_MspInit 函数用于存放 GPIO、NVIC 和时钟相关的代码，这里首先判断定时器的寄存器基地址，满足条件后，首先设置使能定时器的时钟，然后设置定时器中断的抢占优先级为 1，响应优先级为 3，最后开启定时器中断。这里没有用到 IO 引脚，所以不用初始化 GPIO。

接着是定时器中断服务函数的定义，这里用的是宏名，其定义如下：

```
/**
 * @brief       定时器中断服务函数
 * @param       无
 * @retval      无
 */
void BTIM_TIMX_INT_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&timx_handle);
}
```

这个函数实际上调用 HAL 库的定时器中断公共处理函数 HAL_TIM_IRQHandler。HAL 库的中断公共处理函数，会根据中断标志位调用各个中断回调函数，中断要处理的逻辑代码就写到这个回调函数中。比如这里我们使用到的是更新中断，定义的更新中断回调函数如下：

```
/**
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == BTIM_TIMX_INT)
    {
        LED1_TOGGLE();
    }
}
```

更新中断回调函数是所有定时器公用的，所以我们就需要在更新中断回调函数中对定时器寄存器基地址进行判断，只有符合对应定时器发生的更新中断，才能进行相应的处理，从而避免多个定时器同时使用到更新中断，导致更新中断代码的逻辑混乱。这里我们使用定时器 6 的更新中断，所以进入更新中断回调函数后，先判断是不是定时器 6 的寄存器基地址，当然这里使用宏的形式，BTIM_TIMX_INT 的原型就是 TIM6，执行的逻辑代码是翻转 LED1。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    btim_timx_int_init(5000 - 1, 8400 - 1);/* 10Khz 计数频率，计数 5K 次为 500ms */

    while (1)
    {
        LED0_TOGGLE();                          /*LED0(红灯) 翻转*/
        delay_ms(200);
    }
}
```

在 main 函数里，先初始化系统和用户的外设代码，然后在 wilhe(1)里每 200ms 翻转一次 LED0。由前面的内容知道，定时器 6 的时钟频率为 84MHZ，而调用 btim_timx_int_init 初始化函数之后，就相当于写入预分频寄存器的值为 8399，写入自动重载寄存器的值为 4999。由公式得：

Tout= ((4999+1)*(8399+1))/84MHZ=0.5s=500ms

4.2.4 下载验证

下载代码后，可以看到 LED0 不停闪烁（每 200ms 翻转一次），而 LED1 也是不停的闪烁，但是闪烁时间较 LED0 慢（500ms 翻转一次）。

4.3 高级定时器

由于 STM32 的高级定时器包含了通用定时器的所有功能，所以我们跳过通用定时器部分，直接来学习高级定时器，大家只要学会了高级定时器的使用，通用定时器也就迎刃而解了。

STM32F407 有 2 个高级定时器（TIM1 和 TIM8）。这些定时器彼此完全独立，不共享任何资源。本小节我们将通过 7 个实验来学习高级定时器的各个功能，分别是高级定时器中断实验、高级定时器 PWM 输出实验、高级定时器互补输出带死区刹车实验、高级定时器输入捕获实验、高级定时器 PWM 输入实验、高级定时器输出比较实验和高级定时器输出指定个数 PWM 实验。

本小节分为如下几部分：

4.3.1 高级定时器简介
4.3.2 高级定时器中断实验
4.3.3 高级定时器 PWM 输出实验
4.3.4 高级定时器互补输出带死区刹车控制实验
4.3.5 高级定时器输入捕获实验
4.3.6 高级定时器 PWM 输入实验
4.3.7 高级定时器输出比较模式实验
4.3.8 高级定时器输出指定个数 PWM 实验

4.3.1 高级定时器简介

STM32F407 的定时器有 14 个之多，其基本特性不尽相同，为了更好地区别各个定时器的特性，我们列了一个表格，如下所示：

定时器类型	定时器	计数器	计数模式	预分频系数（整数）	DMA 请求	捕获/比较通道	互补输出
基本定时器	TIM6 TIM7	16 位	递增	1~65536	可以	0	无
通用定时器	TIM2 TIM5	32 位	递增、递减、中央对齐	1~65536	可以	4	无
	TIM3 TIM4	16 位	递增、递减、中央对齐	1~65536	可以	4	无
	TIM9 TIM12	16 位	递增	1~65536	不可以	2	无
	TIM10 TIM11 TIM13 TIM14	16 位	递增	1~65536	可以	1	无
高级定时器	TIM1 TIM8	16 位	递增、递减、中央对齐	1~65536	可以	4	有

表 4.3.1.1 定时器基本特性表

注：该表参考数据手册 《STM32F407IGT6.pdf》的 2.2.21 小节（第 30 页）

由上表知道：该 STM32 芯片的 TIM2 和 TIM5 计数器是 32 位的，其他的都是 16 位的。通用定时器和高级定时器其实也就是在基本定时器的基础上，添加了一些其他功能，如：输入捕获、输出比较、输出 PWM 和单脉冲模式等，其特性有一些的差异，但是基本原理都一样。互补输出则是高级定时器特有的功能。

高级定时器的框图

下面先来学习高级定时器框图，通过学习高级定时器框图会有一个很好的整体掌握，同时对之后的编程也会有一个清晰的思路。

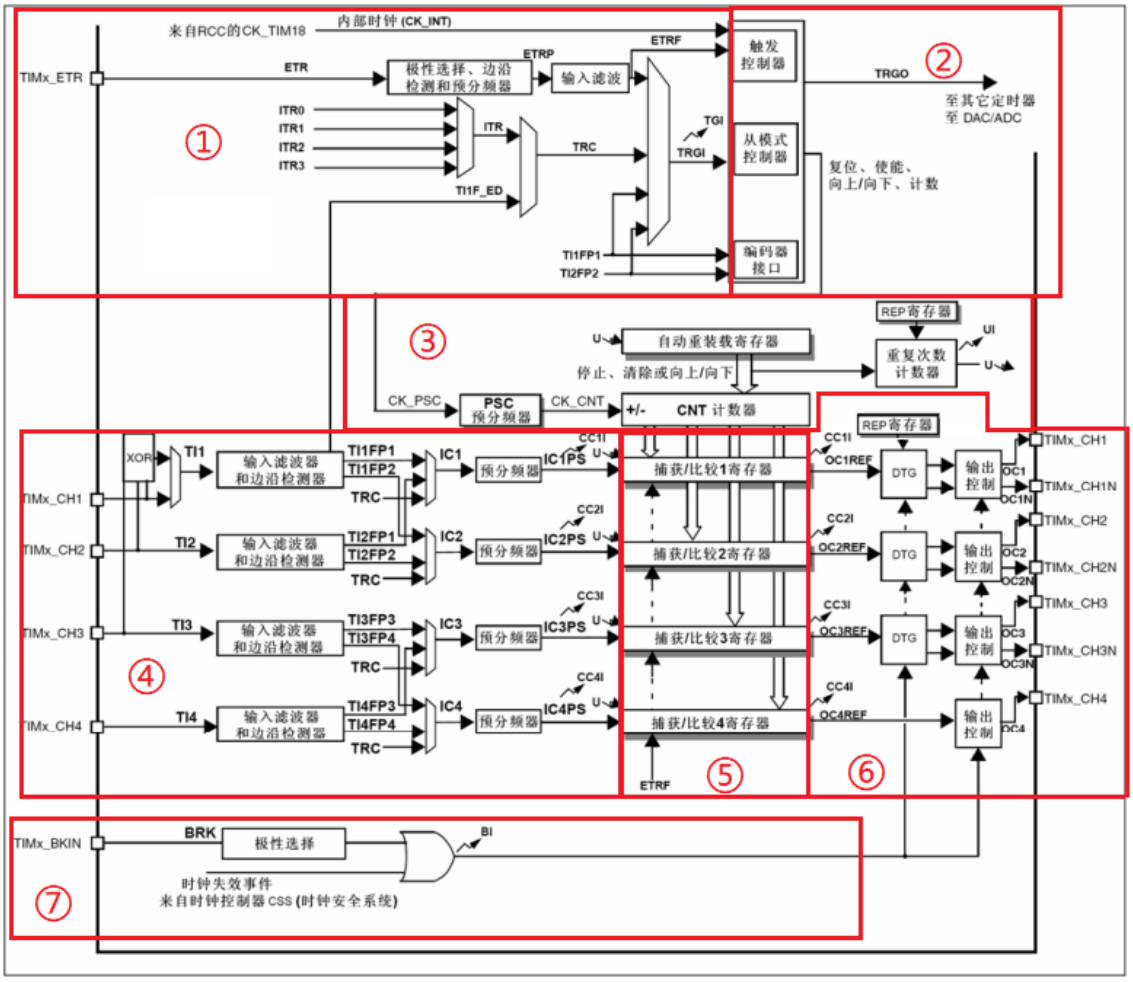


图 4.3.1.1 高级定时器框图

如上图，高级定时器的框图比基本定时器的框图复杂许多，为了方便介绍，我们将其分成七个部分讲解：

① 时钟源

高级定时器时钟可由下列的时钟源提供：

1）内部时钟 (CK_INT)

2）外部时钟模式 1：外部输入引脚 (TIx)，x=1，2，3，4

3）外部时钟模式 2：外部触发输入 (ETR)

4）内部触发输入 (ITRx)：使用一个定时器作为另一定时器的预分频器

高级定时器时钟源的设置方法如下表所示：

定时器时钟类型	设置方法
内部时钟(CK_INT)	设置 TIMx_SMCR 的 SMS=000
外部时钟模式 1：外部输入引脚(TIx)	设置 TIMx_SMCR 的 SMS=111
外部时钟模式 2：外部触发输入(ETR)	设置 TIMx_SMCR 的 ECE=1
内部触发输入(ITRx)	设置可参考《STM32F4xx 参考手册_V4（中文版）.pdf》14.3.4 小节

表 4.3.1.2 高级定时器时钟源设置方法

②控制器

控制器包括：从模式控制器、编码器接口和触发控制器（TRGO）。从模式控制器可以控制计数器复位、启动、递增/递减、计数。编码器接口针对编码器计数。触发控制器用来提供触发信号给别的外设，比如为其他定时器提供时钟或者为 DAC/ADC 的触发转换提供信号。

③时基单元

高级定时器时基单元包括：计数器寄存器(TIMx_CNT)、预分频器寄存器(TIMx_PSC)、自

动重载寄存器(TIMx_ARR)、重复计数器寄存器（TIMx_RCR）。其中，前 3 个寄存器的内容和基本定时器是基本一样的，大家可以参考基本定时器的介绍。

下面我们来介绍一些不同的点：

1，高级定时器的计数模式有三种：递增计数模式、递减计数模式和中心对齐模式。

递增计数模式在讲解基本定时器的时候已经讲过了，那么对应到递减计数模式就很好理解了。就是来了一个计数脉冲，计数器就减 1，直到计数器寄存器的值减到 0，减到 0 时定时器溢出，由于是递减计数，故而称为定时器下溢，定时器溢出就会伴随着更新事件的发生。然后计数器又从自动重载寄存器影子寄存器的值开始继续递减计数，如此循环。最后是中心对齐模式，字面上不太好理解。该模式下，计数器先从 0 开始递增计数，直到计数器的值等于自动重载寄存器影子寄存器的值减 1 时，定时器上溢，同时生成更新事件，然后从自动重载寄存器影子寄存器的值开始递减计算，直到计数值等于 1 时，定时器下溢，同时生成更新事件，然后又从 0 开始递增计数，依此循环。每次定时器上溢或下溢都会生成更新事件。计数器的计数模式的设置请参考 TIMx_CR1 寄存器的位 CMS 和位 DIR。

下面通过一张图给大家展示定时器工作在不同计数模式下，更新事件发生的情况。

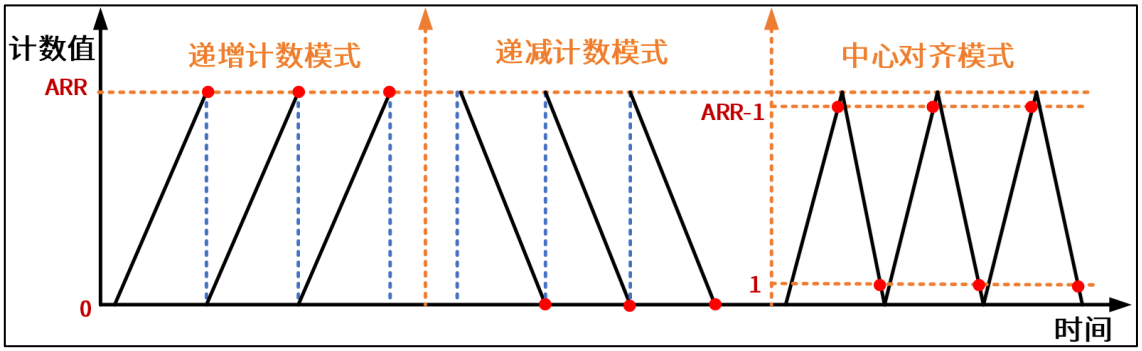


图 4.3.1.2 更新事件发生条件

上图中，纵轴表示计数器的计数值，横轴表示时间，ARR 表示自动重载寄存器的值，小红点就是更新事件发生的时间点。举个例子，递增计数模式下，当计数值等于 ARR 时，计数器的值被复位为 0，定时器溢出，并伴随着更新事件的发生，后面继续递增计数。递减计数模式和中心对齐模式请参考前面的描述。

上表的描述属于硬件更新事件发生条件，我们还可以通过 UG 位产生软件更新事件。

关于影子寄存器和定时器溢出时间计算公式等内容可以参考基本定时器的相关内容。

2，重复计数器寄存器（TIMx_RCR）是高级定时器特有的。

在 F4 系列中，高级定时器 TIM1 和 TIM8 都有重复计数器。下面来介绍一下重复计数器有什么作用？在学习基本定时器和通用定时器的时候，我们知道定时器发生上溢或者下溢时，会直接生成更新事件。但是有重复计数器的定时器并不完全是这样的，定时器每次发生上溢或下溢时，重复计数器的值会减一，当重复计数器的值为 0 时，再发生一次上溢或者下溢才会生成定时器更新事件。如果我们设置重复计数器寄存器 RCR 的值为 N，那么更新事件将在定时器发生 N+1 次上溢或下溢时发生。

这里需要注意的是重复计数器寄存器是具有影子寄存器的，所以 RCR 寄存器只是起缓冲的作用。RCR 寄存器的值会在更新事件发生时，被转移至其影子寄存器中，从而真正生效。

重复计数器的特性，在控制生成 PWM 信号时很有用，后面会有相应的实验。

④输入捕获

输入捕获包括：4 个输入捕获通道、输入滤波和边沿检测以及预分频器等部分。IO 端口通过复用功能与这些通道相连。配置好 IO 端口的复用功能后，将需要测量的信号输入到相应的 IO 端口，输入捕获部分可以对输入的信号的上升沿，下降沿或者双边沿进行捕获，常见的测量有：测量输入信号的脉冲宽度、测量 PWM 输入信号的频率和占空比等。

下面简单说一下测量高电平脉冲宽度的工作原理，方便大家的理解：一般先要设置输入捕获的边沿检测极性，如：我们设置上升沿检测，那么当检测到上升沿时，定时器会把计数器 CNT 的值锁存到相应的捕获/比较寄存器 TIMx_CCRy 里，y=1~4。然后我们再设置边沿检测为下降沿检测，当检测到下降沿时，定时器会把计数器 CNT 的值再次锁存到相应的捕获/比较

寄存器 TIMx_CCRy 里。最后，我们将前后两次锁存的 CNT 的值相减，就可以算出高电平脉冲期间内计数器的计数个数，再根据定时器的计数频率就可以计算出这个高电平脉冲的时间。如果要测量的高电平脉宽时间长度超过定时器的溢出时间周期，就会发生溢出，这时候我们还需要做定时器溢出的额外处理。低电平脉冲捕获同理。

⑤输入捕获和输出比较公用部分

该部分需要结合第④部分或者第⑥部分共同完成相应功能。

⑥输出比较

输出比较包括：4 个输出比较通道、3 个互补通道、死区发生器以及输出控制器，用于输出比较模式或 PWM 输出模式。

高级定时器输出比较部分和通用定时器相比，多了带死区控制的互补输出功能。图 4.3.1.1 第⑥部分的 TIMx_CH1N、TIMx_CH2N 和 TIMx_CH3N 分别是定时器通道 1、通道 2 和通道 3 的互补输出通道，通道 4 是没有互补输出通道的。DTG 是死区发生器，死区时间由 DTG[7:0]位来配置。如果不使用互补通道和死区时间控制，那么高级定时器 TIM1 和 TIM8 和通用定时器的输出比较部分使用方法基本一样，只是要注意 MOE 位得置 1 定时器才能输出。

如果使用互补通道，那么就有一定的区别了，具体我们在高级定时器互补输出带死区控制实验小节再来介绍。

⑦断路功能

断路功能也称刹车功能，一般用于电机控制的刹车。F4 系列有一个断路通道，断路源可以是刹车输入引脚（TIMx_BKIN），也可以是一个时钟失败事件。时钟失败事件由复位时钟控制器中的时钟安全系统产生。系统复位后，断路功能默认被禁止，MOE 位为低。

使能断路功能的方法：将 TIMx_BDTR 的位 BKE 置 1。断路输入引脚 TIMx_BKIN 的输入有效电平可通过 TIMx_BDTR 寄存器的位 BKP 设置。

使能刹车功能后：由 TIMx_BDTR 的 MOE、OSSI、OSSR 位，TIMx_CR2 的 OISx、OISxN 位，TIMx_CCER 的 CCxE、CCxNE 位控制 OCx 和 OCxN 输出状态。无论何时，OCx 和 OCxN 输出都不能同时处在有效电平。

当发生断路输入后，会怎么样？

1，MOE 位被异步地清零，OCx 和 OCxN 为无效、空闲或复位状态(由 OSSI 位选择)。

2，OCx 和 OCxN 的状态：由相关控制位状态决定，当使用互补输出时：根据情况自动控制输出电平，参考《STM32F4xx 参考手册_V4（中文版）.pdf》手册第 382 页的表 73 具有断路功能的互补通道 Ocx 和 OcxN 的控制位。

3，BIF 位置 1，如果使能了 BIE 位，还会产生刹车中断；如果使能了 TDE 位，会产生 DMA 请求。

4，如果 AOE 位置 1，在下一个更新事件 UEV 时，MOE 位被自动置 1。

高级定时器框图部分就简单介绍到这里，下面通过实际的实验来学习高级定时器。

4.3.2 高级定时器中断实验

本小节我们来学习使用高级定时器中断，以定时器 8 中断为例，首先来了解相关的寄存器。

4.3.2.1 TIM1/TIM8 寄存器

下面介绍 TIM1/TIM8 这些高级定时器中使用到的几个重要的寄存器，其他更多关于定时器的资料可以参考《STM32F4xx 参考手册_V4（中文版）.pdf》的第 14 章。

- **控制寄存器 1（TIMx_CR1）**

TIM1/TIM8 的控制寄存器 1 描述如图 4.3.2.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS[1:0]		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 7 **ARPE**：自动重载预装载使能 (Auto-reload preload enable)

0：TIMx_ARR 寄存器不进行缓冲
1：TIMx_ARR 寄存器进行缓冲

位 4 **DIR**：方向 (Direction)

0：计数器递增计数
1：计数器递减计数

注意：当定时器配置为中心对齐模式或编码器模式时，该位为只读状态。

位 0 **CEN**：计数器使能 (Counter enable)

0：禁止计数器
1：使能计数器

注意：只有事先通过软件将 CEN 位置 1，才可以使用外部时钟、门控模式和编码器模式。而触发模式可通过硬件自动将 CEN 位置 1。

图 4.3.2.1.1 TIMx_CR1 寄存器

上图中，我们只列出了本章需要用到的一些位，其中：位 7（APRE）用于控制自动重载寄存器的影子寄存器是否有效，在基本定时器的时候已经讲过，请回顾。

DIR 位，用于控制定时器的计数方向，我们需要向上计数模式，所以设置 DIR=0。

CEN 位，用于使能计数器的工作，必须要设置该位为 1，才可以开始计数。

- **从模式控制寄存器（TIMx_SMCR）**

TIM1/TIM8 的从模式控制寄存器描述如图 4.3.2.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			Res.	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw

位 2:0 **SMS**：从模式选择 (Slave mode selection)

选择外部信号时，触发信号 (TRGI) 的有效边沿与外部输入上所选的极性相关（请参见输入控制寄存器和控制寄存器说明）。
000：禁止从模式——如果 CEN = “1”，预分频器时钟直接由内部时钟提供。
001：编码器模式 1——计数器根据 TI1FP1 电平在 TI2FP2 边沿递增/递减计数。
010：编码器模式 2——计数器根据 TI2FP2 电平在 TI1FP1 边沿递增/递减计数。
011：编码器模式 3——计数器在 TI1FP1 和 TI2FP2 的边沿计数，计数的方向取决于另外一个信号的电平。
100：复位模式——在出现所选触发输入 (TRGI) 上升沿时，重新初始化计数器并生成一个寄存器更新事件。
101：门控模式——触发输入 (TRGI) 为高电平时使能计数器时钟。只要触发输入变为低电平，计数器立即停止计数（但不复位）。计数器的启动和停止都是受控的。
110：触发模式——触发信号 TRGI 出现上升沿时启动计数器（但不复位）。只控制计数器的启动。
111：外部时钟模式 1——由所选触发信号 (TRGI) 的上升沿提供计数器时钟。

注意：如果将 TI1F_ED 选作触发输入 (TS=100)，则不得使用门控模式。实际上，TI1F 每次转换时，TI1F_ED 都输出 1 个脉冲，而门控模式检查的则是触发信号的电平。

图 4.3.2.1.2 TIMx_SMCR 寄存器

该寄存器的 SMS[2:0]位，用于从模式选择，其实就是选择计数器输入时钟的来源。比如高级定时器中断实验我们设置 SMS[2:0]=000，禁止从模式，这样 PSC 预分频器的时钟就直接来自内部时钟（CK_INT），按照我们例程 sys_stm32_clock_init 函数的配置，频率为 168Mhz。这里我们多数情况是默认状态即选择来自于内部时钟。

- **TIMx DMA/中断使能寄存器（TIMx_DIER）**

TIM1/TIM8 的 DMA/中断使能寄存器描述如图 4.3.2.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

位 0 **UIE**：更新中断使能 (Update interrupt enable)
0：禁止更新中断
1：使能更新中断

图 4.3.2.1.3 TIMx_DIER 寄存器

该寄存器用于使能/失能触发 DMA 请求、捕获/比较中断以及更新中断。本实验只用到更新中断，所以把位 0（UIE）置 1 即可。

- **状态寄存器（TIMx_SR）**

TIM1/TIM8 的状态寄存器描述如图 4.3.2.1.4 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

位 0 **UIF**：更新中断标志 (Update interrupt flag)
- 该位在发生更新事件时通过硬件置 1。但需要通过软件清零。
  0：未发生更新。
  1：更新中断挂起。该位在以下情况下更新寄存器时由硬件置 1：
- 上溢或下溢（对于 TIM2 到 TIM5）以及当 TIMx_CR1 寄存器中 UDIS = 0 时。
- TIMx_CR1 寄存器中的 URS = 0 且 UDIS = 0，并且由软件使用 TIMx_EGR 寄存器中的 UG 位重新初始化 CNT 时。

TIMx_CR1 寄存器中的 URS=0 且 UDIS=0，并且 CNT 由触发事件重新初始化时（参见同步控制寄存器说明）。

图 4.3.2.1.4 TIMx_SR 寄存器

该寄存器都是一些中断标志位，比如更新中断标志位、捕获/比较中断标志位等。在高级定时器中断实验我们用到更新中断标志位，当定时器更新中断到来后，位 0（UIF）会由硬件置 1，我们需要在中断服务函数里面把该位清零。

- **计数寄存器（TIMx_CNT）**

TIM1/TIM8 的计数器寄存器描述如图 4.3.2.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 15:0 **CNT[15:0]**：计数器值 (Counter value)

图 4.3.2.1.5 TIMx_CNT 寄存器

TIM1/TIM8 的计数寄存器是 16 位的，计数模式可以是递增计数模式、递减计数模式和中心对齐计数模式。其他定时器和基本定时器一样，可以直接写该寄存器设置计数的初始值，也可以读取该寄存器获取计数器值。

- **预分频寄存器（TIMx_PSC）**

TIM1/TIM8 的预分频寄存器描述如图 4.3.2.1.6 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 15:0 **PSC[15:0]**：预分频器值 (Prescaler value)
计数器时钟频率 (CK_CNT) 等于 f_{CK_PSC} / (PSC[15:0] + 1)。
PSC 包含每次发生更新事件（包括计数器通过 TIMx_EGR 寄存器中的 UG 位清零时，或在配置为“复位模式”时通过触发控制器清零时）时要装载到活动预分频器寄存器的值。

图 4.3.2.1.6 TIMx_PSC 寄存器

所有定时器的预分频寄存器都是 16 位的，即写入该寄存器的数值范围是 0 到 65535，表示 1 到 65536 分频。比如我们要 16800 分频，就往该寄存器写入 16799。

- **自动重载寄存器（TIMx_ARR）**

TIM1/TIM8 的自动重载寄存器描述如图 4.3.2.1.7 所示：

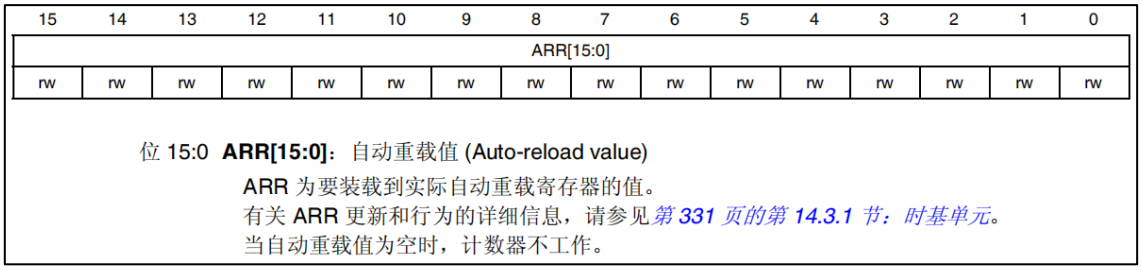


图 4.3.2.1.7 TIMx_ARR 寄存器

在 F4 系列中，TIM2 和 TIM5 的自动重载寄存器是 32 位的，其他定时器的自动重载寄存器是一个 16 位的，该寄存器可以由 APRE 位设置是否进行缓冲。计数器的值会和自动重载寄存器的影子寄存器进行比较，当两者相等，定时器就会溢出，从而发生更新事件，如果打开了更新中断，还会进入中断里面。

4.3.2.2 硬件设计

1. 例程功能

LED0 用来指示程序正在运行，200ms 翻转一次。LED1 在定时器中断中翻转，500ms 进入中断一次。

2. 硬件资源

1）LED 灯

LED0 - PE0

LED1 - PE1

2）定时器 8

3. 原理图

定时器属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们通过 LED1 来指示 STM32F407 的定时器进入中断的频率，LED0 则指示程序的运行状态。

4.3.2.3 程序设计

本实验的相关 HAL 库驱动以及实验配置步骤请参考基本定时器相关内容，基本一样。不同点是基本定时器只能是递增计数模式，高级定时器可以递增计数模式、递减计数模式和中心对齐模式。

4.3.2.3.1 程序流程图

下面看看本实验的程序流程图，main 函数中并没有对 LED1 的操作，我们把对 LED1 的操作放到定时器的中断中进行处理：

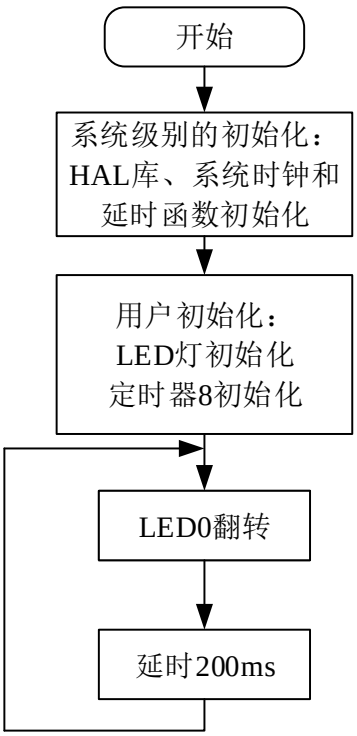


图 4.3.2.3.1.1 高级定时器中断实验程序流程图

4.3.2.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义：

```
/* 高级定时器 定义 */

#define ATIM_TIMX_INT                   TIM8
#define ATIM_TIMX_INT_IRQn              TIM8_UP_TIM13_IRQn
#define ATIM_TIMX_INT_IRQHandler        TIM8_UP_TIM13_IRQHandler
/* TIM8 时钟使能 */
#define ATIM_TIMX_INT_CLK_ENABLE()  do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)
```

通过修改这 4 个宏定义，可以支持 TIM1~TIM8 任意一个定时器。

下面再来看一下 atim.c 文件的代码，主要包括两个函数，先来看看高级定时器的初始化函数，其定义如下：

```
/**
 * @brief       高级定时器 TIMX 定时中断初始化函数
 * @note
 *              高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void atim_timx_int_init(uint16_t arr, uint16_t psc)
{
    ATIM_TIMX_INT_CLK_ENABLE()                                  /* 使能 TIMx 时钟 */

    g_timx_handle.Instance = ATIM_TIMX_INT;                     /* 定时器 x */
```

```
    g_timx_handle.Init.Prescaler = psc;                     /* 预分频系数 */
    g_timx_handle.Init.CounterMode = TIM_COUNTERMODE_UP;    /* 递增计数模式 */
    g_timx_handle.Init.Period = arr;                        /* 自动装载值 */
    HAL_TIM_Base_Init(&g_timx_handle);

    /* 设置中断优先级，抢占优先级 1，子优先级 3 */
    HAL_NVIC_SetPriority(ATIM_TIMX_INT_IRQn, 1, 3);
    HAL_NVIC_EnableIRQ(ATIM_TIMX_INT_IRQn);                 /* 开启 ITMx 中断 */

    HAL_TIM_Base_Start_IT(&g_timx_handle);  /* 使能定时器 x 和定时器 x 更新中断 */
}
```

这里配置的参数和基本定时器中断实验的是一样的，只是这里没有使用到 HAL 库的 HAL_TIM_Base_MspInit 函数来存放 NVIC 和使能时钟的代码，而是全部存放到 atim_timx_int_init 函数里。在一个项目中，用到多个定时器时，建议大家使用这种方式来处理代码，这样方便代码的管理。

下面再来看看定时器中断服务函数，其定义如下：

```
/**
 * @brief       定时器中断服务函数
 * @param       无
 * @retval      无
 */
void GTIM_TIMX_INT_IRQHandler(void)
{
    /* 以下代码没有使用定时器 HAL 库共用处理函数来处理，而是直接通过判断中断标志位的方式 */
    if(__HAL_TIM_GET_FLAG(&g_timx_handle, TIM_FLAG_UPDATE) != RESET)
    {
        LED1_TOGGLE();
        /* 清除定时器溢出中断标志位 */
        __HAL_TIM_CLEAR_IT(&g_timx_handle, TIM_IT_UPDATE);
    }
}
```

可以看到，这里我们没有使用 HAL 库的定时器公共处理函数来处理中断部分的代码，而是通过自行判断中断标志位的方式来处理。只不过获取标志位的方式还是使用 HAL 库的函数宏__HAL_TIM_GET_FLAG()，大家也可以直接使用寄存器的方式来操作。

通过__HAL_TIM_GET_FLAG()获取中断标志位并判断是否了中断，然后处理中断程序，最后通过__HAL_TIM_CLEAR_IT()将中断标志位清零，这样就完成了一次对中断的处理。这样的方式来处理中断，也是大家学习 HAL 库需要掌握的。在一个项目中，用到多个定时器相关中断时，建议大家使用这种方式来处理代码，这样方便代码的管理

4.3.2.4 下载验证

下载代码后，可以看到 LED0 不停闪烁（每 400ms 一个周期），而 LED1 也是不停的闪烁，但是闪烁时间较 LED0 慢（每 1s 一个周期）。

4.3.3 高级定时器 PWM 输出实验

本小节我们来学习使用高级定时器的 PWM 输出模式。

脉冲宽度调制(PWM)，是英文“Pulse Width Modulation”的缩写，简称脉宽调制，是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术。我们可以让定时器产生 PWM，在计数器频率固定时，PWM 频率或者周期由自动重载寄存器（TIMx_ARR）的值决定，其占空比由捕获/比较寄存器（TIMx_CCRx）的值决定。PWM 产生原理示意图如下图所示：

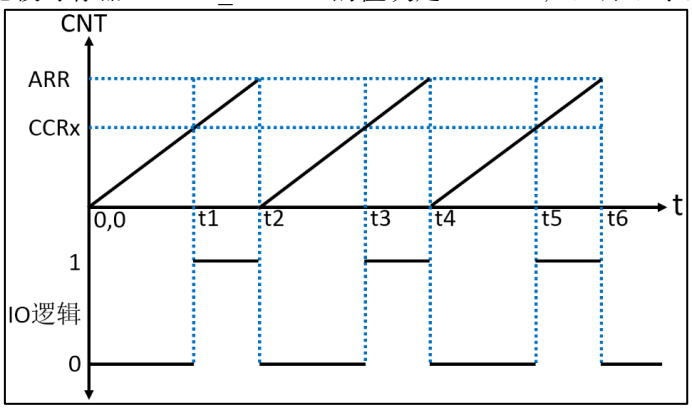


图 4.3.3.1 PWM 原理示意图

上图中，定时器工作在递增计数模式，纵轴是计数器的计数值 CNT，横轴表示时。当 CNT<CCRx 时，IO 输出低电平（逻辑 0）；当 CNT>=CCRx 时，IO 输出高电平（逻辑 1）；当 CNT=ARR 时，定时器溢出，CNT 的值被清零，然后继续递增，依次循环。在这个循环中，改变 CCRx 的值，就可以改变 PWM 的占空比，改变 ARR 的值，就可以改变 PWM 的频率，这就是 PWM 输出的原理。

定时器产生 PWM 的方式有许多种，下面我们以边沿对齐模式（即递增计数模式/递减计数模式）为例，PWM 模式 1 或者 PWM 模式 2 产生 PWM 的示意图，如下图所示：

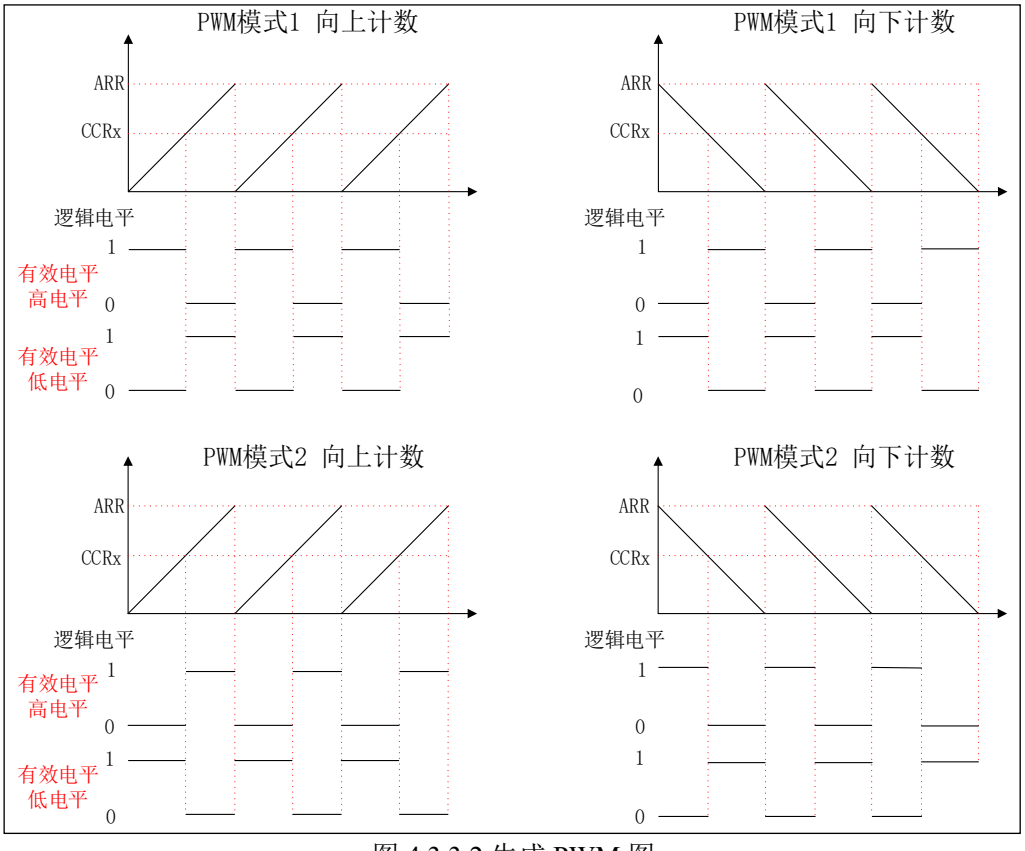


图 4.3.3.2 生成 PWM 图

STM32F407 的定时器除了 TIM6 和 TIM7，其他的定时器都可以用来产生 PWM 输出。其中高级定时器 TIM1 和 TIM8 可以同时产生多达 7 路的 PWM 输出。而通用定时器也能同时产生多达 4 路的 PWM 输出！本实验我们以使用 TIM8 的 CH1 产生一路 PWM 输出为例进行学习。

4.3.3.1 TIM1/TIM8 寄存器

要使用 STM32F407 的高级定时器输出 PWM，除了需要上一小节介绍的寄存器外，我们还会用到另外 5 个寄存器，包括：捕获/比较模式寄存器（TIMx_CCMR1/2）、捕获/比较使能寄存器（TIMx_CCER）、捕获/比较寄存器（TIMx_CCR1~4）、重复计数器寄存器（TIMx_RCR）、断路和死区寄存器（TIMx_BDTR）。上一小节介绍过的寄存器，这里就不再赘述，大家有需要的话可以回顾高级定时器中断章节，接下来我们介绍一下另外的 5 个寄存器。

- **捕获/比较模式寄存器 1/2（TIMx_CCMR1/2）**

TIM1/TIM8 的捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器一般有 2 个：TIMx _CCMR1 和 TIMx _CCMR2。TIMx_CCMR1 控制 CH1 和 CH2，而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 4.3.3.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2 CE	OC2M[2:0]			OC2 PE	OC2 FE	CC2S[1:0]		OC1 CE	OC1M[2:0]			OC1 PE	OC1 FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.3.1.1 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下，功能不一样，我们前面已经说过。比如我们要让 TIM8 的 CH1 输出 PWM 波为例，该寄存器的模式设置位 OC1M[2:0]就是对应着通道 1 的模式设置，此部分由 3 位组成,总共可以配置成 8 种模式，我们使用的是 PWM 模式，所以这 3 位必须设置为 110 或者 111，分别对应 PWM 模式 1 和 PWM 模式 2。这两种 PWM 模式的区别就是输出有效电平的极性相反，这里我们设置为 PWM 模式 2。位 3 OC1PE 是输出比较通道 1 的预装使能，该位需要置 1，另外 CC1S[1:0]用于设置通道 1 的方向（输入/输出）默认设置为 0，就是设置通道作为输出使用。

- **捕获/比较使能寄存器（TIMx_CCER）**

TIM1/TIM8 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关。TIMx_CCER 寄存器描述如图 4.3.3.1.2 所示：

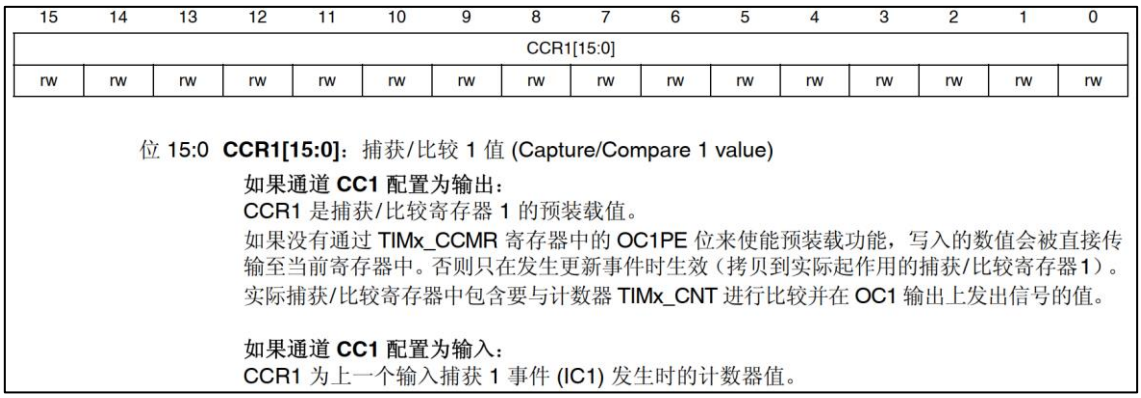
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CC4P	CC4E	CC3NP	CC3NE	CC3P	CC3E	CC2NP	CC2NE	CC2P	CC2E	CC1NP	CC1NE	CC1P	CC1E
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.3.1.2 TIMx_CCER 寄存器

该寄存器比较简单，要让 TIM8 的 CH1 输出 PWM 波，这里我们要使能 CC1E 位，该位是通道 1 输入/输出使能位，要想 PWM 从 IO 口输出，这个位必须设置为 1。CC1P 位是设置通道 1 的输出极性，我们设置 0，即 OC1 高电平有效。

- **捕获/比较寄存器 1/2/3/4（TIMx_CCR1/2/3/4）**

捕获/比较寄存器（TIMx_CCR1/2/3/4），该寄存器总共有 4 个，对应 4 个通道 CH1~CH4。我们使用的是通道 1，所以来看看 TIMx_CCR1 寄存器描述如图 4.3.3.1.3 所示：



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 15:0 **CCR1[15:0]**：捕获/比较 1 值 (Capture/Compare 1 value)

如果通道 CC1 配置为输出：
CCR1 是捕获/比较寄存器 1 的预装载值。
如果没有通过 TIMx_CCMR 寄存器中的 OC1PE 位来使能预装载功能，写入的数值会被直接传输至当前寄存器中。否则只在发生更新事件时生效（拷贝到实际起作用的捕获/比较寄存器1）。
实际捕获/比较寄存器中包含要与计数器 TIMx_CNT 进行比较并在 OC1 输出上发出信号的值。

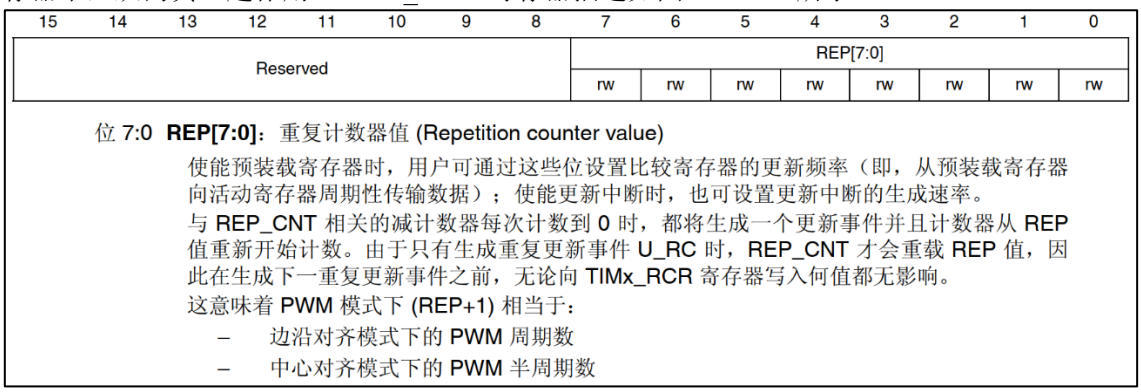
如果通道 CC1 配置为输入：
CCR1 为上一个输入捕获 1 事件 (IC1) 发生时的计数器值。

图 4.3.3.1.3 TIMx_CCR1 寄存器

在输出模式下，捕获/比较寄存器影子寄存器的值与 CNT 的值比较，根据比较结果产生相应动作，利用这点，我们通过修改这个寄存器的值，就可以控制 PWM 的占空比了。

- **重复计数器寄存器（TIMx_RCR）**

重复计数器寄存器仅高级定时器拥有，此寄存器用于设置重复计数器值，因为它具有影子寄存器，所以它本身只是起缓冲作用。当更新事件发生时，该寄存器的值会转移到其影子寄存器中，从而真正起作用。TIMx_ RCR 寄存器描述如图 4.3.3.1.4 所示：



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								REP[7:0]							
								rw	rw	rw	rw	rw	rw	rw	rw

位 7:0 **REP[7:0]**：重复计数器值 (Repetition counter value)

使能预装载寄存器时，用户可通过这些位设置比较寄存器的更新频率（即，从预装载寄存器向活动寄存器周期性传输数据）；使能更新中断时，也可设置更新中断的生成速率。
与 REP_CNT 相关的减计数器每次计数到 0 时，都将生成一个更新事件并且计数器从 REP 值重新开始计数。由于只有生成重复更新事件 U_RC 时，REP_CNT 才会重载 REP 值，因此在生成下一重复更新事件之前，无论向 TIMx_RCR 寄存器写入何值都无影响。
这意味着 PWM 模式下 (REP+1) 相当于：

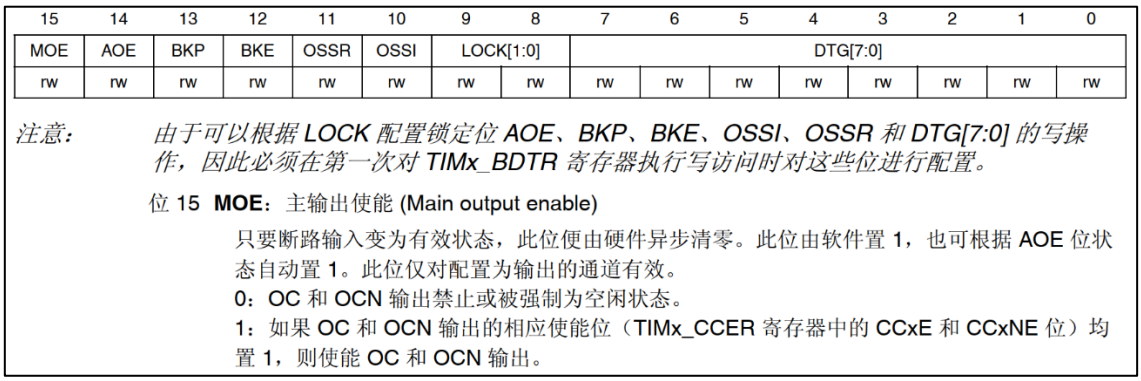
- 边沿对齐模式下的 PWM 周期数
- 中心对齐模式下的 PWM 半周期数

图 4.3.3.1.4 TIMx_ RCR 寄存器

该寄存器的 REP[7:0]位是低 8 位有效，即最大值 255。因为这个寄存器只是起缓冲作用，如果大家对该寄存器写入值后，想要立即生效，可以通过对 UG 位写 1，产生软件更新事件。注意：如果是通用定时器，则没有重复计数器寄存器。

- **TIM1/TIM8 断路和死区寄存器（TIMx_BDTR）**

高级定时器 TIM1/TIM8 的通道用作输出时，还必须配置断路和死区寄存器（TIMx_BDTR）的位 MOE，该寄存器各位描述如图 4.3.3.1.5 所示：



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOE	AOE	BKP	BKE	OSSR	OSSI	LOCK[1:0]		DTG[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

注意： *由于可以根据 LOCK 配置锁定位 AOE、BKP、BKE、OSSI、OSSR 和 DTG[7:0] 的写操作，因此必须在第一次对 TIMx_BDTR 寄存器执行写访问时对这些位进行配置。*

位 15 **MOE**：主输出使能 (Main output enable)

只要断路输入变为有效状态，此位便由硬件异步清零。此位由软件置 1，也可根据 AOE 位状态自动置 1。此位仅对配置为输出的通道有效。
0：OC 和 OCN 输出禁止或被强制为空闲状态。
1：如果 OC 和 OCN 输出的相应使能位（TIMx_CCER 寄存器中的 CCxE 和 CCxNE 位）均置 1，则使能 OC 和 OCN 输出。

图 4.3.3.1.5 TIMx_BDTR 寄存器

该寄存器，我们只需要关注位 15（MOE），要想高级定时器的 PWM 正常输出，则必须设置 MOE 位为 1，否则不会有输出。注意：如果是通用定时器，则没有断路和死区寄存器。

4.3.3.2 硬件设计

1. 例程功能

通过 TIM8_CH1（由 PI5 复用）输出 PWM，从而实现输出脉冲到该 IO 上。然后使用示波器查看 PI5 输出的 PWM 波形，LED0 闪烁提示系统正在运行。

2. 硬件资源

1）LED 灯：
LED0 – PE0
2）定时器 8，使用 TIM8 通道 1，由 PI5 复用。
3）示波器，使用示波器连接开发板的 PI5 和 GND 引脚。

3. 原理图

定时器属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们通过示波器查看 PI5 输出波形情况，LED0 指示程序是否正常运行。

4.3.3.3 程序设计

4.3.3.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面介绍基本定时器已经介绍了部分，这里我们再介绍几个本实验用到的函数。

1. HAL_TIM_PWM_Init 函数

定时器的 PWM 输出模式初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_PWM_Init(TIM_HandleTypeDef *htim);
```

- **函数描述：**
  用于初始化定时器的 PWM 输出模式。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，基本定时器的时候已经介绍。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。
- **注意事项：**
  该函数实现的功能以及使用方法和 HAL_TIM_Base_Init 都是类似的，作用都是初始化定时器的 ARR 和 PSC 等参数。为什么 HAL 库要提供这个函数而不直接让我们使用 HAL_TIM_Base_Init 函数呢？这是因为 HAL 库为定时器的针对 PWM 输出定义了单独的 MSP 回调函数 HAL_TIM_PWM_MspInit，所以当我们调用 HAL_TIM_PWM_Init 进行 PWM 初始化之后，该函数内部会调用 MSP 回调函数 HAL_TIM_PWM_MspInit。而当我们使用 HAL_TIM_Base_Init 初始化定时器参数的时候，它内部调用的回调函数为 HAL_TIM_Base_MspInit，这里大家注意区分。

2. HAL_TIM_PWM_ConfigChannel 函数

定时器的 PWM 通道设置初始化函数。其声明如下：

```
HAL_StatusTypeDef HAL_TIM_PWM_ConfigChannel(TIM_HandleTypeDef *htim,
                                            TIM_OC_InitTypeDef *sConfig, uint32_t Channel);
```

- **函数描述：**
  该函数用于设置定时器的 PWM 通道。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。
  形参 2 是 TIM_OC_InitTypeDef 结构体类型指针变量，用于配置定时器的输出比较参数。
  重点了解一下 TIM_OC_InitTypeDef 结构体指针类型，其定义如下：

```
typedef struct
```

```
{
  uint32_t OCMode;           /* 输出比较模式选择，寄存器的时候说过了，共 8 种模式 */
  uint32_t Pulse;            /* 设置比较值 */
  uint32_t OCPolarity;       /* 设置输出比较极性 */
  uint32_t OCNPolarity;      /* 设置互补输出比较极性 */
  uint32_t OCFastMode;       /* 使能或失能输出比较快速模式 */
  uint32_t OCIdleState;      /* 选择空闲状态下的非工作状态（OC1 输出） */
  uint32_t OCNIdleState;     /* 设置空闲状态下的非工作状态（OC1N 输出） */
} TIM_OC_InitTypeDef;
```

我们重点关注前三个结构体成员。成员变量 OCMode 用来设置模式，这里我们设置为 PWM 模式 2。成员变量 Pulse 用来设置捕获比较值。成员变量 TIM_OCPolarity 用来设置输出极性。其他成员 TIM_OutputNState，TIM_OCNPolarity，TIM_OCIdleState 和 TIM_OCNIdleState 后面用到再介绍。

形参 3 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。这里我们使用的是定时器 8 的通道 1，所以取值为 TIM_CHANNEL_1 即可。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

3. HAL_TIM_PWM_Start 函数

定时器的 PWM 输出启动函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim,uint32_t Channel);
```

- **函数描述：**

  用于使能通道输出和启动计数器，即启动 PWM 输出。

- **函数形参：**

  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。

  形参 2 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

- **注意事项：**

  对于单独使能定时器的方法，在上一章定时器实验我们已经讲解。实际上，HAL 库也同样提供了单独使能定时器的输出通道函数，函数为：

```
void TIM_CCxChannelCmd(TIM_TypeDef *TIMx, uint32_t Channel,
                                          uint32_t ChannelState);
```

HAL_TIM_PWM_Start 函数内部也调用了该函数。

4. HAL_TIM_ConfigClockSource 函数

配置定时器时钟源函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_ConfigClockSource(TIM_HandleTypeDef *htim,
                                    TIM_ClockConfigTypeDef *sClockSourceConfig);
```

- **函数描述：**

  用于配置定时器时钟源。

- **函数形参：**

  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。

  形参 2 是 TIM_ClockConfigTypeDef 结构体类型指针变量，用于配置定时器时钟源参数。

  TIM_ClockConfigTypeDef 定义如下：

```
typedef struct
{
  uint32_t ClockSource;          /* 时钟源 */
  uint32_t ClockPolarity;        /* 时钟极性 */
  uint32_t ClockPrescaler;       /* 定时器预分频器 */
  uint32_t ClockFilter;          /* 时钟过滤器 */
} TIM_ClockConfigTypeDef;
```

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

- **注意事项：**

  该函数主要配置 TIMx_SMCR 寄存器。默认情况下，定时器的时钟源是内部时钟。本实

验就是使用内部时钟的，所以我们不用对时钟源就行初始化，默认即可。这里只是让大家知道有这个函数可以设定时器的时钟源。比如用 HAL_TIM_ConfigClockSource 初始化选择内部时钟，方法如下：

```
TIM_HandleTypeDef timx_handle;                                  /* 定时器 x 句柄 */
TIM_ClockConfigTypeDef sClockSourceConfig = {0};
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;      /* 选择内部时钟 */
HAL_TIM_ConfigClockSource(&timx_handle, &sClockSourceConfig);
```

后面的定时器初始化凡是用到内部时钟我们都没有去初始化，系统默认即可。

定时器 PWM 输出模式配置步骤

1）开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1，对应 IO 是 PI5，它们的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE();          /* 使能定时器 8 */
__HAL_RCC_GPIOI_CLK_ENABLE();         /* 开启 GPIOI 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的 PWM 输出功能时，通过 HAL_TIM_PWM_Init 函数初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数，我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能以及优先级设置等代码。

3）设置 TIMx_CHy 的 PWM 模式，输出比较极性，比较值等参数

在 HAL 库中，通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为 PWM1 模式或者 PWM2 模式，根据需求设置输出比较极性，设置比较值（控制占空比）等。

4）使能 TIMx，使能 TIMx 的 CHy 输出

在 HAL 库中，通过调用 HAL_TIM_PWM_Start 函数来使能 TIMx 的某个通道输出 PWM。

5）修改 TIM2_CCR1 来控制占空比

在经过以上设置之后，PWM 其实已经开始输出了，只是其占空比和频率都是固定的，而我们通过修改比较值来控制 PWM 的输出占空比。HAL 库中提供一个修改占空比的宏定义：

```
__HAL_TIM_SET_COMPARE (__HANDLE__, __CHANNEL__, __COMPARE__)
```

__HANDLE__是 TIM_HandleTypeDef 结构体类型指针变量，__CHANNEL__对应 PWM 的输出通道，__COMPARE__则是要写到捕获/比较寄存器（TIMx_CCR1/2/3/4）的值。实际上该宏定义最终还是往对应的捕获/比较寄存器写入比较值来控制 PWM 波的占空比，如下解析：

比如我们要修改定时器 8 通道 1 的输出比较值（控制占空比），寄存器操作方法：

```
TIM8->CCR1 = ledrpwmval; /* ledrpwmval 是比较值，并且动态变化的，
                            所以我们要周期性调用这条语句，已达到及时修改 PWM 的占空比 */
```

__HAL_TIM_SET_COMPARE 这个宏定义函数最终也是调用这个寄存器操作的，所以说我们使用 HAL 库的函数其实就是间接操作寄存器的。

4.3.3.3.2 程序流程图

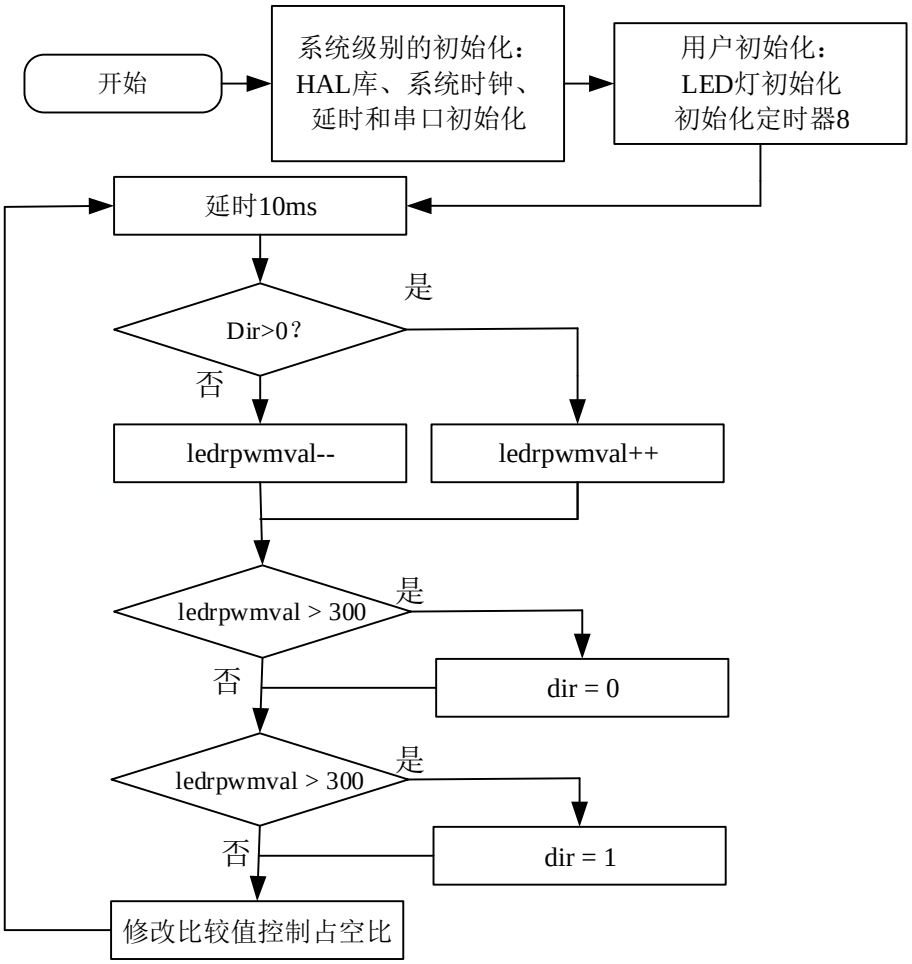


图 4.3.3.3.2.1 高级定时器 PWM 输出实验程序流程图

4.3.3.3.3 程序解析

这里我们这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义：

```
/* TIMX PWM 输出定义 */

#define ATIM_TIMX_PWM_CHY_GPIO_PORT       GPIOI
#define ATIM_TIMX_PWM_CHY_GPIO_PIN        GPIO_PIN_5
#define ATIM_TIMX_PWM_CHY_GPIO_CLK_ENABLE()
        do{  __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)      /* PI 口时钟使能 */

/* TIMX 引脚复用设置
 * 因为 PI5 默认并不是 TIM8 的功能脚，必须开启复用，PI5 才能用作 TIM8 的 PWM 输出引脚
 */
#define ATIM_TIMX_PWM_CHY_GPIO_AF         GPIO_AF3_TIM8

#define ATIM_TIMX_PWM                     TIM8
#define ATIM_TIMX_PWM_CHY                 TIM_CHANNEL_1 /* 通道 Y，  1<= Y <=4 */
#define ATIM_TIMX_PWM_CHY_CLK_ENABLE()
        do{  __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)      /* TIM8 时钟使能 */
```

可以把上面的宏定义分成两部分，第一部分是定时器 8 输出通道 1 对应的 IO 口的宏定义。第二部分则是定时器 8 输出通道 1 的相应宏定义。

下面看 atim.c 的程序，首先是高级定时器 PWM 输出初始化函数。

```
/**
 * @brief       高级定时器 TIMX 通道 Y PWM 输出 初始化函数（使用 PWM 模式 1）
 * @note
 *              高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void atim_timx_pwm_chy_init(uint16_t arr, uint16_t psc)
{
    g_timx_pwm_chy_handle.Instance = ATIM_TIMX_PWM;      /* 定时器 x */
    g_timx_pwm_chy_handle.Init.Prescaler = psc;          /* 定时器分频 */
    g_timx_pwm_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP;/* 向上计数 */
    g_timx_pwm_chy_handle.Init.Period = arr;             /* 自动重装载值 */
    g_timx_pwm_chy_handle.Init.RepetitionCounter = 0;    /* 不重复计数*/
    HAL_TIM_PWM_Init(&g_timx_pwm_chy_handle);            /* 初始化 PWM */

    g_timx_oc_pwm_chy.OCMode = TIM_OCMODE_PWM2;          /* 模式选择 PWM2 */
    /* 设置比较值，此值用来确定占空比 ，默认比较值为自动重装载值的一半,即占空比为 50%*/
    g_timx_oc_pwm_chy.Pulse = ( arr + 1 ) / 2;
    g_timx_oc_pwm_chy.OCPolarity = TIM_OCPOLARITY_HIGH;/* 输出比较极性为高 */
    HAL_TIM_PWM_ConfigChannel(&g_timx_pwm_chy_handle, & g_timx_oc_pwm_chy,
                              ATIM_TIMX_PWM_CHY);        /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_Start(&g_timx_pwm_chy_handle,
                              ATIM_TIMX_PWM_CHY);        /* 开启对应 PWM 通道 */
}
```

HAL_TIM_PWM_Init 初始化 TIM8 并设置 TIM8 的 ARR 和 PSC 等参数，其次通过调用函数 HAL_TIM_PWM_ConfigChannel 设置 TIM8_CH1 的 PWM 模式以及比较值等参数，最后通过调用函数 HAL_TIM_PWM_Start 来使能 TIM8 以及使能 PWM 通道 TIM8_CH1 输出。

本实验我们使用 PWM 的 MSP 初始化回调函数 HAL_TIM_PWM_MspInit 来存放时钟、GPIO 的初始化代码，其定义如下：

```
/**
 * @brief       定时器底层驱动，时钟使能，引脚配置
                此函数会被 HAL_TIM_PWM_Init()调用
 * @param       htim:定时器句柄
 * @retval      无
 */
void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == ATIM_TIMX_PWM)
    {
        GPIO_InitTypeDef gpio_init_struct;
        ATIM_TIMX_PWM_CHY_CLK_ENABLE();                     /* 开启 IO 时钟 */
        ATIM_TIMX_PWM_CHY_GPIO_CLK_ENABLE();                /* 开启定时器 X 时钟 */

        gpio_init_struct.Pin = ATIM_TIMX_PWM_CHY_GPIO_PIN; /* 通道 y 的 GPIO 口 */
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;            /* 复用推挽输出 */
        gpio_init_struct.Pull = GPIO_PULLUP;                /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;      /* 高速 */
        gpio_init_struct.Alternate = ATIM_TIMX_PWM_CHY_GPIO_AF;   /* 端口复用 */
        HAL_GPIO_Init(ATIM_TIMX_PWM_CHY_GPIO_PORT, &gpio_init_struct);
    }
}
```

该函数首先判断定时器寄存器基地址，符合条件后，开启对应的 GPIO 时钟和定时器时

钟，并且初始化 GPIO。上面是使用 HAL 库标准的做法，我们亦可把 HAL_TIM_PWM_MspInit 函数里面的代码直接放到 gtim_timx_pwm_chy_init 函数里。这样做的好处是当一个项目中用到多个定时器时，代码的移植性、可读性好，方便管理。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint16_t ledrpwmval = 0;
    uint8_t dir = 1;

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7); /* 设置时钟, 168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    /* 168M/168=1M 的计数频率，自动重装载为 500，那么 PWM 频率为 1M/500=2kHz */
    gtim_timx_pwm_chy_init(500 - 1, 168 - 1);

    while (1)
    {
        delay_ms(10);

        if (dir)ledrpwmval++;               /* dir==1 ledrpwmval 递增 */
        else ledrpwmval--;                  /* dir==0 ledrpwmval 递减 */

        if (ledrpwmval > 300)dir = 0;       /* ledrpwmval 到达 300 后，方向为递减 */
        if (ledrpwmval == 0)dir = 1;        /* ledrpwmval 递减到 0 后，方向改为递增 */

        /* 修改比较值控制占空比 */
        __HAL_TIM_SET_COMPARE(&g_timx_pwm_chy_handle, GTIM_TIMX_PWM_CHY,
                              ledrpwmval);
    }
}
```

本小节开头我们就说 PWM 波频率由自动重载寄存器（TIMx_ARR）的值决定，其占空比则由捕获/比较寄存器（TIMx_CCRx）的值决定。下面结合实际看看具体怎么计算：

定时器 8 的时钟为 2 倍 APB2，即频率为 168MHZ，而调用 atim_timx_pwm_chy_init 初始化函数之后，就相当于写入预分频寄存器的值为 167，写入自动重载寄存器的值为 499。由定时器溢出公式得：

$$Tout= ((arr+1)*(psc+1))/Tclk= ((499+1)*(167+1))/168000000=0.0005s$$

再由频率是周期的倒数关系得到 PWM 的频率为 2000Hz。

占空比怎么计算的呢？结合图 4.3.3.1，我们分两种情况分析，输出比较极性为低和输出比较极性为高，它们的情况正好相反。因为在 main 函数中的比较值是动态变化的，不利于我们计算占空比，我们假设比较值固定为 200，在本实验中可以调用如下语句得到。

```
__HAL_TIM_SET_COMPARE(&g_timx_pwm_chy_handle, GTIM_TIMX_PWM_CHY, 200);
```

我们在 gtim_timx_pwm_chy_init 函数中设置了输出比较极性为高，那么当比较值固定为 200 时，占空比 =((arr+1) – CCR1) / (arr+1) = (500-200)/500=60%。其中 arr 是写入自动重载寄存器（TIMx_ARR）的值，CCR1 就是写入捕获/比较寄存器 1（TIMx_CCR1）的值。这里我们还需要提醒一下，占空比是指在一个周期内，高电平时间相对于总时间所占的比例。

另外一种情况：设置了输出比较极性为低，那么当比较值固定为 200 时，占空比 =CCR1 / (arr+1) = 200/500=40%。可以看到输出比较极性为低和输出比较极性为高的占空比正好反过来。

在这里，我们也用了 DS100 示波器进行验证，效果图如图 4.3.3.3.1 所示：

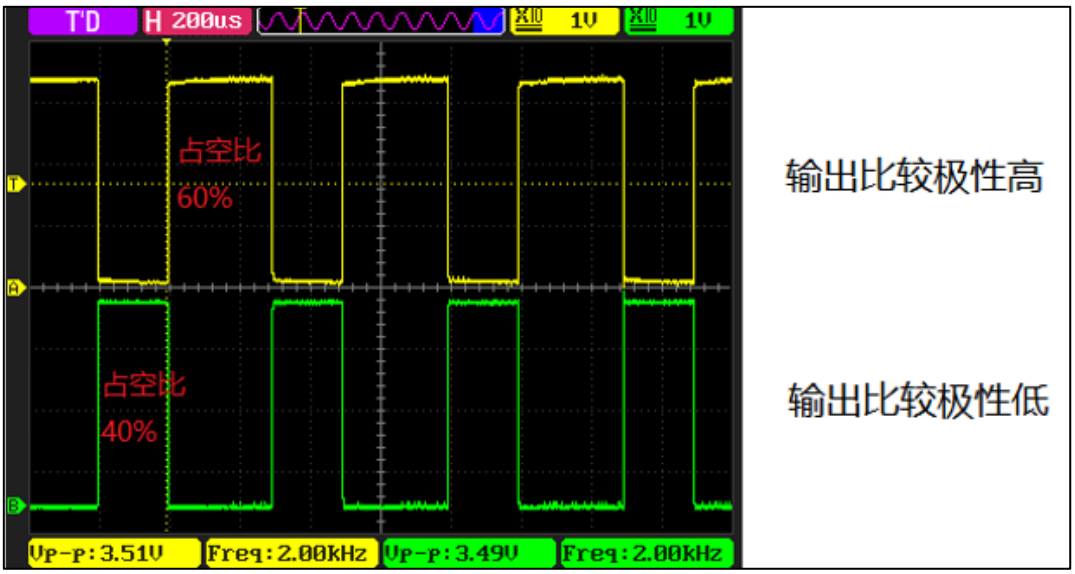


图 4.3.3.3.1 验证效果图

这里把输出比较极性低和输出比较极性高的 PWM 波形都显示出来了。本实验默认设置 PWM 模式 2、输出比较极性高，当 CCR1 寄存器的值设置为 200 时，对应的 PWM 波形如上图黄色的波形图。如果把输出比较极性设置为低，对应的波形图就是绿色的波形图了。

大家感兴趣也可以自行用示波器进行验证。

4.3.3.4 下载验证

下载代码后，我们需要使用示波器连接开发板的 PI5 和 GND 引脚，在示波器上就可以看到频率为 2k Hz，占空比不断变化的 PWM 波形，效果图如图 4.3.3.4.1 所示：

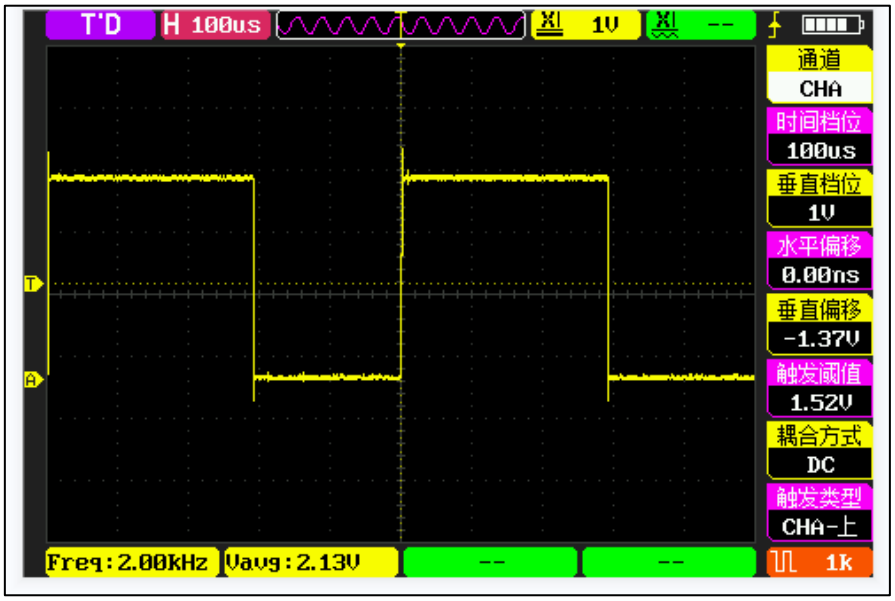


图 4.3.3.4.1 验证效果图

4.3.4 高级定时器互补输出带死区刹车控制实验

本小节我们来学习使用高级定时器的互补输出带死区控制功能。对于刚接触这个知识的朋友可能会问：什么是互补输出？还带死区控制？What？下面给大家简单说一下。

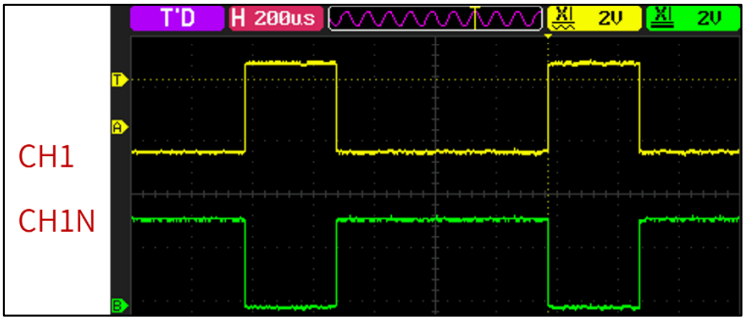


图 4.3.4.1 互补输出

上图中，CH1 输出黄色的 PWM，它的互补通道 CH1N 输出绿色的 PWM。通过对比，可以知道这两个 PWM 刚好是反过来的，CH1 的 PWM 为高电平期间，CH1N 的 PWM 则是低电平，反之亦然，这就是互补输出。

下面来看一下什么是带死区控制的互补输出？

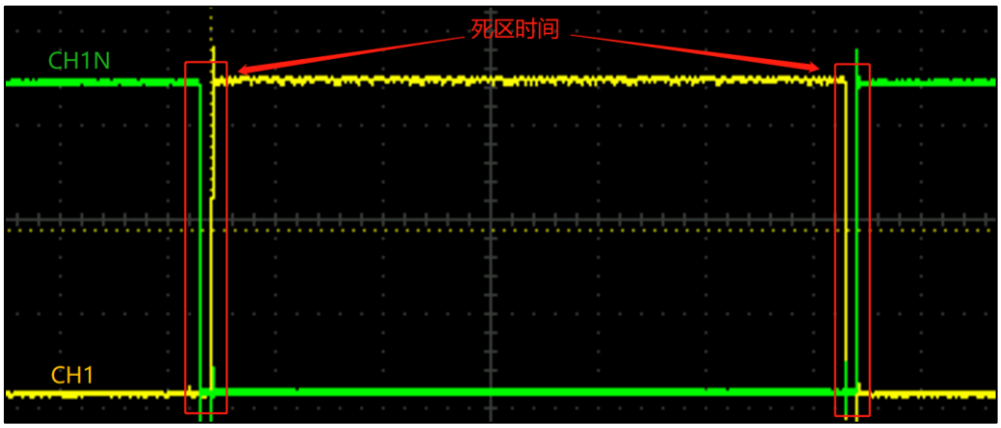


图 4.3.4.2 带死区控制的互补输出

上图中，CH1 输出的 PWM 和 CH1N 输出的 PWM 在高低电平转换间，插入了一段时间才实现互补输出。这段时间称为死区时间，可以通过 TIMx_BDTR 寄存器的 DTG[7:0]位配置控制死区时间的长度，后面会详细讲解如何配置死区时间。上图中，箭头指出的两段死区时间的长度是一样的，因为都是由同一个死区发生器产生。

理解了互补输出和带死区控制的互补输出，下面来看一下带死区控制的互补输出有什么用？带死区控制的互补输出经常被用于控制电机的 H 桥中，下面给大家画了一个 H 桥的简图：

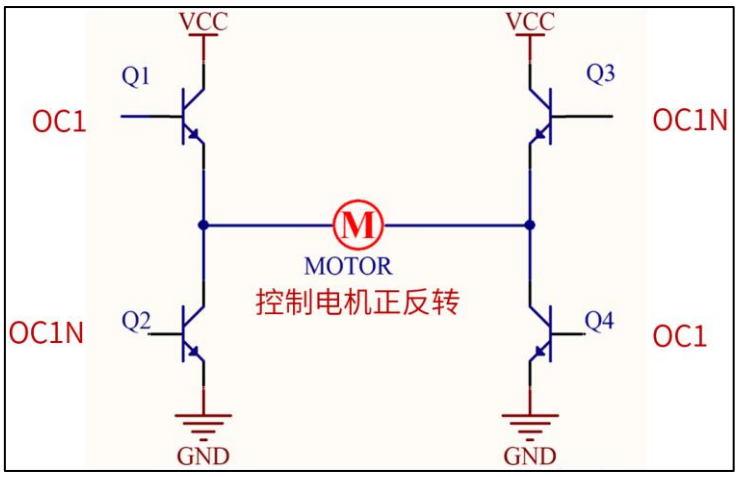


图 4.3.4.3 H 桥简图

图 4.3.4.3 是 H 桥的简图，实际控制电机正反转的 H 桥会根据复杂些，而且更多的是使用 MOS 管，这里只是为了解释带死区控制的互补输出在 H 桥中的控制逻辑原理，大家理解原理就行。上图的 H 桥搭建全部使用的是 NPN，并且导通逻辑都是基极为高电平时导通。如果 Q1 和 Q4 三极管导通，那么电机的电流方向是从左到右（假设电机正转）；如果 Q2 和 Q3 三极管导通，那么电机的电流方向是从右到左（假设电机反转）。上述就是 H 桥控制电机正反转的逻辑原理。但是同一侧的三极管是不可以同时导通的，否则会短路，比如：Q1 和 Q2 同时导通或者 Q3 和 Q4 同时导通，这都是不可取的。

下面大家想一下图 4.3.4.1 的 OC1（CH1）和 OC1N（CH1N）输出的 PWM 输入到图 4.3.4.3 的 H 桥中，会怎样？按理来说应该是 OC1N 输出高电平的时候，OC1 输出就是低电平，刚好 Q2 和 Q3 导通，电机的电流方向是从右到左（假设电机反转）；反之，OC1 输出高电平的时候，OC1N 输出就是低电平，刚好 Q1 和 Q4 导通，电机的电流方向是从左到右（假设电机正转），这似乎已经完美解决电机正反转问题了。实际上，元器件是有延迟特性的，比如：控制信号从 OC1 传导至电机，是要经过一定的时间的，复杂的 H 桥电路更是如此。由于元器件特性，就会导致直接使用互补输出信号驱动 H 桥时存在短路现象。为了避免这种情况，于是就有了带死区控制的互补输出来驱动 H 桥电路。如图 4.3.4.2 的死区时间就是为了解决元器件延迟特性的。用户必须根据与输出相连接的器件及其特性来调整死区时间。

死区时间计算

下面来看一下定时器的死区时间是怎么计算并设置的？死区时间是由 TIMx_CR1 寄存器的 CKD[1:0]位和 TIMx_BDTR 寄存器的 DTG[7:0]位来设置，如下图所示：

TIMx_CR1

CKD[1:0]：时钟分频 (Clock division)

此位域指示定时器时钟 (CK_INT) 频率与死区发生器以及数字滤波器（ETR、TIx）所使用的死区及采样时钟 (t_{DTS}) 之间的分频比，

00：$t_{DTS}=t_{CK_INT}$
01：$t_{DTS}=2*t_{CK_INT}$
10：$t_{DTS}=4*t_{CK_INT}$
11：保留，不要设置成此值

TIMx_BDTR

DTG[7:0]：配置死区发生器 (Dead-time generator setup)

此位域定义插入到互补输出之间的死区持续时间。DT 与该持续时间相对应。

DTG[7:5]=0xx => DT=DTG[7:0]x t_{dtg}，其中 $t_{dtg}=t_{DTS}$。
DTG[7:5]=10x => DT=(64+DTG[5:0])xt_{dtg}，其中 $T_{dtg}=2xt_{DTS}$。
DTG[7:5]=110 => DT=(32+DTG[4:0])xt_{dtg}，其中 $T_{dtg}=8xt_{DTS}$。
DTG[7:5]=111 => DT=(32+DTG[4:0])xt_{dtg}，其中 $T_{dtg}=16xt_{DTS}$。
示例：如果 T_{DTS}=125ns (8MHz)，则可能的死区值为：
0 到 15875 ns（步长为 125 ns）
16 us 到 31750 ns（步长为 250 ns）
32 us 到 63us（步长为 1 us）
64 us 到 126 us（步长为 2 us）

图 4.3.4.4 CKD[1:0]和 DTG[7:0]位

死区时间计算分三步走：

第一步：通过 CKD[1:0]位确定 tDTS。根据 CKD[1:0]位的描述，可以得到下面的式子：

$$tDTS = \frac{2\text{^}CKD[1:0]}{Tclk}$$

CKD[1:0]：CKD[1:0]位设置的值。

Tclk：定时器的时钟源频率（单位为 MHz）。

假设定时器时钟源频率是 168MHz，我们设置 CKD[1:0]位的值为 2，代入上面的式子可得：

$$tDTS = \frac{2\text{^}CKD[1:0]}{Tclk} = \frac{2\text{^}2}{168000000} = \frac{4}{168000000} = 23.81ns$$

通过上式可得 tDTS 约等于 23.81ns，本实验例程中我们也是这样设置的。

第二步：根据 DTG[7:5]选择计算公式。

第三步：代入选择的公式计算。

下面给大家举个例子，假设定时器时钟源频率是 168MHz，我们设置 CKD[1:0]位的值为 2，DTG[7:0]位的值为 250。从上面的例子知道 CKD[1:0]位的值为 2，得到的 tDTS=23.81ns。下面来看一下 DTG[7:0]位的值为 250，应该选择 DTG[7:0]位描述中哪条公式？250 的二进制数为 11111010，即 DTG[7:5]为 111，所以选择第四条公式：DT=(32+ DTG[4:0]) * t_{dtg}，其中 t_{dtg} = 16 * tDTS。可以看到手册上的式子符号大小写乱乱的，这里大小写不敏感。由手册的公式可以得到 DT = (32+ DTG[4:0]) * 16 * tDTS = (32+ 26) * 16 * 23.81ns = 22095.68ns = 22.01us，即死区时间为 22.01us。死区时间计算方法就给大家介绍到这里。

关于互补输出和死区插入的更多内容请看《STM32F4xx 参考手册_V4（中文版）.pdf》的 14.3.11 小节，下面我们介绍相关的寄存器。

4.3.4.1 TIM1/TIM8 寄存器

高级定时器互补输出带死区控制除了用到定时器的时基单元：计数器寄存器(TIMx_CNT)、预分频器寄存器(TIMx_PSC)、自动重载寄存器(TIMx_ARR)之外。主要还用到以下这些寄存器：

- **控制寄存器 1（TIMx_CR1）**

TIM1/TIM8 的控制寄存器 1 描述如图 4.3.4.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS[1:0]		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 9:8 **CKD[1:0]**：时钟分频 (Clock division)

此位域指示定时器时钟 (CK_INT) 频率与死区发生器以及数字滤波器（ETR、TIx）所使用的死区及采样时钟 (t_{DTS}) 之间的分频比，

00：$t_{DTS} = t_{CK_INT}$

01：$t_{DTS} = 2 * t_{CK_INT}$

10：$t_{DTS} = 4 * t_{CK_INT}$

11：保留，不要设置成此值

位 7 **ARPE**：自动重载预装载使能 (Auto-reload preload enable)

0：TIMx_ARR 寄存器不进行缓冲

1：TIMx_ARR 寄存器进行缓冲

位 0 **CEN**：计数器使能 (Counter enable)

0：禁止计数器

1：使能计数器

图 4.3.4.1.1 TIMx_CR1 寄存器

上图中我们只列出了本实验需要用的一些位，其中：位 7（APRE）用于控制自动重载寄存器是否进行缓冲，在基本定时器的时候已经讲过，请回顾。本实验中，我们把该位置 1。

CKD[1:0]位指示定时器时钟（CK_INT）频率与死区发生器以及数字滤波器（ETR、TIx）所使用的死区及采样时钟（tDTS）之间的分频比。我们设置 CKD[1:0]位为 10，结合高级定时器时钟源频率等于 APB2 总线时钟频率，即 168MHz，可以得到 tDTS=23.81ns。

CEN 位，用于使能计数器的工作，必须要设置该位为 1，才可以开始计数。

- **捕获/比较模式寄存器 1/2（TIMx_CCMR1/2）**

TIM1/TIM8 的捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器一般有 2 个：TIMx_CCMR1 和 TIMx _CCMR2。TIMx_CCMR1 控制 CH1 和 CH2，而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 4.3.4.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2 CE	OC2M[2:0]			OC2 PE	OC2 FE	CC2S[1:0]		OC1 CE	OC1M[2:0]			OC1 PE	OC1 FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.4.1.2 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下，功能不一样，我们现在用到输出比较模式。关于该寄存器的详细说明，请参考《STM32F4xx 参考手册_V4（中文版）.pdf》第 375 页。

本实验我们用到了定时器 8 输出比较的通道 1，所以我们需要配置 TIM1_CCMR1 模式设置位 OC1M[2:0]，我们使用的是 PWM 模式 1，所以这 3 位必须设置为 110。

- **捕获/比较使能寄存器（TIMx_CCER）**

TIM1/TIM8 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 4.3.4.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CC4P	CC4E	CC3NP	CC3NE	CC3P	CC3E	CC2NP	CC2NE	CC2P	CC2E	CC1NP	CC1NE	CC1P	CC1E
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.4.1.3 TIMx_CCER 寄存器

该寄存器比较简单，要让 TIM8 的通道 1 输出，我们需要把对应的捕获/比较 1 输出使能位 CC1E 置 1。因为本实验中，我们需要实现互补输出，所以还需要把 CC1NE 位置 1，使能互补通道输出。CC1P 和 CC1NP 分别是通道 1 输出和通道 1 互补输出的极性设置位。这里我们把 CC1P 和 CC1NP 位都置 1，即输出极性为低，就可以得到互补的 PWM。

- **捕获/比较寄存器 1/2/3/4（TIMx_CCR1/2/3/4）**

捕获/比较寄存器（TIMx_CCR1/2/3/4），该寄存器总共有 4 个，对应 4 个通道 CH1~CH4。我们使用的是通道 1，所以来看看 TIMx_CCR1 寄存器描述如图 4.3.4.1.4 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.4.1.4 TIMx_CCR1 寄存器

该寄存器 16 位有效位，本实验中可以通过改变该寄存器的值来改变 PWM 的占空比。

- **断路和死区寄存器（TIMx_BDTR）**

TIM1/TIM8 断路和死区寄存器，该寄存器各位描述如图 4.3.4.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOE	AOE	BKP	BKE	OSSR	OSSI	LOCK[1:0]		DTG[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 14 **AOE**：自动输出使能 (Automatic output enable)

0：MOE 只能由软件置 1

1：MOE 可由软件置 1，也可在发生下一更新事件时自动置 1(如果断路输入无效)

注意：只要编程了 LOCK（TIMx_BDTR 寄存器中的 LOCK 位）级别 1，此位即无法修改。

位 13 **BKP**：断路极性 (Break polarity)

0：断路输入 BRK 为低电平有效

1：断路输入 BRK 为高电平有效

注意：只要编程了 LOCK（TIMx_BDTR 寄存器中的 LOCK 位）级别 1，此位即无法修改。

注意：对该位执行任何写操作后，都需要经过 1 个 APB 时钟周期的延迟才生效。

位 12 **BKE**：断路使能 (Break enable)

0：禁止断路输入（BRK 和 CSS 时钟故障事件）

1：使能断路输入（BRK 和 CSS 时钟故障事件）

注意：编程了 LOCK（TIMx_BDTR 寄存器中的 LOCK 位）级别 1 后，此位即无法修改。

注意：对该位执行任何写操作后，都需要经过 1 个 APB 时钟周期的延迟才生效。

位 7:0 **DTG[7:0]**：配置死区发生器 (Dead-time generator setup)

此位域定义插入到互补输出之间的死区持续时间。DT 与该持续时间相对应。

DTG[7:5]=0xx => DT=DTG[7:0]x t_{dtg}，其中 $t_{dtg}=t_{DTS}$。

DTG[7:5]=10x => DT=(64+DTG[5:0])xt_{dtg}，其中 $T_{dtg}=2xt_{DTS}$。

DTG[7:5]=110 => DT=(32+DTG[4:0])xt_{dtg}，其中 $T_{dtg}=8xt_{DTS}$。

DTG[7:5]=111 => DT=(32+DTG[4:0])xt_{dtg}，其中 $T_{dtg}=16xt_{DTS}$。

图 4.3.4.1.5 TIMx_BDTR 寄存器

该寄存器控制定时器的断路和死区控制的功能。我们先看断路控制，用到断路输入功能（断路输入引脚为 PA6），位 BKE 置 1 即可。

位 BKP 选择断路输入信号有效电平。本实验中，我们选择低电平有效，即 BKP 为 0。

位 AOE 是自动输出使能位，如果使能 AOE 位，那么在我们输入刹车信号后再断开了刹车信号，互补的 PWM 会自动恢复输出，如果失能 AOE 位，那么在输入刹车信号后再断开了刹车信号，互补的 PWM 就不会恢复输出，而是一直保持刹车信号输入时的状态。为了方便观察，我们使能该位，即置 1。

位 MOE 是使能主输出，想要高级定时器的通道正常输出，则必须设置 MOE 位为 1。

最后是 DTG[7:0]位，用于设置死区时间，前面已经教过大家怎么设置了。这里以我们例程的设置为例，CKD[1:0] 设置为 10，定时器时钟源频率是 168MHz，所以 tDTS = 23.81ns。

本例程的 DTG[7:0]位的值设置为十进制 100，即二进制数 0110 0100。DTG[7:5]=011，符合第一条式子：DT=DTG[7:0] * t_{dtg}，其中 t_{dtg} = tDTS。DT 是死区时间，可以得到 DT = 100*23.81 ns = 2.38us。到后面下载验证小节，我们通过示波器验证一下这个死区时间计算的理论值和实际值是否一样。

4.3.4.2 硬件设计

1. 例程功能

1、利用 TIM8_CH1(PI5)输出 70%占空比的 PWM 波，它的互补输出通道(PH13)则是输出 30%占空比的 PWM 波。

2、刹车功能，当给刹车输入引脚(PA6)输入低电平时，进行刹车，即 PI5 和 PH13 停止输出 PWM 波。

3、LED0 闪烁指示程序运行。

2. 硬件资源

1）LED 灯
LED0 – PE0

2）定时器 8
TIM8 正常输出通道 PI5
TIM8 互补输出通道 PH13
TIM8 刹车输入 PA6

3. 原理图

定时器属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们需要通过示波器观察 PI5 和 PH13 引脚 PWM 输出的情况。还可以通过给 PA6 引脚接入低电平进行刹车。

4.3.4.3 程序设计

4.3.4.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面已经介绍了部分，这里我们再介绍几个本实验用到的函数。

1. HAL_TIMEx_ConfigBreakDeadTime 函数

定时器的断路和死区时间配置初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIMEx_ConfigBreakDeadTime(TIM_HandleTypeDef *htim,
                          TIM_BreakDeadTimeConfigTypeDef
*sBreakDeadTimeConfig);
```

- **函数描述：**
  用于初始化定时器的断路（即刹车）和死区时间。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，基本定时器的时候已经介绍。
  形参 2 是 TIM_BreakDeadTimeConfigTypeDef 结构体类型指针变量，用于配置断路和死区参

数，其定义如下：

```
typedef struct
{
  uint32_t OffStateRunMode;      /* 运行模式下的关闭状态选择 */
  uint32_t OffStateIDLEMode;     /* 空闲模式下的关闭状态选择 */
  uint32_t LockLevel;            /* 寄存器锁定配置 */
  uint32_t DeadTime;             /* 死区时间设置 */
  uint32_t BreakState;           /* 断路（即刹车）输入使能控制 */
  uint32_t BreakPolarity;        /* 断路输入极性 */
  uint32_t BreakFilter;          /* 断路输入滤波器 */
  uint32_t AutomaticOutput;      /* 自动恢复输出使能控制 */
} TIM_BreakDeadTimeConfigTypeDef;
```

- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

2. HAL_TIMEx_PWMN_Start 函数

定时器的互补输出启动函数。其声明如下：

```
HAL_StatusTypeDef HAL_TIMEx_PWMN_Start(TIM_HandleTypeDef *htim,
                                       uint32_t Channel);
```

- **函数描述：**
  该函数用于启动定时器的互补输出。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。
  形参 2 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

定时器互补输出带死区控制配置步骤

1）开启 TIMx 和通道输出以及刹车输入的 GPIO 时钟，配置该 IO 口的复用功能输出

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1，对应 IO 是 PI5，互补输出通道引脚是 PH13，刹车输入引脚是 PA6，它们的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE();            /* 使能定时器 8 */
__HAL_RCC_GPIOH_CLK_ENABLE();           /* 开启 GPIOH 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

这里我们要使用定时器的 PWM 模式功能，所以调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。注意：本实验要使用该函数配置 TIMx_CR1 寄存器的 CKD[1:0] 位，从而确定 t DTS，方便后续设置死区时间。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数，但是为了方便移植代码，提高不同实验代码的独立性，我们就直接在 atim_timx_cplm_pwm_init 函数中，使能定时器时钟和 GPIO 时钟，初始化通道对应 IO 引脚等。

3）设置定时器为 PWM 模式，输出比较极性，互补输出极性等参数

通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为 PWM1 模式，根据需求设置 OCy 输出极性和 OCyN 互补输出极性等。

4）设置死区参数。

通过 HAL_TIMEx_ConfigBreakDeadTime 函数来设置死区参数，比如：设置死区时间、运行模式的关闭输出状态、空闲模式的关闭输出状态、刹车输入有效信号极性和是否允许刹车后自动恢复输出等。

5）启动 Ocy 输出以及 OCyN 互补输出

通过 HAL_TIM_PWM_Start 函数启动 OCy 输出，通过 HAL_TIMEx_PWMN_Start 函数启动启动 OCyN 互补输出。

4. 3.4.3.2 程序流程图

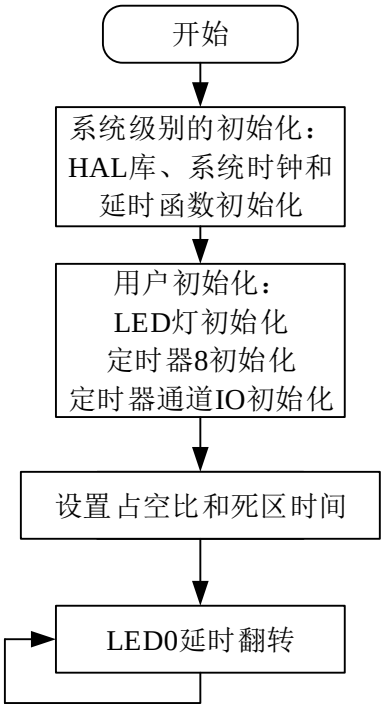


图 4.3.4.3.2.1 高级定时器互补输出带死区控制实验

4.3.4.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义：

```
/* TIMX 互补输出定义 */

/* 主输出通道引脚 */
#define ATIM_TIMX_CPLM_CHY_GPIO_PORT            GPIOI
#define ATIM_TIMX_CPLM_CHY_GPIO_PIN             GPIO_PIN_5
#define ATIM_TIMX_CPLM_CHY_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)    /* PI 口时钟使能 */

/* 互补输出通道引脚 */
#define ATIM_TIMX_CPLM_CHYN_GPIO_PORT           GPIOH
#define ATIM_TIMX_CPLM_CHYN_GPIO_PIN            GPIO_PIN_13
#define ATIM_TIMX_CPLM_CHYN_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOH_CLK_ENABLE(); }while(0)    /* PH 口时钟使能 */

/* 刹车输入引脚 */
#define ATIM_TIMX_CPLM_BKIN_GPIO_PORT           GPIOA
#define ATIM_TIMX_CPLM_BKIN_GPIO_PIN            GPIO_PIN_6
#define ATIM_TIMX_CPLM_BKIN_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)    /* PA 口时钟使能 */

/* TIMX 引脚复用设置
 * 因为 PI5、PH13、PA6 默认并不是 TIM8 的功能脚，必须开启复用，PI5、PH13、PA6 才能用作
   TIM8 的互补输出及刹车引脚。
 */
#define ATIM_TIMX_CPLM_CHY_GPIO_AF              GPIO_AF3_TIM8

#define ATIM_TIMX_CPLM                          TIM8
```

```
#define ATIM_TIMX_CPLM_CHY                      TIM_CHANNEL_1
#define ATIM_TIMX_CPLM_CHY_CCRY                 ATIM_TIMX_CPLM->CCR1
#define ATIM_TIMX_CPLM_CLK_ENABLE()
        do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)   /* TIM8 时钟使能 */
```

可以把上面的宏定义分成两部分，第一部分包括是定时器 8 输出、互补输出和刹车输入通道对应的 IO 口的宏定义，第二部分则是定时器 8 的相应宏定义。注意：因为 PI5、PH13、PA6 默认并不是 TIM8 的功能脚，必须开启复用， PI5、PH13、PA6 才能用作 TIM8 的互补输出及刹车引脚。

下面来看 atim.c 文件的程序，首先是高级定时器互补输出初始化函数，其定义如下：

```
/**
 * @brief       高级定时器 TIMX 互补输出 初始化函数（使用 PWM 模式 1）
 * @note
 *              配置高级定时器 TIMX 互补输出，一路 OCy 一路 OCyN，并且可以设置死区时间
 *
 *              高级定时器的时钟来自 APB2，而 PCLK2 = 168Mhz，我们设置 PPRE2 不分频，因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法：Tout = ((arr + 1) * (psc + 1)) / Ft us
 *              Ft=定时器工作频率，单位 : Mhz
 *
 * @param       arr: 自动重装值。
 * @param       psc: 时钟预分频数
 * @retval      无
 */

void atim_timx_cplm_pwm_init(uint16_t arr, uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct = {0};
    TIM_OC_InitTypeDef tim_oc_cplm_pwm = {0};

    ATIM_TIMX_CPLM_CLK_ENABLE();                    /* 开启定时器 x 时钟 */
    ATIM_TIMX_CPLM_CHY_GPIO_CLK_ENABLE();           /* 开启主输出通道的 GPIO 时钟 */
    ATIM_TIMX_CPLM_CHYN_GPIO_CLK_ENABLE();          /* 开启互补通道的 GPIO 时钟 */
    ATIM_TIMX_CPLM_BKIN_GPIO_CLK_ENABLE();          /* 开启刹车输入的 GPIO 时钟 */

    gpio_init_struct.Pin = ATIM_TIMX_CPLM_CHY_GPIO_PIN;  /* 主输出通道的 IO */
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;             /* 复用推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;                 /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;       /* 高速 */
    gpio_init_struct.Alternate = ATIM_TIMX_CPLM_CHY_GPIO_AF; /* IO 复用 */
    HAL_GPIO_Init(ATIM_TIMX_CPLM_CHY_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = ATIM_TIMX_CPLM_CHYN_GPIO_PIN; /* 互补通道的 IO */
    HAL_GPIO_Init(ATIM_TIMX_CPLM_CHYN_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = ATIM_TIMX_CPLM_BKIN_GPIO_PIN; /* 刹车输入的 IO */
    HAL_GPIO_Init(ATIM_TIMX_CPLM_BKIN_GPIO_PORT, &gpio_init_struct);

    /* 设置定时器 */
    g_timx_cplm_pwm_handle.Instance = ATIM_TIMX_CPLM;     /* 定时器 x */
    g_timx_cplm_pwm_handle.Init.Prescaler = psc;          /* 预分频系数 */
    g_timx_cplm_pwm_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数 */
    g_timx_cplm_pwm_handle.Init.Period = arr;             /* 自动重装载值 */
    /* CKD[1:0] = 10, tDTS = 4 * tCK_INT = Ft / 4 = 42Mhz */
    g_timx_cplm_pwm_handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV4;
    g_timx_cplm_pwm_handle.Init.AutoReloadPreload =
                  TIM_AUTORELOAD_PRELOAD_ENABLE; /* 使能影子寄存器 TIMx_ARR */
    HAL_TIM_PWM_Init(&g_timx_cplm_pwm_handle) ;

    /* 设置 PWM 输出 */
    tim_oc_cplm_pwm.OCMode = TIM_OCMODE_PWM1;             /* PWM 模式 1 */
```

```
    tim_oc_cplm_pwm.OCPolarity = TIM_OCPOLARITY_LOW;        /* OCy 低电平有效 */
    tim_oc_cplm_pwm.OCNPolarity = TIM_OCNPOLARITY_LOW;      /* OCyN 低电平有效 */
    HAL_TIM_PWM_ConfigChannel(&g_timx_cplm_pwm_handle, &tim_oc_cplm_pwm,
                              ATIM_TIMX_CPLM_CHY);

    /* 设置死区参数，开启死区中断 */
    g_sbreak_dead_time_config.LockLevel = TIM_LOCKLEVEL_OFF;/* 不用寄存器锁功能 */
    g_sbreak_dead_time_config.BreakState = TIM_BREAK_ENABLE;/* 使能刹车输入 */
    /* 刹车输入有效信号极性为低 */
    g_sbreak_dead_time_config.BreakPolarity = TIM_BREAKPOLARITY_LOW;
    /* 使能 AOE 位，允许刹车结束后自动恢复输出 */
    g_sbreak_dead_time_config.AutomaticOutput = TIM_AUTOMATICOUTPUT_ENABLE;
    HAL_TIMEx_ConfigBreakDeadTime(&g_timx_cplm_pwm_handle,
                                  &g_sbreak_dead_time_config);
    /* OCy 输出使能 */
    HAL_TIM_PWM_Start(&g_timx_cplm_pwm_handle, ATIM_TIMX_CPLM_CHY);
    /* OCyN 输出使能 */
    HAL_TIMEx_PWMN_Start(&g_timx_cplm_pwm_handle, ATIM_TIMX_CPLM_CHY);
}
```

在 atim_timx_cplm_pwm_init 函数中，没有使用 HAL 库的 MSP 回调，而是把相关的初始化都写到该函数里面。

第一部分，使能定时器和相关通道对应的 GPIO 时钟，以及初始化相关 IO 引脚。

第二部分，通过 HAL_TIM_PWM_Init 函数初始化定时器的 ARR 和 PSC 等参数。

第三部分，通过 HAL_TIM_PWM_ConfigChannel 函数设置 PWM 模式 1、输出极性等。

第四部分，通过 HAL_TIMEx_ConfigBreakDeadTime 函数配置断路功能。

最后一定记得要调用 HAL_TIM_PWM_Start 函数和 HAL_TIMEx_PWMN_Start 函数启动通道输出和互补通道输出。

为了方便，我们还定义了设置输出比较值和死区时间的函数，其定义如下：

```
/**
 * @brief       定时器 TIMX 设置输出比较值 & 死区时间
 * @param       ccr: 输出比较值
 * @param       dtg: 死区时间
 *   @arg       dtg[7:5]=0xx 时，死区时间 = dtg[7:0] * tDTS
 *   @arg       dtg[7:5]=10x 时，死区时间 = (64 + dtg[6:0]) * 2  * tDTS
 *   @arg       dtg[7:5]=110 时，死区时间 = (32 + dtg[5:0]) * 8  * tDTS
 *   @arg       dtg[7:5]=111 时，死区时间 = (32 + dtg[5:0]) * 16 * tDTS
 *   @note      tDTS = 1 / (Ft /  CKD[1:0]) = 1 / 42M = 23.8ns
 * @retval      无
 */
void atim_timx_cplm_pwm_set(uint16_t ccr, uint8_t dtg)
{
    g_sbreak_dead_time_config.DeadTime = dtg;       /* 死区时间设置 */
    HAL_TIMEx_ConfigBreakDeadTime(&g_timx_cplm_pwm_handle,
                                  &g_sbreak_dead_time_config);  /*重设死区时间*/
    __HAL_TIM_MOE_ENABLE(&g_timx_cplm_pwm_handle); /* MOE=1,使能主输出 */
    ATIM_TIMX_CPLM_CHY_CCRY = ccr;                  /* 设置比较寄存器 */
}
```

通过重新调用 HAL_TIMEx_ConfigBreakDeadTime 函数设置死区时间，注意这里的 g_sbreak_dead_time_config 是全局结构体变量，在 atim_timx_cplm_pwm_init 函数已经初始化其他结构体成员了，这里只是对 DeadTime 成员（死区时间）配置。死区时间的计算方法前面已经讲解过，这里只要把要设置的 DTG[7:0]值，通过 dtg 形参赋值给 DeadTime 结构体成员就行。另外一个形参是 ccr，用于设置捕获/比较寄存器的值，即控制 PWM 的占空比。

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);     /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
```

```
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    /* 1Mhz 的计数频率，1Khz 的 PWM 周期 */
    atim_timx_cplm_pwm_init(1000 - 1, 168 - 1);
    atim_timx_cplm_pwm_set(300, 100);           /* 占空比:7:3, 死区时间 100 * tDTS */
    while (1)
    {
        LED0_TOGGLE();                          /*LED0(红灯) 翻转*/
        delay_ms(200);
    }
}
```

先看 atim_timx_cplm_pwm_init(1000 - 1, 168 - 1)这个语句，这两个形参分别设置自动重载寄存器的值为 999，以及定时器预分频器寄存器的值为 167。先看预分频系数，我们设置为 168 分频，定时器 1 的时钟源频率等于 APB2 总线时钟频率，即 168MHz，可以得到计数器的计数频率是 1MHz，即每 1us 计数一次。再到自动重载寄存器的值为 999 决定的是 PWM 的频率（周期），可以得到 PWM 的周期为(999+1)*1us = 1000us = 1ms。边沿对齐模式下，使用 PWM 模式 1 或者 PWM 模式 2，得到的 PWM 周期是定时器溢出时间。这里的 1ms，也可以直接通过定时器溢出时间计算公式 Tout= ((arr+1)*(psc+1))/Tclk 得到。

调用 atim_timx_cplm_pwm_set(300, 100) 这个语句，相当于设置捕获/比较寄存器的值为 300，DTG[7:0]的值为 100。通过计算可以得到 PWM 的占空比为 70%，死区时间为 2.38us。根据 PWM 生成原理分析，再结合图 4.3.3.2 产生 PWM 示意图，以及我们在 atim_timx_cplm_pwm_init 函数配置 PWM 模式 1、OCy 输出极性为低，占空比的计算很简单，可以由(1000-300)/1000 得到。关于死区时间的计算方法，前面已经讲解过，这里以 DTG[7:0]的值为 100 为例，再来讲解一遍计算过程。我们例程配置 CKD[1:0]位的值为 2，可以得到 tDTS = 23.81ns。基于这个前提，通过改变 DTG[7:0]的值，可以得到不同的死区时间。这里我们配置 DTG[7:0]的值为 100，即二进制数 0110 0100，符合第一种情况 dtg[7:5]=0xx 时，死区时间 DT = DTG [7:0] * tDTS。可以得到死区时间 DT = 100*23.81 ns = 2.38us。

下面我们下载到开发板验证一下。

4.3.4.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了。 我们需要借助 DS100 示波器观察 PI5 正常输出和 PH13 互补输出 PWM 的情况，示波器显示截图如图 4.3.4.4.1 所示：

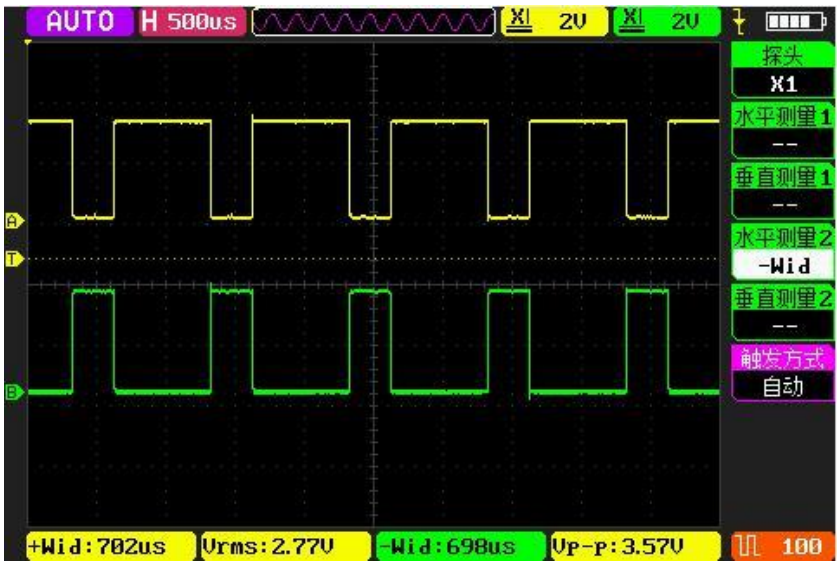


图 4.3.4.4.1 PI5 正常输出和 PH13 互补输出 PWM 的情况

图 4.3.4.4.1 中的由上到下分别是 PI5 输出 70%占空比的 PWM 波和 PB13 互补输出 30%占空比的 PWM 波。互补输出的 PWM 波的正脉宽减去正常的 PWM 的负脉宽的值除以 2 就是死

区时间，也可以是正常的 PWM 的正脉宽减去互补输出的 PWM 波的负脉宽的值除以 2。我们使用第一种方法得到：死区时间 =（702 - 698）/2 us=2us。与我们理论到的的值 2.38us 基本一样，这样的误差是正常的。

要是不相信，我们再举个例子，我们把调用的函数改为 atim_timx_cplm_pwm_set(300, 250)，即配置 DTG[7:0]的值为 250，这个例子的计算过程在本实验前面死区时间计算的内容讲过，这里就不再赘述。经过计算得到死区时间 DT = 22.0864 us。修改好代码后，编译下载到板子，示波器显示截图如下图所示：

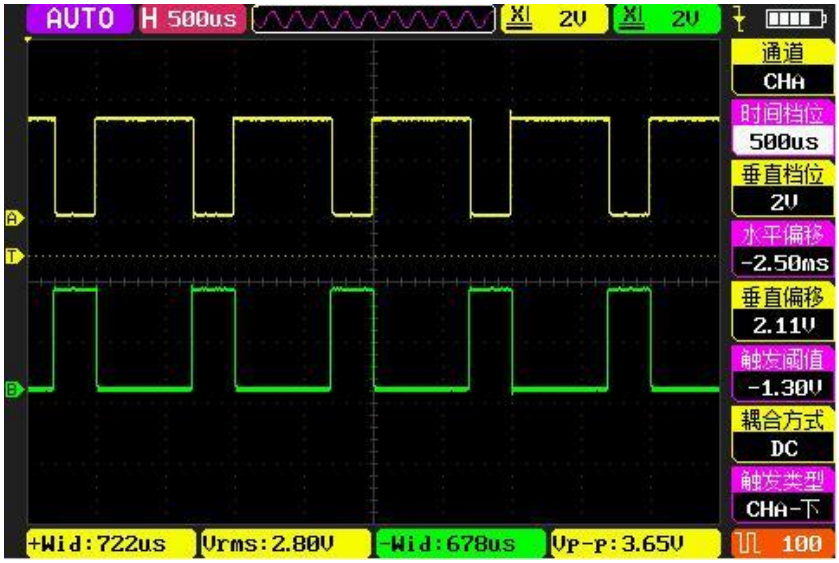


图 4.3.4.4.2 修改程序后 PI5 正常输出和 PH13 互补输出 PWM 的情况

由图 4.3.4.4.2 可得到，死区时间 =（722 - 678）/2 us= 22us。与我们理论到的的值 22.0864us 是差不多的，误差在正常范围。由此证明我们的死区时间设置是没有问题。

刹车功能验证：当给刹车输入引脚（PA6）接入低电平（这里直接用杜邦线连接 PA6 到 GND）时，就会进行刹车，即 PI5 和 PH13 停止输出 PWM 波，如图 4.3.4.4.3 所示：

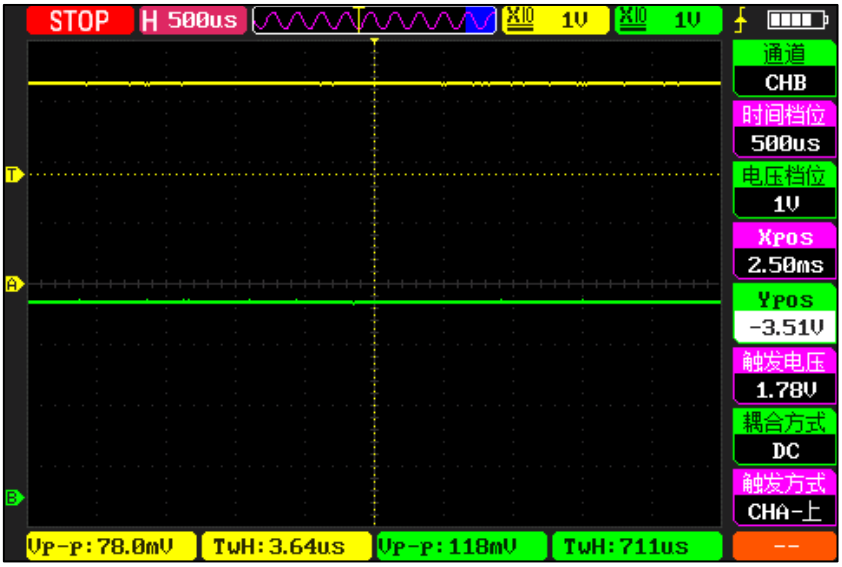


图 4.3.4.4.3 刹车后的输出情况

因为我们使能了 AOE 位（即把该位置 1），如果刹车输入为无效极性时，MOE 位在发生下一个更新事件时自动置 1，恢复运行模式（即继续输出 PWM）。因此当停止给 PA6 接入高电平（拔掉之前连接的杜邦线），PWM 会自动恢复输出。

4.3.5 高级定时器输入捕获实验

本小节我们来学习使用高级定时器的输入捕获模式。

输入捕获模式可以用来测量脉冲宽度或者测量频率。我们以测量脉宽为例，用一个简图来说明输入捕获的原理，如图 4.3.5.1 所示：

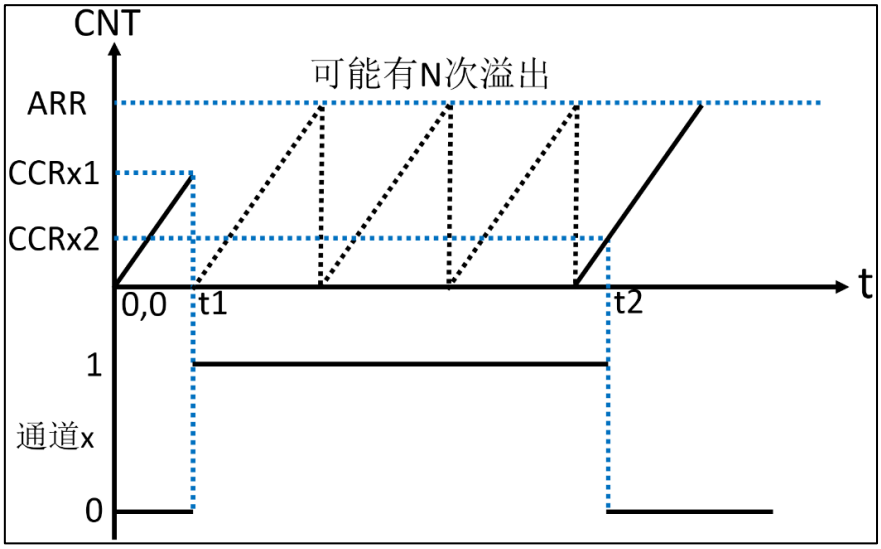


图 4.3.5.1 输入捕获脉宽测量原理

图 4.3.5.1 中，t1 到 t2 的时间段，就是我们需要测量的高电平时间。测量方法如下：假如定时器工作在递增计数模式，首先设置定时器通道 x 为上升沿捕获，这样在 t1 时刻上升沿到来时，就会发生捕获事件。这里我们还会打开捕获中断，所以捕获事件发生就意味着捕获中断也会发生。在捕获中断里将计数器值清零，并设置通道 x 为下降沿捕获，这样 t2 时刻下降沿到来时，就会发生捕获事件和捕获中断。捕获事件发生时，计数器的值会被锁存到捕获/比较寄存器中（比如通道 1 对应的是 CCR1 寄存器）。那么在捕获中断里，我们读取捕获/比较寄存器就可以获取到高电平脉冲时间内，计数器计数的个数，从而可以算出高电平脉冲的时间。这里是假设定时器没有溢出为前提的。

实际上，t1 到 t2 时间段，定时器可能会产生 N 次溢出，这就需要我们对定时器溢出做相应的处理，防止高电平太长，导致测量出错。在 t1 到 t2 时间段，假设定时器溢出 N 次，那么高电平脉冲时间内，计数器计数的个数计算方法为：N*（ARR+1）+CCRx2，CCRx2 表示 t2 时间点，捕获/比较寄存器的值。经过计算得到高电平脉宽时间内计数器计数个数后，用这个个数乘以计数器的计数周期，就可得到高电平持续的时间。就是输入捕获测量高电平脉宽时间的整个过程。

STM32F407 的定时器除了 TIM6 和 TIM7，其他定时器都有输入捕获功能。输入捕获，简单的说就是通过检测 TIMx_CHy 上的边沿信号，在边沿信号发生跳变（比如上升沿/下降沿）时，会发生捕获事件，将当前定时器的值（TIMx_CNT）锁存到对应通道的捕获/比较寄存器（TIMx_CCRy）里，完成一次捕获。同时还可以配置捕获事件发生时是否触发捕获中断/DMA。另外还要考虑测量的过程中是否可能发生定时器溢出，如果可能溢出，还要做溢出处理。

4.3.5.1 TIM1/TIM8 寄存器

高级定时器输入捕获实验需要用到的寄存器有：TIMx_ARR、TIMx_PSC、TIMx_CCMR1、TIMx_CCER、TIMx_DIER、TIMx_CR1、TIMx_CCR1 这些寄存器在前面的章节都有提到，在这里只需针对性的介绍。

- **捕获/比较模式寄存器 1/2（TIMx_CCMR1/2）**

该寄存器我们在 PWM 输出实验时讲解了他作为输出功能的配置，现在重点学习输入捕获模式的配置。因为本实验我们用到定时器 1 通道 1 输入，所以我们要看 TIMx_CCMR1 寄存，其描述如图 4.3.5.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 7:4 **IC1F**：输入捕获 1 滤波器 (Input capture 1 filter)

此位域可定义 TI1 输入的采样频率和适用于 TI1 的数字滤波器带宽。数字滤波器由事件计数器组成，每 N 个事件才视为一个有效边沿：

0000：无滤波器，按 f_{DTS} 频率进行采样
0001：$f_{SAMPLING}=f_{CK_INT}$，N=2
0010：$f_{SAMPLING}=f_{CK_INT}$，N=4
0011：$f_{SAMPLING}=f_{CK_INT}$，N=8
0100：$f_{SAMPLING}=f_{DTS}/2$，N=6
0101：$f_{SAMPLING}=f_{DTS}/2$，N=8
0110：$f_{SAMPLING}=f_{DTS}/4$，N=6
0111：$f_{SAMPLING}=f_{DTS}/4$，N=8
1000：$f_{SAMPLING}=f_{DTS}/8$，N=6
1001：$f_{SAMPLING}=f_{DTS}/8$，N=8
1010：$f_{SAMPLING}=f_{DTS}/16$，N=5
1011：$f_{SAMPLING}=f_{DTS}/16$，N=6
1100：$f_{SAMPLING}=f_{DTS}/16$，N=8
1101：$f_{SAMPLING}=f_{DTS}/32$，N=5
1110：$f_{SAMPLING}=f_{DTS}/32$，N=6
1111：$f_{SAMPLING}=f_{DTS}/32$，N=8

注意：在当前硅版本中，当 ICxF[3:0]= 1、2 或 3 时，将用 CK_INT 代替公式中的 f_{DTS}。

位 3:2 **IC1PSC**：输入捕获 1 预分频器 (Input capture 1 prescaler)

此位域定义 CC1 输入 (IC1) 的预分频比。
只要 CC1E=0（TIMx_CCER 寄存器），预分频器便立即复位。
00：无预分频器，捕获输入上每检测到一个边沿便执行捕获
01：每发生 2 个事件便执行一次捕获
10：每发生 4 个事件便执行一次捕获
11：每发生 8 个事件便执行一次捕获

位 1:0 **CC1S**：捕获/比较 1 选择 (Capture/Compare 1 selection)

此位域定义通道方向（输入/输出）以及所使用的输入。
00：CC1 通道配置为输出
01：CC1 通道配置为输入，IC1 映射到 TI1 上
10：CC1 通道配置为输入，IC1 映射到 TI2 上
11：CC1 通道配置为输入，IC1 映射到 TRC 上。此模式仅在通过 TS 位（TIMx_SMCR 寄存器）选择内部触发输入时有效

注意：仅当通道关闭时（TIMx_CCER 中的 CC1E = 0），才可向 CC1S 位写入数据。

图 4.3.5.1.1 TIMx_CCMR1 寄存器

该寄存器在输入模式和输出模式下，功能是不一样，所以需要看准不同模式的描述，请打开手册查看。TIMx_CCMR1 寄存器对应于通道 1 和通道 2 的设置，CCMR2 寄存器对应通道 3 和通道 4。如：TIMx_CCMR1 寄存器位[7：0]用于捕获/比较通道 1 的控制，而位[15：8]则用于捕获/比较通道 2 的控制。我们用到定时器 1 通道 1 输入，所以需要配置 TIMx_CCMR1 的位[7:0]。

其中 CC1S[1:0]，这两个位用于 CCR1 的通道配置，这里我们设置 IC1S[1:0]=01，也就是配置 IC1 映射在 TI1 上。

输入捕获 1 预分频器 IC1PSC[1:0]，这个比较好理解。我们是 1 次边沿就触发 1 次捕获，所以选择 00 就行了。

输入捕获 1 滤波器 IC1F[3:0]，这个用来设置输入采样频率和数字滤波器长度。其中，f_{CK_INT} 是定时器时钟源频率，按照例程的配置为 168Mhz，而 f_{DTS} 则是根据 TIMx_CR1 的 CKD[1:0] 的设置来确定的，如果 CKD[1:0]设置为 00，那么 $f_{DTS}=f_{CK_INT}$。N 值采样次数，举个简单的例子：假设 IC1F[3:0]=0011，并设置 IC1 映射到 TI1 上。表示以 f_{CK_INT} 为采样频率，当连续 8 次都是采样到 TI1 为高电平或者低电平，滤波器才输出一个有效输出边沿。当 8 次采样中有高有低，那就保持原来的输出，这样可以滤除高频干扰信号，从而达到滤波的效果。这里，我们不做滤波处理，所以设置 IC1F[3:0]=0000。

- **捕获/比较使能寄存器（TIMx_CCER）**

TIM1/TIM8 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 4.3.5.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

位 1 **CC1P**：*捕获/比较 1 输出极性 (Capture/Compare 1 output Polarity)*。

CC1 通道配置为输出：

0：OC1 高电平有效

1：OC1 低电平有效

CC1 通道配置为输入：

CC1NP/CC1P 位可针对触发或捕获操作选择 TI1FP1 和 TI2FP1 的极性。

00：非反相/上升沿触发

电路对 TIxFP1 上升沿敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 未反相（在门控模式或编码器模式下执行触发操作）。

01：反相/下降沿触发

电路对 TIxFP1 下降沿敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 反相（在门控模式或编码器模式下执行触发操作）。

10：保留，不使用此配置。

11：非反相/上升沿和下降沿均触发

电路对 TIxFP1 上升沿和下降沿都敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 未反相（在门控模式下执行触发操作）。编码器模式下不得使用此配置。

位 0 **CC1E**：*捕获/比较 1 输出使能 (Capture/Compare 1 output enable)*。

CC1 通道配置为输出：

0：关闭——OC1 未激活

1：开启——在相应输出引脚上输出 OC1 信号

CC1 通道配置为输入：

此位决定了是否可以实际将计数器值捕获到输入捕获/比较寄存器 1 (TIMx_CCR1) 中。

0：禁止捕获

1：使能捕获

图 4.3.5.1.2 TIMx_CCER 寄存器

我们要用到这个寄存器的最低 2 位，CC1E 和 CC1P 位。要使能输入捕获，必须设置 CC1E=1，而 CC1P 则根据自己的需要来配置。我们这里是保留默认设置值 0，即高电平触发捕获。

- **TIMx DMA/中断使能寄存器（TIMx_DIER）**

TIM1/TIM8 的 DMA/中断使能寄存器描述如图 4.3.5.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

位 0 **UIE**：更新中断使能 (Update interrupt enable)

0：禁止更新中断

1：使能更新中断

图 4.3.5.1.3 TIMx_DIER 寄存器

我们需要用到中断来处理捕获数据，所以必须开启通道 1 的捕获比较中断，即 CC1IE 设置为 1。同时我们还需要在定时器溢出中断中累计定时器溢出的次数，所以还需要使能定时器的更新中断，即 UIE 置 1。

控制寄存器：TIMx_CR1，我们只用到了它的最低位，也就是用来使能定时器的。

最后再来看看捕获/比较寄存器 1：TIMx_CCR1，该寄存器用来存储捕获发生时，TIMx_CNT 的值，我们从 TIMx_CCR1 就可以读出通道 1 捕获发生时刻的 TIMx_CNT 值，通过两次捕获（一次上升沿捕获，一次下降沿捕获）的差值，就可以计算出高电平脉冲的宽度（注意，对于脉宽太长的情况，还要计算定时器溢出的次数）。

4.3.5.2 硬件设计

1. 例程功能

1、本实验我们基于前面的高级定时器 PWM 输出实验来实现，使用 TIM8_CH1（PI5）输出 PWM 波形，然后使用 TIM1_CH1（PA8）来做输入捕获，捕获 PI5 上的高电平脉宽，并将脉宽时间通过串口打印出来，这里能测试的最长时间为：4194303 us。

2、LED0 闪烁指示程序运行。

2. 硬件资源

1) LED 灯：
LED0 - PE0
2) 定时器 8，使用 TIM8 通道 1，将 PI5 复用到 TIM8_CH1。
3) 定时器 1，使用 TIM1 通道 1，将 PA8 复用到 TIM1_CH1。

3. 原理图

定时器属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们借助 TIM8_CH1 输出 PWM 波形作为输入脉冲源并通过串口上位机来监测定时器输入捕获的情况。

4.3.5.3 程序设计

4.3.5.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面已经介绍了部分，这里我们再介绍几个本实验用到的函数。

1. HAL_TIM_IC_Init 函数

定时器的输入捕获模式初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_IC_Init(TIM_HandleTypeDef *htim);
```

- **函数描述：**
  用于初始化定时器的输入捕获模式。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，基本定时器的时候已经介绍。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。
- **注意事项：**
  与 PWM 输出实验一样，当使用定时器做输入捕获功能时，在 HAL 库中并不使用定时器初始化函数 HAL_TIM_Base_Init 来实现，而是使用输入捕获特定的定时器初始化函数 HAL_TIM_IC_Init。该函数内部还会调用输入捕获初始化回调函数 HAL_TIM_IC_MspInit 来初始化输入通道对应的 GPIO（复用），以及输入捕获相关的配置。

2. HAL_TIM_IC_ConfigChannel 函数

定时器的输入捕获通道设置初始化函数。其声明如下：

```
HAL_StatusTypeDef HAL_TIM_IC_ConfigChannel(TIM_HandleTypeDef *htim,
                                           TIM_IC_InitTypeDef *sConfig, uint32_t Channel);
```

- **函数描述：**
  该函数用于设置定时器的输入捕获通道。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。
  形参 2 是 TIM_IC_InitTypeDef 结构体类型指针变量，用于配置定时器的输入捕获参数。
  重点了解一下 TIM_IC_InitTypeDef 结构体指针类型，其定义如下：

```
typedef struct
{
  uint32_t ICPolarity;      /* 输入捕获触发方式选择，比如上升、下降和双边沿捕获 */
  uint32_t ICSelection;     /* 输入捕获选择，用于设置映射关系 */
  uint32_t ICPrescaler;     /* 输入捕获分频系数 */
  uint32_t ICFilter;        /* 输入捕获滤波器设置 */
} TIM_IC_InitTypeDef;
```

该结构体成员我们现在介绍一下。成员变量 ICPolarity 用来设置输入信号的有效捕获极性，取值范围为：TIM_ICPOLARITY_RISING(上升沿捕获)，TIM_ICPOLARITY_FALLING（下降沿捕获）和 TIM_ICPOLARITY_BOTHEDGE（双边沿捕获）。成员变量 ICSelection 用来设置映射关系，我们配置 IC1 直接映射在 TI1 上，选择 TIM_ICSELECTION_DIRECTTI（另外

还有两个输入通道 TIM_ICSELECTION_INDIRECTTI 和 TIM_ICSELECTION_TRC）。成员变量 ICPrescaler 用来设置输入捕获分频系数，可以设置为 TIM_ICPSC_DIV1（不分频），TIM_ICPSC_DIV2（2 分频），TIM_ICPSC_DIV4（4 分频）以及 TIM_ICPSC_DIV8（8 分频），本实验需要设置为不分频，所以选值为 TIM_ICPSC_DIV1。成员变量 ICFilter 用来设置滤波器长度，这里我们不使用滤波器，所以设置为 0。

形参 3 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

3. HAL_TIM_IC_Start_IT 函数

启动定时器输入捕获模式函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_IC_Start_IT(TIM_HandleTypeDef *htim,
                                      uint32_t Channel);
```

- **函数描述：**

用于启动定时器的输入捕获模式，且开启输入捕获中断。

- **函数形参：**

形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。

形参 2 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

- **注意事项：**

如果我们不需要开启输入捕获中断，只是开启输入捕获功能，HAL 库函数为：

```
HAL_StatusTypeDef HAL_TIM_IC_Start(TIM_HandleTypeDef *htim, uint32_t Channel);
```

定时器输入捕获模式配置步骤

1）开启 TIMx 和输入通道的 GPIO 时钟，配置该 IO 口的复用功能输入

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输入。本实验我们默认用到定时器 1 通道 1，对应 IO 是 PA8，它们的时钟开启方法如下：

```
__HAL_RCC_TIM1_CLK_ENABLE();            /* 使能定时器 1 */
__HAL_RCC_GPIOA_CLK_ENABLE();           /* 开启 GPIOA 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的输入捕获功能时，我们调用的是 HAL_TIM_IC_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_IC_MspInit 函数，我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能以及优先级设置等代码。

3）设置 TIMx_CHy 的输入捕获模式，开启输入捕获

在 HAL 库中，定时器的输入捕获模式是通过 HAL_TIM_IC_ConfigChannel 函数来设置定时器某个通道为输入捕获通道，包括映射关系，输入滤波和输入分频等。

4）使能定时器更新中断，开启捕获功能以及捕获中断，配置定时器中断优先级

通过__HAL_TIM_ENABLE_IT 函数使能定时器更新中断。

通过 HAL_TIM_IC_Start_IT 函数使能定时器并开启捕获功能以及捕获中断。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

因为我们要捕获的是高电平信号的脉宽，所以，第一次捕获是上升沿，第二次捕获时下降沿，必须在捕获上升沿之后，设置捕获边沿为下降沿，同时，如果脉宽比较长，那么定时器就会溢出，对溢出必须做处理，否则结果就不准了。

5）编写中断服务函数

定时器 1 做输入捕获实验，需要用到两个中断服务函数：TIM1_UP_TIM10_IRQHandler 和 TIM1_CC_IRQHandler，前者是更新中断服务函数，我们用来记录溢出次数，后者是捕获中断服务函数，用来处理捕获的事件。HAL 库为了使用方便，提供了一个定时器中断通用处理

函数 HAL_TIM_IRQHandler，该函数会调用一些定时器相关的回调函数，用于给用户处理定时器中断到了之后，需要处理的程序。本实验我们除了用到更新（溢出）中断回调函数 HAL_TIM_PeriodElapsedCallback 之外，还要用到捕获中断回调函数 HAL_TIM_IC_CaptureCallback。详见本实验例程源码。

4.3.5.3.2 程序流程图

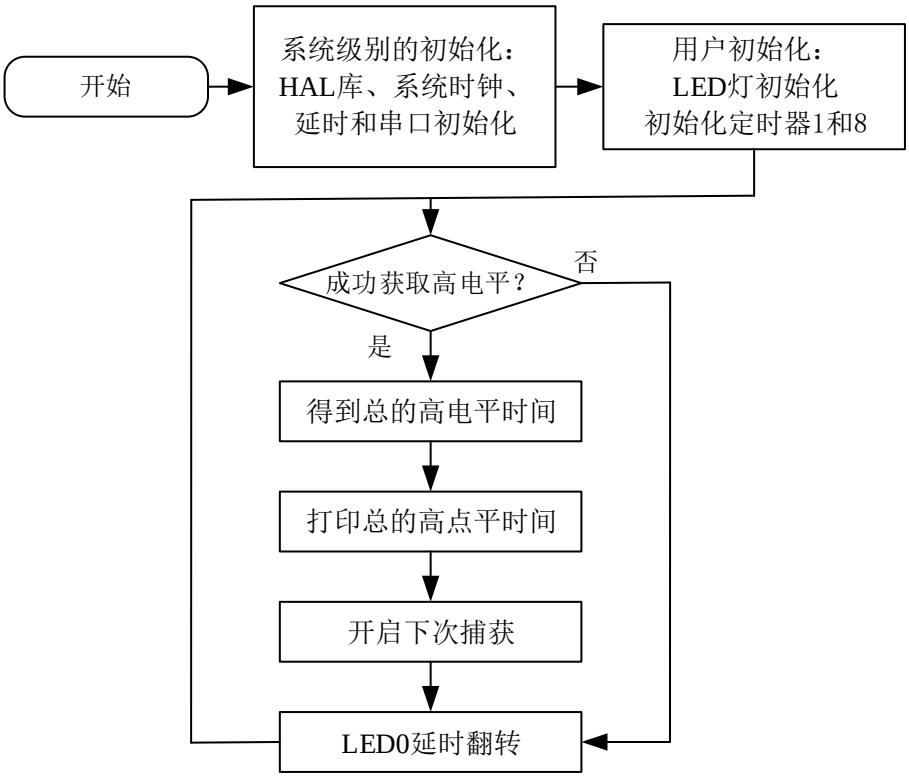


图 4.3.5.3.2.1 高级定时器输入捕获实验程序流程图

4.3.5.3.3 程序解析

本实验需要利用 TIM8_CH1 来输出 PWM 作为输入捕获的脉冲源，关于 PWM 输出的部分我们这里不再赘述，大家可以回顾 4.3.3 章节的内容。

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义：

```
/****************** 第一部分 高级定时器 PWM 输出实验相关宏定义 ********************/

/* TIMX PWM 输出定义 */

#define ATIM_TIMX_PWM_CHY_GPIO_PORT          GPIOI
#define ATIM_TIMX_PWM_CHY_GPIO_PIN           GPIO_PIN_5
#define ATIM_TIMX_PWM_CHY_GPIO_CLK_ENABLE()
do{   __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0) /* PI 口时钟使能 */

/* TIMX 引脚复用设置
 * 因为 PI5 默认并不是 TIM8 的功能脚，必须开启复用，PI5 才能用作 TIM8 的 PWM 输出引脚.
 */
#define ATIM_TIMX_PWM_CHY_GPIO_AF          GPIO_AF3_TIM8

#define ATIM_TIMX_PWM                      TIM8
#define ATIM_TIMX_PWM_CHY                  TIM_CHANNEL_1 /* 通道 Y,1<= Y <=4 */
```

```
#define ATIM_TIMX_PWM_CHY_CLK_ENABLE()
        do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)      /* TIM8 时钟使能 */

/****************** 第二部分 高级定时器 输入捕获实验相关宏定义 ******************/

 /* TIMX 输入捕获定义
  * 这里的输入捕获使用定时器 TIM1_CH1
  */
#define ATIM_TIMX_CAP_CHY_GPIO_PORT             GPIOA
#define ATIM_TIMX_CAP_CHY_GPIO_PIN              GPIO_PIN_8
#define ATIM_TIMX_CAP_CHY_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)     /* PA 口时钟使能 */

/* TIMX 引脚复用设置
 * 因为 PA8 默认并不是 TIM1 的功能脚, 必须开启复用, PA8 才能用作 TIM1 的输入捕获引脚
 */
#define ATIM_TIMX_CAP_CHY_GPIO_AF       GPIO_AF1_TIM1

#define ATIM_TIMX_CAP                   TIM1
#define ATIM_TIMX_CAP_IRQn              TIM1_CC_IRQn
#define ATIM_TIMX_CAP_IRQHandler        TIM1_CC_IRQHandler
#define ATIM_TIMX_INT_IRQn              TIM1_UP_TIM10_IRQn
#define ATIM_TIMX_INT_IRQHandler        TIM1_UP_TIM10_IRQHandler
#define ATIM_TIMX_CAP_CHY               TIM_CHANNEL_1    /* 通道 Y,1<= Y <=4 */
#define ATIM_TIMX_CAP_CHY_CLK_ENABLE()
        do{ __HAL_RCC_TIM1_CLK_ENABLE(); }while(0)      /* TIM1 时钟使能 */
```

可以把上面的宏定义分成两部分，第一部分是定时器 8 输出通道 1 对应的 IO 口的宏定义，第二部分则是定时器 1 输入通道 1 的相应宏定义。

下面来看 atim.c 的程序，我们只看输入捕获的部分，首先是高级定时器输入捕获初始化函数。

```
/**
 * @brief     高级定时器 TIMX 通道 Y 输入捕获 初始化函数
 * @note
 *            高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *            高级定时器时钟 = 168Mhz
 *            定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *            Ft=定时器工作频率,单位:Mhz
 *
 * @param     arr: 自动重装值
 * @param     psc: 时钟预分频数
 * @retval    无
 */
void atim_timx_cap_chy_init(uint32_t arr, uint16_t psc)
{
    TIM_IC_InitTypeDef timx_ic_cap_chy = {0};

    g_timx_cap_chy_handle.Instance = ATIM_TIMX_CAP;         /* 定时器 x */
    g_timx_cap_chy_handle.Init.Prescaler = psc;             /* 定时器分频 */
    g_timx_cap_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数 */
    g_timx_cap_chy_handle.Init.Period = arr;                /* 自动重装载值 */
    HAL_TIM_IC_Init(&g_timx_cap_chy_handle);

    timx_ic_cap_chy.ICPolarity = TIM_ICPOLARITY_RISING;     /* 上升沿捕获 */
    timx_ic_cap_chy.ICSelection = TIM_ICSELECTION_DIRECTTI;/* 映射到 TI1 上 */
    timx_ic_cap_chy.ICPrescaler = TIM_ICPSC_DIV1;           /* 不分频 */
    timx_ic_cap_chy.ICFilter = 0;                           /* 不滤波 */
    HAL_TIM_IC_ConfigChannel(&g_timx_cap_chy_handle, &timx_ic_cap_chy,
                             ATIM_TIMX_CAP_CHY);            /* 配置 TIM1 通道 1 */
    /* 使能更新中断 */
    __HAL_TIM_ENABLE_IT(&g_timx_cap_chy_handle, TIM_IT_UPDATE);
    /* 开启捕获 TIM1 的通道 1 */
```

```
    HAL_TIM_IC_Start_IT(&g_timx_cap_chy_handle, ATIM_TIMX_CAP_CHY);
}
```

HAL_TIM_IC_Init 初始化定时器的基础工作参数，如：ARR 和 PSC 等，第二部分则是调用 HAL_TIM_IC_ConfigChannel 函数配置输入捕获通道映射关系，滤波和分频等。最后是使能更新中断和使能通道输入以及定时器捕获中断。通道对应的 IO、时钟开启和 NVIC 的初始化都在 HAL_TIM_IC_MspInit 函数里编写，其定义如下：

```
/**
 * @brief       高级定时器输入捕获初始化接口
 *              HAL 库调用的接口，用于配置不同的输入捕获
 * @param       htim:定时器句柄
 * @note        此函数会被 HAL_TIM_IC_Init()调用
 * @retval      无
 */
void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == ATIM_TIMX_CAP)                        /* 输入通道捕获 */
    {
        GPIO_InitTypeDef gpio_init_struct;
        ATIM_TIMX_CAP_CHY_CLK_ENABLE();                         /* 使能 TIMx 时钟 */
        ATIM_TIMX_CAP_CHY_GPIO_CLK_ENABLE();                    /* 开启 IO 的时钟 */

        gpio_init_struct.Pin = ATIM_TIMX_CAP_CHY_GPIO_PIN;      /* 捕获的 GPIO 口 */
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;                /* 复用推挽输出 */
        gpio_init_struct.Pull = GPIO_PULLDOWN;                  /* 下拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
        gpio_init_struct.Alternate = ATIM_TIMX_CAP_CHY_GPIO_AF; /* 端口复用 */
        HAL_GPIO_Init(ATIM_TIMX_CAP_CHY_GPIO_PORT, &gpio_init_struct);

        HAL_NVIC_SetPriority(ATIM_TIMX_INT_IRQn, 2, 3);
        HAL_NVIC_EnableIRQ(ATIM_TIMX_INT_IRQn);
        HAL_NVIC_SetPriority(ATIM_TIMX_CAP_IRQn, 0, 0);
        HAL_NVIC_EnableIRQ(ATIM_TIMX_CAP_IRQn);
    }
}
```

该函数调用 HAL_GPIO_Init 函数初始化定时器输入通道对应的 IO，并且开启 GPIO 时钟，使能定时器。其中要注意 IO 口复用功能的选择一定要选对了。最后配置中断抢占优先级和响应优先级，以及打开定时器中断。

通过上面的两个函数输入捕获的初始化就完成了，下面先来介绍两个变量。

```
/* 输入捕获状态(g_timxchy_cap_sta)
 * [7]  :0,没有成功的捕获;1,成功捕获到一次.
 * [6]  :0,还没捕获到高电平;1,已经捕获到高电平了.
 * [5:0]:捕获高电平后溢出的次数,最多溢出 63 次,所以最长捕获值= 63*65536+65535 = 4194303
 *      注意:为了通用,我们默认 ARR 和 CCRy 都是 16 位寄存器,对于 32 位的定时器(如:TIM5),
 *           也只按 16 位使用
 *      按 1us 的计数频率,最长溢出时间为:4194303 us, 约 4.19 秒
 *     (说明一下：正常 32 位定时器来说,1us 计数器加 1,溢出时间:4294 秒)
 */
uint8_t g_timxchy_cap_sta = 0;      /* 输入捕获状态 */
uint16_t g_timxchy_cap_val =0 ;     /* 输入捕获值 */
```

这两个变量用于辅助实现高电平捕获。其中 g_timxchy_cap_sta，是用来记录捕获状态，(这个变量，我们把它当成一个寄存器那样来使用)。对其各位赋予状态含义，描述如下表所示：

g_timxchy_cap_sta		
bit7	bit6	bit5~0
捕获完成标志	捕获到高电平标志	捕获高电平后定时器溢出的次数

表 4.3.5.3.3.1 g_timxchy_cap_sta 各位描述

变量 g_timxchy_cap_sta 的位[5:0]是用于记录捕获高电平定时器溢出次数，总共 6 位，所

以最多可以记录溢出的次数为 2 的 6 次方减一次，即 63 次。

变量 g_timxchy_cap_val，则用来记录捕获到下降沿的时候，TIM1_CNT 寄存器的值。

下面开始看中断服务函数的逻辑程序，HAL_TIM_IRQHandler 函数会调用下面两个回调函数，我们的逻辑代码就是放在回调函数里，函数定义如下：

```
/**
 * @brief       定时器输入捕获中断处理回调函数
 * @param       htim:定时器句柄指针
 * @note        该函数在 HAL_TIM_IRQHandler 中会被调用
 * @retval      无
 */
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    if ((g_timxchy_cap_sta & 0X80) == 0)         /* 还没成功捕获 */
    {
        if (g_timxchy_cap_sta & 0X40)            /* 捕获到一个下降沿 */
        {
            g_timxchy_cap_sta |= 0X80;           /* 标记成功捕获到一次高电平脉宽 */
            g_timxchy_cap_val =
            HAL_TIM_ReadCapturedValue(&g_timx_cap_chy_handler,
                                      ATIM_TIMX_CAP_CHY); /* 获取当前的捕获值 */
            TIM_RESET_CAPTUREPOLARITY(&g_timx_cap_chy_handler,
                                      ATIM_TIMX_CAP_CHY); /* 一定要先清除原来的设置 */
            TIM_SET_CAPTUREPOLARITY(&g_timx_cap_chy_handler, ATIM_TIMX_CAP_CHY,
                        TIM_ICPOLARITY_RISING);  /* 配置 TIMx 通道 1 上升沿捕获 */
        }
        else                                     /* 还未开始,第一次捕获上升沿 */
        {
            g_timxchy_cap_sta = 0;               /* 清空 */
            g_timxchy_cap_val = 0;
            g_timxchy_cap_sta |= 0X40;           /* 标记捕获到了上升沿 */
            __HAL_TIM_DISABLE(&g_timx_cap_chy_handler);       /* 关闭定时器 x */
            __HAL_TIM_SET_COUNTER(&g_timx_cap_chy_handler,0); /* 计数器清零 */
            TIM_RESET_CAPTUREPOLARITY(&g_timx_cap_chy_handler,
                           ATIM_TIMX_CAP_CHY);   /* 一定要先清除原来的设置！！*/
            TIM_SET_CAPTUREPOLARITY(&g_timx_cap_chy_handler, ATIM_TIMX_CAP_CHY,
                      TIM_ICPOLARITY_FALLING);   /* 定时器 x 通道 1 设置为下降沿捕获 */
            __HAL_TIM_ENABLE(&g_timx_cap_chy_handler);        /* 使能定时器 x */
        }
    }
}

/**
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @note        此函数会被定时器中断函数共同调用的
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == ATIM_TIMX_CAP)
    {
        if ((g_timxchy_cap_sta & 0X80) == 0)             /* 还没成功捕获 */
        {
            if (g_timxchy_cap_sta & 0X40)                /* 已经捕获到高电平了 */
            {
                if ((g_timxchy_cap_sta & 0X3F) == 0X3F)  /* 高电平太长了 */
                {
                    TIM_RESET_CAPTUREPOLARITY(&g_timx_cap_chy_handle,
                                              ATIM_TIMX_CAP_CHY); /* 一定要先清除原来的设置 */
                    /* 配置 TIMx 通道 1 上升沿捕获 */
                    TIM_SET_CAPTUREPOLARITY(&g_timx_cap_chy_handle,
```

```
                                ATIM_TIMX_CAP_CHY, TIM_ICPOLARITY_RISING);
                g_timxchy_cap_sta |= 0X80;          /* 标记成功捕获了一次 */
                g_timxchy_cap_val = 0XFFFF;
            }
            else                                    /* 累计定时器溢出次数 */
            {
                g_timxchy_cap_sta++;
            }
        }
    }
}
```

现在我们来介绍一下，捕获高电平脉宽的思路：首先，设置 TIM5_CH1 捕获上升沿，然后等待上升沿中断到来，当捕获到上升沿中断，此时如果 g_timxchy_cap_sta 的第 6 位为 0，则表示还没有捕获到新的上升沿，就先把 g_timxchy_cap_sta、g_timxchy_cap_val 和 TIM5_CNT 寄存器等清零，然后再设置 g_timxchy_cap_sta 的第 6 位为 1，标记捕获到高电平，最后设置为下降沿捕获，等待下降沿到来。如果等待下降沿到来期间，定时器发生了溢出，就用 g_timxchy_cap_sta 变量对溢出次数进行计数，当最大溢出次数来到的时候，就强制标记捕获完成，并配置定时器通道上升沿捕获。当下降沿到来的时候，先设置 g_timxchy_cap_sta 的第 7 位为 1，标记成功捕获一次高电平，然后读取此时的定时器值到 g_timxchy_cap_val 里面，最后设置为上升沿捕获，回到初始状态。

这样，我们就完成一次高电平捕获了，只要 g_timxchy_cap_sta 的第 7 位一直为 1，那么就不会进行第二次捕获，我们在 main 函数处理完捕获数据后，将 g_timxchy_cap_sta 置零，就可以开启第二次捕获。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint32_t temp = 0;
    uint8_t t = 0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    /* PWM 周期为 500ms，占空比为 50%，也就是高电平时间 = 250ms = 250000us */
    atim_timx_pwm_chy_init(5000 - 1, 16800 - 1);
    atim_timx_cap_chy_init(0xFFFF, 168 - 1);    /* 以 1Mhz 的频率计数 捕获 */

    while (1)
    {
        if (g_timxchy_cap_sta & 0X80)           /* 成功捕获到了一次高电平 */
        {
            temp = g_timxchy_cap_sta & 0X3F;
            temp *= 0xFFFF;                     /* 溢出时间总和 */
            temp += g_timxchy_cap_val;          /* 得到总的高电平时间 */
            printf("HIGH:%d us\r\n", temp);     /* 打印总的高点平时间 */
            g_timxchy_cap_sta = 0;              /* 开启下一次捕获*/
        }

        t++;

        if (t > 20)                             /* 200ms 进入一次 */
        {
            t = 0;
            LED0_TOGGLE();                      /* LED0 闪烁 ,提示程序运行 */
        }
        delay_ms(10);
    }
}
```

首先看到 atim_timx_pwm_chy_init(5000 - 1, 16800 - 1)这个语句，这里初始化了定时器 8 的 PWM 输出，PWM 的周期为 500ms，占空比为 50%，也就是一个周期内高电平时间 =250ms = 250000us。

再看到 atim_timx_cap_chy_init(0XFFFF, 168 - 1)这个语句，这两个形参分别设置自动重载寄存器的值为 65535，以及定时器预分频系数为 167。定时器 1 是 16 位的计数器，这里的自动重载寄存器的值只设置为 65535，最大值，我们只用到 16 位计数器的计数值。预分频系数，我们设置为 168 分频，定时器 1 的时钟是 2 倍的 APB2，即 168MHZ，可以得到计数器的计数频率是 1MHZ，即 1us 计数一次，所以我们的捕获时间精度是 1us。这里可以知道定时器的溢出时间是 65536us。

While(1)无限循环通过判断 g_timxchy_cap_sta 的第 7 位，来获知有没有成功捕获到一次高电平，如果成功捕获，先计算总的高电平时间，再通过串口传输到电脑。

4.3.5.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，我们再打开串口调试助手，选择对应的串口端口，我这边是 COM3，然后把 PI5 通过杜邦线接到 PA8，就可以看到串口打印 TIM8_CH1(PI5)输出的 PWM 脉冲的高电平持续时间，如图 4.3.5.4.1 所示：

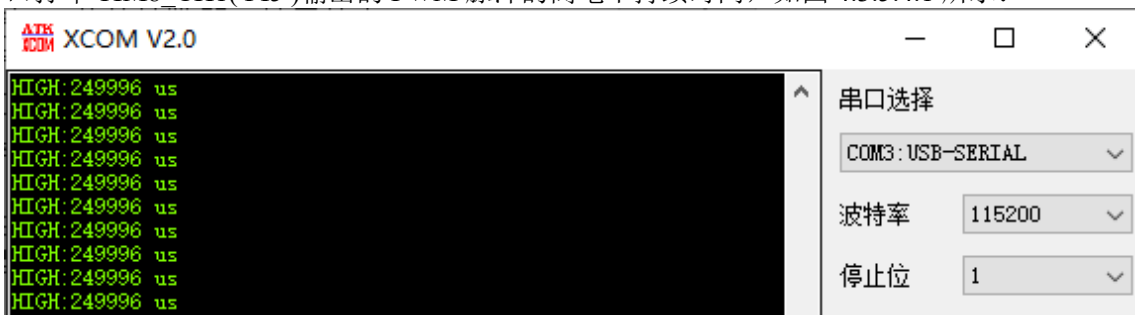


图 4.3.5.4.1 打印捕获到的高电平时间

我们定时器 8 输出的 PWM 周期是 500ms，占空比为 50%，所以高低电平都为 250ms，由上图可知捕获到的高电平时间在 250ms 左右，存在微小误差是正常现象。

4.3.6 高级定时器 PWM 输入实验

本小节我们来学习使用高级定时器 PWM 输入模式，此模式是输入捕获模式的一个特例。PWM 输入模式经常被应用于测量 PWM 脉宽和频率。PWM 输入模式在《STM32F4xx 参考手册_V4（中文版）.pdf》手册 347 页有详细的文字描述。下面我们结合这些文字，配合高级定时器框图给大家介绍 PWM 输入的工作原理。

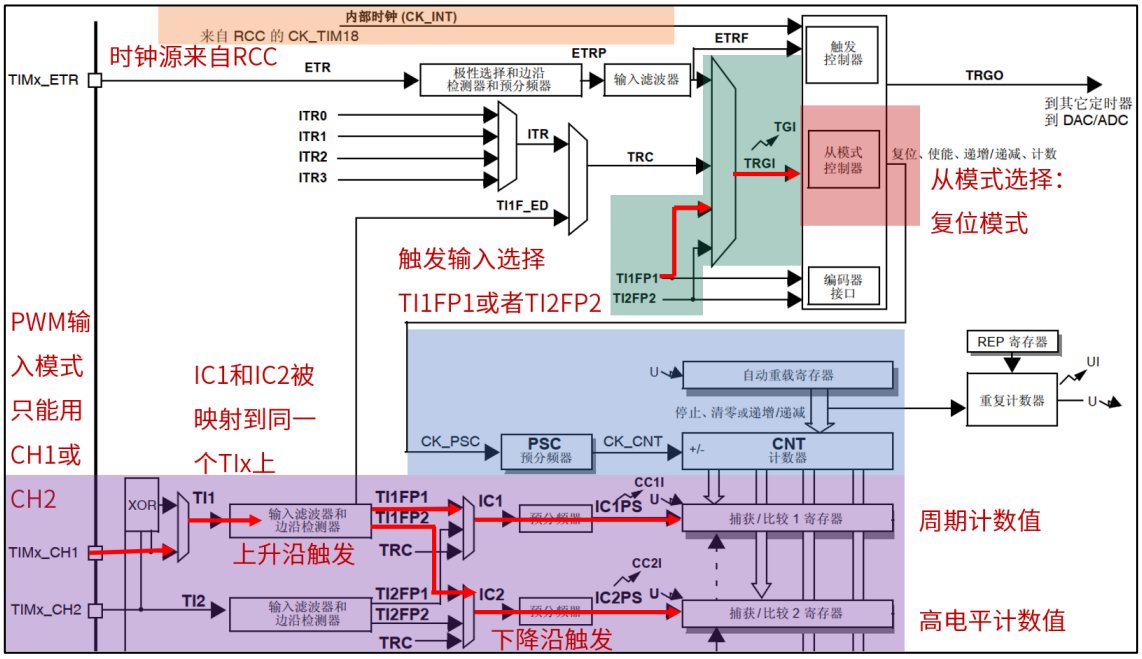


图 4.3.6.1 PWM 输入模式工作原理示意图

第一，确定定时器时钟源。本实验中我们使用内部时钟（CK_INT），F4 系列高级定时器挂载在 APB2 总线上，按照 sys_stm32_clock_init 函数的配置，定时器时钟频率等于 APB2 总线时钟频率，即 168MHz。计数器的计数频率确定了测量的精度。

第二，确定 PWM 输入的通道。PWM 输入模式下测量 PWM，PWM 信号输入只能从通道 1（CH1）或者通道 2（CH2）输入。

第三，确定 IC1 和 IC2 的捕获边沿。这里以通道 1（CH1）输入 PWM 为例，一般我们习惯设置 IC1 捕获边沿为上升沿捕获，IC2 捕获边沿为下降沿捕获。

第四，选择触发输入信号（TRGI）。这里也是以通道 1（CH1）输入 PWM 为例，那么我们就应该选择 TI1FP1 为触发输入信号。如果是通道 2（CH2）输入 PWM，那就选择 TI2FP2 为触发输入信号。可以看到这里并没有对应通道 3（CH3）或者通道 4（CH4）的触发输入信号，所以我们只选择通道 1 或者通道 2 作为 PWM 输入的通道。

第五，从模式选择：复位模式。复位模式的作用是：在出现所选触发输入 (TRGI) 上升沿时，重新初始化计数器并生成一个寄存器更新事件。

第六，读取一个 PWM 周期内计数器的计数个数，以及高电平期间的计数个数，再结合计数器的计数周期(即计一个数的时间)，最终通过计算得到输入的 PWM 周期和占空比等参数。以通道 1（CH1）输入 PWM，设置 IC1 捕获边沿为上升沿捕获，IC2 捕获边沿为下降沿捕获为例，那么 CCR1 寄存器的值+1 就是 PWM 周期内计数器的计数个数，CCR2 寄存器的值+1 就是 PWM 高电平期间计数器的计数个数。通过这两个值就可以计算出 PWM 的周期或者占空比等参数。

再举个例子，以通道 1（CH1）输入 PWM，设置 IC1 捕获边沿为下降沿捕获，IC2 捕获边沿为上升沿捕获为例，那么 CCR1 寄存器的值+1 依然是 PWM 周期内计数器的计数个数，但是 CCR2 寄存器的值+1 就是 PWM 低电平期间计数器的计数个数。通过这两个得到的参数依然可以计算出 PWM 的其它参数。这个大家了解一下就可以了，一般我们使用第六介绍的例子。

通过上面的描述，如果大家还不理解，下面我们结合 PWM 输入模式时序来分析一下。

PWM 输入模式时序图如图 22.5.2 所示：

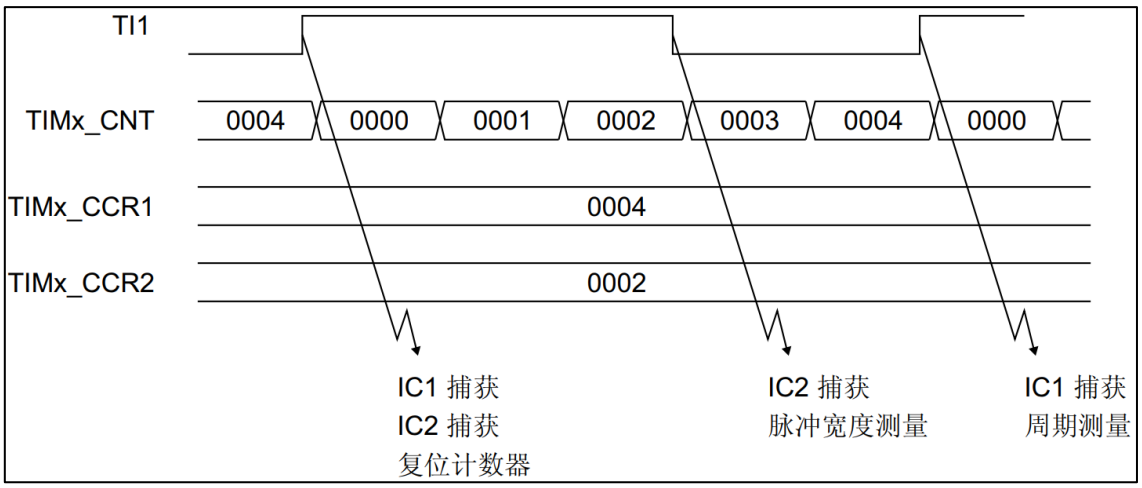


图 4.3.6.2 PWM 输入模式时序图

图 4.3.6.2 是以通道 1（CH1）输入 PWM，设置 IC1 捕获边沿为上升沿捕获，IC2 捕获边沿为下降沿捕获为例的 PWM 输入模式时序图。

从时序图可以看出，计数器的计数模式是递增计数模式。从左边开始看，当 TI1 来了上升沿时，计数器的值被复位为 0（原因是从模式选择为复位模式），IC1 和 IC2 都发生捕获事件。然后计数器的值计数到 2 的时候，IC2 发生了下降沿捕获，捕获事件会导致这时候的计数器的值被锁存到 CCR2 寄存器中，该值+1 就是高电平期间计数器的计数个数。最后计数器的值计数到 4 的时候，IC1 发生了上升沿捕获，捕获事件会导致这时候的计数器的值被锁存到 CCR1 寄存器中，该值+1 就是 PWM 周期内计数器的计数个数。

假设计数器的计数频率是 168MHz，那我们就可以计算出这个 PWM 的周期、频率和占空比等参数了。下面就以这个为例给大家计算一下。由计数器的计数频率为 168MHz，可以得到计数器计一个数的时间是 5.95ns（即测量的精度是 5.95ns）。知道了测量精度，再来计算 PWM 的周期，PWM 周期 =(4+1)*(1/168000000)＝29.75ns，那么 PWM 的频率就是 33.6MHz。占空比 ＝(2+1)/(4+1) =3/5（即占空比为 60%）。

4.3.6.1 TIM1/TIM8 寄存器

高级定时器 PWM 输入模式实验除了用到定时器的时基单元：计数器寄存器(TIMx_CNT)、预分频器寄存器(TIMx_PSC)、自动重载寄存器(TIMx_ARR) 之外。主要还用到以下这些寄存器：

- **从模式控制寄存器（TIMx_SMCR）**

  TIM1/TIM8 的从模式控制寄存器描述如图 4.3.6.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			Res.	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw

位 6:4 **TS**：触发选择 (Trigger selection)

此位域可选择将要用于同步计数器的触发输入。
000：内部触发 0 (ITR0)
001：内部触发 1 (ITR1)。
010：内部触发 2 (ITR2)。
011：内部触发 3 (ITR3)。
100：TI1 边沿检测器 (TI1F_ED)
101：滤波后的定时器输入 1 (TI1FP1)
110：滤波后的定时器输入 2 (TI2FP2)
111：外部触发输入 (ETRF)

注意：这些位只能在未使用的情况下（例如，SMS=000 时）进行更改，以避免转换时出现错误的边沿检测。

位 2:0 **SMS**：从模式选择 (Slave mode selection)

选择外部信号时，触发信号 (TRGI) 的有效边沿与外部输入上所选的极性相关（请参见输入控制寄存器和控制寄存器说明）。
000：禁止从模式——如果 CEN = “1”，预分频器时钟直接由内部时钟提供。
001：编码器模式 1——计数器根据 TI1FP1 电平在 TI2FP2 边沿递增/递减计数。
010：编码器模式 2——计数器根据 TI2FP2 电平在 TI1FP1 边沿递增/递减计数。
011：编码器模式 3——计数器在 TI1FP1 和 TI2FP2 的边沿计数，计数的方向取决于另外一个信号的电平。
100：复位模式——在出现所选触发输入 (TRGI) 上升沿时，重新初始化计数器并生成一个寄存器更新事件。
101：门控模式——触发输入 (TRGI) 为高电平时使能计数器时钟。只要触发输入变为低电平，计数器立即停止计数（但不复位）。计数器的启动和停止都是受控的。
110：触发模式——触发信号 TRGI 出现上升沿时启动计数器（但不复位）。只控制计数器的启动。
111：外部时钟模式 1——由所选触发信号 (TRGI) 的上升沿提供计数器时钟。

注意：如果将 TI1F_ED 选作触发输入 (TS=100)，则不得使用门控模式。实际上，TI1F 每次转换时，TI1F_ED 都输出 1 个脉冲，而门控模式检查的则是触发信号的电平。

图 4.3.6.1.1 TIMx_SMCR 寄存器

该寄存器的 SMS[2:0]位，用于从模式选择。比如在本实验中我们需要用到复位模式，所以设置 SMS[2:0]=100。TS[2:0]位是触发选择，我们设置为滤波后的定时器输入 1 (TI1FP1)，即 TS[2:0]为 101。

- **捕获/比较模式寄存器 1/2（TIMx_CCMR1/2）**

TIM1/TIM8 的捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器一般有 2 个：TIMx_CCMR1 和 TIMx _CCMR2。TIMx_CCMR1 控制 CH1 和 CH2，而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 4.3.6.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2 CE	OC2M[2:0]			OC2 PE	OC2 FE	CC2S[1:0]		OC1 CE	OC1M[2:0]			OC1 PE	OC1 FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.6.1.2 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下，功能不一样，我们现在用到输入捕获模式。关于该寄存器的详细说明，请参考《STM32F4xx 参考手册_V4（中文版）.pdf》第 375 页，14.4.7 节。

本实验我们通过定时器 1 通道 1 输入 PWM 信号，所以 IC1 和 IC2 都映射到 TI1 上。配置 CC1S[1:0]=01、CC2S [1:0]=10，其他位不用设置，默认为 0 即可。

- **捕获/比较使能寄存器（TIMx_CCER）**

TIM1/TIM8 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 4.3.6.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CC4P	CC4E	CC3NP	CC3NE	CC3P	CC3E	CC2NP	CC2NE	CC2P	CC2E	CC1NP	CC1NE	CC1P	CC1E
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.6.1.3 TIMx_CCER 寄存器

IC1 捕获上升沿，所以 CC1P 位置 0，即捕获发生在 IC1 的上升沿。IC2 捕获下降沿，所以 CC2P 位置 1，即捕获发生在 IC1 的下降沿。设置好捕获边沿后，还需要使能这两个通道捕获，即 CC1E 和 CC2E 位置 1。

- **捕获/比较寄存器 1/2/3/4（TIMx_CCR1/2/3/4）**

捕获/比较寄存器（TIMx_CCR1/2/3/4），该寄存器总共有 4 个，对应 4 个通道 CH1~CH4。我们使用的是通道 1，所以来看看 TIMx_CCR1 寄存器描述如图 4.3.6.1.4 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 15:0 **CCR1[15:0]**：捕获/比较 1 值 (Capture/Compare 1 value)

如果通道 CC1 配置为输出：
CCR1 是捕获/比较寄存器 1 的预装载值。
如果没有通过 TIMx_CCMR 寄存器中的 OC1PE 位来使能预装载功能，写入的数值会被直接传输至当前寄存器中。否则只在发生更新事件时生效（拷贝到实际起作用的捕获/比较寄存器1）。
实际捕获/比较寄存器中包含要与计数器 TIMx_CNT 进行比较并在 OC1 输出上发出信号的值。

如果通道 CC1 配置为输入：
CCR1 为上一个输入捕获 1 事件 (IC1) 发生时的计数器值。

图 4.3.6.1.4 TIMx_CCR1 寄存器

本实验中，CCR1 寄存器用于获取 PWM 周期内计数器的计数个数。CCR2 寄存器用于获取 PWM 高电平期间计数器的计数个数。

- **DMA/中断使能寄存器（TIMx_DIER）**

DMA/中断使能寄存器描述如图 4.3.6.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	COMDE	CC4DE	CC3DE	CC2DE	CC1DE	UDE	BIE	TIE	COMIE	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 1 **CC1IE**：捕获/比较 1 中断使能 (Capture/Compare 1 interrupt enable)
0：禁止 CC1 中断
1：使能 CC1 中断

位 0 **UIE**：更新中断使能 (Update interrupt enable)
0：禁止更新中断
1：使能更新中断

图 4.3.6.1.5 TIMx_DIER 寄存器

该寄存器位 0（UIE）用于使能或者禁止更新中断，因为本实验我们用到更新中断，所以该位需要置 1。位 1（CC1IE）用于使能或者禁止捕获/比较 1 中断，我们用到捕获中断，所以该位需要置 1。

4.3.6.2 硬件设计

1. 例程功能

首先通过 TIM8_CH1(PI5)输出 PWM 波。然后把 PI5 输出的 PWM 波用杜邦线接入 PA8（定时器 1 通道 1），最后通过串口打印 PWM 波的脉宽和频率等信息。LED0 闪烁来提示程序正在运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）定时器 8 通道 1（PI5）输出 PWM 波
定时器 1 通道 1（PA8）输入 PWM 波

3. 原理图

定时器属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们把 PI5 引脚输出的 PWM 波用杜邦线接入 PA8 引脚，然后通过电脑串口上位机软件观察打印出来的信息。

4.3.6.3 程序设计

定时器 PWM 输入模式实验用到的 HAL 库中的驱动代码在前面实验都有介绍过了。 我们在程序解析再详细讲解应用到的函数，下面介绍一下高级定时器 PWM 输入模式的配置步骤。

高级定时器 PWM 输入模式配置步骤

1）开启 TIMx 和输入通道的 GPIO 时钟，配置该 IO 口的复用功能输入

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输入。本实验我们默认用到定时器 1 通道 1，对应 IO 是 PA8，它们的时钟开启方法如下：

```
__HAL_RCC_TIM1_CLK_ENABLE();            /* 使能定时器 1 */
__HAL_RCC_GPIOA_CLK_ENABLE();           /* 开启 GPIOA 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的输入捕获功能时，我们调用的是 HAL_TIM_IC_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_IC_MspInit 函数，但是为不跟前面的实验共用该回调函数，提高独立性，我们就直接在 atim_timx_pwmin_chy_init 函数中，使能定时器时钟和 GPIO 时钟，初始化通道对应 IO 引脚等。

3）从模式配置，IT1 触发更新

通过 HAL_TIM_SlaveConfigSynchronization 函数，配置从模式：复位模式、定时器输入触发源、边缘检测、是否滤波等。

4）设置 IC1 捕获相关参数。

通过 HAL_TIM_IC_ConfigChannel 函数来设置定时器捕获通道 1 的工作方式，包括边缘检测极性、映射关系，输入滤波和输入分频等。

5）设置 IC2 捕获相关参数。

通过 HAL_TIM_IC_ConfigChannel 函数来设置定时器捕获通道 2 的工作方式，包括边缘检测极性、映射关系，输入滤波和输入分频等。

6）使能定时器更新中断，开启捕获功能，配置定时器中断优先级

通过__HAL_TIM_ENABLE_IT 函数使能定时器更新中断。

通过 HAL_TIM_IC_Start_IT 函数使能定时器并开启通道 1 和通道 2 的捕获功能，使能捕获中断。

通 HAL_NVIC_EnableIRQ 函数使能定时器中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

7）编写中断服务函数

TIM1 和 TIM8 有独立的输入捕获中断服务函数，分别是 TIM1_CC_IRQHandler 和 TIM8_CC_IRQHandler，其他定时器则没有，所以如果是 TIM1 和 TIM8 可以直接使用输入捕获中断服务函数来处理输入捕获中断。在使用 TIM1 的时候，如果要考虑定时器 1 溢出，可以重定义更新中断服务函数 TIM1_UP_TIM10_IRQHandler。如果使用 HAL 库的中断回调机制，可以在相关中断服务函数中直接调用定时器中断公共处理函数 HAL_TIM_IRQHandler，然后我们直接重定义相关的中断回调函数来编写中断程序即可。

本实验我们不考虑溢出的情况，来实现一个简单的 PWM 输入功能，只需要用到捕获中断回调函数 HAL_TIM_IC_CaptureCallback，大家如果需要更加宽频的 PWM 输入功能，可以加上溢出中断的处理逻辑。

4.3.6.3.1 程序流程图

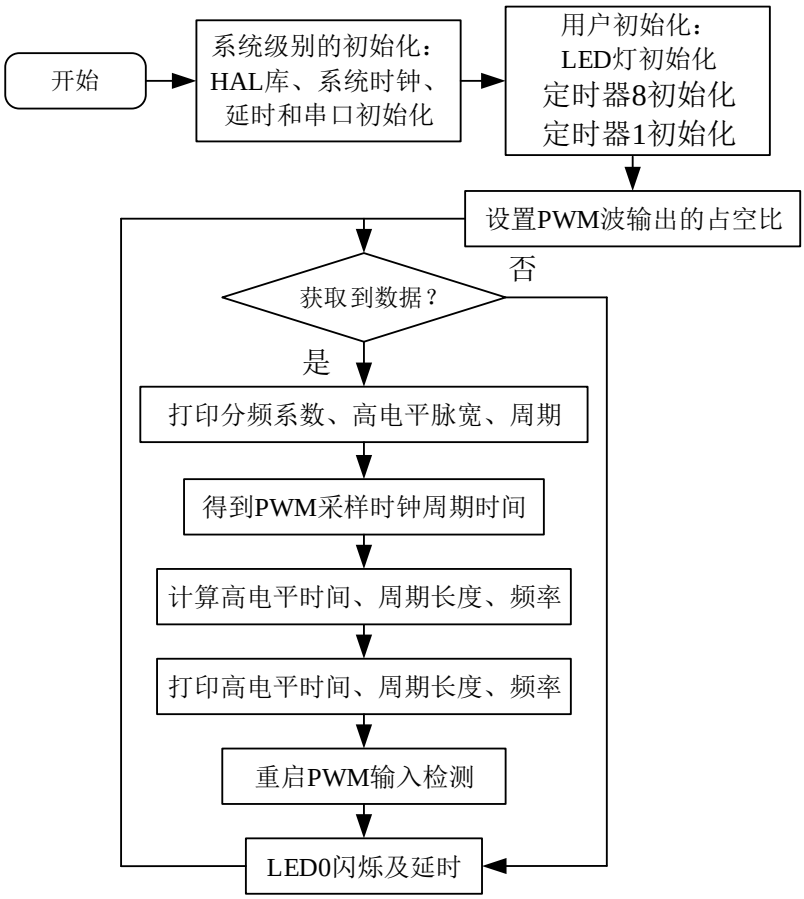


图 4.3.6.3.1.1 高级定时器 PWM 输入模式实验程序流程图

4.3.6.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义：

```
/******************  第一部分 高级定时器 PWM 输出实验相关宏定义  *******************/

/* TIMX PWM 输出定义 */

#define ATIM_TIMX_PWM_CHY_GPIO_PORT         GPIOI
#define ATIM_TIMX_PWM_CHY_GPIO_PIN          GPIO_PIN_5
#define ATIM_TIMX_PWM_CHY_GPIO_CLK_ENABLE()
        do{  __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)     /* PI 口时钟使能 */

/* TIMX 引脚复用设置
 * 因为 PI5 默认并不是 TIM8 的功能脚，必须开启复用，PI5 才能用作 TIM8 的 PWM 输出引脚.
 */
#define ATIM_TIMX_PWM_CHY_GPIO_AF           GPIO_AF3_TIM8

#define ATIM_TIMX_PWM                       TIM8
#define ATIM_TIMX_PWM_CHY                   TIM_CHANNEL_1 /* 通道 Y,1<= Y <=4 */
#define ATIM_TIMX_PWM_CHY_CLK_ENABLE()
        do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)       /* TIM8 时钟使能 */

/****************** 第二部分 高级定时器 PWM 输入实验相关宏定义 *******************/
```

```
/* TIMX PWM输入定义 */

#define ATIM_TIMX_PWMIN_CHY_GPIO_PORT        GPIOA
#define ATIM_TIMX_PWMIN_CHY_GPIO_PIN         GPIO_PIN_8
#define ATIM_TIMX_PWMIN_CHY_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)      /* PA口时钟使能 */

/* TIMX 引脚复用设置
 * 因为PA8默认并不是TIM1的功能脚, 必须开启复用, PA8才能用作TIM1的PWM输入引脚
 */
#define ATIM_TIMX_PWMIN_CHY_GPIO_AF         GPIO_AF1_TIM1

#define ATIM_TIMX_PWMIN                     TIM1
#define ATIM_TIMX_PWMIN_CH1                 TIM_CHANNEL_1 /* 通道Y,1<= Y <=2 */
#define ATIM_TIMX_PWMIN_CH2                 TIM_CHANNEL_2 /* 通道Y,1<= Y <=2 */
#define ATIM_TIMX_PWMIN_CHY_CLK_ENABLE()
        do{ __HAL_RCC_TIM1_CLK_ENABLE(); }while(0)       /* TIM1 时钟使能 */

 /* TIM1 / TIM8 有独立的捕获中断服务函数,需要单独定义,对于TIM2~5等,则不需要以下定义 */
#define ATIM_TIMX_PWMIN_CC_IRQn             TIM1_CC_IRQn
#define ATIM_TIMX_PWMIN_CC_IRQHandler       TIM1_CC_IRQHandler
```

可以把上面的宏定义分成两部分，第一部分包括是定时器 8 通道 1 对应的 IO 口的宏定义，第二部分则是定时器 1 的相应宏定义以及 TIM1/ TIM8 独有的捕获中断服务函数。

下面看 atim.c 的程序，关于定时器 8 的 PWM 输出部分这里就不再赘述，大家有需要的的话可以回顾 4.3.3 章节。首先看高级定时器 PWM 输入模式初始化函数，其定义如下：

```
/**
 * @brief       高级定时器TIMX 通道Y PWM输入模式 初始化函数
 * @note
 *              高级定时器的时钟来自APB2, 而PCLK2 = 168Mhz, 我们设置PPRE2不分频, 因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       无
 * @retval      无
 */
void atim_timx_pwmin_chy_init(void)
{
    GPIO_InitTypeDef gpio_init_struct   = {0};
    TIM_SlaveConfigTypeDef slave_config = {0};
    TIM_IC_InitTypeDef tim_ic_pwmin_chy = {0};

    ATIM_TIMX_PWMIN_CHY_CLK_ENABLE();                       /* 使能TIM1时钟 */
    ATIM_TIMX_PWMIN_CHY_GPIO_CLK_ENABLE();                  /* 开启捕获IO的时钟 */

    gpio_init_struct.Pin = ATIM_TIMX_PWMIN_CHY_GPIO_PIN;    /* PWM输入IO */
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;                /* 复用推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLDOWN;                  /* 下拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
    gpio_init_struct.Alternate = ATIM_TIMX_PWMIN_CHY_GPIO_AF;     /* 端口复用 */
    HAL_GPIO_Init(ATIM_TIMX_PWMIN_CHY_GPIO_PORT, &gpio_init_struct);

    g_timx_pwmin_chy_handle.Instance = ATIM_TIMX_PWMIN;     /* 定时器1 */
    g_timx_pwmin_chy_handle.Init.Prescaler = 168 - 1;       /* 定时器预分频系数 */
    g_timx_pwmin_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP;/* 递增计数 */
    g_timx_pwmin_chy_handle.Init.Period = 65535;            /* 自动重装载值 */
    HAL_TIM_IC_Init(&g_timx_pwmin_chy_handle);              /* 初始化定时器 */

    /* 从模式配置, IT1触发更新 */
    slave_config.SlaveMode = TIM_SLAVEMODE_RESET;           /* 从模式: 复位模式 */
    slave_config.InputTrigger = TIM_TS_TI1FP1;              /* 触发源: TI1FP1 */
```

```
    /* 一开始捕获到上升沿复位计数值 */
    slave_config.TriggerPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
    slave_config.TriggerFilter = 0;                              /* 不滤波 */
    HAL_TIM_SlaveConfigSynchro(&g_timx_pwmin_chy_handle, &slave_config);

    /* IC1 捕获：上升沿触发 TI1FP1 */
    tim_ic_pwmin_chy.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
    tim_ic_pwmin_chy.ICSelection = TIM_ICSELECTION_DIRECTTI; /* IC1 映射到 TI1 */
    tim_ic_pwmin_chy.ICPrescaler = TIM_ICPSC_DIV1;           /* 不分频 */
    tim_ic_pwmin_chy.ICFilter = 0;                           /* 不滤波 */
    HAL_TIM_IC_ConfigChannel(&g_timx_pwmin_chy_handle, &tim_ic_pwmin_chy,
                             ATIM_TIMX_PWMIN_CH1); /* 配置 TIM1 通道 1 */

    /* IC2 捕获：下降沿触发 TI1FP2 */
    tim_ic_pwmin_chy.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
    tim_ic_pwmin_chy.ICSelection = TIM_ICSELECTION_INDIRECTTI; /*IC2 映射到 TI1*/
    HAL_TIM_IC_ConfigChannel(&g_timx_pwmin_chy_handle, &tim_ic_pwmin_chy,
                             ATIM_TIMX_PWMIN_CH2); /* 配置 TIM1 通道 2 */
    /* 开启 TIM1 的捕获通道 1 */
    HAL_TIM_IC_Start_IT(&g_timx_pwmin_chy_handle, ATIM_TIMX_PWMIN_CH1);
    /* 开启 TIM1 的捕获通道 2 */
    HAL_TIM_IC_Start_IT(&g_timx_pwmin_chy_handle, ATIM_TIMX_PWMIN_CH2);
    /* TIM1/TIM8 有独立的输入捕获中断服务函数 */
    if ( ATIM_TIMX_PWMIN == TIM1 || ATIM_TIMX_PWMIN == TIM8)
    {
        HAL_NVIC_SetPriority(ATIM_TIMX_PWMIN_CC_IRQn, 1, 3);  /* 设置中断优先级*/
        HAL_NVIC_EnableIRQ(ATIM_TIMX_PWMIN_CC_IRQn);           /* 开启 TIMx 中断 */
    }
}
```

在 atim_timx_pwmin_chy_init 函数中，没有使用 HAL 库的 MSP 回调，而是把相关的初始化都写到该函数里面。

第一部分，使能定时器和相关通道对应的 GPIO 时钟，以及初始化相关 IO 引脚。

第二部分，通过 HAL_TIM_IC_Init 函数初始化定时器的 ARR 和 PSC 等参数。

第三部分，通过 HAL_TIM_SlaveConfigSynchronization 函数配置从模式，复位模式等。

第四部分，通过 HAL_TIM_IC_ConfigChannel 函数分别配置 IC1 和 IC2。

第五部分，配置 NVIC，使能定时器中断，配置抢占优先级和响应优先级。

最后，通过调用 HAL_TIM_IC_Start_IT 函数使能捕获中断，并且使能定时器。

注意：我们在 atim_timx_pwmin_chy_init 函数中直接设置了定时器 1 的预分频系数和重装载值，它的频率 = 168M / 168 = 1M Hz，周期为 1us，重装载值为 65535，定时器 1 的溢出时间为 65536us。当计数溢出的时候，我们不做处理，也就是说，按照这个配置，只能检测 PWM 周期不超过 65536us 的信号。如果大家需要测量更长周期的 PWM 波形，可以自行修改代码，对计数溢出进行相应的处理。

下面要介绍的就是中断服务函数，及对应的回调函数。我们这里只用到 ATIM_TIMX_PWMIN_CC_IRQHandler 这个函数，即高级定时器独有的输入捕获中断服务函数。在中断服务函数我们调用 HAL 库的中断处理函数 HAL_TIM_IRQHandler 来处理，然后我们的逻辑程序就放在定时器输入捕获中断处理回调函数 HAL_TIM_IC_CaptureCallback 里。这两个函数定义如下：

```
/**
 * @brief       定时器 TIMX 输入捕获 中断服务函数
 *   @note      仅 TIM1/TIM8 有这个函数,其他普通定时器没有这个中断服务函数!
 * @param       无
 * @retval      无
 */
void ATIM_TIMX_PWMIN_CC_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&g_timx_pwmin_chy_handle);    /* 定时器共用处理函数 */
```

```
}

/**
 * @brief       定时器输入捕获中断处理回调函数
 * @param       htim:定时器句柄指针
 * @note        该函数在 HAL_TIM_IRQHandler 中会被调用，
                PWM 输入属于输入捕获中的特例
 * @retval      无
 */
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == ATIM_TIMX_PWMIN)
    {
        if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)     /* 通道 1 捕获到上升沿 */
        {
            __HAL_TIM_SET_COUNTER(&g_timx_pwmin_chy_handle,0);

            g_timxchy_pwmin_cval = HAL_TIM_ReadCapturedValue(htim,
                                   ATIM_TIMX_PWMIN_CH1)+1;   /* 记录整个周期的计数值 */
            g_timxchy_pwmin_sta = 1;                         /* 标记捕获成功 */
        }
        else if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2)/* 通道 2 捕获到下降沿 */
        {
            g_timxchy_pwmin_hval =HAL_TIM_ReadCapturedValue(htim,
                                   ATIM_TIMX_PWMIN_CH2)+1; /* 记录高电平时间的计数值 */
        }
    }
}
```

在输入捕获中断处理回调函数中，我们先判断通道 1 是否捕获到了上升沿，如果捕获到了，就把计数器清零，通过调用 HAL_TIM_ReadCapturedValue 函数获取整个周期对应的计数值，然后标记捕获成功，如果捕获到通道 2 的下降沿，就通过调用 HAL_TIM_ReadCapturedValue 函数获取高电平时间对应的计数值。

下面介绍一下待测试的 PWM 怎么得到。因为在高级定时器 PWM 输出实验中我们已经编写了 PWM 输出的程序，所以这里直接使用高级定时器的 PWM 输出实验的代码进行初始化，从而让 TIM8_CH1(PI5)输出 PWM 波。然后我们用杜邦线把 PI5 和 PA8 连接起来。这样 PI5 输出的 PWM 就可以输入到 PA8（定时器 8 通道 1）进行测量。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t t = 0;

    HAL_Init();                                     /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);             /* 设置时钟,168Mhz */
    delay_init(168);                                /* 延时初始化 */
    usart_init(115200);                             /* 串口初始化为 115200 */
    led_init();                                     /* 初始化 LED */
    /* 10K Hz 的计数频率，50ms 的周期，其中高电平占一半也就是 25ms = 25000us */
    atim_timx_pwm_chy_init(500 - 1, 16800 - 1);
    /* 初始化 PWN 输入，以 1Mhz 的频率计数，最大捕获周期 65536 us */
    atim_timx_pwmin_chy_init();

    while (1)
    {
        t++;
        if(t % 20 == 0)
        {
            LED0_TOGGLE();
        }
        if(g_timxchy_pwmin_sta == 1)            /* 成功捕获 */
        {
```

```
            g_timxchy_pwmin_sta = 0;            /* 清除成功捕获标志 */

            printf("\r\n");
            printf("高电平时间：%d us\r\n",g_timxchy_pwmin_hval);
            printf("PWM 周期：%d us\r\n",g_timxchy_pwmin_cval );
            printf("PWM 频率：%d Hz\r\n",1000000/g_timxchy_pwmin_cval);
        }
        delay_ms(10);
    }
}
```

先看 atim_timx_pwm_chy_init(500 - 1, 16800 - 1)这个语句，这两个形参分别设置自动重载寄存器的值为 499，以及定时器预分频系数为 16799。先看预分频系数，我们设置为 16800 分频，定时器 8 的时钟是 2 倍的 APB2，即 168MHZ，可以得到计数器的计数频率是 10k HZ，即 100us 计数一次。再到自动重载寄存器的值为 499 决定的是 PWM 波的频率（周期），可以得到 PWM 的周期为 500*100us = 50000us， 频率为 20Hz。然后我们在定时器 8 的初始化时已经设置比较值为 arr 的一半，即 PWM 的占空比为 50%，所以一个周期的高电平时间为 25000us。下载验证的时候验证一下捕获到的与输出的是否一致。

atim_timx_pwmin_chy_init 这个语句，就初始化 PWM 输入捕获。然后在无限循环中每 10ms 判断 g_timxchy_pwmin_sta 标志变量，是否捕获到数据，捕获到就打印和计数相关信息。

下面我们下载到开发板验证一下。

4.3.6.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，我们再打开串口调试助手，选择对应的串口端口，我这边是 COM3。然后用杜邦线把 PI5 引脚连接到 PA8 引脚，就可以看到串口助手不断打印 PWM 波的信息，如图 4.3.6.4.1 所示：

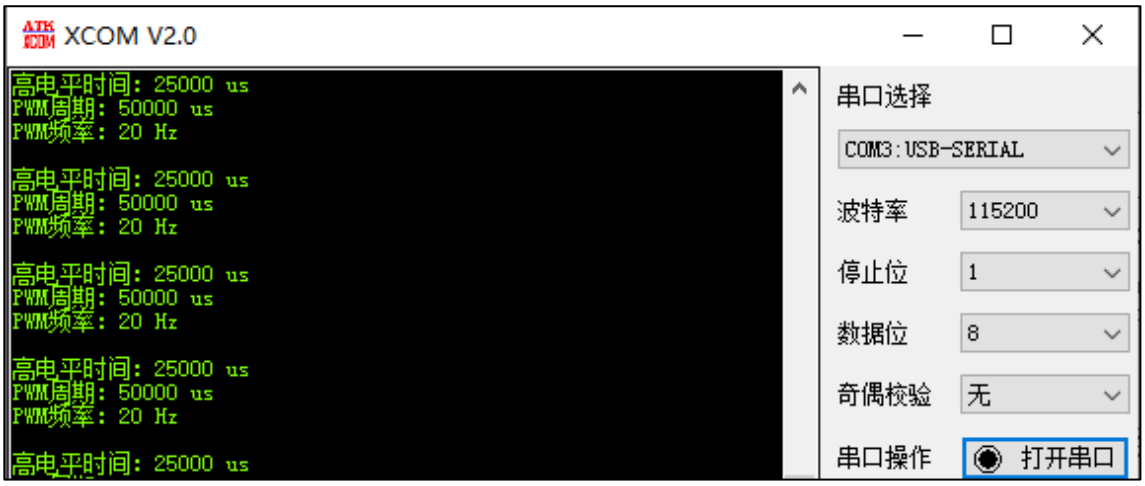


图 4.3.6.4.1 打印 PWM 波的信息

可以看到打印出来的 PWM 波信息为：周期是 50000us，频率是 20Hz，占空比是 50%，和我们的预想结果一样。

大家可以通过 atim_timx_pwm_chy_init 函数来设置 PWM 波的周期和频率，通过 g_atimx_oc_pwm_chy_handle.Pulse 这个结构体成员设置占空比。

注意：本实验没有对计数溢出做处理，所以可测量的 PWM 波范围有一定局限性（最大周期 65536us），大家如果需要更加宽频的 PWM 检测功能，则可以加上计数溢出的处理。

4.3.7 高级定时器输出比较模式实验

本小节我们来学习使用高级定时器输出比较模式下的翻转功能，通过定时器的 3 个通道分别输出 3 个 50%占空比、不同相位的 PWM。

输出比较模式下翻转功能作用是：当计数器的值等于捕获/比较寄存器影子寄存器的值时，OC1REF 发生翻转，进而控制通道输出（OCx）翻转。通过翻转功能实现输出 PWM 的具体原理如下：PWM 频率由自动重载寄存器（TIMx_ARR）的值决定，在这个过程中，只要自动重载寄存器的值不变，那么 PWM 占空比就固定为 50%。我们可以通过捕获/比较寄存器（TIMx_CCRx）的值改变 PWM 的相位。生成 PWM 的原理如图 4.3.7.1 所示：

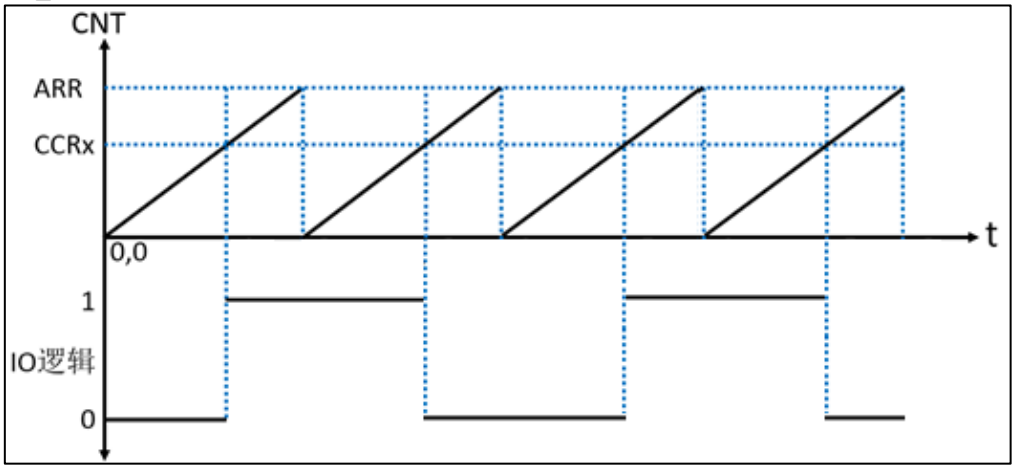


图 4.3.7.1 输出比较模式翻转功能输出 PWM 波原理示意图

本实验就是根据图 4.3.7.1 的原理来设计的，具体实验是：我们设置固定的 ARR 值为 999，那么 PWM 占空比固定为 50%，通过改变 3 个通道的捕获/比较寄存器（TIMx_CCRx）的值使得每个通道输出的 PWM 的相位都不一样，注意捕获/比较寄存器的值设置范围是：0 ~ ARR。比如：TIMx_CCR1=250-1，TIMx_CCR2=500-1，TIMx_CCR3=750-1，那么可以得到通道 1~通道 3 输出的 PWM 的相位分别是：25%、50%、75%。翻转功能输出的 PWM 周期，这里用 T 表示，其计算公式如下：

$$T = 2*(arr+1)*((psc+1)/ Tclk)$$

其中：

T：翻转功能输出的 PWM 周期（单位为 s）。

Tclk：定时器的时钟源频率（单位为 MHz）。

arr：自动重装寄存器（TIMx_ARR）的值。

psc：预分频器寄存器（TIMx_PSC）的值。

4.3.7.1 TIM1/TIM8 寄存器

高级定时器输出比较模式除了用到定时器的时基单元：计数器寄存器(TIMx_CNT)、预分频器寄存器(TIMx_PSC)、自动重载寄存器(TIMx_ARR) 之外。主要还用到以下这些寄存器：

- **控制寄存器 1（TIMx_CR1）**

TIM1/TIM8 的控制寄存器 1 描述如图 4.3.7.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS[1:0]		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 7 **ARPE**：自动重载预装载使能 (Auto-reload preload enable)

0：TIMx_ARR 寄存器不进行缓冲
1：TIMx_ARR 寄存器进行缓冲

位 4 **DIR**：方向 (Direction)

0：计数器递增计数
1：计数器递减计数

注意：当定时器配置为中心对齐模式或编码器模式时，该位为只读状态。

位 0 **CEN**：计数器使能 (Counter enable)

0：禁止计数器
1：使能计数器

注意：只有事先通过软件将 CEN 位置 1，才可以使用外部时钟、门控模式和编码器模式。而触发模式可通过硬件自动将 CEN 位置 1。

图 4.3.7.1.1 TIMx_CR1 寄存器

上图中我们只列出了本实验需要用的一些位，其中：位 7（APRE）用于控制自动重载寄存器是否具有缓冲作用，在基本定时器的时候已经讲过，请回顾。本实验中，我们把该位置 1。

位 4（DIR）用于配置计数器的计数方向，本实验默认置 0 即可。

位 CEN 位，用于使能计数器的工作，必须要设置该位为 1，才可以开始计数。

其它位保持复位值即可。

- **捕获/比较模式寄存器 1/2（TIMx_CCMR1/2）**

TIM1/TIM8 的捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器一般有 2 个：TIMx_CCMR1 和 TIMx _CCMR2。TIMx_CCMR1 控制 CH1 和 CH2，而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 4.3.7.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.7.1.2 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下，功能不一样，我们现在用到输出比较模式。关于该寄存器的详细说明，请参考《STM32F4xx 参考手册_V4（中文版）.pdf》第 432 页，15.4.7 节。

本实验我们用到了定时器 8 输出比较的 3 个通道，所以我们需要配置 TIM1_CCMR1 和 TIM1_CCMR2。以 TIM1_CCMR1 寄存器为例，模式设置位 OC1M[2:0]就是对应着通道 1 的模式设置，此部分由 3 位组成，总共可以配置成 8 种模式，我们使用的是翻转功能，所以这 3 位必须设置为 011。通道 2 也是如此，将位 OC2M[2:0]设置为 011。通道 3 就要设置 TIM1_CCMR2 寄存器的位 OC3M[2:0]。除此之外，我们还要设置输出比较的预装载使能位，如通道 1 对应输出比较的预装载使能位 OC1PE 置 1，其他通道也要把相应位置 1。

- **捕获/比较使能寄存器（TIMx_CCER）**

TIM1/TIM8 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 4.3.7.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留		CC4P	CC4E	CC3NP	CC3NE	CC3P	CC3E	CC2NP	CC2NE	CC2P	CC2E	CC1NP	CC1NE	CC1P	CC1E
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.7.1.3 TIMx_CCER 寄存器

该寄存器比较简单，要让 TIM1 的 3 个通道都输出，我们需要把对应的捕获/比较 1 输出使能位置 1。通道 1 到通道 3 的使能位分别是：CC1E、CC2E、CC3E，我们把这 3 个位置置 1，使能通道输出。

- **捕获/比较寄存器 1/2/3/4（TIMx_CCR1/2/3/4）**

捕获/比较寄存器（TIMx_CCR1/2/3/4），该寄存器总共有 4 个，对应 4 个通道 CH1~CH4。

本实验需要用到 1~3 通道，以通道 1 对应的 TIMx_ CCR1 寄存器为例，其描述如图 4.3.7.1.4 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 15:0 **CCR1[15:0]**：捕获/比较 1 值 (Capture/Compare 1 value)

如果通道 CC1 配置为输出：
CCR1 是捕获/比较寄存器 1 的预装载值。
如果没有通过 TIMx_CCMR 寄存器中的 OC1PE 位来使能预装载功能，写入的数值会被直接传输至当前寄存器中。否则只在发生更新事件时生效（拷贝到实际起作用的捕获/比较寄存器1）。
实际捕获/比较寄存器中包含要与计数器 TIMx_CNT 进行比较并在 OC1 输出上发出信号的值。

如果通道 CC1 配置为输入：
CCR1 为上一个输入捕获 1 事件 (IC1) 发生时的计数器值。

图 4.3.7.1.4 TIMx_CCR1 寄存器

这里，我们通过改变 TIMx_CCR1/2/3 寄存器的值来改变 3 个通道输出的 PWM 的相位。

- **TIM1/TIM8 断路和死区寄存器（TIMx_BDTR）**

本实验用的是高级定时器，我们还需要配置：断路和死区寄存器（TIMx_BDTR），该寄存器各位描述如图 4.3.7.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOE	AOE	BKP	BKE	OSSR	OSSI	LOCK[1:0]		DTG[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

注意：由于可以根据 LOCK 配置锁定位 AOE、BKP、BKE、OSSI、OSSR 和 DTG[7:0] 的写操作，因此必须在第一次对 TIMx_BDTR 寄存器执行写访问时对这些位进行配置。

位 15 **MOE**：主输出使能 (Main output enable)

只要断路输入变为有效状态，此位便由硬件异步清零。此位由软件置 1，也可根据 AOE 位状态自动置 1。此位仅对配置为输出的通道有效。
0：OC 和 OCN 输出禁止或被强制为空闲状态。
1：如果 OC 和 OCN 输出的相应使能位（TIMx_CCER 寄存器中的 CCxE 和 CCxNE 位）均置 1，则使能 OC 和 OCN 输出。

图 4.3.7.1.5 TIMx_BDTR 寄存器

该寄存器，我们只需要关注位 15（MOE），要想高级定时器的通道正常输出，则必须设置 MOE 位为 1，否则不会有输出。

4.3.7.2 硬件设计

1. 例程功能

使用输出比较模式的翻转功能，通过定时器 8 的 3 路通道输出占空比固定为 50%、相位分别是 25%、50%和 75%的 PWM。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）PI5 复用为 TIM8_CH1
PI6 复用为 TIM8_CH2
PI7 复用为 TIM8_CH3

3. 原理图

定时器属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们需要通过示波器观察 PI5、PI6 和 PI7 引脚 PWM 输出的情况。

4.3.7.3 程序设计

4.3.7.3.1 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面已经介绍了部分，请回顾，这里我们再介绍几个本实验用到的函数。

1. HAL_TIM_OC_Init 函数

定时器的输出比较模式初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_OC_Init(TIM_HandleTypeDef *htim);
```

- **函数描述：**
  用于初始化定时器的输出比较模式。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，基本定时器的时候已经介绍。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

2. HAL_TIM_OC_ConfigChannel 函数

定时器的输出比较通道设置初始化函数。其声明如下：

```
HAL_StatusTypeDef HAL_TIM_OC_ConfigChannel(TIM_HandleTypeDef *htim,
                                           TIM_OC_InitTypeDef *sConfig, uint32_t Channel);
```

- **函数描述：**
  该函数用于初始化定时器的输出比较通道。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。
  形参 2 是 TIM_OC_InitTypeDef 结构体类型指针变量，用于配置定时器的输出比较参数。在高级定时器 PWM 输出实验已经介绍过 TIM_OC_InitTypeDef 结构体指针类型。
  形参 3 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

3. HAL_TIM_OC_Start 函数

定时器的输出比较启动函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_OC_Start(TIM_HandleTypeDef *htim, uint32_t Channel);
```

- **函数描述：**
  用于启动定时器的输出比较模式。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。
  形参 2 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。
- **注意事项：**
  HAL 库也同样提供了单独使能定时器的输出通道函数，函数为：

```
void TIM_CCxChannelCmd(TIM_TypeDef *TIMx, uint32_t Channel,
                       uint32_t ChannelState);
```

HAL_TIM_OC_Start 函数内部也调用了该函数。

定时器输出比较模式配置步骤

1）开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1、2、3，对应 IO 是 PI5\PI6\PI7，它们的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE();            /* 使能定时器 8 */
__HAL_RCC_GPIOI_CLK_ENABLE();           /* 开启 GPIOI 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的输出比较模式时，我们调用的是 HAL_TIM_OC_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用 HAL_TIM_OC_MspInit 函数，我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能以及优先级设置等代码。

3）设置定时器为输出比较模式，输出比较极性，输出比较值、翻转功能等参数

在 HAL 库中，通过 HAL_TIM_OC_ConfigChannel 函数来设置定时器为输出比较模式，根据需求设置输出比较的极性，设置输出比较值、翻转功能等。

最后我们通过__HAL_TIM_ENABLE_OCxPRELOAD 函数使能通道的预装载。

4）开启定时器并输出 PWM

通过 HAL_TIM_OC_Start 函数使能定时器并开启输出。

4.3.7.3.2 程序流程图

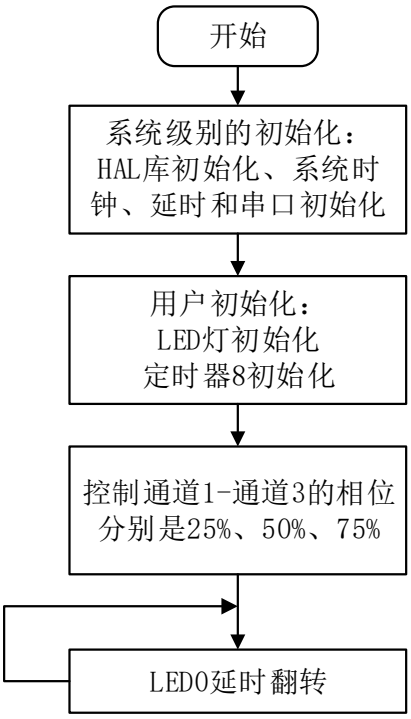


图 4.3.7.3.2.1 高级定时器输出比较模式实验程序流程图

4.3.7.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义：

```
/* TIMX PWM 输出定义 */

#define ATIM_TIMX_COMP_CH1_GPIO_PORT          GPIOI
#define ATIM_TIMX_COMP_CH1_GPIO_PIN           GPIO_PIN_5
#define ATIM_TIMX_COMP_CH1_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)       /* PI 口时钟使能 */

#define ATIM_TIMX_COMP_CH2_GPIO_PORT          GPIOI
#define ATIM_TIMX_COMP_CH2_GPIO_PIN           GPIO_PIN_6
#define ATIM_TIMX_COMP_CH2_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)       /* PI 口时钟使能 */

#define ATIM_TIMX_COMP_CH3_GPIO_PORT           GPIOI
#define ATIM_TIMX_COMP_CH3_GPIO_PIN            GPIO_PIN_7
```

```
#define ATIM_TIMX_COMP_CH3_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)   /* PI 口时钟使能 */

/* TIMX 引脚复用设置
 * 因为 PI5、PI6、PI7 默认并不是 TIM8 的功能脚，必须开启复用，才能用作 TIM8 的 PWM 输出引脚
 */
#define ATIM_TIMX_COMP_GPIO_AF   GPIO_AF3_TIM8         /* 端口复用到 TIM8 */

#define ATIM_TIMX_COMP           TIM8
#define ATIM_TIMX_COMP_CH1_CCRX  ATIM_TIMX_COMP->CCR1 /* 通道 1 的输出比较寄存器 */
#define ATIM_TIMX_COMP_CH2_CCRX  ATIM_TIMX_COMP->CCR2 /* 通道 2 的输出比较寄存器 */
#define ATIM_TIMX_COMP_CH3_CCRX  ATIM_TIMX_COMP->CCR3 /* 通道 3 的输出比较寄存器 */
#define ATIM_TIMX_COMP_CLK_ENABLE()
        do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)    /* TIM8 时钟使能 */
```

可以把上面的宏定义分成两部分，第一部分是定时器 8 输出通道 1~通道 3 对应的 IO 口的宏定义。第二部分则是定时器 8 的相应宏定义。

下面来看到 atim.c 文件的程序，首先是高级定时器输出比较模式初始化函数，其定义如下：

```
/**
 * @brief     高级定时器 TIMX 输出比较模式 初始化函数（使用输出比较模式）
 * @note
 *            配置高级定时器 TIMX 3 路输出比较模式 PWM 输出,实现 50%占空比,不同相位控制
 *            注意,本例程输出比较模式,每 2 个计数周期才能完成一个 PWM 输出,因此输出频率减半
 *            另外,我们还可以开启中断在中断里面修改 CCRx,从而实现不同频率/不同相位的控制
 *            但是我们不推荐这么使用,因为这可能导致非常频繁的中断,从而占用大量 CPU 资源
 *
 *            高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *            高级定时器时钟 = 168Mhz
 *            定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us
 *            Ft=定时器工作频率,单位:Mhz
 *
 * @param     arr: 自动重装值。
 * @param     psc: 时钟预分频数
 * @retval    无
 */
void atim_timx_comp_pwm_init(uint16_t arr, uint16_t psc)
{
    TIM_OC_InitTypeDef sConfigOC = {0};

    g_timx_comp_pwm_handle.Instance = ATIM_TIMX_COMP; /* 定时器 x */
    g_timx_comp_pwm_handle.Init.Prescaler = psc  ;    /* 定时器分频 */
    g_timx_comp_pwm_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 向上计数 */
    g_timx_comp_pwm_handle.Init.Period = arr;         /* 自动重装载值 */
    g_timx_comp_pwm_handle.Init.AutoReloadPreload =
                      TIM_AUTORELOAD_PRELOAD_ENABLE;  /* 使能影子寄存器 TIMx_ARR */
    HAL_TIM_OC_Init(&g_timx_comp_pwm_handle);         /* 输出比较模式初始化 */

    sConfigOC.OCMode = TIM_OCMODE_TOGGLE;             /* 比较输出模式 */
    sConfigOC.Pulse = 250 - 1;                        /* 设置输出比较寄存器的值 */
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;       /* 输出比较极性为高 */
    HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle, &sConfigOC,
                             TIM_CHANNEL_1);    /* 初始化定时器的输出比较通道 1 */
    /* 通道 1 预装载使能 */
    __HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_1);

    sConfigOC.Pulse = 500 - 1;
    HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle, &sConfigOC,
                             TIM_CHANNEL_2);    /* 初始化定时器的输出比较通道 2 */
    /* 通道 2 预装载使能 */
    __HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_2);
```

```
    sConfigOC.Pulse = 750 - 1;
    sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
    HAL_TIM_OC_ConfigChannel(&g_timx_comp_pwm_handle, &sConfigOC,
                             TIM_CHANNEL_3);    /* 初始化定时器的输出比较通道 3 */
    /* 通道 3 预装载使能 */
    __HAL_TIM_ENABLE_OCxPRELOAD(&g_timx_comp_pwm_handle, TIM_CHANNEL_3);

    /* 开启通道 x 输出 */
    HAL_TIM_OC_Start(&g_timx_comp_pwm_handle,TIM_CHANNEL_1);
    HAL_TIM_OC_Start(&g_timx_comp_pwm_handle,TIM_CHANNEL_2);
    HAL_TIM_OC_Start(&g_timx_comp_pwm_handle,TIM_CHANNEL_3);
}
```

在 atim_timx_comp_pwm_init 函数中，首先调用 HAL_TIM_OC_Init 函数初始化定时器的 ARR 和 PSC 等参数。然后通过调用函数 HAL_TIM_OC_ConfigChannel 设置通道 1~通道 3 的工作参数，包括：输出比较模式功能、输出比较寄存器的值，输出极性等。接着调用 __HAL_TIM_ENABLE_OCxPRELOAD 函数宏使能 CCR1/2/3 寄存器的预装载。最后通过调用函数 HAL_TIM_OC_Start 来使能 TIM1 通道 1~通道 3 输出。

HAL_TIM_OC_Init 函数会调用 HAL_TIM_OC_MspInit 回调函数，我们把使能定时器和通道对应的 IO 时钟、IO 初始化的代码存放到该函数里，其定义如下：

```
/**
 * @brief       定时器底层驱动，时钟使能，引脚配置
                此函数会被 HAL_TIM_OC_Init()调用
 * @param       htim:定时器句柄
 * @retval      无
 */
void HAL_TIM_OC_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == ATIM_TIMX_COMP)
    {
        GPIO_InitTypeDef gpio_init_struct;

        ATIM_TIMX_COMP_CLK_ENABLE();                /* 使能定时器 x 时钟 */
        ATIM_TIMX_COMP_CH1_GPIO_CLK_ENABLE();       /* 使能通道 1 的 IO 时钟 */
        ATIM_TIMX_COMP_CH2_GPIO_CLK_ENABLE();       /* 使能通道 2 的 IO 时钟 */
        ATIM_TIMX_COMP_CH3_GPIO_CLK_ENABLE();       /* 使能通道 3 的 IO 时钟 */

        /* 初始化通道 1 的 IO */
        gpio_init_struct.Pin = ATIM_TIMX_COMP_CH1_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;
        gpio_init_struct.Pull = GPIO_NOPULL;
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;
        gpio_init_struct.Alternate = ATIM_TIMX_COMP_GPIO_AF;
        HAL_GPIO_Init(ATIM_TIMX_COMP_CH1_GPIO_PORT, &gpio_init_struct);

        /* 初始化通道 2 的 IO */
        gpio_init_struct.Pin = ATIM_TIMX_COMP_CH2_GPIO_PIN;
        HAL_GPIO_Init(ATIM_TIMX_COMP_CH2_GPIO_PORT, &gpio_init_struct);

        /* 初始化通道 3 的 IO */
        gpio_init_struct.Pin = ATIM_TIMX_COMP_CH3_GPIO_PIN;
        HAL_GPIO_Init(ATIM_TIMX_COMP_CH3_GPIO_PORT, &gpio_init_struct);
    }
}
```

该函数主要是使能定时器和通道对应的 IO 时钟，初始化 IO 口。

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
```

```
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    /* 1Mhz 的计数频率，1Khz 的周期 */
    atim_timx_comp_pwm_init(1000 - 1, 168 - 1);
    ATIM_TIMX_COMP_CH1_CCRX = 250 - 1;          /* 通道 1 相位 25% */
    ATIM_TIMX_COMP_CH2_CCRX = 500 - 1;          /* 通道 2 相位 50% */
    ATIM_TIMX_COMP_CH3_CCRX = 750 - 1;          /* 通道 3 相位 75% */

    while (1)
    {
        LED0_TOGGLE();                          /*LED0(红灯) 翻转*/
        delay_ms(200);
    }
}
```

本小节开头我们讲解了输出比较模式翻转功能如何产生 PWM 波，下面结合程序一起计算出 PWM 波的周期，频率等参数。

定时器 8 时钟源的时钟频率等于 APB2 总线时钟频率，即 168MHz，而调用 atim_timx_comp_pwm_init(1000 - 1, 168 - 1)初始化函数之后，就相当于写入预分频寄存器的值为 167，写入自动重载寄存器的值为 999。将这些参数代入本小节介绍的翻转功能输出的 PWM 周期计算公式，可得：

$$T = 2*(arr+1)*((psc+1)/ Tclk) = 2*(999+1)*((167+1)/ 168000000) = 0.002s$$

由上述式子得到 PWM 周期为 2ms，频率为 500Hz。ARR 值为固定为 1000，所以占空比则固定为 50%。定时器 8 通道 1~通道 3 输出的 PWM 波的相位分别是：25%、50%、75%。

4.3.7.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了。 我们需要借助示波器观察 PI5、PI6 和 PI7 引脚 PWM 输出的情况，如图 4.3.7.4.1 所示：

图 4.3.7.4.1 相位为 25%、50%、75%的 PWM 波

图 4.3.7.4.1 中，由上到下分别是引脚 PI5、PI6 和 PI7 输出的 PWM 波，即分别对应的是 TIM8_CH1、TIM8_CH2 和 TIM8_CH3 输出的相位为 25%、50%和 75%的 PWM 波。大家可以把其中一个通道的捕获/比较寄存器的值设置为 0，那么就可以得到 PWM 初相位的波形，即相位为 0%。

4.3.8 高级定时器输出指定个数 PWM 实验

要实现定时器输出指定个数 PWM，只需要掌握下面几点内容：

第一，如果大家还不清楚定时器是如何输出 PWM 的，请回顾通用定时器 PWM 输出实验的内容，这部分的知识是一样的。但是需要注意的是：我们需要把 MOE 位置 1，这样高级定时器的通道才能输出。

第二，要清楚重复计数器特性，设置重复计数器寄存器 RCR 的值为 N，那么更新事件将在定时器发生 N+1 次上溢或下溢时发生。换句话来说就是，想要指定输出 N 个 PWM，只需要把 N-1 写入 RCR 寄存器。因为在边沿对齐模式下，定时器溢出周期对应着 PWM 周期，我们只要在更新事件发生时，停止输出 PWM 就行。

第三，为了保证定时器输出指定个数的 PWM 后，定时器马上停止继续输出，我们使能更新中断，并在定时器中断里关闭计数器。

原理部分我们就讲到这里，下面直接开始寄存器的介绍。

4.3.8.1 TIM1/TIM8 寄存器

下面介绍 TIM1/TIM8 这些高级定时器中使用到的几个重要的寄存器，其他更多关于定时器的资料可以参考《STM32F4xx 参考手册_V4（中文版）.pdf》的第 14 章。

- **控制寄存器 1（TIMx_CR1）**

TIM1/TIM8 的控制寄存器 1 描述如图 4.3.8.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS[1:0]		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 7 **ARPE**：自动重载预装载使能 (Auto-reload preload enable)

0：TIMx_ARR 寄存器不进行缓冲
1：TIMx_ARR 寄存器进行缓冲

位 4 **DIR**：方向 (Direction)

0：计数器递增计数
1：计数器递减计数

注意：当定时器配置为中心对齐模式或编码器模式时，该位为只读状态。

位 0 **CEN**：计数器使能 (Counter enable)

0：禁止计数器
1：使能计数器

注意：只有事先通过软件将 CEN 位置 1，才可以使用外部时钟、门控模式和编码器模式。而触发模式可通过硬件自动将 CEN 位置 1。

图 4.3.8.1.1 TIMx_CR1 寄存器

上图中我们只列出了本章需要用的一些位，其中：位 7（APRE）用于控制自动重载寄存器是否具有缓冲作用，在基本定时器的时候已经讲过，请回顾。在本实验中我们把该位要置 1，这样就算改变 ARR 寄存器的值，该值也不会马上生效，而是等待之前设置的 PWM 完整输出后（发生更新事件）才生效。位 4（DIR）用于配置计数器的计数方向，这里我们默认置 0。位 0（CEN），用于使能计数器的工作，必须要设置该位为 1，才可以开始计数。

- **捕获/比较模式寄存器 1（TIMx_CCMR1）**

TIM1/TIM8 的捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器一般有 2 个：TIMx _CCMR1 和 TIMx _CCMR2。TIMx_CCMR1 控制 CH1 和 CH2，而 TIMx_CCMR2 控制 CH3 和 CH4。TIMx_CCMR1 寄存器描述如图 4.3.8.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2 CE	OC2M[2:0]			OC2 PE	OC2 FE	CC2S[1:0]		OC1 CE	OC1M[2:0]			OC1 PE	OC1 FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.8.1.2 TIMx_CCMR1 寄存器

该寄存器的有些位在不同模式下，功能不一样，我们前面已经说过。比如我们要让 TIM1

的 CH1 输出 PWM 波为例，该寄存器的模式设置位 OC1M[2:0]就是对应着通道 1 的模式设置，此部分由 3 位组成,总共可以配置成 8 种模式，我们使用的是 PWM 模式，所以这 3 位必须设置为 110 或者 111，分别对应 PWM 模式 1 和 PWM 模式 2。这两种 PWM 模式的区别就是输出有效电平的极性相反，这里我们设置为 PWM 模式 1。位 3 OC1PE 是输出比较通道 1 的预装使能，该位需要置 1，另外 CC1S[1:0]用于设置通道 1 的方向（输入/输出）默认设置为 0，就是设置通道作为输出使用。

- **捕获/比较使能寄存器（TIMx_CCER）**

TIM1/TIM8 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关。TIMx_CCER 寄存器描述如图 4.3.8.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CC4P	CC4E	CC3NP	CC3NE	CC3P	CC3E	CC2NP	CC2NE	CC2P	CC2E	CC1NP	CC1NE	CC1P	CC1E
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 4.3.8.1.3 TIM15_CCER 寄存器

该寄存器比较简单，要让 TIM1 的 CH1 输出 PWM 波，这里我们要使能 CC1E 位，该位是通道 1 输入/输出使能位，要想 PWM 从 IO 口输出，这个位必须设置为 1。CC1P 位是设置通道 1 的输出极性，我们设置 0，即 OC1 高电平有效。

- **事件产生寄存器（TIMx_EGR）**

TIM1/TIM8 的事件产生寄存器，该寄存器主要是用户用软件更新各类事件和某些寄存器位的操作寄存器。TIMx_EGR 寄存器描述如图 4.3.8.1.4 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								BG	TG	COMG	CC4G	CC3G	CC2G	CC1G	UG
								w	w	w	w	w	w	w	w

图 4.3.8.1.4 TIMx_EGR 寄存器

UG 位是更新事件的控制位，作用和定时器溢出时产生的更新事件一样，区别是这里是通过软件产生的，而定时器溢出是硬件自己完成的。只有开启了更新中断，这两种方式都可以产更新中断。本实验用到该位去产生软件更新器事件，在需要的时候把 UG 位置 1 即可，会由硬件自动清零。

- **重复计数器寄存器（TIMx_RCR）**

重复计数器寄存器用于设置重复计数器值，因为它具有影子寄存器，所以它本身只是起缓冲作用。当更新事件发生时，该寄存器的值会转移到其影子寄存器中，从而真正起作用。TIMx_RCR 寄存器描述如图 4.3.8.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								REP[7:0]							
								rw	rw	rw	rw	rw	rw	rw	rw

位 7:0 **REP[7:0]**：重复计数器值 (Repetition counter value)

使能预装载寄存器时，用户可通过这些位设置比较寄存器的更新频率（即，从预装载寄存器向活动寄存器周期性传输数据）；使能更新中断时，也可设置更新中断的生成速率。

与 REP_CNT 相关的减计数器每次计数到 0 时，都将生成一个更新事件并且计数器从 REP 值重新开始计数。由于只有生成重复更新事件 U_RC 时，REP_CNT 才会重载 REP 值，因此在生成下一重复更新事件之前，无论向 TIMx_RCR 寄存器写入何值都无影响。

这意味着 PWM 模式下 (REP+1) 相当于：

– 边沿对齐模式下的 PWM 周期数

– 中心对齐模式下的 PWM 半周期数

图 4.3.8.1.5 TIMx_ RCR 寄存器

该寄存器的 REP[7:0]位是低 8 位有效，即最大值 255。因为这个寄存器只是起缓冲作用，如果大家对该寄存器写入值后，想要立即生效，可以通过对 UG 位写 1，产生软件更新事件。

- **捕获/比较寄存器 1/2/3/4（TIMx_CCR1/2/3/4）**

捕获/比较寄存器（TIMx_CCRx），该寄存器总共有 4 个，对应 TIM1/TIM8 的 4 个通道 CH1-CH4。我们使用的是 TIM8 的通道 1，所以来看看 TIMx_CCR1 寄存器描述，如图 4.3.8.1.6 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 15:0 **CCR1[15:0]**：捕获/比较 1 值 (Capture/Compare 1 value)

如果通道 CC1 配置为输出：
CCR1 是捕获/比较寄存器 1 的预装载值。
如果没有通过 TIMx_CCMR 寄存器中的 OC1PE 位来使能预装载功能，写入的数值会被直接传输至当前寄存器中。否则只在发生更新事件时生效（拷贝到实际起作用的捕获/比较寄存器1）。
实际捕获/比较寄存器中包含要与计数器 TIMx_CNT 进行比较并在 OC1 输出上发出信号的值。

如果通道 CC1 配置为输入：
CCR1 为上一个输入捕获 1 事件 (IC1) 发生时的计数器值。

图 4.3.8.1.6 TIMx_CCR1 寄存器

在输出模式下，捕获/比较寄存器影子寄存器的值与 CNT 的值比较，根据比较结果产生相应动作，利用这点，我们通过修改这个寄存器的值，就可以控制 PWM 的占空比了。

- **TIM1/TIM8 断路和死区寄存器（TIMx_BDTR）**

如果是通用定时器，则配置以上说的寄存器就够了，但是如果是高级定时器的通道用作输出时，还必须配置断路和死区寄存器（TIMx_BDTR）的位 MOE，该寄存器各位描述如图 4.3.8.1.7 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOE	AOE	BKP	BKE	OSSR	OSSI	LOCK[1:0]		DTG[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

注意：由于可以根据 LOCK 配置锁定位 AOE、BKP、BKE、OSSI、OSSR 和 DTG[7:0] 的写操作，因此必须在第一次对 TIMx_BDTR 寄存器执行写访问时对这些位进行配置。

位 15 **MOE**：主输出使能 (Main output enable)

只要断路输入变为有效状态，此位便由硬件异步清零。此位由软件置 1，也可根据 AOE 位状态自动置 1。此位仅对配置为输出的通道有效。
0：OC 和 OCN 输出禁止或被强制为空闲状态。
1：如果 OC 和 OCN 输出的相应使能位（TIMx_CCER 寄存器中的 CCxE 和 CCxNE 位）均置 1，则使能 OC 和 OCN 输出。

图 4.3.8.1.7 TIMx_BDTR 寄存器

本实验，我们只需要关注该寄存器的位 15（MOE），要想高级定时器的 PWM 正常输出，则必须设置 MOE 位为 1，否则不会有输出。

4.3.8.2 硬件设计

1. 例程功能

将 TIM8_CH1（由 PI5 复用）输出 PWM，然后利用示波器查看 PWM 波形。上电默认输出 5 个 PWM 波，之后按一下按键 KEY0，就输出 5 个 PWM 波。LED0 闪烁指示程序运行状态。

2. 硬件资源

1）LED 灯：LED0 - PE0
2）独立按键
KEY0 - PE2
3）定时器 8，使用 TIM8 通道 1，将 TIM8_CH1 输出到 PI5。

3. 原理图

定时器属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。我们通过示波器来检测 STM32F407 的定时器的 PWM 输出情况，同时还用按键 KEY0 进行控制。

4.3.8.3 程序设计

本实验用到的 HAL 库驱动请回顾高级定时器 PWM 输出实验的介绍。下面介绍一下定时器输出指定个数 PWM 的配置步骤。

定时器输出指定个数 PWM 配置步骤

1）开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1，对应 IO 是 PI5，它们的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE();            /* 使能定时器 8 */
__HAL_RCC_GPIOI_CLK_ENABLE();           /* 开启 GPIOI 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的 PWM 模式功能时，我们调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数，我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能以及优先级设置等代码。

3）设置定时器为 PWM 模式，输出比较极性，比较值等参数

在 HAL 库中，通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为 PWM1 模式或者 PWM2 模式，根据需求设置输出比较的极性，设置比较值（控制占空比）等。

本实验我们设置 TIM8 的通道 1 为 PWM1 模式，输出比较极性为高。捕获/比较寄存器的值（即比较值）设置为自动重装载值的一半，即 PWM 占空比为 50%。

4）使能定时器更新中断，开启定时器并输出 PWM，配置定时器中断优先级

通过__HAL_TIM_ENABLE_IT 函数使能定时器更新中断。

通过 HAL_TIM_PWM_Start 函数使能定时器并开启输出 PWM。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

5）编写中断服务函数

定时器 8 更新中断服务函数为：TIM8_UP_TIM13_IRQHandler，当发生更新中断的时候，程序就会执行该中断服务函数。HAL 库提供了一个定时器中断公共处理函数 HAL_TIM_IRQHandler，该函数会根据中断类型调用相关的中断回调函数。用户根据自己的需要重定义这些中断回调函数来处理中断程序。本实验我们不使用 HAL 库的中断回调机制，而是把中断程序写在定时器中断服务函数里。详见本章例程源码。

4.3.8.3.1 程序流程图

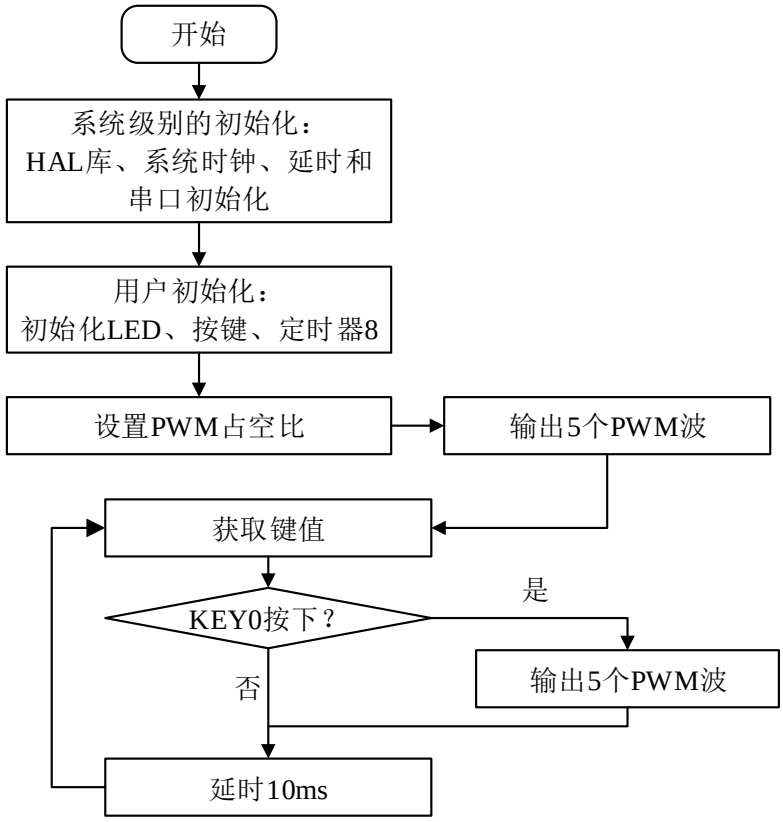


图 4.3.8.3.1.1 高级定时器输出指定个数 PWM 实验程序流程图

4.3.8.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。高级定时器驱动源码包括两个文件：atim.c 和 atim.h。

首先看 atim.h 头文件的几个宏定义：

```
/* TIMX PWM 输出定义 */

#define ATIM_TIMX_NPWM_CHY_GPIO_PORT        GPIOI
#define ATIM_TIMX_NPWM_CHY_GPIO_PIN         GPIO_PIN_5
#define ATIM_TIMX_NPWM_CHY_GPIO_CLK_ENABLE()
        do{  __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)    /* PI 口时钟使能 */

/* TIMX 引脚复用设置
 * 因为 PI5 默认并不是 TIM8 的功能脚，必须开启复用，才能用作 TIM8 的 PWM 输出引脚
 */
#define ATIM_TIMX_NPWM_CHY_GPIO_AF          GPIO_AF3_TIM8

#define ATIM_TIMX_NPWM                      TIM8
#define ATIM_TIMX_NPWM_IRQn                 TIM8_UP_TIM13_IRQn
#define ATIM_TIMX_NPWM_IRQHandler           TIM8_UP_TIM13_IRQHandler
#define ATIM_TIMX_NPWM_CHY                  TIM_CHANNEL_1 /* 通道 Y,1<= Y <=4 */
#define ATIM_TIMX_NPWM_CHY_CLK_ENABLE()
        do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)      /* TIM8 时钟使能 */
```

可以把上面的宏定义分成两部分，第一部分是定时器 8 输入通道 1 对应的 IO 口的宏定义，第二部分则是定时器 8 输入通道 1 的相应宏定义。

下面看 atim.c 的程序，首先是输出指定个数 PWM 初始化函数，其定义如下：

```
/**
 * @brief       高级定时器 TIMX 通道 Y 输出指定个数 PWM 初始化函数
```

```
 * @note
 *            高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *            高级定时器时钟 = 168Mhz
 *            定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *            Ft=定时器工作频率,单位:Mhz
 *
 * @param      arr: 自动重装值
 * @param      psc: 时钟预分频数
 * @retval     无
 */
void atim_timx_npwm_chy_init(uint16_t arr, uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct;

    ATIM_TIMX_NPWM_CHY_GPIO_CLK_ENABLE();                   /* IO 时钟使能 */
    ATIM_TIMX_NPWM_CHY_CLK_ENABLE();                        /* TIMx 时钟使能 */

    gpio_init_struct.Pin = ATIM_TIMX_NPWM_CHY_GPIO_PIN; /* 通道 y 的 GPIO */
    gpio_init_struct.Mode = GPIO_MODE_AF_PP;                /* 复用推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;                    /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
    gpio_init_struct.Alternate = ATIM_TIMX_NPWM_CHY_GPIO_AF;       /* 端口复用 */
    HAL_GPIO_Init(ATIM_TIMX_NPWM_CHY_GPIO_PORT, &gpio_init_struct);

    g_timx_npwm_chy_handle.Instance = ATIM_TIMX_NPWM; /* 定时器 x */
    g_timx_npwm_chy_handle.Init.Prescaler = psc;      /* 定时器分频 */
    g_timx_npwm_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP; /* 递增计数 */
    g_timx_npwm_chy_handle.Init.Period = arr;         /* 自动重装载值 */
    g_timx_npwm_chy_handle.Init.AutoReloadPreload =
                    TIM_AUTORELOAD_PRELOAD_ENABLE;    /* 使能 TIMx_ARR 进行缓冲 */
    HAL_TIM_PWM_Init(&g_timx_npwm_chy_handle);        /* 初始化 PWM */

    timx_oc_npwm_chy.OCMode = TIM_OCMODE_PWM1;        /* 模式选择 PWM1 */
    /* 比较值为重装载值的 50%，即占空比为 50% */
    timx_oc_npwm_chy.Pulse = ( arr + 1 )/2;
    timx_oc_npwm_chy.OCPolarity = TIM_OCPOLARITY_HIGH;  /* 输出比较极性为高 */
    HAL_TIM_PWM_ConfigChannel(&g_timx_npwm_chy_handle, &timx_oc_npwm_chy,
                              ATIM_TIMX_NPWM_CHY);      /* 配置 TIMx 通道 y */
    /* 开启对应 PWM 通道 */
    HAL_TIM_PWM_Start(&g_timx_npwm_chy_handle, ATIM_TIMX_NPWM_CHY);
    /* 开启更新中断 */
    __HAL_TIM_ENABLE_IT(&g_timx_npwm_chy_handle, TIM_IT_UPDATE);

    HAL_NVIC_SetPriority(ATIM_TIMX_NPWM_IRQn, 1, 3);
    HAL_NVIC_EnableIRQ(ATIM_TIMX_NPWM_IRQn);            /* 开启 ITMx 中断 */
}
```

atim_timx_npwm_chy_init 函数包含了输出通道对应 IO 的初始代码、NVIC、使能时钟、定时器基础工作参数和输出模式配置的所有代码。下面来看看该函数的代码内容。

第一部分使能定时器和 GPIO 的时钟。

第二部分调用 HAL_TIM_PWM_Init 函数初始化定时器基础参数，如：ARR 和 PSC 等。

第三部分是定时器输出通道对应的 IO 的初始化。

第四部分调用 HAL_TIM_PWM_ConfigChannel 设置 PWM 模式以及比较值等参数。

第五部分是 NVIC 的初始化，配置抢占优先级、响应优先级和开启 NVIC 定时器中断。

最后是使能更新中断和使能通道输出。

为了方便代码的管理和移植性等，这里就没有使用 HAL_TIM_PWM_MspInit 这个函数来存放使能时钟、GPIO、NVIC 相关的代码，而是全部存放在 atim_timx_npwm_chy_init 函数中。

下面我们看设置 PWM 个数的函数，其定义如下：

```
/* g_npwm_remain 表示当前还剩下多少个脉冲要发送
```

```
 * 每次最多发送 256 个脉冲
 */
static uint32_t g_npwm_remain = 0;

/**
 * @brief       高级定时器 TIMX NPWM 设置 PWM 个数
 * @param       npwm: PWM 的个数, 范围: 1~2^32-1
 * @retval      无
 */
void atim_timx_npwm_chy_set(uint32_t npwm)
{
    if (npwm == 0)
    {
        return;
    }
    g_npwm_remain = npwm;                       /* 保存脉冲个数 */
    /* 产生一次更新事件,在中断里面处理脉冲输出 */
    HAL_TIM_GenerateEvent(&a_timx_pwm_chy_handle, TIM_EVENTSOURCE_UPDATE);
    __HAL_TIM_ENABLE(&a_timx_pwm_chy_handle); /* 使能定时器 TIMX */
}
```

我们要输出多少个周期的 PWM 就用这个函数来设置。该函数作用是把我们设置输出的 PWM 个数的值赋值给静态全局变量 g_npwm_remain，该变量会在更新中断服务函数回调函数中发挥作用。最后对 TIMx_EGR 寄存器 UG 位写 1，产生一次更新事件，并使能定时器。

下面来介绍定时器中断服务函数，其定义如下：

```
/**
 * @brief       高级定时器更新中断服务函数
 * @param       无
 * @retval      无
 */
void ATIM_TIMX_NPWM_IRQHandler(void)
{
    uint16_t npwm = 0;

    /* 以下代码没有使用定时器 HAL 库共用处理函数来处理，而是直接通过判断中断标志位的方式 */
    if(__HAL_TIM_GET_FLAG(&g_timx_npwm_chy_handle, TIM_FLAG_UPDATE) != RESET)
    {
        if (g_npwm_remain >= 256)               /* 还有大于 256 个脉冲需要发送 */
        {
            g_npwm_remain = g_npwm_remain - 256;
            npwm = 256;
        }
        else if (g_npwm_remain % 256)           /* 还有位数（不到 256）个脉冲要发送 */
        {
            npwm = g_npwm_remain % 256;
            g_npwm_remain = 0;                  /* 没有脉冲了 */
        }

        if (npwm)                               /* 有脉冲要发送 */
        {
            /* 设置 RCR 值为 npwm-1, 即 npwm 个脉冲 */
            ATIM_TIMX_NPWM->RCR = npwm - 1;
            /* 产生一次更新事件,在中断里面处理脉冲输出 */
            HAL_TIM_GenerateEvent(&g_timx_npwm_chy_handle,
                                  TIM_EVENTSOURCE_UPDATE);
            __HAL_TIM_ENABLE(&g_timx_npwm_chy_handle);    /* 使能定时器 TIMX */
        }
        else
        {
            /* 关闭定时器 TIMX,使用__HAL_TIM_DISABLE 需要失能通道输出，所以不用 */
            ATIM_TIMX_NPWM->CR1 &= ~(1 << 0);
        }
            /* 清除定时器溢出中断标志位 */
```

```
        __HAL_TIM_CLEAR_IT(&g_timx_npwm_chy_handle, TIM_IT_UPDATE);
    }
}
```

这里我们没有使用 HAL 库的中断回调机制，而是想寄存器操作一样，直接通过判断中断标志位处理中断。通过__HAL_TIM_GET_FLAG 函数宏判断是否发生更新中断，然后进行更新中断的代码处理，最后通过__HAL_TIM_CLEAR_IT 函数宏清除更新中断标志位。

因为重复计数器寄存器 (TIM8_RCR)是 8 位有效的，所以在定时器中断服务函数中首先对全局变量 g_npwm_remain（即我们要输出的 PWM 个数）进行判断，是否大于 256，如果大于 256，那就得分次写入重复计数器寄存器。写入重复计数寄存器后，需要产生软件更新事件把 RCR 寄存器的值更新到 RCR 影子寄存器中，最后一定不要忘记清除定时器更新中断标志位。

在 main 函数里面编写如下代码：

```
int main(void)
{
    uint8_t key = 0;
    uint8_t t = 0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    /* 10Khz 的计数频率,2Hz 的 PWM 频率，占空比为 50% */
    atim_timx_npwm_chy_init(5000 - 1, 16800 - 1);

    atim_timx_npwm_chy_set(5);                  /* 输出 5 个 PWM 波*/

    while (1)
    {
       key = key_scan(0);

        if (key == KEY0_PRES)                   /* KEY0 按下 */
        {
            /* 输出 5 个 PWM 波(控制 TIM8_CH1, 即 PI5 输出 5 个脉冲) */
            atim_timx_npwm_chy_set(5);
        }

        t++;
        if (t > 50)
        {
            t = 0;
            LED0_TOGGLE();                      /* 控制 LED0 闪烁，提示程序运行状态 */
        }
        delay_ms(10);
    }
}
```

先看 atim_timx_npwm_chy_init(5000 - 1, 16800 - 1);这个语句，这两个形参分别设置自动重载寄存器的值为 4999，以及定时器预分频系数为 16799。按照 sys_stm32_clock_init 函数的配置，定时器 8 的时钟频率等于 APB2 总线时钟频率，即 168MHz，可以得到计数器的计数频率是 10KHz。自动重载寄存器的值决定的是 PWM 周期或频率，计数器计 5000 个数所用的时间是 PWM 的周期。在边沿对齐模式下，定时器的溢出周期等于 PWM 的周期。根据定时器溢出时间计算公式，可得：

Tout= ((arr+1)*(psc+1))/Tclk= ((4999+1)*(16799+1))/168000000=0.5s

再由频率是周期的倒数关系得到 PWM 的频率为 2Hz。

我们在 atim_timx_npwm_chy_init 这个函数的里面已经配置了比较值为 ARR 的一半，即 PWM 的占空比为 50%。

初始化完定时器 8 的 PWM 输出之后，就输出 5 个 PWM 波，然后在 while 循环检测按键

是否按下，如果按键 0 按下，则输出 5 个 PWM 波。

4.3.8.4 下载验证

下载代码后，可以看到 LED0 不停闪烁，代表程序正在运行。

下面我们使用正点原子 DS100 手持数字示波器，把 PI5 引脚波形截获，具体如下：

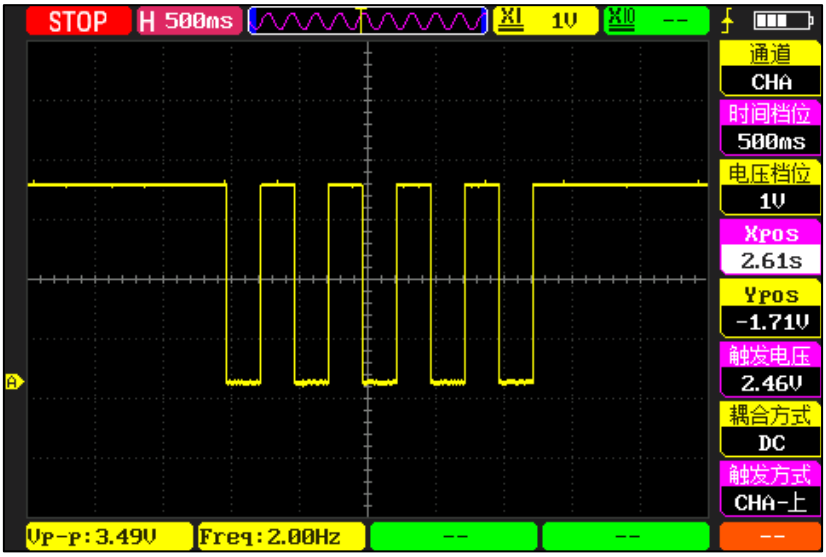


图 4.3.8.4.1 PI5 引脚波形图

图 4.3.8.4.1 中，PWM 脉冲数为 5 个，频率是 2HZ，占空比 50%，请大家自行测量。

4.4 ADC

本小节我们将介绍 STM32F407 的 ADC（Analog-to-digital converters，模数转换器）功能。ADC 应用于电机控制的多个方面，例如：电源电压采集、电机电流采集、驱动板温度采集等。我们通过四个实验来学习 ADC，分别是单通道 ADC 采集实验、单通道 ADC 采集（DMA 读取）实验、多通道 ADC 采集（DMA 读取）实验和内部温度传感器实验。

本小节主要分为如下几个部分：

4.4.1 ADC 简介
4.4.2 单通道 ADC 采集实验
4.4.3 单通道 ADC 采集（DMA 读取）实验
4.4.4 多通道 ADC 采集（DMA 读取）实验
4.4.5 内部温度传感器实验

4.4.1 ADC 简介

ADC 即模拟数字转换器，英文详称 Analog-to-digital converter，可以将外部的模拟信号转换为数字信号。

STM32F407IGT6 芯片拥有 3 个 ADC，都可以独立工作，其中 ADC1 和 ADC2 还可以组成双重模式（提高采样率）。这些 ADC 都是 12 位逐次逼近型的模拟数字转换器，有 19 个通道，可测量 16 个外部信号源、2 个内部信号源和 Vbat 通道的信号。ADC 中的各个通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以以左对齐或者右对齐存储在 16 位数据寄存器中。

STM32F407 的 ADC 主要特性我们可以总结为以下几条：

1、12 位分辨率
2、转换结束、注入转换结束和发生模拟看门狗事件时产生中断
3、单次和连续转换模式
4、自校准
5、带内嵌数据一致性的数据对齐
6、采样间隔可以按通道分别编程
7、规则转换和注入转换均有外部触发选项
8、间断模式
9、双重模式（带 2 个或以上 ADC 的器件）
10、ADC 供电要求：2.4V 到 3.6V
11、ADC 输入范围：$V_{REF-} \leqslant V_{IN} \leqslant V_{REF+}$
12、规则通道转换期间有 DMA 请求产生

下面来介绍 ADC 的框图：

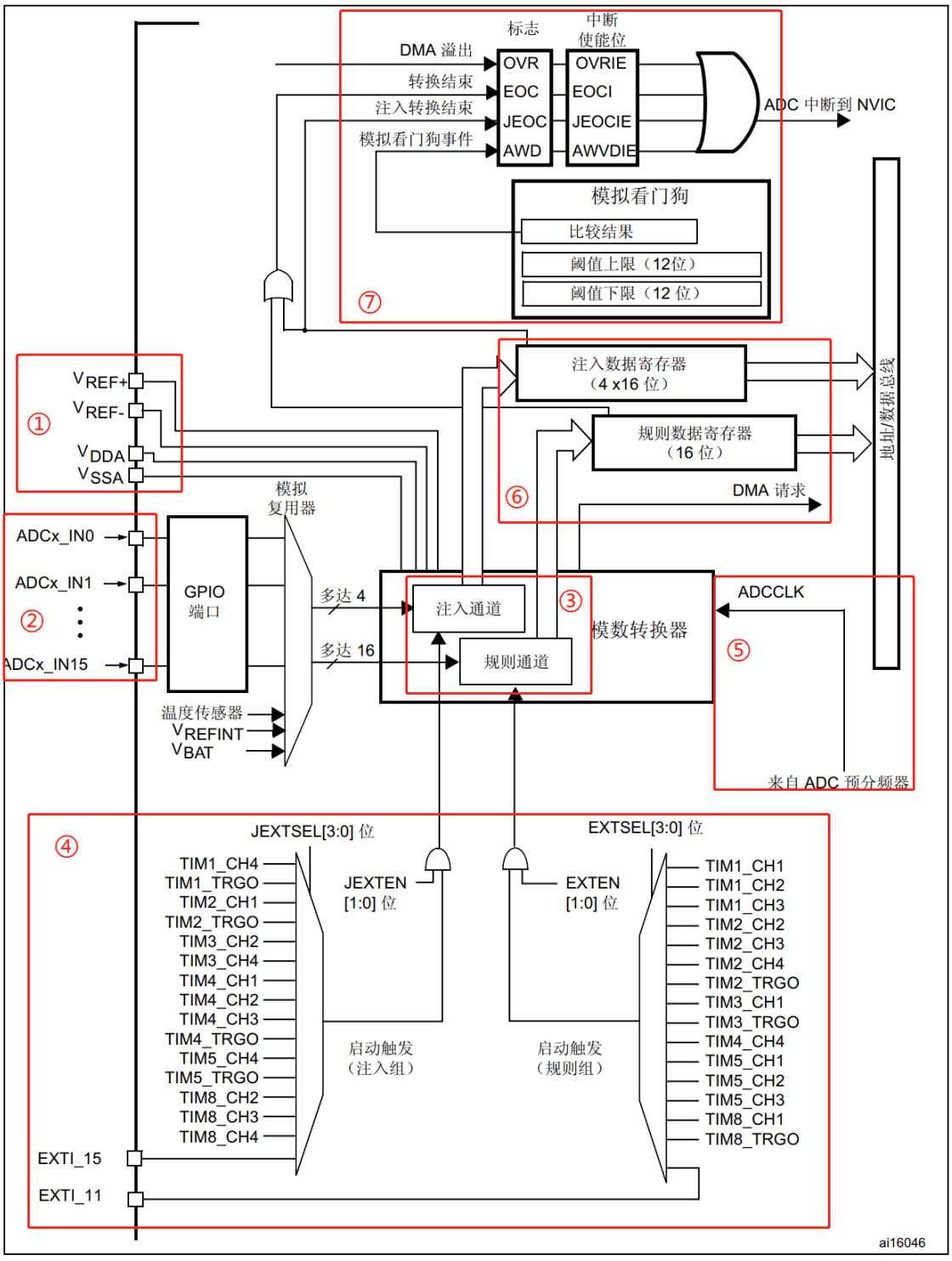


图 4.4.1.1 ADC 框图

图 4.4.1.1 中，我们按照 ADC 的配置流程标记了七处位置，分别如下：

① 输入电压

在前面 ADC 的主要特性也对输入电压有所提及，ADC 输入范围 $V_{REF-} \leqslant V_{IN} \leqslant V_{REF+}$，由 V_{REF-}、V_{REF+}、V_{DDA} 和 V_{SSA} 决定的。下面看一下这几个参数的关系，如图 4.4.1.2 所示：

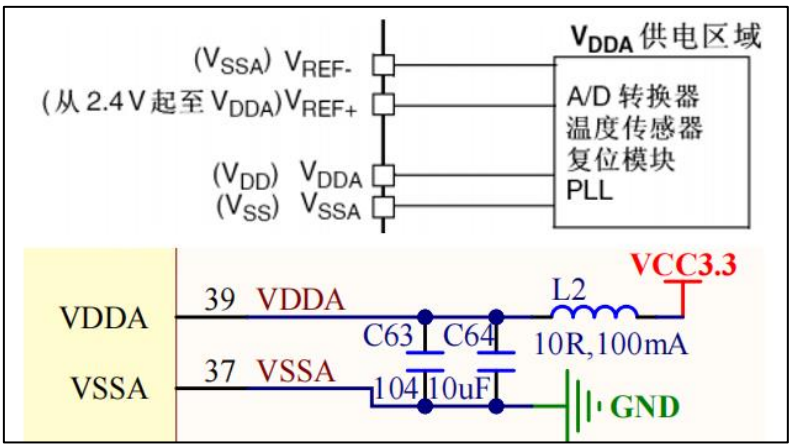


图 4.4.1.2 参数关系图

从上图可以知道，V_{DDA} 和 V_{REF+}接 VCC3.3，而 V_{SSA} 和 V_{REF-}是接地，所以 ADC 的输入范围即 0~3.3V。

② 输入通道

在确定好了 ADC 输入电压后，如何把外部输入的电压输送到 ADC 转换器中呢？在这里引入了通道概念，在前面也提及到了 ADC1 和 ADC2 都有 16 个外部通道和 3 个内部通道。外部通道对应的是上图中的 ADCx_IN0、ADCx_IN1…ADCx_IN15。ADC1 的通道 16 就是内部通道，连接到芯片内部的温度传感器，通道 17 连接到 Vrefint，通道 18 连接到 VBAT，而 ADC2 和 ADC3 的通道 16~18 都是连接到内部的 VSS。具体的 ADC 通道表见表 4.4.1.1 所示：

ADC1	IO	ADC2	IO	ADC3	IO
通道 0	PA0	通道 0	PA0	通道 0	PA0
通道 1	PA1	通道 1	PA1	通道 1	PA1
通道 2	PA2	通道 2	PA2	通道 2	PA2
通道 3	PA3	通道 3	PA3	通道 3	PA3
通道 4	PA4	通道 4	PA4	通道 4	PF6
通道 5	PA5	通道 5	PA5	通道 5	PF7
通道 6	PA6	通道 6	PA6	通道 6	PF8
通道 7	PA7	通道 7	PA7	通道 7	PF9
通道 8	PB0	通道 8	PB0	通道 8	PF10
通道 9	PB1	通道 9	PB1	通道 9	PF3
通道 10	PC0	通道 10	PC0	通道 10	PC0
通道 11	PC1	通道 11	PC1	通道 11	PC1
通道 12	PC2	通道 12	PC2	通道 12	PC2
通道 13	PC3	通道 13	PC3	通道 13	PC3
通道 14	PC4	通道 14	PC4	通道 14	PF4
通道 15	PC5	通道 15	PC5	通道 15	PF5
通道 16	连接内部温度传感器	通道 16	连接内部 VSS	通道 16	连接内部 VSS
通道 17	连接内部 Vrefint	通道 17	连接内部 VSS	通道 17	连接内部 VSS
通道 18	连接内部 VBAT	通道 18	连接内部 VSS	通道 18	连接内部 VSS

表 4.4.1.1 ADC 通道表

③ 转换顺序

当 ADC 的多个通道以任意顺序进行转换就诞生了成组转换，这里有两种成组转换类型：

规则组和注入组。规则组就是图中的规则通道，注入组就是图中的注入通道。为了避免大家对输入通道，以及规则通道和注入通道的理解混淆，后面规则通道以规则组来代称，注入通道以注入组来代称。

规则组最多允许 16 个输入通道进行转换，而注入组最多允许 4 个输入通道进行转换。这里讲解一下规则组和注入组。

规则组（规则通道）

规则组，按字面理解，“规则”就是按照一定的顺序，相当于正常运行的程序，平常用到最多也是规则组。

注入组（注入通道）

注入组，按字面理解，“注入”就是打破原来的状态，相当于中断。当程序执行的时候，中断是可以打断程序的执行。同这个类似，注入组转换可以打断规则组的转换。假如在规则组转换过程中，注入组启动，那么注入组被转换完成之后，规则组才得以继续转换。

为了便于理解，下面看一下规则组和注入组的执行优先级对比图，如图 4.4.1.3 所示：

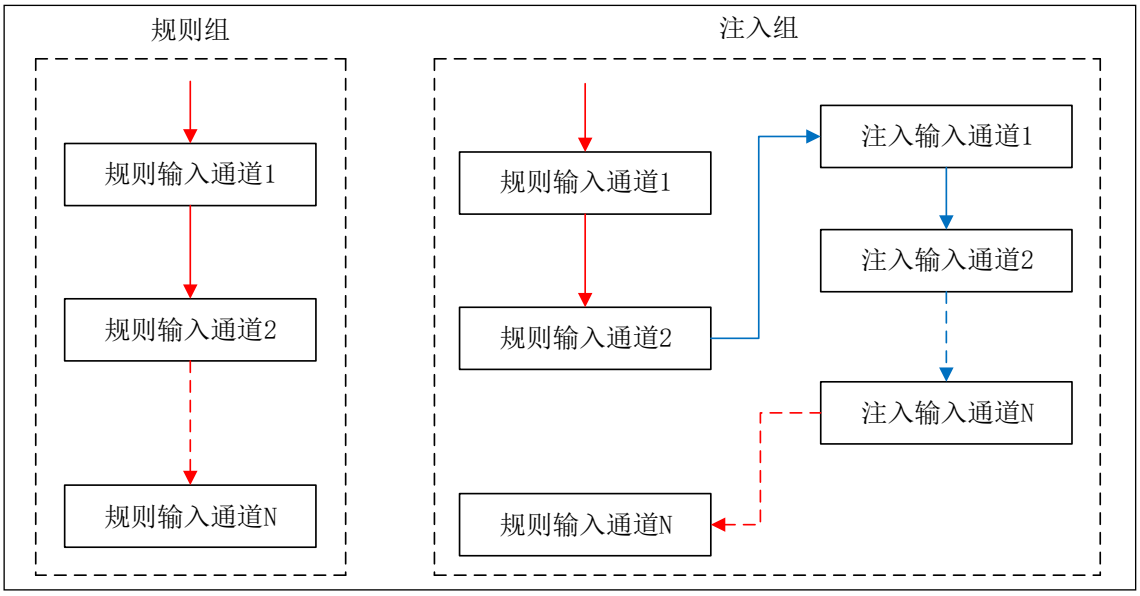


图 4.4.1.3 规则组和注入组的执行优先级对比图

了解了规则组和注入组的概念后，下面来看看它们的转换顺序，即转换序列。转换序列可以分为规则序列和注入序列。下面分别来介绍它们。

规则序列

规则组最多允许 16 个输入通道进行转换，那么就需要设置通道转换的顺序，即规则序列。规则序列寄存器有 3 个，分别为 SQR1、SQR2 和 SQR3。SQR3 控制规则序列中的第 1 个到第 6 个转换；SQR2 控制规则序列中第 7 个到第 12 个转换；SQR1 控制规则序列中第 13 个到第 16 个转换。规则序列寄存器控制关系汇总如表 4.4.1.2 所示：

规则序列寄存器控制关系汇总

寄存器	寄存器位	功能	取值
SQR3	SQ1 [4 : 0]	设置第 1 个转换的通道	输入通道 0~18
	SQ2 [4 : 0]	设置第 2 个转换的通道	输入通道 0~18
	SQ3 [4 : 0]	设置第 3 个转换的通道	输入通道 0~18
	SQ4 [4 : 0]	设置第 4 个转换的通道	输入通道 0~18
	SQ5 [4 : 0]	设置第 5 个转换的通道	输入通道 0~18
	SQ6 [4 : 0]	设置第 6 个转换的通道	输入通道 0~18
SQR2	SQ7 [4 : 0]	设置第 7 个转换的通道	输入通道 0~18

	SQ8 [4 : 0]	设置第 8 个转换的通道	输入通道 0~18
	SQ9 [4 : 0]	设置第 9 个转换的通道	输入通道 0~18
	SQ10 [4 : 0]	设置第 10 个转换的通道	输入通道 0~18
	SQ11 [4 : 0]	设置第 11 个转换的通道	输入通道 0~18
	SQ12 [4 : 0]	设置第 12 个转换的通道	输入通道 0~18
SQR1	SQ13 [4 : 0]	设置第 13 个转换的通道	输入通道 0~18
	SQ14 [4 : 0]	设置第 14 个转换的通道	输入通道 0~18
	SQ15 [4 : 0]	设置第 15 个转换的通道	输入通道 0~18
	SQ16 [4 : 0]	设置第 16 个转换的通道	输入通道 0~18
	SQL [3 : 0]	设置规则序列要转换的通道数	1~16

表 4.4.1.2 规则序列寄存器控制关系汇总表

从上表可以知道，当我们想设置 ADC 的某个输入通道在规则序列的第 1 个转换，只需要把相应的输入通道号的值写入 SQR3 寄存器中的 SQ1[4:0]位即可。例如想让输入通道 5 先进行转换，那么就可以把 5 这个数值写入 SQ1[4:0]位。如果还想让输入通道 8 在第 2 个转换，那么就可以把 8 这个数值写入 SQ2[4:0]位。最后还要设置你的这个规则序列的输入通道个数，只需把输入通道个数写入 SQR1 的 SQL[3:0]位。注意：写入 0 到 SQL[3:0]位，表示这个规则序列有 1 个输入通道的意思，而不是 0 个输入通道。

注入序列

注入序列，跟规则序列差不多，决定的是注入组的顺序。注入组最大允许 4 个通道输入，它的注入序列由 JSQR 寄存器配置。注入序列寄存器 JSQR 控制关系如表 4.4.1.3 所示：

注入序列寄存器控制关系汇总			
寄存器	寄存器位	功能	取值
JSQR	JSQ1 [4 : 0]	设置第 1 个转换的通道	通道 1~4
	JSQ2 [4 : 0]	设置第 2 个转换的通道	通道 1~4
	JSQ3 [4 : 0]	设置第 3 个转换的通道	通道 1~4
	JSQ4 [4 : 0]	设置第 4 个转换的通道	通道 1~4
	JL [1 : 0]	需要转换多少个通道	1~4

表 4.4.1.3 注入序列寄存器控制关系汇总表

注入序列有多少个输入通道，只需要把输入通道个数写入到 JL [1 : 0]位，范围是 0~3。注意：写入 0 表示这个注入序列有一个输入通道，而不是 0 个输入通道。这个内容很简单。编程时容易犯错的是注入序列的转换顺序问题，下面给大家讲解一下。

如果 JL[1 : 0]位的值小于 3，即设置注入序列要转换的通道个数小于 4，则注入序列的转换顺序是从 JSQx[4 : 0]（x=4-JL[1:0]）开始。例如：JL [1 : 0]=10、JSQ4 [4 : 0]= 00100、JSQ3 [4 : 0]= 00011、JSQ2 [4 : 0]= 00111、JSQ1 [4 : 0]= 00010，意味着这个注入序列的转换顺序是：7、3、4，而不是 2、7、3。如果 JL[1 : 0]=00，那么转换顺序是从 JSQ4 [4 : 0]开始。

④ 触发源

在配置好输入通道以及转换顺序后，就可以进行触发转换了。ADC 的触发转换有两种方法：分别是通过 ADON 位或外部事件触发转换。

（1）ADON 位触发转换

当 ADC_CR2 寄存器的 ADON 位为 1 时，再独立给 ADON 位写 1（其它位不能一起改变，这是为了防止误触发），这时会启动转换。这种控制 ADC 启动转换的方式非常简单。

（2）外部触发转换

另一种方法是通过外部事件触发转换，如定时器捕获、EXTI 线和软件触发，可以分为规则组外部触发和注入组外部触发。

规则组外部触发使用方法是：通过 EXTEN[1:0]位使能规则通道的外部触发，并且通过

EXTSEL[3:0]位选择规则组启动转换的触发源。

注入组外部触发使用方法是：通过 JEXTEN [1:0]位使能注入通道的外部触发并且通过 JEXTSEL[3:0]位选择注入组启动转换的触发源。

ADC1 和 ADC2 的触发源是一样的，ADC3 的触发源和 ADC1/2 有所不同，这个需要注意。

⑤ 转换时间

（1）ADC 时钟

在学习转换时间之前，我们先来了解 ADC 时钟。从标号⑤框出来部分可以看到 ADC 时钟是要经过 ADC 预分频器的，那么 ADC 的时钟源是什么？ADC 预分频器的分频系数可以设置的范围又是多少？以及 ADC 时钟频率的最大值又是多少？下面将为大家解答。

ADC 的输入时钟是由 APB2 经过分频产生，分频系数是由 RCC_CFGR 寄存器中的 PPRE2[2:0]进行设置的，可选择 2/4/6/8/16 分频。需要注意的是，STM32F407 的 ADC 的输入时钟频率最大值是 36MHz，如果超过这个值将会导致 ADC 的转换结果准确度下降。

一般我们设置 APB2 为 84MHz。为了不超过 ADC 的最大输入时钟频率 36MHz，我们设置 ADC 的预分频器分频系数为 4，就可以得到 ADC 的输入时钟频率为 84MHz/4，即 21MHz。例程中，我们也是如此设置的。

（2）转换时间

STM32F407 的 ADC 总转换时间的计算公式如下：

T_{CONV} = 采样时间 + 逐次逼近时间（T_{SAR}）

采样时间可通过 ADC_SMPR1 和 ADC_SMPR2 寄存器中的 SMP[2:0]位编程，ADC_SMPR1 控制的是通道 10~18，ADC_SMPR2 控制的是通道 0~9。每个输入通道都支持通过编程来选择不同的采样时间，采样时间可选的范围如下：

- SMP = 000：3 个 ADC 时钟周期
- SMP = 001：15 个 ADC 时钟周期
- SMP = 010：28 个 ADC 时钟周期
- SMP = 011：56 个 ADC 时钟周期
- SMP = 100：84 个 ADC 时钟周期
- SMP = 101：112 个 ADC 时钟周期
- SMP = 110：144 个 ADC 时钟周期
- SMP = 111：480 个 ADC 时钟周期

逐次逼近时间（T_{SAR}）是由分辨率决定的，分辨率通过对 ADCx_CR1 寄存器的 RES[1:0]位进行编程，可将分辨率配置为 12 位、10 位、8 位、6 位。逐次逼近时间和分辨率的对应关系如下所示：

- 12 位： 12 个 ADC 时钟周期
- 10 位： 10 个 ADC 时钟周期
- 8 位 ： 8 个 ADC 时钟周期
- 6 位 ： 6 个 ADC 时钟周期

可以看出，采样时间最小是 3 个时钟周期，逐次逼近时间最小是 6 个时钟周期。下面以我们例程的 ADC 时钟配置为例，来给大家计算一下 ADC 的最短转换时间，计算过程如下：

T_{CONV} =3 个 ADC 时钟周期 +12 个 ADC 时钟周期 =15 个 ADC 时钟周期

例程中，APB2 的时钟是 84MHz，经过 ADC 时钟预分频器的 4 分频后，ADC 时钟频率为 21MHz。代入上式可得到：

$$T_{CONV} = 15 \text{ 个 ADC 时钟周期} = \left(\frac{1}{21000000}\right) * 15\ s = 0.71us$$

⑥ 数据寄存器

ADC 转换完成后的数据输出寄存器。根据转换组的不同，规则组的完成转换的数据输出到 ADC_DR 寄存器，注入组的完成转换的数据输出到 ADC_JDRx 寄存器。假如是使用双重或三重模式，规则组的数据会存放在 ADC_CDR 寄存器。下面给大家简单介绍一下这三个寄存器。

（1）ADC 规则数据寄存器（ADC_DR）

ADC 规则组数据寄存器 ADC_DR 是一个 32 位的寄存器，独立模式时只使用到该寄存器低 16 位保存 ADC1/2/3 的规则转换数据。

因为 ADC 的精度是 12 位的，ADC_DR 寄存器无论高 16 位还是低 16，存放数据的位宽都是 16 位的，所以需要选择数据对齐方式。由 ADC_CR2 寄存器的 ALIGN 位设置数据对齐方式，可选择：右对齐或者左对齐。

细心的朋友可能发现，规则组最多有 16 个输入通道，而 ADC 规则数据寄存器只有一个，如果一个规则组用到好几个通道，数据怎么读取？如果使用多通道转换，那么这些通道的数据也会存放在 DR 里面，按照规则组的顺序，上一个通道转换的数据，会被下一个通道转换的数据覆盖掉，所以当通道转换完成后要及时把数据取走。比较常用的方法是使用 DMA 模式。当规则组的通道转换结束时，就会产生 DMA 请求，这样就可以及时把转换的数据搬运到用户指定的目的地址存放。注意：只有 ADC1 和 ADC3 可以产生 DAM 请求，而由 ADC2 转换的数据可以通过双 ADC 模式，利用 ADC1 的 DMA 功能传输。

（2）双重和三重模式 ADC 规则数据寄存器（ADC_CDR）

双重和三重模式 ADC 规则组数据寄存器 ADC_CDR 是一个 32 位的寄存器，该寄存器的高 16 位用于保存 ADC2 转换的数据，低 16 位用于保存 ADC1 转换的数据。

（3）ADC 注入数据寄存器 x（ADC_JDRx）（x=1~4）

ADC 注入数据寄存器有 4 个，注入组最多有 4 个输入通道，刚好每个通道都有自己对应的数据寄存器。ADC_JDRx 寄存器是 32 位的，低 16 位有效，高 16 位保留，数据也同样需要选择对齐方式。也是由 ADC_CR2 寄存器的 ALIGN 位设置数据对齐方式，可选择：右对齐或者左对齐。

⑦ 中断

ADC 中断可分为三种：规则组转换结束中断、注入组转换结束中断、设置了模拟看门狗状态位中断。它们都有独立的中断使能位，分别由 ADC_CR 寄存器的 EOCIE、JEOCIE、AWDIE 位设置，对应的标志位分别是 EOC、JEOC、AWD。

模拟看门狗中断

模拟看门狗中断发生条件：首先通过 ADC_LTR 和 ADC_HTR 寄存器设置低阈值和高阈值，然后开启了模拟看门狗中断后，当被 ADC 转换的模拟电压低于低阈值或者高于高阈值时，就会产生中断。例如我们设置高阈值是 3.0V，那么模拟电压超过 3.0V 的时候，就会产生模拟看门狗中断，低阈值的情况类似。

DMA 请求

规则组和注入组的转换结束后，除了产生中断外，还可以产生 DMA 请求，把转换好的数据存储在内存里面，防止读取不及时数据被覆盖。

⑧ 单次转换模式和连续转换模式

单次转换模式和连续转换模式在框图中是没有标号，为了更好地学习后续的内容，这里简单给大家讲讲。

（1）单次转换模式

通过将 ADC_CR2 寄存器的 CONT 位置 0 选择单次转换模式。该模式下，ADC 只执行一次转换，由 ADC_CR2 寄存器的 ADON 位启动（只适用于规则组），也可以通过外部触发启动（适用于规则组或注入组）。

如果规则组的一个输入通道被转换，那么转换的数据被储存在 16 位 ADC_DR 寄存器中、EOC（转换结束）标志位被置 1、如果设置了 EOCIE 位，则产生中断，然后 ADC 停止。

如果注入组的一个输入通道被转换，那么转换的数据被储存在 16 位 ADC_DRJx 寄存器中、JEOC（转换结束）标志位被置 1、如果设置了 JEOCIE 位，则产生中断，然后 ADC 停止。

（2）连续转换模式

通过将 ADC_CR2 寄存器的 CONT 位置 1 选择连续转换模式。该模式下，ADC 完成上一个通道的转换后会马上自动地启动下一个通道的转换，由 ADC_CR2 寄存器的 ADON 位启动，也可以通过外部触发启动。

如果规则组的一个输入通道被转换，那么转换的数据被储存在 16 位 ADC_DR 寄存器中、EOC（转换结束）标志位被置 1、如果设置了 EOCIE 位，则产生中断。

如果注入组的一个输入通道被转换，那么转换的数据被储存在 16 位 ADC_DRJx 寄存器

中、JEOC（转换结束）标志位被置 1、如果设置了 JEOCIE 位，则产生中断。

⑨ 扫描模式

扫描模式在框图中是没有标号，为了更好地学习后续的内容，这里简单给大家讲讲。

可以通过 ADC_CR1 寄存器的 SCAN 位配置是否使用扫描模式。如果选择扫描模式，ADC 会扫描所有被 ADC_SQRx 寄存器或 ADC_JSQR 选中的所有通道，并对规则组或者注入组的每个通道执行单次转换，然后停止转换。但如果还设置了 CONT 位，即选择连续转换模式，那么转换不会在选择组的最后一个通道上停止，而是再次从选择组的第一个通道继续转换。

如果设置了 DMA 位，在每次 EOC 后， DMA 控制器把规则组通道的转换数据传输到 SRAM 中。而注入通道转换的数据总是存储在 ADC_JDRx 寄存器中。

到这里我们基本上介绍了 ADC 的大多数基础的知识点，其它知识后面用到会继续补充，如果还有不懂的内容，请参考《STM32F4xx 参考手册(中文版).pdf》的第 11 章。

4.4.2 单通道 ADC 采集实验

本实验我们来学习单通道 ADC 采集。本实验使用规则组单通道的单次转换模式，并且通过软件触发，即由 ADC_CR2 寄存器的 SWSTART 位启动。下面先带大家来了解本实验要配置的寄存器。

4.4.2.1 ADC 寄存器

这里，我们只介绍本实验用到的寄存器的关键位，其它寄存器后续用到会继续介绍。

- **ADC 控制寄存器 1（ADC_CR1）**

ADC 控制寄存器 1 描述如图 4.4.2.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					OVRIE	RES		AWDEN	JAWDEN	Reserved					
					rw	rw	rw	rw	rw						

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISC EN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 8 **SCAN**：扫描模式 (Scan mode)

通过软件将该位置 1 和清零可使能/禁止扫描模式。在扫描模式下，转换通过 ADC_SQRx 或 ADC_JSQRx 寄存器选择的输入。

0：禁止扫描模式

1：使能扫描模式

注意：EOCIE 位置 1 时将生成 EOC 中断：

- *如果 EOCS 位清零，在每个规则组序列转换结束时*
- *如果 EOCS 位置 1，在每个规则通道转换结束时*

注意：JEOCIE 位置 1 时，JEOC 中断仅在最后一个通道转换结束时生成。

图 4.4.2.1.1 ADC_CR1 寄存器

SCAN 位用于选择是否使用扫描模式。本实验我们使用单通道采集，所以没必要选择扫描模式，该位置 0 即可。

- **ADC 控制寄存器 2（ADC_CR2）**

ADC 控制寄存器 2 描述如图 4.4.2.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved	SWSTART	EXTEN		EXTSEL[3:0]				reserved	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved				ALIGN	EOCS	DDS	DMA	Reserved						CONT	ADON
				rw	rw	rw	rw							rw	rw

位 30 **SWSTART**：开始转换规则通道 (Start conversion of regular channels)
通过软件将该位置 1 可开始转换，而硬件会在转换开始后将该位清零。
0：复位状态
1：开始转换规则通道
注意：该位只能在 ADON = 1 时置 1，否则不会启动转换。

位 29:28 **EXTEN**：规则通道的外部触发使能 (External trigger enable for regular channels)
通过软件将这些位置 1 和清零可选择外部触发极性和使能规则组的触发。
00：禁止触发检测
01：上升沿上的触发检测
10：下降沿上的触发检测
11：上升沿和下降沿上的触发检测

位 11 **ALIGN**：数据对齐 (Data alignment)
此位由软件置 1 和清零。请参见*图 38* 和*图 39*。
0：右对齐
1：左对齐

位 1 **CONT**：连续转换 (Continuous conversion)
此位由软件置 1 和清零。该位置 1 时，转换将持续进行，直到该位清零。
0：单次转换模式
1：连续转换模式

位 0 **ADON**：A/D 转换器开启 / 关闭 (A/D Converter ON / OFF)
此位由软件置 1 和清零。
注意： 0：禁止 ADC 转换并转至掉电模式
1：使能 ADC

图 4.4.2.1.2 ADC_CR2 寄存器

该寄存器我们针对性的介绍一些位：ADON 位用于打开或关闭 AD 转换器，还可以用于触发 ADC 转换。CONT 位用于设置单次转换模式还是连续转换模式，本实验我们使用单次转换模式，所以 CONT 位置 0 即可。ALIGN 用于设置数据对齐，我们使用右对齐，所以该位设置为 0。EXTEN[1:0]位用于设置外部触发边沿，本实验使用软件触发，所以设置为 00 即可。SWSTART 位用于开始规则通道的转换，即软件触发转换。

- **ADC 采样事件寄存器 1（ADC_SMPR1）**

ADC 采样事件寄存器 1 描述如图 4.4.2.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					SMP18[2:0]			SMP17[2:0]			SMP16[2:0]			SMP15[2:1]	
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 26:0 **SMPx[2:0]**：通道 X 采样时间选择 (Channel x sampling time selection)
通过软件写入这些位可分别为各个通道选择采样时间。在采样周期期间，通道选择位必须保持不变。
注意： 000：3 个周期　　100：84 个周期
001：15 个周期　　101：112 个周期
010：28 个周期　　110：144 个周期
011：56 个周期　　111：480 个周期

图 4.4.2.1.3 ADC_SMPR1 寄存器

- **ADC 采样事件寄存器 2（ADC_SMPR2）**

ADC 采样事件寄存器 2 描述如图 4.4.2.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]	
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 29:0 **SMPx[2:0]**：通道 X 采样时间选择 (Channel x sampling time selection)
通过软件写入这些位可分别为各个通道选择采样时间。在采样周期期间，通道选择位必须保持不变。

注意：
000：3 个周期
001：15 个周期
010：28 个周期
011：56 个周期
100：84 个周期
101：112 个周期
110：144 个周期
111：480 个周期

图 4.4.2.1.4 ADC_SMPR2 寄存器

ADC 采样时间设置需要由两个寄存器设置，ADC_SMPR1 和 ADC_SMPR2，分别设置通道 10~18 和通道 0~9 的采样时间，每个通道用 3 个位设置。可以看出 ADC 的每个通道的采样时间是支持单独设置的。

一般每个要转换的通道，采样时间建议尽量长一点，以获得较高的准确度，但是这样会降低 ADC 的转换速率，看大家怎么衡量选择了。本实验中，我们设置采样时间是 480 个周期，ADC 精度是 12 位。结合前面介绍过的转换时间公式：

$$T_{CONV} = \text{采样时间} + 12 \text{ 个周期}$$

以及例程中，APB2 的时钟是 84MHz，经过 ADC 时钟预分频器的 4 分频后，ADC 时钟频率为 21MHz。代入上式可得到：

$$T_{CONV} = 480 + 12 \text{ 个 ADC 时钟周期} = \left(\frac{1}{21000000}\right) * 492\ s \approx 23.4us$$

由上式可得，ADC 的转换时间大约是 23.4us。

- **ADC 规则序列寄存器 1（ADC_SQR1）**

ADC 规则序列寄存器共有 3 个，这几个寄存器的功能都差不多，这里我们仅介绍一下 ADC 规则序列寄存器 1（ADC_SQR1），描述如图 4.4.2.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								L[3:0]				SQ16[4:1]			
								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0	SQ15[4:0]					SQ14[4:0]					SQ13[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

位 23:20 **L[3:0]**：规则通道序列长度 (Regular channel sequence length)
通过软件写入这些位可定义规则通道转换序列中的转换总数。
0000：1 次转换
0001：2 次转换
...
1111：16 次转换

位 19:15 **SQ16[4:0]**：规则序列中的第十六次转换 (16th conversion in regular sequence)
通过软件写入这些位，并将通道编号 (0..18) 分配为转换序列中的第十六次转换。

位 14:10 **SQ15[4:0]**：规则序列中的第十五次转换 (15th conversion in regular sequence)

位 9:5 **SQ14[4:0]**：规则序列中的第十四次转换 (14th conversion in regular sequence)

位 4:0 **SQ13[4:0]**：规则序列中的第十三次转换 (13th conversion in regular sequence)

图 4.4.2.1.5 ADC_SQR1 寄存器

L[3:0]用于设置规则组序列的长度，取值范围：0~15，表示规则组的长度是 1~16。本实验只用了 1 个输入通道，所以 L[3:0]位设置为 0000 即可。

SQ13[4:0]~SQ16[4:0]位设置规则组序列的第 13~16 个转换编号，第 1~12 个转换编号的设置请查看 ADC_SQR2 和 ADC_SQR3 寄存器。设置过程非常简单，忘记了请参考前面给大家

整理出来的规则序列寄存器控制关系汇总表。

本实验我们使用单通道，ADC1 通道 3，所以规则组序列里只有一个输入通道，我们将 ADC_SQR3 寄存器的 SQ1[4:0]位的值设置为 3 即可。

- **ADC 规则数据寄存器（ADC_DR）**

ADC 规则数据寄存器描述如图 4.4.2.1.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 15:0 **DATA[15:0]**：规则数据 (Regular data)
这些位为只读。它们包括来自规则通道的转换结果。数据有左对齐和右对齐两种方式，如*图 38* 和*图 39* 所示。

图 4.4.2.1.6 ADC_DR 寄存器

在规则序列中 AD 转换结果都将被存在这个寄存器里面，而注入通道的转换结果被保存在 ADC_JDRx 里面。该寄存器的数据可以通过 ADC_CR2 的 ALIGN 位设置左对齐还是右对齐。在读取数据的时候要注意。

- **ADC 状态寄存器（ADC_SR）**

ADC 状态寄存器描述如图 4.4.2.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										OVR	STRT	JSTRT	JEOC	EOC	AWD
										rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

位 5 **OVR**：溢出 (Overrun)
数据丢失时，硬件将该位置 1（在单一模式或双重/三重模式下）。但需要通过软件清零。溢出检测仅在 DMA = 1 或 EOCS = 1 时使能。
0：未发生溢出
1：发生溢出

位 4 **STRT**：规则通道开始标志 (Regular channel start flag)
规则通道转换开始时，硬件将该位置 1。但需要通过软件清零。
0：未开始规则通道转换
1：已开始规则通道转换

位 3 **JSTRT**：注入通道开始标志 (Injected channel start flag)
注入组转换开始时，硬件将该位置 1。但需要通过软件清零。
0：未开始注入组转换
1：已开始注入组转换

位 2 **JEOC**：注入通道转换结束 (Injected channel end of conversion)
组内所有注入通道转换结束时，硬件将该位置 1。但需要通过软件清零。
0：转换未完成
1：转换已完成

位 1 **EOC**：规则通道转换结束 (Regular channel end of conversion)
规则组通道转换结束后，硬件将该位置 1。通过软件或通过读取 ADC_DR 寄存器将该位清零。
0：转换未完成 (EOCS=0) 或转换序列未完成 (EOCS=1)
1：转换已完成 (EOCS=0) 或转换序列已完成 (EOCS=1)

位 0 **AWD**：模拟看门狗标志 (Analog watchdog flag)
当转换电压超过在 ADC_LTR 和 ADC_HTR 寄存器中编程的值时，硬件将该位置 1。但需要通过软件清零。
0：未发生模拟看门狗事件
1：发生模拟看门狗事件

图 4.4.2.1.7 ADC_SR 寄存器

该寄存器保存了 ADC 转换时的各种状态。本实验我们通过 EOC 位的状态来判断 ADC 转换是否完成，如果查询到 EOC 位被硬件置 1，就可以从 ADC_DR 寄存器中读取转换结果，否则需要等待转换完成。

至此，本章要用到的 ADC 相关寄存器全部介绍完毕了，对于未介绍的部分，请大家参考《STM32F4xx 参考手册_V4（中文版）.pdf》第 11 章相关内容。

4.4.2.2 硬件设计

1. 例程功能

采集 ADC1 通道 3（PA3）上的电压，并在 LCD 模块上面显示 ADC 规则数据寄存器 12 位的转换值以及将该值换算成电压后的电压值。我们使用杜邦线将 PA3 引脚接到 DC 0~3.3V 的电源上（外部电源记得共地），即可采集到 ADC 数据。ADC 采集到的数据和转换后的电压值将在 LCD 屏中显示。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）ADC1 ：通道 3 – PA3

3. 原理图

ADC 属于 STM32F407 内部资源，实际上我们只需要软件设置就可以正常工作，但是还需要我们将待测量的电压源连接到 ADC 通道上，以便 ADC 测量。本实验通过 ADC1 的通道 3（PA3）来采集外部电压值，我们只需要 1 根杜邦线，一端接到 PA3 上，另外一端就接在要测试的电压点(如果是外部电源记得共地)，即可测量电压。**一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。**

4.4.2.3 程序设计

4.4.2.3.1 ADC 的 HAL 库驱动

ADC 在 HAL 库中的驱动代码在 stm32f4xx_hal_adc.c 和 stm32f4xx_hal_adc_ex.c 文件（及其头文件）中。

1. HAL_ADC_Init 函数

ADC 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_ADC_Init(ADC_HandleTypeDef *hadc);
```

- **函数描述：**
  用于初始化 ADC。
- **函数形参：**
  形参 1 是 ADC_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
  ADC_TypeDef                 *Instance;                   /* ADC 寄存器基地址 */
  ADC_InitTypeDef             Init;                        /* ADC 参数初始化结构体 */
  __IO uint32_t               NbrOfCurrentConversionRank; /* 当前转换序列 */
  DMA_HandleTypeDef           *DMA_Handle;                 /* DMA 配置结构体 */
  HAL_LockTypeDef             Lock;                        /* ADC 锁定对象 */
  __IO uint32_t               State;                       /* ADC 工作状态 */
  __IO uint32_t               ErrorCode;                   /* ADC 错误代码 */
}ADC_HandleTypeDef;
```

该结构体定义和其他外设比较类似，我们着重看第二个成员变量 Init 含义，它是结构体 ADC_InitTypeDef 类型，结构体 ADC_InitTypeDef 定义为：

```
typedef struct
{
```

```
  uint32_t ClockPrescaler;                  /* 设置预分频系数，即 PRESC[3:0]位 */
  uint32_t Resolution;                      /* 配置 ADC 的分辨率 */
  uint32_t DataAlign;                       /* 设置数据的对齐方式 */
  uint32_t ScanConvMode;                    /* 扫描模式 */
  uint32_t EOCSelection;                    /* 转换完成标志位 */
  FunctionalState ContinuousConvMode;       /* 开启连续转换模式否则就是单次转换模式 */
  uint32_t NbrOfConversion;                 /* 设置转换通道数目 */
  FunctionalState DiscontinuousConvMode;    /* 是否使用规则通道组间断模式 */
  uint32_t NbrOfDiscConversion;             /* 单次转换通道的数目 */
  uint32_t ExternalTrigConv;                /* ADC 外部触发源选择 */
  uint32_t ExternalTrigConvEdge;            /* ADC 外部触发极性*/
  FunctionalState DMAContinuousRequests;    /* DMA 转换请求模式*/
} ADC_InitTypeDef;
```

1) ClockPrescaler：ADC 预分频系数选择，可选的分频系数为 2、4、6、8、16。
2) Resolution：配置 ADC 的分辨率，可选的分辨率有 12 位、10 位、8 位和 6 位。分辨率越高，转换数据精度越高，转换时间也越长；反之分辨率越低，转换数据精度越低，转换时间也越短。
3) DataAlign：用于设置数据的对齐方式，这里可以选择右对齐或者是左对齐，该参数可选为：ADC_DATAALIGN_RIGHT 和 ADC_DATAALIGN_LEFT。
4) ScanConvMode：配置是否使用扫描。如果是单通道转换使用 ADC_SCAN_DISABLE，如果是多通道转换使用 ADC_SCAN_ENABLE。
5) EOCSelection：可选参数为 ADC_EOC_SINGLE_CONV 和 ADC_EOC_SEQ_CONV，指定转换结束时是否产生 EOS 中断或事件标志。
6) ContinuousConvMode：可选参数为 ENABLE 和 DISABLE，配置自动连续转换还是单次转换。使用 ENABLE 配置为使能自动连续转换；使用 DISABLE 配置为单次转换，转换一次后停止需要手动控制才重新启动转换。
7) NbrOfConversion：设置常规转换通道数目，范围是：1~16。
8) DiscontinuousConvMode：配置是否使用规则通道组间断模式，比如要转换的通道有 1、2、5、7、8、9，那么第一次触发会进行通道 1 和 2，下次触发就是转换通道 5 和 7，这样不连续的转换，依次类推。此参数只有将 ScanConvMode 使能，还有 ContinuousConvMode 失能的情况下才有效，不可同时使能。
9) NbrOfDiscConversion：配置间断模式的通道个数，禁止规则通道组间断模式后，此参数忽略。
10) ExternalTrigConv：外部触发方式的选择，如果使用软件触发，那么外部触发会关闭。
11) ExternalTrigConvEdge：外部触发极性选择，如果使用外部触发，可以选择触发的极性，可选有禁止触发检测、上升沿触发检测、下降沿触发检测以及上升沿和下降沿均可触发检测。
12) DMAContinuousRequests：指定 DMA 请求是否以一次性模式执行(当达到转换次数时，DMA 传输停止)或在连续模式下(DMA 传输无限制，无论转换的数量)。注:在连续模式下，DMA 必须配置为循环模式。否则，当达到 DMA 缓冲区最大指针时将触发溢出。注意:当常规组和注入组都没有转换时(禁用 ADC，或启用 ADC，没有连续模式或可以启动转换的外部触发器)，必须修改此参数。该参数可设置为“启用”或“禁用”。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

2. HAL_ADC_ConfigChannel 函数

ADC 通道配置函数，其声明如下：

```
HAL_StatusTypeDef HAL_ADC_ConfigChannel(ADC_HandleTypeDef *hadc,
                                        ADC_ChannelConfTypeDef *sConfig);
```

- **函数描述：**

  调用了 HAL_ADC_Init 函数配置了相关的功能后，就可以调用此函数配置 ADC 具体通道。

- **函数形参：**

  形参 1 是 ADC_HandleTypeDef 结构体类型指针变量。

形参 2 是 ADC_ChannelConfTypeDef 结构体类型指针变量，用于配置 ADC 采样时间，使用的通道号，单端或者差分方式的配置等。该结构体定义如下：

```
typedef struct
{
  uint32_t Channel;                     /* ADC 转换通道*/
  uint32_t Rank;                        /* ADC 转换顺序 */
  uint32_t SamplingTime;                /* ADC 采样周期 */
  uint32_t Offset;                      /* 保留功能 */
} ADC_ChannelConfTypeDef;
```

1) Channel：ADC 转换通道，范围：0~18。
2) Rank：在规则转换中的规则组的转换顺序，可以选择 1~16。
3) SamplingTime：ADC 的采样周期，最大 480 个 ADC 时钟周期。
4) Offset：未定义，不需要关注。

- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

3. HAL_ADC_Start 函数

ADC 转换启动函数，其声明如下：

```
HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef *hadc);
```

- **函数描述：**
  当配置好 ADC 的基础的功能后，就调用此函数启动 ADC。
- **函数形参：**
  ADC_HandleTypeDef 结构体类型指针变量。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

4. HAL_ADC_PollForConversion 函数

等待 ADC 规则组转换完成函数，其声明如下：

```
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef *hadc,
                                            uint32_t Timeout);
```

- **函数描述：**
  一般先调用 HAL_ADC_Start 函数启动转换，再调用该函数等待转换完成，然后再调用 HAL_ADC_GetValue 函数来获取当前的转换值。
- **函数形参：**
  形参 1 是 ADC_HandleTypeDef 结构体类型指针变量。
  形参 2 是等待转换的等待时间，单位是毫秒（ms）。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

5. HAL_ADC_GetValue 函数

获取常规组 ADC 转换值函数，其声明如下：

```
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef *hadc);
```

- **函数描述：**
  一般先调用 HAL_ADC_Start 函数启动转换，再调用 HAL_ADC_PollForConversion 函数等待转换完成，然后再调用 HAL_ADC_GetValue 函数来获取当前的转换值。
- **函数形参：**
  形参 1 是 ADC_HandleTypeDef 结构体类型指针变量。
- **函数返回值：**
  当前的转换值，uint32_t 类型数据。

单通道 ADC 采集配置步骤

1）开启 ADCx 和通道输入的 GPIO 时钟，配置该 IO 口的模拟输入功能

首先开启 ADCx 的时钟，然后配置 GPIO 为模拟输入。本实验我们默认用到 ADC1 通道 3，对应 IO 是 PA3，它们的时钟开启方法如下：

```
__HAL_RCC_ADC1_CLK_ENABLE();            /* 使能 ADC1 时钟 */
```

```
__HAL_RCC_GPIOA_CLK_ENABLE();           /* 开启 GPIOA 时钟 */
```

IO 口模拟输入功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 ADCx， 配置其工作参数

通过 HAL_ADC_Init 函数来设置 ADCx 时钟分频系数、分辨率、模式、扫描方式、对齐方式等信息。

注意：该函数会调用：HAL_ADC_MspInit 回调函数来存放 ADC 及 GPIO 时钟使能、GPIO 初始化等代码。

3）配置 ADC 通道并启动 AD 转换器

在 HAL 库中，通过 HAL_ADC_ConfigChannel 函数来选择要配置 ADC 的通道，并设置规则序列、采样时间等。

配置好 ADC 通道之后，通过 HAL_ADC_Start 函数启动 AD 转换器。

4）读取 ADC 值

这里选择查询方式读取，在读取 ADC 值之前需要调用 HAL_ADC_PollForConversion 等待上一次转换结束。然后就可以通过 HAL_ADC_GetValue 来读取 ADC 值。

4.4.2.3.2 程序流程图

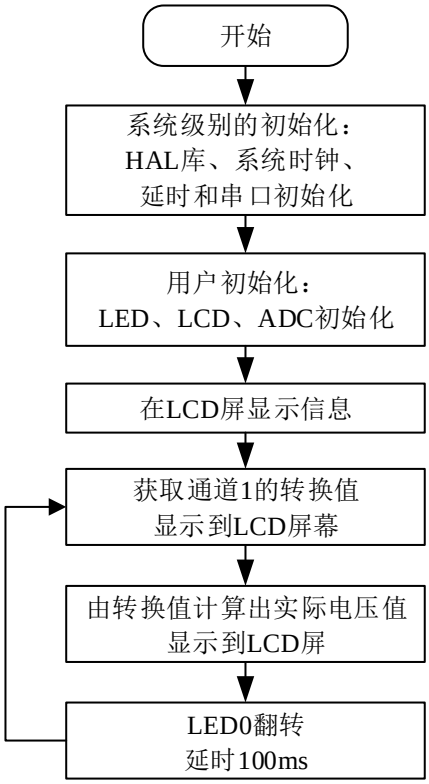


图 4.4.2.3.2.1 单通道 ADC 采集实验程序流程图

4.4.2.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。ADC 驱动源码包括两个文件：adc.c 和 adc.h。

adc.h 文件针对 ADC 及通道引脚定义了一些宏定义，具体如下：

```
/* ADC 及引脚 定义 */

#define ADC_ADCX_CHY_GPIO_PORT                  GPIOA
#define ADC_ADCX_CHY_GPIO_PIN                   GPIO_PIN_3
#define ADC_ADCX_CHY_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)    /* PA 口时钟使能 */
```

```
#define ADC_ADCX                                    ADC1
#define ADC_ADCX_CHY                                ADC_CHANNEL_3
#define ADC_ADCX_CHY_CLK_ENABLE()
        do{ __HAL_RCC_ADC1_CLK_ENABLE(); }while(0)      /* ADC1 时钟使能 */
```

ADC 的通道与引脚的对应关系在《STM32F407IGT6.pdf》数据手册可以查到，我们这里使用 ADC1 的通道 3，在数据手册中的表格为：

PA3	I/O	FT	(4)	USART2_RX/TIM5_CH4 / TIM9_CH2 / TIM2_CH4 / OTG_HS_ULPI_D0 / ETH_MII_COL/ EVENTOUT	ADC123_IN3

表 4.4.2.3.3.1 ADC 通道 3 对应引脚查看表

下面直接开始介绍 adc.c 的程序，首先是 ADC 初始化函数。

```
/**
 * @brief       ADC 初始化函数
 * @param       无
 * @retval      无
 */
void adc_init(void)
{
    g_adc_handle.Instance = ADC_ADCX;
    g_adc_handle.Init.ClockPrescaler =
                            ADC_CLOCKPRESCALER_PCLK_DIV4;  /* 4 分频，21Mhz */
    g_adc_handle.Init.Resolution = ADC_RESOLUTION_12B;   /* 12 位模式 */
    g_adc_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT;   /* 右对齐 */
    g_adc_handle.Init.ScanConvMode = DISABLE;            /* 非扫描模式 */
    g_adc_handle.Init.ContinuousConvMode = DISABLE;      /* 关闭连续转换 */
    g_adc_handle.Init.NbrOfConversion = 1;               /*只使用到一个规则序列 */
    g_adc_handle.Init.DiscontinuousConvMode = DISABLE;   /* 禁止不连续采样模式 */
    g_adc_handle.Init.NbrOfDiscConversion = 0;           /* 不连续采样通道数为 0 */
    g_adc_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START; /* 软件触发 */
    g_adc_handle.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    g_adc_handle.Init.DMAContinuousRequests = DISABLE;   /* 关闭 DMA 请求 */
    HAL_ADC_Init(&g_adc_handle);                         /* 初始化 ADC */
}
```

该函数调用 HAL_ADC_Init 函数配置了选择哪个 ADC、数据对齐方式、是否使用扫描模式等参数。另外 HAL_ADC_Init 函数会调用它的 MSP 回调函数 HAL_ADC_MspInit，该函数用来存放使能 ADC 和通道对应 IO 的时钟和初始化 IO 口等代码，其定义如下：

```
/**
 * @brief       ADC 底层驱动，引脚配置，时钟使能
                此函数会被 HAL_ADC_Init()调用
 * @param       hadc:ADC 句柄
 * @retval      无
 */
void HAL_ADC_MspInit(ADC_HandleTypeDef *hadc)
{
    if(hadc->Instance == ADC_ADCX)
    {
        GPIO_InitTypeDef gpio_init_struct;

        ADC_ADCX_CHY_CLK_ENABLE();                      /* 使能 ADCx 时钟 */
        ADC_ADCX_CHY_GPIO_CLK_ENABLE();                 /* 开启 GPIO 时钟 */

        gpio_init_struct.Pin = ADC_ADCX_CHY_GPIO_PIN;   /* ADC 采集对应 IO */
        gpio_init_struct.Mode = GPIO_MODE_ANALOG;       /* 模拟输入 */
        HAL_GPIO_Init(ADC_ADCX_CHY_GPIO_PORT, &gpio_init_struct);
    }
}
```

可以看到在 HAL_ADC_MspInit 函数中，我们使能了 ADC 和通道对应 IO 时钟、初始化 IO 为模拟输入模式。

接下来要介绍的函数是 adc_channel_set，其定义如下：

```
/**
 * @brief       设置 ADC 通道采样时间
 * @param       adcx : adc 句柄指针,ADC_HandleTypeDef
 * @param       ch   : 通道号, ADC_CHANNEL_0~ADC_CHANNEL_18
 * @param       stime: 采样时间  0~7, 对应关系为:
 *   @arg       ADC_SAMPLETIME_3CYCLES,  3 个 ADC 时钟周期
                ADC_SAMPLETIME_15CYCLES, 15 个 ADC 时钟周期
 *   @arg       ADC_SAMPLETIME_28CYCLES, 28 个 ADC 时钟周期
                ADC_SAMPLETIME_56CYCLES, 56 个 ADC 时钟周期
 *   @arg       ADC_SAMPLETIME_84CYCLES, 84 个 ADC 时钟周期
                ADC_SAMPLETIME_112CYCLES,112 个 ADC 时钟周期
 *   @arg       ADC_SAMPLETIME_144CYCLES,144 个 ADC 时钟周期
                ADC_SAMPLETIME_480CYCLES,480 个 ADC 时钟周期
 * @param       rank: 多通道采集时需要设置的采集编号,
                假设你定义 channle1 的 rank=1，channle2 的 rank=2,
                那么对应你在 DMA 缓存空间的变量数组 AdcDMA[0] 就 i 是 channle1 的转换结
                果，AdcDMA[1]就是通道 2 的转换结果。
                单通道 DMA 设置为 ADC_REGULAR_RANK_1
 * @arg         编号 1~16：ADC_REGULAR_RANK_1~ADC_REGULAR_RANK_16
 * @retval      无
 */
void adc_channel_set(ADC_HandleTypeDef *adc_handle, uint32_t ch,
                                    uint32_t rank, uint32_t stime)
{
    ADC_ChannelConfTypeDef adc_channel;

    adc_channel.Channel = ch;                                /* 通道 */
    adc_channel.Rank = rank;                                 /* 设置采样序列 */
    adc_channel.SamplingTime = stime;                        /* 设置采样时间 */
    HAL_ADC_ConfigChannel( adc_handle, &adc_channel);   /* 通道配置 */
}
```

该函数主要是通过 HAL_ADC_ConfigChannel 函数选择要配置的 ADC 规则组通道，并设置通道的序列号和采样时间。

下面要介绍的是获得 ADC 转换后的结果函数，其定义如下：

```
/**
 * @brief       获取 ADC 值
 * @param       ch: 通道值 0~18，取值范围为：ADC_CHANNEL_0~ADC_CHANNEL_18
 * @retval      无
 */
uint32_t adc_get_result(uint32_t ch)
{
    /* 设置通道，序列和采样时间 */
    adc_channel_set(&g_adc_handle , ch, 1, ADC_SAMPLETIME_480CYCLES);

    HAL_ADC_Start(&g_adc_handle);                          /* 开启 ADC */

    HAL_ADC_PollForConversion(&g_adc_handle, 10);     /* 轮询转换 */

    /* 返回最近一次 ADC1 规则组的转换结果 */
    return (uint16_t)HAL_ADC_GetValue(&g_adc_handle);
}
```

该函数先是调用我们自己定义的 adc_channel_set 函数选择 ADC 通道、设置转换序列号和采样时间等，接着调用 HAL_ADC_Start 启动转换，然后调用 HAL_ADC_PollForConversion 函数等待转换完成，最后调用 HAL_ADC_GetValue 函数获取转换结果。

接下来要介绍的函数是获取 ADC 某通道多次转换结果平均值函数，函数定义如下：

```
/**
```

```
 * @brief       获取通道 ch 的转换值，取 times 次,然后平均
 * @param       ch      : 通道号, 0~18
 * @param       times   : 获取次数
 * @retval      通道 ch 的 times 次转换结果平均值
 */
uint32_t adc_get_result_average(uint32_t ch, uint8_t times)
{
    uint32_t temp_val = 0;
    uint8_t t;

    for (t = 0; t < times; t++)         /* 获取 times 次数据 */
    {
        temp_val += adc_get_result(ch); /* 累加 */
        delay_ms(5);
    }
    return temp_val / times;            /* 返回平均值 */
}
```

该函数用于获取 ADC 多次转换结果的平均值，从而提高准确度。

最后在 main 函数里面编写如下代码：

```
int main(void)
{
    uint16_t adcx;
    float temp;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    adc_init();                                 /* 初始化 ADC */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "ADC TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "ADC1_CH3_VAL:", BLUE);

    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 130, 200, 16, 16, "ADC1_CH3_VOL:0.000V", BLUE);

    while (1)
    {
        /* 获取 ADC 通道的转换值，10 次取平均 */
        adcx = adc_get_result_average(ADC_ADCX_CHY, 10);
        /* 显示 ADC 采样后的平均值 */
        lcd_show_xnum(134, 110, adcx, 5, 16, 0, BLUE);

        /* 获取计算后的带小数的实际电压值，比如 3.1111 */
        temp = (float)adcx * (3.3 / 4096);

        adcx = temp;                /* 赋值整数部分给 adcx 变量，因为 adcx 为 u16 整形 */
        /* 显示电压值的整数部分，3.1111 的话，这里就是显示 3 */
        lcd_show_xnum(134, 130, adcx, 1, 16, 0, BLUE);

        /* 把已经显示的整数部分去掉，留下小数部分，比如 3.1111 - 3 = 0.1111 */
        temp -= adcx;

        /* 小数部分乘以 1000，例如：0.1111 就转换为 111.1，相当于保留三位小数 */
        temp *= 1000;
        /* 显示小数部分（前面转换为了整形显示），这里显示的就是 111 */
        lcd_show_xnum(150, 130, temp, 3, 16, 0X80, BLUE);
```

```
        LED0_TOGGLE();
        delay_ms(100);
    }
}
```

main 函数中，我们在 LCD 上显示一些提示信息后，将每隔 100ms 刷新一次 ADC1 通道 3 的值，并显示读到的 ADC 值（数字量），以及其转换成模拟量后的电压值。同时控制 LED0 闪烁，以提示程序正在运行。ADC 值的显示简单介绍一下：首先在 LCD 固定位置显示了小数点，先计算出整数部分在小数点前面显示，然后计算出小数部分，在小数点后面显示。这样就能在 LCD 上面显示转换结果的整数和小数部分。

4.4.2.4 下载验证

下载代码后，可以看到 LCD 显示如图 4.4.2.4.1 所示：

图 4.4.2.4.1 单通道 ADC 采集实验测试图

上图中，我们使用 1 根杜邦线将 PA3 接到要测试的电压点（如果是外部电源记得共地），即可测出电压值，可测量的电压范围：0~3.3V。LED0 闪烁，提示程序运行。

注意：一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。

4.4.3 单通道 ADC 采集（DMA 读取）实验

本实验我们来学习单通道 ADC 采集（DMA 读取）。本实验使用规则组单通道的连续转换模式，并且通过软件触发，即由 ADC_CR2 寄存器的 SWSTART 位启动。由于使用连续转换模式，所以使用 DMA 读取转换结果的方式。下面先带大家来了解本实验要配置的寄存器。

4.4.3.1 ADC&DMA 寄存器

本实验我们很多的设置和单通道 ADC 采集实验是一样的，所以下面介绍寄存器的时候我们不会继续全部都介绍，而是针对性选择与单通道 ADC 采集实验不同设置的 ADC_CR2 寄存器进行介绍，其他的配置基本一样的。另外因为我们用到 DMA 读取数据，所以还会介绍如何配置相关 DMA 的寄存器。

- **ADC 配置寄存器（ADC_CR2）**

ADCx 配置寄存器描述如图 4.4.3.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved	SWSTART	EXTEN		EXTSEL[3:0]				reserved	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved				ALIGN	EOCS	DDS	DMA	Reserved						CONT	ADON
				rw	rw	rw	rw							rw	rw

位 8 **DMA**：直接存储器访问模式（对于单一 ADC 模式）(Direct memory access mode (for single ADC mode))
此位由软件置 1 和清零。有关详细信息，请参见 DMA 控制器一章。
0：禁止 DMA 模式
1：使能 DMA 模式

位 1 **CONT**：连续转换 (Continuous conversion)
此位由软件置 1 和清零。该位置 1 时，转换将持续进行，直到该位清零。
0：单次转换模式
1：连续转换模式

图 4.4.3.1.1 ADC_CR2 寄存器

ADC_CR2 寄存器中我们主要跟前面设置不同的有两个位，分别如下：

DMA 位用于配置使用 DMA 模式，本实验该位置 1。在单通道 ADC 采集实验中，默认设置为 0，即不使用 DMA 模式，规则组转换的结果存储在 ADC_DR 寄存器，然后通过手动读取 ADC_DR 寄存器的方式得到转换结果。本实验我们使用 ADC 的连续转换模式，并通过 DMA 读取转换结果，这样 DMA 就会自动在 ADC_DR 寄存器中读取转换结果。

CONT 位用于设置单次转换模式还是连续转换模式，本实验我们使用连续转换模式，所以 CONT 位置 1 即可。

这里介绍 ADC_CR2 寄存器的这两个位，其它请参考上一个实验的配置。下面介绍 DMA 一些比较重要的寄存器配置。

- **DMA 数据流 x 外设地址寄存器（DMA_SxPAR）（x = 0…7）**

DMA 数据流 x 外设地址寄存器描述如图 4.4.3.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PAR[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PAR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:0 **PAR[31:0]**：外设地址 (Peripheral address)
读/写数据的外设数据寄存器的基址。
这些位受到写保护，只有 DMA_SxCR 寄存器中的 EN 为“0”时才可以写入。

图 4.4.3.1.2 DMA_SxPAR 寄存器

该寄存器存放的是 DMA 读或者写数据的外设数据寄存器的基址。本实验，我们需要通过 DMA 读取 ADC 转换后存放在 ADC 规则数据寄存器 (ADC_DR) 的结果数据。所以我们需要给 DMA_SxPAR 寄存器写入 ADC_DR 寄存器的地址。这样配置后，DMA 就会从 ADC_DR 寄存器的地址读取 ADC 的转换后的数据到某个内存空间。这个内存空间地址需要我们通过 DMA_SxPAR 寄存器来设置，比如定义一个变量，把这个变量的地址值写入该寄存器。

注意：DMA_SxPAR 寄存器受到写保护，只有 DMA_SxCR 寄存器中的 EN 为“0”时才可以写入，即先要禁止通道开启才可以写入。

- **DMA 数据流 x 存储器地址寄存器（DMA_SxM0AR）（x = 0…7）**

DMA 数据流 x 存储器地址寄存器描述如图 4.4.3.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
M0A[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M0A[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:0 **M0A[31:0]**：存储器 0 地址 (Memory 0 address)
读/写数据的存储区 0 的基址。
这些位受到写保护，只有在以下情况下才可以写入：
– 禁止数据流（DMA_SxCR 寄存器中的位 EN=“0”）或
– 使能数据流（DMA_SxCR 寄存器中的 EN=“1”）并且 DMA_SxCR 寄存器中的位 CT =“1”（在双缓冲区模式下）。

图 4.4.3.1.3 DMA_SxM0AR 寄存器

该寄存器存放的是 DMA 读或者写数据的目标存放的地址。同样的，该寄存器也是受写保护，只有当 DMA_SxCR 的 EN 位为 0 时才可以写入。

- **DMA 数据流 x 数据项数寄存器（DMA_SxNDTR）（x = 0…7）**

DMA 数据流 x 数据项数寄存器描述如图 4.4.3.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NDT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:16 保留，必须保持复位值。

位 15:0 **NDT[15:0]**：要传输的数据项数目 (Number of data items to transfer)
要传输的数据项数目（0 到 65535）。只有在禁止数据流时，才能向此寄存器执行写操作。使能数据流后，此寄存器为只读，用于指示要传输的剩余数据项数。每次 DMA 传输后，此寄存器将递减。
传输完成后，此寄存器保持为零（数据流处于正常模式时），或者在以下情况下自动以先前编程的值重载：
– 以循环模式配置数据流时。
– 通过将 EN 位置“1”来重新使能数据流时
如果该寄存器的值为零，则即使使能数据流，也无法完成任何事务。

图 4.4.3.1.4 DMA_SxNDTR 寄存器

前面的 DMA_SxPAR 寄存器是传输的源地址，而 DMA_SxM0AR 寄存器是传输的目的地址，DMA_SxNDTR 寄存器则是要传输的数据项数目（0 到 65535）。

其他的 DMA 寄存器我们就不一一介绍了，大家有需要的话可以查阅参考手册。

4.4.3.2 硬件设计

1. 例程功能

采集 ADC1 通道 3（PA3）上的电压，并在 LCD 上面显示 ADC 规则数据寄存器 12 位的转换值以及将该值换算成电压后的电压值。我们使用杜邦线将 PA3 引脚接到 DC 0~3.3V 的电源上（外部电源记得共地），即可采集到 ADC 数据。ADC 采集到的数据和转换后的电压值将

在 LCD 屏中显示。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）ADC1 ：通道 3 – PA3
5）DMA（DMA2 通道 0）

3. 原理图

ADC 属于 STM32F407 内部资源，实际上我们只需要软件设置就可以正常工作，但是还需要我们将待测量的电压源连接到 ADC 通道上，以便 ADC 测量。本实验通过 ADC1 的通道 3（PA3）来采集外部电压值，我们只需要 1 根杜邦线，一端接到 PA3 上，另外一端就接在要测试的电压点(如果是外部电源记得共地)，即可测量电压。**一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。**

4.4.3.3 程序设计

4.4.3.3.1 ADC & DMA 的 HAL 库驱动

单通道 ADC 采集实验已经介绍了一部分 ADC 的 HAL 库 API 函数，我们这里要介绍的是 HAL_DMA_Start_IT 和 HAL_ADC_Start_DMA 函数。

1. HAL_DMA_Start_IT 函数

启动 DMA 传输并开启相关中断函数，其声明如下：

```
HAL_StatusTypeDef HAL_DMA_Start_IT (DMA_HandleTypeDef *hdma,
                uint32_t SrcAddress, uint32_t DstAddress, uint32_t DataLength);
```

- **函数描述：**
  用于启动 DMA 传输，并开启相关中断，DMA1 和 DMA2 都是用的这个函数。
- **函数形参：**
  形参 1 是 DMA_HandleTypeDef 结构体类型指针变量。
  形参 2 是 DMA 传输的源地址。
  形参 3 是 DMA 传输的目的地址。
  形参 4 是要传输的数据项数目。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

2. HAL_ADC_Start_DMA 函数

启动 ADC（DMA 传输）方式函数，其声明如下：

```
HAL_StatusTypeDef HAL_ADC_Start_DMA(ADC_HandleTypeDef* hadc,
                                    uint32_t *pData, uint32_t Length);
```

- **函数描述：**
  用于启动 ADC，转换结果以 DMA 传输方式读取。
- **函数形参：**
  形参 1 是 ADC_HandleTypeDef 结构体类型指针变量。
  形参 2 是 ADC 采样数据传输的目的地址。
  形参 3 是要传输的数据项数目。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。
- **注意事项：**
  HAL_ADC_Start_DMA 和 HAL_DMA_Start 都是配置并启动 DMA 的函数，区别在于：HAL_ADC_Start_DMA 比较局限性，只是用于启动 ADC 的数据传输。HAL_DMA_Start 则适

用性较广泛，任何能使用 DMA 传输的场景都可以用该函数启动。实际应用中看个人的需求选择用哪个函数。在例程中我们使用的是 HAL_ADC_Start_DMA 函数。如果我们需要使用 DMA 中断，我们还可以使用 HAL_DMA_Start_IT 函数，使能了 DMA 全部的中断。

单通道 ADC 采集（DMA 读取）配置步骤

1）开启 ADCx 和通道输入的 GPIO 时钟，配置该 IO 口的模拟输入功能

首先开启 ADCx 的时钟，然后配置 GPIO 为模拟输入。本实验我们默认用到 ADC1 通道 3，对应 IO 是 PA3，它们的时钟开启方法如下：

```
__HAL_RCC_ADC1_CLK_ENABLE ();            /* 使能 ADC1 时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE();            /* 开启 GPIOA 时钟 */
```

IO 口模拟输入功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 ADCx， 配置其工作参数

通过 HAL_ADC_Init 函数来设置 ADCx 时钟分频系数、分辨率、模式、扫描方式、对齐方式等信息。

注意：该函数会调用：HAL_ADC_MspInit 回调函数来存放 ADC 及 GPIO 时钟使能、GPIO 初始化等代码。我们也可以不存放在这个函数里，本实验就没用到这个 MSP 回调函数。

3）配置 ADC 通道并启动 AD 转换器

在 HAL 库中，通过 HAL_ADC_ConfigChannel 函数来选择要配置 ADC 的通道，并设置规则序列、采样时间等。

配置好 ADC 通道之后，通过 HAL_ADC_Start_DMA 函数启动 AD 转换器。

4）初始化 DMA

通过 HAL_DMA_Init 函数初始化 DMA，包括配置通道，外设地址，存储器地址，传输数据量等。

HAL 库为了处理各类外设的 DMA 请求，在调用相关函数之前，需要调用一个宏定义标识符，来连接 DMA 和外设句柄。这个宏定义为__HAL_LINKDMA。

5）使能 DMA 对应数据流中断，配置 DMA 中断优先级，使能 ADC，使能并启动 DMA

通过 HAL_ADC_Start_DMA 函数开启 ADC 转换，通过 DMA 传输结果。

通过 HAL_DMA_Start_IT 函数启动 DMA 读取，使能 DMA 中断。

通过 HAL_NVIC_EnableIRQ 函数使能 DMA 数据流中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

6）编写中断服务函数

DMA 的每个数据流都有一个中断服务函数，比如 DMA2_ Stream4 的中断服务函数为 DMA2_Stream4_IRQHandler。简单的做法就是，在对应的中断服务函数里面，通过判断相关的中断标志位的方式，完成中断逻辑代码，最后清除该中断标志位，本实验的做法就是如此。

除此之外，我们还可以通过调用 HAL 库所提供的 DMA 中断公用处理函数 HAL_DMA_IRQHandler，然后定重新义相关的中断回调处理函数。

4.4.3.3.2 程序流程图

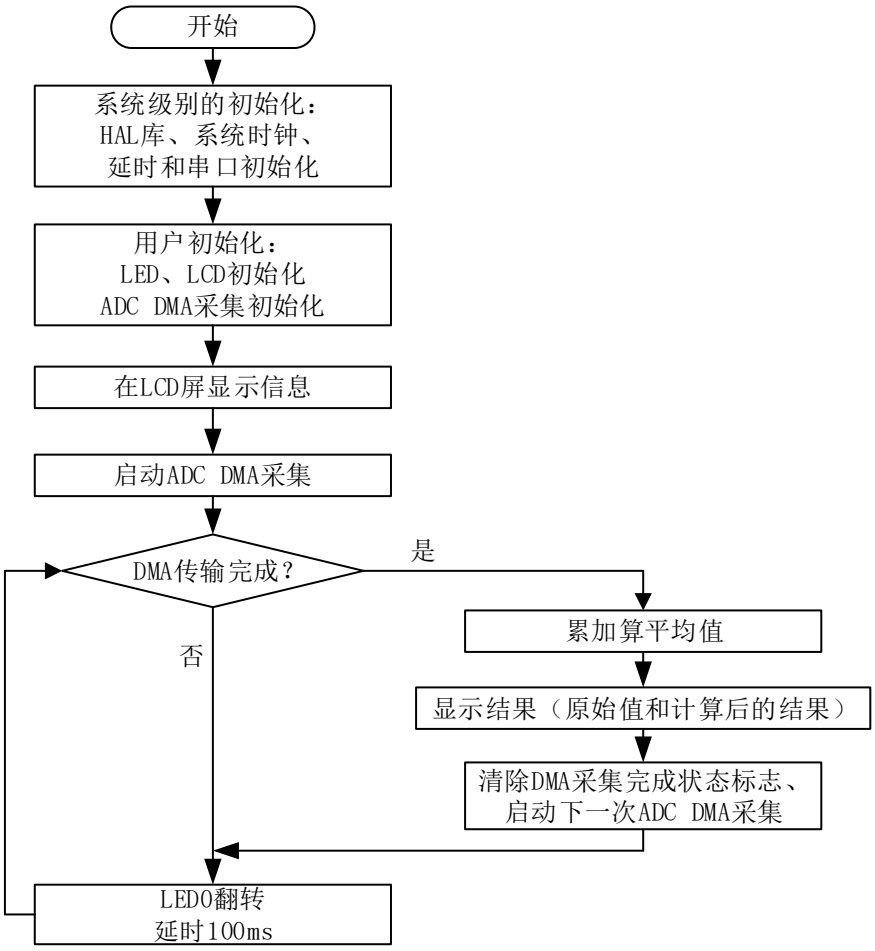


图 4.4.3.3.2.1 单通道 ADC 采集（DMA 读取）实验程序流程图

4.4.3.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。由于本实验用到 DMA，所以在 adc.h 头文件定义了以下一些宏定义：

```
/* ADC 单通道/多通道 DMA 采集 DMA 及通道 定义 */

#define ADC_ADCX_DMASx                  DMA2_Stream4
#define ADC_ADCX_DMASx_Chanel           DMA_CHANNEL_0
#define ADC_ADCX_DMASx_IRQn             DMA2_Stream4_IRQn
#define ADC_ADCX_DMASx_IRQHandler       DMA2_Stream4_IRQHandler

/* 判断 DMA2 Stream4 传输完成标志，这是一个假函数形式,只能用在 if 等语句里面 */
#define ADC_ADCX_DMASx_IS_TC()
        ( __HAL_DMA_GET_FLAG(&g_dma_adc_handle, DMA_FLAG_TCIF0_4) )

/* 清除 DMA2 Stream4 传输完成标志 */
#define ADC_ADCX_DMASx_CLR_TC()
        do{ __HAL_DMA_CLEAR_FLAG(&g_dma_adc_handle, DMA_FLAG_TCIF0_4); }
        while(0)
```

下面给大家介绍 adc.c 文件里面的函数，首先是 ADC DMA 读取初始化函数。

```
/**
 * @brief       ADC DMA 读取 初始化函数
 * @param       mar : 存储器地址
 * @retval      无
 */
```

```
void adc_dma_init(uint32_t mar)
{
    GPIO_InitTypeDef gpio_init_struct;
    ADC_ChannelConfTypeDef adc_ch_conf = {0};

    ADC_ADCX_CHY_CLK_ENABLE();                              /* 使能 ADCx 时钟 */
    ADC_ADCX_CHY_GPIO_CLK_ENABLE();                         /* 开启 GPIO 时钟 */

    /* 大于 DMA2 的基地址，则为 DMA2 的数据流通道了 */
    if ((uint32_t)ADC_ADCX_DMASx > (uint32_t)DMA2)
    {
        __HAL_RCC_DMA2_CLK_ENABLE();                        /* DMA2 时钟使能 */
    }
    else
    {
        __HAL_RCC_DMA1_CLK_ENABLE();                        /* DMA1 时钟使能 */
    }

    /* 设置 AD 采集通道对应 IO 引脚工作模式 */
    gpio_init_struct.Pin = ADC_ADCX_CHY_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_ANALOG;
    HAL_GPIO_Init(ADC_ADCX_CHY_GPIO_PORT, &gpio_init_struct);

    /* 初始化 DMA */
    g_dma_adc_handle.Instance = ADC_ADCX_DMASx;             /* 设置 DMA 数据流 */
    g_dma_adc_handle.Init.Channel = DMA_CHANNEL_0;          /* 设置 DMA 通道 */
    /* 从外设到存储器模式 */
    g_dma_adc_handle.Init.Direction = DMA_PERIPH_TO_MEMORY;
    g_dma_adc_handle.Init.PeriphInc = DMA_PINC_DISABLE;     /* 外设非增量模式 */
    g_dma_adc_handle.Init.MemInc = DMA_MINC_ENABLE;         /* 存储器增量模式 */
    /* 外设数据长度:16 位 */
    g_dma_adc_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    /* 存储器数据长度:16 位 */
    g_dma_adc_handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    g_dma_adc_handle.Init.Mode = DMA_NORMAL;                /* 外设流控模式 */
    g_dma_adc_handle.Init.Priority = DMA_PRIORITY_MEDIUM;   /* 中等优先级 */
    HAL_DMA_Init(&g_dma_adc_handle);

    g_adc_dma_handle.Instance = ADC_ADCX;
    /* 4 分频，21Mhz */
    g_adc_dma_handle.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
    g_adc_dma_handle.Init.Resolution = ADC_RESOLUTION_12B; /* 12 位模式 */
    g_adc_dma_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT; /* 右对齐 */
    g_adc_dma_handle.Init.ScanConvMode = DISABLE;           /* 非扫描模式 */
    g_adc_dma_handle.Init.ContinuousConvMode = ENABLE;      /* 开启连续转换 */
    /* 本实验用到 1 个规则通道序列 */
    g_adc_dma_handle.Init.NbrOfConversion = 1;
    /* 禁止不连续采样模式 */
    g_adc_dma_handle.Init.DiscontinuousConvMode = DISABLE;
    g_adc_dma_handle.Init.NbrOfDiscConversion = 0;       /* 不连续采样通道数为 0 */
    g_adc_dma_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;  /* 软件触发 */
    g_adc_dma_handle.Init.DMAContinuousRequests = ENABLE;   /* 开启 DMA 请求 */
    HAL_ADC_Init(&g_adc_dma_handle);                        /* 初始化 ADC */

    /* 把 ADC 和 DMA 连接起来 */
    __HAL_LINKDMA(&g_adc_dma_handle, DMA_Handle, g_dma_adc_handle);

    /* 配置 ADC 通道 */
    adc_ch_conf.Channel = ADC_ADCX_CHY;                     /* 通道 */
    adc_ch_conf.Rank = 1;                                   /* 序列 */
    adc_ch_conf.SamplingTime = ADC_SAMPLETIME_480CYCLES;    /* 采样时间 */
```

```
    HAL_ADC_ConfigChannel(&g_adc_dma_handle, &adc_ch_conf); /* 通道配置 */

    /* 配置 DMA 数据流请求中断优先级 */
    HAL_NVIC_SetPriority(ADC_ADCX_DMASx_IRQn, 3, 3);
    HAL_NVIC_EnableIRQ(ADC_ADCX_DMASx_IRQn);

    /* 启动 DMA，并开启中断 */
    HAL_DMA_Start_IT(&g_dma_adc_handle, (uint32_t)&ADC1->DR, mar, 0);
    HAL_ADC_Start_DMA(&g_adc_dma_handle,&mar,0); /* 开启 ADC，通过 DMA 传输结果 */
}
```

adc_dma_init 函数包含了输入通道对应 IO 的初始代码、NVIC、使能时钟、ADC 时钟预分频系数、ADC 工作参数和 ADC 通道配置等代码。下面来看看该函数的代码内容。

第一部分，使能 ADC、DMA 和 GPIO 的时钟。

第二部分，设置 ADC 采集通道对应 IO 引脚工作模式。

第三部分，初始化 DMA、ADC，配置 ADC 时钟预分频系数为 4，得到 ADC 的输入时钟频率是 21MHz。

第四部分，通过__HAL_LINKDMA 宏定义将 DMA 相关的配置关联到 ADC 的句柄中。

第五部分，配置 ADC 通道。

第六部分，配置 DMA 数据流请求中断优先级，并使能该中断。

第七部分，启动 DMA 并开启 DMA 中断，以及启动 ADC 并通过 DMA 传输转换结果。

为了方便代码的管理和移植性等，这里就没有使用 HAL_ADC_MspInit 这个函数来存放使能时钟、GPIO、NVIC 相关的代码，而是全部存放在 adc_dma_init 函数中。

接下来给大家介绍使能一次 ADC DMA 传输函数，其定义如下：

```
/**
 * @brief       使能一次 ADC DMA 传输
 * @param       cndtr: DMA 传输的次数
 * @retval      无
 */
void adc_dma_enable(uint16_t cndtr)
{
    __HAL_ADC_DISABLE(&g_adc_dma_handle);           /* 先关闭 ADC */

    __HAL_DMA_DISABLE(&g_dma_adc_handle);           /* 关闭 DMA 传输 */
    g_dma_adc_handle.Instance->NDTR = cndtr;        /* 重设 DMA 传输数据量 */
    __HAL_DMA_ENABLE(&g_dma_adc_handle);            /* 开启 DMA 传输 */

    __HAL_ADC_ENABLE(&g_adc_dma_handle);            /* 重新启动 ADC */
    ADC_ADCX->CR2 |= 1 << 30;                       /* 启动规则转换通道 */
}
```

该函数的某些部分我们使用寄存器来操作，因为用 HAL 库操作会对 adc_dma_init 配置好的某些参数修改。HAL_DMA_Start_IT 函数已经配置好了 DMA 传输的源地址和目标地址，本函数只需要调用 g_dma_adc_handle.Instance->CNDTR=cndtr;语句给 DMA_SxNDTR 寄存器写入要传输的数据量，然后启动 DMA 就可以传输了。

下面介绍的是 ADC DMA 采集中断服务函数，函数定义如下：

```
/**
 * @brief       ADC DMA 采集中断服务函数
 * @param       无
 * @retval      无
 */
void ADC_ADCX_DMASx_IRQHandler(void)
{
    if (ADC_ADCX_DMASx_IS_TC())
    {
        g_adc_dma_sta = 1;          /* 标记 DMA 传输完成 */
        ADC_ADCX_DMASx_CLR_TC();    /* 清除 DMA2 数据流 4 传输完成中断 */
    }
}
```

在该函数里，通过判断 DMA 传输完成标志位是否是 1，是 1 就给 g_adc_dma_sta 变量赋值为 1，标记 DMA 传输完成，最后清除 DMA 的传输完成标志位。

最后在 main.c 里面编写如下代码：

```
#define ADC_DMA_BUF_SIZE        100             /* ADC DMA 采集 BUF 大小 */
uint16_t g_adc_dma_buf[ADC_DMA_BUF_SIZE];  /* ADC DMA BUF */

extern uint8_t g_adc_dma_sta;               /* DMA 传输状态标志,0,未完成;1,已完成 */


int main(void)
{
    uint16_t i, adcx;
    uint32_t sum;
    float temp;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */

    adc_dma_init((uint32_t)&g_adc_dma_buf); /* 初始化 ADC DMA 采集 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "ADC DMA TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "ADC1_CH3_VAL:", BLUE);

    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 130, 200, 16, 16, "ADC1_CH3_VOL:0.000V", BLUE);

    adc_dma_enable(ADC_DMA_BUF_SIZE);       /* 启动 ADC DMA 采集 */

    while (1)
    {
        if (g_adc_dma_sta == 1)
        {
            /* 计算 DMA 采集到的 ADC 数据的平均值 */
            sum = 0;

            for (i = 0; i < ADC_DMA_BUF_SIZE; i++)          /* 累加 */
            {
                sum += g_adc_dma_buf[i];
            }

            adcx = sum / ADC_DMA_BUF_SIZE;                  /* 取平均值 */

            /* 显示结果 */
            lcd_show_xnum(134, 110, adcx, 4, 16, 0, BLUE); /* 显示采样后的原始值 */

            /* 获取计算后的带小数的实际电压值，比如 3.1111 */
            temp = (float)adcx * (3.3 / 4096);

            /* 赋值整数部分给 adcx 变量，因为 adcx 为 u16 整形 */
            adcx = temp;

            /* 显示电压值的整数部分，3.1111 的话，这里就是显示 3 */
            lcd_show_xnum(134, 130, adcx, 1, 16, 0, BLUE);

            /* 把已经显示的整数部分去掉，留下小数部分，比如 3.1111-3=0.1111 */
            temp -= adcx;
```

```
            /* 小数部分乘以 1000，例如：0.1111 就转换为 111.1，相当于保留三位小数 */
            temp *= 1000;
            /* 显示小数部分（前面转换为了整形显示），这里显示的就是 111. */
            lcd_show_xnum(150, 130, temp, 3, 16, 0X80, BLUE);

            g_adc_dma_sta = 0;                      /* 清除 DMA 采集完成状态标志 */
            adc_dma_enable(ADC_DMA_BUF_SIZE);       /* 启动下一次 ADC DMA 采集 */
        }
        LED0_TOGGLE();
        delay_ms(100);
    }
}
```

此部分代码，和单通道 ADC 采集实验十分相似，只是这里使能了 DMA 传输数据，DMA 传输的数据存放在 g_adc_dma_buf 数组里，这里我们对数组的数据取平均值，减少误差。在 LCD 屏显示结果的处理和单通道 ADC 采集实验一样。首先我们在 LCD 固定位置显示了小数点，然后后面计算步骤中，先计算出整数部分在小数点前面显示，然后计算出小数部分，在小数点后面显示。这样就在 LCD 上面显示转换结果的整数和小数部分。

4.4.3.4 下载验证

下载代码后，LED0 闪烁，提示程序运行，可以看到 LCD 显示如图 4.4.3.4.1 所示：

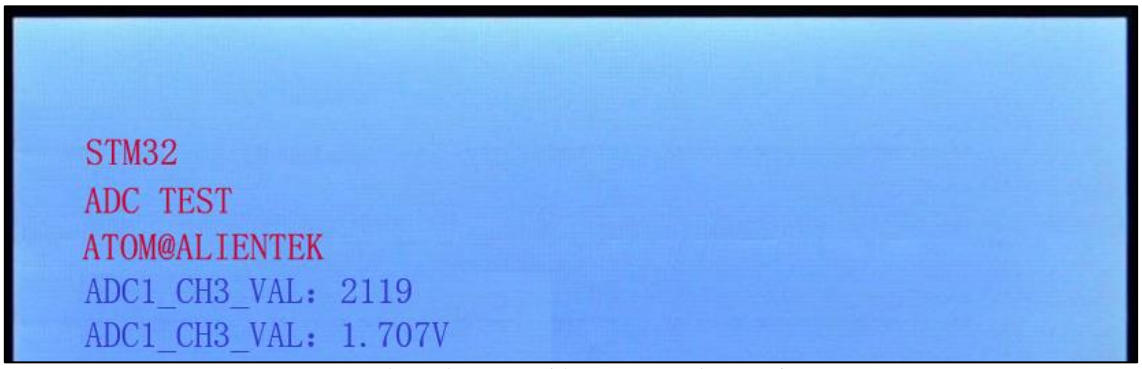


图 4.4.3.4.1 单通道 ADC 采集（DMA 读取）实验测试图

上图中，我们使用 1 根杜邦线将 PA3 接到要测试的电压点（如果是外部电源记得共地），即可测出电压值，可测量的电压范围：0~3.3V。LED0 闪烁，提示程序运行。

注意：一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。

4.4.4 多通道 ADC 采集（DMA 读取）实验

本实验我们来学习多通道 ADC 采集（DMA 读取）。本实验使用规则组多通道的连续转换模式，并且通过软件触发，即由 ADC_CR2 寄存器的 SWSTART 位启动。由于使用连续转换模式，所以使用 DMA 读取转换结果的方式。下面先带大家来了解本实验要配置的寄存器。

4.4.4.1 ADC 寄存器

本实验我们很多的设置和单通道 ADC 采集（DMA 读取）实验是一样的，所以下面介绍寄存器的时候我们不会继续全部都介绍，而是针对性选择与单通道 ADC 采集（DMA 读取）实验不同设置的 ADC_SQRx 寄存器进行介绍，其他的配置基本一样的。另外我们用到 DMA 读取数据，配置上和单通道 ADC 采集（DMA 读取）实验是一样的。

ADC 规则序列寄存器有四个（ADC_SQR1~ ADC_SQR3），具体怎么配置，需要看我们用多少个通道，比如本实验我们使用 3 个通道同时采集 ADC 数据，具体配置如下：

- **ADC 规则序列寄存器 1（ADC_SQR1）**

ADC 规则序列寄存器 1 描述如图 4.4.4.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								L[3:0]				SQ16[4:1]			
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0	SQ15[4:0]					SQ14[4:0]					SQ13[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

位 31:24 保留，必须保持复位值。

位 23:20 **L[3:0]**：规则通道序列长度 (Regular channel sequence length)
通过软件写入这些位可定义规则通道转换序列中的转换总数。
0000：1 次转换
0001：2 次转换
...
1111：16 次转换

位 19:15 **SQ16[4:0]**：规则序列中的第十六次转换 (16th conversion in regular sequence)
通过软件写入这些位，并将通道编号 (0..18) 分配为转换序列中的第十六次转换。

位 14:10 **SQ15[4:0]**：规则序列中的第十五次转换 (15th conversion in regular sequence)

位 9:5 **SQ14[4:0]**：规则序列中的第十四次转换 (14th conversion in regular sequence)

位 4:0 **SQ13[4:0]**：规则序列中的第十三次转换 (13th conversion in regular sequence)

图 4.4.4.1.1 ADC_SQR1 寄存器

L[3:0]位用于设置规则序列的长度，取值范围：0~15，表示规则序列长度为 1~16。本实验使用到 3 个通道，所以设置这几个位的值为 2 即可。

SQ13[4:0]~SQ16[4:0]位设置规则组序列的第 13~16 个转换编号，第 1~12 个转换编号的设置请查看 ADC_SQR2 和 ADC_SQR3 寄存器。设置过程非常简单，忘记了请参考前面给大家整理出来的规则序列寄存器控制关系汇总表。

下面我们来看看本实验是怎么设置的：SQ1[4:0]位赋值为 3、SQ2[4:0]位赋值为 4、SQ3[4:0]位赋值为 5，即规则序列 1 到 3 分别对应的输入通道是 3 到 5。SQ1~SQ3 位都是在 ADC_SQR3 寄存器中配置。

4.4.4.2 硬件设计

1. 例程功能

使用 ADC1 采集（DMA 读取）通道 3\4\5\的电压，在 LCD 模块上面显示对应的 ADC 转换值以及换算成电压后的电压值。可以使用杜邦线连接 PA3\PA4\PA5 到你想测量的电压源（0~3.3V），然后通过 LCD 显示的电压值。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）ADC1 ：通道 3 - PA3、通道 4 - PA4、通道 5 - PA5
5）DMA（DMA2 通道 0）

3. 原理图

ADC 和 DMA 属于 STM32F407 内部资源，实际上我们只需要软件设置就可以正常工作，但是还需要我们将待测量的电压源连接到 ADC 通道上，以便 ADC 测量。本实验，我们通过 ADC1 的通道 3\4\5 来采集外部电压值，并通过 DMA 来读取。

4.4.4.3 程序设计

4.4.4.3.1 ADC 的 HAL 库驱动

本实验用到的 ADC 的 HAL 库 API 函数前面都介绍过，具体调用情况请看程序解析部分。下面介绍多通道 ADC 采集（DMA 读取）配置步骤。

多通道 ADC 采集（DMA 读取）配置步骤

1）开启 ADCx 和通道输入的 GPIO 时钟，配置该 IO 口的模拟输入功能

首先开启 ADCx 的时钟，然后配置 GPIO 为模拟输入模式。本实验我们默认用到 ADC1 通道 3、4、5，对应 IO 是 PA3、PA4 和 PA5，它们的时钟开启方法如下：

```
__HAL_RCC_ADC1_CLK_ENABLE ();          /* 使能 ADC1 时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE();          /* 开启 GPIOA 时钟 */
```

IO 口模拟输入功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 ADCx， 配置其工作参数

通过 HAL_ADC_Init 函数来设置 ADCx 时钟分频系数、分辨率、模式、扫描方式、对齐方式等信息。

注意：该函数会调用：HAL_ADC_MspInit 回调函数来存放 ADC 及 GPIO 时钟使能、GPIO 初始化等代码。我们也可以不存放在这个函数里，本实验就没用到这个 MSP 回调函数。

3）配置 ADC 通道并启动 AD 转换器

在 HAL 库中，通过 HAL_ADC_ConfigChannel 函数来选择要配置 ADC 的通道，并设置规则序列、采样时间等。

配置好 ADC 通道之后，通过 HAL_ADC_Start_DMA 函数启动 AD 转换器。

4）初始化 DMA

通过 HAL_DMA_Init 函数初始化 DMA，包括配置通道，外设地址，存储器地址，传输数据量等。

HAL 库为了处理各类外设的 DMA 请求，在调用相关函数之前，需要调用一个宏定义标识符，来连接 DMA 和外设句柄。这个宏定义为__HAL_LINKDMA。

5）使能 DMA 对应数据流中断，配置 DMA 中断优先级，使能 ADC，使能并启动 DMA

通过 HAL_ADC_Start_DMA 函数开启 ADC 转换。
通过 HAL_DMA_Start_IT 函数启动 DMA 读取，使能 DMA 中断。
通过 HAL_NVIC_EnableIRQ 函数使能 DMA 数据流中断。
通过 HAL_NVIC_SetPriority 函数设置中断优先级。

6）编写中断服务函数

DMA 的每个数据流都有一个中断服务函数，比如 DMA2_ Stream4 的中断服务函数为 DMA2_Stream4_IRQHandler。简单的做法就是，在对应的中断服务函数里面，通过判断相

关的中断标志位的方式，完成中断逻辑代码，最后清除该中断标志位，本实验的做法就是如此。

除此之外，我们还可以通过调用 HAL 库所提供的 DMA 中断公用处理函数 HAL_DMA_IRQHandler，然后定重新义相关的中断回调处理函数。

4.4.4.3.2 程序流程图

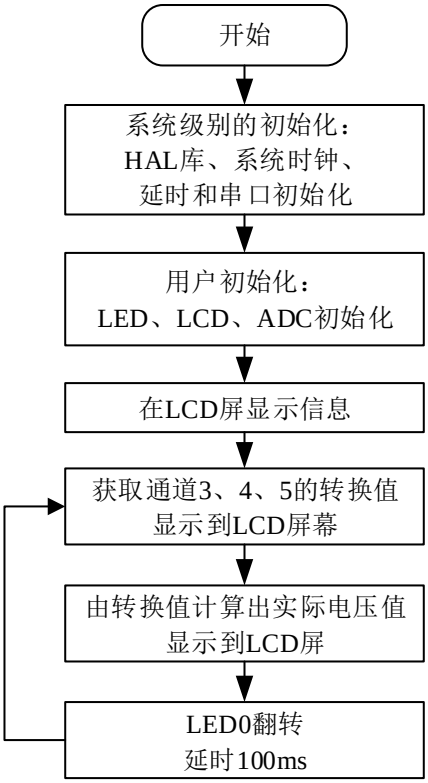


图 4.4.4.3.2.1 多通道 ADC 采集（DMA 读取）实验程序流程图

4.4.4.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。本实验的 adc.h 头文件只是在单通道 ADC 采集（DMA 读取）实验代码的基础上添加了两个通道的宏定义，具体定义如下：

```
/* ADC 及引脚 定义 */

#define ADC_ADCX_CH3_GPIO_PORT              GPIOA
#define ADC_ADCX_CH3_GPIO_PIN               GPIO_PIN_3
#define ADC_ADCX_CH3_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)         /* PA 口时钟使能 */

#define ADC_ADCX_CH4_GPIO_PORT              GPIOA
#define ADC_ADCX_CH4_GPIO_PIN               GPIO_PIN_4
#define ADC_ADCX_CH4_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)         /* PA 口时钟使能 */

#define ADC_ADCX_CH5_GPIO_PORT              GPIOA
#define ADC_ADCX_CH5_GPIO_PIN               GPIO_PIN_5
#define ADC_ADCX_CH5_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)         /* PA 口时钟使能 */

#define ADC_ADCX                            ADC1
#define ADC_ADCX_CH3                        ADC_CHANNEL_3
#define ADC_ADCX_CH4                        ADC_CHANNEL_4
```

```
#define ADC_ADCX_CH5                        ADC_CHANNEL_5
#define ADC_ADCX_CHY_CLK_ENABLE()
        do{ __HAL_RCC_ADC1_CLK_ENABLE(); }while(0)     /* ADC1 时钟使能 */

#define ADC_CH_NUM                          3               /* 需要转换的通道数目 */
#define ADC_COLL                            1000            /* 单采集次数 */
#define ADC_SUM                             ADC_CH_NUM * ADC_COLL /* 总采集次数 */

/* ADC 单通道/多通道 DMA 采集 DMA 及通道 定义*/
#define ADC_ADCX_DMASx                      DMA2_Stream4    /* DMA2 数据流 4 */
#define ADC_ADCX_DMASx_Chanel               DMA_CHANNEL_0   /* DMA 通道 0 */
#define ADC_ADCX_DMASx_IRQn                 DMA2_Stream4_IRQn
#define ADC_ADCX_DMASx_IRQHandler           DMA2_Stream4_IRQHandler

/* 判断 DMA2 Stream4 传输完成标志，这是一个假函数形式,只能用在 if 等语句里面 */
#define ADC_ADCX_DMASx_IS_TC()
        ( __HAL_DMA_GET_FLAG(&g_dma_nch_adc_handle, DMA_FLAG_TCIF0_4) )

/* 清除 DMA2 Stream4 传输完成标志 */
#define ADC_ADCX_DMASx_CLR_TC()
        do{ __HAL_DMA_CLEAR_FLAG(&g_dma_nch_adc_handle,DMA_FLAG_TCIF0_4); }
        while(0)
```

在 adc.h 头文件中，我们添加了 ADC1 通道 4、5 相关的宏定义。

下面开始介绍 adc.c 的函数，首先看 ADC 的 N 通道(3 通道) DMA 读取初始化函数，其定义如下：

```
/**
 * @brief       ADC N 通道(3 通道) DMA 读取 初始化函数
 * @param       无
 * @retval      无
 */
void adc_nch_dma_init(void)
{
    GPIO_InitTypeDef gpio_init_struct = {0};
    ADC_ChannelConfTypeDef sConfig = {0};

    ADC_ADCX_CHY_CLK_ENABLE();                  /* 使能 ADCx 时钟 */
    ADC_ADCX_CH3_GPIO_CLK_ENABLE();             /* 开启 GPIO 时钟 */
    ADC_ADCX_CH4_GPIO_CLK_ENABLE();
    ADC_ADCX_CH5_GPIO_CLK_ENABLE();

    /* 大于 DMA2 基地址，则为 DMA2 的数据流通道了 */
    if ((uint32_t)ADC_ADCX_DMASx > (uint32_t)DMA2)
    {
        __HAL_RCC_DMA2_CLK_ENABLE();            /* DMA2 时钟使能 */
    }
    else
    {
        __HAL_RCC_DMA1_CLK_ENABLE();            /* DMA1 时钟使能 */
    }

    /* 设置 ADC1 通道 3~5 对应的 IO 口模拟输入 */
    gpio_init_struct.Pin = ADC_ADCX_CH3_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_ANALOG;
    HAL_GPIO_Init(ADC_ADCX_CH3_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = ADC_ADCX_CH4_GPIO_PIN;
    HAL_GPIO_Init(ADC_ADCX_CH4_GPIO_PORT, &gpio_init_struct);

    gpio_init_struct.Pin = ADC_ADCX_CH5_GPIO_PIN;
    HAL_GPIO_Init(ADC_ADCX_CH5_GPIO_PORT, &gpio_init_struct);

    /* DMA 配置 */
    g_dma_nch_adc_handle.Instance = ADC_ADCX_DMASx;
```

```
    g_dma_nch_adc_handle.Init.Channel = DMA_CHANNEL_0;       /* 设置 DMA 通道 */
    /* 外设到存储器模式 */
    g_dma_nch_adc_handle.Init.Direction = DMA_PERIPH_TO_MEMORY;
    g_dma_nch_adc_handle.Init.PeriphInc = DMA_PINC_DISABLE; /* 外设非增量模式 */
    g_dma_nch_adc_handle.Init.MemInc = DMA_MINC_ENABLE;      /* 存储器增量模式 */
     /* 外设数据长度:16 位 */
    g_dma_nch_adc_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    /* 存储器数据长度:16 位 */
    g_dma_nch_adc_handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    g_dma_nch_adc_handle.Init.Mode = DMA_NORMAL;              /* 外设流控模式 */
    g_dma_nch_adc_handle.Init.Priority = DMA_PRIORITY_MEDIUM; /* 中等优先级 */
    HAL_DMA_Init(&g_dma_nch_adc_handle);

    /* 将 DMA 与 adc 联系起来 */
    __HAL_LINKDMA(&g_adc_nch_dma_handle,DMA_Handle,g_dma_nch_adc_handle);

    g_adc_nch_dma_handle.Instance = ADC_ADCX;
    /* 4 分频，21Mhz */
    g_adc_nch_dma_handle.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
    g_adc_nch_dma_handle.Init.Resolution = ADC_RESOLUTION_12B; /* 12 位模式 */
    g_adc_nch_dma_handle.Init.ScanConvMode = ENABLE;           /* 使能扫描模式 */
    g_adc_nch_dma_handle.Init.ContinuousConvMode = ENABLE;     /* 使能连续转换 */
    /* 禁止规则通道组间断模式 */
    g_adc_nch_dma_handle.Init.DiscontinuousConvMode = DISABLE;
    g_adc_nch_dma_handle.Init.ExternalTrigConvEdge =
                                ADC_EXTERNALTRIGCONVEDGE_NONE;   /* 使用软件触发 */
    g_adc_nch_dma_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    g_adc_nch_dma_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT;  /* 右对齐 */
    /* 本实验用到 3 个规则通道序列 */
    g_adc_nch_dma_handle.Init.NbrOfConversion = ADC_CH_NUM;
    /* 开启 DMA 连续转换 */
    g_adc_nch_dma_handle.Init.DMAContinuousRequests = ENABLE;
    HAL_ADC_Init(&g_adc_nch_dma_handle);

    sConfig.Channel = ADC_ADCX_CH3;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);  /* 配置 ADC 通道 3 */

    sConfig.Channel = ADC_ADCX_CH4;
    sConfig.Rank = 2;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);  /* 配置 ADC 通道 4 */

    sConfig.Channel = ADC_ADCX_CH5;
    sConfig.Rank = 3;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);  /* 配置 ADC 通道 5 */

    /* 配置中断 */
    HAL_NVIC_SetPriority(ADC_ADCX_DMASx_IRQn, 2, 1);
    HAL_NVIC_EnableIRQ(ADC_ADCX_DMASx_IRQn);

    /* 开启 ADC，使用 DMA 读取 */
    HAL_ADC_Start_DMA(&g_adc_nch_dma_handle,(uint32_t *)g_adc_value,ADC_SUM);
}
```

adc_nch_dma_init 函数包含了输出通道对应 IO 的初始代码、NVIC、使能时钟、ADC 时钟预分频系数、ADC 工作参数和 ADC 通道配置等代码。大部分代码和单通道 ADC 采集(DMA 读取)实验一样，下面来看看该函数的代码内容。

第一部分，使能 ADC、DMA 和 GPIO 的时钟。

第二部分，设置 ADC 采集通道对应 IO 引脚工作模式，这里用到 6 个通道。

第三部分，初始化 DMA、ADC，配置 ADC 时钟预分频系数为 4，得到 ADC 的输入时钟频率是 21MHz。

第四部分，通过__HAL_LINKDMA 宏定义将 DMA 相关的配置关联到 ADC 的句柄中。

第五部分，配置 ADC 通道，这里有 3 个通道需要配置。

第六部分，配置 DMA 数据流请求中断优先级，并使能该中断。

第七部分，启动 DMA 并开启 DMA 中断，以及启动 ADC 并通过 DMA 传输转换结果。

为了方便代码的管理和移植性等，这里就没有使用 HAL_ADC_MspInit 这个函数来存放使能时钟、GPIO、NVIC 相关的代码，而是全部存放在 adc_dma_init 函数中。

接下来我们看 ADC 采集中断回调函数，其定义如下：

```
/**
 * @brief       ADC 采集中断回调函数
 * @param       * hadc: ADC 句柄
 * @retval      无
 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    HAL_ADC_Stop_DMA(hadc);                         /* 关闭 DMA 传输 */

    adc[0] = adc_get_result_average(0);             /* 计算 ADC 平均值 */
    adc[1] = adc_get_result_average(1);
    adc[2] = adc_get_result_average(2);

    HAL_ADC_Start_DMA(&g_adc_nch_dma_handle, (uint32_t*)&g_adc_value, ADC_SUM);
}
```

进入中断回调函数之后，先关闭 DMA 传输，然后进行 3 个通道的 ADC 平均值计算，计算完平均值后再重新开启 DMA。ADC 平均值计算函数在单通道 ADC 采集实验已经介绍过，这里不再赘述。

在 main.c 里面编写如下代码：

```
extern int16_t adc[3];                          /* ADC 平均值 */
float g_adc_u_value[ADC_CH_NUM] = {0};          /* 存储 ADC 转换后的电压值 */

int main(void)
{
    uint16_t i,adcx;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    adc_nch_dma_init();                         /* ADC DMA 初始化 */

    lcd_show_string(30,  50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30,  70, 200, 16, 16, "ADC 3CH DMA TEST", RED);
    lcd_show_string(30,  90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    lcd_show_string(30, 110, 200, 16, 16, "ADC1_CH3_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 130, 200, 16, 16, "ADC1_CH3_VOL:0.000V", BLUE);

    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 150, 200, 16, 16, "ADC1_CH4_VAL:", BLUE);
    lcd_show_string(30, 170, 200, 16, 16, "ADC1_CH4_VOL:0.000V", BLUE);

    lcd_show_string(30, 190, 200, 16, 16, "ADC1_CH5_VAL:", BLUE);
    /* 先在固定位置显示小数点 */
    lcd_show_string(30, 210, 200, 16, 16, "ADC1_CH5_VOL:0.000V", BLUE);

    while (1)
    {
        for(i = 0; i<3; i++)
```

```
        {
            /* 显示 ADC 采样后的平均值 */
            lcd_show_xnum(134, 110+(i*40), adc[i], 5, 16, 0, BLUE);
            g_adc_u_value[i] = (float)adc[i] * 3.3f /4096;     /* 计算电压值 */

            adcx = g_adc_u_value[i];
            /* 显示电压值的整数部分，3.1111 的话，这里就是显示 3 */
            lcd_show_xnum(134, 130+(i*40), adcx, 1, 16, 0, BLUE);
             /* 把已经显示的整数部分去掉，留下小数部分，比如 3.1111-3=0.1111 */
            g_adc_u_value[i] -= adcx;
             /* 小数部分乘以 1000，例如：0.1111 就转换为 111.1，相当于保留三位小数 */
            g_adc_u_value[i] *= 1000;
            /* 显示小数部分（前面转换为了整形显示），这里显示的就是 111. */
            lcd_show_xnum(150, 130+(i*40), g_adc_u_value[i],3,16,0X80, BLUE);
        }
        LED1_TOGGLE();
        delay_ms(100);
    }
}
```

main 函数中，我们初始化了 ADC 和 DMA，在 LCD 模块上显示一些提示信息后，每隔 100ms 刷新一次 ADC1 通道 3、4、5 的值，并显示读到的 ADC 值（数字量），以及它们转换成模拟量后的电压值。同时控制 LED0 闪烁，以提示程序正在运行。ADC 值的显示简单介绍一下：首先在 LCD 固定位置显示了小数点，先计算出整数部分在小数点前面显示，然后计算出小数部分，在小数点后面显示。这样就能在 LCD 上面显示转换结果的整数和小数部分。

4.4.4.4 下载验证

下载代码后，LED0 闪烁，提示程序运行。可以看到 LCD 显示如图 4.4.4.4.1 所示：

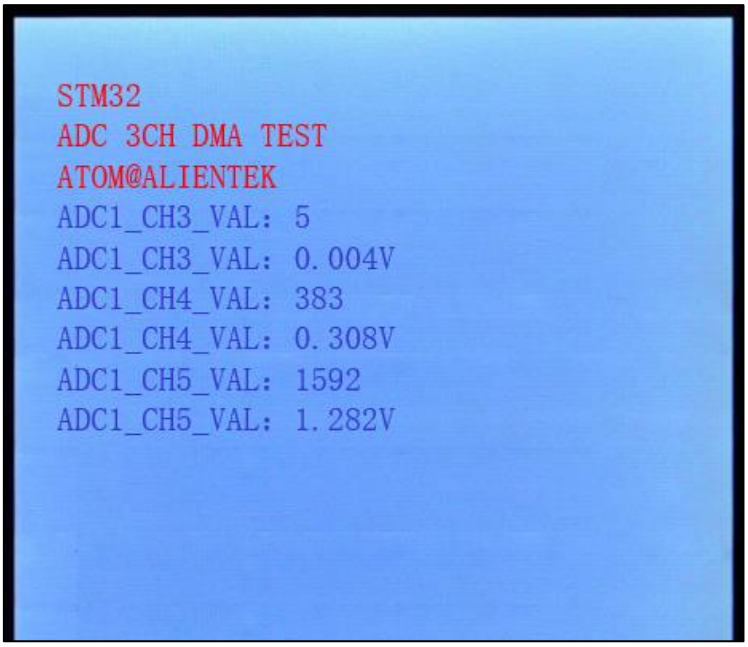


图 4.4.4.4.1 多通道 ADC 采集（DMA 读取）实验测试图

使用 ADC1 采集（DMA 读取）通道 3\4\5 的电压，在 LCD 模块上面显示对应的 ADC 转换值以及换算成电压后的电压值。可以使用杜邦线连接 PA3\PA4\PA5 到你想测量的电压源（0~3.3V），如果测量外部电源需要共地。

这 4 个通道可以同时测量不同测试点的电压，只需要用杜邦线分别接到不同的电压测试点即可。**注意：一定要保证测试点的电压在 0~3.3V 的电压范围，否则可能烧坏我们的 ADC，甚至是整个主控芯片。**

4.4.5 内部温度传感器实验

本实验我们来学习 STM32F407 的内部温度传感器并使用它来读取温度值，然后在 LCD 模块上显示出来。下面先带大家来了解一下 STM32 的内部温度传感器。

4.4.5.1 内部温度传感器简介

STM32 有一个内部的温度传感器，可以用来测量 CPU 及周围的温度（内部温度传感器更适合于检测温度的变化，需要测量精确温度的情况下，建议使用外置传感器）。对于 STM32F407 来说，该温度传感器在内部和 ADC1_IN16 输入通道相连接，此通道把传感器输出的电压转换成数字值，温度传感器模拟输入采样时间不能小于 10us。STM32 内部温度传感器支持的温度范围为：-40~125 度。精度为±1.5℃左右。

STM32 内部温度传感器的使用很简单，只要设置一下内部 ADC，并激活其内部温度传感器通道就差不多了。关于 ADC 的设置，我们在上一章已经进行了详细的介绍，这里就不再多说。接下来我们介绍一下和温度传感器设置相关的两个地方。

第一个地方，我们要使用 STM32 的内部温度传感器，必须先激活 ADC 的内部通道，通过 ADC_CR2 的 TSVREFE 位设置。设置该位为 1 则启用内部温度传感器。

第二个地方，STM32F4 的内部温度传感器固定的连接在 ADC1 的通道 16 上，所以，我们在设置好 ADC1 之后只要读取通道 16 的值，就是温度传感器返回来的电压值了。根据这个值，我们就可以计算出当前温度。计算公式如下：

$$T(℃) = \{ (Vsense - V25) / Avg_Slope \} + 25$$

式子中：

V25 = Vsense 在 25 度时的数值（典型值为：0.76）

Avg_Slope = 温度与 Vsense 曲线的平均斜率（单位：mv/℃或 uv/℃）（典型值：2.5mv/℃）。

利用以上公式，我们就可以方便的计算出当前温度传感器的温度了。

4.4.5.2 硬件设计

1. 例程功能

通过 ADC 的通道 16 读取 STM32F407 内部温度传感器的电压值，并将其转换为温度值，显示在 LCD 屏上。LED0 闪烁用于提示程序正在运行。

2. 硬件资源

1）LED 灯

LED0 – PE0

2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)

3）正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)

4）ADC1 通道 16

5）内部温度传感器

3. 原理图

ADC 和内部温度传感器都属于 STM32F407 内部资源，实际上我们只需要软件设置就可以正常工作，我们需要用到 TFTLCD 模块显示结果。

4.4.5.3 程序设计

4.4.5.3.1 ADC 的 HAL 库驱动

本实验用到的 ADC 的 HAL 库 API 函数前面都介绍过，具体调用情况请看程序解析部分。下面介绍读取内部温度传感器 ADC 值的配置步骤。

读取 STM32 内部温度传感器 ADC 值的配置步骤

1）开启 ADC 时钟

通过__HAL_RCC_ADC1_CLK_ENABLE 函数开启 ADC1 的时钟。

2）设置 ADC，开启内部温度传感器

调用 HAL_ADC_Init 函数来设置 ADC1 时钟分频系数、分辨率、模式、扫描方式等信息。

注意：该函数会调用：HAL_ADC_MspInit 回调函数来完成对 ADC 底层的初始化，包括：ADC1 时钟使能、ADC1 时钟源的选择等。

3）配置 ADC 通道并启动 AD 转换器

调用 HAL_ADC_ConfigChannel()函数配置 ADC1 通道 16，根据需求设置通道、规则序列、采样时间等。然后通过 HAL_ADC_Start 函数启动 AD 转换器。

4）读取 ADC 值，计算温度。

这里选择查询方式读取，在读取 ADC 值之前需要调用 HAL_ADC_PollForConversion 等待上一次转换结束。然后就可以通过 HAL_ADC_GetValue 来读取 ADC 值。最后根据上面介绍的公式计算出温度传感器的温度值。

4.4.5.3.2 程序流程图

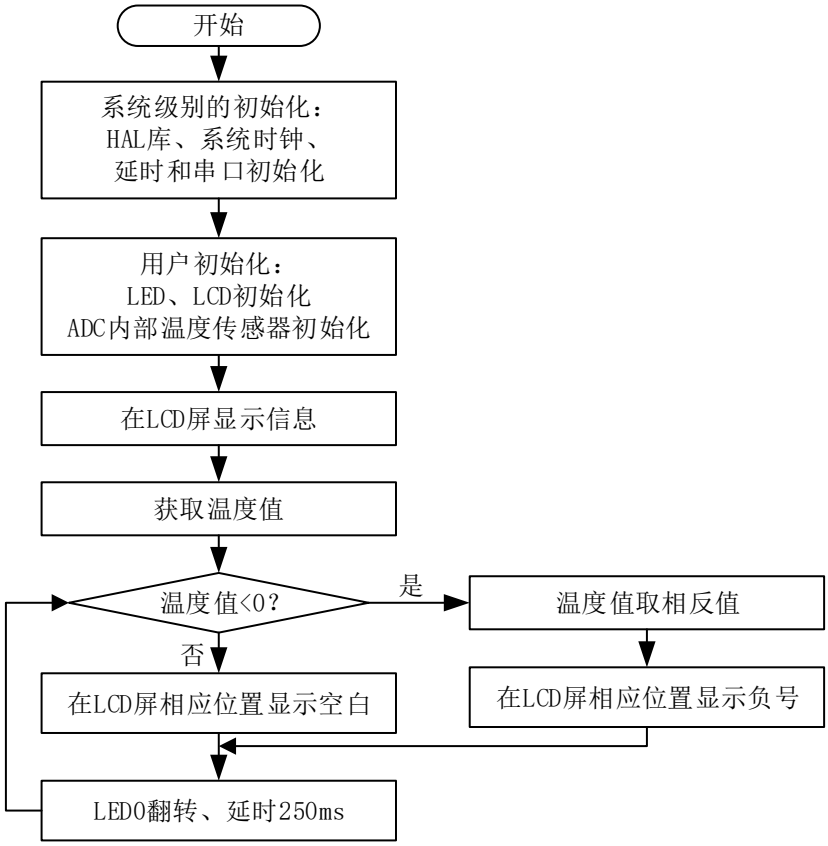


图 4.4.5.3.2.1 内部温度传感器实验程序流程图

4.4.5.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。内部温度传感器驱动源码包括两个文件：adc.c 和 adc.h。首先看 adc.h 头文件，其定义如下：

```
#define ADC_ADCX                    ADC1
#define ADC_TEMPSENSOR_CHX          ADC_CHANNEL_TEMPSENSOR /* 内部温度传感器专用通道 */

#define ADC_ADCX_CHY_CLK_ENABLE()
        do{ __HAL_RCC_ADC1_CLK_ENABLE(); }while(0)      /* ADC1 时钟使能 */
```

ADC_CHANNEL_TEMPSENSOR 就是内部温度传感器的通道宏定义，它实际上连接到了 ADC1 的通道 16，因此，我们也可以直接用 ADC_CHANNEL_16 来定义内部温度温度传感器的通道。

下面我们直接介绍与内部温度传感器相关的 adc.c 的代码，首先是 ADC 内部温度传感器初始化函数，其定义如下：

```
/**
 * @brief       ADC 内部温度传感器 初始化函数
 * @param       无
 * @retval      无
 */
void adc_temperature_init(void)
{
    ADC_ChannelConfTypeDef adc_ch_conf = {0};
    ADC_ADCX_CHY_CLK_ENABLE();

    g_adc_handle.Instance = ADC_ADCX;
    /* 4 分频，21Mhz */
    g_adc_handle.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
    g_adc_handle.Init.Resolution = ADC_RESOLUTION_12B; /* 12 位模式 */
    g_adc_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT; /* 右对齐 */
    g_adc_handle.Init.ScanConvMode = DISABLE;          /* 非扫描模式 */
    g_adc_handle.Init.ContinuousConvMode = DISABLE;    /* 关闭连续转换 */
    g_adc_handle.Init.NbrOfConversion = 1;             /* 1 个规则通道序列 */
    g_adc_handle.Init.DiscontinuousConvMode = DISABLE; /* 禁止不连续采样模式 */
    g_adc_handle.Init.NbrOfDiscConversion = 0;         /* 不连续采样通道数为 0 */
    g_adc_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;    /* 软件触发 */
    g_adc_handle.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    g_adc_handle.Init.DMAContinuousRequests = DISABLE;          /* 关闭 DMA 请求 */
    HAL_ADC_Init(&g_adc_handle);                                /* 初始化 ADC */

    adc_ch_conf.Channel = ADC_TEMPSENSOR_CHX;                   /* ADC 通道 */
    adc_ch_conf.Rank = 1;                                       /* 序列 */
    adc_ch_conf.SamplingTime = ADC_SAMPLETIME_480CYCLES;        /* 采样时间 */
    HAL_ADC_ConfigChannel(&g_adc_handle,&adc_ch_conf);          /* 通道配置 */
}
```

内部温度传感器相关的 ADC 初始化步骤与普通 ADC 类似，只是采集通道换成了内部温度传感器专用的通道，这里不再赘述。

下面讲解一下获取内部温度传感器温度值函数，其定义如下：

```
/**
 * @brief       获取内部温度传感器温度值
 * @param       无
 * @retval      温度值(扩大了 100 倍,单位:℃)
 */
short adc_get_temperature(void)
{
    uint32_t adcx;
    short result;
    double temperature;

    /* 读取内部温度传感器通道,20 次取平均 */
    adcx = adc_get_result_average(ADC_TEMPSENSOR_CHX, 20);
    temperature = (float)adcx * (3.3 / 4096);               /* 转化为电压值 */
    temperature = (temperature - 0.76) / 0.0025 + 25;       /* 计算温度 */
    result = temperature *= 100;                            /* 扩大 100 倍 */
    return result;
}
```

该函数先是调用前面 ADC 实验章节写好的 adc_get_result_average 函数取获取通道 ch 的转换值，然后通过温度转换公式，返回温度值。

在 main.c 里面编写如下代码：

```
int main(void)
{
    short temp;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    adc_temperature_init();                     /* 初始化 ADC 内部温度传感器采集 */

    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "Temperature TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 120, 200, 16, 16, "TEMPERATE: 00.00C", BLUE);

    while (1)
    {
        temp = adc_get_temperature();           /* 得到温度值 */
        if (temp < 0)                           /* 显示负号 */
        {
            temp = -temp;
            lcd_show_string(30 + 10 * 8, 120, 16, 16, 16, "-", BLUE);
        }
        else                                    /* 无符号 */
        {
            lcd_show_string(30 + 10 * 8, 120, 16, 16, 16, " ", BLUE);
        }
        /* 显示整数部分 */
        lcd_show_xnum(30 + 11 * 8, 120, (temp / 100), 2, 16, 0, BLUE);
        /* 显示小数部分 */
        lcd_show_xnum(30 + 14 * 8, 120, (temp % 100), 2, 16, 0X80, BLUE);

        LED0_TOGGLE();                          /* LED0 闪烁,提示程序运行 */
        delay_ms(250);
    }
}
```

该部分的代码逻辑很简单，先是得到温度值，再根据温度值判断正负值，来显示温度符号，再显示整数和小数部分。

4.4.5.4 下载验证

将程序下载到开发板后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示的内容如图 4.4.5.4.1 所示：

图 4.4.5.4.1 内部温度传感器实验测试图

大家可以看看显示温度值与实际是否相符合（因为芯片会发热，所以一般会比实际温度偏高）。

4.5 DAC

本小节我们将介绍 STM32F407 的 DAC（Digital-to-analog converters，数模转换器）功能。我们通过三个实验来学习 DAC，分别是 DAC 输出实验、DAC 输出三角波实验和 DAC 输出正弦波实验。

本小节分为如下几个部分：

4.5.1 DAC 简介
4.5.2 DAC 输出实验
4.5.3 DAC 输出三角波实验
4.5.4 DAC 输出正弦波实验

4.5.1 DAC 简介

STM32F407 的 DAC 模块（数字/模拟转换模块）是 12 位数字输入，电压输出型的 DAC。DAC 可以配置为 8 位或 12 位模式，也可以与 DMA 控制器配合使用。DAC 工作在 12 位模式时，数据可以设置成左对齐或右对齐。DAC 模块有 2 个输出通道，每个通道都有单独的转换器。在双 DAC 模式下，2 个通道可以独立地进行转换，也可以同时进行转换并同步地更新 2 个通道的输出。DAC 可以通过引脚输入参考电压 Vref+以获得更精确的转换结果。

STM32 的 DAC 模块主要特点有：

① 两个 DAC 转换器：各对应一个输出通道；
② 12 位模式下数据采用左对齐或右对齐；
③ 同步更新功能；
④ DMA 下溢错误检测；
⑤ 噪声\三角波形生成；
⑥ DAC 双通道单独或同时转换；
⑦ 每个通道都有 DMA 功能；

DAC 通道框图如图 4.5.1.1 所示：

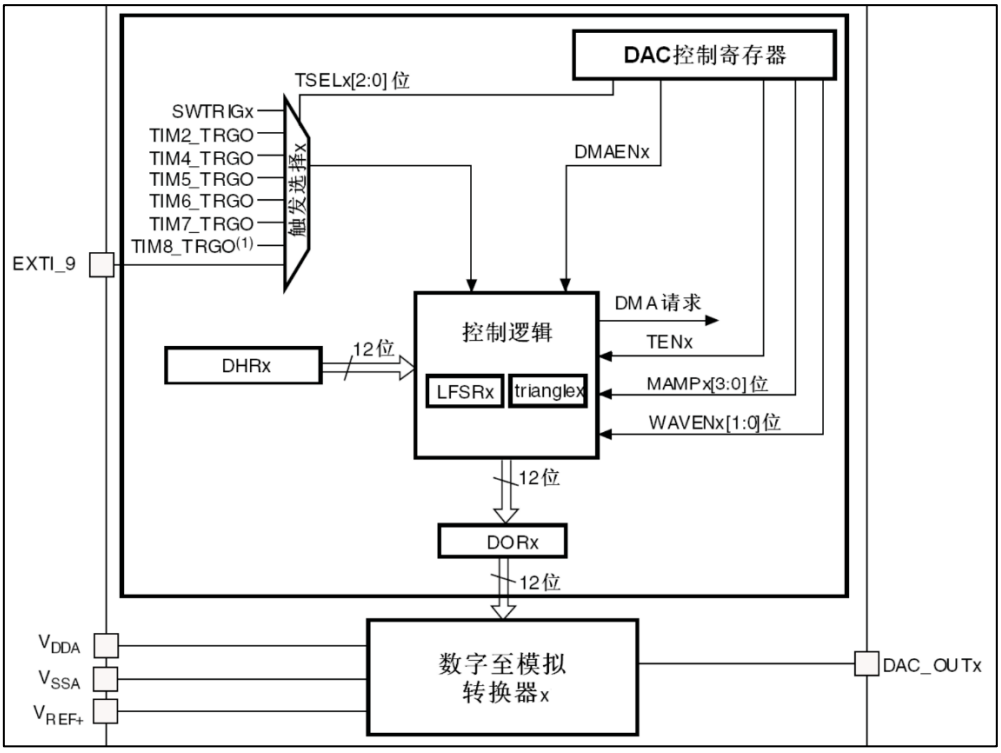


图 4.5.1.1 DAC 通道框图

图中 4.5.1.1 中，V_{DDA} 和 V_{SSA} 为 DAC 模块模拟部分的供电，而 Vref+则是 DAC 模块的参考电压输入引脚。DAC_OUTx 就是 DAC 的两个输出通道了（对应 PA4 或者 PA5 引脚）。DAC 的这些输入/输出引脚信息如下表所示：

引脚名称	信号类型	说明
V_{REF+}	正模拟参考电压输入	DAC 高/正参考电压，$V_{REF+} \leqslant V_{DDAmax}$
VDDA	模拟电源输入	模拟电源
VSSA	模拟电源地输入	模拟电源地
DAC_OUTx	模拟输出信号	DAC 通道 x 模拟输出，x=1/2

表 4.5.1.1 DAC 输入/输出引脚

从图 4.5.1.1 可以看出，DAC 输出是受 DORx（x=1/2，下同）寄存器直接控制的，但是我们不能直接往 DORx 寄存器写入数据，而是通过 DHRx 间接的传给 DORx 寄存器，实现对 DAC 输出的控制。

前面我们提到，STM32F407 的 DAC 支持 8/12 位模式，8 位模式的时候是固定的右对齐的，而 12 位模式又可以设置左对齐/右对齐。DAC 单通道模式下的数据寄存器对齐方式，总共有 3 种情况，如下图所示：

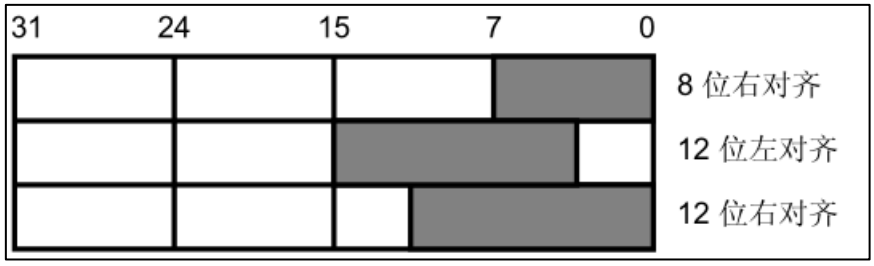


图 4.5.1.2 DAC 单通道模式下的数据寄存器对齐方式

① 8 位数据右对齐：用户将数据写入 DAC_DHR8Rx[7:0]位（实际存入 DHRx[11:4]位）。

② 12 位数据左对齐：用户将数据写入 DAC_DHR12Lx[15:4]位（实际存入 DHRx[11:0]位）。

③ 12 位数据右对齐：用户将数据写入 DAC_DHR12Rx[11:0]位（实际存入 DHRx[11:0]位）。

我们本章实验中使用的都是单通道模式下的 DAC 通道 1，采用 12 位右对齐格式，所以采用第③种情况。另外 DAC 还具有双通道转换功能。

对于 DAC 双通道（可用时），也有三种可能的方式，如下图所示：

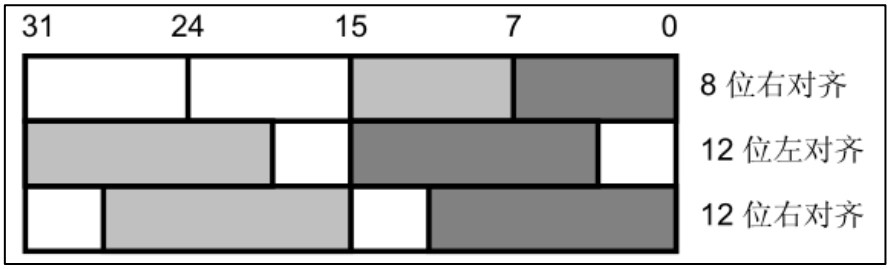


图 4.5.1.3 DAC 双通道模式下的数据寄存器对齐方式

① 8 位数据右对齐：用户将 DAC 通道 1 的数据写入 DAC_DHR8RD[7:0]位（实际存入 DHR1 [11:4]位），将 DAC 通道 2 的数据写入 DAC_DHR8RD[15:8]位（实际存入 DHR2 [11:4]位）。

② 12 位数据左对齐：用户将 DAC 通道 1 的数据写入 DAC_DHR12LD [15:4]位（实际存入 DHR1[11:0]位），将 DAC 通道 2 的数据写入 DAC_DHR12LD [31:20]位（实际存入 DHR2[11:0]位）。

③ 12 位数据右对齐：用户将 DAC 通道 1 的数据写入 DAC_DHR12RD [11:0]位（实际存入 DHR1[11:0]位），将 DAC 通道 2 的数据写入 DAC_DHR12RD [27:16]位（实际存入 DHR2[11:0]位）。

DAC 可以通过软件或者硬件触发转换，通过配置 TENx 控制位来决定。如果没有选中硬件触发（寄存器 DAC_CR1 的 TENx 位置 0），存入寄存器 DAC_DHRx 的数据会在 1 个 APB1

时钟周期后自动传至寄存器 DAC_DORx。如果选中硬件触发（寄存器 DAC_CR1 的 TENx 位置 1），数据传输在触发发生以后 3 个 APB1 时钟周期后完成。一旦数据从 DAC_DHRx 寄存器装入 DAC_DORx 寄存器，在经过时间 $t_{SETTLING}$ 之后，输出即有效，这段时间的长短依电源电压和模拟输出负载的不同会有所变化。我们可以从《STM32F407IGT6.pdf》数据手册查到 $t_{SETTLING}$ 的典型值为 3us，最大是 6us，所以 DAC 的转换速度最快是 333K 左右。

不使用硬件触发（TEN=0），其转换的时间框图如图 4.5.1.4 所示：

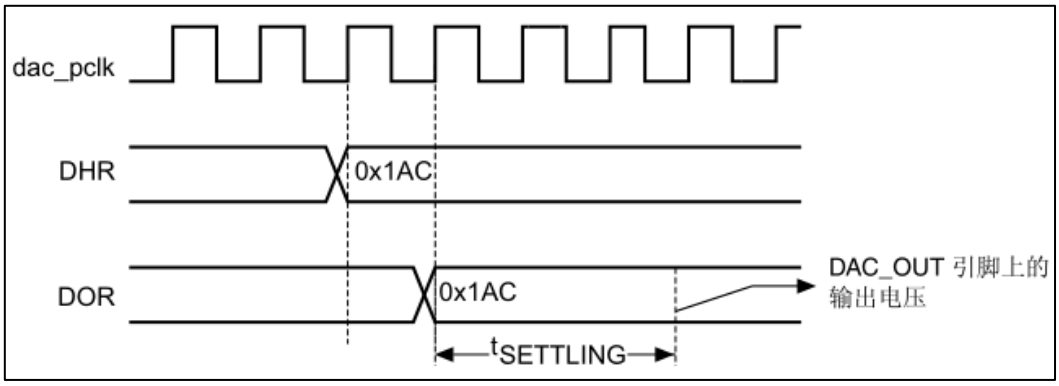


图 4.5.1.4 TEN=0 时 DAC 模块转换时间框图

当 DAC 的参考电压为 VREF+的时候，DAC 的输出电压是线性的从 0~VREF+，12 位模式下 DAC 输出电压与 VREF +以及 DORx 的计算公式如下：

DACx 输出电压= VREF *（DORx/4095）

如果使用硬件触发（TEN=1），可通过外部事件（定时计数器、外部中断线）触发 DAC 转换。由 TSELx[2:0]控制位来决定选择 8 个触发事件中的一个来触发转换。这 8 个触发事件如下表所示：

源	类型	TSEL[2:0]
Timer 6 TRGO event	片上定时器的内部信号	000
Timer 8 TRGO event		001
Timer 7 TRGO event		010
Timer 5 TRGO event		011
Timer 2 TRGO event		100
Timer 4 TRGO event		101
EXTI line9	外部引脚	110
SWTRIG	软件控制位	111

表 4.5.1.3 DAC 触发选择

原表见《STM32F4xx 参考手册_V4（中文版）.pdf》第 292 页表 58。

每个 DAC 通道都有 DMA 功能，两个 DMA 通道分别用于处理两个 DAC 通道的 DMA 请求。如果 DMAENx 位置 1 时，如果发生外部触发（而不是软件触发），就会产生一个 DMA 请求，然后 DAC_DHRx 寄存器的数据被转移到 DAC_NORx 寄存器。

4.5.2 DAC 输出实验

本实验我们来学习 DAC 输出。

4.5.2.1 DAC 寄存器

下面，我们介绍要实现 DAC 的通道 1 输出，需要用到的一些 DAC 寄存器。

- **DACx 控制寄存器（DACx_CR）**

DACx 控制寄存器描述如图 4.5.2.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		DMAUDRIE2	DMAEN2	MAMP2[3:0]				WAVE2[1:0]		TSEL2[2:0]			TEN2	BOFF2	EN2
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		DMAUDRIE1	DMAEN1	MAMP1[3:0]				WAVE1[1:0]		TSEL1[2:0]			TEN1	BOFF1	EN1
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 7:6 **WAVE1[1:0]**：DAC 1 通道噪声/三角波生成使能 (DAC channel1 noise/triangle wave generation enable)

这些位将由软件置 1 和清零。

00：禁止生成波

01：使能生成噪声波

1x：使能生成三角波

注意：只在位 TEN1 = 1（使能 DAC 1 通道触发）时使用。

位 5:3 **TSEL1[2:0]**：DAC 1 通道触发器选择 (DAC channel1 trigger selection)

这些位用于选择 DAC 1 通道的外部触发事件。

000：定时器 6 TRGO 事件

001：定时器 8 TRGO 事件

010：定时器 7 TRGO 事件

011：定时器 5 TRGO 事件

100：定时器 2 TRGO 事件

101：定时器 4 TRGO 事件

110：外部中断线 9

111：软件触发

注意：只在位 TEN1 = 1（使能 DAC 1 通道触发）时使用。

位 2 **TEN1**：DAC 1 通道触发使能 (DAC channel1 trigger enable)

此位由软件置 1 和清零，以使能/禁止 DAC 1 通道触发。

0：禁止 DAC 1 通道触发，写入 DAC_DHRx 寄存器的数据在一个 APB1 时钟周期之后转移到 DAC_DOR1 寄存器

1：使能 DAC 1 通道触发，DAC_DHRx 寄存器的数据在三个 APB1 时钟周期之后转移到 DAC_DOR1 寄存器

注意：如果选择软件触发，DAC_DHRx 寄存器的内容只需一个 APB1 时钟周期即可转移到 DAC_DOR1 寄存器。

位 1 **BOFF1**：DAC 1 通道输出缓冲器禁止 (DAC channel1 output buffer disable)

此位由软件置 1 和清零，以使能/禁止 DAC 1 通道输出缓冲器。

0：使能 DAC 1 通道输出缓冲器

1：禁止 DAC 1 通道输出缓冲器

位 0 **EN1**：DAC 1 通道使能 (DAC channel1 enable)

此位由软件置 1 和清零，以使能/禁止 DAC 1 通道。

0：禁止 DAC 1 通道

1：使能 DAC 1 通道

图 4.5.2.1.1 DACx_CR 寄存器

DAC_CR 的低 16 位用于控制通道 1，而高 16 位用于控制通道 2，我们这里仅列出本实验需要设置的一些位：

EN1 位：用于 DAC 通道 1 的使能，我们要用到 DAC 通道 1 的输出，该位必须设置为 1。

BOFF1 位：用于 DAC 输出缓存控制，本教程的三个 DAC 实验我们都不使用输出缓存，即该位设置为 1。

TEN1 位：用于 DAC 通道 1 的触发使能，我们设置该位为 0，不使用触发。写入 DHR1 的值会在 1 个 APB1 周期后传送到 DOR1，然后输出到 PA4 口上。

TSEL1[2:0]位，用于选择 DAC 通道 1 的触发方式，这里我们没有用到外部触发，所以这几位设置为 0 即可。

WAVE1[1:0]位，用于控制 DAC 通道 1 的噪声/波形输出功能，我们这里没用到波形发生器，所以默认设置为 00，不使能噪声/波形输出。

MAMP[3:0]位，是屏蔽/幅值选择器，用来在噪声生成模式下选择屏蔽位，在三角波生成模式下选择波形的幅值。本实验没有用到波形发生器，所以设置为 0 即可。

DMAEN1 位，用于 DAC 通道 1 的 DMA 使能，本实验没有用到 DMA 功能，所以设置为 0。

- **DACx 通道 1 12 位右对齐数据保持寄存器（DACx_DHR12R1）**

DACx 通道 1 12 位右对齐数据保持寄存器描述如图 4.5.2.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				DACC2DHR[11:0]											
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:12 保留，必须保持复位值。

位 11:0 **DACC2DHR[11:0]**：DAC 2 通道 12 位右对齐数据 (DAC channel2 12-bit right-aligned data)
这些位由软件写入，用于为 DAC 2 通道指定 12 位数据。

图 4.5.2.1.2 DAC_DHR12R1 寄存器

该寄存器用来设置 DAC 输出，通过写入 12 位数据到该寄存器，就可以在 DAC 输出通道 1（PA4）得到我们所要的结果。

4.5.2.2 硬件设计

1. 例程功能

使用 KEY0/KEY1 两个按键，控制 STM32 内部 DAC 的通道 1 输出电压大小，然后通过 ADC1 的通道 3 采集 DAC 输出的电压，在 LCD 模块上面显示 ADC 采集到的电压值以及 DAC 的设定输出电压值等信息。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）独立按键 ：
KEY0 - PE2
KEY1 - PE3
5）ADC1 ：
通道 3 - PA3
6）DAC1 ：
通道 1 - PA4

3. 原理图

我们只需要通过杜邦线连接 PA3 和 PA4，就可以使得 ADC1 通道 3 和 DAC1 通道 1 连接起来。

4.5.2.3 程序设计

4.5.2.3.1 DAC 的 HAL 库驱动

DAC 在 HAL 库中的驱动代码在 stm32f4xx_hal_dac.c 和 stm32f4xx_hal_dac_ex.c 文件（及其头文件）中。

1. HAL_DAC_Init 函数

DAC 的初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_Init(DAC_HandleTypeDef *hdac);
```

- **函数描述：**

用于初始化 DAC。

- **函数形参：**

形参 1 是 DAC_HandleTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
  DAC_TypeDef                 *Instance;        /* DAC 寄存器基地址 */
  __IO HAL_DAC_StateTypeDef   State;            /* DAC 工作状态 */
  HAL_LockTypeDef             Lock;             /* DAC 锁定对象 */
  DMA_HandleTypeDef           *DMA_Handle1;     /* 通道 1 的 DMA 处理句柄指针 */
  DMA_HandleTypeDef           *DMA_Handle2;     /* 通道 2 的 DMA 处理句柄指针 */
  __IO uint32_t               ErrorCode;        /* DAC 错误代码 */
} DAC_HandleTypeDef;
```

从该结构体看到该函数并没有设置任何 DAC 相关寄存器，即没有对 DAC 进行任何配置，它只是 HAL 库提供用来在软件上初始化 DAC，为后面 HAL 库操作 DAC 做好准备。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

- **注意事项：**

DAC 的 MSP 初始化函数 HAL_DAC_MspInit，该函数声明如下：

```
void HAL_DAC_MspInit(DAC_HandleTypeDef* hdac);
```

2. HAL_DAC_ConfigChannel 函数

DAC 的通道参数初始化函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_ConfigChannel(DAC_HandleTypeDef *hdac,
                                        DAC_ChannelConfTypeDef *sConfig, uint32_t Channel);
```

- **函数描述：**

该函数用来配置 DAC 通道的触发类型以及输出缓冲。

- **函数形参：**

形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。

形参 2 是 DAC_ChannelConfTypeDef 结构体类型指针变量，其定义如下：

```
typedef struct
{
  uint32_t DAC_Trigger;          /* DAC 触发源的选择 */
  uint32_t DAC_OutputBuffer;     /* 输出缓冲*/
} DAC_ChannelConfTypeDef;
```

形参 3 用于选择要配置的通道，可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

3. HAL_DAC_Start 函数

使能启动 DAC 转换通道函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_Start(DAC_HandleTypeDef *hdac, uint32_t Channel);
```

- **函数描述：**

使能启动 DAC 转换通道。

- **函数形参：**

形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。

形参 2 用于选择要启动的通道，可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。

- **函数返回值：**

HAL_StatusTypeDef 枚举类型的值。

4. HAL_DAC_SetValue 函数

DAC 的通道输出值函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_SetValue(DAC_HandleTypeDef *hdac, uint32_t Channel,
                                   uint32_t Alignment, uint32_t Data);
```

- **函数描述：**

配置 DAC 的通道输出值。

- **函数形参：**

形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。

形参 2 用于选择要输出的通道，可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。
形参 3 用于指定数据对齐方式。
形参 4 设置要加载到选定数据保存寄存器中的数据。

- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

5. HAL_DAC_GetValue 函数

DAC 读取通道输出值函数，其声明如下：

```
uint32_t HAL_DAC_GetValue(DAC_HandleTypeDef *hdac, uint32_t Channel);
```

- **函数描述：**
  获取所选 DAC 通道的最后一个数据输出值。
- **函数形参：**
  形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。
  形参 2 用于选择要读取的通道，可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。
- **函数返回值：**
  获取到的输出值。

DAC 输出配置步骤

1）开启 DAC 和输出通道的 GPIO 时钟，配置该 IO 口的模拟输出功能

首先开启 DAC 和 GPIO 的时钟，然后配置 GPIO 为模拟模式。本实验我们默认用到 DAC1 通道 1，对应 IO 是 PA4，它们的时钟开启方法如下：

```
__HAL_RCC_DAC_CLK_ENABLE ();            /* 使能 DAC1 时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE();           /* 开启 GPIOA 时钟 */
```

IO 口模拟输出功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 DAC

通过 HAL_DAC_Init 函数来设置需要初始化的 DAC。该函数并没有设置任何 DAC 相关寄存器，也就是说没有对 DAC 进行任何配置，它只是 HAL 库提供用来在软件上初始化 DAC。

注意：该函数会调用 HAL_DAC_MspInit 函数来存放 DAC 和对应通道的 IO 时钟使能和初始化 IO 等代码。

3）配置 DAC 通道并启动 DA 转换器

在 HAL 库中，通过 HAL_DAC_ConfigChannel 函数来设置配置 DAC 的通道，根据需求设置触发类型以及输出缓冲。

配置好 DAC 通道之后，通过 HAL_DAC_Start 函数启动 DA 转换器。

4）设置 DAC 的输出值

通过 HAL_DAC_SetValue 函数设置 DAC 的输出值。

4.5.2.3.2 程序流程图

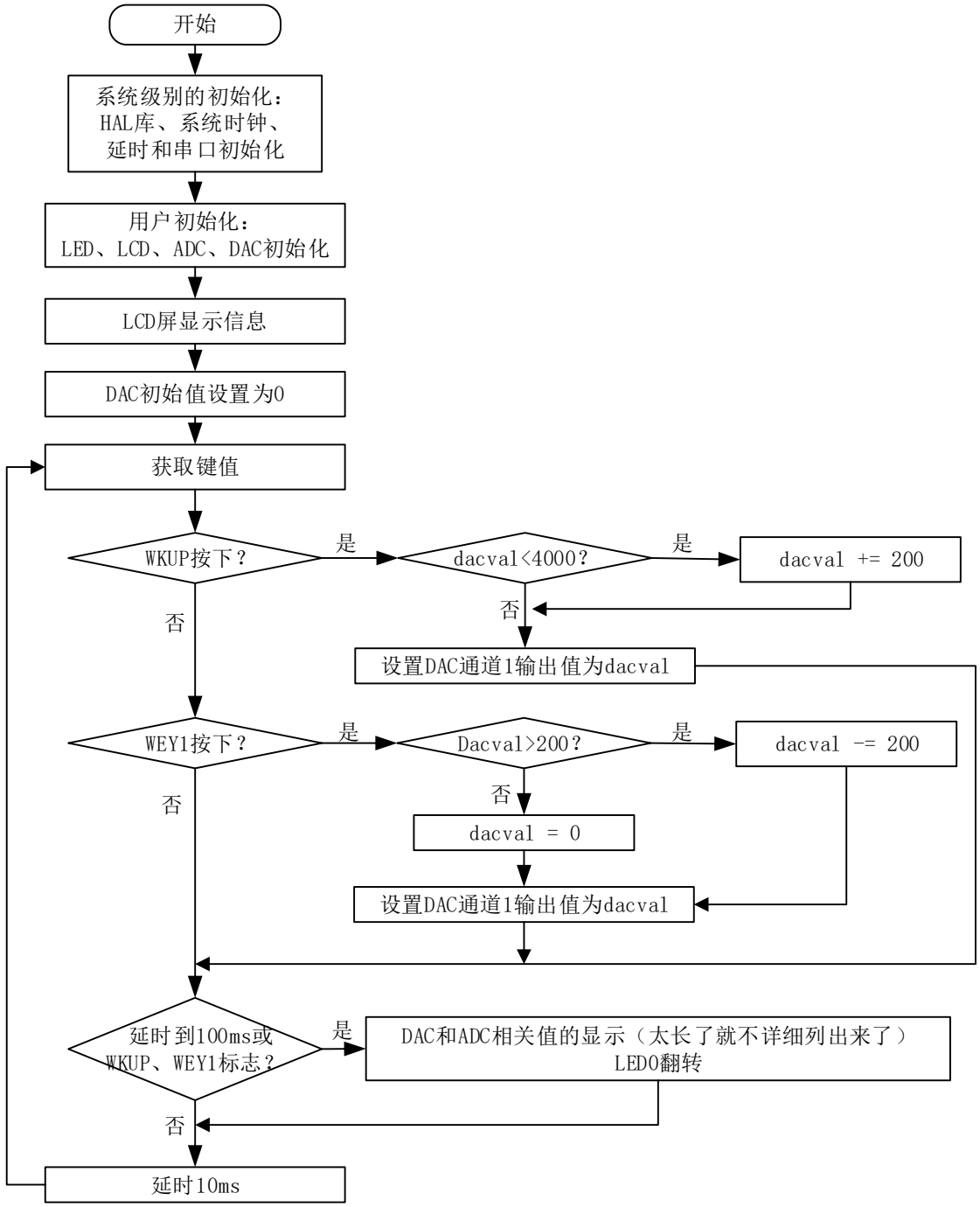


图 4.5.2.3.2.1 DAC 输出实验程序流程图

4.5.2.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DAC 驱动源码包括两个文件：dac.c 和 dac.h。另外我们使用 ADC1 来测量 DAC 输出的电压是否准确，所以还需要用到单通道 ADC 采集实验里面的 adc.c 和 adc.h 文件，关于 ADC 的程序这里就不再讲解。

dac.h 文件只有一些函数声明，下面直接开始介绍 dac.c 的程序，首先是 DAC 初始化函数。

```
/**
 * @brief       DAC 初始化函数
 * @param       outx: 要初始化的通道. 1,通道 1; 2,通道 2
```

```
 * @retval      无
 */
void dac_init(uint8_t outx)
{
    GPIO_InitTypeDef gpio_init_struct;
    DAC_ChannelConfTypeDef dac_ch_conf;

    __HAL_RCC_DAC_CLK_ENABLE();                                 /* 使能 DAC1 的时钟 */
    /* 使能 DAC OUT1/2 的 IO 口时钟(都在 PA 口,PA4/PA5) */
    __HAL_RCC_GPIOA_CLK_ENABLE();

    /* STM32 单片机, 总是 PA4=DAC1_OUT1, PA5=DAC1_OUT2 */
    gpio_init_struct.Pin = (outx==1)? GPIO_PIN_4 : GPIO_PIN_5;
    gpio_init_struct.Mode = GPIO_MODE_ANALOG;
    HAL_GPIO_Init(GPIOA, &gpio_init_struct);

    g_dac_handle.Instance = DAC;
    HAL_DAC_Init(&g_dac_handle);                                /* 初始化 DAC */

    dac_ch_conf.DAC_Trigger = DAC_TRIGGER_NONE;                 /* 不使用触发功能 */
    dac_ch_conf.DAC_OutputBuffer = DAC_OUTPUTBUFFER_DISABLE;    /* 关闭输出缓冲 */

    switch(outx)
    {
        case 1:             /* DAC 通道 1 配置 */
            HAL_DAC_ConfigChannel(&g_dac_handle,&dac_ch_conf,DAC_CHANNEL_1);
            HAL_DAC_Start(&g_dac_handle,DAC_CHANNEL_1);         /* 开启 DAC 通道 1 */
            break;

        case 2:             /* DAC 通道 2 配置 */
            HAL_DAC_ConfigChannel(&g_dac_handle,&dac_ch_conf,DAC_CHANNEL_2);
            HAL_DAC_Start(&g_dac_handle,DAC_CHANNEL_2);         /* 开启 DAC 通道 2 */
            break;
        default:break;
    }
}
```

该函数主要调用 HAL_DAC_Init 和 HAL_DAC_ConfigChannel 函数初始化 DAC，并调用 HAL_DAC_Start 函数使能 DAC 通道。HAL_DAC_Init 函数会调用 HAL_DAC_MspInit 回调函数，该函数用于存放 DAC 和对应通道的 IO 时钟使能和初始化 IO 等代码。本实验为了让 dac_init 函数支持 DAC 的 OUT1/2 两个通道的初始化，就没有用到该函数。

下面是设置 DAC 通道 1/2 输出电压函数，其定义如下：

```
/**
 * @brief       设置通道 1/2 输出电压
 * @param       outx: 1,通道 1; 2,通道 2
 * @param       vol : 0~3300,代表 0~3.3V
 * @retval      无
 */
void dac_set_voltage(uint8_t outx, uint16_t vol)
{
    double temp = vol;
    temp /= 1000;
    temp = temp * 4096 / 3.3;

    if (temp >= 4096)temp = 4095;           /* 如果值大于等于 4096, 则取 4095 */

    if (outx == 1)                          /* 通道 1 */
    {
        /* 12 位右对齐数据格式设置 DAC 值 */
        HAL_DAC_SetValue(&g_dac_handle,DAC_CHANNEL_1,DAC_ALIGN_12B_R,temp);
    }
    else                                    /* 通道 2 */
    {
```

```
        /* 12 位右对齐数据格式设置 DAC 值 */
        HAL_DAC_SetValue(&g_dac_handle,DAC_CHANNEL_2,DAC_ALIGN_12B_R,temp);
    }
}
```

该函数实际就是将电压值转换为 DAC 输入值，形参 1 用于设置通道，形参 2 设置要输出的电压值，设置的范围：0~3300，代表 0~3.3V。

最后在 main 函数里面编写如下代码：

```
int main(void)
{
    uint16_t adcx;
    float temp;
    uint8_t t = 0;
    uint16_t dacval = 0;
    uint8_t key;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    adc_init();                                 /* 初始化 ADC */
    dac_init(1);                                /* 初始化 DAC1_OUT1 通道 */

    lcd_show_string(30,  50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30,  70, 200, 16, 16, "DAC TEST", RED);
    lcd_show_string(30,  90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY1:+  KEY0:-", RED);

    lcd_show_string(30, 130, 200, 16, 16, "DAC VAL:", BLUE);
    lcd_show_string(30, 150, 200, 16, 16, "DAC VOL:0.000V", BLUE);
    lcd_show_string(30, 170, 200, 16, 16, "ADC VOL:0.000V", BLUE);

    while (1)
    {
        t++;
        key = key_scan(0);                      /* 按键扫描 */
        if (key == KEY1_PRES)                   /* 按键 1 按下 */
        {
            if (dacval < 4000)
            {
                dacval += 200;                  /* 输出增大 200 */
            }
            HAL_DAC_SetValue(&g_dac_handle, DAC_CHANNEL_1, DAC_ALIGN_12B_R,
                                                                    dacval);
        }
        else if (key == KEY0_PRES)              /* 按键 0 按下 */
        {
            if (dacval > 200)
            {
                dacval -= 200;                  /* 输出减少 200 */
            }
            else
            {
                dacval = 0;
            }
            HAL_DAC_SetValue(&g_dac_handle, DAC_CHANNEL_1, DAC_ALIGN_12B_R,
                                                                    dacval);
        }
        /* KEY0/KEY1 按下了,或者定时时间到了 */
        if (t == 10 || key == KEY0_PRES || key == KEY1_PRES)
        {
```

```
            /* 读取前面设置 DAC1_OUT1 的值 */
            adcx = HAL_DAC_GetValue(&g_dac_handle, DAC_CHANNEL_1);
            lcd_show_xnum(94, 130, adcx, 4, 16, 0, BLUE);  /* 显示 DAC 寄存器值 */

            temp = (float)adcx * (3.3 / 4096);              /* 得到 DAC 电压值 */
            adcx = temp;
            lcd_show_xnum(94, 150, temp, 1, 16, 0, BLUE);  /* 显示电压值整数部分 */

            temp -= adcx;
            temp *= 1000;
            /* 显示电压值的小数部分 */
            lcd_show_xnum(110, 150, temp, 3, 16, 0X80, BLUE);

            /* 得到 ADC1 通道 3 的转换结果 */
            adcx = adc_get_result_average(ADC_ADCX_CHY, 20);
            temp = (float)adcx * (3.3 / 4096);             /* 得到 ADC 电压值 */
            adcx = temp;
            lcd_show_xnum(94, 170, temp, 1, 16, 0, BLUE); /* 显示电压值整数部分 */

            temp -= adcx;
            temp *= 1000;
            /* 显示电压值的小数部分 */
            lcd_show_xnum(110, 170, temp, 3, 16, 0X80, BLUE);

            LED0_TOGGLE();                                 /* LED0 闪烁 */
            t = 0;
        }
        delay_ms(10);
    }
}
```

此部分代码，我们通过 KEY0 和 KEY1 来实现对 DAC 输出的幅值控制。按下 KEY1 增加，按 KEY0 减小。同时在 LCD 上面显示 DHR12R1 寄存器的值、DAC 设置输出电压以及 ADC 采集到的 DAC 输出电压。

4.5.2.4 下载验证

下载代码后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示如图 4.5.2.4.1 所示：

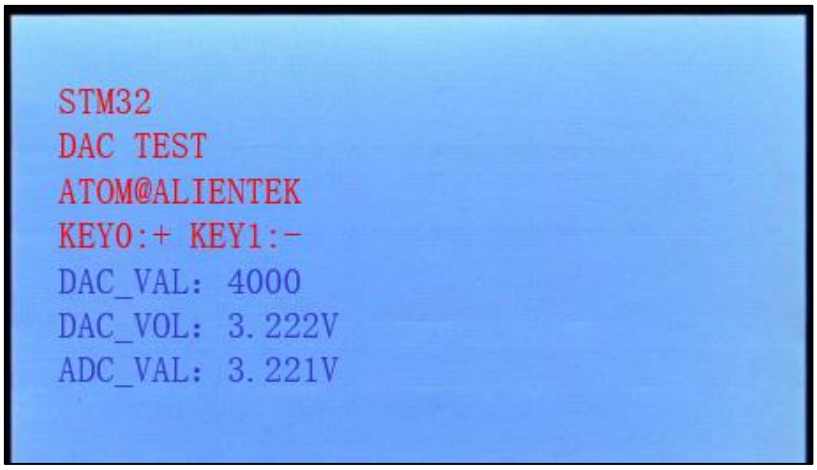


图 4.5.2.4.1 DAC 输出实验测试图

验证试验前记得先通过杜邦线连接 PA3 和 PA4，然后我们可以通过按 KEY1 按键，增加 DAC 输出的电压，这时 ADC 采集到的电压也会增大，通过按 KEY0 减小 DAC 输出的电压，这时 ADC 采集到的电压也会减小。

4.5.3 DAC 输出三角波实验

本实验我们来学习使用如何让 DAC 输出三角波，DAC 初始化部分还是用 DAC 输出实验的，所以做本实验的前提是先学习 DAC 输出实验。

4.5.3.1 DAC 寄存器

本实验用到的寄存器在 DAC 输出实验都有介绍。

4.5.3.2 硬件设计

1. 例程功能

使用 DAC 输出三角波，通过 KEY0/KEY1 两个按键，控制 DAC1 的通道 1 输出两种三角波，需要通过示波器接 PA4 进行观察。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）独立按键 ：
KEY0 - PE2
KEY1 - PE3
5）DAC1 ：
通道 1 - PA4

3. 原理图

我们只需要把示波器的探头接到 DAC1 通道 1（PA4）引脚，就可以在示波器上显示 DAC 输出的波形。

4.5.3.3 程序设计

本实验用到的 DAC 的 HAL 库 API 函数前面都介绍过，具体调用情况请看程序解析部分。下面介绍 DAC 输出三角波的配置步骤。

DAC 输出三角波配置步骤

1）开启 DACx 和输出通道的 GPIO 时钟，配置该 IO 口的模拟输出功能

首先开启 DACx 和 GPIO 的时钟，然后配置 GPIO 为模拟模式。本实验我们默认用到 DAC1 通道 1，对应 IO 是 PA4，它们的时钟开启方法如下：

```
__HAL_RCC_DAC_CLK_ENABLE ();            /* 使能 DAC1 时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE();           /* 开启 GPIOA 时钟 */
```

IO 口模拟输出功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 DACx

通过 HAL_DAC_Init 函数来设置需要初始化的 DAC。该函数并没有设置任何 DAC 相关寄存器，也就是说没有对 DAC 进行任何配置，它只是 HAL 库提供用来在软件上初始化 DAC。

注意：该函数会调用 HAL_DAC_MspInit 函数来存放 DAC 和对应通道的 IO 时钟使能和初始化 IO 等代码。

3）配置 DAC 通道并启动 DA 转换器

在 HAL 库中，通过 HAL_DAC_ConfigChannel 函数来设置配置 DAC 的通道，根据需求设置触发类型以及输出缓冲。

配置好 DAC 通道之后，通过 HAL_DAC_Start 函数启动 DA 转换器。

4）设置 DAC 的输出值

通过 HAL_DAC_SetValue 函数设置 DAC 的输出值。这里我们根据三角波的特性，创建了 dac_triangular_wave 函数用于控制输出三角波。

4.5.3.3.1 程序流程图

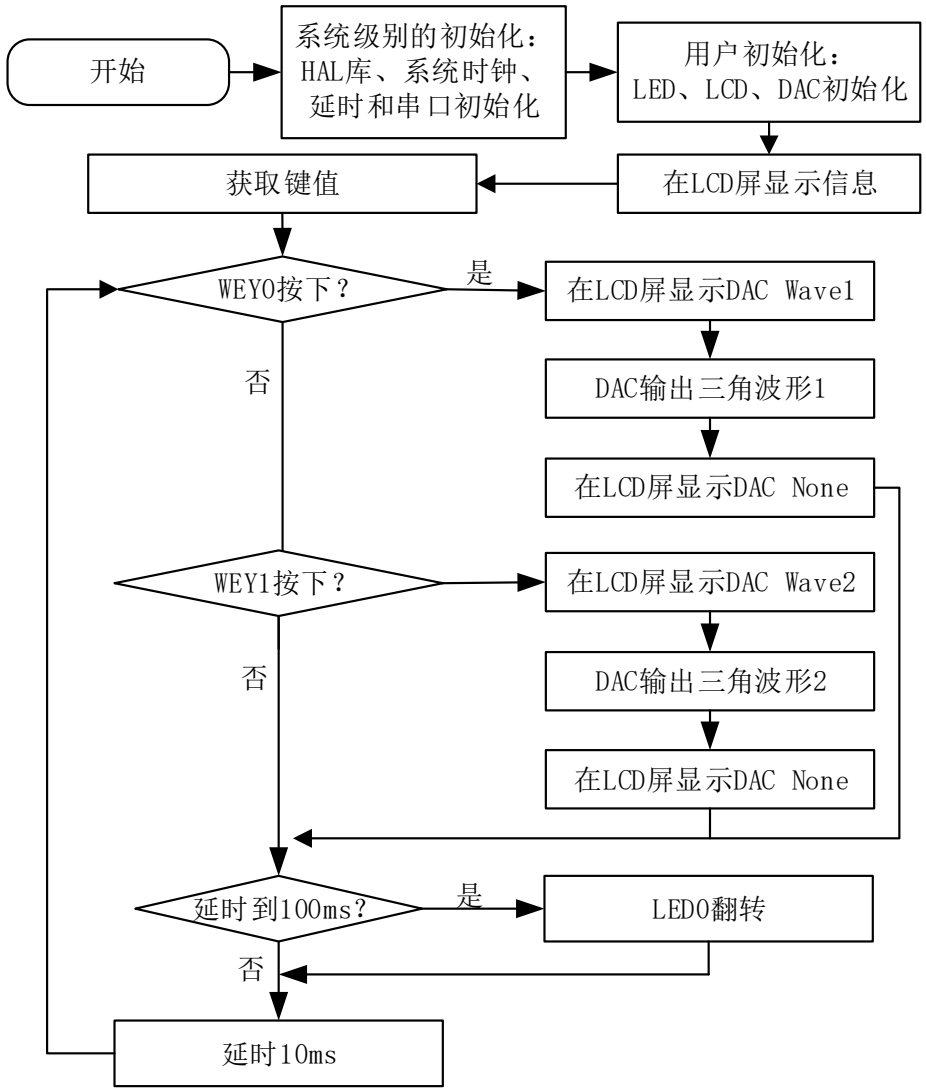


图 4.5.3.3.1.1 DAC 输出三角波实验程序流程图

4.5.3.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DAC 驱动源码包括两个文件：dac.c 和 dac.h。本实验代码在 DAC 输出实验代码后进行追加。

dac.h 文件只有一些声明，下面直接开始介绍 dac.c 的程序，本实验的 DAC 初始化我们还是用到 dac_init 函数，就添加了一个设置 DAC_OUT1 输出三角波函数，其定义如下：

```
/**
 * @brief       设置 DAC_OUT1 输出三角波
 *   @note      输出频率 ≈ 1000 / (dt * samples) Khz，不过在 dt 较小的时候，比如小于 5us
 *              时，由于 delay_us 本身就不准了(调用函数，计算等都需要时间，延时很小的时候，这些
 *              时间会影响到延时)，频率会偏小.
 *
```

```
 * @param     maxval : 最大值(0 < maxval < 4096), (maxval + 1)必须大于等于
samples/2
 * @param     dt     : 每个采样点的延时时间(单位: us)
 * @param     samples: 采样点的个数, samples 必须小于等于(maxval + 1) * 2 ,
                       且 maxval 不能等于 0
 * @param     n      : 输出波形个数,0~65535
 *
 * @retval    无
 */
void dac_triangular_wave(uint16_t maxval, uint16_t dt, uint16_t samples,
                                                              uint16_t n)
{
    uint16_t i, j;
    float incval;                                   /* 递增量 */
    float Curval;                                   /* 当前值 */

    if(samples > ((maxval + 1) * 2))return ;        /* 数据不合法 */

    incval = (maxval + 1) / (samples / 2);          /* 计算递增量 */

    for(j = 0; j < n; j++)
    {
        Curval = 0;                                 /* 先输出 0 */
        HAL_DAC_SetValue(&g_dac_handle,DAC_CHANNEL_1,DAC_ALIGN_12B_R,Curval);
        for(i = 0; i < (samples / 2); i++)          /* 输出上升沿 */
        {
            Curval  +=  incval;                     /* 新的输出值 */
            HAL_DAC_SetValue(&g_dac_handle,DAC_CHANNEL_1,DAC_ALIGN_12B_R,
                                                                  Curval);
            delay_us(dt);
        }
        for(i = 0; i < (samples / 2); i++)          /* 输出下降沿 */
        {
            Curval  -=  incval;                     /* 新的输出值 */
            HAL_DAC_SetValue(&g_dac_handle,DAC_CHANNEL_1,DAC_ALIGN_12B_R,
                                                                  Curval);
            delay_us(dt);
        }
    }
}
```

该函数用于设置 DAC 通道 1 输出三角波，输出频率 ≈ 1000 / (dt * samples) Khz，形参意义在源码已经有详细注释。该函数中，我们使用 HAL_DAC_SetValue 函数来设置 DAC 的输出值，这样得到的三角波在示波器上可以看到。如果有跳动现象（不平稳），是正常的，因为调用函数，计算等都需要时间，这样就会导致输出的波形是不太稳定的。越高性能的 MCU，得到的波形会越稳定。除此之外，用 HAL 库函数操作的效率没有直接操作寄存器高，如果对波形质量要求较高，可以直接操作 DHR12R1 寄存器，这样得到的波形会相对稳定些。

最后在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t t = 0;
    uint8_t key;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    dac_init(1);                                /* 初始化 DAC1_OUT1 通道 */
```

```
    lcd_show_string(30, 50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30, 70, 200, 16, 16, "DAC Triangular WAVE TEST", RED);
    lcd_show_string(30, 90, 200, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(30, 110, 200, 16, 16, "KEY0:Wave1  KEY1:Wave2", RED);
    lcd_show_string(30, 130, 200, 16, 16, "DAC None", BLUE); /* 默认提示无输出 */

    while (1)
    {
        t++;
        key = key_scan(0);                      /* 按键扫描 */
        if (key == KEY0_PRES)                   /* 高采样率 , 约 88.5Hz 波形 */
        {
            lcd_show_string(30, 130, 200, 16, 16, "DAC Wave1 ", BLUE);
            /* 幅值 4095, 采样点间隔 5us, 2000 个采样点, 100 个波形 */
            dac_triangular_wave(4095, 5, 2000, 100);
            lcd_show_string(30, 130, 200, 16, 16, "DAC None  ", BLUE);
        }
        else if (key == KEY1_PRES)              /* 低采样率 , 约 100Hz 波形 */
        {
            lcd_show_string(30, 130, 200, 16, 16, "DAC Wave2 ", BLUE);
            /* 幅值 4095, 采样点间隔 500us, 20 个采样点, 100 个波形 */
            dac_triangular_wave(4095, 500, 20, 100);
            lcd_show_string(30, 130, 200, 16, 16, "DAC None  ", BLUE);
        }

        if (t == 10)                            /* 定时时间到了 */
        {
            LED0_TOGGLE();                      /* LED0 闪烁 */
            t = 0;
        }
        delay_ms(10);
    }
}
```

该部分代码功能是，按下 KEY0 后，DAC 输出三角波 1，按下 KEY1 后，DAC 输出三角波 2，将 dac_triangular_wave 的形参代入公式：输出频率 ≈ 1000 / (dt * samples) Khz，得到三角波 1 和三角波 2 的频率都是 0.1KHZ，由于三角波 1 采样点之间时延比较小，频率会偏小。

4.5.3.4 下载验证

下载代码后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示如图 4.5.3.4.1 所示：

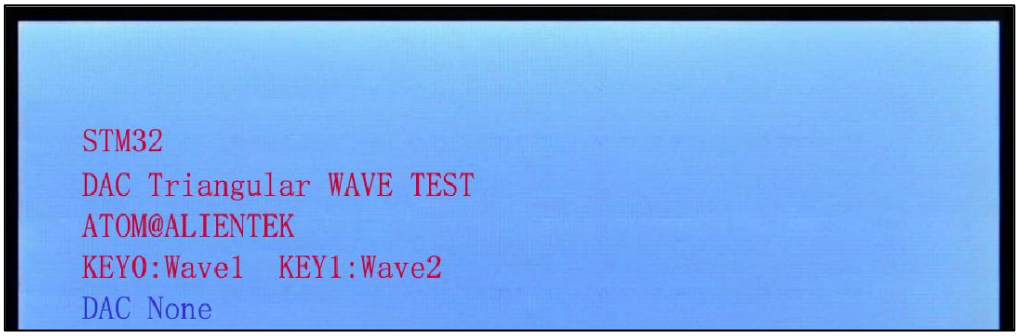


图 4.5.3.4.1 DAC 输出三角波实验测试图

没有按下任何按键之前，LCD 屏显示 DAC None，当按下 KEY0 后，DAC 输出三角波 1，LCD 屏显示 DAC Wave1，三角波 1 输出完成后 LCD 屏继续显示 DAC None，当按下 KEY1 后，DAC 输出三角波 2，LCD 屏显示 DAC Wave2，三角波 2 输出完成后 LCD 屏继续显示 DAC None。

其中三角波 1 和三角波 2 在示波器的显示情况如下图所示：

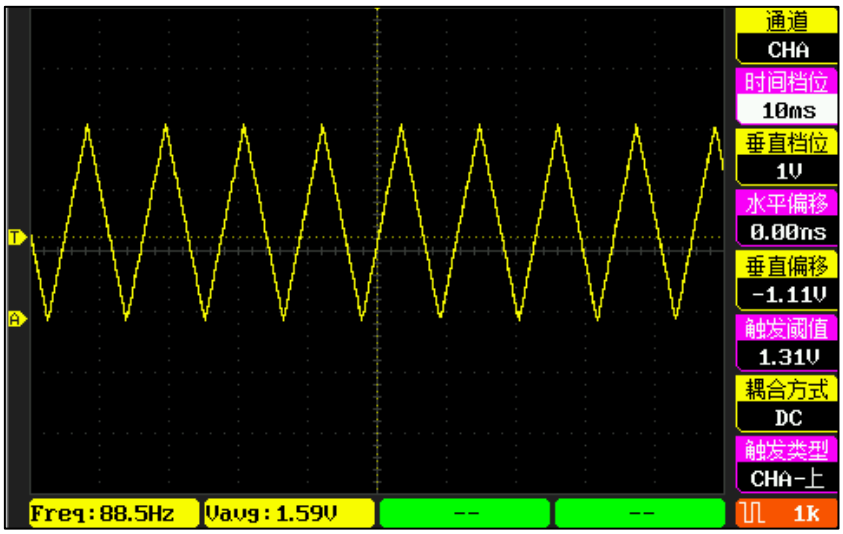


图 4.5.3.4.2 DAC 输出的三角波 1

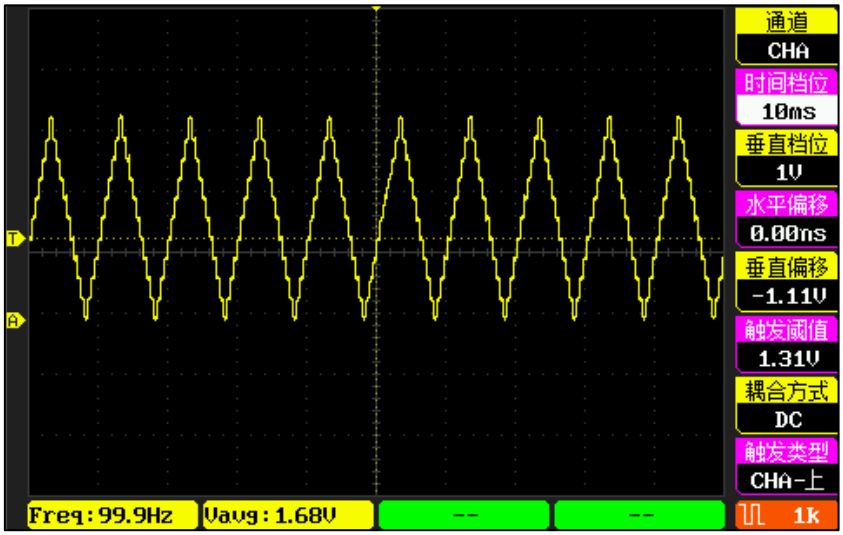


图 4.5.3.4.3 DAC 输出的三角波 2

由上面两副测试图可以知道，三角波 1 的频率是 88.5Hz，三角波 2 的频率是 99.9Hz。三角波 2 基本接近我们算出来的结果 0.1KHz，三角波 1 有较大误差，在介绍 dac_triangular_wave 函数时也说了原因，加上三角波 1 的采样率比较高，所以误差就会比较大。

4.5.4 DAC 输出正弦波实验

本实验我们来学习使用如何让 DAC 输出正弦波。实验将用定时器 2 的更新事件来触发 DAC 进行转换，输出正弦波，并以 DMA 的方式传输数据。

4.5.4.1 DAC 寄存器

本实验用到的寄存器在 DAC 输出实验都有介绍。

4.5.4.2 硬件设计

1. 例程功能

使用 DAC1 的通道 1 输出正弦波，需要通过示波器接 PA4 进行观察。TFTLCD 显示实验名称。LED0 闪烁，提示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）串口 1(PB6/PB7 连接在板载 USB 转串口芯片 CH340 上面)
3）正点原子 2.8/3.5/4.3/7/10 寸 TFTLCD 模块(仅限 MCU 屏，16 位 8080 并口驱动)
4）定时器 2
5）DAC1 ：
通道 1 - PA4
6）DMA(DMA1_Channel7)

3. 原理图

我们只需要把示波器的探头接到 DAC1 通道 1（PA4）引脚，就可以在示波器上显示 DAC 输出的波形。

4.5.4.3 程序设计

4.5.4.3.1 DAC 的 HAL 库驱动

下面将介绍本实验用到且没有介绍过的 HAL 库 API 函数。

1. HAL_DAC_Start_DMA 函数

启动 DAC 使用 DMA 方式传输函数，其声明如下：

```
HAL_StatusTypeDef HAL_DAC_Start_DMA(DAC_HandleTypeDef *hdac, uint32_t Channel,
                                    uint32_t *pData, uint32_t Length, uint32_t Alignment);
```

- **函数描述：**
  用于启动 DAC 使用 DMA 的方式。
- **函数形参：**
  形参 1 是 DAC_HandleTypeDef 结构体类型指针变量。
  形参 2 用于选择要启动的通道，可选择 DAC_CHANNEL_1 或者 DAC_CHANNEL_2。
  形参 3 是使用 DAC 输出数据缓冲区的指针。
  形参 4 是 DAC 输出数据的长度。
  形参 5 是指定 DAC 通道的数据对齐方式，有：DAC_ALIGN_8B_R（8 位右对齐）、DAC_ALIGN_12B_L（12 位左对齐）和 DAC_ALIGN_12B_R（12 位右对齐）三种方式。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

2. HAL_TIMEx_MasterConfigSynchronization 函数

配置主模式下的定时器触发输出选择函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIMEx_MasterConfigSynchronization(
              TIM_HandleTypeDef *htim, TIM_MasterConfigTypeDef *sMasterConfig);
```

- **函数描述：**
  用于配置主模式下的定时器触发输出选择。
- **函数形参：**
  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。
  形参 2 是 TIM_MasterConfigTypeDef 结构体类型指针变量，用于配置定时器工作在主/从模式，以及触发输出(TRGO 和 TRGO2)的选择。
- **函数返回值：**
  HAL_StatusTypeDef 枚举类型的值。

DAC 输出正弦波配置步骤

1）开启 DACx、DMA 和输出通道的 GPIO 时钟，配置该 IO 口的模拟输出功能

首先开启 DACx、DMA 和 GPIO 的时钟，然后配置 GPIO 为模拟模式。本实验我们默认用到 DAC1 通道 1，对应 IO 是 PA4，它们的时钟开启方法如下：

```
__HAL_RCC_DAC_CLK_ENABLE ();            /* 使能 DAC 时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE();           /* 开启 GPIOA 时钟 */
__HAL_RCC_DMA1_CLK_ENABLE ();           /* 开启 DMA1 时钟 */
```

IO 口模拟输出功能是通过函数 HAL_GPIO_Init 来配置的。

2）配置 DMA 并关联 DAC

通过 HAL_DMA_Init 函数初始化 DMA，包括配置通道，外设地址，存储器地址，传输数据量等。

HAL 库为了处理各类外设的 DMA 请求，在调用相关函数之前，需要调用一个宏定义标识符，来连接 DMA 和外设句柄。这个宏定义为__HAL_LINKDMA。

3）初始化 DACx

通过 HAL_DAC_Init 函数来设置需要初始化的 DAC。该函数并没有设置任何 DAC 相关寄存器，也就是说没有对 DAC 进行任何配置，它只是 HAL 库提供用来在软件上初始化 DAC。

注意：该函数会调用 HAL_DAC_MspInit 函数来存放 DAC 和对应通道的 IO 时钟使能和初始化 IO 等代码。

4）配置定时器控制触发 DAC

通过 HAL_TIM_Base_Init 函数设置定时器溢出频率。

通过 HAL_TIMEx_MasterConfigSynchronization 函数配置定时器溢出事件用做触发器。

通过 HAL_TIM_Base_Start 函数启动计数。

5）产生正弦波序列

通过 dac_creat_sin_buf 函数产生一组正弦波序列，其本质上就是正弦波曲线上的点。

6）配置 DAC 通道并开启 DMA 传输

通过 HAL_DAC_ConfigChannel 函数来设置配置 DAC 的通道，根据需求设置触发类型以及输出缓冲等。

再通过 HAL_DAC_Start_DMA 函数启动 DMA 传输以及 DAC 输出。

4.5.4.3.2 程序流程图

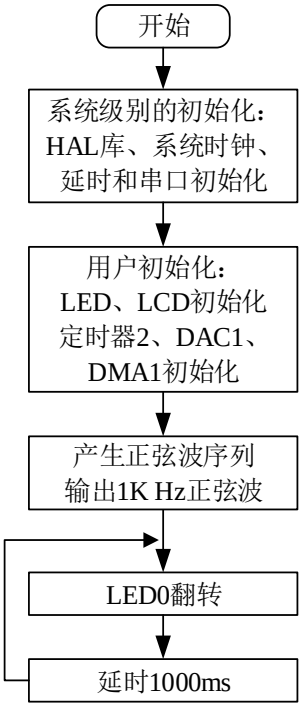


图 4.5.4.3.2.1 DAC 输出正弦波实验程序流程图

4.5.4.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。DAC 驱动源码包括两个文件：dac.c 和 dac.h。

dac.h 文件只有一些声明，下面直接开始介绍 dac.c 的程序，首先看产生正弦波序列的函数，其定义如下：

```
uint16_t g_dac_sin_buf[4096];                /* 正弦波序列数据缓冲区 */

/**
 * @brief       产生正弦波函序列
 * @note        需保证: maxval > samples/2
 *
 * @param       maxval : 峰值(0 < maxval < 2048)
 * @param       samples: 采样点的个数
 *
 * @retval      无
 */
void dac_creat_sin_buf(uint16_t maxval, uint16_t samples)
{
    uint8_t i;
    float outdata = 0;

    if( maxval <= (samples/2) )return ;         /* 数据不合法 */
    /*
     * 正弦波最小正周期为 2π，约等于 2 * 3.1415926
     * 曲线上相邻的两个点在 x 轴上的间隔 = 2 * 3.1415926 / 采样点数量
     * DAC 无法输出负电压，所以需要将曲线向上偏移一个峰值的量，让整个曲线都落在正数区域
     */
    float inc = (2 * 3.1415926) / samples;      /* 计算相邻两个点的 x 轴间隔 */

    for (i = 0; i < samples; i++)               /* 连续打 samples 个点 */
    {
        /*
```

```
         * 正弦波函数解析式：y = Asin(wx + φ) + b
         * 计算每个点的 y 值，将峰值放大 maxval 倍，并将曲线向上偏移 maxval 到正数区域
         */
        outdata = maxval * sin(inc * i) + maxval;

        if (outdata > 4095)
        {
            outdata = 4095;                         /* y 值上限限定 */
        }
        g_dac_sin_buf[i] = outdata;
    }
}
```

该函数用于产生正弦波序列，即正弦波曲线上的各个采样点，各个点最终控制的是 DAC_DHR12R1 寄存器的值。在 DAC 输出频率固定、波形周期一定的情况下，采样点越多，实际的曲线越接近于正弦波，但是波形的频率也会越小。

产生正弦波序列函数的实现思路请看注释，值得注意的是，DAC 是无法输出负电压的，所以我们需要将正弦波的曲线向上偏移一个峰值 maxval 的量，让整个曲线都落在正数区域。

下面介绍 DAC DMA 初始化函数，其定义如下：

```
/**
 * @brief       DAC DMA 初始化函数
 * @param       outx: 要初始化的通道.1,通道 1; 2,通道 2
 * @param       cndtr: DMA 通道单次传输数据量（采样点数量）
 * @retval      无
 */
void dac_init(uint8_t outx,uint16_t cndtr)
{
    GPIO_InitTypeDef gpio_init_struct;
    DAC_ChannelConfTypeDef DACCH1_Config;

    __HAL_RCC_DAC_CLK_ENABLE();             /* 使能 DAC 的时钟 */
    __HAL_RCC_GPIOA_CLK_ENABLE();           /* 使能 DAC OUT1/2 的 IO 口时钟 */
    __HAL_RCC_DMA1_CLK_ENABLE();

    /* STM32 单片机, 总是 PA4=DAC1_OUT1, PA5=DAC1_OUT2 */
    gpio_init_struct.Pin = (outx==1)? GPIO_PIN_4 : GPIO_PIN_5;
    gpio_init_struct.Mode = GPIO_MODE_ANALOG;
    gpio_init_struct.Pull = GPIO_PULLUP;
    HAL_GPIO_Init(GPIOA, &gpio_init_struct);

    g_dma_dac_handle.Instance = DMA1_Stream5;               /* DMA1 数据流 5 */
    g_dma_dac_handle.Init.Channel = DMA_CHANNEL_7;          /* 通道 7 */
    g_dma_dac_handle.Init.Direction = DMA_MEMORY_TO_PERIPH;/* 从存储器到外设 */

    g_dma_dac_handle.Init.PeriphInc = DMA_PINC_DISABLE;     /* 外设非增量模式 */
    g_dma_dac_handle.Init.MemInc = DMA_MINC_ENABLE;         /* 存储器增量模式 */
    /* 外设数据长度:16 位 */
    g_dma_dac_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    /* 存储器数据长度:16 位 */
    g_dma_dac_handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    g_dma_dac_handle.Init.Mode = DMA_CIRCULAR;              /* 循环模式 */
    g_dma_dac_handle.Init.Priority = DMA_PRIORITY_LOW;      /* 中等优先级 */
    g_dma_dac_handle.Init.FIFOMode = DMA_FIFOMODE_DISABLE; /* 不开启 FIFO */
    HAL_DMA_Init(&g_dma_dac_handle);                        /* 初始化 DMA */

    /* 把 DAC 和 DMA 关联 */
    __HAL_LINKDMA(&g_dac_dma_handle,DMA_Handle1,g_dma_dac_handle);

    g_dac_dma_handle.Instance = DAC;
    HAL_DAC_Init(&g_dac_dma_handle);                        /* 初始化 DAC */

    /* 使用定时器 2 的 TRGO 事件触发 DAC 转换 */
```

```
    DACCH1_Config.DAC_Trigger = DAC_TRIGGER_T2_TRGO;
    /* DAC1 输出缓冲关闭 */
    DACCH1_Config.DAC_OutputBuffer = DAC_OUTPUTBUFFER_DISABLE;

    dac_creat_sin_buf(2048,cndtr);                /* 产生正弦波序列，即画点 */

    switch(outx)
    {
        case 1:
            HAL_DAC_ConfigChannel(&g_dac_dma_handle,&DACCH1_Config,
                                  DAC_CHANNEL_1);             /* DAC 通道 1 配置 */
            HAL_DAC_Start_DMA(&g_dac_dma_handle,DAC_CHANNEL_1,
           (uint32_t *)g_dac_sin_buf,cndtr,DAC_ALIGN_12B_R); /* 开启 DAC 通道 1 */
            break;
        case 2:
            HAL_DAC_ConfigChannel(&g_dac_dma_handle,&DACCH1_Config
                                  DAC_CHANNEL_2);             /* DAC 通道 2 配置 */
           HAL_DAC_Start_DMA(&g_dac_dma_handle,DAC_CHANNEL_2,
           (uint32_t *)g_dac_sin_buf,cndtr,DAC_ALIGN_12B_R); /* 开启 DAC 通道 2 */
           break;
        default:break;
    }
}
```

该函数用于初始化 DAC 用 DMA 的方式输出正弦波，我们采用定时器 2 触发 DAC 进行转换输出。本函数用到的 API 函数起前面都介绍过，请结合前面介绍过的相关内容来理解源码。值得注意的是，这里调用了 dac_creat_sin_buf 函数来产生正弦波序列，采样点的个数通过入口参数 cndtr 传进来。

接着看定时器相关的 gtim.c 文件，我们只关注通用定时器的初始化函数，其定义如下：

```
/**
 * @brief       通用定时器 TIMX 初始化函数
 * @param       arr: 自动重装值。
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void gtim_timx_int_init(uint16_t arr, uint16_t psc)
{
    TIM_MasterConfigTypeDef sMasterConfig = {0};
    GTIM_TIMX_INT_CLK_ENABLE();                                 /* 使能 TIMx 时钟 */

    g_timx_handle.Instance = GTIM_TIMX_INT;                     /* 通用定时器 x */
    g_timx_handle.Init.Prescaler = psc;                         /* 预分频系数 */
    g_timx_handle.Init.CounterMode = TIM_COUNTERMODE_UP;        /* 递增计数模式 */
    g_timx_handle.Init.Period = arr;                            /* 自动装载值 */
    HAL_TIM_Base_Init(&g_timx_handle);

    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;        /* 更新事件用于触发 */
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    HAL_TIMEx_MasterConfigSynchronization(&g_timx_handle, &sMasterConfig);

    HAL_TIM_Base_Start(&g_timx_handle);                         /* 使能定时器 x */
}
```

该函数调用了 HAL_TIM_Base_Init 函数对定时器 2 的基本参数进行了初始化，并通过 HAL_TIMEx_MasterConfigSynchronization 函数配置 TIM2，将其更新事件作为 DAC 输出的触发源，然后开启定时器计数。

最后在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);     /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
```

```
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    gtim_timx_int_init(10-1,84-1);              /* 初始化定时器，定时触发 DAC */
    dac_init(1,100);     /* 初始化 DAC1_OUT1 通道,100 个点，正弦波频率 1k Hz */

    lcd_show_string(30,  50, 200, 16, 16, "STM32", RED);
    lcd_show_string(30,  70, 200, 16, 16, "DAC DMA Sine WAVE TEST", RED);
    lcd_show_string(30,  90, 200, 16, 16, "ATOM@ALIENTEK", RED);

    while (1)
    {
        LED0_TOGGLE();                          /* LED0 翻转 */
        delay_ms(1000);
    }
}
```

gtim_timx_int_init 函数初始化定时器 2，其计数频率为 1M Hz，计数溢出（发生一次更新事件）的频率为 100K Hz，不记得怎么计算的朋友，请回顾基本定时器的相关内容，这里直接把公式列出：

$$Tout= ((arr+1)*(psc+1))/Tclk$$

看到 dac_init(1,100);这个语句，第一个形参是 DAC 的通道，本实验使用通道 1；第二个形参是采集点的个数，本实验默认的采集点个数为 100。可以得到正弦波的频率为 100KHz/100 = 1KHz。

4.5.4.4 下载验证

下载代码后，可以看到 LED0 不停的闪烁，提示程序已经在运行了。LCD 显示如图 4.5.4.4.1 所示：

图 4.5.4.4.1 DAC 输出三角波实验测试图

接下来我们借助示波器来观察波形，注意要将探头接到 PA4 的排针上（注意开发板和示波器共地）。实际输出的 1KHz（100 个采样点）正弦波如下图所示：

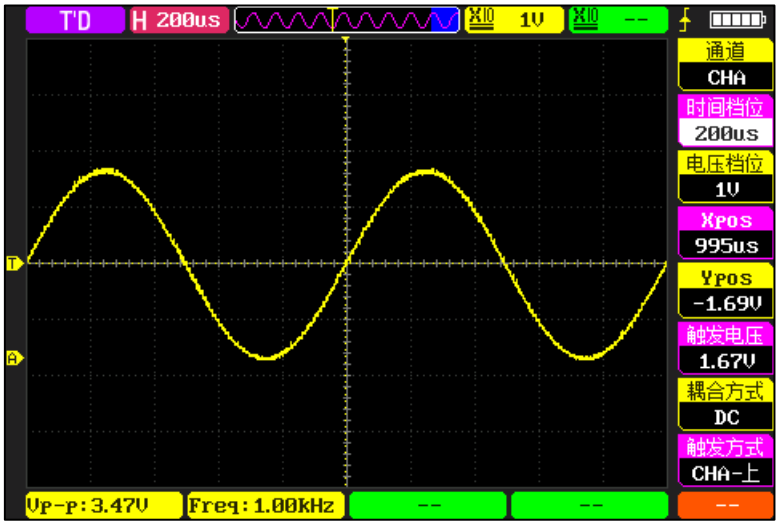


图 4.5.4.4.2 DAC 输出的 1K Hz 正弦波

第二篇 电机专题篇

功夫不负有心人，相信学习至此你已经掌握了基础篇介绍的知识。我们希望通过前面的章节你已经掌握了电机的一些基础知识以及电机控制相关的一些 STM32 外设。本篇我们将和大家一起来学习电机专题教程，实现电机的基础驱动、各类算法以及闭环控制，这些知识在实际项目中经常会用到，希望大家认真学习和掌握，以便将来更好、更快的完成实际项目开发。

本篇将采取一章一实例的方式，由浅入深，从最基础的电机控制到各种优秀算法，带领大家进入电机控制的精彩世界。

第 5 章 直流有刷电机

本章我们主要来学习直流有刷电机及其驱动板的基础知识，并实现直流有刷电机的基础驱动实验。

本章分为如下几个小节：

5.1 直流有刷电机简介
5.2 直流有刷电机原理
5.3 直流有刷驱动板
5.4 直流有刷电机基础驱动实验

5.1 直流有刷电机简介

1）概念：直流有刷电机（BDC）是一种内含电刷装置，可以将直流电能转换成机械能的电动机。

2）结构：直流有刷电机由定子、转子、电刷和换向器这四个结构组成。拆解后的直流有刷电机如图 5.1.1 所示：

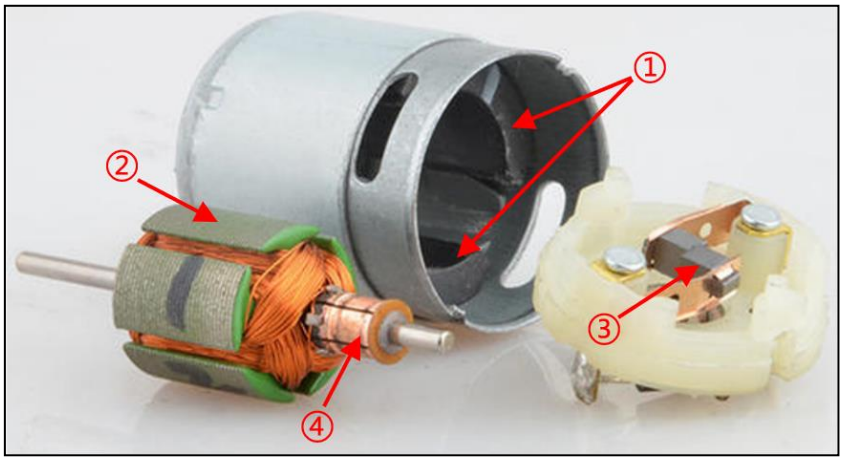


图 5.1.1 直流有刷电机拆解图

下面我们来分析一下图 5.1.1 中各个结构的作用：

① 定子：用于产生固定的磁场，通常由永磁体或电磁绕组制成。
② 转子：由一个或多个铜线绕组构成，通电后可以在磁场中受力运动。
③ 电刷：将外部电流输入到转子绕组上。
④ 换向器：改变转子绕组中电流的流向，是电机可以持续转动的关键结构。

3）优缺点：

优点	缺点
调速性能好、驱动简单、操控方便、成本低	寿命短、可靠性差、换向火花易产生电磁干扰

表 5.1.1 直流有刷电机优缺点

4）应用场景：直流有刷电机被广泛应用于电动玩具车、电风扇、汽车座椅、印刷机械等方面。

在一些对电机输出扭矩（即输出的力）有高要求的场景，我们会给直流有刷电机加上减速齿轮组，以增大输出扭矩，这一类电机就是**直流有刷减速电机**。其原理为：在电机输出功率一定的条件下，转速和扭矩成反比例关系，我们通过减速齿轮组降低电机的转速，即可提高电机的输出扭矩。

在直流有刷电机专题教程中，我们所使用的电机就是直流有刷减速电机，它的实物图如图 5.1.2 所示：

图 5.1.2 直流有刷减速电机

5）电机参数：

① 额定电压：电机正常工作的电压。
② 额定电流：也叫负载电流，电机带负载正常工作时的电流。
③ 额定转速：也叫负载转速，单位是 r/min，也常用 RPM 表示。
④ 额定扭矩：电机额定电流下输出力的大小，单位常用 kg·cm 或 N·m。
⑤ 减速比：电机转子原始转速与减速后的输出转速之比，表示为 N：1。

注意：电机不要使用过大负载，以防止电机堵转，造成电机过热甚至烧毁！

5.2 直流有刷电机原理

1）左手定则

直流有刷电机原理的本质是：**通电导线在磁场中受力运动**。因此，我们要了解直流有刷电机的工作原理，就离不开一个物理知识：左手定则。

左手定则是判断通电导线处于磁场中时，所受安培力 F (或运动)的方向、磁感应强度 B 的方向以及通电导体棒的电流 I 三者方向之间的关系的定律。

左手定则的具体内容：将左手四指并拢伸直，使拇指与其他四指在平面内垂直，手掌方向代表磁场的方向（从 N 级到 S 级），四指代表电流的方向（从正极到负极），那拇指所指的方向就是受力的方向。我们可以借助示意图对左手定则进行理解，示意图如图 5.2.1 所示：

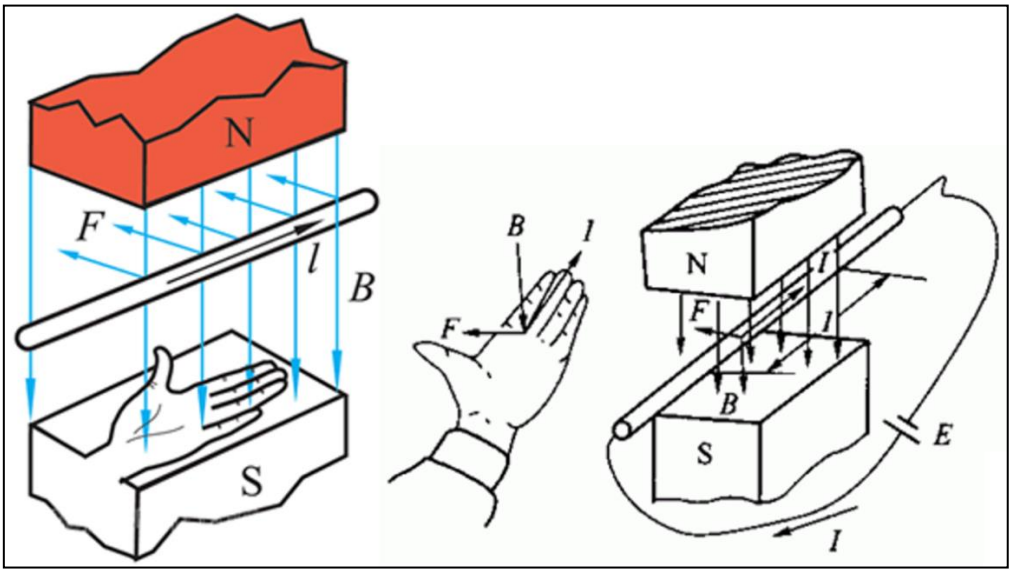


图 5.2.1 左手定则示意图

2）基本工作原理

有了左手定则这一个理论支撑后，我们接着来分析直流有刷电机的基本工作原理。为了

方便分析，我们先把直流有刷电机的结构简化，简化后的结构如图 5.2.2 所示：

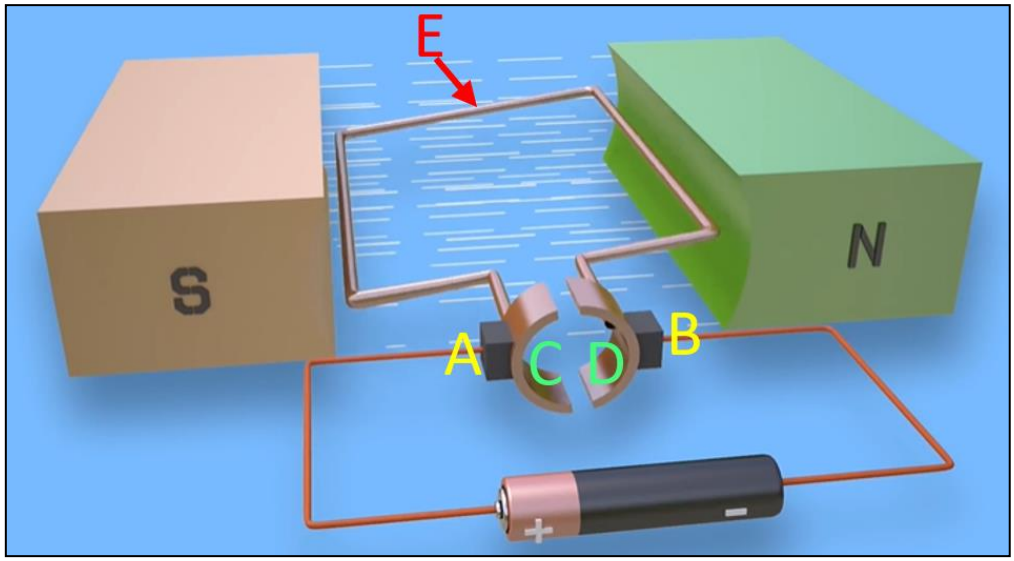


图 5.2.2 直流有刷电机结构简图

上图 5.2.2 中， A 和 B 代表的是两块电刷，C 和 D 代表两片换向器，E 代表简化后的单匝转子线圈，S 和 N 为两块定子磁极，磁场方向从 N 极到 S 极。在上一小节中我们已经介绍了直流有刷电机各个结构的作用，如果大家还没有理解的话，可以回顾 5.1 章节。

我们接下来具体分析直流有刷电机的基本工作原理，其工作原理如图 5.2.3 所示：

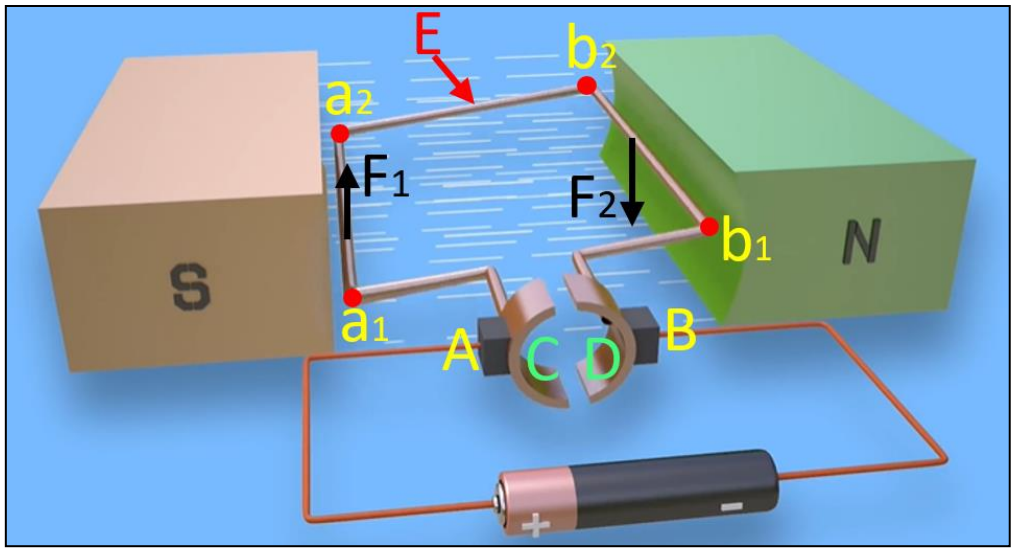


图 5.2.3 直流有刷电机工作原理

图 5.2.3 中，直流有刷电机的工作原理可分为以下几步：

第一步，电流从电池正极流出，进入电刷 A，经过换向器 C，输入到转子线圈 E 的左侧导线，此时已经知道电流的方向（从 a1 到 a2）和磁场的方向（N 极到 S 极），根据左手定则，可以判断出线圈 E 左侧导线的受力 F1 是垂直于导线向上的。

第二步，电流从转子线圈 E 的左侧流向右侧（从 b2 到 b1），经过换向器 D 和电刷 B，流回电池负极，同理，我们根据左手定则，就可以判断出线圈 E 右侧导线的受力 F2 是垂直于导线向下的。结合转子线圈 E 左右两侧导线的受力情况，可得知线圈会沿顺时针方向转动。

第三步，当转子线圈 E 沿顺时针方向转过一定角度（在简化模型上就是 90°，实际应用中并不一定），换向器 C 和 D 的位置会改变，此时电刷和换向器的接触关系就改变了，如图 5.2.4 所示：

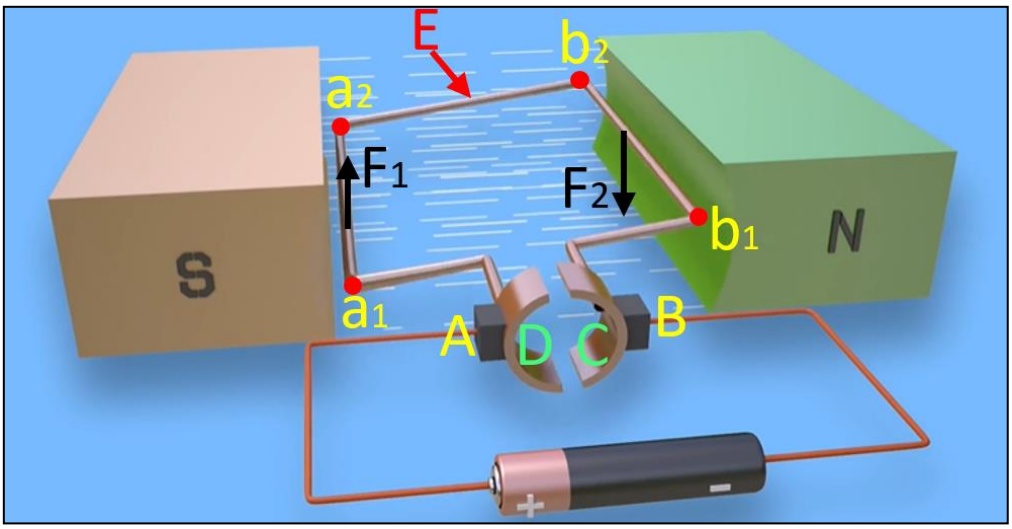


图 5.2.4 换向器位置改变

上图 5.2.4 中，电流的方向发生了变化，此时电流从电刷 A 流入换向器 D，经过转子线圈 E，再流出到换向器 C 和电刷 B。这个时候，虽然转子线圈 E 中电流的方向发生了改变，但是从转子线圈 E 的整体受力来看，左右两侧导线受力的方向并没有改变，所以转子线圈 E 会继续沿顺时针方向旋转下去，这个就是直流有刷电动机最基本的工作原理。

注意：单匝转子线圈运动到磁场的不同位置时，其受力是不均匀的，因此，在实际的应用中，转子通常都会有三匝或以上的线圈，它的受力情况会复杂很多，但是基本的原理是相通的。

3）调速原理

直流有刷电机的调速原理很简单，我们只需要改变电源电压（输入电流）的大小，转子线圈的受力也会随之变化，这样就可以改变电机的转速了。在实际的应用中，我们就是通过控制直流有刷电机的输入电压来实现调速的。

4）换向原理

根据直流有刷电机的工作原理可知，当转子线圈中电流的方向发生变化，线圈的受力情况也会发生改变，因此，我们只需要改变转子线圈中的电流方向即可实现电机换向。

5.3 直流有刷驱动板

本小节我们主要介绍直流有刷驱动板的 H 桥原理、硬件接口以及实物接线，关于直流有刷驱动板产品简介部分，大家可以回顾 2.2 章节，这里不再赘述。

1）简易 H 桥驱动原理

要让直流有刷电机转动起来是非常简单的：在允许的电压范围之内，给它提供稳定的直流电源即可使其转动。在此基础上，如果还需要实现电机换向，就要相应地调整电源的正负极接线了。为了方便地实现直流有刷电机的换向操作，我们引入了相关的控制电路：H 桥电路。

接下来看一个使用三极管搭建的简易 H 桥电路，如图 5.3.1 所示：

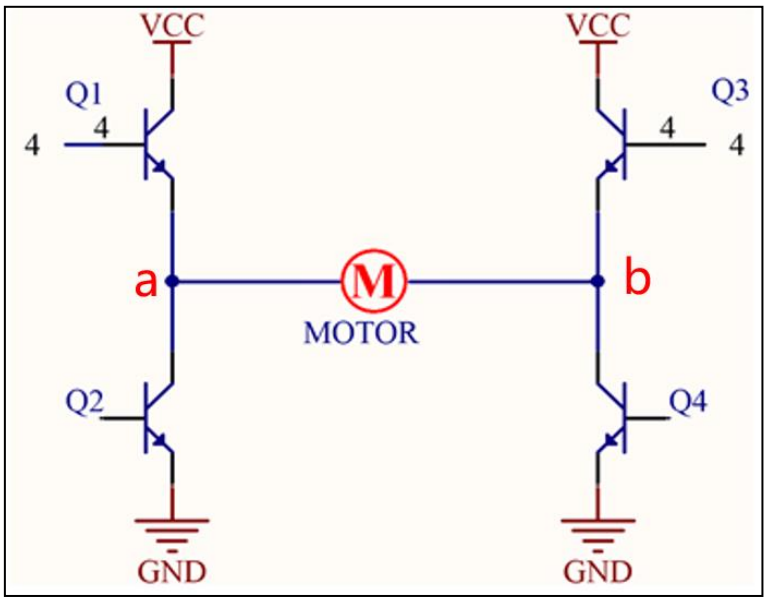


图 5.3.1 三极管搭建的简易 H 桥电路

图 5.3.1 是使用三极管搭建的简易 H 桥电路，其中 MOTOR 表示直流有刷电机，Q1、Q2、Q3 和 Q4 为 4 个三极管，其中 Q1 和 Q3 接在了电源正极，Q2 和 Q4 接在了电源负极。

上图的 H 桥搭建全部使用的是 NPN，并且导通逻辑都是基极为高电平时导通。如果 Q1 和 Q4 三极管导通，那么电机的电流方向是从点 a 到点 b，假设此时电机正转；如果 Q2 和 Q3 三极管导通，那么电机的电流方向是从点 b 到点 a，此时电机就反转了。上述就是 H 桥控制电机正反转的逻辑原理。在驱动电机时，必须保证 H 桥的同一侧三极管不会同时导通，否则将会发生短路，比如：Q1 和 Q2 同时导通或者 Q3 和 Q4 同时导通，这都是不可取的。

2）完整的 H 桥电路

在大功率的直流有刷驱动板中，H 桥电路往往是用 MOS 管搭建的。我们在简易 H 桥的基础上，加入一些控制电路和保护电路，这就组成了一个相对完整的 H 桥电路。下面我们以 ATK-PD6010D 直流有刷驱动板的 H 桥电路为例，简要分析一下实际应用中的 H 桥电路设计，具体电路如图 5.3.2 所示：

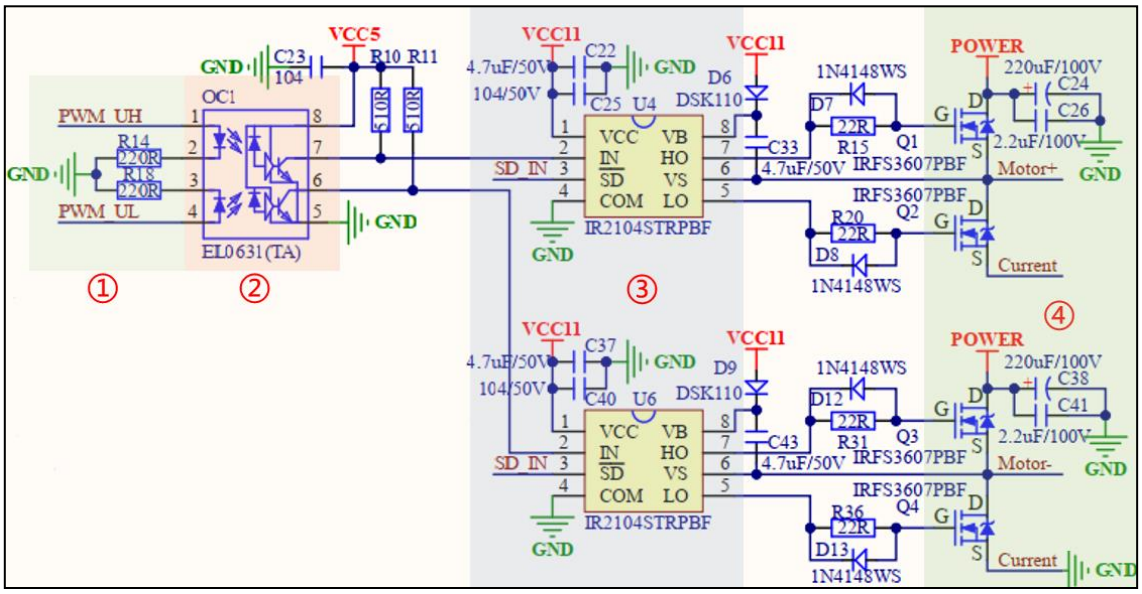


图 5.3.2 完整 H 桥驱动电路

图 5.3.2 中，我们只标注了主要的电路，不同序号所代表电路功能如下：

① 控制信号输入电路；

② 光耦隔离保护电路，控制信号经过该光耦之后会反相；

③ 全桥驱动电路，由 2 个 IR2104 半桥驱动芯片组成，可以驱动 4 个 MOS 管组成的 H 桥，除此之外，该电路还可以防止 H 桥的同一侧 MOS 管同时导通；

④ H 桥电路，由 4 个 MOS 组成，其中 Q1 和 Q3 连接到电源的正极，Q2 和 Q4 连接到电源的负极，这一部分电路的原理和简易 H 桥的是一样的。需要注意的是，直流有刷电机的电源接口并没有正负极之分，这里的 Motor+、Motor-只是为了方便接线和描述。

接下来我们以一个实例来理解一下这个完整的 H 桥电路的控制逻辑，具体的逻辑如图 5.3.3 所示：

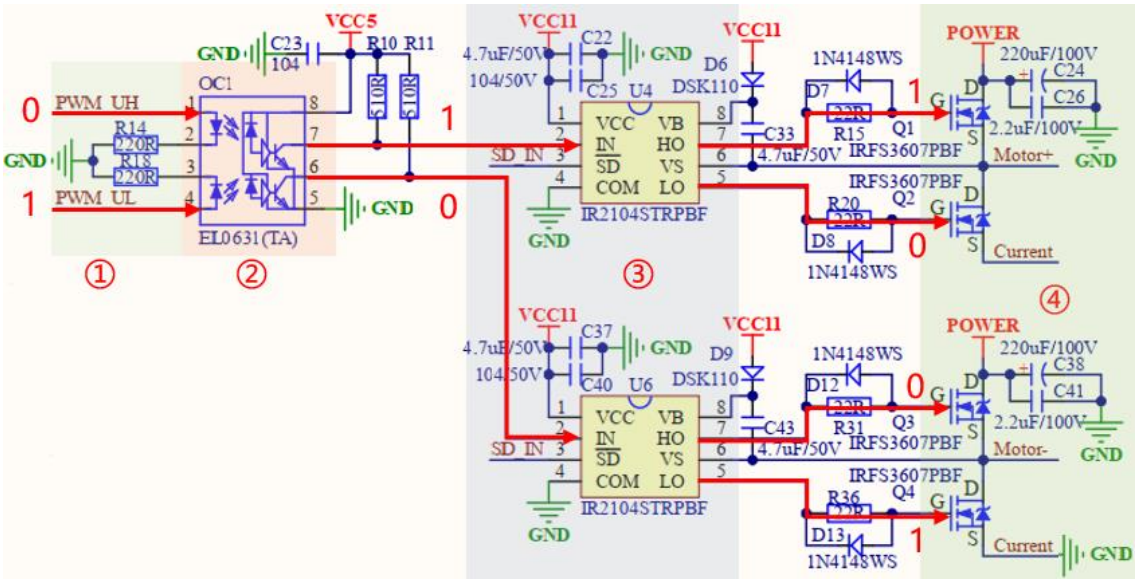


图 5.3.3 完整的 H 桥电路控制逻辑

图 5.3.3 中，我们从左到右，根据序号对控制逻辑进行分析：

① 首先给 EL0631 光耦（上图中序号②）输入两路相反的控制信号，0 代表低电平，1 代表高电平，其中 PWM_UH 为 0，PWM_UL 为 1。

② 控制信号经过光耦的反相之后进行输出，此时两路信号的高低电平状况就和之前相反了。反相后的两路信号分别输入到上、下两个 IR2104S 半桥芯片的 IN 引脚（上图中序号③）。

③ IR2104S 半桥芯片的 IN 和 SD 引脚是信号输入引脚，HO 和 LO 是信号输出引脚，输入信号和输出信号的逻辑如下表所示：

输入		输出	
IN（引脚 2）	SD（引脚 3）	HO（引脚 7）	LO（引脚 5）
0	**0**	**0**	**0**
1	**0**	**0**	**0**
0	**1**	**0**	**1**
1	**1**	**1**	**0**

表 5.3.1 半桥芯片输入输出信号逻辑表

从表 5.3.1 中我们可以得出三个结论：1、只要 SD 引脚输入低电平，所有的输出都会停止，即 HO 和 LO 都输出低电平；2、当 SD 引脚为高电平时，HO 引脚输出电平的高低状况和 IN 引脚是一致的；3、HO 和 LO 引脚永远不会同时输出高电平，这就保证了 H 桥上同一侧的 MOS 管不会同时导通，从而避免短路事故的发生。

④ 假设 SD 引脚输入了高电平，我们根据逻辑表，结合此时上下两个半桥芯片的 IN 引脚输入电平，就可以得出上下两个半桥芯片的输出电平状况（上图中序号④）。此时 Q1 和 Q4 导通，Motor+接到了 POWER（VCC），Motor-接到了 GND，电机就可以正常工作了（假设此时电机正转）。

上述的内容就是完整 H 桥电路的控制逻辑，接下来我们看一下它是如何控制直流有刷电机换向和速度的。

3）方向和速度控制

有了完整的 H 桥电路之后，就可以方便地控制电机的正反转方向和速度，具体的原理如下：

① 控制方向。只需要改变两路输入信号的电平状况即可，最简单的方法就是利用两个 IO 输出作为控制信号，IO 翻转即可改变电机方向。

② 电机调速。固定一路输入信号的电平，另一路信号利用 PWM 波作为输入，这样即可调节驱动板输出到电机的电压，从而实现电机的速度控制。

③ 同时控制方向和速度。此时两路的控制信号应满足：1、两路控制信号既可以固定输出高、低电平，又可以输出 PWM 波（不要求在同一时间）；2、当一路信号输出为 PWM 波的时候，另一路信号会固定输出一个电平。STM32 高级定时器的 PWM 互补输出功能就可以很好地满足这两个条件。

4）直流有刷驱动板接口

这里我们介绍一下直流有刷驱动板的一些接口，手把手地带大家搭建直流有刷电机专题的硬件平台，本教程所使用的 ATK-PD6010D 直流有刷驱动板接口如图 5.3.4 所示：

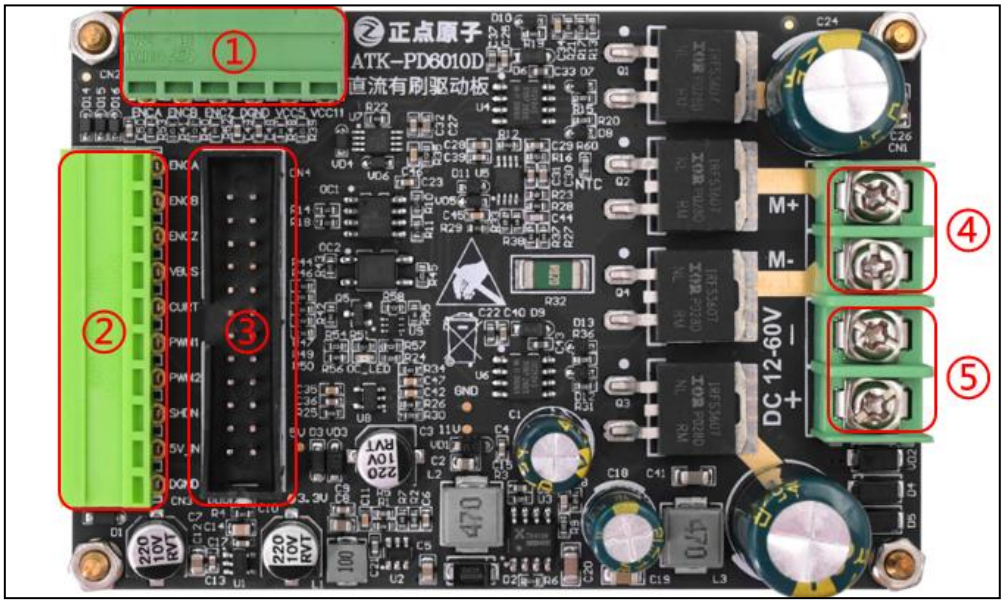


图 5.3.4 直流有刷驱动板接口

图 5.3.4 中，各个接口的功能如下：

① 编码器接口 CN2，包括 3 相编码器接口、5V 电源接口和 11V 电源接口。这个接口用于连接电机的编码器，实现电机的测速功能。该接口预留了 2 种供电电压的引脚，用户可根据实际使用的编码器工作电压来选择。

② 控制、采集信号备用接口 CN3，该接口是一个通用的接口，可以很方便的连接其他开发板，实现驱动板和开发板之间的信息交互，它的功能包括：两路 PWM 信号输入、SD（刹车）信号输入、电压电流温度检测信号输出和编码器三相信号输出。

③ 控制、采集信号专用接口 CN4，该接口是一个专用的接口，我们整合了直流有刷、无刷电机的驱动接口在一起，只需要通过一根 24P 排线，即可方便地连接 DMF407 电机开发板，实现驱动板和开发板之间的信息交互，它的功能和备用接口是一样的，在实际应用中，我们只需要用到这个接口的一部分引脚。

④ 电机电源输出接口，包括 M+、 M-两个接线端，实际上直流有刷电机的电源接线并没有正负之分，接线的不同只会影响电机旋转的方向，但是我们为了在教学中统一描述电机的正反转方向，这里默认把驱动板的 M+接到电机的 M+，M-接到电机的 M-接口。

注意：在后续的直流有刷电机实验中，如果你发现电机的旋转方向和例程源码描述不一致，请检查 M+和 M-的接线！

⑤ 驱动板电源输入接口，这个接口是驱动板的供电接口，同时，电机的电源也来自于这个接口，它可接入的电压范围是 DC12~60V。

注意：驱动板的输入电压需要根据电机的工作电压来确定！此电源接口区分正负极，大家在接驱动板输入电源的时候一定需要注意正负极接线！

5）直流有刷驱动板实物接线

下面我们将手把手地教大家搭建直流有刷电机驱动的硬件平台，这一部分内容十分重要，大家一定要掌握。首先需要准备一些材料，如图 5.3.5 所示：

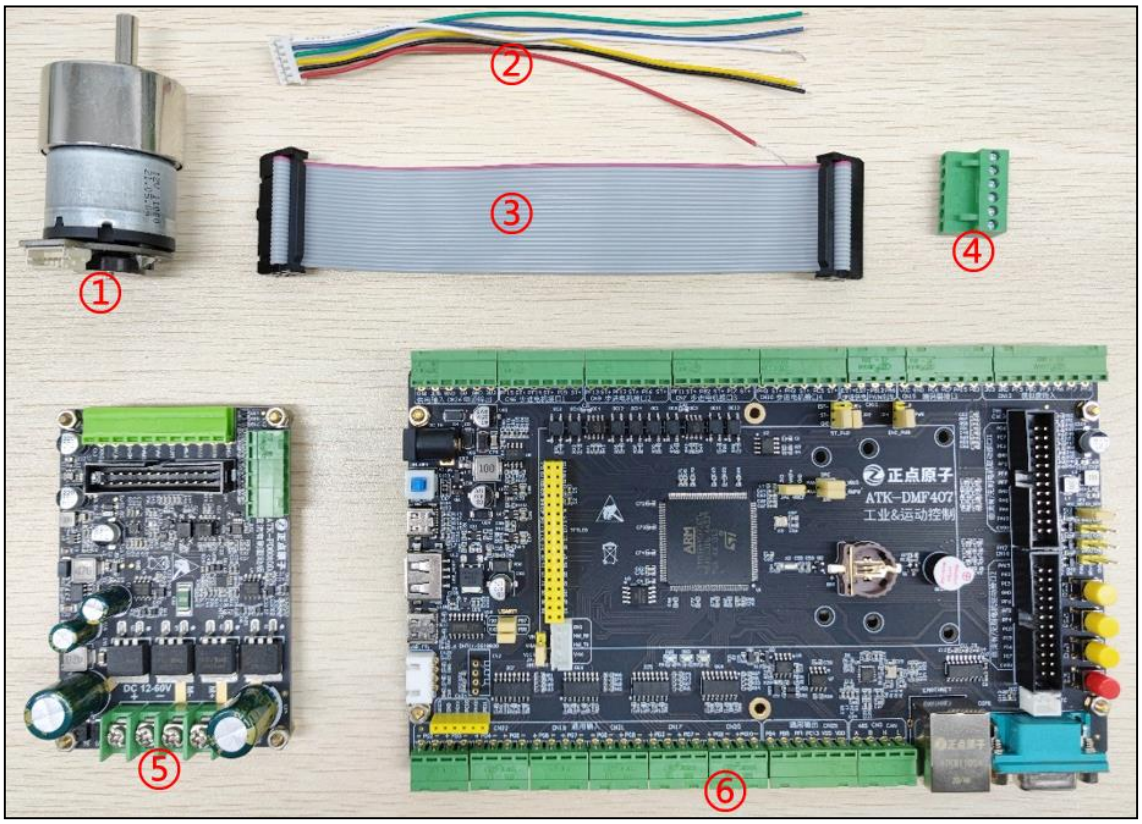


图 5.3.5 硬件平台搭建所需材料

图 5.3.5 中，不同的序号对应的材料如下：

① 直流有刷电机；
② 6pin 彩色排线；
③ 专用 24pin 排线（排线颜色不定）；
④ 编码器接口 6pin 端子；
⑤ 直流有刷驱动板；
⑥ 电机开发板；

除了之外，我们还需要准备一个符合电机工作电压的直流电源。本教程中所使用的直流有刷电机的工作电压是 12V，这里就准备一个 12V 的直流电源即可。

有了这些材料之后，我们接下来就进行实物的接线。首先用 24pin 的专用排线连接驱动板和电机，这里以开发板的直流有刷/无刷电机驱动接口 1 为例，接线如图 5.3.6 所示：

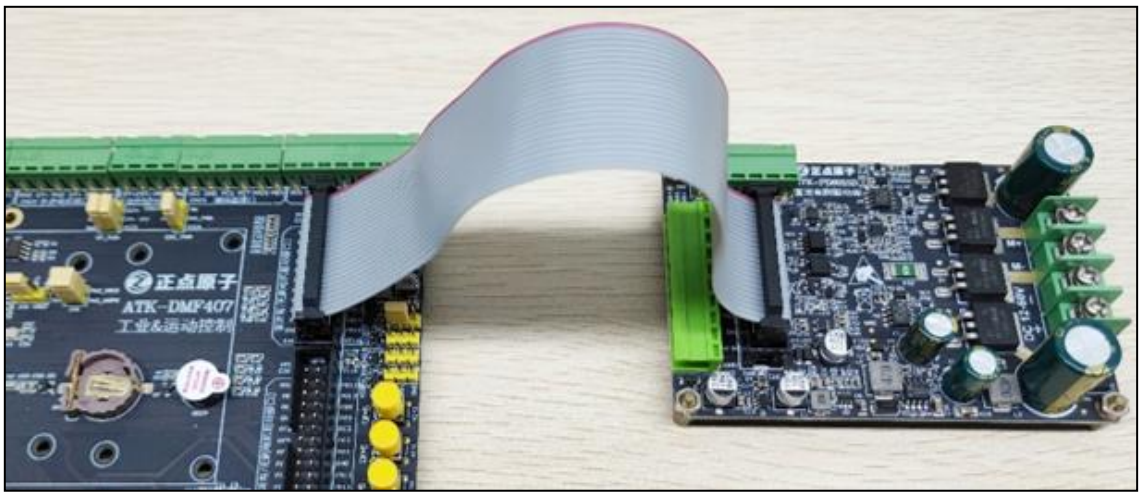


图 5.3.6 连接驱动板和开发板

注意：24pin 的排线接口是区分方向的，一定不要硬怼！

接好驱动板和开发板之后，我们先了解一下直流有刷电机的接口线序，具体如图 5.3.7 所示：

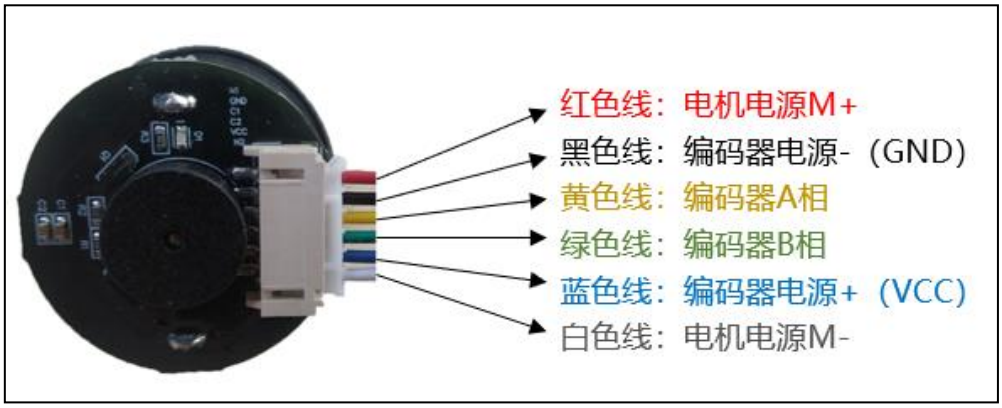


图 5.3.7 直流有刷电机接口线序

了解接口的线序之后，就可以开始连接驱动板和直流有刷电机了。我们只需要根据驱动板的丝印和电机的接口线序说明来连接它们即可，如图 5.3.8 所示：

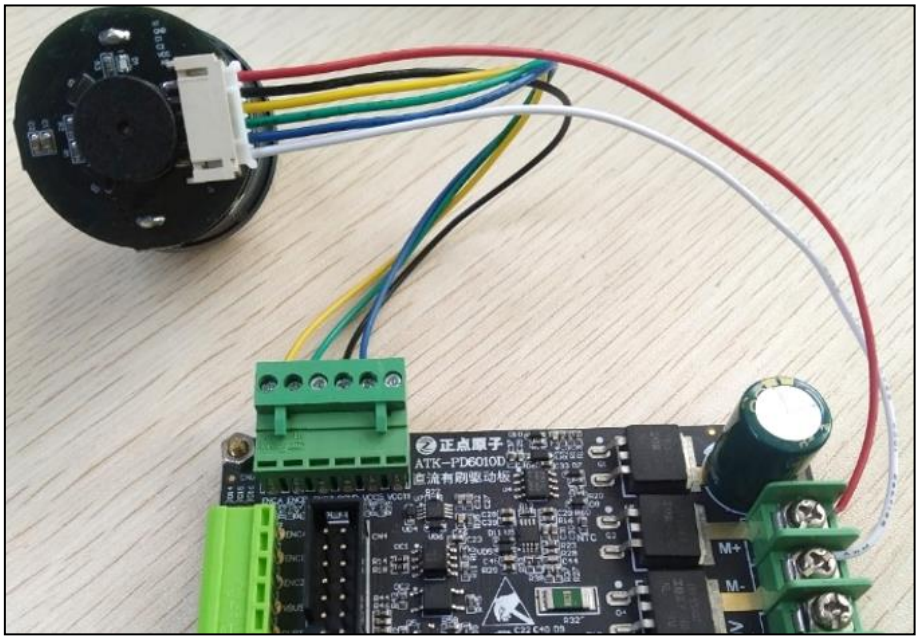


图 5.3.8 连接驱动板和电机

注意：1、为了在教学中统一描述正反转方向，电机的输出电源按 M+接 M+，M-接 M-的方式进行连接；2、编码器的电源电压有 5V 和 11V 两个选择，大家一定要根据实际编码器的工作电压来选择，例如本教程中所使用的编码器（装在直流有刷电机尾部）的工作电压是 5V，这里就接 VCC5 即可，同时还需要区分编码器电源的正负极。

接好驱动板和电机之后，就可以给驱动板供电了，如图 5.3.9 所示：

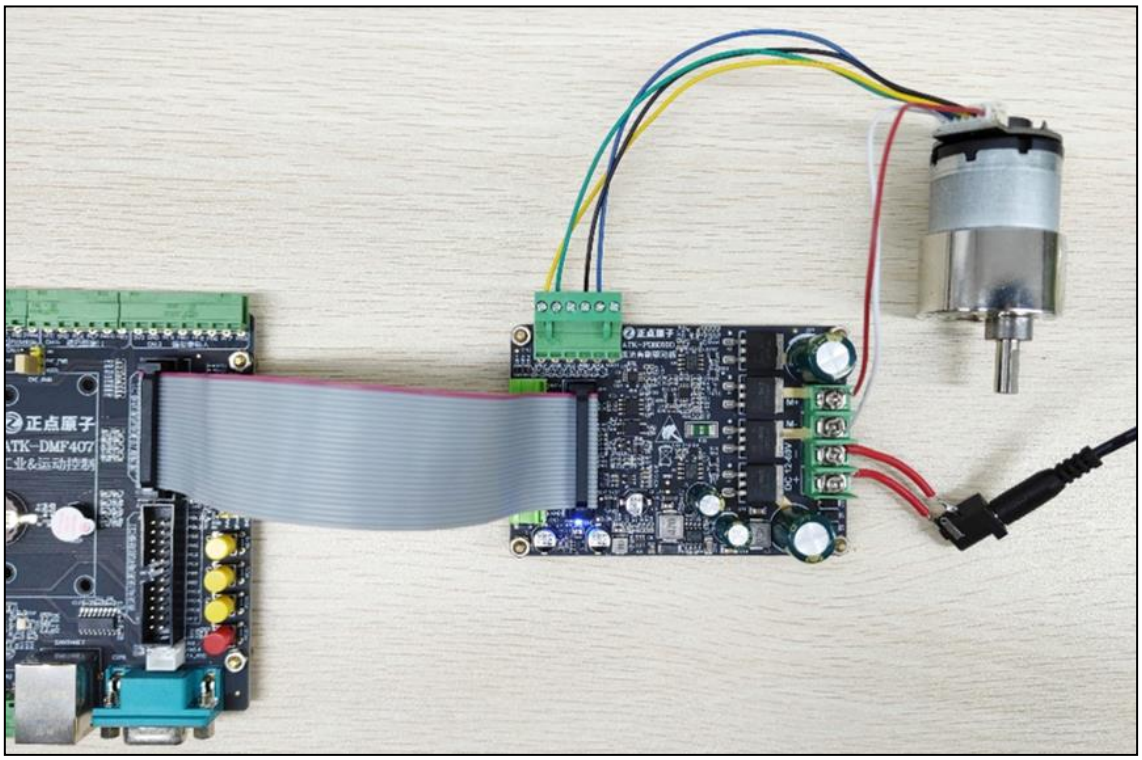


图 5.3.9 驱动板供电

我们在直流有刷电机教程中所使用的电机的工作电压是 12V，这里就给驱动板接一个 12V 的直流电源即可。这里再次强调一遍：驱动板电源的正负极一定不要接反！

注意：后续的直流有刷专题教程中，如果没有特别说明实物接线的，均按照上述接线方式（开发板需要单独供电）。至此，直流有刷驱动板的 H 桥电路原理以及实物连接就介绍完了，有了这些基础之后，我们就可以开始学习直流有刷电机的基础驱动了。

5.4 直流有刷电机基础驱动实验

本实验我们来学习直流有刷电机的基础驱动，大家需要注意，关于定时器配置以及底层寄存器的内容，这里不会再介绍，如果对这方面知识还不熟悉的话，大家可以回顾高级定时器的章节。

5.4.1 直流有刷电机基础驱动原理

直流有刷电机的基础驱动包括：调速和换向。在前面的 H 桥驱动电路中我们有介绍过，需要实现直流有刷电机的调速和换向，可以利用 STM32 高级定时器的 PWM 互补输出功能。具体的实现逻辑如下：

假设我们让 TIM1_CH1（主通道）输出 PWM 波，TIM1_CH1N（互补通道）固定输出高电平，此时只要调节主通道输出的 PWM 占空比即可调整电机上的电压，进而控制电机的速度。当电机需要换向的时候，我们就让主通道固定输出高电平，互补通道输出 PWM 即可。

5.4.2 硬件设计

1. 例程功能

1、本实验以电机开发板的直流有刷电机驱动接口 1 为例，利用 TIM1_CH1（PA8）和 TIM1_CHN1（PB13）输出控制信号来驱动电机，利用 PF10 来控制半桥芯片的 SD 引脚进行电机刹车。

2、当按键 0 按下，就增大 PWM 的比较值变量，当按键 1 按下，就减小 PWM 的比较值变量。比较值变量的绝对值越大，电机速度越快。当比较值变量为正数时电机正转，反之电机反转，按下按键 2 则马上停止电机。

3、LED0 闪烁指示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）独立按键
KEY0 - PE2
KEY1 - PE3
KEY2 - PE4
3）定时器 1
TIM1 主输出通道 PA8
TIM1 互补输出通道 PB13
4）SD（刹车）信号输出 PF10

3. 原理图

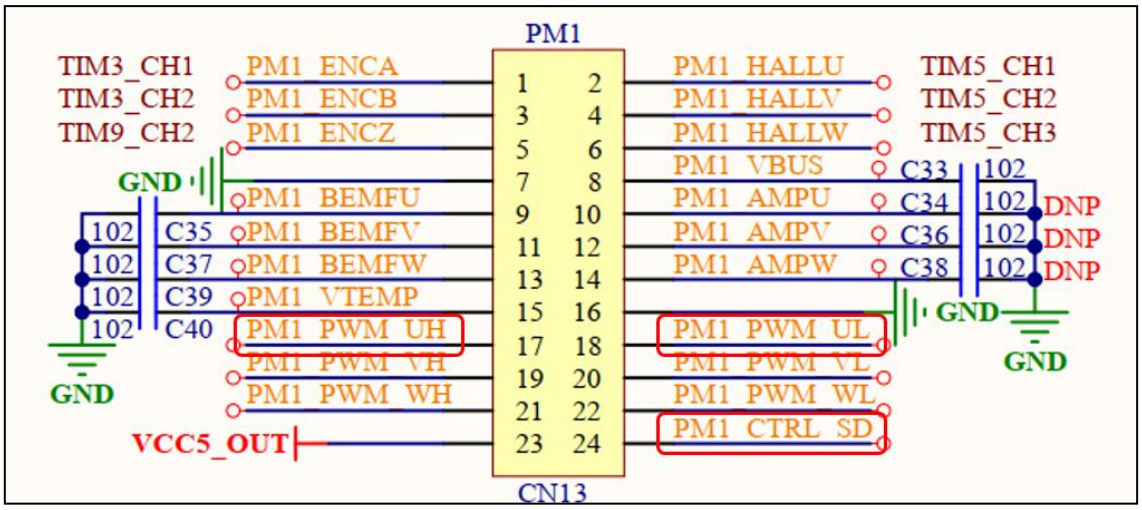


图 5.4.2.1 直流有刷电机接口原理图

图 5.4.2.1 就是 DMF407 电机开发板的直流有刷电机接口 1 的原理图，本实验我们只需要用到了 PM1_PWM_UH（PA8）、PM1_PWM_UL（PB13）以及 PM1_CTRL_SD（PF10）这 3 个引脚，其中 PA8 和 PB13 用于输出所需的 PWM 控制信号，PF10 用于输出刹车信号。

注意：因为有刷、无刷电机共用一套驱动接口，所以原理图、开发板丝印以及驱动板丝印的引脚名称不一定一一对应。为了方便大家使用直流有刷电机的两个驱动接口，我们把这些引脚的对应关系梳理出来，接口 1 引脚对应关系详见下表 5.4.2.1：

开发板接口丝印/原理图	直流有刷驱动板丝印	功能
PC6 / PM1_ENCA	ENCA	编码器接口 A
PC7 / PM1_ENCB	ENCB	编码器接口 B
PE6 / PM1_ENCZ	ENCZ	编码器接口 Z
PA0 / PM1_VTEMP	TEMP	温度检测引脚

PB1 / PM1_VBUS	VBUS	电压检测引脚
PB0 / PM1_AMPU	CURT	电流检测引脚
PA8 / PM1_PWM_UH	PWM1	PWM 主通道输出引脚
PB13 / PM1_PWM_UL	PWM2	PWM 互补通道输出引脚
PF10 / PM1_CTRL_SD	SHDN	刹车引脚

表 5.4.2.1 有刷电机驱动接口 1 引脚对应关系

直流有刷电机驱动接口 2 的引脚对应关系详见下表 5.4.2.2：

开发板接口丝印/原理图	直流有刷驱动板丝印	功能
PA15 / PM2_ENCA	ENCA	编码器接口 A
PB3 / PM2_ENCB	ENCB	编码器接口 B
PE5 / PM2_ENCZ	ENCZ	编码器接口 Z
PC0 / PM2_VTEMP	TEMP	温度检测引脚
PA5 / PM2_VBUS	VBUS	电压检测引脚
PC2 / PM2_AMPU	CURT	电流检测引脚
PI5 / PM2_PWM_UH	PWM1	PWM 主通道输出引脚
PH13 / PM2_PWM_UL	PWM2	PWM 互补通道输出引脚
PF2 / PM2_CTRL_SD	SHDN	刹车引脚

表 5.4.2.2 有刷电机驱动接口 2 引脚对应关系

5.4.3 程序设计

本实验用到的 HAL 库驱动请回顾高级定时器 PWM 互补输出实验的介绍。下面介绍一下直流有刷电机基础驱动实验中定时器的配置步骤。

基础驱动相关的定时器配置步骤

1）开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出。

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 1 通道 1，对应 IO 是 PA8，互补输出通道引脚是 PB13，它们的时钟开启方法如下：

```
__HAL_RCC_TIM1_CLK_ENABLE();            /* 使能定时器 1 */
__HAL_RCC_GPIOA_CLK_ENABLE();           /* 开启 GPIOA 时钟 */
__HAL_RCC_GPIOB_CLK_ENABLE();           /* 开启 GPIOB 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数。

使用定时器的 PWM 模式功能时，我们调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数来完成对定时器底层以及其输出通道 IO 的初始化，包括：定时器及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

3）设置定时器为 PWM 模式，输出比较极性，互补输出极性等参数。

通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为 PWM1 模式，根据需求设置 OCy 输出极性和 OCyN 互补输出极性等。

4）设置死区参数。

通过 HAL_TIMEx_ConfigBreakDeadTime 函数来设置死区参数，比如：设置死区时间、运行模式的关闭输出状态、空闲模式的关闭输出状态等，本实验没有用到定时器的刹车功能。

5）启动 Ocy 输出以及 OCyN 互补输出。

通过 HAL_TIM_PWM_Start 函数启动 OCy 输出，通过 HAL_TIMEx_PWMN_Start 函数启动启动 OCyN 互补输出，需要注意的是，PWM 输出的控制是放在电机的控制函数里面。

5.4.3.1 程序流程图

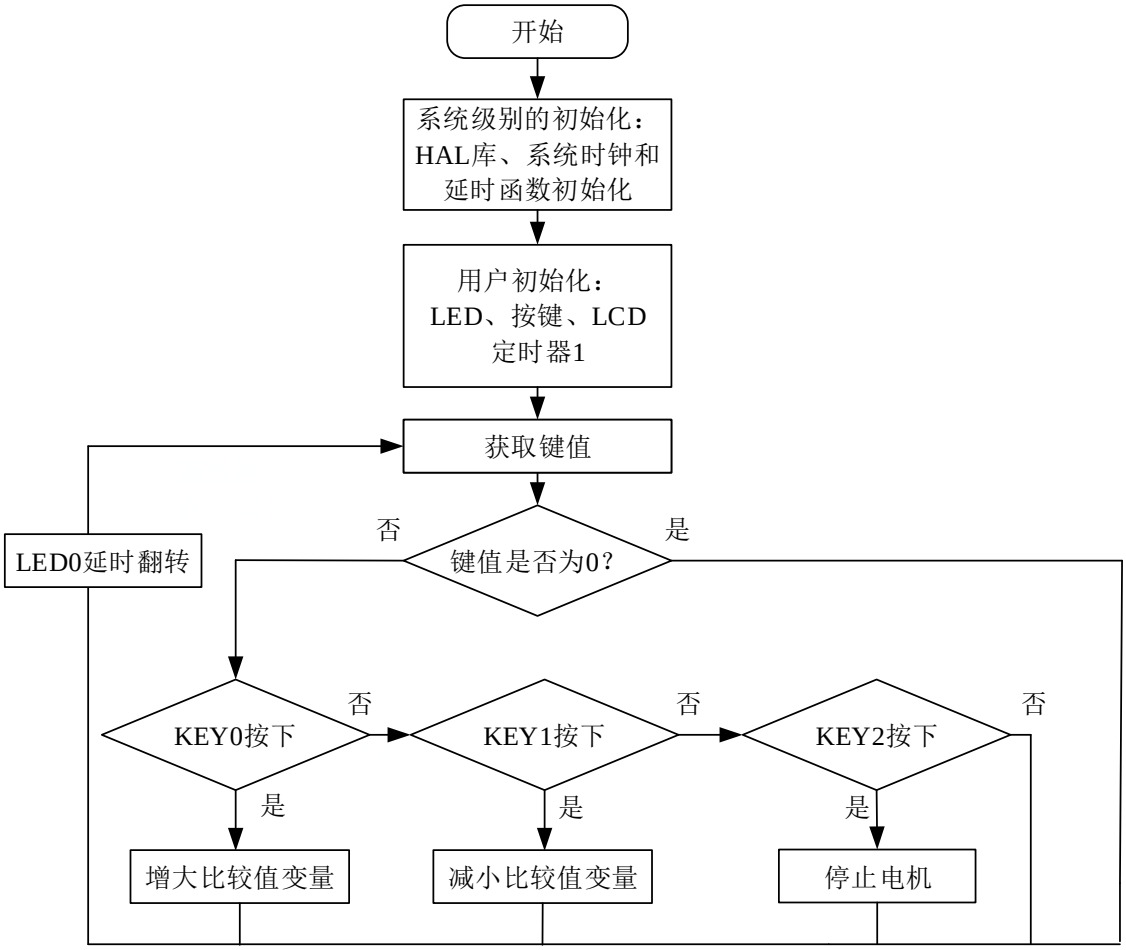


图 5.4.3.1.1 基础驱动程序流程图

5.4.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。直流有刷电机的定时器驱动源码包括两个文件：dcmotor_tim.c 和 dcmotor_tim.h。

首先看 dcmotor_tim.h 头文件的几个宏定义：

```
/* TIMX 互补输出模式 定义 */

/* 主输出通道引脚 */
#define ATIM_TIMX_CPLM_CHY_GPIO_PORT            GPIOA
#define ATIM_TIMX_CPLM_CHY_GPIO_PIN             GPIO_PIN_8
#define ATIM_TIMX_CPLM_CHY_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)    /* PA 口时钟使能 */

/* 互补输出通道引脚 */
#define ATIM_TIMX_CPLM_CHYN_GPIO_PORT           GPIOB
#define ATIM_TIMX_CPLM_CHYN_GPIO_PIN            GPIO_PIN_13
#define ATIM_TIMX_CPLM_CHYN_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)    /* PB 口时钟使能 */

/* TIMX 引脚复用设置
 * 因为 PA8/PB13，默认并不是 TIM1 的功能脚，必须开启复用，PA8/PB13 才能用作 TIM1 的功能
```

```
 */
#define ATIM_TIMX_CPLM_CHY_GPIO_AF           GPIO_AF1_TIM1

/* 互补输出使用的定时器 */
#define ATIM_TIMX_CPLM                        TIM1
#define ATIM_TIMX_CPLM_CHY                    TIM_CHANNEL_1
#define ATIM_TIMX_CPLM_CLK_ENABLE()
        do{ __HAL_RCC_TIM1_CLK_ENABLE(); }while(0)    /* TIM1 时钟使能 */
```

可以把上面的宏定义分成两部分，第一部分是定时器 1 输出通道 1 和互补通道 1 对应的 IO 口的宏定义。第二部分则是定时器 1 相应宏定义。

下面看 dcmotor_tim.c 的程序，首先是高级定时器互补输出初始化函数。

```
/**
 * @brief     高级定时器 TIMX 互补输出 初始化函数（使用 PWM 模式 1）
 * @note
 *            配置高级定时器 TIMX 互补输出，一路 OCy 一路 OCyN，并且可以设置死区时间
 *
 *            高级定时器的时钟来自 APB2，而 PCLK2 = 168Mhz，我们设置 PPRE2 不分频，因此
 *            高级定时器时钟 = 168Mhz
 *            定时器溢出时间计算方法：Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *            Ft=定时器工作频率，单位 : Mhz
 *
 * @param     arr: 自动重装值。
 * @param     psc: 时钟预分频数
 * @retval    无
 */

void atim_timx_cplm_pwm_init(uint16_t arr, uint16_t psc)
{
    TIM_OC_InitTypeDef sConfigOC ;
    TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig;

    g_atimx_cplm_pwm_handle.Instance = ATIM_TIMX_CPLM;      /* 定时器 x */
    g_atimx_cplm_pwm_handle.Init.Prescaler = psc;           /* 定时器预分频系数 */
    g_atimx_cplm_pwm_handle.Init.CounterMode = TIM_COUNTERMODE_UP;/* 向上计数 */
    g_atimx_cplm_pwm_handle.Init.Period = arr;              /* 自动重装载值 */
    g_atimx_cplm_pwm_handle.Init.RepetitionCounter = 0;     /* 不重复计数 */
    g_atimx_cplm_pwm_handle.Init.AutoReloadPreload =
                TIM_AUTORELOAD_PRELOAD_ENABLE;          /* 使能影子寄存器 */
    HAL_TIM_PWM_Init(&g_atimx_cplm_pwm_handle) ;

    /* 设置 PWM 输出 */
    sConfigOC.OCMode = TIM_OCMODE_PWM1;                 /* PWM 模式 1 */
    sConfigOC.Pulse = 0;                                /* 比较值为 0 */
    sConfigOC.OCPolarity = TIM_OCPOLARITY_LOW;          /* OCy 低电平有效 */
    sConfigOC.OCNPolarity = TIM_OCNPOLARITY_LOW;        /* OCyN 低电平有效 */
    sConfigOC.OCFastMode = TIM_OCFAST_ENABLE;           /* 使用快速模式 */
    sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;      /* 主通道的空闲状态 */
    sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;    /* 互补通道的空闲状态 */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_cplm_pwm_handle, &sConfigOC,
                        ATIM_TIMX_CPLM_CHY);    /* 配置后默认清 CCER 的互补输出位 */

    /* 设置死区参数，开启死区中断 */
    sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_ENABLE;   /* OSSR 设置为 1 */
    sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE; /* OSSI 设置为 0 */
    /* 上电只能写一次，需要更新死区时间时只能用此值 */
    sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
    sBreakDeadTimeConfig.DeadTime = 0X0F;                     /* 死区时间 */
    /* BKE = 0，关闭 BKIN 检测 */
    sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
    /* BKP = 1，BKIN 低电平有效 */
```

```
    sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_LOW;
    /* 使能 AOE 位，允许刹车后自动恢复输出 */
    sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
    HAL_TIMEx_ConfigBreakDeadTime(&g_atimx_cplm_pwm_handle,
                                  &sBreakDeadTimeConfig);
}
```

HAL_TIM_PWM_Init 初始化 TIM1 并设置 TIM1 的 ARR 和 PSC 等参数， 然后通过调用函数 HAL_TIM_ConfigChannel 设置 TIM1_CH1 的 PWM 模式以及比较值等参数，最后通过调用函数 HAL_TIMEx_ConfigBreakDeadTime 来配置死区功能的参数。死区时间需要根据驱动电路的开关延时特性来设置，本教程所使用的驱动板设置死区时间为 0X0F 左右即可。

注意：我们设置了两个通道在无效状态时的输出为高电平，也就是说，当某个通道关闭 PWM 输出的时候，该通道会一直输出高电平。在电机控制的时候，我们只需要开启其中一个通道的 PWM 输出，让另一个通道处于无效状态（一直输出高电平），这样即可控制 PWM 的输出以实现电机的调速了。

本实验我们使用 PWM 的 MSP 初始化回调函数 HAL_TIM_PWM_MspInit 来存放时钟、GPIO 的初始化代码，其定义如下：

```
/**
 * @brief       定时器底层驱动，时钟使能，引脚配置
                此函数会被 HAL_TIM_PWM_Init()调用
 * @param       htim:定时器句柄
 * @retval      无
 */
void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == ATIM_TIMX_CPLM)
    {
        GPIO_InitTypeDef gpio_init_struct;

        ATIM_TIMX_CPLM_CHY_GPIO_CLK_ENABLE();    /* 通道 X 对应 IO 口时钟使能 */
        ATIM_TIMX_CPLM_CHYN_GPIO_CLK_ENABLE();   /* 互补通道对应 IO 口时钟使能 */
        ATIM_TIMX_CPLM_CLK_ENABLE();             /* 定时器 x 时钟使能 */

        /* 配置 PWM 主通道引脚 */
        gpio_init_struct.Pin = ATIM_TIMX_CPLM_CHY_GPIO_PIN;
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;
        gpio_init_struct.Pull = GPIO_NOPULL;
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH ;
        gpio_init_struct.Alternate = ATIM_TIMX_CPLM_CHY_GPIO_AF;  /* 端口复用 */
        HAL_GPIO_Init(ATIM_TIMX_CPLM_CHY_GPIO_PORT, &gpio_init_struct);

        /* 配置 PWM 互补通道引脚 */
        gpio_init_struct.Pin = ATIM_TIMX_CPLM_CHYN_GPIO_PIN;
        HAL_GPIO_Init(ATIM_TIMX_CPLM_CHYN_GPIO_PORT, &gpio_init_struct);
    }
}
```

该函数首先判断定时器寄存器基地址，符合条件后，开启对应的 GPIO 时钟和定时器时钟，然后初始化 GPIO。上面是使用 HAL 库标准的做法，我们亦可把 HAL_TIM_PWM_MspInit 函数里面的代码直接放到 gtim_timx_pwm_chy_init 函数里。这样做的好处是当一个项目中用到多个定时器时，代码的移植性、可读性好，方便管理。

介绍完了直流有刷电机的定时器驱动源码，接着我们来看电机控制相关源码，包括两个文件:dc_motor.c 和 dc_motor.h。

首先看 dc_motor.h 文件的几个宏定义：

```
/*
 * 停止引脚操作宏定义
 * 此引脚控制 H 桥是否生效以达到开启和关闭电机的效果
 */
#define SHUTDOWN1_Pin               GPIO_PIN_10
#define SHUTDOWN1_GPIO_Port         GPIOF
```

```
#define SHUTDOWN2_Pin                GPIO_PIN_2
#define SHUTDOWN2_GPIO_Port          GPIOF
#define SHUTDOWN_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)   /* PF 口时钟使能 */

/* 电机停止引脚定义 这里默认是接口 1 */
#define ENABLE_MOTOR
        HAL_GPIO_WritePin(SHUTDOWN1_GPIO_Port,SHUTDOWN1_Pin,GPIO_PIN_SET)
#define DISABLE_MOTOR
        HAL_GPIO_WritePin(SHUTDOWN1_GPIO_Port,SHUTDOWN1_Pin,GPIO_PIN_RESET)
```

这部分内容是 SD 刹车引脚相关的 IO 口的宏定义，我们默认使用的是接口 1。

下面看 dc_motor.c 的程序，首先是电机初始化函数。

```
/**
 * @brief       电机初始化
 * @param       无
 * @retval      无
 */
void dcmotor_init(void)
{
    SHUTDOWN_GPIO_CLK_ENABLE();
    GPIO_InitTypeDef gpio_init_struct = {0};

    /* SD 引脚设置，设置为推挽输出 */
    gpio_init_struct.Pin = SHUTDOWN_Pin|SHUTDOWN2_Pin;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull = GPIO_NOPULL;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(SHUTDOWN_GPIO_Port, &gpio_init_struct);

    /* SD 拉低，关闭输出 */
    HAL_GPIO_WritePin(GPIOF, SHUTDOWN2_Pin|SHUTDOWN_Pin, GPIO_PIN_RESET);
}
```

电机初始化函数主要就是对 SD 引脚进行初始化，并将其拉低，让电机初始化之后默认不启动。大家也可以根据实际的需求来设置电机的初始状态。

接着看电机开启和电机停止函数：

```
/**
 * @brief       电机开启
 * @param       无
 * @retval      无
 */
void dcmotor_start(void)
{
    ENABLE_MOTOR;            /* 拉高 SD 引脚，开启电机 */
}

/**
 * @brief       电机停止
 * @param       无
 * @retval      无
 */
void dcmotor_stop(void)
{
    /* 关闭主通道输出 */
    HAL_TIM_PWM_Stop(&g_atimx_cplm_pwm_handle, TIM_CHANNEL_1);
    /* 关闭互补通道输出 */
    HAL_TIMEx_PWMN_Stop(&g_atimx_cplm_pwm_handle, TIM_CHANNEL_1);
    DISABLE_MOTOR;           /* 拉低 SD 引脚，停止电机 */
}
```

电机开启函数主要是将 SD 引脚拉高，让电机处于允许启动状态（此时电机还不一定会启动）；电机停止函数中，先关闭两个通道的输出，再拉低 SD 引脚，停止电机。

接下来看直流有刷电机的速度、方向设置以及控制函数：

```
/**
 * @brief       电机速度设置
 * @param       para:比较寄存器值
 * @retval      无
 */
void dcmotor_speed(uint16_t para)
{
    if (para < (__HAL_TIM_GetAutoreload(&g_atimx_cplm_pwm_handle) - 0x0F))
    {
        __HAL_TIM_SetCompare(&g_atimx_cplm_pwm_handle, TIM_CHANNEL_1, para);
    }
}

/**
 * @brief       电机旋转方向设置
 * @param       para:方向 0 正转，1 反转
 * @note        以电机正面，顺时针方向旋转为正转
 * @retval      无
 */
void dcmotor_dir(uint8_t para)
{
    HAL_TIM_PWM_Stop(&g_atimx_cplm_pwm_handle, TIM_CHANNEL_1);
    HAL_TIMEx_PWMN_Stop(&g_atimx_cplm_pwm_handle, TIM_CHANNEL_1);

    if (para == 0)          /* 正转 */
    {
        HAL_TIM_PWM_Start(&g_atimx_cplm_pwm_handle, TIM_CHANNEL_1)
    }
    else if (para == 1)     /* 反转 */
    {
        HAL_TIMEx_PWMN_Start(&g_atimx_cplm_pwm_handle, TIM_CHANNEL_1);
    }
}

/**
 * @brief       电机控制
 * @param       para: pwm 比较值 ,正数电机为正转，负数为反转
 * @note        根据传入的参数控制电机的转向和速度
 * @retval      无
 */
void motor_pwm_set(float para)
{
    int val = (int)para;

    if (val >= 0)
    {
        dcmotor_dir(0);       /* 正转 */
        dcmotor_speed(val);
    }
    else
    {
        dcmotor_dir(1);       /* 反转 */
        dcmotor_speed(-val);
    }
}
```

① 电机速度设置函数：先判断用户设置的比较值大小是否超出限制范围，如果没有超出限制，再调用__HAL_TIM_SetCompare 函数设置定时器比较值的大小，进而控制电机的速度。

② 电机旋转方向设置函数：先关闭两个通道的输出，然后判断用户设置的电机方向，如果是正转，就调用 HAL_TIM_PWM_Start 来控制主通道输出 PWM，互补通道保持无效状态（输出高电平）；如果是反转，就调用 HAL_TIMEx_PWMN_Star 来控制互补通道输出 PWM，主通道保持无效状态（输出高电平）。

③ 电机控制函数：这个函数综合了电机的速度和方向设置功能，我们可以通过入口参数

传入比较值，函数会根据传入参数的正负来设置电机方向，正数则正转，负数则反转，数值的绝对值大小决定了电机的转速。大家需要注意，比较值本身是没有负数的，在实际的应用中，我们会以一个比较值的变量作为入口参数。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key,t;
    int32_t motor_pwm = 0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    atim_timx_cplm_pwm_init(8400 - 1, 0);       /* 168Mhz 的计数频率 */
    dcmotor_init();                             /* 初始化电机 */

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10, 10, 200, 16, 16, "DcMotor Test", g_point_color);
    lcd_show_string(10, 30, 200, 16, 16, "KEY0:Start forward", g_point_color);
    lcd_show_string(10, 50, 200, 16, 16, "KEY1:Start backward", g_point_color);
    lcd_show_string(10, 70, 200, 16, 16, "KEY2:Stop", g_point_color);

    printf("KEY0：增加比较值，KEY1：减小比较值，KEY2：停止电机\r\n");

    while (1)
    {
        key = key_scan(0);                      /* 按键扫描 */
        if(key == KEY0_PRES)                    /* 当 key0 按下 */
        {
            /* 因为不同的电机最小启动电压不同，可能在第一次增加的时候电机还不能转起来 */
            motor_pwm += 400;
            if (motor_pwm == 0)
            {
                dcmotor_stop();                 /* 停止则立刻响应 */
                motor_pwm = 0;
            }
            else
            {
                dcmotor_start();                /* 开启电机 */
                if (motor_pwm >= 8400)          /* 限速 */
                {
                    motor_pwm = 8400;
                }
            }
            motor_pwm_set(motor_pwm);           /* 设置电机 PWM 的占空比 */
        }

        else if(key == KEY1_PRES)               /* 当 key1 按下 */
        {
            motor_pwm -= 400;
            if (motor_pwm == 0)
            {
                dcmotor_stop();                 /* 停止则立刻响应 */
                motor_pwm = 0;
            }
            else
            {
                dcmotor_start();                /* 开启电机 */
```

```
                if (motor_pwm <= -8400)         /* 限速 */
                {
                    motor_pwm = -8400;
                }
            }
            motor_pwm_set(motor_pwm);           /* 设置电机 PWM 的占空比 */
        }

        else if(key == KEY2_PRES)               /* 当 key2 按下 */
        {
            dcmotor_stop();                     /* 关闭电机 */
            motor_pwm = 0;
            motor_pwm_set(motor_pwm);           /* 设置电机 PWM 的占空比 */
        }

        delay_ms(10);
        t++;
        if(t % 20 == 0)
        {
            LED0_TOGGLE();                      /* LED0(红灯) 翻转 */
        }
    }
}
```

先看 atim_timx_cplm_pwm_init(8400 - 1, 0)这个语句，我们设置自动重载寄存器的值为 8400-1，预分频系数为 0，可得计数器的计数频率为 168MHZ， PWM 的周期为 50us。接着就是调用电机初始化函数，然后在 LCD 显示一些按键功能提示信息。

进入到 while 循环之后，我们不断地扫描按键，如果 KEY0 按下，就增大比较值变量 motor_pwm，如果 KEY1 按下，就减小比较值变量 motor_pwm，然后通过调用 motor_pwm_set 函数，传入比较值变量 motor_pwm，对电机的速度以及方向进行控制。当 KEY2 按下，电机将马上停止。

注意：本实验默认的比较值步进量为 400，因为不同电机的最小启动电压不一样，有的电机在第一次按下按键之后可能还不能正常启动。

下面我们下载到开发板验证一下。

5.4.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 及串口输出一些按键功能提示信息，当我们按下 KEY0，比较值变量 motor_pwm 将增大；按下 KEY1，比较值变量 motor_pwm 将减小；按下 KEY2，电机将停止。比较值变量 motor_pwm 为正数时，电机正转，反之电机反转，其绝对值越大，电机的速度越快。大家可自行下载验证看看现象。

第 6 章 直流有刷驱动板电压温度电流采集

本章我们主要来学习直流有刷驱动板的电压、温度、电流采集原理，并实现相关的采集实验。

本章分为如下几个小节：

6.1 电压温度电流采集原理
6.2 硬件设计
6.3 程序设计
6.4 下载验证

6.1 电压温度电流采集原理

1）电压采集原理

由于直流有刷电机驱动板的电源电压远超 STM32 内部 ADC 所能采集的范围，我们并不能直接使用 ADC 进行电压采集，而是需要使用一些硬件电路对电源电压进行处理，使其减小到 ADC 采集范围，具体原理图如图 6.1.1 所示：

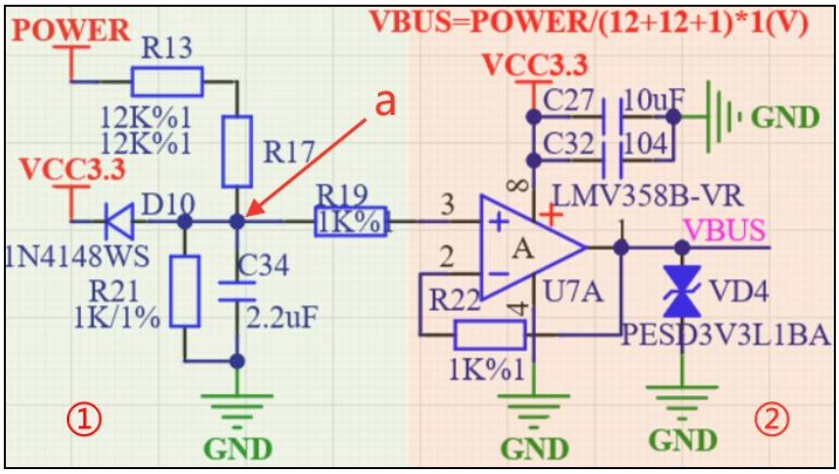


图 6.1.1 电压采集电路

图 6.1.1 中电压采集电路的实现原理如下：

① 电阻分压，利用三个电阻（R13、R17、R21）对电源电压 POWER 进行分压，可得 a 点的电压 Va = POWER /（12+12+1）；

② 电压跟随电路，其作用就是让 VBUS = Va，即 VBUS = POWER /（12+12+1）= POWER/25；

通过这两部分电路的处理，我们就可以直接使用 ADC 采集 VBUS 的电压，再根据公式：POWER= 25 * VBUS，即可算出电源电压。

2）电流采集原理

由于 STM32 内部 ADC 并不能直接对电流进行采集，所以我们需要先把电流的信号转换为电压的信号，具体的原理图如图 6.1.2 所示：

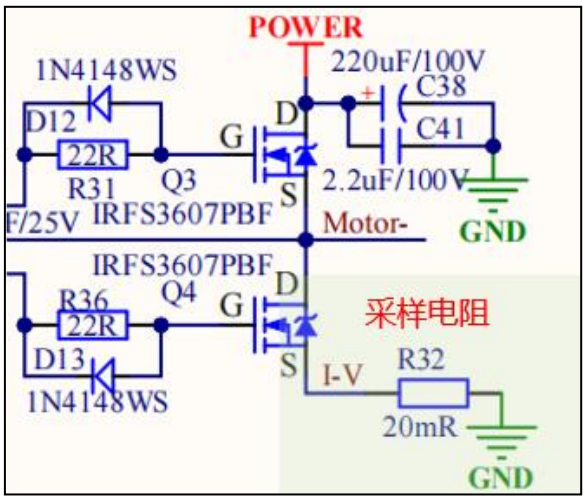


图 6.1.2 采样电路

图 6.1.2 中，我们在 H 桥的回路中串接一个 20mR 的采样电阻，这样就可以得到一个采样电压 I-V = 0.02R * 实际电流 I，但是这个采样电压太小了，直接用 ADC 进行采集的话偏差较大，我们需要对它进行差分放大，具体的原理图如图 6.1.3 所示：

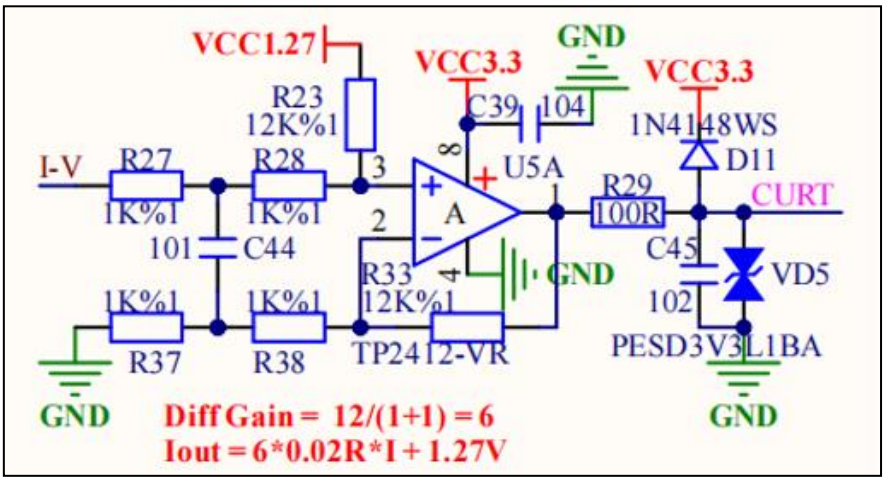


图 6.1.3 电流采集电路

图 6.1.3 中，差分放大倍数= 12 /（1+1）=6 倍，参考电压为 1.27V，由此可得输出电压 Iout（即 CURT 点的电压）=6 * 0.02R * 实际电流 I + 1.27V，我们只需要采集 Iout 的电压，再根据公式：实际电流 I=（输出电压 Iout-1.27）/0.12A，即可算出实际电流。

3）温度采集原理

本实验中我们采集的是驱动板的温度，同样需要用到 ADC 进行采集，具体的原理图如图 6.1.4 所示：

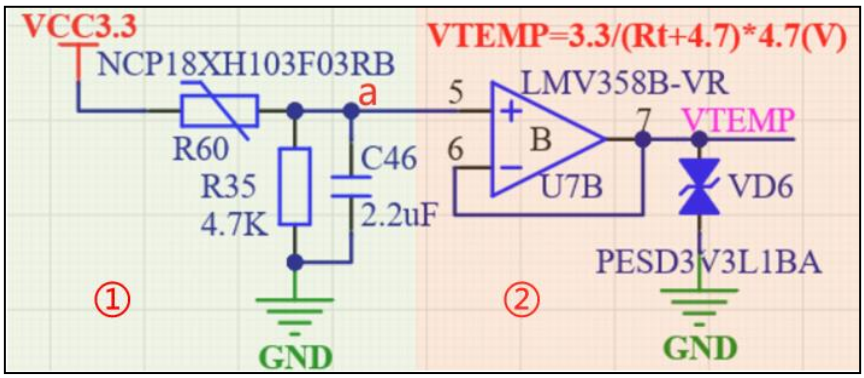


图 6.1.4 温度采集电路

图 6.1.4 中，我们利用 NCP18XH103F03RB 热敏电阻对温度进行检测，该热敏电阻是负温

度系数电阻，温度越高，其电阻值越低。温度采集电路的原理实现如下：

① 电阻分压，利用热敏电阻和 R35 电阻（4.7K）对 VCC3.3 进行分压，可得 a 点的电压 Va = 4.7 * 3.3 V /（4.7 + 热敏电阻阻值 Rt）；

② 电压跟随电路，其作用就是让 VTEMP = Va，即 VTEMP = 4.7 * 3.3 V /（4.7 + 热敏电阻阻值 Rt）；

通过这两部分电路的处理，我们就可以直接使用 ADC 采集 VTEMP 的电压，再通过公式即可算出热敏电阻阻值 Rt。最后我们再根据厂家给出的公式，即可算出该阻值所对应的实际温度值，具体的换算公式将会在代码注释中介绍。

注意：我们所使用的温度换算公式以及相关参数均来自《NCP18XH103F03RB 热敏电阻.pdf》这个数据手册，路径：A 盘→7，硬件资料→2，芯片资料 文件夹下可以找到。

6.2 硬件设计

1. 例程功能

1、本实验以电机开发板的直流有刷电机驱动接口 1 为例，基于基础驱动代码功能，加入多通道 ADC 采集，利用 ADC1_CH9（由 PB1 复用）、ADC1_CH0（由 PA0 复用）和 ADC1_CH8（由 PB0 复用）三个通道，分别检测电压、温度和电流采集电路对应的输出电压，然后进一步算出电压、电流和温度值。

2、当按键 0 按下，就增大 PWM 的比较值变量，当按键 1 按下，就减小 PWM 的比较值变量。比较值变量的绝对值越大，电机速度越快。当比较值变量为正数时电机正转，反之电机反转，按下按键 2 则马上停止电机。

3、屏幕显示按键功能信息。

4、串口 1 打印驱动板电压、电流、温度信息。

5、LED0 闪烁指示程序运行。

2. 硬件资源

1）LED 灯
   LED0 - PE0
2）独立按键
   KEY0 - PE2
   KEY1 - PE3
   KEY2 - PE4
3）定时器 1
   TIM1 正常输出通道 PA8
   TIM1 互补输出通道 PB13
4）SD（刹车）信号输出 PF10
5）ADC1
   ADC1 通道 9 PB1（电压）
   ADC1 通道 0 PA0（温度）
   ADC1 通道 8 PB0（电流）
6）串口 1
   USART1_TX PB6（发送）
   USART1_RX PB7（接收）

3. 原理图

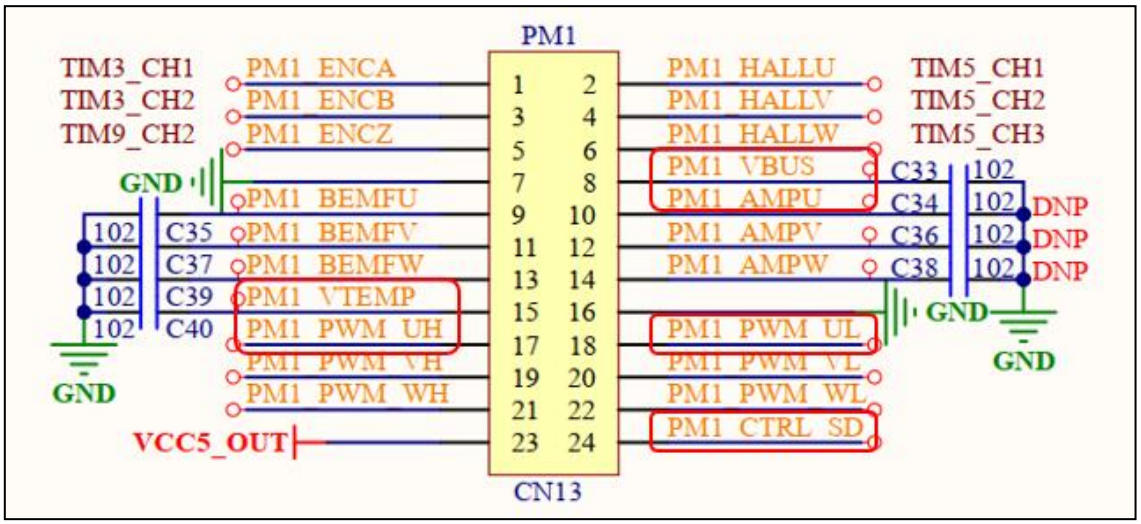


图 6.2.1 直流有刷电机接口原理图

图 6.2.1 就是我们 DMF407 电机开发板的直流有刷电机接口 1 原理图，本实验我们只需要用到了 PM1_PWM_UH（PA8）、PM1_PWM_UL（PB13）、PM1_CTRL_SD（PF10）、PM1_VBUS（PB1）、PM1_AMPU（PB0）和 PM1_VTEMP（PA0）这 6 个引脚，其中 PA8 和 PB13 用于输出所需的 PWM 控制信号，PF10 用于输出刹车信号，PB1、PA0 和 PB0 分别用于检测电源电压、温度和电流采集电路对应的输出电压。

本实验的硬件接线部分和基础驱动实验基本一致，这里需要用一根 USB-C 数据线连接串口 1 到电脑，才可以在串口调试助手查看数据，接线如图 6.2.2 所示：

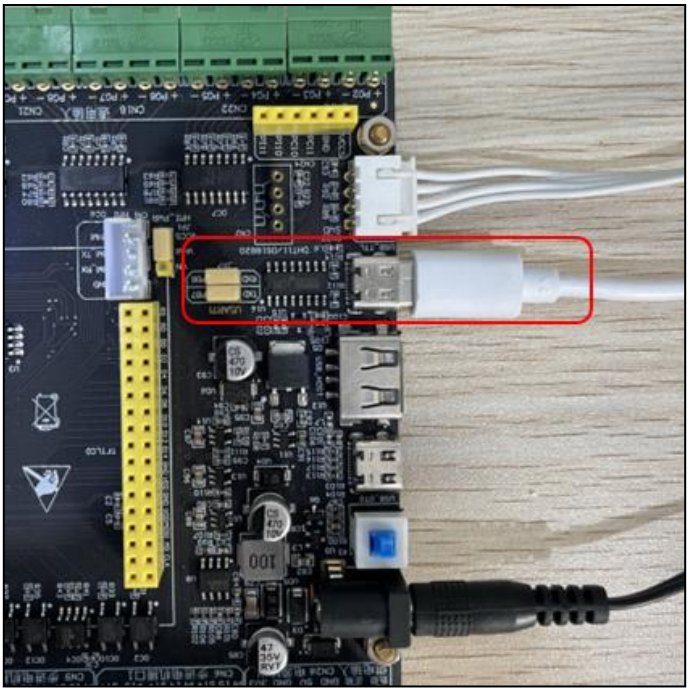


图 6.2.2 串口 1 连接电脑示意图

6.3 程序设计

本实验是基于基础驱动实验实现的，其中所用到的 HAL 库驱动请回顾高级定时器以及 ADC 章节的内容。下面介绍一下电压电流温度采集相关的 ADC、DMA 配置步骤。

ADC、DMA 配置步骤

1）开启 ADCx 和输入通道的 GPIO 时钟，配置该 IO 口的模拟输入功能。

首先开启 ADCx 的时钟，然后配置 GPIO 为模拟输入模式。本实验我们默认用到 ADC1 通道 9、0、8，对应 IO 是 PB1、PA0 和 PB0，它们的时钟开启方法如下：

```
__HAL_RCC_ADC1_CLK_ENABLE ();           /* 使能 ADC1 时钟 */
__HAL_RCC_GPIOA_CLK_ENABLE ();          /* 开启 GPIOA 时钟 */
__HAL_RCC_GPIOB_CLK_ENABLE ();          /* 开启 GPIOB 时钟 */
```

IO 口模拟输入功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 ADCx，设置 ADCx 的参数。

通过 HAL_ADC_Init 函数来设置 ADCx 时钟分频系数、分辨率、模式、扫描方式、对齐方式等信息。

注意：该函数会调用：HAL_ADC_MspInit 函数，但是为了方便代码管理和移植，我们就直接在 adc_nch_dma_init 函数中，使能 ADC 时钟和 GPIO 时钟，初始化通道对应 IO 引脚等。

3）配置 ADC 通道

通过 HAL_ADC_ConfigChannel 函数来配置 ADC 的通道，根据需求设置通道、序列、采样时间，等等。

4）配置 DMA

首先开启 DMA 的时钟，然后通过 HAL_DMA_Init 函数配置 DMA，包括：数据流、通道、外设地址、存储器地址、传输方向，等等。

HAL 库为了处理各类外设的 DMA 请求，在调用相关函数之前，需要调用一个宏定义标识符，来连接 DMA 和外设句柄。这个宏定义为__HAL_LINKDMA。

5）使能 DMA 对应数据流中断，配置 DMA 中断优先级，使能 ADC 和 DMA

通过 HAL_NVIC_EnableIRQ 函数使能 DMA 数据流中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

通过 HAL_ADC_Start_DMA 函数开启 ADC 转换和 DMA 传输。

6）编写中断服务函数

DMA 的每个数据流都有一个对应的中断服务函数，比如 DMA2_Stream4 的中断服务函数为 DMA2_Stream4_IRQHandler。如果使用 HAL 库的中断回调机制，可以在相关中断服务函数中直接调用 DMA 中断公共处理函数 HAL_DMA_IRQHandler，然后我们直接重定义相关的中断回调函数来编写中断程序即可。

6.3.1 程序流程图

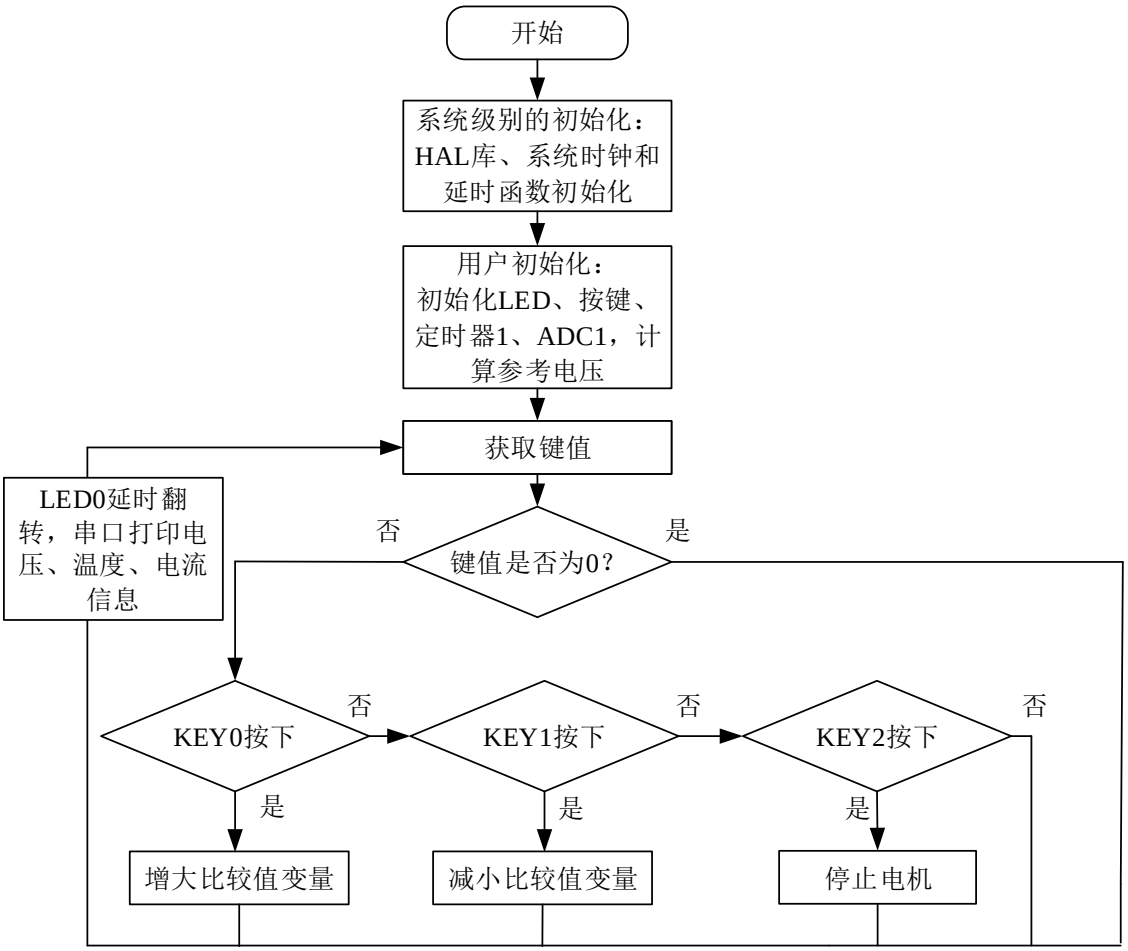


图 6.3.1.1 电压、温度、电流采集程序流程图

6.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。定时器驱动相关源码与基础驱动章节一致，这里不再赘述。我们先看 ADC 相关的程序，ADC 驱动源码包括两个文件：adc.c 和 adc.h。

首先看 adc.h 头文件的几个宏定义：

```
/* ADC 及引脚 定义 */

#define ADC_ADCX_CH0_GPIO_PORT          GPIOB
#define ADC_ADCX_CH0_GPIO_PIN           GPIO_PIN_1
#define ADC_ADCX_CH0_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)      /* PB 口时钟使能 */

#define ADC_ADCX_CH1_GPIO_PORT          GPIOA
#define ADC_ADCX_CH1_GPIO_PIN           GPIO_PIN_0
#define ADC_ADCX_CH1_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)      /* PA 口时钟使能 */

#define ADC_ADCX_CH2_GPIO_PORT          GPIOB
#define ADC_ADCX_CH2_GPIO_PIN           GPIO_PIN_0
#define ADC_ADCX_CH2_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)      /* PB 口时钟使能 */

#define ADC_ADCX                ADC1
#define ADC_ADCX_CH0            ADC_CHANNEL_9            /* 电压测量通道 */
```

```
#define ADC_ADCX_CH1            ADC_CHANNEL_0               /* 温度测量通道 */
#define ADC_ADCX_CH2            ADC_CHANNEL_8               /* 电流测量通道 */

#define ADC_ADCX_CHY_CLK_ENABLE()
        do{ __HAL_RCC_ADC1_CLK_ENABLE(); }while(0)          /* ADC1 时钟使能 */

#define ADC_CH_NUM              3                           /* 需要转换的通道数目 */
#define ADC_COLL                1000                        /* 单采集次数 */
#define ADC_SUM                 ADC_CH_NUM * ADC_COLL       /* 总采集次数 */

/*  DMA 传输相关 定义 */
#define ADC_ADCX_DMASx              DMA2_Stream4            /* 数据流 4 */
#define ADC_ADCX_DMASx_Chanel       DMA_CHANNEL_0           /* 通道 0 */
#define ADC_ADCX_DMASx_IRQn         DMA2_Stream4_IRQn
#define ADC_ADCX_DMASx_IRQHandler   DMA2_Stream4_IRQHandler
```

可以把上面的宏定义分成两部分，第一部分是 ADC 通道的 IO 口以及采集次数的宏定义。第二部分则是 ADC 相关的 DMA 宏定义。

下面看 adc.c 的程序，首先是 ADC 初始化函数。

```
/**
 * @brief       ADC 初始化函数
 * @param       无
 * @retval      无
 */
void adc_init(void)
{
    ADC_ChannelConfTypeDef sConfig = {0};

    g_adc_nch_dma_handle.Instance = ADC_ADCX;                   /* ADCx */
    /* 4 分频，ADCCLK = PCLK2/4 = 84/4 = 21Mhz */
    g_adc_nch_dma_handle.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;

    g_adc_nch_dma_handle.Init.Resolution = ADC_RESOLUTION_12B; /* 12 位模式 */
    g_adc_nch_dma_handle.Init.ScanConvMode = ENABLE;            /* 开启扫描模式 */
    g_adc_nch_dma_handle.Init.ContinuousConvMode = ENABLE;      /* 连续转换模式 */
    /* 禁止不连续采样模式 */
    g_adc_nch_dma_handle.Init.DiscontinuousConvMode = DISABLE;
    g_adc_nch_dma_handle.Init.ExternalTrigConvEdge =
                        ADC_EXTERNALTRIGCONVEDGE_NONE;          /* 不用外部触发 */
    /* 软件触发 */
    g_adc_nch_dma_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    g_adc_nch_dma_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT; /* 右对齐 */
    /* 使用转换通道数，需根据实际转换通道去设置 */
    g_adc_nch_dma_handle.Init.NbrOfConversion = ADC_CH_NUM;
    /* 开启 DMA 连续转换 */
    g_adc_nch_dma_handle.Init.DMAContinuousRequests = ENABLE;
    g_adc_nch_dma_handle.Init.EOCSelection = ADC_EOC_SEQ_CONV;
    HAL_ADC_Init(&g_adc_nch_dma_handle);

    /* 配置使用的 ADC 通道，采样序列里的第几个转换，增加或者减少通道需要修改这部分 */
    sConfig.Channel = ADC_ADCX_CH0;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);

    sConfig.Channel = ADC_ADCX_CH1;
    sConfig.Rank = 2;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);

    sConfig.Channel = ADC_ADCX_CH2;
    sConfig.Rank = 3;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);
}
```

ADC 初始化函数通过 HAL_ADC_Init 和 HAL_ADC_ConfigChannel 配置 ADC1 的相关参数，这一部分的原理和多通道 ADC 采集（使用 DMA）实验是一样的，这里不再赘述。

注意：代码中所使用的 ADC 通道是通过宏定义传入的，其中的 ADC_ADCX_CH0、ADC_ADCX_CH2、ADC_ADCX_CH2 实际上分别对应的是 ADC1_CH9、ADC1_CH0 和 ADC1_CH8。

接下来看 ADC 相关的 DMA 传输初始化函数。

```
/**
 * @brief       ADC DMA 传输 初始化函数
 * @note        本函数还是使用 adc_init 对 ADC 进行大部分配置,有差异的地方再单独配置
 * @param       par         : 外设地址
 * @param       mar         : 存储器地址
 * @retval      无
 */
void adc_nch_dma_init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    ADC_ADCX_CHY_CLK_ENABLE();                  /* 开启 ADCx 时钟 */
    ADC_ADCX_CH0_GPIO_CLK_ENABLE();             /* 开启 GPIO 时钟 */
    ADC_ADCX_CH1_GPIO_CLK_ENABLE();
    ADC_ADCX_CH2_GPIO_CLK_ENABLE();

    /* AD 采集引脚模式设置,模拟输入 */
    GPIO_InitStruct.Pin = ADC_ADCX_CH0_GPIO_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(ADC_ADCX_CH0_GPIO_PORT, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = ADC_ADCX_CH1_GPIO_PIN;
    HAL_GPIO_Init(ADC_ADCX_CH1_GPIO_PORT, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = ADC_ADCX_CH2_GPIO_PIN;
    HAL_GPIO_Init(ADC_ADCX_CH2_GPIO_PORT, &GPIO_InitStruct);

    adc_init();                                 /* 初始化 ADC */
    /* 大于 DMA1_Channel7, 则为 DMA2 的通道 */
    if ((uint32_t)ADC_ADCX_DMASx > (uint32_t)DMA2)
    {
        __HAL_RCC_DMA2_CLK_ENABLE();                        /* DMA2 时钟使能 */
    }
    else
    {
        __HAL_RCC_DMA1_CLK_ENABLE();                        /* DMA1 时钟使能 */
    }

    /* DMA 配置 */
    g_dma_nch_adc_handle.Instance = ADC_ADCX_DMASx;         /* 设置 DMA 通道 */
    g_dma_nch_adc_handle.Init.Channel = DMA_CHANNEL_0;      /* DMA 通道 x */
    /* DIR = 1 ,  外设到存储器模式 */
    g_dma_nch_adc_handle.Init.Direction = DMA_PERIPH_TO_MEMORY;
    g_dma_nch_adc_handle.Init.PeriphInc = DMA_PINC_DISABLE;/* 外设非增量模式 */
    g_dma_nch_adc_handle.Init.MemInc = DMA_MINC_ENABLE;     /* 存储器增量模式 */
    /* 外设数据长度:16 位 */
    g_dma_nch_adc_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    /* 存储器数据长度:16 位 */
    g_dma_nch_adc_handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    g_dma_nch_adc_handle.Init.Mode = DMA_CIRCULAR;          /* 外设流控模式 */
    g_dma_nch_adc_handle.Init.Priority = DMA_PRIORITY_MEDIUM; /* 中等优先级 */
    HAL_DMA_Init(&g_dma_nch_adc_handle);

    /* 把 ADC 和 DMA 关联，用 DMA 传输 ADC 数据 */
```

```
    __HAL_LINKDMA(&g_adc_nch_dma_handle,DMA_Handle,g_dma_nch_adc_handle);
    /* ADC DMA 中断配置 */
    HAL_NVIC_SetPriority(ADC_ADCX_DMASx_IRQn, 2, 1);
    HAL_NVIC_EnableIRQ(ADC_ADCX_DMASx_IRQn);
    /* 开启 ADC 的 DMA 传输 */
    HAL_ADC_Start_DMA(&g_adc_nch_dma_handle,
                      (uint32_t *)g_adc_value,ADC_CH_NUM * ADC_COLL);
}
```

该函数首先配置 ADC 采集相关 IO，然后调用 adc_init 函数对 ADC 进行初始化，接着开启 DMA 的时钟以及初始化相关的 DMA。最后把 ADC 和 DMA 关联，使能 DMA 中断，并开启 ADC 和 DMA。

下面我们来看 ADC 采集中断服务回调函数。

```
/**
 * @brief       ADC 采集中断服务回调函数
 * @param       无
 * @retval      无
 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    if (hadc->Instance == ADC_ADCX)
    {
        HAL_ADC_Stop_DMA(&g_adc_nch_dma_handle);  /* 关闭 DMA 转换 */
        calc_adc_val(g_adc_val);                  /* 计算 ADC 的平均值 */
        HAL_ADC_Start_DMA(&g_adc_nch_dma_handle, (uint32_t *)&g_adc_value,
                          (uint32_t)(ADC_SUM));   /* 启动 DMA 转换 */
    }
}
```

在 ADC 中断回调函数中，先关闭 DMA 转换，然后调用 calc_adc_val 函数计算 ADC 平均值（滤波），calc_adc_val 函数我们后面再介绍。滤波后的 ADC 值存储到 g_adc_val 这个数组当中，我们将通过滤波后的 ADC 值来进行电压、温度以及电流的计算。计算完 ADC 平均值后，再次启动 DMA 转换。

接下来看电压、电流以及温度计算相关的程序，包括两个文件：dc_motor.c 和 dc_motor.h，我们这里只介绍本实验新增的内容。

首先看 dc_motor.h 头文件的几个宏定义：

```
/* 电流计算公式：
 * I=（最终输出电压-初始参考电压）/（6*0.02）A
 * ADC 值转换为电压值：电压=ADC 值*3.3/4096，这里电压单位为 V，我们换算成
   mV,4096/1000=4.096，后面就直接算出为 mA
 * 整合公式可以得出电流 I= （当前 ADC 值-初始参考 ADC 值）* （3.3 / 4.096 / 0.12）
 */
#define ADC2CURT    (float)(3.3f / 4.096f / 0.12f)

/* 电压计算公式：
 * V_POWER = V_BUS * 25
 * ADC 值转换为电压值：电压=ADC 值*3.3/4096
 * 整合公式可以得出电压 V_POWER= ADC 值 *（3.3f * 25 / 4096）
 */
#define ADC2VBUS    (float)(3.3f * 25 / 4096)
```

这里定义的是电流、电压计算相关的宏，其中涉及的公式是根据我们前面介绍的电流、电压采集电路原理得出的，当我们需要计算电流和电压的时候，直接调用相应的宏定义即可，具体的公式推导过程大家可以直接看代码的注释。

下面看 dc_motor.c 的程序，首先看 ADC 平均值计算函数。

```
/**
 * @brief       计算 ADC 的平均值（滤波）
 * @param       * p ：存放 ADC 值的指针地址
 * @note        此函数对电压、温度、电流对应的 ADC 值进行滤波，
 *              p[0]-p[2]对应的分别是电压、温度和电流
 * @retval      无
```

```
 */
void calc_adc_val(uint16_t * p)
{
    uint32_t temp[3] = {0,0,0};
    int i;
    for(i=0;i<ADC_COLL;i++)        /* 循环 ADC_COLL 次取值，累加 */
    {
        temp[0] += g_adc_value[0+i*ADC_CH_NUM];
        temp[1] += g_adc_value[1+i*ADC_CH_NUM];
        temp[2] += g_adc_value[2+i*ADC_CH_NUM];
    }
    temp[0] /= ADC_COLL;           /* 取平均值 */
    temp[1] /= ADC_COLL;
    temp[2] /= ADC_COLL;
    p[0] = temp[0];               /* 存入电压 ADC 通道平均值 */
    p[1] = temp[1];               /* 存入温度 ADC 通道平均值 */
    p[2] = temp[2];               /* 存入电流 ADC 通道平均值 */
}
```

这个函数非常简单，它的功能就是把电压、温度以及电流采集通道的 ADC 值分别累计 1000 次，然后取平均值（滤波），最后再把 ADC 平均值存入相应的数组元素中。

接下来看温度值计算函数，其定义如下：

```
/*
    Rt = Rp *exp(B*(1/T1-1/T2))

    Rt 是热敏电阻在 T1 温度下的阻值；
    Rp 是热敏电阻在 T2 常温下的标称阻值；
    exp 是 e 的 n 次方，e 是自然常数，就是自然对数的底数，近似等于 2.7182818；
    B 值是热敏电阻的重要参数，教程中用到的热敏电阻 B 值为 3380；
    这里 T1 和 T2 指的是开尔文温度，T2 是常温 25℃，即(273.15+25)K
    T1 就是所求的温度
*/
const float Rp = 10000.0f;           /* 10K */
const float T2 = (273.15f + 25.0f); /* T2 */
const float Bx = 3380.0f;            /* B */
const float Ka = 273.15f;

/**
 * @brief       计算温度值
 * @param       para: 温度采集对应 ADC 通道的值（已滤波）
 * @note        计算温度分为两步：
                1.根据 ADC 采集到的值计算当前对应的 Rt
                2.根据 Rt 计算对应的温度值
 * @retval      温度值
 */
float get_temp(uint16_t para)
{
    float Rt;
    float temp;

    /*
        第一步：
        Rt = 3.3 * 4700 / VTEMP - 4700 ,其中 VTEMP 就是温度检测通道采集回来的电压
        值,VTEMP = ADC 值* 3.3/4096
        由此我们可以计算出当前 Rt 的值：Rt = 3.3f * 4700.0f / (para * 3.3f /
        4096.0f ) - 4700.0f;
    */
    Rt = 3.3f * 4700.0f / (para * 3.3f / 4096.0f ) - 4700.0f;

    /*
        第二步：
        根据当前 Rt 的值来计算对应温度值：Rt = Rp *exp(B*(1/T1-1/T2))
    */
```

```
    temp = Rt / Rp;  /* 解出 exp(B*(1/T1-1/T2)) ，即 temp = exp(B*(1/T1-1/T2)) */
    temp = log(temp);        /* 解出 B*(1/T1-1/T2) ，即 temp = B*(1/T1-1/T2) */
    temp /= Bx;              /* 解出 1/T1-1/T2 ，即 temp = 1/T1-1/T2 */
    temp += (1.0f / T2);     /* 解出 1/T1 ，即 temp = 1/T1 */
    temp = 1.0f / (temp);    /* 解出 T1 ，即 temp = T1 */
    temp -= Ka;              /* 计算 T1 对应的摄氏度 */
    return temp;             /* 返回温度值 */
}
```

代码中首先定义了一些关于热敏电阻的参数，各参数的含义我们在代码注释中已经详细介绍，它们的值是根据实际的热敏电阻型号来确定的，详见《NCP18XH103F03RB 热敏电阻.pdf》这个数据手册，路径：A 盘→7，硬件资料→2，芯片资料 文件夹下可以找到。

温度计算函数的实现分为两个步骤：

第一步：根据采集到的 ADC 值计算出当前热敏电阻阻值 Rt，公式如下：

$$Rt = 3.3 * 4700 / VTEMP - 4700$$

公式中 VTEMP 是温度检测通道对应的电压值，结合 ADC 换算公式 VTEMP = ADC 值* 3.3/4096 可得 Rt= 3.3 * 4700/ (ADC 值 * 3.3 / 4096) - 4700。

第二步：根据当前 Rt 的值来计算对应温度值，公式如下：

$$Rt = Rp * \exp(B * (1/T1 - 1/T2))$$

公式中的 T1 就是所求的温度，它的默认单位是华氏度，我们最后还需要将其转换成摄氏度。温度计算函数的实现本质上就是这两个公式的代码体现，大家一定要跟着公式去理解代码。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key;
    uint16_t t;
    int32_t motor_pwm = 0;
    uint16_t init_adc_val;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    atim_timx_cplm_pwm_init(8400 - 1, 0);       /* 168Mhz 的计数频率 */
    dcmotor_init();                             /* 初始化电机 */
    adc_nch_dma_init();                         /* ADC DMA 传输初始化 */

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10, 10, 200, 16, 16, "DcMotor Test", g_point_color);
    lcd_show_string(10, 30, 200, 16, 16, "KEY0:Start forward", g_point_color);
    lcd_show_string(10, 50, 200, 16, 16, "KEY1:Start backward", g_point_color);
    lcd_show_string(10, 70, 200, 16, 16, "KEY2:Stop", g_point_color);

    delay_ms(20);

    /* init_adc_val 存储电流测量对应的参考电压 ADC 值，这里进行滤波 */
    init_adc_val = g_adc_val [2];               /* 取出第一次得到的值 */
    for(t=0;t<1000;t++)
    {
        init_adc_val += g_adc_val [2];          /* 现在的值和上一次存储的值相加 */
        init_adc_val /= 2;                      /* 取平均值 */
        delay_ms(1);
    }
```

```
    while (1)
    {
        key = key_scan(0);                          /* 按键扫描 */
        if(key == KEY0_PRES)                        /* 当 key0 按下 */
        {
            /* 因为不同的电机最小启动电压不同，可能在第一次增加的时候电机还不能转起来 */
            motor_pwm += 400;
            if (motor_pwm == 0)
            {
                dcmotor_stop();                     /* 停止则立刻响应 */
                motor_pwm = 0;
            }
            else
            {
                dcmotor_start();                    /* 开启电机 */
                if (motor_pwm >= 8400)              /* 限速 */
                {
                    motor_pwm = 8400;
                }
            }
            motor_pwm_set(motor_pwm);               /* 设置电机方向、转速 */
        }

        else if(key == KEY1_PRES)                   /* 当 key1 按下 */
        {
            motor_pwm -= 400;
            if (motor_pwm == 0)
            {
                dcmotor_stop();                     /* 停止则立刻响应 */
                motor_pwm = 0;
            }
            else
            {
                dcmotor_start();                    /* 开启电机 */
                if (motor_pwm <= -8400)             /* 限速 */
                {
                    motor_pwm = -8400;
                }
            }
            motor_pwm_set(motor_pwm);               /* 设置电机方向、转速 */
        }

        else if(key == KEY2_PRES)                   /* 当 key2 按下 */
        {
            LED1_TOGGLE();
            dcmotor_stop();                         /* 关闭电机 */
            motor_pwm = 0;
            motor_pwm_set(motor_pwm);               /* 设置电机方向、转速 */
        }

        delay_ms(10);
        t++;
        if(t % 20 == 0)
        {
            LED0_TOGGLE();                          /* LED0(红灯) 翻转 */
            printf("KEY0: 增加比较值，KEY1: 减小比较值，KEY2: 停止电机\r\n\r\n");
            printf("Valtage:%.1fV \r\n",g_adc_val[0]*ADC2VBUS);  /* 打印电压值 */
            printf("Temp:%.1fC \r\n",get_temp(g_adc_val[1]));    /* 打印温度值 */
            printf("Current:%.1fmA \r\n\r\n",abs(g_adc_val[2]-
                                        init_adc_val)*ADC2CURT);/* 打印电流值 */
        }
    }
}
```

初始化的内容和基础驱动章节基本相同，只是多了 adc_nch_dma_init()这个语句，对 ADC 以及 DMA 进行初始化。

注意：在电流采集电路的原理介绍中，我们有提到参考电压的值为 1.27V。在实际的应用中，为了获取更精确的参考值，我们不直接使用 1.27V 作为参考电压，而是采用以下方式获得参考电压：在进入 while 循环之前（此时电机没有启动），读取 1000 次电流采集通道对应的 ADC 值，然后取平均值，进而计算出参考电压。

在 while 循环里面，除了基础驱动的内容，我们还利用串口 1 来打印电压、温度以及电流的值，它们的计算过程如下：

数组元素 g_adc_val [0]~ g_adc_val [2]所存储的数据分别对应的是电压、温度、电流采集通道的 ADC 平均值，我们取出相应的 ADC 平均值，代入公式即可算出结果。值得注意的是，电流是没有负数的，因此需要调用 abs 函数对电流值取绝对值。

6.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 显示一些按键功能提示信息，当我们按下 KEY0，比较值变量 motor_pwm 将增大；按下 KEY1，比较值变量 motor_pwm 将减小；按下 KEY2，电机将停止。比较值变量 motor_pwm 为正数时，电机正转，反之电机反转，其绝对值越大，电机的速度越快。我们再打开串口调试助手，选择对应的串口端口，我这边是 COM3，可以看到串口打印的按键信息、电压、温度以及电流值，如图 6.4.1 所示：

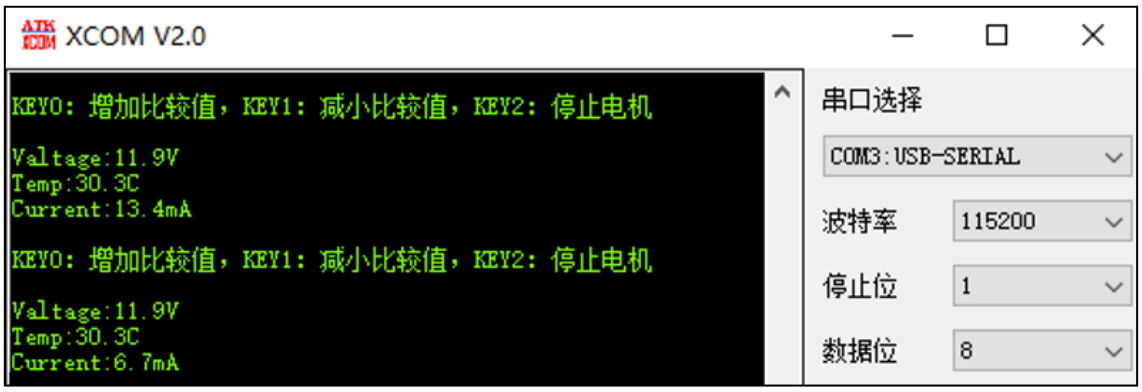


图 6.4.1 串口打印按键信息、电压、温度以及电流值

第 7 章 直流有刷电机编码器测速

本章我们主要来学习编码器测速的原理，并实现直流有刷电机的速度检测。在实际的电机控制中，我们时常需要关注电机的转速，并以此来了解电机的运行状态，因此，如何获得一个准确的电机速度至关重要。

本章分为如下几个小节：

7.1 编码器简介
7.2 编码器测速原理
7.3 编码器测速实现

7.1 编码器简介

1）**概念**：编码器是一种能将直线位移、角位移数据转换为脉冲信号、二进制编码的设备。它本质上就是一个传感器，可以把角位移或直线位移转换成电信号，并反馈给控制器，使控制器知道当前机械运动的位置、角度等信息。编码器的实物图如图 7.1.1 所示：

图 7.1.1 编码器实物

2）特点：精度高、结构简单、体积小、使用可靠、性价比高，等等。

3）应用场景：数控机床、机器人、雷达、光电经纬仪、地面指挥仪、高精度闭环调速系统、伺服系统等诸多领域。

4）编码器的分类：编码器的分类方式有很多，我们这里列举两种分类方式：按检测原理和编码类型，详见下图：

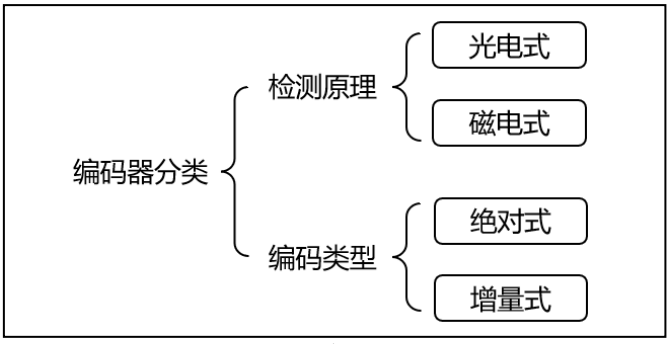


图 7.1.2 编码器分类

从图 7.1.2 中可以看到，编码器按照检测原理可以分为光电式和磁电式；按照编码类型可分为增量式和绝对式。在实际的应用中，这四类编码器并不是相对独立的，它们经过组合后，就变成了光电绝对式、光电增量式、磁电绝对式和磁电增量式这四种编码器。

5）编码器原理：我们这里主要介绍磁电增量式、光电增量式以及光电绝对式这 3 种常用编码器的工作原理。

① 磁电增量式

原理：利用霍尔效应，将位移转换成计数脉冲，用脉冲个数计算位移和速度。

磁电增量式编码器的具体工作原理如图 7.1.3 所示：

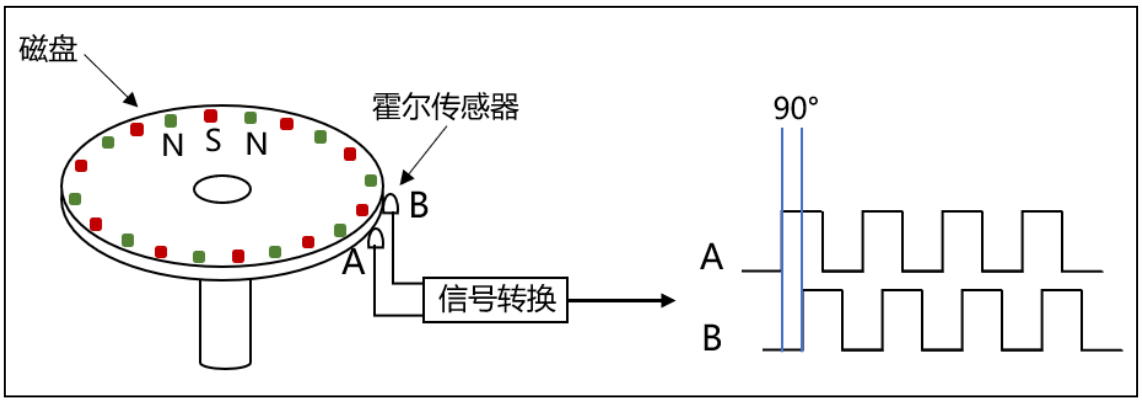


图 7.1.3 磁电增量式编码器工作原理

图 7.1.3 中，磁电增量式编码器的结构包含：磁盘、霍尔传感器以及信号转换电路 3 个部分，其中，磁盘是由一个个交替排布的 S 极和 N 极磁极组成；霍尔传感器可以把磁场的变化转换成电信号的变化，它通常有 A、B 两相（有的还有 Z 相），这两相的安装位置形成一定的夹角，这使得输出的 A、B 两相信号有 90° 的相位差；信号转换电路可以把电信号转换成脉冲信号。

在实际应用中，磁盘会装在电机的转轴上，它会随着电机的转轴旋转，而磁盘上面的 S 极和 N 极就会交替地经过霍尔传感器的 A、B 两相，霍尔传感器就可以把磁盘上的磁场变化转换为电信号的变化，输入到信号转换电路中，经过信号的转换之后，我们就可以得到 A、B 两相脉冲信号了。从上图中可以看到，A、B 两相脉冲信号存在 90° 的相位差，而磁盘的正反转方向就决定了是 A 相信号在前还是 B 相在前。

② 光电增量式

原理：利用光电系统，将位移转换成计数脉冲，用脉冲个数计算位移和速度。

光电增量式编码器的具体工作原理如图 7.1.4 所示：

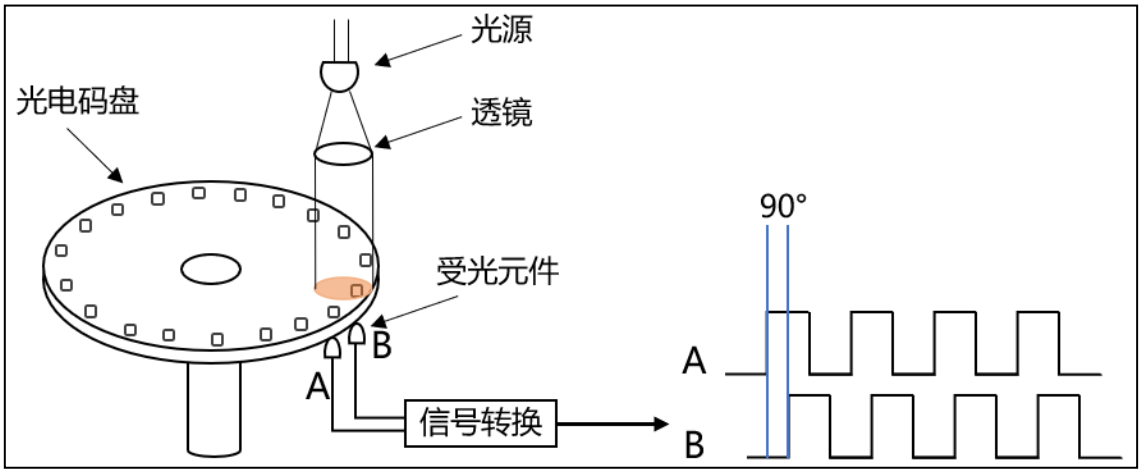


图 7.1.4 光电增量式编码器工作原理

图 7.1.4 中，光电增量式编码器的结构包含：光电码盘、光源、透镜、受光元件以及信号转换电路 5 个部分，其中，光电码盘上有一个个均匀排布的透光孔；光源和透镜形成一个聚光系统；受光元件可以把光线的变化转换成电信号的变化，它通常有 A、B 两相（有的还有 Z 相），这两相的安装位置形成一定的夹角，这使得输出的 A、B 两相信号有 90° 的相位差；信号转换电路可以把电信号转换成脉冲信号。

在实际应用中，光电码盘会装在电机的转轴上，它会随着电机的转轴旋转，而码盘上面的透光孔会间歇性地经过 A、B 两相，受光元件就可以把光线变化转换为电信号的变化，输入到信号转换电路中，经过信号的转换之后，我们就可以得到 A、B 两相脉冲信号了。从上图中可以看到，A、B 两相脉冲信号存在 90° 的相位差，而码盘的正反转方向就决定了是 A 相信号在前还是 B 相在前。

③ 光电绝对式

原理：当码盘处于不同位置（角度）时，光敏元件根据受光与否转换出相应的电平信号，最后转换成二进制数输出。

光电绝对式编码器的结构总体来说和光电增量式很类似，都是由光电码盘、光源、透镜、受光元件以及信号转换电路 5 个部分，但是它们码盘结构和输出信号含义不同。光电绝对式编码器的自然二进制码盘如图 7.1.5 所示：

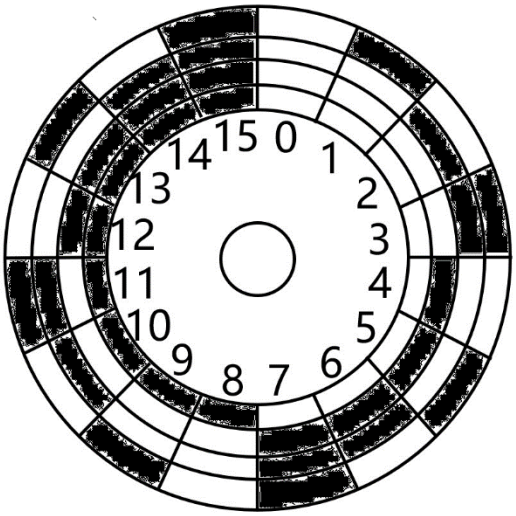


图 7.1.5 自然二进制码盘

图 7.1.5 中，光电绝对式编码器的二进制码盘上有很多圈线槽，我们称为码道。上图中的二进制码盘有 4 个码道，按照自然二进制的方式排列，这个码盘一共被分为 2^4＝16 个区域，这些区域中，黑块不透光，代表 1；白块透光，代表 0。当码盘随着电机转轴旋转，光线会照射到不同的区域，受光元件就能感受到不同的光线情况，最后经过信号的处理，就可以直接输出该区域对应的二进制码了，而我们通过这个二进制码即可得出码盘（电机转轴）的当前位置（角度）。大家需要注意：二进制码盘的每一个位置对应一个确定的二进制码，因此这一类编码器常被应用于位置以及角度测量。

上述的自然二进制码盘读数很方便直观，但是它在实际应用中容易造成读数偏差很大，例如：当码盘停止旋转时，光线照射在 0000 和 1111 这两个相邻的区域之间，此时输出的二进制数可能是 0000~1111 中的任何一个，此时的读数和码盘的实际位置可能就相差很远了。为了避免读数和实际位置出现巨大偏差，我们可以改进一下二进制码的排列方式，使用格雷码形式，如图 7.1.6 所示：

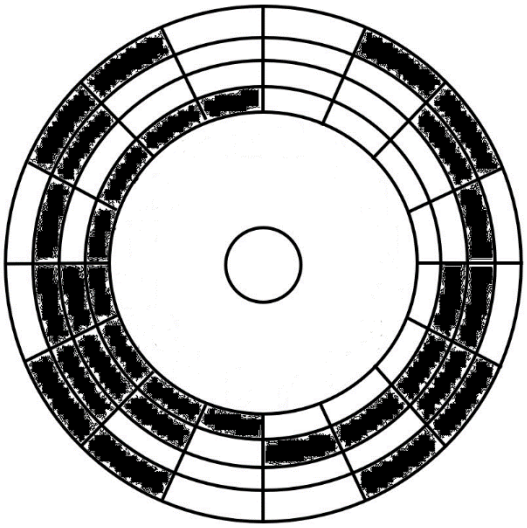


图 7.1.6 格雷码码盘

图 7.1.6 中，格雷码盘有 4 个码道，同样的也能表示 16 个二进制数，但是任意相邻的两个区域之间的二进制码只有一位不同。当我们采用格雷码盘时，如果码盘停止旋转，光线照射

到码盘相邻两个区域之间，其最终输出的二进制数最多只会相差一位，此时位置的偏差范围就很小了。

6）编码器基本参数：

① 分辨率：编码器每个计数单位之间产生的距离，它是编码器可以测量到的最小的距离。对于增量式**编码器，分辨率表示为编码器的转轴每旋转一圈所输出的脉冲数（PPR），也称为多少线，直流有刷电机教程中所使用的编码器是 11 线的。**

② 精度：编码器分辨率和精度是两个独立的概念，**精度**是指编码器输出的信号数据与实际位置之间的误差，常用角分′、角秒″表示。

③ 最大响应频率：编码器每秒能输出的最大脉冲数，单位 Hz，也称为 PPS。

④ 最大转速：指编码器机械系统所能承受的最高转速。

7.2 编码器测速原理

本章节中我们所使用的编码器是磁电增量式编码器，它安装在直流有刷电机的尾部，实物图如图 7.2.1 所示：

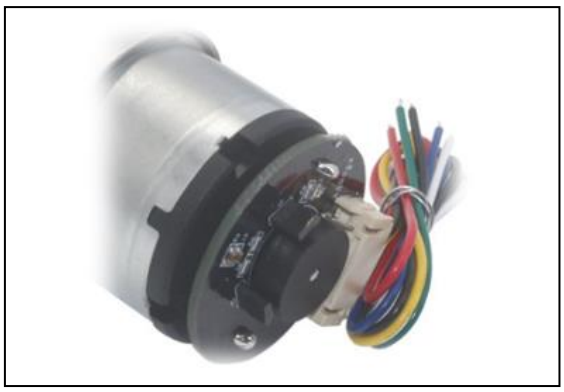


图 7.2.1 直流有刷电机编码器实物

图 7.2.1 中，直流有刷电机的编码器有 A、B 两相，它们会输出两个相位差为 90° 的脉冲。当电机正转时，A 相脉冲在前；当电机反转时，则是 B 相脉冲在前。

有了 A、B 两相脉冲信号之后，我们应该如何去处理这些信号，把它们转换成电机的转速呢？这里就涉及到一个非常重要的功能：定时器编码器接口模式。

STM32 定时器的编码器接口模式就相当于带有方向选择的外部时钟，也就是说，在此模式下，外部输入的脉冲信号可以作为计数器的时钟，而计数的方向则是由脉冲相位的先后所决定的。定时器编码器接口模式的原理如图 7.2.2 所示：

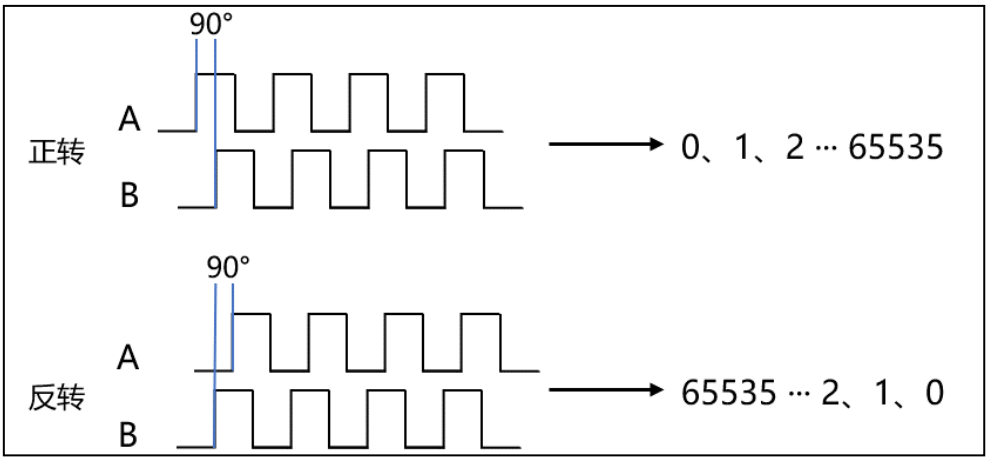


图 7.2.2 定时器编码器接口原理

图 7.2.2 中，当电机（编码器）正转时，输出两相脉冲信号，A 相脉冲在前，此时编码器接口把脉冲信号作为计数器的脉冲，计数方式为递增计数；当电机（编码器）反转时，计数方式就变成了递减计数。

上述的内容只是对编码器接口的原理进行简单的介绍，让大家有一个感性的认识，接下来我们深入研究一下编码器接口的原理：

1）编码器接口框图

我们首先看编码器的接口框图部分，了解一下脉冲信号进入编码器接口的途径，这里以通用定时器为例，具体框图如图 7.2.3 所示：

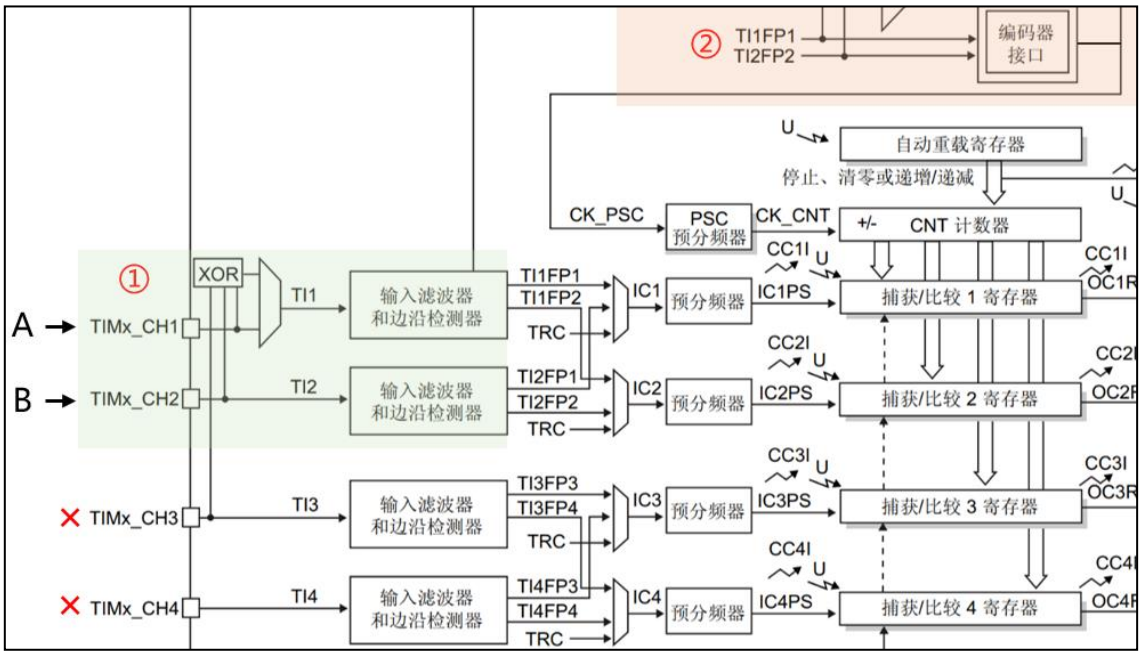


图 7.2.3 通用定时器编码器接口框图

图 7.2.3 中，A、B 两相脉冲信号从 TIMx_CH1 和 TIMx_CH2 这两个通道输入，经过滤波器和边沿检测器（可以设置滤波和反相）的处理，进入到编码器接口控制器中。大家需要注意，TIMx_CH3 和 TIMx_CH4 是不支持编码器接口模式的。

2）编码器接口计数原理

编码器接口可以利用输入脉冲的边沿进行计数，我们通过计数值的变化量，就可以算出输入脉冲信号的变化量，也就可以进一步计算出电机的转速了。接下来我们看一下编码器接口是如何根据脉冲边沿计数的，它的计数方向与输入脉冲信号的关系如表 7.2.1 所示：

有效边沿	相反信号的电平（TI1FP1 对应 TI2，TI2FP2 对应 TI1）	TI1FP1 信号		TI2FP2 信号	
		上升	下降	上升	下降
仅在 TI1 处计数	高	递减	递增	不计数	不计数
	低	递增	递减	不计数	不计数
仅在 TI2 处计数	高	不计数	不计数	递增	递减
	低	不计数	不计数	递减	递增
在 TI1 和 TI2 处均计数	高	递减	递增	递增	递减
	低	递增	递减	递减	递增

表 7.2.1 计数方向与输入脉冲信号的关系

表 7.2.1 来自于《STM32F4xx 参考手册_V4（中文版）.pdf》中的表 75（第 415 页），上表中的有效边沿我们可以通过代码去配置，一共有 3 种边沿检测方式，其中仅在 TI1 处计数表示只检测 TI1 上的脉冲边沿变化，这时候的计数方向需要结合 TI2FP2 上的电平情况来确定，其他的两种边沿检测方式原理也是一样的，下面我们以一个实例来理解这个表格的内容：

假设我们把 A 相接在 CH1（TI1），B 相接在 CH2（TI2），选择仅在 TI1 处计数（仅检测 A 相边沿）。此时编码器接口计数方向和输入脉冲信号的关系如下表：

有效边沿	B 相信号	A 相信号	
		上升	下降
仅在 A 相边沿计数	高	递减	递增
	低	递增	递减

表 7.2.2 仅在 A 相边沿计数

编码器输出的 A、B 两相脉冲信号如图 7.2.4 所示：

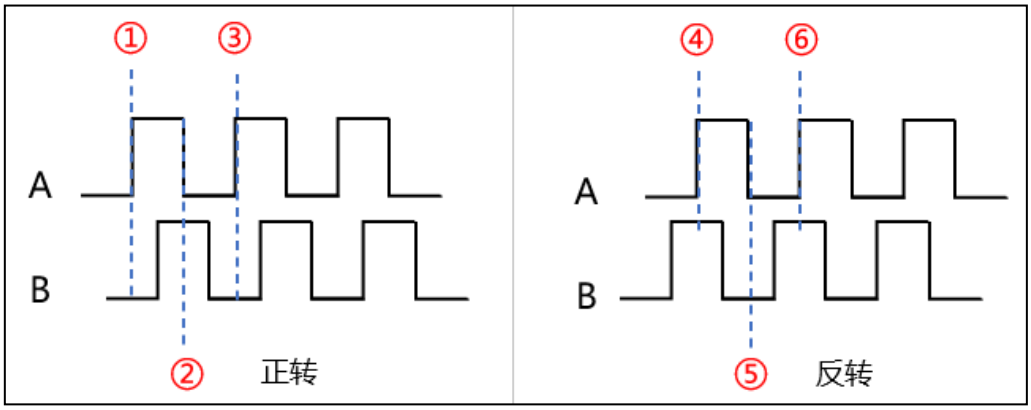


图 7.2.4 A、B 相脉冲信号

图 7.2.4 中，A、B 两相输出的脉冲信号有两种情况：当编码器正转，A 相在前；当编码器反转，B 相在前，我们选择仅在 TI1 处计数，也就是只检测 A 相的边沿。接下来我们分别介绍这两种情况下的计数方向：

正转：当 A 相上升沿到来时（图中①处），我们需要关注 B 相的电平高低，从图 7.2.4 中可看到 B 相此时是低电平，结合表 7.2.2，可以得知此时计数方向为递增计数；当 A 相下降沿到来时（图中②处），从图中可以看到 B 相此时是高电平，结合表 7.2.2，可以得知此时计数方向为递增计数；当 A 相上升沿再次到来时（图中③处），同理可得此时计数方向为递增计数。综上所得，我们可以知道此时编码器正转对应的计数方向就是递增计数。

反转：当 A 相上升沿到来时（图中④处），我们需要关注 B 相的电平高低，从图 7.2.4 中可看到 B 相此时是高电平，结合表 7.2.2，可以得知此时计数方向为递减计数；当 A 相下降沿到来时（图中⑤处），从图中可以看到 B 相此时是低电平，结合表 7.2.2，可以得知此时计数方向为递减计数；当 A 相上升沿再次到来时（图中⑥处），同理可得此时计数方向为递减计数。综上所得，我们可以知道此时编码器反转对应的计数方向就是递减计数。

上述的就是仅在 TI1 处计数的实例分析，其他两种边沿计数方式的原理是一样的，大家可以举一反三，我们这里就不再展开分析。

注意：1、选择仅在 TI1 或者 TI2 处计数，就相当于对脉冲信号进行了 2 倍频（两个边沿），此时如果编码器输出 10 个脉冲信号，那么就会计数 20 次。2、选择的是在 TI1 和 TI2 处均计数，就相当于对脉冲信号进行了 4 倍频，此时如果编码器输出 10 个脉冲信号，那么就会计数 40 次。因此，我们通过计数次数来计算电机速度的时候，需要除以相应的倍频系数。

至此，A、B 两相脉冲信号的变化就转换成了定时器的计数变化。接下来我们就可以通过一分钟内计数的变化量来计算电机的速度，具体公式如下：

```
电机转速 = 一分钟内计数变化量 / 倍频系数 / 编码器线数 / 减速比
```

到这里，编码器的测速原理就介绍完了，接下来我们开始实现直流有刷电机的测速实验。

7.3 编码器测速实现

本小节我们来学习使用直流有刷电机的编码器测速功能，关于编码器测速的原理大家可以回顾上一小节的内容。我们这里选择通用定时器的编码器接口来实现测速功能，而高级定时器则用于电机的基础驱动。

7.3.1 TIM2/TIM3/TIM4/TIM5 寄存器

下面介绍 TIM2/TIM3/TIM4/TIM5 的几个与编码器测速相关且重要的寄存器，相关内容可以参考《STM32F4xx 参考手册_V4（中文版）.pdf》定时器相关章节。

- **控制寄存器 1（TIMx_CR1）**

TIM2/TIM3/TIM4/TIM5 的控制寄存器 1 描述如图 7.3.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 4 **DIR**：方向 (Direction)

0：计数器递增计数
1：计数器递减计数

注意：当定时器配置为中心对齐模式或编码器模式时，该位为只读状态。

位 0 **CEN**：计数器使能 (Counter enable)

0：禁止计数器
1：使能计数器

注意：只有事先通过软件将 CEN 位置 1，才可以使用外部时钟、门控模式和编码器模式。而触发模式可通过硬件自动将 CEN 位置 1。

图 7.3.1.1 TIMx_CR1 寄存器

上图中我们只列出了本实验需要用的位：CEN 位，此位用于使能计数器的工作，必须要设置该位为 1，才可以开始计数；DIR 位，在编码器模式下可通过读取该位来判断计数方向。

- **捕获/比较模式寄存器 1（TIMx_CCMR1）**

该寄存器在编码器接口模式中作为输入功能，其描述如图 7.3.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 7:4 **IC1F**：输入捕获 1 滤波器 (Input capture 1 filter)

此位域可定义 TI1 输入的采样频率和适用于 TI1 的数字滤波器带宽。数字滤波器由事件计数器组成，每 N 个事件才视为一个有效边沿：

0000：无滤波器，按 f_{DTS} 频率进行采样
0001：$f_{SAMPLING}=f_{CK_INT}$，N=2
0010：$f_{SAMPLING}=f_{CK_INT}$，N=4
0011：$f_{SAMPLING}=f_{CK_INT}$，N=8
0100：$f_{SAMPLING}=f_{DTS}/2$，N=6
0101：$f_{SAMPLING}=f_{DTS}/2$，N=8
0110：$f_{SAMPLING}=f_{DTS}/4$，N=6
0111：$f_{SAMPLING}=f_{DTS}/4$，N=8
1000：$f_{SAMPLING}=f_{DTS}/8$，N=6
1001：$f_{SAMPLING}=f_{DTS}/8$，N=8
1010：$f_{SAMPLING}=f_{DTS}/16$，N=5
1011：$f_{SAMPLING}=f_{DTS}/16$，N=6
1100：$f_{SAMPLING}=f_{DTS}/16$，N=8
1101：$f_{SAMPLING}=f_{DTS}/32$，N=5
1110：$f_{SAMPLING}=f_{DTS}/32$，N=6
1111：$f_{SAMPLING}=f_{DTS}/32$，N=8

注意：在当前硅版本中，当 ICxF[3:0]= 1、2 或 3 时，将用 CK_INT 代替公式中的 f_{DTS}。

位 3:2 **IC1PSC**：输入捕获 1 预分频器 (Input capture 1 prescaler)

此位域定义 CC1 输入 (IC1) 的预分频比。
只要 CC1E=0（TIMx_CCER 寄存器），预分频器便立即复位。
00：无预分频器，捕获输入上每检测到一个边沿便执行捕获
01：每发生 2 个事件便执行一次捕获
10：每发生 4 个事件便执行一次捕获
11：每发生 8 个事件便执行一次捕获

位 1:0 **CC1S**：捕获/比较 1 选择 (Capture/Compare 1 selection)

此位域定义通道方向（输入/输出）以及所使用的输入。
00：CC1 通道配置为输出
01：CC1 通道配置为输入，IC1 映射到 TI1 上
10：CC1 通道配置为输入，IC1 映射到 TI2 上
11：CC1 通道配置为输入，IC1 映射到 TRC 上。此模式仅在通过 TS 位（TIMx_SMCR 寄存器）选择内部触发输入时有效

注意：仅当通道关闭时（TIMx_CCER 中的 CC1E = 0），才可向 CC1S 位写入数据。

图 7.3.1.2 TIMx_CCMR1 寄存器

大家需要注意：TIMx_CCMR1 寄存器对应于通道 1 和通道 2 的设置，CCMR2 寄存器对

应通道 3 和通道 4，而通道 3 和通道 4 是不支持编码器接口模式的，所以这里只用到 TIMx_CCMR1。

上图中的 CC1S[1:0]这两个位用于 CCR1 的通道配置，这里我们设置 IC1S[1:0]=01，也就是配置 IC1 映射在 TI1 上。

输入捕获 1 预分频器 IC1PSC[1:0]，这个比较好理解。我们是 1 次边沿就触发 1 次计数，所以选择 00 就是了。

输入捕获 1 滤波器 IC1F[3:0]，这个用来设置输入采样频率和数字滤波器长度。其中，f_{CK_INT} 是定时器时钟源频率，按照例程的配置为 84Mhz，而 f_{DTS} 则是根据 TIMx_CR1 的 CKD[1:0]的设置来确定的，如果 CKD[1:0]设置为 00，那么 $f_{DTS}=f_{CK_INT}$。N 值采样次数，举个简单的例子：假设 IC1F[3:0]=0011，并设置 IC1 映射到 TI1 上。表示以 f_{CK_INT} 为采样频率，当连续 8 次都是采样到 TI1 为高电平或者低电平，滤波器才输出一个有效输出边沿。当 8 次采样中有高有低，那就保持原来的输出，这样可以滤除高频干扰信号，从而达到滤波的效果。

- **捕获/比较使能寄存器（TIMx_CCER）**

TIM2/TIM3/TIM4/TIM5 的捕获/比较使能寄存器，该寄存器控制着各个输入输出通道的开关和极性。TIMx_CCER 寄存器描述如图 7.3.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

位 1 **CC1P**：*捕获/比较 1 输出极性 (Capture/Compare 1 output Polarity)*。

CC1 通道配置为输出：

0：OC1 高电平有效

1：OC1 低电平有效

CC1 通道配置为输入：

CC1NP/CC1P 位可针对触发或捕获操作选择 TI1FP1 和 TI2FP1 的极性。

00：非反相/上升沿触发

电路对 TIxFP1 上升沿敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 未反相（在门控模式或编码器模式下执行触发操作）。

01：反相/下降沿触发

电路对 TIxFP1 下降沿敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 反相（在门控模式或编码器模式下执行触发操作）。

10：保留，不使用此配置。

11：非反相/上升沿和下降沿均触发

电路对 TIxFP1 上升沿和下降沿都敏感（在复位模式、外部时钟模式或触发模式下执行捕获或触发操作），TIxFP1 未反相（在门控模式下执行触发操作）。编码器模式下不得使用此配置。

位 0 **CC1E**：*捕获/比较 1 输出使能 (Capture/Compare 1 output enable)*。

CC1 通道配置为输出：

0：关闭——OC1 未激活

1：开启——在相应输出引脚上输出 OC1 信号

CC1 通道配置为输入：

此位决定了是否可以实际将计数器值捕获到输入捕获/比较寄存器 1 (TIMx_CCR1) 中。

0：禁止捕获

1：使能捕获

图 7.3.1.3 TIMx_CCER 寄存器

我们要用到这个寄存器的 4 个位，分别是 CC1E、CC1P、CC2E 和 CC2P 位，上图中只列出了 CCIE 和 CCIP 位，其他两个位同理。要使能编码器接口模式，必须设置 CC1E 和 CC2E 为 1，而 CC1P 和 CC2P 设置的是边沿触发的方向。

- **从模式控制寄存器（TIMx_SMCR）**

TIM2/TIM3/TIM4/TIM5 的从模式控制寄存器描述如图 7.3.1.4 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			Res.	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw

位 2:0 **SMS**：从模式选择 (Slave mode selection)

选择外部信号时，触发信号 (TRGI) 的有效边沿与外部输入上所选的极性相关（请参见输入控制寄存器和控制寄存器说明）。
000：禁止从模式——如果 CEN = “1”，预分频器时钟直接由内部时钟提供。
001：编码器模式 1——计数器根据 TI1FP1 电平在 TI2FP2 边沿递增/递减计数。
010：编码器模式 2——计数器根据 TI2FP2 电平在 TI1FP1 边沿递增/递减计数。
011：编码器模式 3——计数器在 TI1FP1 和 TI2FP2 的边沿计数，计数的方向取决于另外一个信号的电平。
100：复位模式——在出现所选触发输入 (TRGI) 上升沿时，重新初始化计数器并生成一个寄存器更新事件。
101：门控模式——触发输入 (TRGI) 为高电平时使能计数器时钟。只要触发输入变为低电平，计数器立即停止计数（但不复位）。计数器的启动和停止都是受控的。
110：触发模式——触发信号 TRGI 出现上升沿时启动计数器（但不复位）。只控制计数器的启动。
111：外部时钟模式 1——由所选触发信号 (TRGI) 的上升沿提供计数器时钟。

注意：如果将 TI1F_ED 选作触发输入 (TS=100)，则不得使用门控模式。实际上，TI1F 每次转换时，TI1F_ED 都输出 1 个脉冲，而门控模式检查的则是触发信号的电平。

图 7.3.1.4 TIMx_SMCR 寄存器

该寄存器的 SMS[2:0]位，用于从模式选择，其实就是选择计数器输入时钟的来源。在编码器模式中，如果需要仅在 TI1 边沿处计数，则设置 SMS[2:0]=010；如果需要仅在 TI2 边沿处计数，则设置 SMS[2:0]=001；如果需要在 TI1、TI2 边沿处都计数，则设置 SMS[2:0]=011。本实验我们在 TI1 和 TI2 边沿处均计数，也就是对脉冲信号进行 4 倍频。

7.3.2 硬件设计

1. 例程功能

1、本实验以电机开发板的直流有刷电机驱动接口 1 为例，基于电压温度电流采集实验，加入编码器测速功能，利用 TIM3_CH1（由 PC6 复用）和 TIM3_CH2（由 PC7 复用）作为编码器脉冲信号输入接口。

2、当按键 0 按下，就增大 PWM 的比较值变量，电机将加速，当按键 1 按下，就减小 PWM 的比较值变量，电机将减速，当比较值变量为正数时电机正转，反之电机反转，按下按键 2 则马上停止电机。

3、定时器 6 中断里面计算电机速度。

4、屏幕显示按键功能信息以及电机转速。

5、串口打印驱动板电压、电流、温度和转速信息。

6、LED0 闪烁指示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）独立按键
KEY0 - PE2
KEY1 - PE3
KEY2 - PE4
3）定时器 1、3、6
TIM1 正常输出通道 PA8
TIM1 互补输出通道 PB13
TIM3 编码器 A 相输入通道 PC6
TIM3 编码器 B 相输入通道 PC7
4）SD（刹车）信号输出 PF10

5）ADC

ADC1 通道 9　PB1（电压）
ADC1 通道 0　PA0（温度）
ADC1 通道 8　PB0（电流）

6）串口 1

USART1_TX　PB6（发送）
USART1_RX　PB7（接收）

3. 原理图

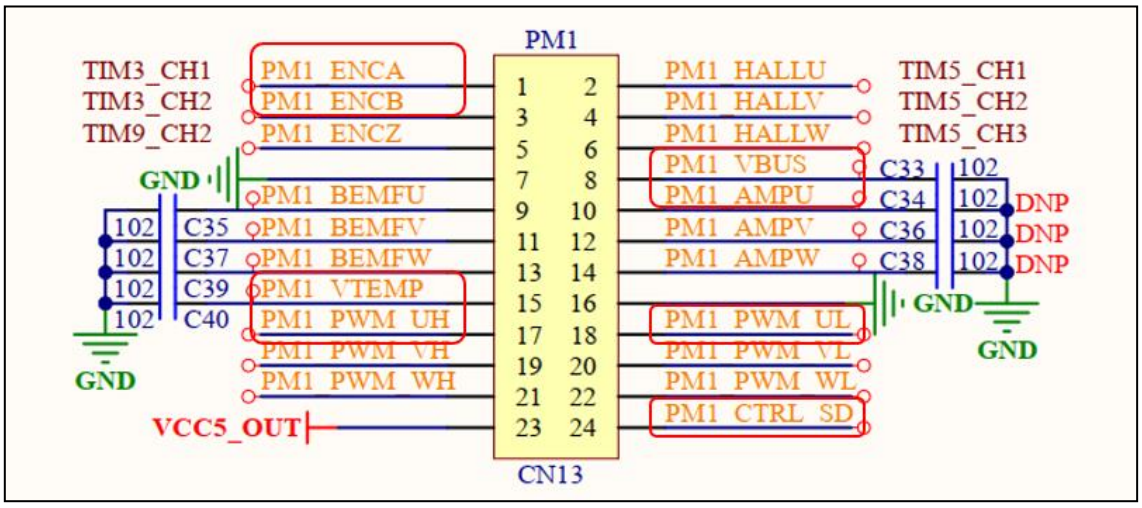


图 7.3.2.1 直流有刷电机接口原理图

图 7.3.2.1 就是我们 DMF407 电机开发板的直流有刷电机接口 1 原理图，本实验我们除了用到基础驱动以及电压温度电流采集所需的引脚，还需要用到 PM1_ENCA（PC6）、PM1_ENCB（PC7）这两个引脚，它们分别用于连接编码器的 A、B 两相。

本实验的硬件接线部分和上一章节一模一样，这里不再赘述，大家可以回顾上一章节的实验内容。

7.3.3 程序设计

7.3.3.1. 定时器的 HAL 库驱动

定时器在 HAL 库中的驱动代码在前面已经介绍了部分，这里我们再介绍几个本实验用到的函数。

1. HAL_TIM_Encoder_Init 函数

定时器的编码器接口初始化函数。其声明如下：

```
HAL_StatusTypeDef HAL_TIM_Encoder_Init(TIM_HandleTypeDef *htim,
                                       TIM_Encoder_InitTypeDef *sConfig);
```

- **函数描述：**

  用于初始化编码器接口模式。

- **函数形参：**

  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量，用于配置定时器基本参数。

  形参 2 是 TIM_Encoder_InitTypeDef 结构体类型指针变量，用于配置编码器模式参数。

  重点了解一下 TIM_Encoder_InitTypeDef 结构体指针类型，其定义如下：

```
typedef struct
{
  uint32_t EncoderMode;      /* 编码器模式 */
  uint32_t IC1Polarity;      /* IC1 输入信号的极性 */
  uint32_t IC1Selection;     /* 输入通道映射 */
  uint32_t IC1Prescaler;     /* 输入捕获预分频器 */
```

```
  uint32_t IC1Filter;        /* 输入捕获滤波器 */
  uint32_t IC2Polarity;      /* IC2 输入信号的极性 */
  uint32_t IC2Selection;     /* 输入通道映射 */
  uint32_t IC2Prescaler;     /* 输入捕获预分频器 */
  uint32_t IC2Filter;        /* 输入捕获滤波器 */
} TIM_Encoder_InitTypeDef;
```

该结构体成员用于配置编码器接口的参数。成员变量 EncoderMode 用来设置编码器模式，我们可以选择仅在 TI1 / TI2 处计数或者在 TI1、TI2 处均计数。本实验中我们选择在 TI1、TI2 处均计数，也就是 4 倍频。

成员变量 IC1Polarity 在编码器模式下用于设置输入信号是否反相，它设定的是 TIMx_CCER 寄存器的 CCxNP 位和 CCxP 位。本实验我们不需要输入信号反相。

成员变量 ICSelection 用来设置通道映射关系，它设定 TIMx_CCMRx 寄存器的 CCxS[1:0] 位的值。在编码器接口模式下，此成员设置为输入模式即可（其他型号芯片不一定），本实验我们选择 TIM_ICSELECTION_DIRECTTI。

成员变量 ICPrescaler 用来设置输入捕获的时钟分频系数，本实验不需要分频，所以选择 TIM_ICPSC_DIV1。

成员变量 ICFilter 用来设置滤波器长度，可选设置 0x0 至 0x0F。它设定 TIMx_CCMRx 寄存器 ICxF[3:0] 位的值。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

2. HAL_TIM_Encoder_Start 函数

定时器的编码器接口模式启动函数，其声明如下：

```
HAL_StatusTypeDef HAL_TIM_Encoder_Start(TIM_HandleTypeDef *htim,
                                        uint32_t Channel);
```

- **函数描述：**

  用于启动定时器的编码器接口模式。

- **函数形参：**

  形参 1 是 TIM_HandleTypeDef 结构体类型指针变量。

  形参 2 是定时器通道，范围：TIM_CHANNEL_1 到 TIM_CHANNEL_4。

- **函数返回值：**

  HAL_StatusTypeDef 枚举类型的值。

定时器的编码器模式配置步骤

1）开启 TIMx 和输入通道的 GPIO 时钟，配置该 IO 口的复用功能输入

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能。本实验我们默认用到定时器 3 通道 1 和通道 2，对应 IO 是 PC6，PC7，它们的时钟开启方法如下：

```
__HAL_RCC_TIM3_CLK_ENABLE();            /* 使能定时器 3 */
__HAL_RCC_GPIOC_CLK_ENABLE();           /* 开启 GPIOC 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数

使用定时器的输入捕获功能时，通过 HAL_TIM_Encoder_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_Encoder_MspInit 函数，我们可以通过后者存放定时器和 GPIO 时钟使能、GPIO 初始化、中断使能以及优先级设置等代码。

3）设置 TIMx_CHy 的编码器模式，开启编码器通道

在 HAL 库中，定时器的编码器接口模式是通过 HAL_TIM_Encoder_Init 函数来设置定时器的编码器通道，包括映射关系，输入滤波和输入分频等。

4）使能定时器更新中断，配置定时器中断优先级

通过__HAL_TIM_ENABLE_IT 函数使能定时器更新中断。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断。

通过 HAL_NVIC_SetPriority 函数设置中断优先级。

编码器在电机运行时会一直旋转并输出脉冲信号，如果时间较长，那么定时器计数就会溢出，我们必须对溢出必须做处理，否则电机速度的计算结果就不准了。

5）编写中断服务函数

定时器 3 中断服务函数为：TIM3_IRQHandler，当发生中断的时候，程序就会执行中断服务函数。HAL 库为了使用方便，提供了一个定时器中断通用处理函数 HAL_TIM_IRQHandler，该函数会调用一些定时器相关的回调函数，用于给用户处理定时器中断到了之后，需要处理的程序。本实验我们用到更新（溢出）中断回调函数 HAL_TIM_PeriodElapsedCallback。详见本例程源码。

7.3.3.2 程序流程图

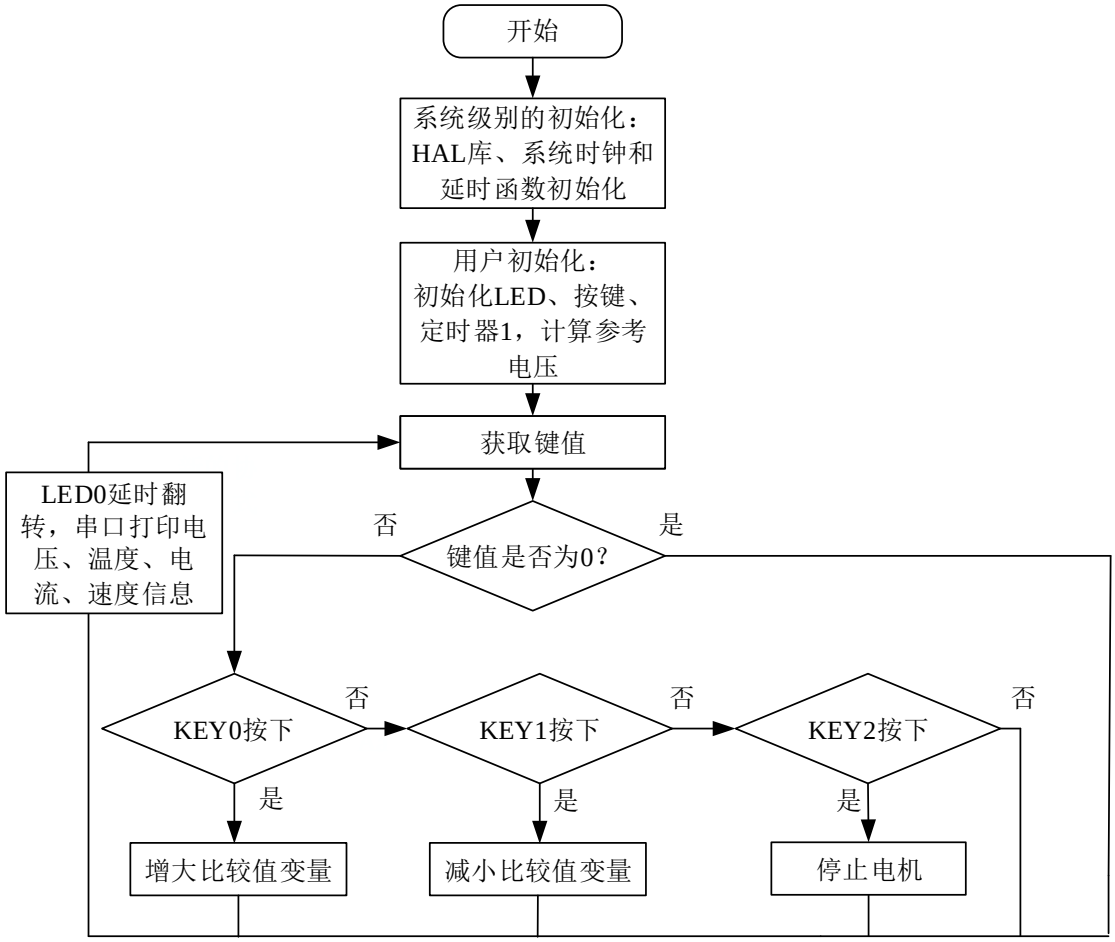


图 7.3.3.2.1 编码器测速程序流程图

7.3.3.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。关于基础驱动和电压温度电流采集的代码我们不再赘述，大家可以回顾相应的章节。

注意：本实验需要用到 TIM3 的编码器模式和 TIM6 的更新中断，相关的定时器驱动源码包括两个文件：dcmotor_tim.c 和 dcmotor_tim.h。

首先看 dcmotor_tim.h 头文件的几个宏定义：

```
/* 通用定时器 定义 */
#define GTIM_TIMX_ENCODER_CH1_GPIO_PORT        GPIOC
#define GTIM_TIMX_ENCODER_CH1_GPIO_PIN         GPIO_PIN_6
#define GTIM_TIMX_ENCODER_CH1_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)  /* PC 口时钟使能 */

#define GTIM_TIMX_ENCODER_CH2_GPIO_PORT        GPIOC
```

```
#define GTIM_TIMX_ENCODER_CH2_GPIO_PIN          GPIO_PIN_7
#define GTIM_TIMX_ENCODER_CH2_GPIO_CLK_ENABLE()
        do{ __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)  /* PC口时钟使能 */

/* TIMX 引脚复用设置
 * 因为PC6/PC7，默认并不是TIM3的功能脚，必须开启复用，才可以用作TIM3的CH1/CH2功能
 */
#define GTIM_TIMX_ENCODERCH1_GPIO_AF          GPIO_AF2_TIM3
#define GTIM_TIMX_ENCODERCH2_GPIO_AF          GPIO_AF2_TIM3

#define GTIM_TIMX_ENCODER                     TIM3
#define GTIM_TIMX_ENCODER_INT_IRQn            TIM3_IRQn
#define GTIM_TIMX_ENCODER_INT_IRQHandler      TIM3_IRQHandler

#define GTIM_TIMX_ENCODER_CH1                 TIM_CHANNEL_1 /* 通道1 */
#define GTIM_TIMX_ENCODER_CH1_CLK_ENABLE()
        do{ __HAL_RCC_TIM3_CLK_ENABLE(); }while(0)         /* TIM3 时钟使能 */

#define GTIM_TIMX_ENCODER_CH2                 TIM_CHANNEL_2 /* 通道2 */
#define GTIM_TIMX_ENCODER_CH2_CLK_ENABLE()
        do{ __HAL_RCC_TIM3_CLK_ENABLE(); }while(0)         /* TIM3 时钟使能 */

/* 基本定时器 定义
 * 捕获编码器值，用于计算速度
 */
#define BTIM_TIMX_INT                         TIM6
#define BTIM_TIMX_INT_IRQn                    TIM6_DAC_IRQn
#define BTIM_TIMX_INT_IRQHandler              TIM6_DAC_IRQHandler
#define BTIM_TIMX_INT_CLK_ENABLE()
        do{ __HAL_RCC_TIM6_CLK_ENABLE(); }while(0)         /* TIM6 时钟使能 */

/* 编码器参数结构体 */
typedef struct
{
  int encode_old;                   /* 上一次计数值 */
  int encode_now;                   /* 当前计数值 */
  float speed;                      /* 编码器速度 */
} ENCODE_TypeDef;

extern ENCODE_TypeDef g_encode;    /* 编码器参数变量 */
```

可以把上面的宏定义分成四部分：

第一部分是通用定时器 3 编码器模式输入通道对应的 IO 口宏定义；

第二部分是通用定时器 3 编码器模式输入通道的相应宏定义；

第三部分是定时器 6 更新中断的相应宏定义；

第四部分是编码器参数的结构体，这个结构体用于管理编码器的计数值和速度。

下面看 dcmotor_tim.c 的程序，首先是通用定时器的编码器接口模式初始化函数。

```
/**
 * @brief       通用定时器 TIMX 通道Y 编码器接口模式 初始化函数
 * @note
 *              通用定时器的时钟来自APB1,当PPRE1 ≥ 2分频的时候
 *              通用定时器的时钟为APB1时钟的2倍，而APB1为42M，所以定时器时钟 = 84Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值。
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void gtim_timx_encoder_chy_init(uint16_t arr, uint16_t psc)
{
    g_timx_encode_chy_handle.Instance = GTIM_TIMX_ENCODER;      /* 定时器x */
```

```
    g_timx_encode_chy_handle.Init.Prescaler = psc;                  /* 定时器分频 */
    g_timx_encode_chy_handle.Init.Period = arr;                     /* 自动重装载值 */
    g_timx_encode_chy_handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    /* TI1、TI2 都检测，4 倍频 */
    g_timx_encoder_chy_handle.EncoderMode = TIM_ENCODERMODE_TI12;
    /* 输入极性，非反向 */
    g_timx_encoder_chy_handle.IC1Polarity = TIM_ICPOLARITY_RISING;
    /* 输入通道选择 */
    g_timx_encoder_chy_handle.IC1Selection = TIM_ICSELECTION_DIRECTTI;
    g_timx_encoder_chy_handle.IC1Prescaler = TIM_ICPSC_DIV1;    /* 不分频 */
    g_timx_encoder_chy_handle.IC1Filter = 10;                   /* 滤波器设置 */

    /* 输入极性，非反向 */
    g_timx_encoder_chy_handle.IC2Polarity = TIM_ICPOLARITY_RISING;
    /* 输入通道选择 */
    g_timx_encoder_chy_handle.IC2Selection = TIM_ICSELECTION_DIRECTTI;
    g_timx_encoder_chy_handle.IC2Prescaler = TIM_ICPSC_DIV1;    /* 一分频 */
    g_timx_encoder_chy_handle.IC2Filter = 10;                   /* 滤波器设置 */
    HAL_TIM_Encoder_Init(&g_timx_encode_chy_handle,
                         &g_timx_encoder_chy_handle); /* 初始化定时器 x 编码器 */
    /* 使能编码器通道 1、2 */
    HAL_TIM_Encoder_Start(&g_timx_encode_chy_handle,GTIM_TIMX_ENCODER_CH1);
    HAL_TIM_Encoder_Start(&g_timx_encode_chy_handle,GTIM_TIMX_ENCODER_CH2);
     /* 使能更新中断、清除中断标志位 */
    __HAL_TIM_ENABLE_IT(&g_timx_encode_chy_handle,TIM_IT_UPDATE);
    __HAL_TIM_CLEAR_FLAG(&g_timx_encode_chy_handle,TIM_IT_UPDATE);
}
```

HAL_TIM_Encoder_Init 初始化定时器 3 的基础工作参数和编码器接口，如：ARR、PSC、编码器模式、滤波器等，第二部分则是调用 HAL_TIM_Encoder_Start 函数开启编码器通道。最后是使能更新中断和清除中断标志位。通道对应的 IO、时钟开启和 NVIC 的初始化都在 HAL_TIM_Encoder_MspInit 函数里编写，其定义如下：

```
/**
 * @brief       定时器底层驱动，时钟使能，引脚配置
                此函数会被 HAL_TIM_Encoder_Init()调用
 * @param       htim:定时器句柄
 * @retval      无
 */
void HAL_TIM_Encoder_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == GTIM_TIMX_ENCODER)
    {
        GPIO_InitTypeDef gpio_init_struct;
        GTIM_TIMX_ENCODER_CH1_GPIO_CLK_ENABLE();                /* 开启 IO 时钟 */
        GTIM_TIMX_ENCODER_CH2_GPIO_CLK_ENABLE();
        GTIM_TIMX_ENCODER_CH1_CLK_ENABLE();                     /*开启定时器时钟*/
        GTIM_TIMX_ENCODER_CH2_CLK_ENABLE();

        gpio_init_struct.Pin = GTIM_TIMX_ENCODER_CH1_GPIO_PIN;  /* 通道 1 的 IO */
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;                /* 复用推挽输出 */
        gpio_init_struct.Pull = GPIO_NOPULL;                    /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
        gpio_init_struct.Alternate = GTIM_TIMX_ENCODERCH1_GPIO_AF; /* 端口复用 */
        HAL_GPIO_Init(GTIM_TIMX_ENCODER_CH1_GPIO_PORT, &gpio_init_struct);

        gpio_init_struct.Pin = GTIM_TIMX_ENCODER_CH2_GPIO_PIN;  /* 通道 2 的 IO */
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;                /* 复用推挽输出 */
        gpio_init_struct.Pull = GPIO_NOPULL;                    /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;          /* 高速 */
        gpio_init_struct.Alternate = GTIM_TIMX_ENCODERCH2_GPIO_AF;
        HAL_GPIO_Init(GTIM_TIMX_ENCODER_CH2_GPIO_PORT, &gpio_init_struct);
```

```
        HAL_NVIC_SetPriority(GTIM_TIMX_INT_IRQn, 2, 0);          /* 优先级设置 */
        HAL_NVIC_EnableIRQ(GTIM_TIMX_INT_IRQn);                  /* 开启中断 */
    }
}
```

该函数调用HAL_GPIO_Init函数初始化定时器输入通道对应的IO，并且开启GPIO时钟，使能定时器。其中要注意IO口复用功能的选择一定要选对了。最后配置中断抢占优先级和响应优先级，以及打开定时器中断。

接着我们看基本定时器6的中断初始化函数及其底层初始化函数。

```
/**
 * @brief       基本定时器 TIMX 定时中断初始化函数
 * @note
 *              基本定时器的时钟来自 APB1,当 PPRE1 ≥ 2 分频的时候
 *              基本定时器的时钟为 APB1 时钟的 2 倍, 而 APB1 为 42M, 所以定时器时钟 = 84Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值。
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void btim_timx_int_init(uint16_t arr, uint16_t psc)
{
    timx_handler.Instance = BTIM_TIMX_INT;                      /* 基本定时器 X */
    timx_handler.Init.Prescaler = psc;                          /* 设置预分频器  */
    timx_handler.Init.CounterMode = TIM_COUNTERMODE_UP;         /* 向上计数器 */
    timx_handler.Init.Period = arr;                             /* 自动装载值 */
    timx_handler.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_Base_Init(&timx_handler);

    HAL_TIM_Base_Start_IT(&timx_handler);       /* 使能基本定时器 x 及其更新中断 */
    __HAL_TIM_CLEAR_IT(&timx_handler,TIM_IT_UPDATE);        /* 清除中断标志位 */
}

/**
 * @brief       定时器底册驱动，开启时钟，设置中断优先级
 *              此函数会被 HAL_TIM_Base_Init()函数调用
 * @param       无
 * @retval      无
 */
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == BTIM_TIMX_INT)
    {
        BTIM_TIMX_INT_CLK_ENABLE();                             /*使能 TIM 时钟*/
        HAL_NVIC_SetPriority(BTIM_TIMX_INT_IRQn, 1, 3);
        HAL_NVIC_EnableIRQ(BTIM_TIMX_INT_IRQn);                 /*开启 ITM3 中断*/
    }
}
```

这一部分函数和基本定时器中断实验是一样的，这里不再赘述，我们使用定时器 6 来定时计算电机速度，本实验中配置溢出时间为1ms，也就是1ms进入一次定时器6的更新中断。

下面开始看中断服务函数的逻辑程序，HAL_TIM_IRQHandler 函数会调用下面的回调函数，我们的逻辑代码就是放在回调函数里，函数定义如下：

```
/**
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @note        此函数会被定时器中断函数共同调用的
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
```

```
    if (htim->Instance == TIM3)
    {
        /* 判断 CR1 的 DIR 位 */
        if(__HAL_TIM_IS_TIM_COUNTING_DOWN(&g_timx_encode_chy_handle))
        {
            g_timx_encode_count--;              /* DIR 位为 1，也就是递减计数 */
        }
        else
        {
            g_timx_encode_count++;              /* DIR 位为 0，也就是递增计数 */
        }
    }
    else if (htim->Instance == TIM6)
    {
        int Encode_now = gtim_get_encode(); /* 获取编码器值，用于计算速度 */
        /* 中位平均值滤除编码器抖动数据，50ms 计算一次速度 */
        speed_computer(Encode_now, 50);
    }
}
```

进入更新中断回调函数后，先判断是不是定时器 3 的寄存器基地址，如果是则读取 CR1 寄存器的 DIR 位，判断计数溢出的方向，如果 DIR 位为 1，就是递减计数，说明是向下溢出，我们让变量 g_timx_encode_count 自减 1；如果 DIR 位为 0，就是递增计数，说明是向上溢出，我们让变量 g_timx_encode_count 自增 1，变量 g_timx_encode_count 将用于计算总的计数值。如果是定时器 6 的寄存器基地址，就先获取编码器的计数总值，然后每隔 50ms 计算一次电机速度，其中所涉及到的两个函数我们后续再介绍。

接下来看编码器计数总值的计算函数。

```
/**
 * @brief       获取编码器的值
 * @param       无
 * @retval      编码器值
 */
int gtim_get_encode(void)
{
    /* 当前计数值+之前累计编码器的值=总的编码器值 */
    return (int32_t)__HAL_TIM_GET_COUNTER(&g_timx_encode_chy_handle) +
                    g_timx_encode_count * 65536;
}
```

代码中通过调用__HAL_TIM_GET_COUNTER 函数来获取当前的计数值，然后进行编码器计数总值的计算，这里需要用到变量 g_timx_encode_count，具体的计算公式如下：

计数总值 = 当前计数值 + g_timx_encode_count * 65536

获取到计数总值之后，我们就可以计算电机的速度了，本实验中电机速度计算相关源码包括两个文件：dc_motor.c 和 dc_motor.h。

首先看 dc_motor.h 头文件的几个宏定义：

```
#define ROTO_RATIO      44  /* 线数*倍频系数，即 11*4=44 */
#define REDUCTION_RATIO 30 /* 减速比 30:1 */

/* 电机参数结构体 */
typedef struct
{
  uint8_t state;              /*电机状态*/
  float current;              /*电机电流*/
  float volatage;             /*电机电压*/
  float power;                /*电机功率*/
  float speed;                /*电机实际速度*/
  int32_t motor_pwm;          /*设置比较值大小 */
} Motor_TypeDef;

extern Motor_TypeDef  g_motor_data;  /*电机参数变量*/
```

ROTO_RATIO 和 REDUCTION_RATIO 是用于速度计算的系数，ROTO_RATIO 等于编码

器的线数*倍频系数，教程中采用的直流有刷电机编码器线数为 11，也就是说编码器每旋转一圈就输出 11 个脉冲信号；REDUCTION_RATIO 代表电机的减速比，我们算出编码器的速度之后，就可以根据减速比计算电机的速度了。Motor_TypeDef 这个结构体是用于管理电机的参数的，我们本实验中只需要用到 speed（实际速度）这个成员。

下面看 dcmotor_tim.c 的程序，我们只看电机速度计算函数。

```
/**
 * @brief       电机速度计算
 * @param       encode_now：当前编码器总的计数值
 *              ms：计算速度的间隔，中断 1ms 进入一次，例如 ms = 5 即 5ms 计算一次速度
 * @retval      无
 */
void speed_computer(int32_t encode_now, uint8_t ms)
{
    uint8_t i = 0, j = 0;
    float temp = 0.0;
    static uint8_t sp_count = 0, k = 0;
    static float speed_arr[10] = {0.0};     /* 存储速度进行滤波运算 */

    if (sp_count == ms)                     /* 计算一次速度 */
    {
        /* 计算电机转速
           第一步 ：计算 ms 毫秒内计数变化量
           第二步 ：计算 1min 内计数变化量：g_encode.speed * ((1000 / ms) * 60 ，
           第三步 ：除以编码器旋转一圈的计数次数（倍频倍数 * 编码器分辨率）
           第四步 ：除以减速比即可得出电机转速
        */
        g_encode.encode_now = encode_now;    /* 取出编码器当前计数值 */
        /* 计算编码器计数值的变化量 */
        g_encode.speed = (g_encode.encode_now - g_encode.encode_old);
        /* 保存一次电机转速 */
        speed_arr[k++] = (float)(g_encode.speed * ((1000 / ms) * 60.0) /
                         (uint16_t)(REDUCTION_RATIO * ROTO_RATIO));
        /* 保存当前编码器的值 */
        g_encode.encode_old = g_encode.encode_now;

        if (k >= 10)                        /* 累计 10 次速度值，后续进行滤波*/
        {
            for (i = 10; i >= 1; i--)       /* 冒泡排序*/
            {
                for (j = 0; j < (i - 1); j++)
                {
                    if (speed_arr[j] > speed_arr[j + 1])  /* 数值比较 */
                    {
                        temp = speed_arr[j];              /* 数值换位 */
                        speed_arr[j] = speed_arr[j + 1];
                        speed_arr[j + 1] = temp;
                    }
                }
            }
            temp = 0.0;
            for (i = 2; i < 8; i++)         /* 去除两边高低数据 */
            {
                temp += speed_arr[i];       /* 将中间数值累加 */
            }
            temp = (float)(temp / 6);       /* 求速度平均值 */
            /* 一阶低通滤波
             * 公式为：Y(n)= qX(n) + (1-q)Y(n-1)
             * 其中 X(n)为本次采样值；Y(n-1)为上次滤波输出值；Y(n)为本次滤波输出值，
               q 为滤波系数
             * q 值越小则上一次输出对本次输出影响越大，整体曲线越平稳，
               但是对于速度变化的响应也会越慢
```

```
            */
            g_motor_data.speed = (float)((g_motor_data.speed * (float)0.52) +
            ((float)0.48 * temp));
            k = 0;
        }
        sp_count = 0;
    }
    sp_count ++;
}
```

电机的速度计算步骤如下：

第一步，计算 50 毫秒内计数变化量；

第二步，计算 1min 内计数变化量：g_encode.speed * ((1000 / ms) * 60；

第三步，除以编码器旋转一圈的计数次数（REDUCTION_RATIO）；

第四步，除以减速比（ROTO_RATIO）即可得出电机转速；

我们累计计算 10 次电机速度，然后进行冒泡排序，把 10 次电机速度值从小到大排序，接着将中间的 6 次速度值累加求平均值，最后再进行一阶低通滤波。由于冒泡排序和一阶低通滤波的详细介绍篇幅过长，我们这里不做展开，大家感兴趣的话可以去网上搜索相关的内容。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key;
    uint16_t t;
    int32_t motor_pwm = 0;
    uint16_t init_adc_val;
    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    atim_timx_cplm_pwm_init(8400 - 1, 0);       /* 168Mhz 的计数频率 */
    dcmotor_init();                             /* 初始化电机 */
    adc_nch_dma_init();                         /* ADC DMA 传输初始化 */
    gtim_timx_encoder_chy_init(0XFFFF, 0);      /* 不分频直接 84M 的计数频率 */
    btim_timx_int_init(1000 - 1 , 84 - 1);      /* 基本定时器初始化，1ms 计数周期 */

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10, 10, 200, 16, 16, "DcMotor Test", g_point_color);
    lcd_show_string(10, 30, 200, 16, 16, "KEY0:Start forward", g_point_color);
    lcd_show_string(10, 50, 200, 16, 16, "KEY1:Start backward", g_point_color);
    lcd_show_string(10, 70, 200, 16, 16, "KEY2:Stop", g_point_color);

    delay_ms(20);

    /* init_adc_val 存储电流测量对应的参考电压 ADC 值，这里进行滤波 */
    init_adc_val = g_adc_val [2];               /* 取出第一次得到的值 */
    for(t=0;t<1000;t++)
    {
        init_adc_val += g_adc_val [2];          /* 现在的值和上一次存储的值相加 */
        init_adc_val /= 2;                      /* 取平均值 */
        delay_ms(1);
    }

    while (1)
    {
        key = key_scan(0);                      /* 按键扫描 */
        if(key == KEY0_PRES)                    /* 当 key0 按下 */
```

```
        {
            /* 因为不同的电机最小启动电压不同，可能在第一次增加的时候电机还不能转起来 */
            motor_pwm += 400;
            if (motor_pwm == 0)
            {
                dcmotor_stop();                 /* 停止则立刻响应 */
                motor_pwm = 0;
            }
            else
            {
                dcmotor_start();                /* 开启电机 */
                if (motor_pwm >= 8400)          /* 限速 */
                {
                    motor_pwm = 8400;
                }
            }
            motor_pwm_set(motor_pwm);           /* 设置电机方向、转速 */
        }

        else if(key == KEY1_PRES)               /* 当 key1 按下 */
        {
            motor_pwm -= 400;
            if (motor_pwm == 0)
            {
                dcmotor_stop();                 /* 停止则立刻响应 */
                motor_pwm = 0;
            }
            else
            {
                dcmotor_start();                /* 开启电机 */
                if (motor_pwm <= -8400)         /* 限速 */
                {
                    motor_pwm = -8400;
                }
            }
            motor_pwm_set(motor_pwm);           /* 设置电机方向、转速 */
        }

        else if(key == KEY2_PRES)               /* 当 key2 按下 */
        {
            LED1_TOGGLE();
            dcmotor_stop();                     /* 关闭电机 */
            motor_pwm = 0;
            motor_pwm_set(motor_pwm);           /*设置电机方向、转速 */
        }

        delay_ms(10);
        t++;
        if(t % 20 == 0)
        {
            lcd_dis();                          /* 显示电机速度信息 */
            LED0_TOGGLE();                      /* LED0(红灯) 翻转 */

            printf("KEY0: 增加比较值，KEY1: 减小比较值，KEY2: 停止电机\r\n\r\n");
            printf("Valtage:%.1fV \r\n",g_adc_val[0]*ADC2VBUS);    /* 打印电压值 */
            printf("Temp:%.1fC \r\n",get_temp(g_adc_val[1]));      /* 打印温度值 */
            printf("Current:%.1fmA \r\n",abs(g_adc_val[2]-
                                    init_adc_val)*ADC2CURT);       /* 打印电流值 */
            printf("电机速度: %.1f RPM\r\n\r\n",g_motor_data.speed);/* 打印速度值 */
        }
    }
}
```

我们只看本实验新增的内容，先看 gtim_timx_encoder_chy_init(0XFFFF, 0)这个语句，这两

个形参分别设置自动重载寄存器的值为 65535，以及预分频器寄存器的值为 0。定时器 3 是 16 位的计数器，这里设置为最大值 65535。预分频系数，我们设置为不分频，定时器 3 的计数频率是 84MHz。接着看 btim_timx_int_init(1000 - 1 , 84 - 1)这个语句，这里设置了定时器 6 的计数频率为 84M/84=1M Hz，计数周期为 1ms，也就是 1ms 进入一次更新中断。最后就是在串口以及屏幕上输出电机速度值了。

7.3.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 上显示按键功能以及电机速度信息，当我们按下 KEY0，比较值变量 motor_pwm 将增大；按下 KEY1，比较值变量 motor_pwm 将减小；按下 KEY2，电机将停止。比较值变量 motor_pwm 为正数时，电机正转，反之电机反转，其绝对值越大，电机的速度越快。我们再打开串口调试助手，选择对应的串口端口，我这边是 COM3，可以看到串口打印的按键信息、电压、温度、电流以及电机速度值，如图 7.3.4.1 所示：

图 7.3.4.1 打印按键信息、电压、温度、电流以及速度值

第 8 章 PID 算法

本章我们主要来学习 PID 算法，它是自动控制算法中的经典之作，被广泛应用于温度控制、水位控制、飞行姿态控制等领域。

本章分为如下几个小节：

8.1 自动控制算法

8.2 PID 算法入门

8.3 PID 算法代码实现

8.1 自动控制系统

在直流有刷电机的基础驱动实验中，如果电机负载不变，我们只要设置固定的占空比（电压），电机的速度就会稳定在目标范围。然而，在实际的应用中，负载可能会发生变化，此时如果还是输出固定的电压，电机的速度就偏离目标范围了，为了解决这个问题，我们需要引入自动控制系统中的闭环控制。接下来我们开始学习自动控制系统的内容。

1）概念：用自动控制装置，对关键参数进行自动控制，使它在受到外界干扰而偏离正常状态时，能够被自动地调节回到目标范围内。

2）应用场景：电水壶保温系统、大棚温控系统、水位控制系统，等等。

3）分类：自动控制系统分为开环控制系统和闭环控制系统 。

① 开环控制系统

在开环控制系统中，系统输出只受输入的控制，没有反馈回路，控制精度和抑制干扰的特性都比较差。开环控制系统的实例如图 8.1.1 所示：

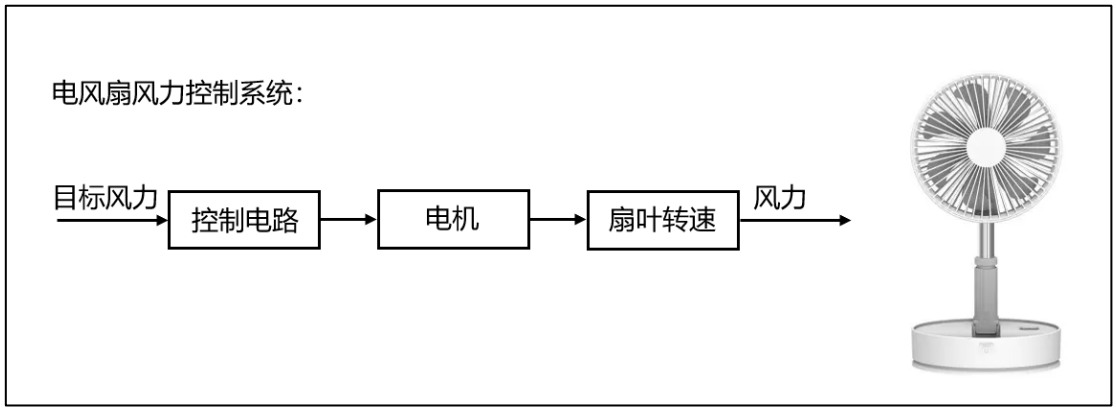


图 8.1.1.开环控制系统实例

图 8.1.1 中，电风扇风力控制系统就是一个开环控制的系统，我们设置好目标风力之后，控制电路就输出相应的电压（假设是电压控制），此时电机的扇叶转速就被控制在目标范围了。理想状态下，风扇的输出风力确实可以稳定在目标值附近，然而，在实际的使用中，电机会逐渐老化，扇叶上的灰尘也会让负载增大，此时我们所设定目标风力和实际风力可能就存在偏差了。

②闭环控制系统

在闭环控制系统中，引入了反馈回路，利用输出（实际值）和输入（目标值）的偏差，对系统进行控制，避免偏离预定目标。闭环控制系统又称反馈控制系统，它的实例如图 8.1.2 所示：

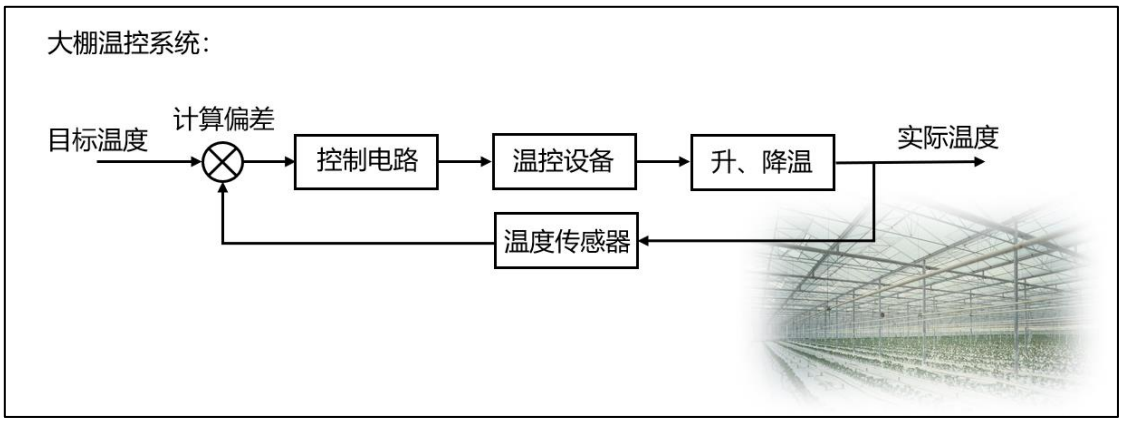


图 8.1.2 闭环控制系统实例

图 8.1.2 中，大棚温控系统就是一个闭环控制的系统，我们设置好目标温度之后，温度传感器会采集棚内的实际温度，然后将目标温度和实际温度进行偏差的计算，计算后的结果输入到控制电路中，控制电路进一步控制温控设备进行升温和降温，此时棚内的实际温度就被控制在目标范围了。当实际温度因外部影响偏离目标值时，温度传感器（反馈电路）就能及时的反馈偏差，让系统自动调节温控设备，使得实际温度逐渐回到目标范围。

通过这个实例，大家就可以感受到闭环控制系统的意义所在，如果我们把上述实例中的控制对象变成电机的速度，反馈电路变成编码器测速电路，即可实现电机速度的闭环控制。

8.2 PID 算法入门

PID 算法是闭环控制系统中常用的算法，PID 分别是 Proportion（比例）、Integral（积分）、Differential（微分）的首字母缩写。它是一种结合比例、积分和微分三个环节于一体的闭环控制算法，具体的控制流程如图 8.2.1 所示：

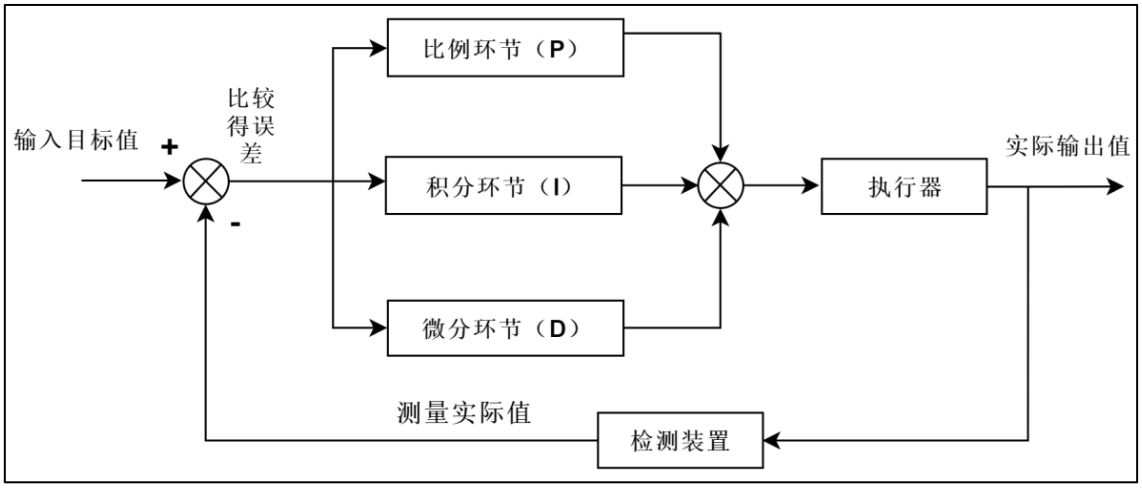


图 8.2.1 PID 控制流程

图 8.2.1 中，我们将输入目标值和实际输出值进行偏差的计算，然后把计算结果输入到 PID 控制算法中，经过比例、积分和微分三个环节的运算，运算后的输出作用于执行器，从而让系统的实际值逐渐靠近目标值。

接下来我们以大棚温控系统为例，来理解 PID 算法中三个环节的作用。

8.2.1 比例环节（ Proportion）

比例环节可以成比例地反应控制系统的偏差信号，即输出与输入偏差成正比，可以用来减小系统的偏差。此环节的公式如下：

$$u = Kp \cdot e$$

u ----- 输出

Kp ----- 比例系数

e ----- 偏差

我们接下来以大棚温控实例来理解这个公式。假设我们现在需要调节棚内温度为 30℃，而实际温度为 10℃，此时的偏差 e=20，由比例环节的公式可知，当 e 确定时，Kp 越大则输出 u 越大，也就是温控系统的调节力度越大，这样就可以更快地达到目标温度；而当 Kp 确定时，偏差 e 越大则输出 u 越大。由此可见，在比例环节中，比例系数 Kp 和偏差 e 越大则系统消除偏差的时间越短，具体的曲线表现如图 8.2.1.1 所示：

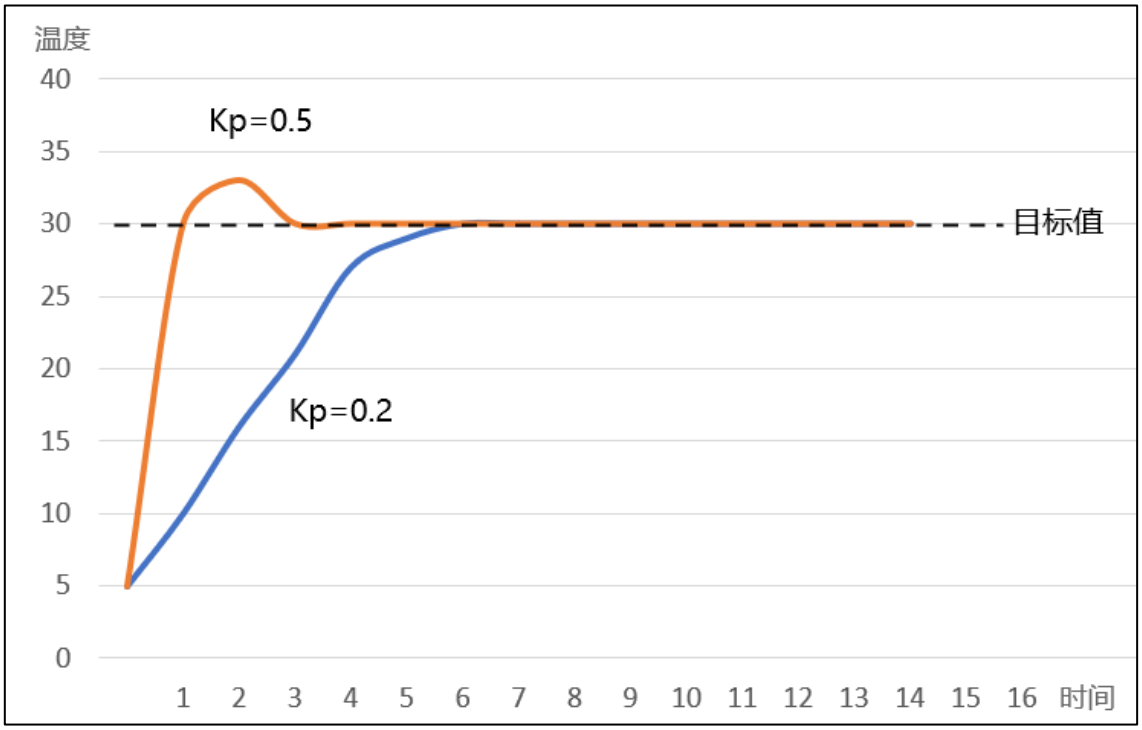


图 8.2.1.1 比例环节曲线图

图 8.2.1.1 中，当 Kp 的值越大时，其对应的橙色曲线达到目标值的时间就越短，与此同时，橙色曲线出现了一定幅度的超调和振荡，这会使得系统的稳定性下降，因此，我们在设置比例系数的时候，并不是越大越好，而是要兼顾消除偏差的时间以及整个系统的稳定性。

然而，在实际的应用中，如果仅有比例环节的控制，可能会给系统带来一个问题：**静态误差**。静态误差是指系统控制过程趋于稳定时，目标值与实测值之间的偏差。我们以大棚温控实例来理解什么是静态误差：

假设我们现在需要调节棚内温度为 30℃，而实际温度为 25℃，此时偏差 e=5，Kp 为固定值，如果此时的输出可以让大棚在半个小时之内升温 5℃，而外部的温差可以让大棚在半个小时之内降温 5℃，也就是说，输出 u 的作用刚好被外部影响抵消了，这就使得偏差会一直存在，此时的曲线如图 8.2.1.2 所示：

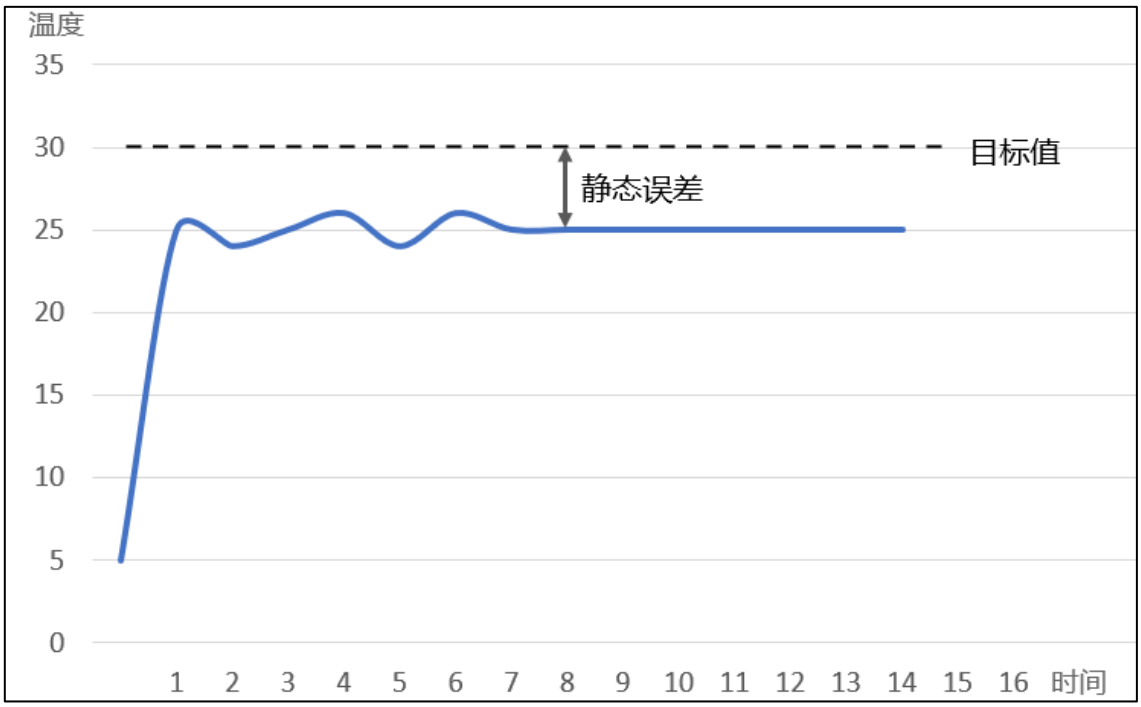


图 8.2.1.2 存在静态误差

大家可能会想到，我们可以通过增大 Kp 来增大输出，以此消除偏差。在实际应用中，此方法的局限性很大，因为我们不能确定偏差的大小，它是在实时变化的，如果我们把 Kp 设置得太大，就会引入超调和振荡，让整个系统的稳定性变差。因此，为了消除静态误差，我们引入了积分环节。

8.2.2 积分环节（Integral）

积分环节可以对偏差 e 进行积分，只要存在偏差，积分环节就会不断起作用，主要用于消除静态误差，提高系统的无差度。引入积分环节后，比例+积分环节的公式如下：

$$u = Kp \cdot \mathrm{e} + Ki \cdot \sum \mathrm{e}$$

u ----- 输出

e ----- 偏差

$\sum e$ ----- 累计偏差

Kp ----- 比例系数

Ki ----- 积分系数

我们依旧以大棚温控实例来理解积分环节的作用。假设现在温控系统的比例环节作用被抵消，存在静态误差 5℃，此时偏差存在，积分环节会一直累计偏差，以此增大输出，从而消除静态误差。从上述公式中可以得知，当积分系数 Ki 或者累计偏差越大时，输出就越大，系统消除静态误差的时间就越短，具体的曲线表现如图 8.2.2.1 所示：

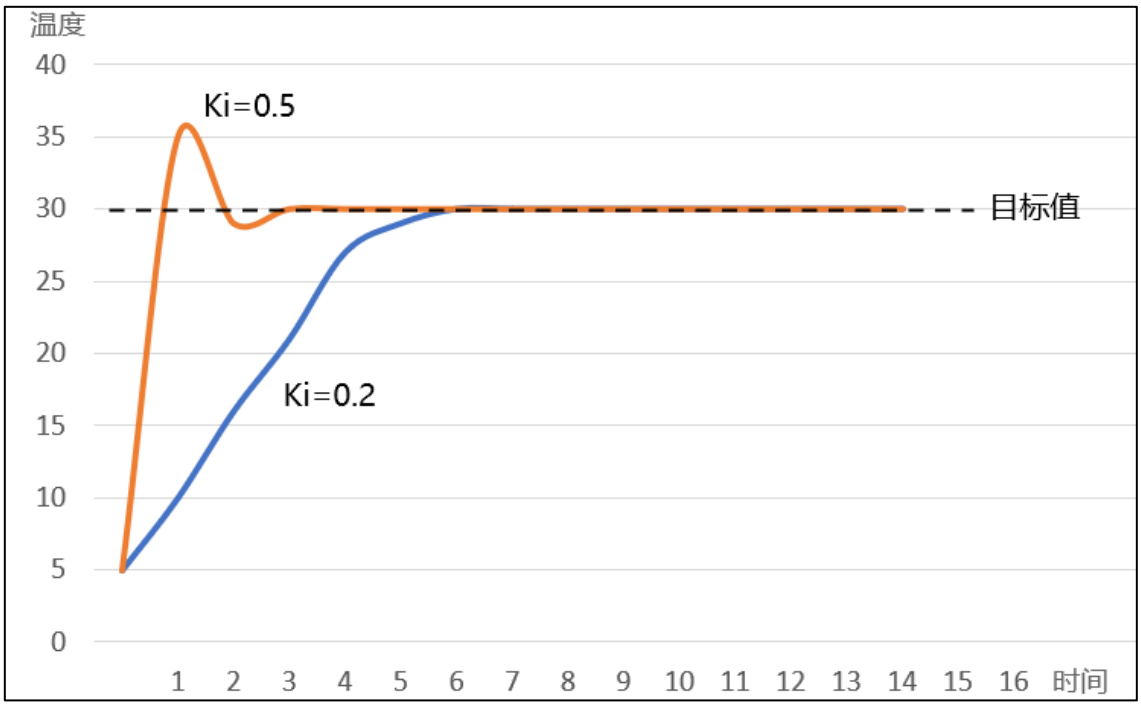


图 8.2.2.1 比例+积分环节曲线图

图 8.2.2.1 中，当 Ki 的值越大时，其对应的橙色曲线达到目标值的时间就越短，与此同时，橙色曲线出现了一定幅度的超调和振荡，这会使得系统的稳定性下降，因此，我们在设置积分系数的时候，并不是越大越好，而是要兼顾消除静态误差的时间以及整个系统的稳定性。

我们前面有说过，只要系统还存在偏差，积分环节就会不断地累计偏差。当系统偏差为 0 的时候，说明已经达到目标值，此时的累计偏差不再变化，但是积分环节依旧在发挥作用（此时往往作用最大），这就很容易产生超调的现象了。因此，我们需要引入微分环节，提前减弱输出，抑制超调的发生。

8.2.3 微分环节（Differential）

微分环节可以反应偏差量的变化趋势，根据偏差的变化量提前作出相应控制，减小超调，克服振荡。引入微分环节后，比例+积分+微分环节的公式如下：

$$u_k = Kp \cdot e_k + Ki\sum_{j=0}^{k} e_j + Kd(e_k - e_{k-1})$$

u_k ---------- 第 k 次输出
Kp ---------- 比例系数
Ki ---------- 积分系数
Kd ---------- 微分系数
e_k ---------- 第 k 次偏差
e_{k-1} ---------- 第 k-1 次偏差
$\sum_{j=0}^{k} e_j$ ---------- 累计 k 次偏差

我们依旧以大棚温控实例来理解微分环节的作用。假设温控系统目标温度为 30℃，在上午八点的时候存在偏差 15℃，经过一段时间的调节，到了上午九点，此时偏差已经缩小到 5℃，偏差的变化量= 九点的偏差（第 k 次）−八点的偏差（第 k-1 次）= −10，结合上述公式可知，此时微分环节会削弱比例和积分环节的作用，减小输出以抑制超调。

至此，PID 算法三个环节的作用已经介绍完，我们最终得到了 PID 算法的一个公式：

$$u_k = Kp \cdot e_k + Ki\sum_{j=0}^{k} e_j + Kd(e_k - e_{k-1})$$

该公式是 PID 离散公式之一，除了离散公式之外，PID 还有连续的公式，但是因为连续的公式不利于机器计算，我们这里不做介绍。大家需要注意，在实际的应用中，并不是每一个系统都需要 PID 的三个环节参与控制的，有的系统只需要比例环节或积分环节就可以控制得很好，除此之外，每一个系统的 PID 系数并不是通用的，这需要根据实际的情况去设置。

8.2.4 PID 算法离散公式

1）位置式 PID 公式

$$u_k = Kp \cdot e_k + Ki\sum_{j=0}^{k} e_j + Kd(e_k - e_{k-1})$$

u_k ---------- 第 k 次输出
Kp ---------- 比例系数
Ki ---------- 积分系数
Kd ---------- 微分系数
e_k ---------- 第 k 次偏差
e_{k-1} ---------- 第 k-1 次偏差
$\sum_{j=0}^{k} e_j$ ---------- 累计 k 次偏差

我们在上一小节中得到的 PID 离散公式称为位置式或全量式 PID 公式，该公式的计算需要全部控制量参与，它的每一次输出都和过去的状态有关。

2）增量式 PID 公式

通过位置式的 PID 公式，我们只需两步即可推导出增量式 PID 公式：

第一步，将 k = k-1 代入位置式 PID 公式，得：

$$u_{k-1} = Kp \cdot e_{k-1} + Ki\sum_{j=0}^{k-1} e_j + Kd(e_{k-1} - e_{k-2})$$

第二步，由u_k -u_{k-1} 得：

$$\Delta u_k = Kp(e_k - e_{k-1}) + Ki \cdot e_k + Kd(e_k - 2e_{k-1} + e_{k-2})$$

至此，增量式 PID 方式就推导完了。从公式中可以看出，增量式 PID 的计算并不需要一直累计偏差，它的输出与近三次的偏差有很大关系。

注意：增量式 PID 公式输出的只是控制量的增量。我们以一个实例来理解这句话：假设电机实际转速为 50RPM，现在我们要让它加速到 60RPM，如果采用的是位置式 PID，系统将直接输出 60RPM 对应的控制量（占空比）；如果采用的是增量式 PID，系统将输出提速 10RPM 对应的控制量（占空比），此时我们还需要加上上次（50RPM）的输出。

介绍完 PID 的两个离散化公式，接下来我们比较一下这两个 PID 公式的不同点，具体如表 8.2.4.1 所示：

对比项	位置式	增量式
输出	全量输出	仅输出增量

偏差	需要一直累计偏差	只考虑最近 3 次偏差
积分作用	有	无
限幅	需要进行输出和积分限幅	只需要进行输出限幅

表 8.2.4.1 两种 PID 公式对比

3）两种 PID 公式的优缺点

① 位置式：

优点：位置式 PID 是一种非递推式算法，带有积分作用，适用于不带积分部件的对象。

缺点：全量计算，计算错误影响很大；需要对偏差进行累加，运算量大。

② 增量式：

优点：只输出增量，计算错误影响小；不需要累计偏差，运算量少，实时性相对较好。

缺点：积分截断效应大，有稳态误差。

8.3 积分饱和问题

在位置式 PID 中，如果系统长时间无法达到目标值，累计偏差（积分）就会变得很大，此时系统的响应就很慢了。我们以一个实例来分析这个问题：

假设某个电机能达到的最大速度为 300RPM，而我们设置了目标速度为 350RPM，这明显是一个不合理的目标值，此时 PID 控制系统的曲线如图 8.3.1 所示：

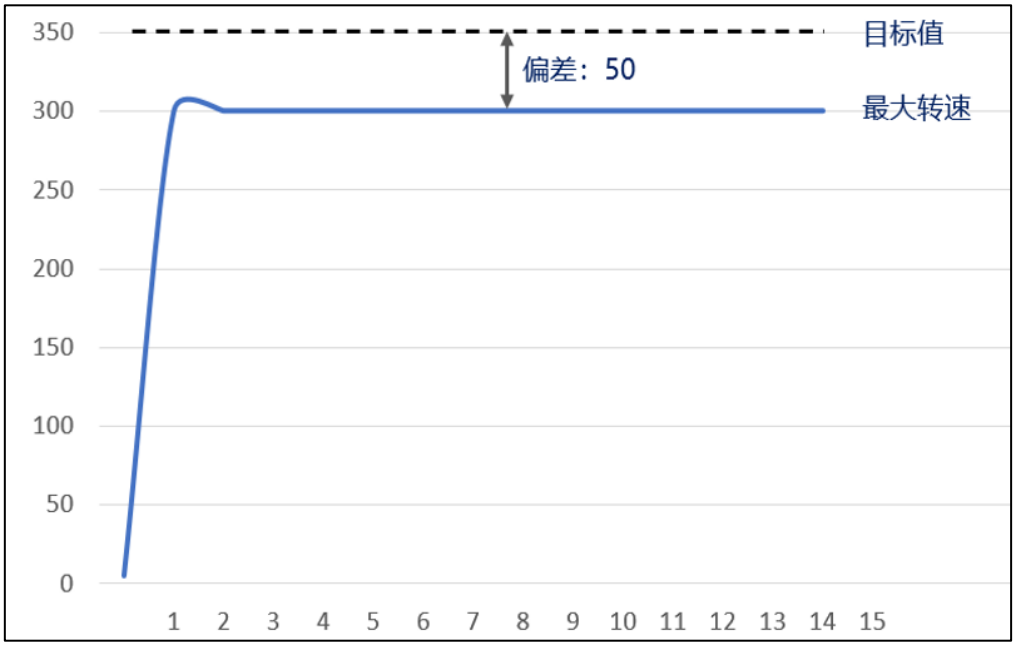


图 8.3.1 不合理目标值

图 8.3.1 中，由于系统长时间无法达到目标值，累计偏差（积分）会变得越来越大，逐渐达到深度饱和的状态，此时我们再设置一个合理范围的目标速度（例如 200RPM），系统就没有办法在短时间内响应了。

为了避免位置式 PID 中可能出现的积分饱和问题，可采取以下措施：

① 优化 PID 曲线，系统越快达到目标值，累计的偏差就越小；

② 限制目标值调节范围，规避可以预见的偏差；

③ 进行积分限幅，在调整好 PID 系数之后，根据实际系统来选择限幅范围。

8.4 PID 算法代码实现

1）控制量相关的结构体

有了 PID 的离散化公式之后，我们实现 PID 算法的代码是非常简单的，这里先回顾一下两个 PID 公式所涉及到的所有控制量：

u_k 、Δu_k ---- 期望输出值（计算后的输出）

Kp ---------- 比例系数

Ki ---------- 积分系数

Kd ---------- 微分系数

e_k ---------- 第 k 次偏差

e_{k-1} ---------- 第 k-1 次偏差

e_{k-2} ---------- 第 k-2 次偏差

$\sum_{j=0}^{k} e_j$ -------- 累计偏差

在代码中，我们可以通过一个结构体来管理这些控制量，具体定义如下：

```
typedef struct
{
    __IO float  SetPoint;           /* 目标值 */
    __IO float  ActualValue;        /* 期望输出值 */
    __IO float  SumError;           /* 偏差累计 */
    __IO float  Proportion;         /* 比例系数 P */
    __IO float  Integral;           /* 积分系数 I */
    __IO float  Derivative;         /* 微分系数 D */
    __IO float  Error;              /* Error[1],第 k 次偏差 */
    __IO float  LastError;          /* Error[-1],第 k-1 次偏差 */
    __IO float  PrevError;          /* Error[-2],第 k-2 次偏差 */
} PID_TypeDef;
```

这个结构体当中的每一个成员所代表的控制量在注释中已经很清楚，这里就不再介绍。值得注意的是，该结构体中多了一个目标值的控制量，它将会用于偏差的计算。

2）PID 算法代码

PID 算法的代码实现本质上就是对于两个离散公式的应用，我们先看位置式 PID 代码：

```
/*
 * @brief       pid 闭环控制
 * @param       *PID: PID 结构体变量地址
 * @param       Feedback_value: 当前实际值
 * @retval      期望输出值
 */
int32_t increment_pid_ctrl(PID_TypeDef *PID,float Feedback_value)
{
    PID->Error = (float)(PID->SetPoint - Feedback_value);    /* 计算偏差 */
    PID->SumError += PID->Error;                             /* 累计偏差 */
    PID->ActualValue = (PID->Proportion * PID->Error)        /* 比例环节 */
                     + (PID->Integral * PID->SumError)       /* 积分环节 */
       + (PID->Derivative * (PID->Error - PID->LastError)); /* 微分环节 */
    PID->LastError = PID->Error;                 /* 存储偏差，用于下次计算 */
    return ((int32_t)(PID->ActualValue));        /* 返回计算后输出的数值 */
}
```

increment_pid_ctrl 函数用来进行位置式 PID 的控制，该函数的 2 个形参：*PID 传入 PID 控制量相关的结构体地址；Feedback_value 传入当前系统的实际值，用于计算偏差。在函数中，我们先计算本次偏差 Error，然后把偏差累计，存入 SumError 成员当中，接着根据位置式的公式进行三个环节的计算，计算后的期望输出存入 ActualValue 成员当中，然后存储本次偏差，最后返回期望输出值。

接着我们看增量式 PID 代码：

```
/*
 * @brief       pid 闭环控制
 * @param       *PID: PID 结构体变量地址
 * @param       Feedback_value: 当前实际值
```

```
 * @retval      期望输出值
 */
int32_t increment_pid_ctrl(PID_TypeDef *PID,float Feedback_value)
{
    PID->Error = (float)(PID->SetPoint - Feedback_value);    /* 计算偏差 */

    PID->ActualValue +=
    /* 比例环节 */
    (PID->Proportion * (PID->Error - PID->LastError))
    /* 积分环节 */
     + (PID->Integral * PID->Error)
    /* 微分环节 */
     + (PID->Derivative * (PID->Error - 2 * PID->LastError + PID->PrevError));

    PID->PrevError = PID->LastError;              /* 存储偏差，用于下次计算 */
    PID->LastError = PID->Error;
    return ((int32_t)(PID->ActualValue));        /* 返回计算后输出的数值 */
}
```

增量式 PID 的代码实现和位置式是非常类似的，所以我们在实际的代码实现中，可以通过一个宏定义来切换这两种不同的算法，值得注意的是，增量式 PID 输出的是调节量，所以计算期望输出值 ActualValue 的时候是自增运算，这一点和位置式 PID 是不一样的。

第 9 章 PID 参数整定

本章我们主要来学习 PID 算法的参数整定，在上一章节中我们有提到，每一个系统的 PID 系数并不是通用的，在不同的系统中运用同样的 PID 系数，其最终所体现的效果可能是相差甚远的，因此，我们需要根据实际的系统进行 PID 的参数整定（调参）。

本章分为如下几个小节：

9.1 采样周期选择
9.2 PID 参数整定
9.3 正点原子 PID 上位机介绍

9.1 采样周期选择

采样周期指的是 PID 控制中实际值的采样时间间隔，其越短，效果越趋于连续，但对硬件资源的占用也越高。在实际的应用中，我们可以使用理论或者经验方法来确定采样周期：

① 理论方法：香农采样定理。这个定理可以用来确定采样周期可选择的最大值，当采样周期超出了这个最大的允许范围，我们所得到的信号就会失真，也就无法较好地还原信号了。香农采样定理的具体原理我们不展开介绍，感兴趣的朋友可以去查找相关的资料，我们这里重点关注经验方法。

② 经验方法：根据控制对象突变能力选择。假设电机当前转速为 20RPM，我们需要提高它的转速到 30RPM，此电机的转速在 1s 之内最大可以突变 10RPM（即电机速度的突变能力），如果我们每 1ms 采集一次电机转速，那么每一次采集到的速度变化量最大为 10RPM / 1000 = 0.01RPM，很明显，此时最大的变化量远远小于当前的速度，这对于我们的 PID 控制效果并没有明显的提升，但是却占用了很多的硬件资源，因此，我们需要根据控制对象的突变能力来选择采样周期。

9.2 PID 参数整定方法

PID 参数整定的方法很多，概括起来有两大类，如表 9.2.1 所示：

理论计算整定法	工程整定法
数学模型	试凑法、临界比例法、一般调节法

表 9.2.1 PID 参数整定方法

理论计算整定法：依据系统的数学模型，经过理论计算确定 PID 参数。这种方法是建立在理想化条件下的，其得到的参数不一定能够直接使用，还需要结合经验以及实际的系统进行调整。

工程整定法：依靠工程经验，直接在控制系统的试验中进行整定，此方法易于掌握，在实际调参中被广泛采用。工程整定法包括：试凑法、临界比例法和一般调节法。

注意：无论采用哪一种方法所得到的 PID 参数，都需要在实际运行中进行最后调整与完善，因此，在 PID 参数整定中，最重要的就是**经验**的积累。

接下来我们了解一下 PID 各个系数调节的效果，这样才能做到在 PID 调参中有的放矢。

① 比例系数：调节作用快，系统一出现偏差，调节器立即将偏差放大输出。

② 积分系数：积分系数的调节会改变输入偏差对于系统输出的影响程度。积分系数越大，消除静差的时间越短，但是过大的积分系数则会导致系统出现超调现象，这在具有惯性的系统中尤为明显。

③ 微分系数：微分系数的调节是偏差变化量对于系统输出的影响程度。微分系数越大，系统对于偏差量的变化越敏感，越能提前响应，进而抑制超调，但是过大的微分系数则会让整个系统出现振荡。

9.2.1 试凑法

1）内容：结合系统的具体情况以及经验，先试凑几组合理的 PID 系数，同时需要观察系统的曲线变化，确定每一个系数对于整个系统曲线的大致影响，然后再根据具体的曲线进行调整。

2）调节思路：

① 先是比例（P），再积分（I），最后是微分（D）；

② 按纯比例系统整定比例系数，使其得到比较理想的调节过程曲线，然后再把比例系数缩小 1.2 倍左右，将积分系数从小到大改变，使其得到较好的调节过程曲线；

③ 在这个积分系数下重新改变比例系数，再看调节过程曲线有无改善；

④ 如有改善，可将原整定的比例系数减少，改变积分系数，这样多次的反复，就可得到合适的比例系数和积分系数；

⑤ 如果存在外界的干扰，系统的稳定性不好，可把比例、积分系数适当减小，使系统足够稳定；

⑥ 如果系统存在小幅度超调，可以将整定好的比例系数和积分系数适当减小，增大微分系数，以得到超调量最小、调节作用时间最短的系统曲线；

9.2.2 临界比例法

1）内容：在闭环的控制系统里，将调节器置于纯比例作用下，从小到大逐渐调节比例系数，直到系统曲线出现等幅振荡，再根据经验公式计算参数。

2）调节思路：

① 将积分、微分系数置零，比例度取适当值，平衡操作一段时间，使控制系统按纯比例作用的方式投入运行。

② 慢慢地增大比例系数，细心观察曲线的变化情况。如果控制过程的曲线波动是衰减的，则把比例系数继续增大；如果曲线波动是发散的，则应把比例系数减小，直至曲线波动呈等幅振荡，此时记下临界比例系数 δK 和临界振荡周期 Tk 的值。

③ 根据记下的比例系数和周期，采用经验公式，计算调节器的参数。

参与控制的环节	Kp	Ki	Kd
P	δK/2	0	0
PI	δK/2.2	Kp/（0.833 • Tk）	0
PID	δK/1.7	Kp/（0.5 • Tk）	0.125 • Tk • Kp

表 9.2.2.1 临界比例法参数表

9.2.3 一般调节法

1）内容：这种方法针对一般的 PID 控制系统所以称之为一般调节法。

2）调节思路：

① 首先将积分、微分系数置零，使系统为纯比例控制。控制对象的值设定为系统允许的最大值的 60%~70%，接着逐渐增大比例系数，直至系统出现振荡；此时再逐渐减小比例系数，直至系统振荡消失，然后记录此时的比例系数，并设定系统的比例系数为当前值的 60%~70%。

② 确定比例系数后，设定一个较小的积分系数，然后逐渐增大积分系数，直至系统出现振荡；此时在逐渐减小积分系数，直至系统振荡消失，然后记录此时的积分系数，并设定系统的积分系数为当前值的 55%~65%。

③ 微分系数一般不用设定，为 0 即可。若系统出现小幅度振荡，并且通过 PI 环节无法优化，这可以采用与确定比例、积分系数相同的方法，微分系数取系统不振荡时的 30%左右。

④ 系统空载、带载联调，再对 PID 参数进行微调，直至满足要求。

9.2.4 实际调参演示

接下来我们将从实际的 PID 系统曲线来理解 PID 各个系数的调节效果，大家一定要注意经验的积累。我们调节 PID 系数的思路如下：

① **先调整比例系数，积分、微分系数设置为 0**，此时的系统只有比例环节参与控制。如果此时系统的曲线出现大幅振荡，如图 9.2.4.1 所示：

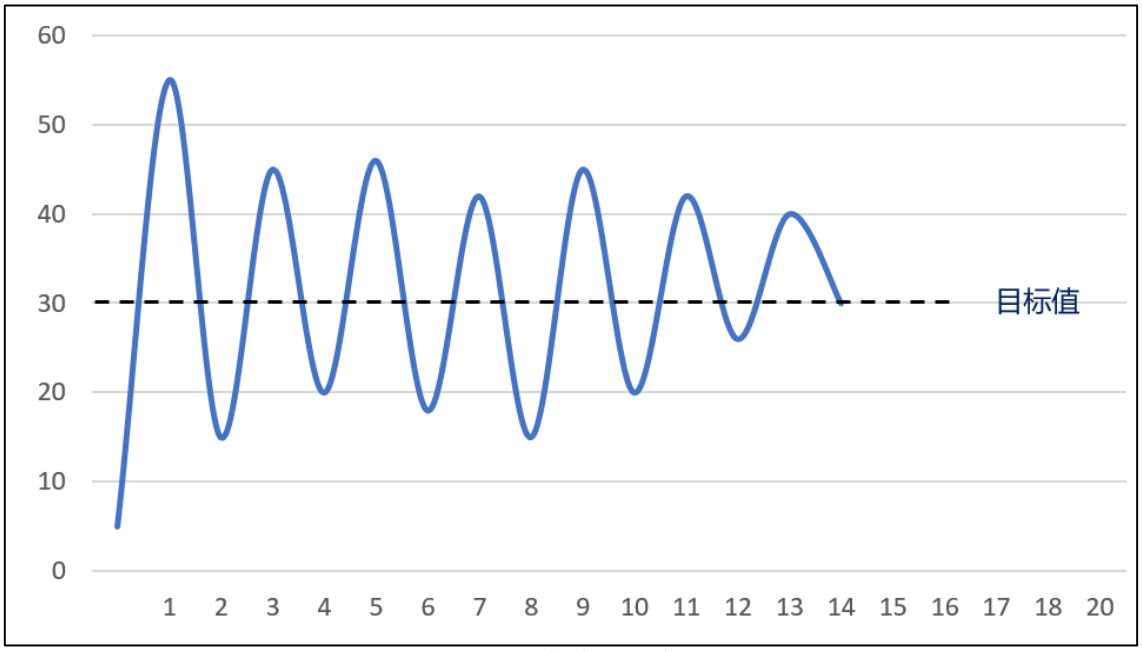


图 9.2.4.1 PID 曲线出现大幅振荡

首先确定硬件上是否出现了故障，例如电压不稳定、电机堵转等，排除了这些之后，那就说明比例系数调节的过大了，这个时候我们可以把比例系数慢慢地减小，并同时观察曲线的变化。

② 当我们调小比例系数之后，曲线的大幅度振荡现象消失，但是曲线依旧存在小幅度的超调现象，并且此时通过调节比例系数已经无法优化曲线，如图 9.2.4.2 所示：

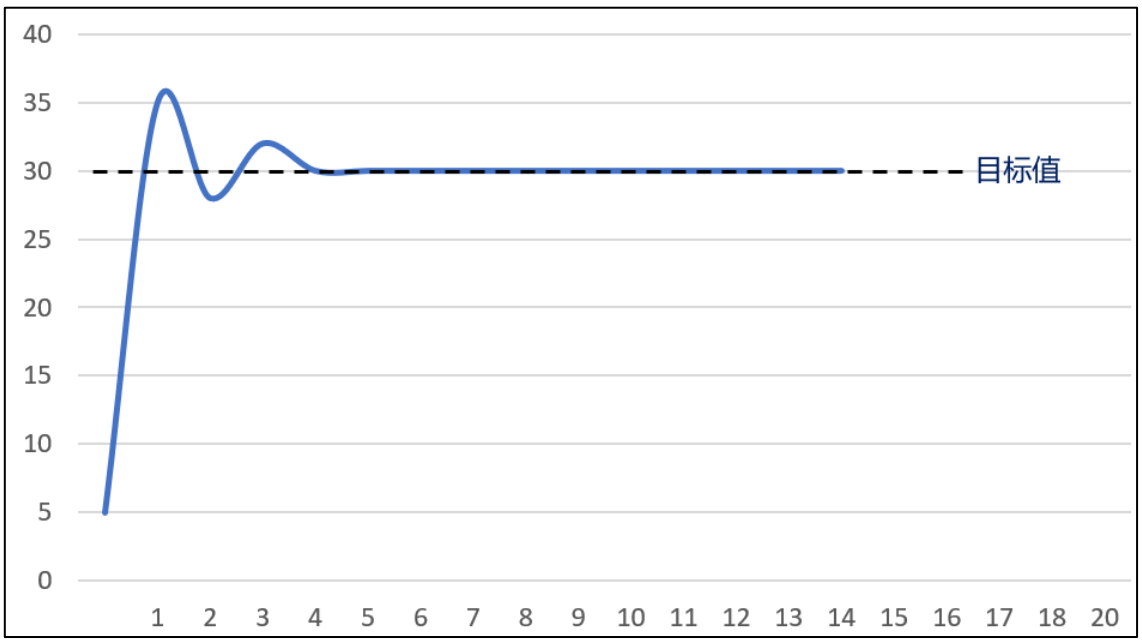


图 9.2.4.2 PID 曲线存在小幅度超调

此时，我们可以慢慢地增大微分系数，并同时观察曲线的变化，从而找到最合适的参数。增大微分系数之后，如果系统的曲线已经较为理想，则说明这个系统只需要比例和微分环节

的控制。

③ 如果在纯比例环节的控制下，系统的实际值始终达不到目标值，存在静态误差，如图 9.2.4.3 所示：

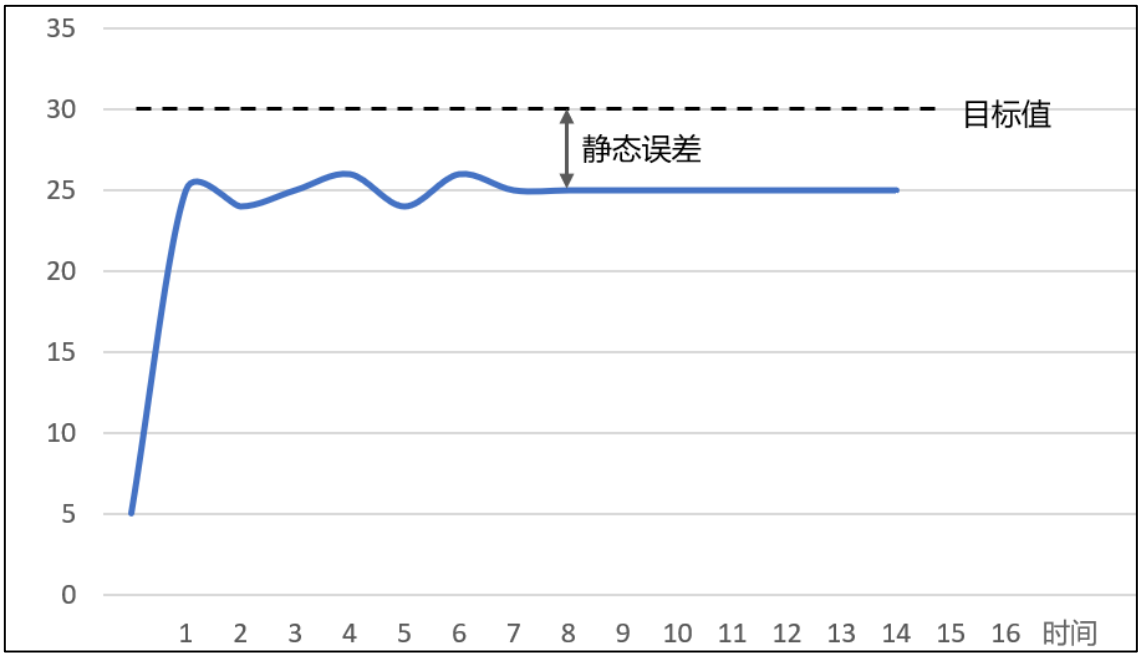


图 9.2.4.3 系统存在静态误差

此时，我们可以逐渐增大积分系数，并同时观察曲线的变化，如果消除静差的时间过长，则可以再适当增大积分系数，但是需要注意兼顾系统的超调量。经过调整之后，如果系统的曲线已经较为理想，则说明这个系统只需要比例和积分环节的控制。

④ 如果系统在比例和积分环节的控制下出现小幅度的超调现象，此时，我们可以慢慢地增大微分系数，并同时观察曲线的变化，从而找到最合适的参数。

以上就是在实际调参中经常遇到的一些问题以及解决方法。然而，在实际应用中，控制系统是多样且复杂的，上述方法只能作为参考，并不是通用的，因此在 PID 调参过程中，大家一定要注意经验的积累。

9.3 正点原子 PID 上位机介绍

为了方便大家调试 PID 参数以及控制电机，我们开发了 PID 调试助手上位机，其采用串口通信，支持多种波特率，具有以下功能：

① 16 路波形实时显示；设备状态以及故障显示。
② 总里程、设备位置、电机类型显示。
③ 支持 10 组 PID 参数调节，可以自定义参数调节范围。
③ 支持多种电机控制指令。

大家可以打开 PID 调试助手来查看它的界面，软件的路径：A 盘→6，软件资料→1，软件→6，PID 调试助手→ATK_PID.exe。

9.3.1 上位机通信协议

上位机和下位机（开发板）之间使用整帧传输的方式进行通信，数据帧采用 CRC16 的校验方式，下面我们简单介绍一下数据帧的格式，如下表 9.3.1.1 所示：

帧头	数据类别	数据域	校验和	帧尾
0xC5	1 字节	0~32 字节	2 字节	0x5C

表 9.3.1.1 数据帧格式

① 帧头：一包数据的头部，固定为 0xC5，长度为 1 个字节。

② 数据类别：数据的属性，例如电机速度，温度等，长度为 1 个字节。

③ 数据域：某个数据类别的值，长度为 0~32 个字节。

④ 校验和：采用 CRC16-MODBUS 校验，校验范围包括帧头、数据类别和数据域，长度为 2 个字节。

④ 帧尾：一包数据的尾部，固定为 0x5C，长度为 1 个字节。

具体的协议内容可查阅《PID 调试助手通信协议.pdf》这个文档，路径：A 盘→6，软件资料→1，软件→6，PID 调试助手→《PID 调试助手通信协议.pdf》。

注意：通信协议的内容不要求熟记，大家只需要了解数据帧的格式即可，当我们需要用到某个功能的时候，再去查阅相关的协议说明。

9.3.2 上位机使用方法

① 双击 ATK_PID.exe 即可打开上位机，上位机主界面如图 9.3.2.1 所示：

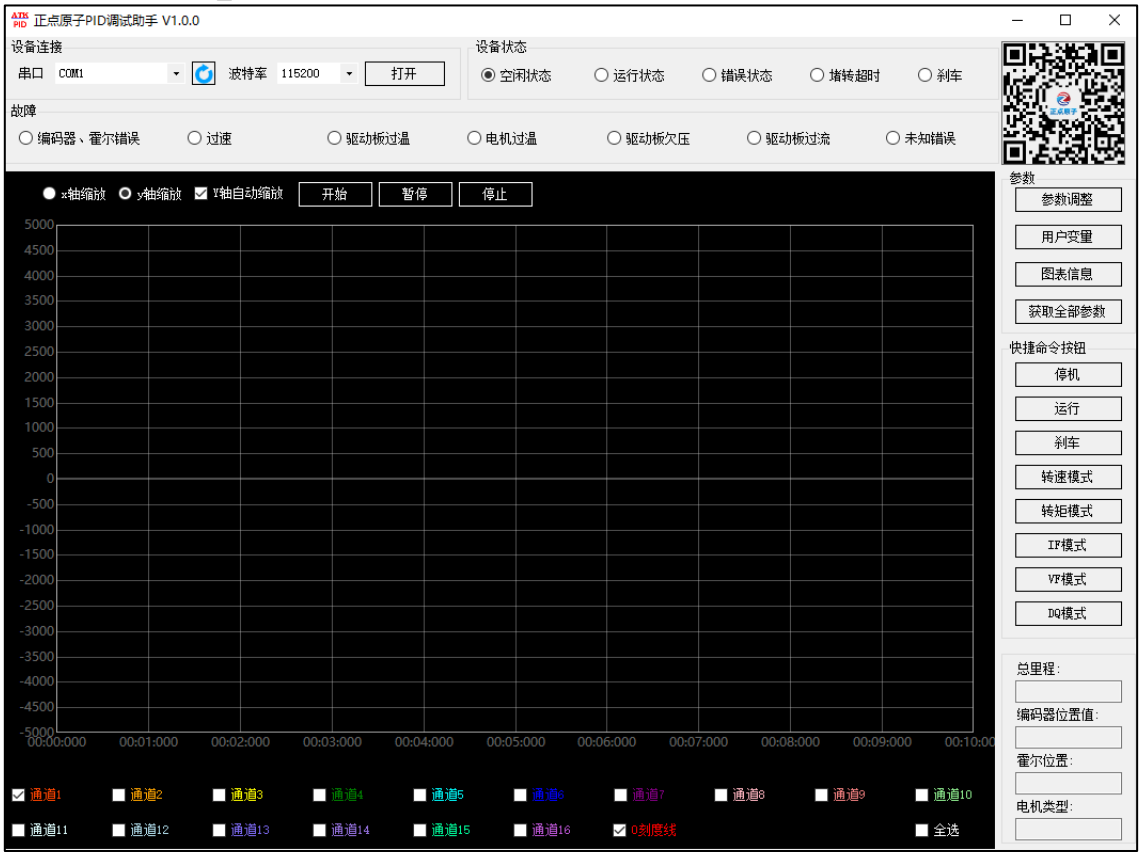


图 9.3.2.1 上位机主界面

② 在**设备连接**处选择下位机对应的 COM 口、波特率，点击“打开”按钮，如果没有显示 COM 口，可以尝试点击蓝色的刷新图标进行刷新，界面如图 9.3.2.2 所示：

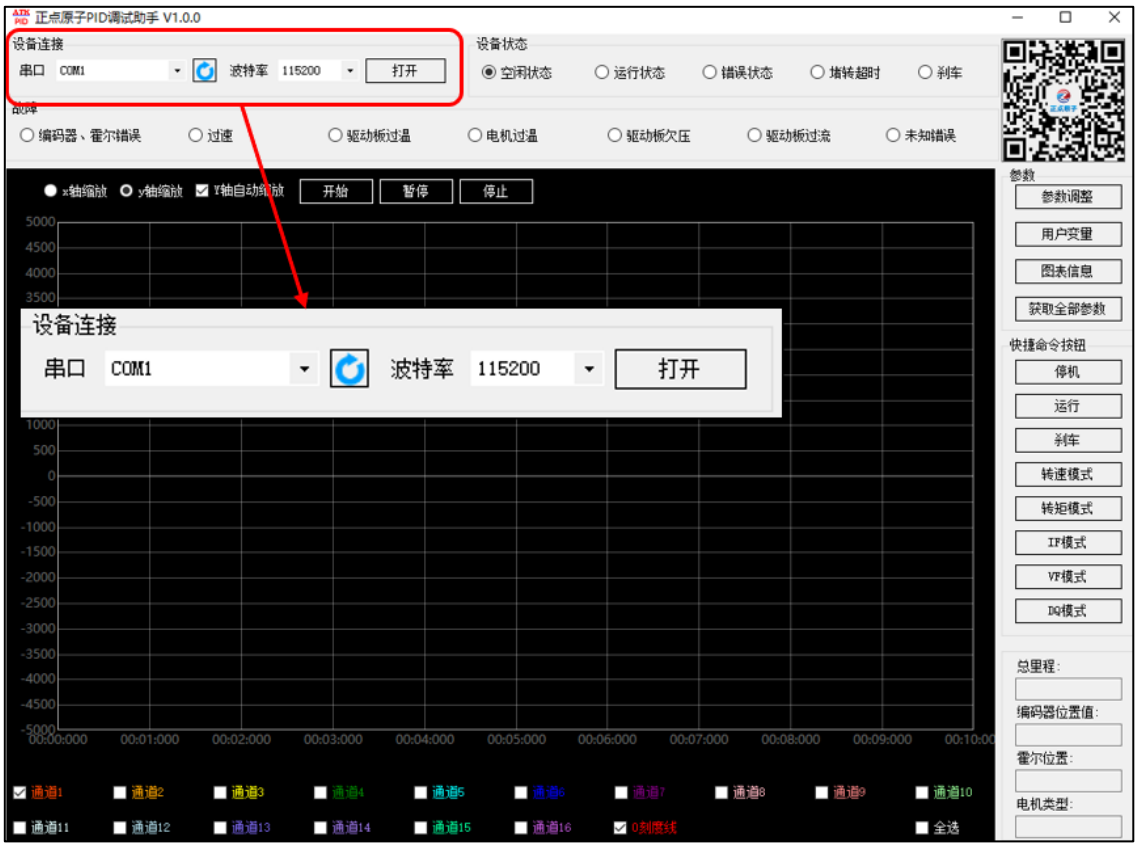


图 9.3.2.2 设备连接

③ 按需选择波形显示通道和缩放方式，点击“开始”即可显示波形，如图 9.3.2.3 所示：

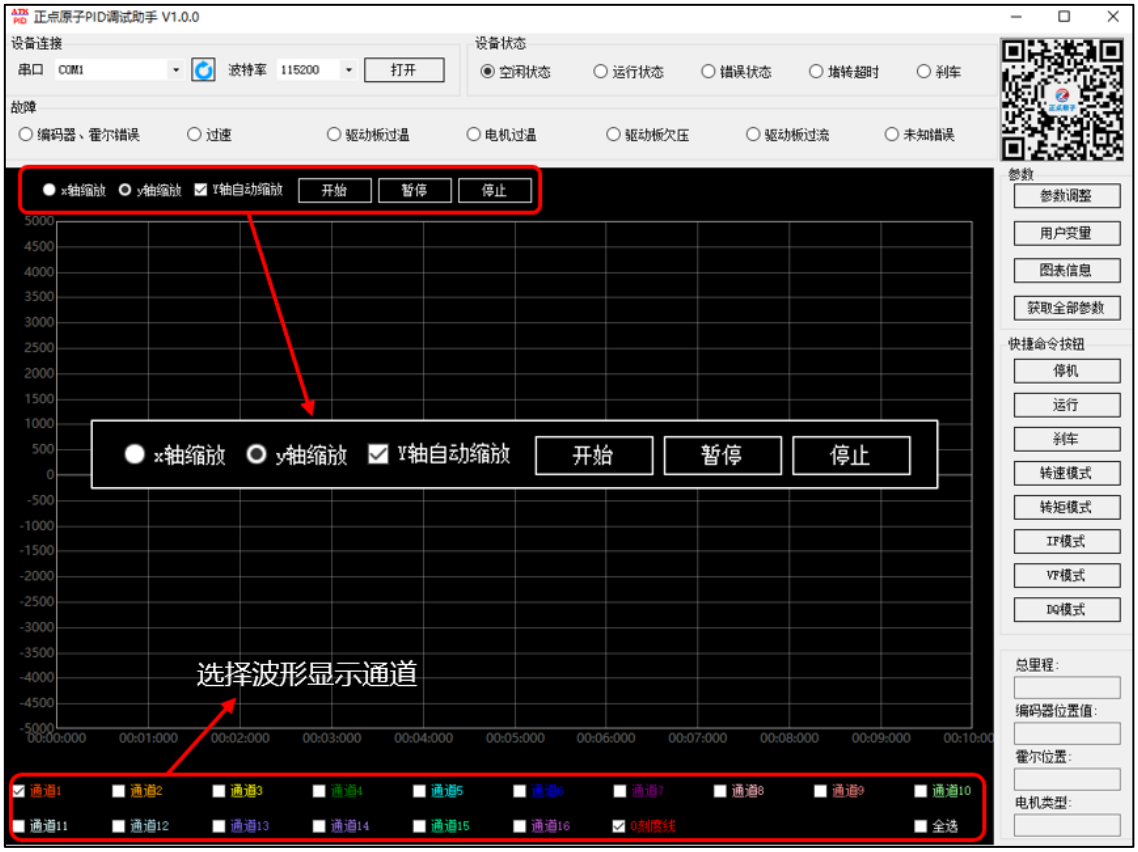


图 9.3.2.3 波形显示设置

④ 观察设备的状态及故障指示，如图 9.3.2.4 所示：

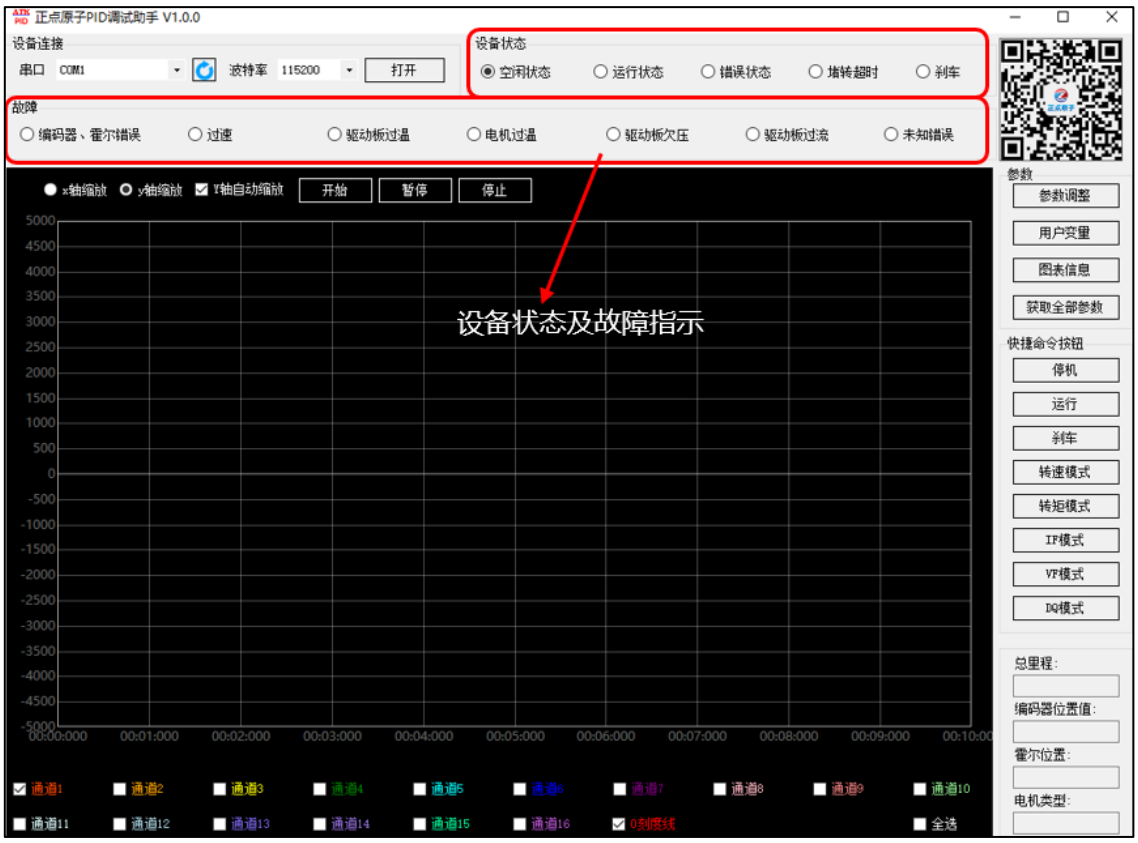


图 9.3.2.4 设备状态及故障指示

⑤ 右侧界面可设置、获取参数，下发控制指令及显示系统数据，如图 9.3.2.5 所示：

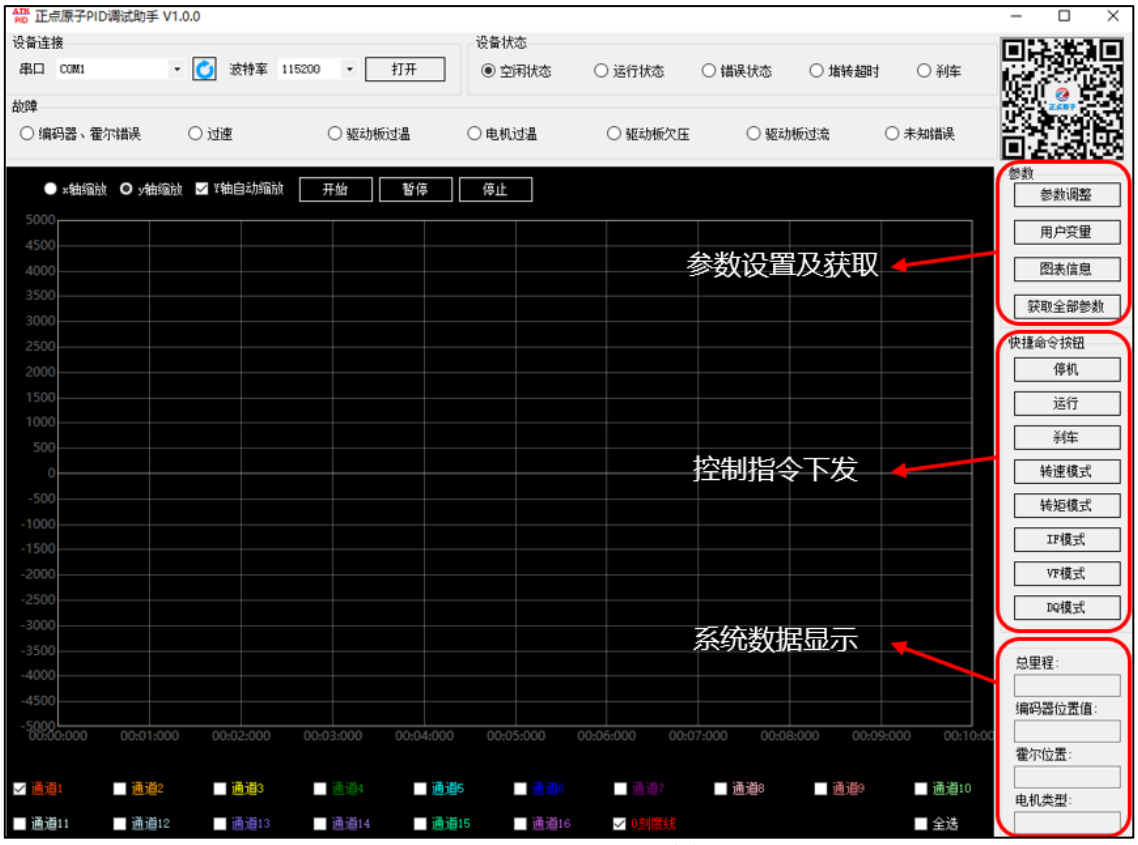


图 9.3.2.5 右侧界面功能

⑥ 上图 9.3.2.5 中，点击右侧的“参数调整”按钮，即可进入 PID 系数及多种参数设置的界面，如图 9.3.2.6 所示：

图 9.3.2.6 PID 系数及其他参数设置

进入到图 9.3.2.6 这个界面之后，我们可以在区域①选择所需的 PID 参数组别（1~10）以及设置相应的 PID 系数。PID 系数可以在输入框手动输入或者拉动横条来选择所需数值，数值确定完后点击“设置 PID”按钮即可设置 PID 系数，如果选择了自动发送指令，当我们拉动完横条之后，上位机就会自动设置对应的参数，其他的参数设置方法同理。

注意：参数的设置是有默认范围的，如果上位机默认的范围不适用，可以点击上图区域③的“参数范围设置”按钮，即可改变参数设置范围，如图 9.3.2.7 所示：

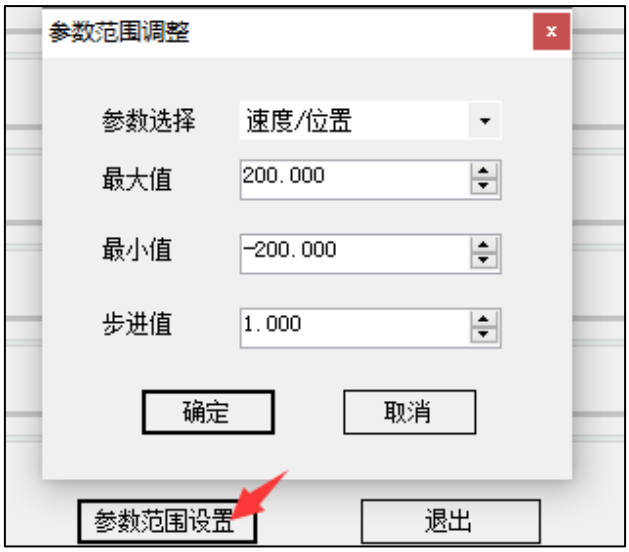


图 9.3.2.7 参数范围设置

关于 PID 上位机调试助手的简单使用就介绍到这里，其他的功能大家可以亲自去上手体验，这只是一个非常简单的工具。

9.3.3 下位机协议代码实现

下位机的协议代码包括三个部分：CRC16 校验函数、底层数据解析及上传函数、应用层数据接收及上传函数。这里我们只讲解**应用层**代码，详细的源码请大家参考《实验 6-5 直流有刷电机-速度环 PID 控制实验》里的 debug.c 和 debug.h 源码。

1）初始化调试函数

```
/**
 * @brief       初始化调试
 * @param       无
 * @retval      无
 */
void debug_init(void)
{
    debug_obj_init(&g_debug);           /* 初始化所需内存 */
}
```

该函数调用了 debug_obj_init 这个底层函数，从结构体 g_debug 的首地址开始，把该结构体所需要使用的空间清零，我们在执行其他的协议代码之前，要先调用该代码进行内存的初始化。

2）目标值范围设置函数

```
/**
 * @brief       设置目标值范围
 * @param       max_limit: 最大值
 * @param       min_limit: 最小值（反转时最大速度）
 * @param       step_max :  最大突变值
 * @retval      无
 */
void debug_set_point_range(float max_limit, float min_limit, float step_max)
{
    static float step_temp = 0.0;

    /* 判断目标值突变是否超过允许范围 */
    if (abs((int)(*debug_rev.speed - step_temp)) > step_max)
    {
        *debug_rev.speed = step_temp;           /* 超过最大突变值，保持原来数值 */
    }

    step_temp = *debug_rev.speed;               /* 保存本次数值 */

    if (*debug_rev.speed >= max_limit)          /* 超过限制 */
    {
        *debug_rev.speed = max_limit;           /* 配置为最大允许值 */
    }

    if (*debug_rev.speed <= min_limit)          /* 超过限制 */
    {
        *debug_rev.speed = min_limit;           /* 配置为最大允许值 */
    }
}
```

该函数可以限制 PID 系统的目标值范围，进而避免不合理目标值所带来的积分饱和问题，与此同时，它还可以限制目标值的突变范围，这对于具有惯性的系统尤为重要。

3）PID 数据上传函数

```
/**
 * @brief       PID 数据上传
 * @param       PIDx      : PID 组（1~10）
 * @param       *SetPoint : 目标速度地址
 * @param       P、I、D   : PID 参数
 * @retval      无
 */
```

```
void debug_send_initdata(upload_type PIDx, float *SetPoint, float P, float I,
float D)
{
    /* 开发板和上位机共用一个 PID 目标值的内存地址，数据同步更方便 */
    debug_rev.speed = (float *)(SetPoint);

    g_debug.pid[PIDx - TYPE_PID1].pid.pidf[0] = P;  /* 传入 P 值 */
    g_debug.pid[PIDx - TYPE_PID1].pid.pidf[1] = I;  /* 传入 I 值 */
    g_debug.pid[PIDx - TYPE_PID1].pid.pidf[2] = D;  /* 传入 D 值 */
    debug_upload_data(&g_debug, PIDx);              /* 发送 PID 参数 */
}
```

该函数用于上传 PID 系数到上位机，第一个入口参数 PIDx 传入的是 PID 参数的组别；第二个入口参数传入的是目标值的地址，这里我们将目标值的上传和下发地址设置为同一个，这样同步目标值数据会更加方便；最后三个入口参数传入的分别是 P、I、D 三个系数的值。

4）参数上传函数

```
/**
 * @brief       电流数据上传
 * @param       U_I、V_I、W_I : 三相电流数据
 * @note        如果只有单相，习惯用 U_I 上传
 * @retval      无
 */
void debug_send_current(float U_I, float V_I, float W_I)
{
    g_debug.amp[0] = U_I;                           /* 传入 U 相电流值 */
    g_debug.amp[1] = V_I;                           /* 传入 V 相电流值 */
    g_debug.amp[2] = W_I;                           /* 传入 W 相电流值 */
    debug_upload_data(&g_debug, TYPE_AMP);          /* 发送电流数据 */
}

/**
 * @brief       电压数据上传
 * @param       valtage : 电压数据
 * @retval      无
 */
void debug_send_valtage(float valtage)
{
    g_debug.bus_vol = valtage;                      /* 传入电压值 */
    debug_upload_data(&g_debug, TYPE_VBUS);         /* 发送电压数据 */
}

/**
 * @brief       功率数据上传
 * @param       power : 功率数据
 * @retval      无
 */
void debug_send_power(float power)
{
    g_debug.power = power;                          /* 传入功率值 */
    debug_upload_data(&g_debug, TYPE_POWER);        /* 发送功率数据 */
}

/**
 * @brief       速度数据上传
 * @param       speed : 速度数据
 * @retval      无
 */
void debug_send_speed(float speed)
{
    g_debug.speed = (int16_t)(speed);               /* 传入速度值 */
    debug_upload_data(&g_debug, TYPE_SPEED);        /* 发送速度数据 */
}
```

```
/**
 * @brief       总里程数据上传
 * @param       len : 总里程数据
 * @retval      无
 */
void debug_send_distance(uint64_t len)
{
    g_debug.sum_len = len;                              /* 传入总里程值 */
    debug_upload_data(&g_debug, TYPE_SUM_LEN);      /* 发送总里程数据 */
}

/**
 * @brief       温度数据上传
 * @param       motor_temp : 电机温度
 * @param       board_temp : 驱动板温度
 * @retval      无
 */
void debug_send_temp(float motor_temp, float board_temp)
{
    g_debug.temp[0] = board_temp;                   /* 传入驱动板温度值 */
    g_debug.temp[1] = motor_temp;                   /* 传入电机温度值 */
    debug_upload_data(&g_debug, TYPE_TEMP);         /* 发送温度数据 */
}

/**
 * @brief       电机状态上传
 * @param       motor_codestae : 电机状态
 * @retval      无
 */
void debug_send_motorstate(motor_state motor_codestae)
{
    g_debug.status = motor_codestae;                /* 传入电机状态 */
    debug_upload_data(&g_debug, TYPE_STATUS);       /* 发送电机状态 */
}

/**
 * @brief       电机类型上传
 * @param       motorcode : 电机类型
 * @retval      无
 */
void debug_send_motorcode(motor_code motorcode)
{
    g_debug.motor_code = motorcode;                 /* 传入电机类型 */
    debug_upload_data(&g_debug, TYPE_MOTOR_CODE);   /* 发送电机类型 */
}

/**
 * @brief       波形数据上传
 * @param       chx : 通道，取值 1~16
 * @param       wave: 数据
 * @retval      无
 */
void debug_send_wave_data(uint8_t chx, int16_t wave)
{
    g_debug.user_data[chx - 1] = wave;              /* 选择通道，传入数据 */
    debug_upload_data(&g_debug, TYPE_USER_DATA);    /* 发送波形数据 */
}
```

这一系列的函数都调用了 debug_upload_data 这个底层函数，它们的作用就是上传相应的参数到上位机，我们只要根据需求调用即可。

5）PID 参数接收函数

```
/**
 * @brief       上位机 PID 参数接收
 * @param       PIDx    : PID 组（1~10）
 * @param       P、I、D : PID 参数
 * @retval      无
 */
void debug_receive_pid(upload_type PIDx, float *P, float *I, float *D)
{
    *P = g_debug.pid[PIDx - TYPE_PID1].pid.pidf[0]; /* 接收 P 参数 */
    *I = g_debug.pid[PIDx - TYPE_PID1].pid.pidf[1]; /* 接收 I 参数 */
    *D = g_debug.pid[PIDx - TYPE_PID1].pid.pidf[2]; /* 接收 D 参数 */
}
```

该函数用于接收上位机下发的 PID 系数，第一个入口参数 PIDx 传入的是 PID 参数的组别；后面的三个入口参数传入的分别是 P、I、D 三个系数的接收地址。

6）上位机命令接收函数

```
/**
 * @brief       上位机命令接收
 * @param       无
 * @retval      0：无效，1：停机，2：运行，3：刹车
 */
uint8_t debug_receive_ctrl_code(void)
{
    static uint8_t rec_r = 0;

    /* 判断命令范围是否正确 */
    if (debug_rev.Ctrl_code >= 0x01 && debug_rev.Ctrl_code <= 0x03)
    {
        rec_r++;
        if (rec_r >= 2)
        {
            rec_r = 0;
            debug_rev.Ctrl_code = 0;
        }
        return debug_rev.Ctrl_code;                 /* 返回命令 */
    }
    return 0;
}
```

该函数用于解析上位机下发的命令，它的返回值如下：0（无效）；1（停机）；2（运行）；3（刹车）。

上述的就是下位机应用层代码，关于底层代码的实现方法，大家可以参考《实验 6-5 直流有刷电机-速度环 PID 控制实验》里的 debug.c 和 debug.h 源码。

第 10 章 直流有刷电机速度环控制实现

本章我们主要来学习直流有刷电机速度环 PID 控制的原理，并实现速度环 PID 控制的实验。

本章分为如下几个小节：

10.1 速度环 PID 控制原理
10.2 硬件设计
10.3 程序设计
10.4 下载验证

10.1 速度环 PID 控制原理

速度环 PID 控制的原理非常简单，我们只需要把 PID 控制流程中的控制对象换成电机速度即可，如图 10.1.1 所示：

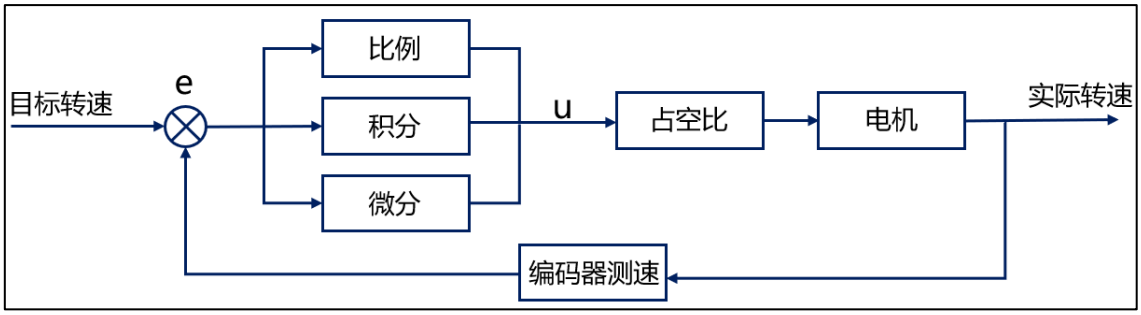


图 10.1.1 速度环 PID 控制流程

图 10.1.1 中，我们先设置目标转速，系统会计算出偏差 e，然后将偏差输入到 PID 控制的三个环节中，PID 计算后的输出值用于控制 PWM 的占空比，进而控制电机的速度。

10.2 硬件设计

1. 例程功能

1、本实验以电机开发板的直流有刷电机驱动接口 1 为例，基于编码器测速实验，加入速度环 PID 控制算法，对电机的速度进行闭环控制。

2、当按键 0 按下，就增大目标速度值；当按键 1 按下，就减小目标速度值。目标速度的绝对值大小决定电机的速度，它的正负决定电机的正反转方向。按下按键 2 则马上停止电机。

3、屏幕显示按键功能、占空比、目标转速以及实际转速。

4、串口 1 和上位机进行数据通信。

5、LED0 闪烁指示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）独立按键
KEY0 - PE2
KEY1 - PE3
KEY2 - PE4
3）定时器 1、3、6
TIM1 正常输出通道 PA8
TIM1 互补输出通道 PB13

TIM3 编码器 A 相输入通道 PC6
TIM3 编码器 B 相输入通道 PC7
4）SD（刹车）信号输出 PF10
5）串口 1
USART1_TX PB6（发送）
USART1_RX PB7（接收）

3. 原理图

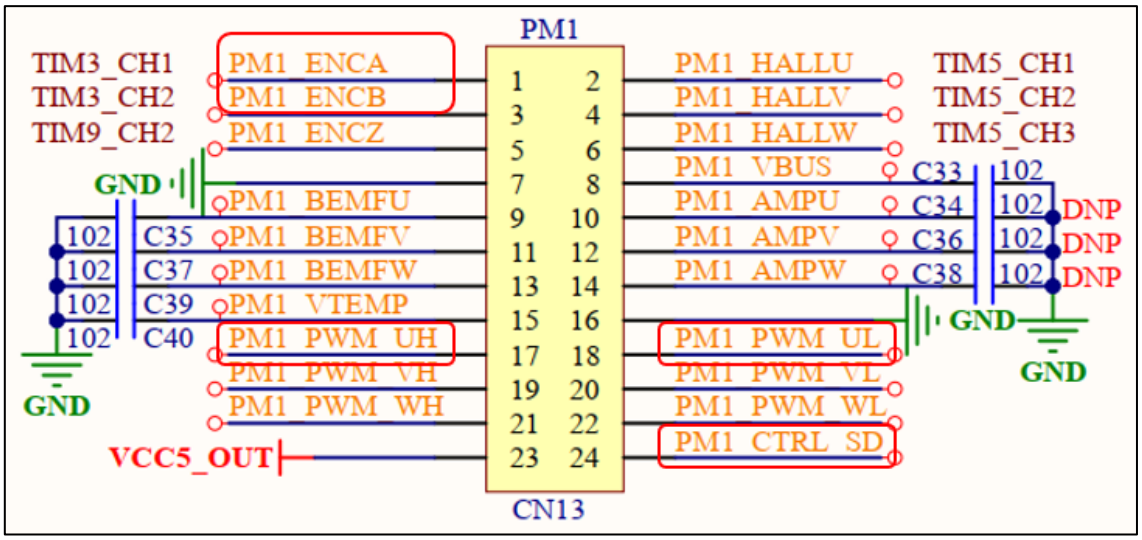


图 10.2.1 直流有刷电机接口原理图

图 10.2.1 就是我们 DMF407 电机开发板的直流有刷电机接口 1 原理图，本实验我们除了用到基础驱动所需的引脚，还需要用到 PM1_ENCA（PC6）、PM1_ENCB（PC7）这两个引脚，它们分别用于连接编码器的 A、B 两相。

本实验的硬件接线部分和编码器测速实验一模一样，这里就不再赘述，大家可以回顾编码器测速实验的内容。

10.3 程序设计

本实验所用到的基础驱动、编码器测速的代码在前面实验都有介绍过了。我们在程序解析中只讲解速度环 PID 控制相关的函数，下面介绍一下速度环 PID 控制的配置步骤。

速度环 PID 控制的配置步骤

1）配置相关定时器

配置基础驱动、编码器测速相关的定时器，此部分内容和编码器测速实验一致。

2）初始化串口 1

初始化串口 1，开启串口接收中断，串口 1 在 PID 控制代码中用于上位机通信。

注意：在 PID 控制的代码中，串口 1 仅用于 PID 数据上传，尽量不要输出其他信息，否则有可能影响 PID 数据。

3）定义 PID 参数结构体变量

为了方便管理 PID 相关的控制量，我们需要定义一个 PID 参数结构体变量，方法如下：

```
PID_TypeDef  g_speed_pid;          /* 速度环 PID 参数结构体 */
```

4）初始化 PID 参数

把 PID 控制系统的目标速度值、期望输出值、累计偏差等清零，然后配置 PID 系数。

5）初始化上位机调试

调用 debug_init 函数初始化所需内存，为上位机的调试做准备。

6）编写中断服务函数

在定时器 6 的更新中断回调函数里面进行速度环 PID 计算，计算后的结果用于控制 PWM 的占空比。

10.3.1 程序流程图

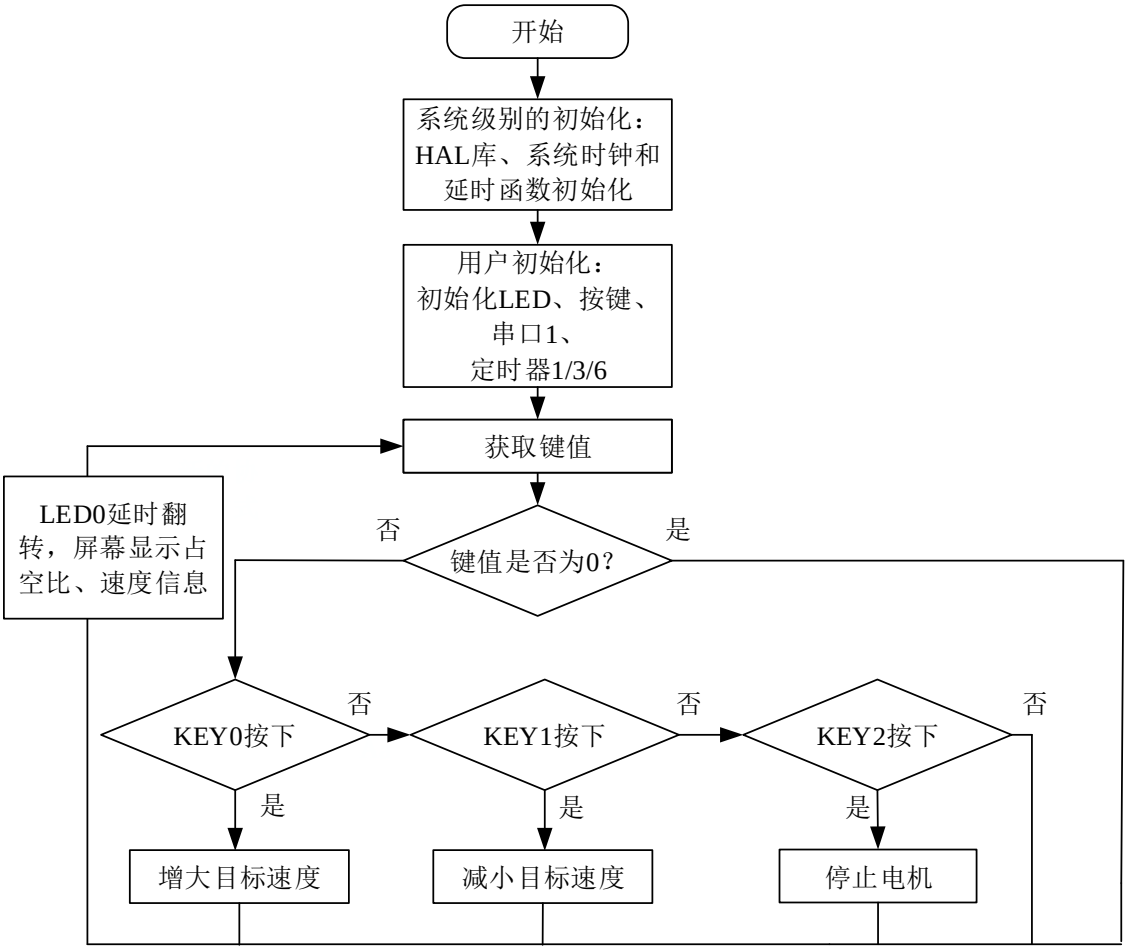


图 10.3.1.1 速度环 PID 控制程序流程图

10.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。关于基础驱动、编码器测速以及上位机协议的代码，这里不再赘述，大家可以回顾相应的章节。首先我们看串口 1 的代码，该串口在 PID 控制代码中用于上位机通信。我们只看串口 1 中断服务函数，其在 usart.c 文件中，具体定义如下：

```
/**
 * @brief       串口 1 中断服务函数
 * @param       无
 * @retval      无
 */
void USART_UX_IRQHandler(void)
{
    uint8_t res;

    /* 接收到数据 */
    if ((__HAL_UART_GET_FLAG(&g_uart1_handle, UART_FLAG_RXNE) != RESET))
    {
        HAL_UART_Receive(&g_uart1_handle, &res, 1, 1000);
        debug_handle(&res);
    }
}
```

该函数非常简单，当上位机有数据下发时，我们直接调用 debug_handle 函数来解析上位机下发的数据。

下面我们看速度环 PID 控制的相关源码，其包括两个文件：pid.c 和 pid.h。

首先看 pid.h 头文件的几个宏定义：

```
/* PID 相关参数 */

#define  INCR_LOCT_SELECT  0        /* 0：位置式 ，1：增量式 */

#if INCR_LOCT_SELECT

/* 增量式 PID 参数相关宏 */
#define  KP      8.50f              /* P 参数*/
#define  KI      5.00f              /* I 参数*/
#define  KD      0.10f              /* D 参数*/
#define  SMAPLSE_PID_SPEED  50      /* 采样周期 单位 ms*/

#else

/* 位置式 PID 参数相关宏 */
#define  KP      10.0f              /* P 参数*/
#define  KI      6.00f              /* I 参数*/
#define  KD      0.5f               /* D 参数*/
#define  SMAPLSE_PID_SPEED  50      /* 采样周期 单位 ms*/

#endif

/* PID 参数结构体 */
typedef struct
{
    __IO float  SetPoint;           /* 设定目标 */
    __IO float  ActualValue;        /* 期望输出值 */
    __IO float  SumError;           /* 误差累计 */
    __IO float  Proportion;         /* 比例常数 P */
    __IO float  Integral;           /* 积分常数 I */
    __IO float  Derivative;         /* 微分常数 D */
    __IO float  Error;              /* Error[1] */
    __IO float  LastError;          /* Error[-1] */
    __IO float  PrevError;          /* Error[-2] */
} PID_TypeDef;

extern PID_TypeDef  g_speed_pid;    /* 速度环 PID 参数结构体 */
```

可以把上面的宏定义分成两部分，第一部分是 PID 计算方式以及 PID 系数的宏定义，我们可以通过改变 INCR_LOCT_SELECT 这个宏的值来选择相应的 PID 计算方式，第二部分则是 PID 参数相关的结构体，这个结构体用于管理 PID 控制所需要的控制量，本实验中我们定义了速度环 PID 参数的结构体变量 g_speed_pid。

下面看 pid.c 的程序，这里我们只介绍 PID 初始化函数，关于 PID 闭环控制的函数介绍请回顾 8.3 章节，PID 初始化函数定义如下：

```
/**
 * @brief       pid 初始化
 * @param       无
 * @retval      无
 */
void pid_init(void)
{
    g_speed_pid.SetPoint = 0;       /* 设定目标值 */
    g_speed_pid.ActualValue = 0.0;  /* 期望输出值 */
    g_speed_pid.SumError = 0.0;     /* 积分值 */
    g_speed_pid.Error = 0.0;        /* Error[1] */
    g_speed_pid.LastError = 0.0;    /* Error[-1] */
```

```
    g_speed_pid.PrevError = 0.0;     /* Error[-2] */
    g_speed_pid.Proportion = KP;     /* 比例常数 Proportional Const */
    g_speed_pid.Integral = KI;       /* 积分常数 Integral Const */
    g_speed_pid.Derivative = KD;     /* 微分常数 Derivative Const */
}
```

该函数主要是将 PID 控制系统的目标速度值、期望输出值、累计偏差等清零，然后配置 PID 系数。

最后要介绍的是定时器的更新中断回调函数，它在 dcmotor_time.c 中实现，具体的定义如下：

```
/**
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @note        此函数会被定时器中断函数共同调用的
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    static uint8_t val = 0;

    /* 定时器 3 相关程序 */
    if (htim->Instance == TIM3)
    {
       /* 判断 CR1 的 DIR 位 */
       if(__HAL_TIM_IS_TIM_COUNTING_DOWN(&g_timx_encode_chy_handle))
       {
           g_timx_encode_count--;            /* DIR 位为 1，也就是递减计数 */
       }
       else
       {
           g_timx_encode_count++;            /* DIR 位为 0，也就是递增计数 */
       }
    }

    /* 定时器 6 相关程序 */
    else if (htim->Instance == TIM6)
    {
       int Encode_now = gtim_get_encode();     /* 获取编码器值，用于计算速度 */

       speed_computer(Encode_now, 5);          /* 5ms 计算一次速度 */

       if (val % SMAPLSE_PID_SPEED == 0)        /* 50ms 进行一次 pid 计算 */
       {
           if (g_run_flag)                      /* 判断电机是否启动了*/
           {
               /* PID 计算，输出比较值（占空比） */
               g_motor_data.motor_pwm = increment_pid_ctrl(&g_speed_pid,
                                                g_motor_data.speed);
               /* 限制输出 */
               if (g_motor_data.motor_pwm >= 8200)
               {
                   g_motor_data.motor_pwm = 8200;
               }
               else if (g_motor_data.motor_pwm <= -8200)
               {
                   g_motor_data.motor_pwm = -8200;
               }

#if DEBUG_ENABLE  /* 发送基本参数*/

               /* 选择通道 1，发送实际速度（波形显示）*/
               debug_send_wave_data( 1 ,g_motor_data.speed);
               /* 选择通道 2，发送目标速度（波形显示）*/
```

```
                debug_send_wave_data( 2 ,g_speed_pid.SetPoint);
                /* 选择通道 3，发送占空比（波形显示）*/
                debug_send_wave_data( 3 ,g_motor_data.motor_pwm * 100 / 8400);
#endif
                motor_pwm_set(g_motor_data.motor_pwm);    /* 设置电机转速 */
            }
            val = 0;
        }
        val ++;
    }
}
```

该函数我们只介绍定时器 6 相关的程序，进入更新中断回调函数后，所执行的代码逻辑如下：

第一步，判断是不是定时器 6 的寄存器基地址，如果是则获取编码器的计数总值，然后每隔 5ms 计算一次电机速度。

第二步，每隔 50ms 进行一次 PID 计算，在计算 PID 之前，需要判断 g_run_flag 是否为 1，如果是则说明电机已经启动，可以开始 PID 计算。

第三步，进行速度环 PID 计算，返回的输出存放在 g_motor_data.motor_pwm 这个成员中，该成员用于设置 PWM 的占空比，除此之外，我们还需要对返回的输出进行限制，这一点非常重要。

第四步，发送实际速度、目标速度以及占空比的波形数据到上位机，最后设置 PWM 的占空比，进而控制电机的转速。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key;
    uint16_t t;
    uint8_t debug_cmd = 0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口 1 初始化，用于上位机调试 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    pid_init();                                 /* 初始化 PID 参数 */
    atim_timx_cplm_pwm_init(8400 - 1 , 0);      /*  168Mhz 的计数频率 */
    dcmotor_init();                             /* 初始化电机 */
    gtim_timx_encoder_chy_init(0XFFFF, 0);      /* 编码器定时器初始化 */
    btim_timx_int_init(1000 - 1 , 84 - 1);      /* 基本定时器初始化，1ms 计数周期 */

#if DEBUG_ENABLE                                /* 开启调试 */

    debug_init();                               /* 初始化调试 */
    debug_send_motorcode(DC_MOTOR);             /* 上传电机类型（直流有刷电机） */
    debug_send_motorstate(IDLE_STATE);          /* 上传电机状态（空闲） */

    /* 同步数据（选择第 1 组 PID,目标速度地址,P,I,D 参数）到上位机 */
    debug_send_initdata(TYPE_PID1, (float *)(&g_speed_pid.SetPoint),
                        KP, KI, KD);

#endif

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10, 10, 200, 16, 16, "DcMotor Test", g_point_color);
    lcd_show_string(10, 30, 200, 16, 16, "KEY0:Start forward", g_point_color);
    lcd_show_string(10, 50, 200, 16, 16, "KEY1:Start backward", g_point_color);
    lcd_show_string(10, 70, 200, 16, 16, "KEY2:Stop", g_point_color);
```

```
    while (1)
    {
        key = key_scan(0);                                  /* 按键扫描 */

        if(key == KEY0_PRES)                                /* 当 key0 按下 */
        {
            g_run_flag = 1;                                 /* 标记电机启动 */
            g_speed_pid.SetPoint += 30;

            if (g_speed_pid.SetPoint == 0)
            {
                dcmotor_stop();                             /* 停止则立刻响应 */
                g_motor_data.motor_pwm = 0;
                motor_pwm_set(g_motor_data.motor_pwm);      /* 设置电机方向、转速 */
            }
            else
            {
                dcmotor_start();                            /* 开启电机 */
                if (g_speed_pid.SetPoint >= 300)            /* 限速 */
                {
                    g_speed_pid.SetPoint = 300;
                }
#if DEBUG_ENABLE
                debug_send_motorstate(RUN_STATE);           /* 上传电机状态（运行） */
#endif
            }
        }

        else if(key == KEY1_PRES)                           /* 当 key1 按下 */
        {
            g_run_flag = 1;                                 /* 标记电机启动 */
            g_speed_pid.SetPoint -= 30;

            if (g_speed_pid.SetPoint == 0)
            {
                dcmotor_stop();                             /* 停止则立刻响应 */
                g_motor_data.motor_pwm = 0;
                motor_pwm_set(g_motor_data.motor_pwm);      /* 设置电机方向、转速 */
            }
            else
            {
                dcmotor_start();                            /* 开启电机 */
                if (g_speed_pid.SetPoint <= -300)           /* 限速 */
                {
                    g_speed_pid.SetPoint = -300;
                }
#if DEBUG_ENABLE
                debug_send_motorstate(RUN_STATE);           /* 上传电机状态（运行） */
#endif
            }
        }

        else if(key == KEY2_PRES)                           /* 当 key2 按下 */
        {
            dcmotor_stop();                                 /* 停止电机 */
            pid_init();                                     /* 重置 pid 参数 */
            g_run_flag = 0;                                 /* 标记电机停止 */
            g_motor_data.motor_pwm = 0;
            motor_pwm_set(g_motor_data.motor_pwm);          /* 设置电机转向、速度 */

#if DEBUG_ENABLE
            debug_send_motorstate(BREAKED_STATE);           /* 上传电机状态（刹车） */
```

```
            debug_send_initdata(TYPE_PID1, (float *)(&g_speed_pid.SetPoint),
                                KP, KI, KD);            /* 同步数据到上位机 */
#endif
        }

#if DEBUG_ENABLE

        /* 查询接收 PID 助手的 PID 参数 */
        debug_receive_pid(TYPE_PID1, (float *)&g_speed_pid.Proportion,
        (float *)&g_speed_pid.Integral, (float *)&g_speed_pid.Derivative);

        debug_set_point_range(300, -300, 300);          /* 设置目标速度范围 */
        debug_cmd = debug_receive_ctrl_code();          /* 读取上位机指令 */

        if (debug_cmd == BREAKED)                       /* 电机刹车 */
        {
            dcmotor_stop();                             /* 停止电机 */
            pid_init();                                 /* 重置 pid 参数 */
            g_run_flag = 0;                             /* 标记电机停止 */
            g_motor_data.motor_pwm = 0;
            motor_pwm_set(g_motor_data.motor_pwm);      /* 设置电机转向、速度 */
            debug_send_motorstate(BREAKED_STATE);       /* 上传电机状态（刹车） */
            debug_send_initdata(TYPE_PID1, (float *)(&g_speed_pid.SetPoint),
                                KP, KI, KD);
        }
        else if (debug_cmd == RUN_CODE)                  /* 电机运行 */
        {
            dcmotor_start();                            /* 开启电机 */
            g_speed_pid.SetPoint = 30;                  /* 设置目标速度：30RPM */
            g_run_flag = 1;                             /* 标记电机启动 */
            debug_send_motorstate(RUN_STATE);           /* 上传电机状态（运行） */
        }
#endif
        t++;
        if(t % 20 == 0)
        {
            lcd_dis();                                  /* 显示数据 */
            LED0_TOGGLE();                              /* LED0(红灯) 翻转 */
#if DEBUG_ENABLE
            debug_send_speed(g_motor_data.speed);       /* 发送速度 */
#endif
        }
        delay_ms(10);
    }
}
```

main.c 的代码逻辑如下：

第一步，初始化相关的定时器、串口，本实验需要利用串口 1 和上位机进行通讯。

第二步，初始化 PID 参数、上位机调试，它们分别调用的是 pid_init 和 debug_init 函数。

第三步，上传电机的状态（空闲）、类型（直流有刷电机）、PID 参数到上位机，它们分别调用的是 debug_send_motorcode、debug_send_motorstate 以及 debug_send_initdata 函数。

第四步，在 while 循环里面检测按键是否按下，如果 key0 按下，则目标速度增加 30，上传电机状态（运行）；如果 key1 按下，则目标速度减小 30，上传电机状态（运行）；如果 key2 按下，则停止电机，重置 PID 参数，并重新上传电机状态（刹车）以及 PID 参数到上位机。

第五步，接收上位机下发的 PID 参数、设置目标速度的调节范围，它们分别调用的是 debug_receive_pid 以及 debug_set_point_range 函数。我们这里重点介绍一下后者，debug_set_point_range 函数的第一、二个入口参数分别传入目标速度的最大值、最小值，它们的正负号代表电机的正反转方向，第三个入口参数传入的是电机速度的最大突变值。

第六步，调用 debug_receive_ctrl_code 函数，接收上位机下发的命令。如果上位机的命令为 BREAKED（刹车），则停止电机，重置 PID 参数，并重新上传电机状态（刹车）以及 PID

参数到上位机：如果上位机的命令是 RUN_CODE，则开启电机，设置目标速度为 30RPM，上传电机状态（运行）到上位机。

第七步，每隔 200ms 更新一次数据到屏幕，调用 debug_send_speed 函数发送实际速度值到上位机。

10.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 上显示按键功能、占空比以及电机速度信息，当我们按下 KEY0，目标速度将增大；按下 KEY1，目标速度将减小；按下 KEY2，电机将停止。目标速度为正数时，电机正转，反之电机反转。我们再打开 PID 调试助手，选择对应的串口端口，我这边是 COM7，接着选择通道 1 和 2，点击“开始”按钮，即可开始显示波形，如下图 10.4.1 所示：

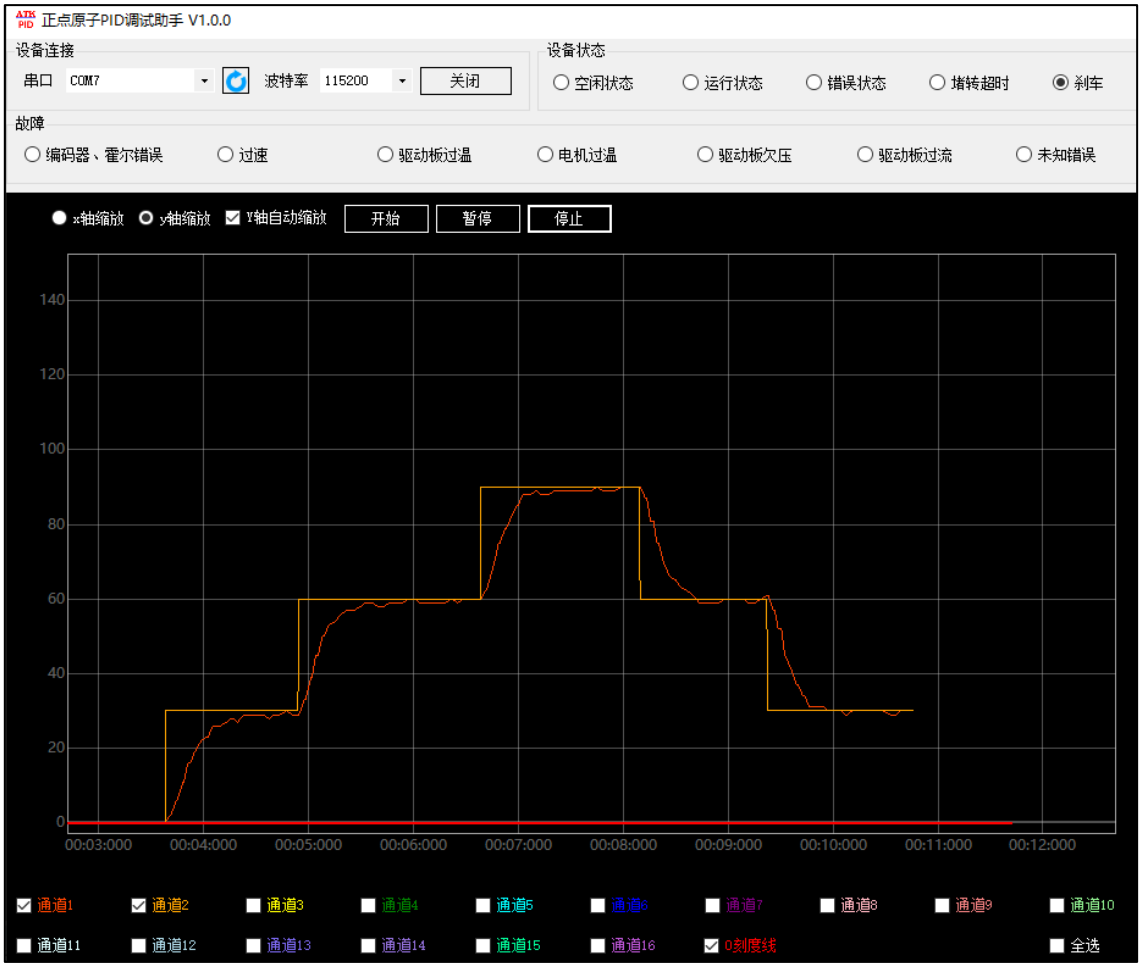


图 10.4.1 速度环 PID 控制波形

图 10.4.1 中，橙线代表目标速度，红线代表实际速度，当我们按下 KEY0，目标速度增大，橙线先发生变化，而红线（实际速度）会逐渐靠近橙线（目标速度）；按下 KEY1，目标速度将减小，曲线的变化同理；按下 KEY2，电机将停止，目标速度将为 0。

注意：1、本实验需要使用 USB 数据线连接开发板的串口 1 到电脑，并启动电机之后才会有波形变化；2、如果发现波形不对，请检查电机的 M+和 M-接线；3、PID 系数并不是通用的，如果 PID 曲线不理想，大家需要根据自己的实际系统去调节。

第 11 章 直流有刷电机电流环控制实现

本章我们主要来学习直流有刷电机电流环 PID 控制的原理，并实现电流环 PID 控制的实验。

本章分为如下几个小节：

11.1 电流环 PID 控制原理
11.2 硬件设计
11.3 程序设计
11.4 下载验证

11.1 电流环 PID 控制原理

电流环 PID 控制的原理非常简单，我们只需要把 PID 控制流程中的控制对象换成电机电流即可，如图 11.1.1 所示：

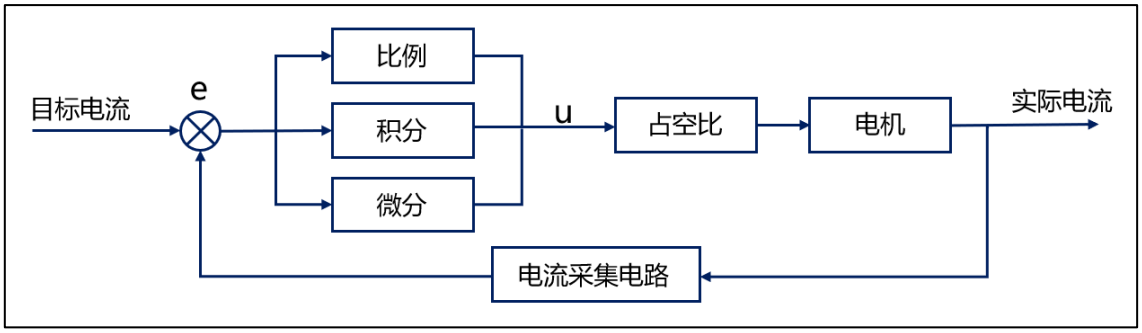


图 11.1.1 电流环 PID 控制流程

图 11.1.1 中，我们先设置目标电流，系统会计算出偏差 e，然后将偏差输入到 PID 控制的三个环节中，PID 计算后的输出值用于控制 PWM 的占空比，进而控制电机的电流。

11.2 硬件设计

1. 例程功能

1、本实验以电机开发板的直流有刷电机驱动接口 1 为例，基于电压温度电流采集实验，加入电流环 PID 控制算法，对电机的电流进行闭环控制。

2、当按键 0 按下，就增大目标电流值；当按键 1 按下，就减小目标电流值，目标电流的大小决定电机的速度。按下按键 2 则马上停止电机。

3、屏幕显示按键功能、占空比、目标电流以及实际电流。

4、串口 1 和上位机进行数据通信。

5、LED0 闪烁指示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）独立按键
KEY0 - PE2
KEY1 - PE3
KEY2 - PE4
3）定时器 1
TIM1 正常输出通道 PA8
TIM1 互补输出通道 PB13

4）SD（刹车）信号输出 PF10
5）ADC
ADC1 通道 8 PB0（电流）
6）串口 1
USART1_TX PB6（发送）
USART1_RX PB7（接收）

3. 原理图

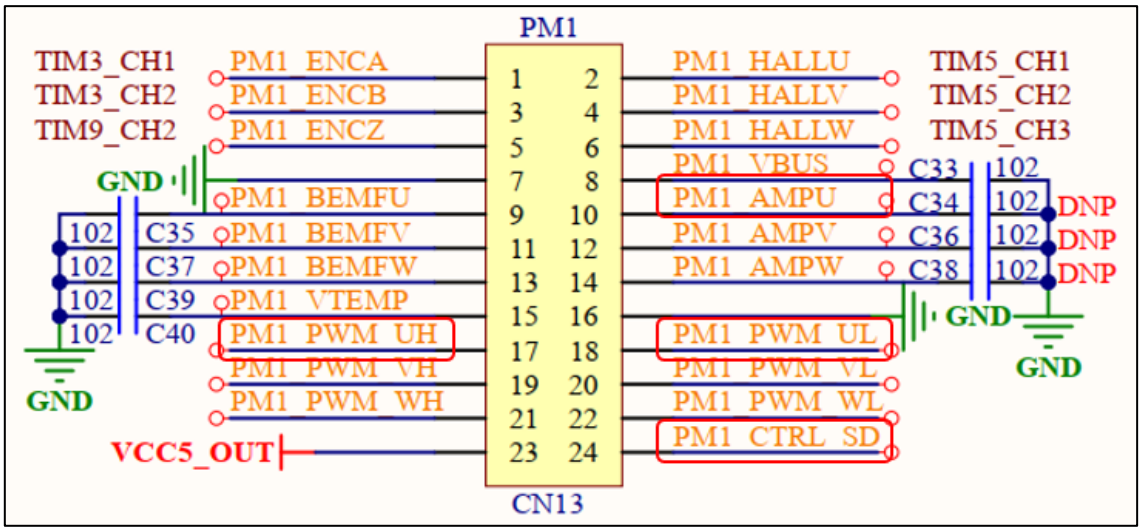


图 11.2.1 直流有刷电机接口原理图

图 11.2.1 就是我们 DMF407 电机开发板的直流有刷电机接口 1 原理图，本实验我们只需要用到了 PM1_PWM_UH（PA8）、PM1_PWM_UL（PB13）、PM1_CTRL_SD（PF10）和 PM1_AMPU（PB0）这 4 个引脚，其中 PA8 和 PB13 用于输出所需的 PWM 控制信号，PF10 用于输出刹车信号，PB0 用于检测电流采集电路对应的输出电压。

本实验的硬件接线部分和编码器测速实验一模一样，这里就不再赘述，大家可以回顾编码器测速实验的内容。

11.3 程序设计

本实验所用到的基础驱动、电流采集的代码在前面实验都有介绍过了。我们在程序解析中只讲解电流环 PID 控制相关的函数，下面介绍一下电流环 PID 控制的配置步骤。

电流环 PID 控制的配置步骤

1）配置相关定时器、ADC

配置基础驱动相关的定时器以及电流采集相关的 ADC。

2）初始化串口 1

初始化串口 1，开启串口接收中断，串口 1 在 PID 控制代码中用于上位机通信。

注意：在 PID 控制的代码中，串口 1 仅用于 PID 数据上传，尽量不要输出其他信息，否则有可能影响 PID 数据。

3）定义 PID 参数结构体变量

为了方便管理 PID 相关的控制量，我们需要定义一个 PID 参数结构体变量，方法如下：

```
PID_TypeDef  g_current_pid;        /* 电流环 PID 参数结构体 */
```

4）初始化 PID 参数

把 PID 控制系统的期望输出值、累计偏差等清零，然后配置目标电流初始值和 PID 系数。

5）初始化上位机调试

调用 debug_init 函数初始化所需内存，为上位机的调试做准备。

6）编写中断服务函数

在定时器 6 的更新中断回调函数里面进行电流环 PID 计算，计算后的结果用于控制 PWM 的占空比。

11.3.1 程序流程图

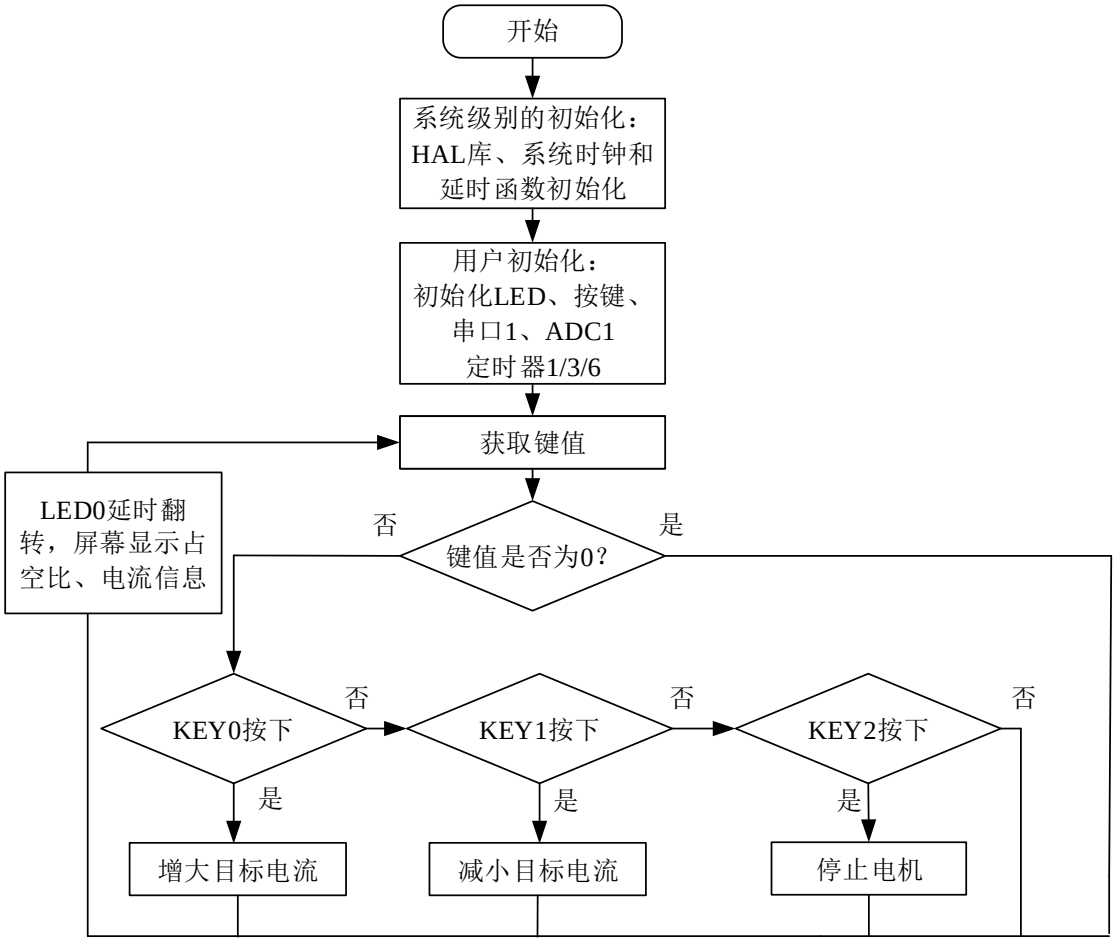


图 11.3.1.1 电流环 PID 控制程序流程图

11.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。关于基础驱动、电流采集以及上位机协议的代码，这里不再赘述，大家可以回顾相应的章节。首先我们看电流环 PID 控制的相关源码，其包括两个文件：pid.c 和 pid.h。

首先看 pid.h 头文件的几个宏定义：

```
/* PID 相关参数 */

#define  INCR_LOCT_SELECT  0        /* 0：位置式，1：增量式 */

#if INCR_LOCT_SELECT

/* 增量式 PID 参数相关宏 */
#define  KP      0.0f               /* P 参数*/
#define  KI      6.0f               /* I 参数*/
#define  KD      4.0f               /* D 参数*/
#define  SMAPLSE_PID_SPEED  50      /* 采样周期 单位 ms*/

#else
```

```
/* 位置式 PID 参数相关宏 */
#define  KP      10.0f                /* P 参数*/
#define  KI      7.0f                 /* I 参数*/
#define  KD      2.0f                 /* D 参数*/
#define  SMAPLSE_PID_SPEED  50        /* 采样周期 单位 ms*/

#endif

/*PID 结构体*/
typedef struct
{
    __IO float  SetPoint;             /* 设定目标 */
    __IO float  ActualValue;          /* 期望输出值 */
    __IO float  SumError;             /* 误差累计 */
    __IO float  Proportion;           /* 比例常数 P */
    __IO float  Integral;             /* 积分常数 I */
    __IO float  Derivative;           /* 微分常数 D */
    __IO float  Error;                /* Error[1] */
    __IO float  LastError;            /* Error[-1] */
    __IO float  PrevError;            /* Error[-2] */
} PID_TypeDef;

extern PID_TypeDef  g_current_pid;   /* 电流环 PID 参数结构体 */
```

可以把上面的宏定义分成两部分，第一部分是 PID 计算方式以及 PID 系数的宏定义，我们可以通过改变 INCR_LOCT_SELECT 这个宏的值来选择相应的 PID 计算方式，第二部分则是 PID 参数相关的结构体，这个结构体用于管理 PID 控制所需要的控制量，本实验中我们定义了电流环 PID 参数的结构体变量 g_ current _pid。

下面看 pid.c 的程序，这里我们只介绍 PID 初始化函数，关于 PID 闭环控制的函数介绍请回顾 8.3 章节，PID 初始化函数定义如下：

```
/**
 * @brief       pid 初始化
 * @param       无
 * @retval      无
 */
void pid_init(void)
{
    g_current _pid.SetPoint = 40;       /* 设定目标值 */
    g_current _pid.ActualValue = 0.0;   /* 期望输出值 */
    g_current _pid.SumError = 0.0;      /* 积分值 */
    g_current _pid.Error = 0.0;         /* Error[1] */
    g_current _pid.LastError = 0.0;     /* Error[-1] */
    g_current _pid.PrevError = 0.0;     /* Error[-2] */
    g_current _pid.Proportion = KP;     /* 比例常数 Proportional Const */
    g_current _pid.Integral = KI;       /* 积分常数 Integral Const */
    g_current _pid.Derivative = KD;     /* 微分常数 Derivative Const */
}
```

该函数主要是将 PID 控制系统的期望输出值、累计偏差等清零，然后配置初始目标电流以及 PID 系数。这里配置初始目标电流为 40mA，是因为电机启动时所需的电流较大，太小的电流无法正常启动电机。

接下来看电流采集相关的源码，我们只介绍滤波算法的内容，该算法在 adc.c 的 ADC 采集中断回调函数中实现，具体定义如下：

```
/**
 * @brief       ADC 采集中断回调函数
 * @param       无
 * @retval      无
 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
```

```
    float temp_c = 0.0;
    static float add_adc = 0;
    static float init_adc_value = 0;
    static uint8_t adc_count1 = 0, adc_count2 = 0;

    if ( hadc->Instance == ADC_ADCX )                    /* 判断是不是 ADC1 */
    {
        adc_count1++;
        HAL_ADC_Stop_DMA(&g_adc_nch_dma_handle);         /* 关闭 DMA 转换 */

        calc_adc_val(g_adc_val);                         /* 计算 ADC 的平均值 */
        add_adc += g_adc_val[2];      /* 取出电流通道对应的 ADC 值进行累计 */

        if (adc_count1 >= 15)                            /* 累计 15 次 */
        {
            add_adc = (float)(add_adc / adc_count1);     /* 取平均值 */

            if (adc_count2 <= 16)     /* 采集 16 次 ADC 平均值计算参考电压的 ADC 值 */
            {
                adc_count2++;
                init_adc_value += add_adc;               /* 对平均值累计求和 */

                if (adc_count2 == 16)                    /* 平均值累计 16 次 */
                {
                    adc_count2 = 17;                     /* 不再进入 */
                    init_adc_value = (init_adc_value / 16.0f); /* 存储初始 ADC 值 */
                }
            }
            if (adc_count2 >= 17)     /* 采集完参考 ADC 值后，采集电流通道当前 ADC 值 */
            {
                temp_c = (add_adc - init_adc_value) * ADC2CURT; /* 计算电流 */

                g_motor_data.current = (float)((g_motor_data.current *
                (float)0.60) + ((float)0.40 * temp_c));  /* 一阶低通滤波 */

                if (g_motor_data.current <= 20)          /* 过滤掉微弱浮动电流 */
                {
                    g_motor_data.current = 0.0;
                }
            }
            add_adc = 0;
            adc_count1 = 0;
        }
        HAL_ADC_Start_DMA(&g_adc_nch_dma_handle, (uint32_t *)&g_adc_value,
                          (uint32_t)(ADC_SUM));          /* 启动 DMA 转换 */
    }
}
```

进入 ADC 采集中断回调函数后，所执行的代码逻辑如下：

第一步，判断是不是 ADC1 的寄存器基地址，如果是则让变量 adc_count1 自增 1，然后关闭 DMA 传输，计算电流采集通道对应的 ADC 平均值，最后把平均值累计到变量 add_adc 中（第一次滤波）。

第二步，当变量 adc_count1 自增到 15 时，变量 add_adc 已经累计 15 次 ADC 平均值，此时再次计算 ADC 平均值并返回到变量 add_adc 中（第二次滤波）。

第三步，当变量 adc_count2 小于等于 16 时，不断地累计变量 add_adc 的值，存入到变量 init_adc_value 中，然后再次计算 ADC 平均值并返回变量 init_adc_value 中（第三次滤波），该变量存储的是参考电压对应的 ADC 值，后续将用于电流的计算。

第四步，当变量 adc_count2 大于等于 17 时，根据公式计算电流值并存入变量 temp_c 中，接着进行一阶低通滤波，过滤掉微弱的浮动电流，最后再启动 DMA 转换。

注意：电流的滤波对于 PID 系统的控制至关重要，如果电流环的 PID 曲线出现了严重的振荡，通过调整 PID 系数已经无法优化，此时可以适当地调整滤波的次数，在系统的响应速

度和稳定性之间寻求平衡点。

最后要介绍的是定时器的更新中断回调函数，它在 dcmotor_time.c 中实现，具体的定义如下：

```
/**
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @note        此函数会被定时器中断函数共同调用的
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    int32_t motor_pwm_temp = 0;
    static uint8_t val = 0;

    /* 定时器 3 相关程序 */
    if (htim->Instance == TIM3)
    {
       /* 判断 CR1 的 DIR 位 */
       if(__HAL_TIM_IS_TIM_COUNTING_DOWN(&g_timx_encode_chy_handle))
       {
          g_timx_encode_count--;                /* DIR 位为 1，也就是递减计数 */
       }
       else
       {
          g_timx_encode_count++;                /* DIR 位为 0，也就是递增计数 */
       }
    }

    /* 定时器 6 相关程序 */
    else if (htim->Instance == TIM6)
    {
       int Encode_now = gtim_get_encode();      /* 获取编码器值，用于计算速度 */

       speed_computer(Encode_now, 5);           /* 5ms 计算一次速度 */

       if (val % SMAPLSE_PID_SPEED == 0)        /* 进行一次 pid 计算 */
       {
          if (g_run_flag)                       /* 判断电机是否启动了*/
          {
             /* 电流环 PID 计算，输出比较值（占空比）*/
             motor_pwm_temp = increment_pid_ctrl(&g_current_pid,
                                                 g_motor_data.current);
             /* 进行一阶低通滤波 */
             g_motor_data.motor_pwm = (int32_t)((g_motor_data.motor_pwm * 0.5)
                                                + (motor_pwm_temp * 0.5));

             if (g_motor_data.motor_pwm >= 8200)       /* 限制占空比 */
             {
                g_motor_data.motor_pwm = 8200;
             }
             else if (g_motor_data.motor_pwm <= 0)
             {
                g_motor_data.motor_pwm = 0;
             }
#if DEBUG_ENABLE                                       /* 发送基本参数*/
             /* 选择通道 1，发送实际电流（波形显示）*/
             debug_send_wave_data( 1 ,g_motor_data.current);
             /* 选择通道 2，发送目标电流（波形显示）*/
             debug_send_wave_data( 2 ,g_current_pid.SetPoint);
             /* 选择通道 3，发送占空比（波形显示）*/
             debug_send_wave_data( 3 ,g_motor_data.motor_pwm * 100 / 8400 );
#endif
```

```
                motor_pwm_set(g_motor_data.motor_pwm);    /* 设置占空比 */
            }
            val = 0;
        }
        val ++;
    }
}
```

该函数我们只介绍定时器 6 相关的程序，进入更新中断回调函数后，所执行的代码逻辑如下：

第一步，判断是不是定时器 6 的寄存器基地址，如果是则获取编码器的计数总值，然后每隔 5ms 计算一次电机速度。

第二步，每隔 50ms 进行一次 PID 计算，在计算 PID 之前，需要判断 g_run_flag 是否为 1，如果是则说明电机已经启动，可以开始 PID 计算。

第三步，进行电流环 PID 计算，返回的输出存放在 motor_pwm_temp 这个变量中；接着进行一阶低通滤波，返回的输出存放在 g_motor_data.motor_pwm 这个成员中，该成员用于设置 PWM 的占空比，除此之外，我们还需要对返回的输出进行限制，这一点非常重要。

第四步，发送实际电流、目标电流以及占空比的波形数据到上位机，最后设置 PWM 的占空比，进而控制电机的电流。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key;
    uint16_t t;
    uint8_t debug_cmd = 0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口 1 初始化，用于上位机调试 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    pid_init();                                 /* 初始化 PID 参数 */
    atim_timx_cplm_pwm_init(8400 - 1 , 0);      /* 168Mhz 的计数频率 */
    dcmotor_init();                             /* 初始化电机 */
    gtim_timx_encoder_chy_init(0XFFFF, 0);      /* 编码器定时器初始化 */
    btim_timx_int_init(1000 - 1 , 84 - 1);      /* 基本定时器初始化，1ms 计数周期 */
    adc_nch_dma_init();                         /* 初始化 ADC、DMA */

#if DEBUG_ENABLE                                /* 开启调试 */

    debug_init();                               /* 初始化调试 */
    debug_send_motorcode(DC_MOTOR);             /* 上传电机类型（直流有刷电机）*/
    debug_send_motorstate(IDLE_STATE);          /* 上传电机状态（空闲） */

    /* 同步数据（选择第 1 组 PID,目标电流地址,P,I,D 参数）到上位机 */
    debug_send_initdata(TYPE_PID1, (float *)(&g_current_pid.SetPoint),
                        KP, KI, KD);
#endif

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10, 10, 200, 16, 16, "DcMotor Test", g_point_color);
    lcd_show_string(10, 30, 200, 16, 16, "KEY0:Start forward", g_point_color);
    lcd_show_string(10, 50, 200, 16, 16, "KEY1:Start backward", g_point_color);
    lcd_show_string(10, 70, 200, 16, 16, "KEY2:Stop", g_point_color);

    delay_ms(500);

    while (1)
```

```
    {
        key = key_scan(0);                                  /* 按键扫描 */
        if(key == KEY0_PRES)                                /* 当 key0 按下 */
        {
            g_run_flag = 1;                                 /* 标记电机启动 */
            dcmotor_start();                                /* 启动电机 */
            g_current_pid.SetPoint += 20;

            if (g_current_pid.SetPoint >= 120)              /* 限制电流：120mA */
            {
                g_current_pid.SetPoint = 120;
            }
#if DEBUG_ENABLE
            debug_send_motorstate(RUN_STATE);               /* 上传电机状态（运行）*/
#endif
        }
        else if(key == KEY1_PRES)                           /* 当 key1 按下 */
        {
            g_current_pid.SetPoint -= 20;

            if (g_current_pid.SetPoint < 60)                /* 目标电流小于 60mA */
            {
                dcmotor_stop();                             /* 停止电机 */
                pid_init();                                 /* 重置 pid 参数 */
                g_run_flag = 0;                             /* 标记电机停止 */
                g_motor_data.motor_pwm = 0;
                motor_pwm_set(g_motor_data.motor_pwm);      /* 设置电机转向、速度 */
#if DEBUG_ENABLE
                debug_send_motorstate(BREAKED_STATE);       /* 上传电机状态（刹车） */
                debug_send_initdata(TYPE_PID1,
                                    (float *)(&g_current_pid.SetPoint), KP, KI, KD);
#endif
            }
        }
        else if(key == KEY2_PRES)                           /* 当 key2 按下 */
        {
            dcmotor_stop();                                 /* 停止电机 */
            pid_init();                                     /* 重置 pid 参数 */
            g_run_flag = 0;                                 /* 标记电机停止 */
            g_motor_data.motor_pwm = 0;
            motor_pwm_set(g_motor_data.motor_pwm);          /* 设置电机转向、速度 */
#if DEBUG_ENABLE
            debug_send_motorstate(BREAKED_STATE);           /* 上传电机状态（刹车）*/
            debug_send_initdata(TYPE_PID1, (float *)(&g_current_pid.SetPoint),
                                KP, KI, KD);
#endif
        }

#if DEBUG_ENABLE
        /* 查询接收 PID 助手的 PID 参数 */
        debug_receive_pid(TYPE_PID1, (float *)&g_current_pid.Proportion,
        (float *)&g_current_pid.Integral, (float *)&g_current_pid.Derivative);

        debug_set_point_range(120, 0, 120);                 /* 设置目标调节范围 */

        debug_cmd = debug_receive_ctrl_code();              /* 读取上位机指令 */

        if (debug_cmd == BREAKED)                           /* 电机刹车 */
        {
            dcmotor_stop();                                 /* 停止电机 */
            pid_init();                                     /* 重置 pid 参数 */
            g_run_flag = 0;                                 /* 标记电机停止 */
```

```
            g_motor_data.motor_pwm = 0;
            motor_pwm_set(g_motor_data.motor_pwm);          /* 设置电机转向、速度 */
            debug_send_motorstate(BREAKED_STATE);           /* 上传电机状态（刹车）*/
            debug_send_initdata(TYPE_PID1, (float *)(&g_current_pid.SetPoint),
                                KP, KI, KD);
        }
        else if (debug_cmd == RUN_CODE)                     /* 电机运行 */
        {
            dcmotor_start();                                /* 启动电机 */
            g_current_pid.SetPoint = 60;                    /* 目标电流：60mA */
            g_run_flag = 1;                                 /* 标记电机启动 */
            debug_send_motorstate(RUN_STATE);               /* 上传电机状态（运行）*/
        }
#endif
        t++;
        if(t % 20 == 0)
        {
            lcd_dis();                                      /* 显示数据 */
            LED0_TOGGLE();                                  /* LED0(红灯) 翻转 */

#if DEBUG_ENABLE
            debug_send_current(g_motor_data.current * 0.001f, 0, 0);/* 发送电流 */
            debug_send_speed(g_motor_data.speed);           /* 发送速度 */
#endif
        }
        delay_ms(10);
    }
}
```

main.c 的代码逻辑如下：

第一步，初始化相关的外设，例如定时器、串口以及 ADC 等。

第二步，初始化 PID 参数、上位机调试，它们分别调用的是 pid_init 和 debug_init 函数。

第三步，上传电机的状态（空闲）、类型（直流有刷电机）、PID 参数到上位机，它们分别调用的是 debug_send_motorcode、debug_send_motorstate 以及 debug_send_initdata 函数。

第四步，在 while 循环里面检测按键是否按下，如果 key0 按下，则目标电流增加 20，上传电机状态（运行）；如果 key1 按下，则目标电流减小 20，上传电机状态（运行）；当目标电流小于 60mA 或者 key2 按下时，停止电机，重置 PID 参数，并重新上传电机状态（刹车）以及 PID 参数到上位机。

第五步，接收上位机下发的 PID 参数、设置目标电流的调节范围，它们分别调用的是 debug_receive_pid 以及 debug_set_point_range 函数。我们这里重点介绍一下后者，debug_set_point_range 函数的第一、二个入口参数分别传入目标电流的最大值、最小值，第三个入口参数传入的是电机电流的最大突变值。

第六步，调用 debug_receive_ctrl_code 函数，接收上位机下发的命令。如果上位机的命令为 BREAKED（刹车），则停止电机，重置 PID 参数，并重新上传电机状态（刹车）以及 PID 参数到上位机；如果上位机的命令是 RUN_CODE，则开启电机，设置目标电流为 60mA，上传电机状态（运行）到上位机。

第七步，每隔 200ms 更新一次数据到屏幕，发送实际速度、实际电流值到上位机。

11.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 上显示按键功能、占空比以及电机电流信息，当我们按下 KEY0，目标电流将增大；按下 KEY1，目标电流将减小；按下 KEY2，电机将停止。我们再打开 PID 调试助手，选择对应的串口端口，我这边是 COM7，接着选择通道 1 和 2，点击“开始”按钮，即可开始显示波形，如下图 11.4.1 所示：

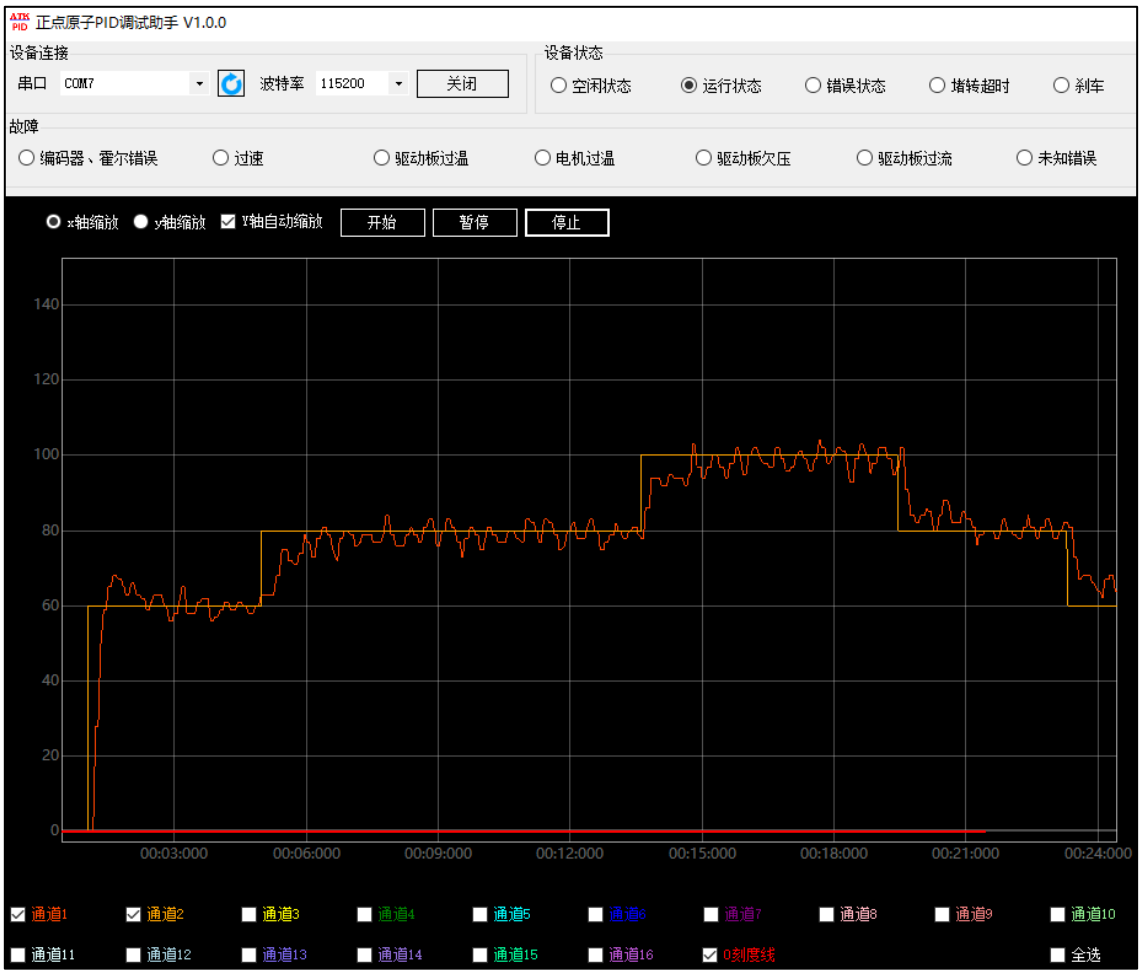


图 11.4.1 电流环 PID 控制波形

图 11.4.1 中，橙线代表目标电流，红线代表实际电流，当我们按下 KEY0，目标电流增大，橙线先发生变化，而红线（实际电流）会逐渐靠近橙线（目标电流）；按下 KEY1，目标电流将减小，曲线的变化同理；按下 KEY2，电机将停止。

注意：1、电流环的波形存在小幅振荡属于正常现象，如果希望波形更稳定，可以适当地调整 PID 系数以及增大滤波次数；2、滤波次数越多则系统的响应越慢，大家需要在系统的响应速度和稳定性之间寻找平衡点；3、开发板上电或者复位之后，需要等待电流稳定（大约 3s）再启动电机；4、PID 系数并不是通用的，如果 PID 曲线不理想，大家需要根据自己的实际系统去调节。

第 12 章 直流有刷电机位置环控制实现

本章我们主要来学习直流有刷电机位置环 PID 控制的原理，并实现位置环 PID 控制的实验。

本章分为如下几个小节：

12.1 位置环 PID 控制原理
12.2 硬件设计
12.3 程序设计
12.4 下载验证

12.1 位置环 PID 控制原理

位置环 PID 控制的原理非常简单，我们用编码器计数总值代表电机的位置，然后把 PID 控制流程中的控制对象换成电机位置即可，如图 12.1.1 所示：

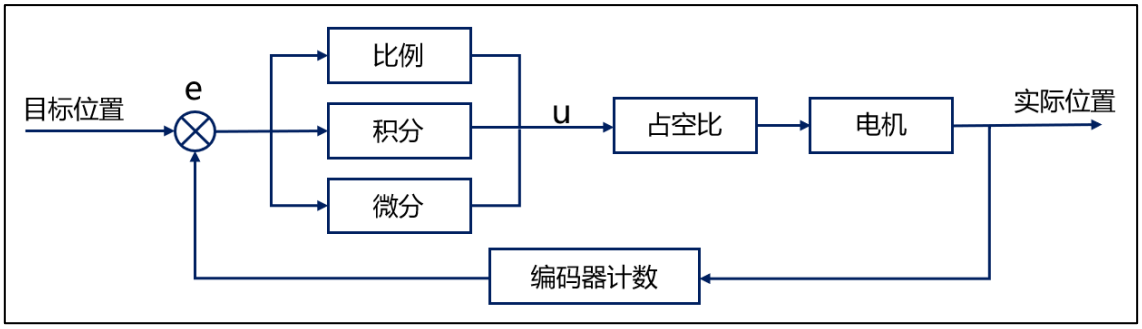


图 12.1.1 位置环 PID 控制流程

图 12.1.1 中，我们首先设置目标位置，系统会计算出偏差 e，然后将偏差输入到 PID 控制的三个环节中，PID 计算后的输出值用于控制 PWM 的占空比，进而控制电机的位置。

12.2 硬件设计

1. 例程功能

1、本实验以电机开发板的直流有刷电机驱动接口 1 为例，基于编码器测速实验，加入位置环 PID 控制算法，对电机的位置进行闭环控制。

2、当按键 0 按下，就增大目标位置值，电机正转到目标位置；当按键 1 按下，就减小目标位置值，电机反转到目标位置；当按键 2 按下，电机回到初始位置。

3、屏幕显示按键功能、占空比、目标位置以及实际位置。

4、串口 1 和上位机进行数据通信。

5、LED0 闪烁指示程序运行。

2. 硬件资源

1）LED 灯
LED0 - PE0
2）独立按键
KEY0 - PE2
KEY1 - PE3
KEY2 - PE4
3）定时器 1、3、6
TIM1 正常输出通道 PA8
TIM1 互补输出通道 PB13

TIM3 编码器 A 相输入通道 PC6
TIM3 编码器 B 相输入通道 PC7
4）SD（刹车）信号输出 PF10
5）串口 1
USART1_TX PB6（发送）
USART1_RX PB7（接收）

3. 原理图

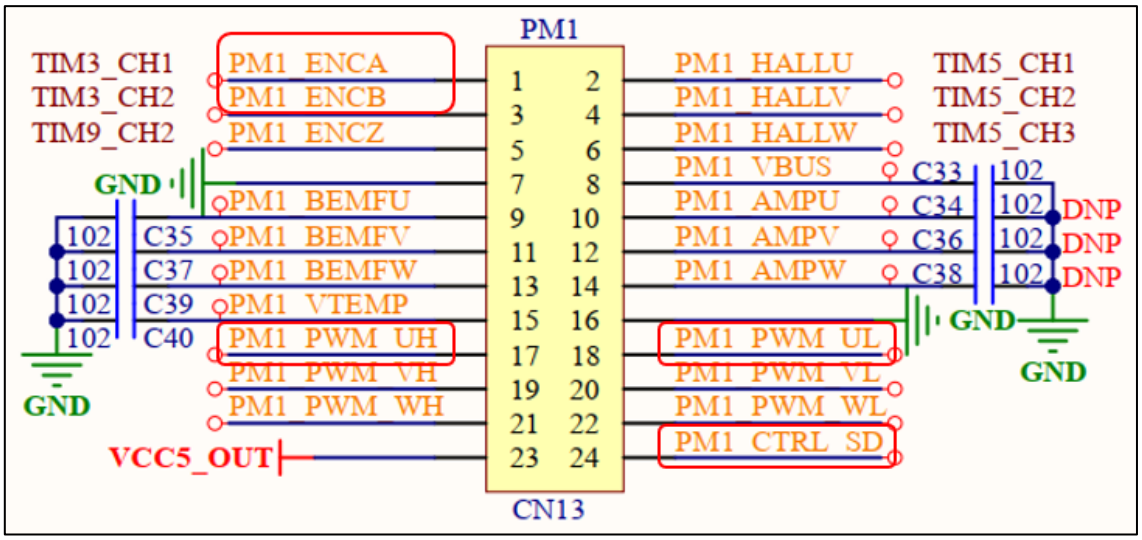


图 12.2.1 直流有刷电机接口原理图

图 12.2.1 就是我们 DMF407 电机开发板的直流有刷电机接口 1 原理图，本实验我们除了用到基础驱动所需的引脚，还需要用到 PM1_ENCA（PC6）、PM1_ENCB（PC7）这两个引脚，它们分别用于连接编码器的 A、B 两相。

本实验的硬件接线部分和编码器测速实验一模一样，这里就不再赘述，大家可以回顾编码器测速实验的内容。

12.3 程序设计

本实验所用到的基础驱动、编码器测速的代码在前面实验都有介绍过了。我们在程序解析中只讲解位置环 PID 控制相关的函数，下面介绍一下位置环 PID 控制的配置步骤。

位置环 PID 控制的配置步骤

1）配置相关定时器

配置基础驱动、编码器测速相关的定时器，此部分内容和编码器测速实验一致。

2）初始化串口 1

初始化串口 1，开启串口接收中断，串口 1 在 PID 控制中用于上位机通信。

注意：在 PID 控制的代码中，串口 1 仅用于 PID 数据上传，尽量不要输出其他信息，否则有可能影响 PID 数据。

3）定义 PID 参数结构体变量

为了方便管理 PID 相关的控制量，我们需要定义一个 PID 参数结构体变量，方法如下：

```
PID_TypeDef  g_location _pid;      /* 位置环 PID 参数结构体 */
```

4）初始化 PID 参数

把 PID 控制系统的目标位置值、期望输出值、累计偏差等清零，然后配置 PID 系数。

5）初始化上位机调试

调用 debug_init 函数初始化所需内存，为上位机的调试做准备。

6）编写中断服务函数

在定时器 6 的更新中断回调函数里面进行位置环 PID 计算，计算后的结果用于控制 PWM 的占空比。

12.3.1 程序流程图

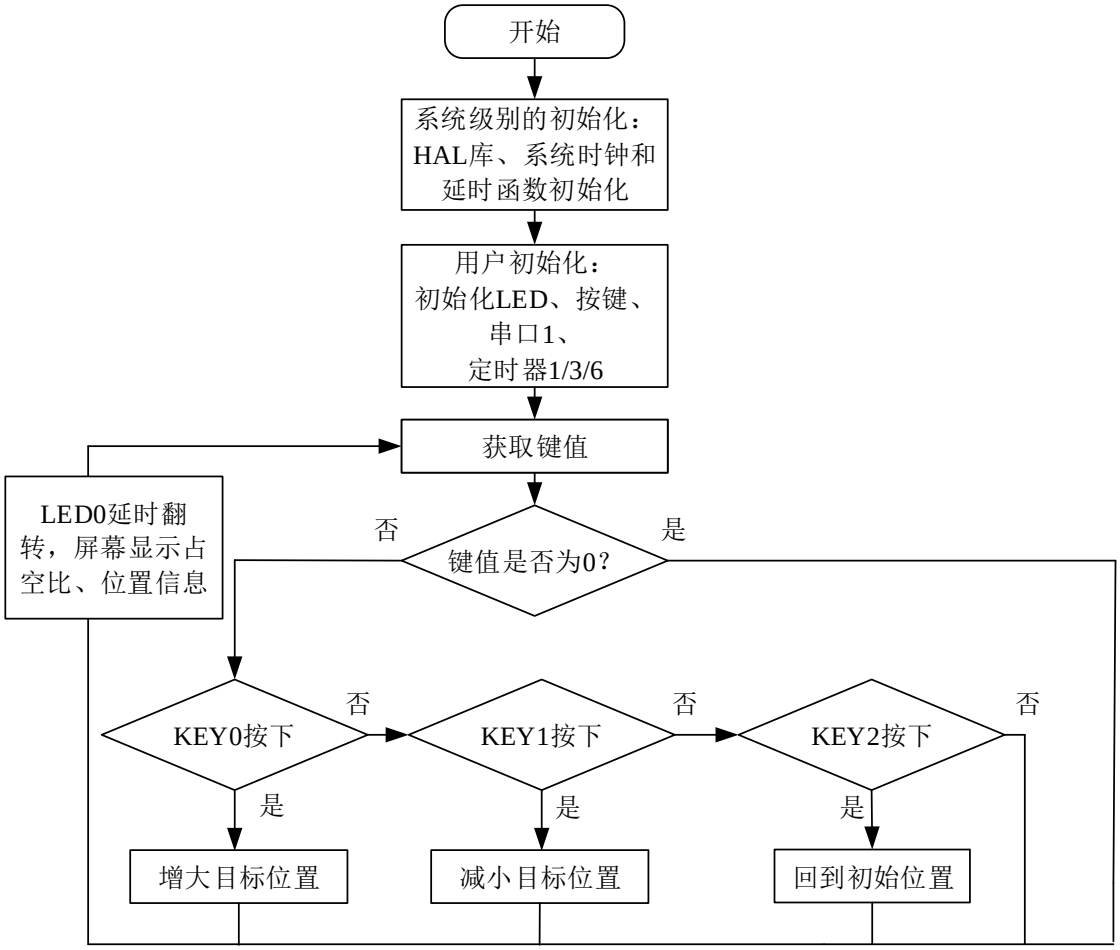


图 12.3.1.1 位置环 PID 控制程序流程图

12.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。关于基础驱动、编码器测速以及上位机协议的代码，这里不再赘述，大家可以回顾相应的章节。首先我们看电流环 PID 控制的相关源码，其包括两个文件：pid.c 和 pid.h。

首先看 pid.h 头文件的几个宏定义：

```
/* PID 相关参数 */

#define  INCR_LOCT_SELECT  1     /* 0：位置式，1：增量式 */

#if INCR_LOCT_SELECT

/* 增量式 PID 参数相关宏 */
#define  KP      15.0f           /* P 参数*/
#define  KI      0.00f           /* I 参数*/
#define  KD      7.50f           /* D 参数*/

#define  SMAPLSE_PID_SPEED  50   /* 采样周期 单位 ms */

#else
```

```
/* 位置式 PID 参数相关宏 */
#define  KP      15.0f           /* P 参数*/
#define  KI      0.00f           /* I 参数*/
#define  KD      7.5f            /* D 参数*/

#define  SMAPLSE_PID_SPEED  50  /* 采样周期 单位 ms */

#endif

/*PID 结构体*/
typedef struct
{
    __IO float  SetPoint;        /* 设定目标 */
    __IO float  ActualValue;     /* 期望输出值 */
    __IO float  SumError;        /* 误差累计 */
    __IO float  Proportion;      /* 比例常数 P */
    __IO float  Integral;        /* 积分常数 I */
    __IO float  Derivative;      /* 微分常数 D */
    __IO float  Error;           /* Error[1] */
    __IO float  LastError;       /* Error[-1] */
    __IO float  PrevError;       /* Error[-2] */
} PID_TypeDef;

extern PID_TypeDef  g_location_pid;  /*位置 PID 参数结构体*/
```

可以把上面的宏定义分成两部分，第一部分是 PID 计算方式以及 PID 系数的宏定义，我们可以通过改变 INCR_LOCT_SELECT 这个宏的值来选择相应的 PID 计算方式，第二部分则是 PID 参数相关的结构体，这个结构体用于管理 PID 控制所需要的控制量，本实验中我们定义了位置环 PID 参数的结构体变量 g_ location _pid。

下面看 pid.c 的程序，这里我们只介绍 PID 初始化函数，关于 PID 闭环控制的函数介绍请回顾 8.3 章节，PID 初始化函数定义如下：

```
/**
 * @brief       pid 初始化
 * @param       无
 * @retval      无
 */
void pid_init(void)
{
    g_location_pid.SetPoint = 0.0;       /* 设定目标值 */
    g_location_pid.ActualValue = 0.0;    /* 期望输出值 */
    g_location_pid.SumError = 0.0;       /* 积分值 */
    g_location_pid.Error = 0.0;          /* Error[1] */
    g_location_pid.LastError = 0.0;      /* Error[-1] */
    g_location_pid.PrevError = 0.0;      /* Error[-2] */
    g_location_pid.Proportion = KP;      /* 比例常数 Proportional Const */
    g_location_pid.Integral = KI;        /* 积分常数 Integral Const */
    g_location_pid.Derivative = KD;      /* 微分常数 Derivative Const */
}
```

该函数主要是将 PID 控制系统的目标位置值、期望输出值、累计偏差等清零，然后配置 PID 系数。

最后要介绍的是定时器的更新中断回调函数，它在 dcmotor_time.c 中实现，具体的定义如下：

```
/**
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @note        此函数会被定时器中断函数共同调用的
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
```

```
    int32_t motor_pwm_temp = 0;
    static uint8_t val = 0;

    /* 定时器 3 相关程序 */
    if (htim->Instance == TIM3)
    {
       /* 判断 CR1 的 DIR 位 */
       if(__HAL_TIM_IS_TIM_COUNTING_DOWN(&g_timx_encode_chy_handle))
       {
           g_timx_encode_count--;              /* DIR 位为 1，也就是递减计数 */
       }
       else
       {
           g_timx_encode_count++;              /* DIR 位为 0，也就是递增计数 */
       }
    }

    /* 定时器 6 相关程序 */
    else if (htim->Instance == TIM6)
    {
       int Encode_now = gtim_get_encode();    /* 获取编码器值，用于计算速度 */

       speed_computer(Encode_now, 5);         /* 5ms 计算一次速度 */

       g_motor_data.location = Encode_now;    /* 获取当前计数总值，用于位置闭环控制 */

       if (val % SMAPLSE_PID_SPEED == 0)      /* 进行一次 pid 计算 */
       {
           if (g_run_flag)                     /* 判断电机是否启动了*/
           {
               /* 位置环 PID 计算，输出比较值（占空比）*/
               motor_pwm_temp = increment_pid_ctrl(&g_location_pid,
                                                   g_motor_data.location);
               /* 进行一阶低通滤波 */
               g_motor_data.motor_pwm = (int32_t)((g_motor_data.motor_pwm * 0.5)
                                       + (motor_pwm_temp * 0.5));

               if (g_motor_data.motor_pwm >= 4200)    /* 限制占空比 */
               {
                   g_motor_data.motor_pwm = 4200;
               }
               else if (g_motor_data.motor_pwm <= -4200)
               {
                   g_motor_data.motor_pwm = -4200;
               }
#if DEBUG_ENABLE                                          /* 发送基本参数*/

               /* 选择通道 1，发送实际位置（波形显示）*/
               debug_send_wave_data( 1 ,g_motor_data.location);

               /* 选择通道 2，发送目标位置（波形显示）*/
               debug_send_wave_data( 2 ,g_location_pid.SetPoint);
#endif
               motor_pwm_set(g_motor_data.motor_pwm);     /* 设置占空比 */
           }
           val = 0;
       }
       val ++;
    }
}
```

该函数我们只介绍定时器 6 相关的程序，进入更新中断回调函数后，所执行的代码逻辑如下：

第一步，判断是不是定时器 6 的寄存器基地址，如果是则获取编码器的计数总值并存入

变量 Encode_now 中，接着计算电机速度，把变量 Encode_now 的值存入 g_motor_data.location 这个成员中。

第二步，每隔 50ms 进行一次 PID 计算，在计算 PID 之前，需要判断 g_run_flag 是否为 1，如果是则说明电机已经启动，可以开始 PID 计算。

第三步，进行位置环 PID 计算，返回的输出存放在 motor_pwm_temp 这个变量中；接着进行一阶低通滤波，返回的输出存放在 g_motor_data.motor_pwm 这个成员中，该成员用于设置 PWM 的占空比，除此之外，我们还需要对返回的输出进行限制，这一点非常重要。

第四步，发送实际位置、目标位置的波形数据到上位机，最后设置 PWM 的占空比，进而控制电机的位置。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key;
    uint16_t t;
    uint8_t debug_cmd = 0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口 1 初始化，用于上位机调试 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    pid_init();                                 /* 初始化 PID 参数 */
    atim_timx_cplm_pwm_init(8400 - 1 , 0);      /* 168Mhz 的计数频率 */
    dcmotor_init();                             /* 初始化电机 */
    gtim_timx_encoder_chy_init(0XFFFF, 0);      /* 编码器定时器初始化 */
    btim_timx_int_init(1000 - 1 , 84 - 1);      /* 基本定时器初始化，1ms 计数周期 */
    adc_nch_dma_init();                         /* 初始化 ADC、DMA */

#if DEBUG_ENABLE                                /* 开启调试 */

    debug_init();                               /* 初始化调试 */
    debug_send_motorcode(DC_MOTOR);             /* 上传电机类型（直流有刷电机） */
    debug_send_motorstate(IDLE_STATE);          /* 上传电机状态（空闲） */

    /* 同步数据（选择第 1 组 PID,目标位置地址,P,I,D 参数）到上位机 */
    debug_send_initdata(TYPE_PID1, (float *)(&g_location_pid.SetPoint),
                        KP, KI, KD);
#endif

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10, 10, 200, 16, 16, "DcMotor Test", g_point_color);
    lcd_show_string(10, 30, 200, 16, 16, "KEY0:Start forward", g_point_color);
    lcd_show_string(10, 50, 200, 16, 16, "KEY1:Start backward", g_point_color);
    lcd_show_string(10, 70, 200, 16, 16, "KEY2:Stop", g_point_color);

    while (1)
    {
        key = key_scan(0);                      /* 按键扫描 */
        if(key == KEY0_PRES)                    /* 当 key0 按下 */
        {
            g_run_flag = 1;                     /* 标记电机启动 */
            dcmotor_start();                    /* 开启电机 */

            /* 正转一圈，电机旋转圈数 = 目标计数值变化量 / 44 / 30 */
            g_location_pid.SetPoint  += 1320;
```

```
            if (g_location_pid.SetPoint >= 6600) /* 限制电机位置（正转最大 5 圈） */
            {
                g_location_pid.SetPoint = 6600;
            }

#if DEBUG_ENABLE
            debug_send_motorstate(RUN_STATE);    /* 上传电机状态（运行） */
#endif
        }
        else if(key == KEY1_PRES)                /* 当 key1 按下 */
        {
            g_run_flag = 1;                      /* 标记电机启动 */
            dcmotor_start();                     /* 开启电机 */
            g_location_pid.SetPoint  -= 1320;    /* 反转一圈 */

            if (g_location_pid.SetPoint <= -6600) /* 限制电机位置（反转最大 5 圈）*/
            {
                g_location_pid.SetPoint = -6600;
            }
#if DEBUG_ENABLE
            debug_send_motorstate(RUN_STATE);     /* 上传电机状态（运行） */
#endif
        }
        else if(key == KEY2_PRES)                 /* 当 key2 按下 */
        {
            g_location_pid.SetPoint = 0;          /* 恢复初始位置 */
        }
#if DEBUG_ENABLE
        /* 查询接收 PID 助手的 PID 参数 */
        debug_receive_pid(TYPE_PID1, (float *)&g_location_pid.Proportion,
     (float *)&g_location_pid.Integral, (float *)&g_location_pid.Derivative);

        debug_set_point_range(6600, -6600, 6600);   /* 设置目标调节范围 */

        debug_cmd = debug_receive_ctrl_code();      /* 读取上位机指令 */

        if (debug_cmd == HALT_CODE)                 /* 电机停机 */
        {
            g_location_pid.SetPoint = 0;            /* 恢复初始位置 */
        }
        else if (debug_cmd == RUN_CODE)             /* 电机运行 */
        {
            g_run_flag = 1;                         /* 标记电机启动 */
            dcmotor_start();                        /* 开启电机 */
            g_location_pid.SetPoint = 1320;         /* 设置目标位置 */
            debug_send_motorstate(RUN_STATE);       /* 上传电机状态（运行） */
        }
#endif
        t++;
        if(t % 20 == 0)
        {
            lcd_dis();                              /* 显示数据 */
            LED0_TOGGLE();                          /* LED0(红灯)翻转 */
            g_debug.encode_p = g_motor_data.location;   /* 传入编码器当前总计数值 */
            debug_upload_data(&g_debug, TYPE_HAL_ENC);  /* 发送编码器当前总计数值 */
        }
        delay_ms(10);
    }
}
```

main.c 的代码逻辑如下：

第一步，初始化相关的外设，例如定时器、串口等。

第二步，初始化 PID 参数、上位机调试，它们分别调用的是 pid_init 和 debug_init 函数。

第三步，上传电机的状态（空闲）、类型（直流有刷电机）、PID 参数到上位机，它们分别调用的是 debug_send_motorcode、debug_send_motorstate 以及 debug_send_initdata 函数。

第四步，在 while 循环里面检测按键是否按下，如果 key0 按下，则目标位置增加 1320，也就是目标计数值增加 1320，根据公式：电机旋转圈数 = 目标计数值变化量 /44/30，电机将会正转一圈；如果 key1 按下，则目标位置减小 1320，电机将会反转一圈，然后上传电机状态（运行）；如果 key2 按下，电机将回到初始位置。

第五步，接收上位机下发的 PID 参数、设置目标位置的调节范围，它们分别调用的是 debug_receive_pid 以及 debug_set_point_range 函数。我们这里重点介绍一下后者，debug_set_point_range 函数的第一、二个入口参数分别传入目标位置的最大值、最小值，它们的正负号代表电机的正反转方向，第三个入口参数传入的是电机位置的最大突变值。

第六步，调用 debug_receive_ctrl_code 函数，接收上位机下发的命令。如果上位机的命令为 HALT_CODE（停机），则让电机回到初始位置；如果上位机的命令是 RUN_CODE，则开启电机，设置目标位置为 1320，上传电机状态（运行）到上位机。

第七步，每隔 200ms 更新一次数据到屏幕，调用 debug_upload_data 函数发送编码器位置值（计数总值）到上位机。

12.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 上显示按键功能、占空比以及电机位置信息，当我们按下 KEY0，目标位置将增大，电机正转；按下 KEY1，目标位置将减小，电机反转；按下 KEY2，电机将回到初始位置。我们再打开 PID 调试助手，选择对应的串口端口，我这边是 COM7，接着选择通道 1 和 2，点击“开始”按钮，即可开始显示波形，如下图 12.4.1 所示：

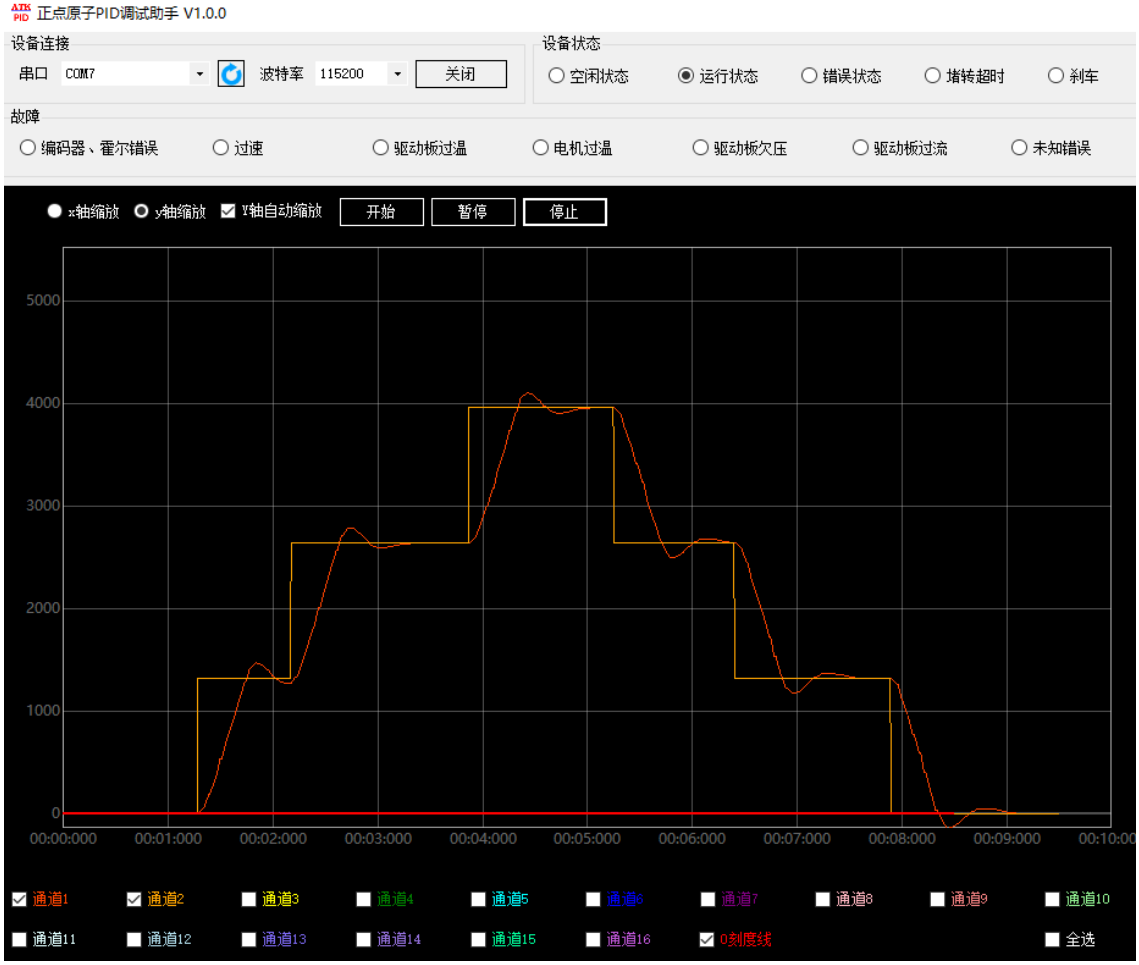


图 12.4.1 位置环 PID 控制波形

图 12.4.1 中，橙线代表目标位置，红线代表实际位置，当我们按下 KEY0，目标位置增大，橙线先发生变化，而红线（实际位置）会逐渐靠近橙线（目标位置）；按下 KEY1，目标速度将减小，曲线的变化同理；按下 KEY2，电机将回到初始位置，目标位置将为 0。

注意：1、本实验需要使用 USB 数据线连接开发板的串口 1 到电脑，并启动电机之后才会有波形变化；2、如果发现波形不对，请检查电机的 M+和 M-接线；3、PID 系数并不是通用的，如果 PID 曲线不理想，大家需要根据自己的实际系统去调节。

第 13 章 直流有刷电机三环控制实现

本章我们主要来学习直流有刷电机三环（电流环+速度环+位置环）PID 控制的原理，并实现三环 PID 控制的实验。

本章分为如下几个小节：

13.1 三环 PID 控制原理
13.2 硬件设计
13.3 程序设计
13.4 下载验证

13.1 三环 PID 控制原理

三环 PID 控制就是将三个 PID 控制系统（例如：电流环、速度环以及位置环）串联起来，然后对前一个系统（外环）的输出进行偏差的计算，计算结果作为后一个系统（内环）的输入，具体的控制流程如图 13.1.1 所示：

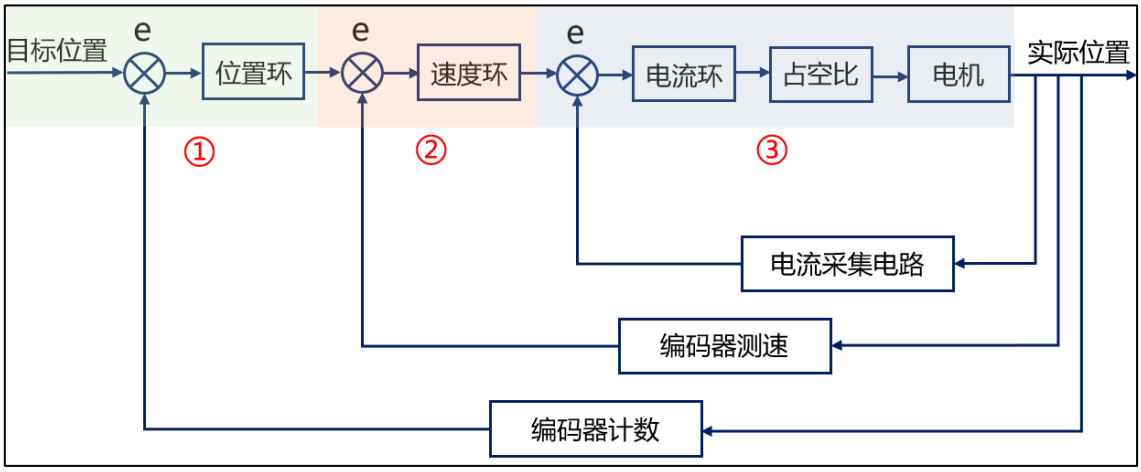


图 13.1.1 三环 PID 控制流程

图 13.1.1 中，电机位置是优先控制的对象，三环 PID 控制的流程如下：

① 首先设置目标位置，系统会计算出位置偏差，然后将偏差输入到位置环（最外环）；

② 位置环的输出和实际速度进行偏差的计算，计算后的偏差输入到速度环（次外环）；

③ 速度环的输出和实际电流进行偏差的计算，计算后的偏差输入到电流环（内环），电流环的输出用于控制 PWM 的占空比，进而控制电机的位置。

注意：多环 PID 控制系统的原理是相通的，大家掌握了三环 PID 控制的内容，其他的多环控制系统就可以迎刃而解了。

13.2 硬件设计

1. 例程功能

1、本实验以电机开发板的直流有刷电机驱动接口 1 为例，融合速度环、电流环以及位置环 PID 控制实验，加入三环 PID 控制算法，对电流、速度以及电机位置进行闭环控制。

2、当按键 0 按下，就增大目标位置值，电机正转到目标位置；当按键 1 按下，就减小目标位置值，电机反转到目标位置；当按键 2 按下，电机回到初始位置。

3、屏幕显示按键功能、占空比、目标位置以及实际位置。

4、串口 1 和上位机进行数据通信。

5、LED0 闪烁指示程序运行。

2. 硬件资源

1）LED 灯
LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器 1、3、6
TIM1 正常输出通道 PA8
TIM1 互补输出通道 PB13
TIM3 编码器 A 相输入通道 PC6
TIM3 编码器 B 相输入通道 PC7
4）ADC
ADC1 通道 8 PB0（电流）
5）SD（刹车）信号输出 PF10
6）串口 1
USART1_TX PB6（发送）
USART1_RX PB7（接收）

3. 原理图

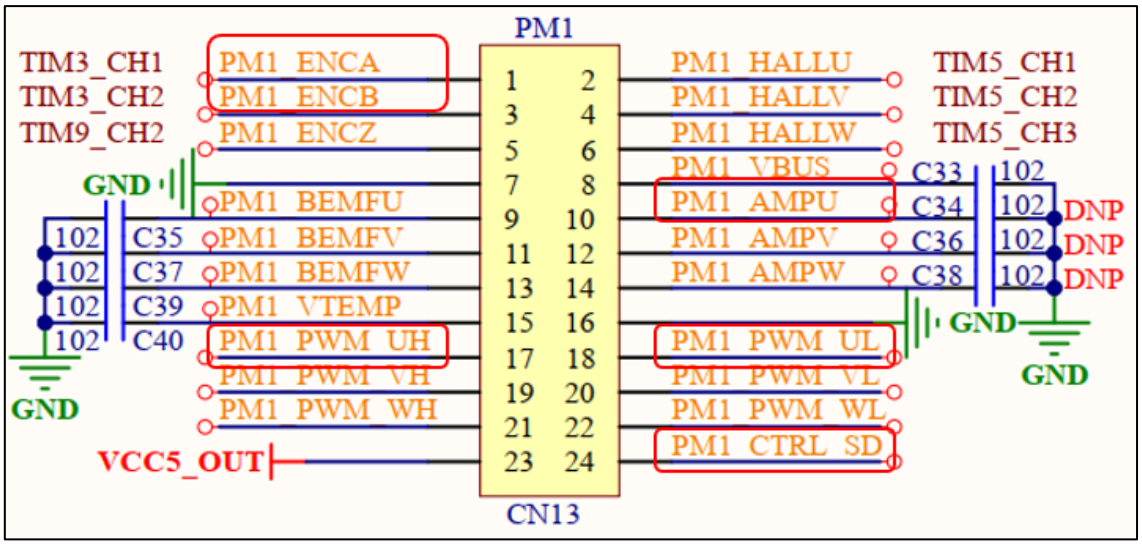


图 13.2.1 直流有刷电机接口原理图

图 13.2.1 就是我们 DMF407 电机开发板的直流有刷电机接口 1 原理图，本实验我们除了用到基础驱动所需的引脚，还需要用到 PM1_ENCA（PC6）、PM1_ENCB（PC7）和 PM1_AMPU（PB0）这 3 个引脚。PC6、PC7 分别用于连接编码器的 A、B 两相，PB0 用于检测电流采集电路对应的输出电压。

本实验的硬件接线部分和上一章节一致，这里不再赘述。

13.3 程序设计

本实验所用到的基础驱动、编码器测速以及电流采集的代码在前面实验都有介绍过了。我们在程序解析中只讲解三环 PID 控制相关的函数，下面介绍一下三环 PID 控制的配置步骤。

三环 PID 控制的配置步骤

1）配置相关定时器

配置基础驱动、编码器测速相关的定时器，实现基础驱动以及编码器测速功能。

2）配置相关 ADC

配置电流采集相关 ADC，实现电流采集功能。

4）初始化串口 1

初始化串口 1，开启串口接收中断，串口 1 在 PID 控制中用于上位机通信。

注意：在 PID 控制的代码中，串口 1 仅用于 PID 数据上传，尽量不要输出其他信息，否则有可能影响 PID 数据。

5）定义 PID 参数结构体变量

为了方便管理 PID 相关的控制量，我们需要定义 3 个 PID 参数结构体变量，方法如下：

```
PID_TypeDef  g_location_pid;     /* 位置环 PID 参数结构体 */
PID_TypeDef  g_speed_pid;        /* 速度环 PID 参数结构体 */
PID_TypeDef  g_current_pid;      /* 电流环 PID 参数结构体 */
```

6）初始化 PID 参数

把三环 PID 控制系统的目标值、期望输出值、累计偏差等清零，然后分别配置各个环的 PID 系数。

7）初始化上位机调试

调用 debug_init 函数初始化所需内存，为上位机的调试做准备。

8）编写中断服务函数

在定时器 6 的更新中断回调函数里面进行三环 PID 计算，计算后的结果用于控制 PWM 的占空比。

13.3.1 程序流程图

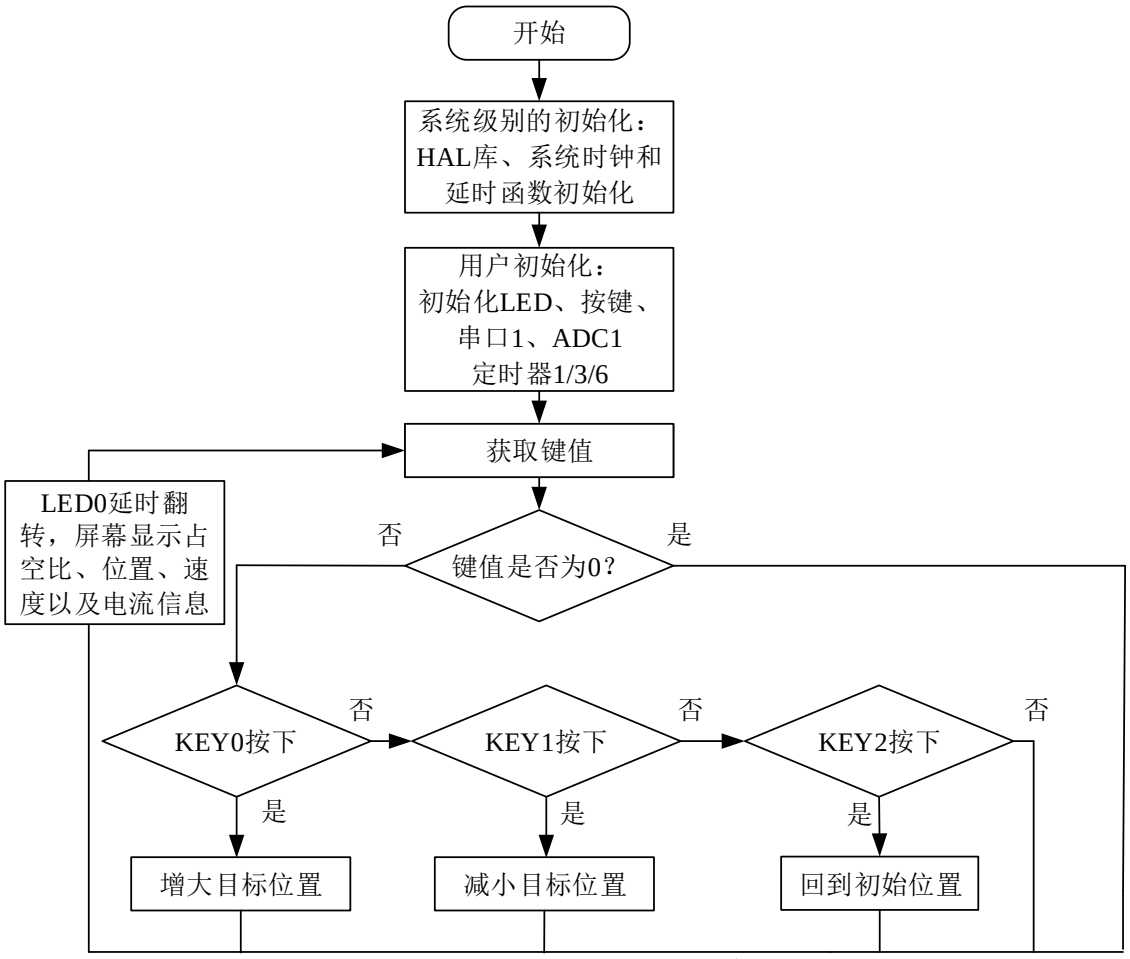


图 13.3.1.1 三环 PID 控制程序流程图

13.3.2 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。首先我们看三环 PID 控制的相关源码，其包括两个文件：pid.c 和 pid.h。

首先看 pid.h 头文件的几个宏定义：

```
/* PID 相关参数 */

#define  INCR_LOCT_SELECT  0      /* 0：位置式，1：增量式 */

#if INCR_LOCT_SELECT

/* 定义位置环 PID 参数相关宏 */
#define  L_KP      0.06f          /* P 参数 */
#define  L_KI      0.00f          /* I 参数 */
#define  L_KD      0.01f          /* D 参数 */

/* 定义速度环 PID 参数相关宏 */
#define  S_KP      5.00f          /* P 参数 */
#define  S_KI      0.30f          /* I 参数 */
#define  S_KD      0.01f          /* D 参数 */

/* 定义电流环 PID 参数相关宏 */
#define  C_KP      8.00f          /* P 参数 */
#define  C_KI      4.00f          /* I 参数 */
#define  C_KD      1.00f          /* D 参数 */
#define  SMAPLSE_PID_SPEED  50    /* 采样周期 单位 ms */

#else

/* 定义位置环 PID 参数相关宏 */
#define  L_KP      0.06f          /* P 参数 */
#define  L_KI      0.00f          /* I 参数 */
#define  L_KD      0.01f          /* D 参数 */

/* 定义速度环 PID 参数相关宏 */
#define  S_KP      5.00f          /* P 参数 */
#define  S_KI      0.30f          /* I 参数 */
#define  S_KD      0.01f          /* D 参数 */

/* 定义电流环 PID 参数相关宏 */
#define  C_KP      8.00f          /* P 参数 */
#define  C_KI      4.00f          /* I 参数 */
#define  C_KD      1.00f          /* D 参数 */
#define  SMAPLSE_PID_SPEED  50    /* 采样周期 单位 ms */

#endif

/*PID 结构体*/
typedef struct
{
    __IO float  SetPoint;          /* 目标值 */
    __IO float  ActualValue;       /* 期望输出值 */
    __IO float  SumError;          /* 误差累计 */
    __IO float  Proportion;        /* 比例常数 P */
    __IO float  Integral;          /* 积分常数 I */
    __IO float  Derivative;        /* 微分常数 D */
    __IO float  Error;             /* Error[1] */
    __IO float  LastError;         /* Error[-1] */
    __IO float  PrevError;         /* Error[-2] */
} PID_TypeDef;
```

```
extern PID_TypeDef  g_location_pid; /* 位置环 PID 参数结构体 */
extern PID_TypeDef  g_speed_pid;    /* 速度环 PID 参数结构体 */
extern PID_TypeDef  g_current_pid;  /* 电流环 PID 参数结构体 */
```

可以把上面的宏定义分成两部分，第一部分是 PID 计算方式以及 PID 系数的宏定义，我们可以通过改变 INCR_LOCT_SELECT 这个宏的值来选择相应的 PID 计算方式，第二部分则是 PID 参数相关的结构体，这个结构体用于管理 PID 控制所需要的控制量，本实验中我们定义了位置环、速度环以及电流环 PID 参数的结构体变量，它们分别是 g_ location _pid、g_speed_pid、g_current_pid。

下面看 pid.c 的程序，我们先看 PID 初始化函数，其定义如下：

```
/**
 * @brief       pid 初始化
 * @param       无
 * @retval      无
 */
void pid_init(void)
{
    /* 初始化位置环 PID 参数 */
    g_location_pid.SetPoint = 0.0;          /* 目标值 */
    g_location_pid.ActualValue = 0.0;       /* 期望输出值 */
    g_location_pid.SumError = 0.0;          /* 积分值*/
    g_location_pid.Error = 0.0;             /* Error[1]*/
    g_location_pid.LastError = 0.0;         /* Error[-1]*/
    g_location_pid.PrevError = 0.0;         /* Error[-2]*/
    g_location_pid.Proportion = L_KP;       /* 比例常数 Proportional Const */
    g_location_pid.Integral = L_KI;         /* 积分常数 Integral Const */
    g_location_pid.Derivative = L_KD;       /* 微分常数 Derivative Const */

    /* 初始化速度环 PID 参数 */
    g_speed_pid.SetPoint = 0.0;             /* 目标值 */
    g_speed_pid.ActualValue = 0.0;          /* 期望输出值 */
    g_speed_pid.SumError = 0.0;             /* 积分值 */
    g_speed_pid.Error = 0.0;                /* Error[1] */
    g_speed_pid.LastError = 0.0;            /* Error[-1] */
    g_speed_pid.PrevError = 0.0;            /* Error[-2] */
    g_speed_pid.Proportion = S_KP;          /* 比例常数 Proportional Const */
    g_speed_pid.Integral = S_KI;            /* 积分常数 Integral Const */
    g_speed_pid.Derivative = S_KD;          /* 微分常数 Derivative Const */

    /* 初始化电流环 PID 参数 */
    g_current_pid.SetPoint = 0.0;           /* 目标值 */
    g_current_pid.ActualValue = 0.0;        /* 期望输出值 */
    g_current_pid.SumError = 0.0;           /* 积分值*/
    g_current_pid.Error = 0.0;              /* Error[1]*/
    g_current_pid.LastError = 0.0;          /* Error[-1]*/
    g_current_pid.PrevError = 0.0;          /* Error[-2]*/
    g_current_pid.Proportion = C_KP;        /* 比例常数 Proportional Const */
    g_current_pid.Integral = C_KI;          /* 积分常数 Integral Const */
    g_current_pid.Derivative = C_KD;        /* 微分常数 Derivative Const */
}
```

该函数主要是将三环 PID 控制系统的目标值、期望输出值、累计偏差等清零，然后配置 PID 系数。

接下来我们看积分限幅函数，其定义如下：

```
/**
 * @brief       积分限幅
 * @param       *PID：PID 结构体变量地址
 * @param       max_limit：最大值
 * @param       min_limit：最小值
 * @retval      无
```

```
 */
void integral_limit( PID_TypeDef *PID , float max_limit, float min_limit )
{
    if (PID->SumError >= max_limit)                          /* 超过限幅 */
    {
        PID->SumError = max_limit;                           /* 限制积分 */
    }
    else if (PID->SumError <= min_limit)                     /* 超过限幅 */
    {
        PID->SumError = min_limit;
    }
}
```

此函数可以对位置式 PID 的积分进行限幅，避免出现积分深度饱和的问题。大家在使用该函数之前，需要先调整好 PID 系数，再根据系统能达到的最大累计偏差来设置限幅。

最后要介绍的是定时器的更新中断回调函数，它在 dcmotor_time.c 中实现，具体的定义如下：

```
/**
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @note        此函数会被定时器中断函数共同调用的
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    static uint8_t val = 0;

    /* 定时器 3 相关程序 */
    if (htim->Instance == TIM3)
    {
      /* 判断 CR1 的 DIR 位 */
      if(__HAL_TIM_IS_TIM_COUNTING_DOWN(&g_timx_encode_chy_handle))
      {
          g_timx_encode_count--;                 /* DIR 位为 1，也就是递减计数 */
      }
      else
      {
          g_timx_encode_count++;                 /* DIR 位为 0，也就是递增计数 */
      }
    }

    /* 定时器 6 相关程序 */
    else if (htim->Instance == TIM6)
    {
      int Encode_now = gtim_get_encode();      /* 获取编码器值，用于计算速度 */

      speed_computer(Encode_now, 5);           /* 5ms 计算一次速度 */

      if (val % SMAPLSE_PID_SPEED == 0)        /* 进行一次 pid 计算 */
      {
          if (g_run_flag)                      /* 判断电机是否启动了 */
          {
              /* 获取当前编码器总计数值，用于位置闭环控制 */
              g_motor_data.location = (float)Encode_now;

              integral_limit(&g_location_pid , 1000 ,-1000); /* 位置环积分限幅 */
              integral_limit(&g_speed_pid , 200 ,-200);      /* 速度环积分限幅 */
              integral_limit(&g_current_pid , 150 ,-150);    /* 电流环积分限幅 */

              /* 设置闭环死区，避免小幅度位置抖动 */
              if((g_location_pid.Error <= 20)&&(g_location_pid.Error >= -20))
              {
                  g_location_pid.Error = 0;      /* 偏差太小了，直接清零 */
```

```
                g_location_pid.SumError = 0;    /* 清除积分 */
            }

            g_motor_data.motor_pwm = increment_pid_ctrl(&g_location_pid,
                   g_motor_data.location);      /* 位置环 PID 控制（最外环） */

            if (g_motor_data.motor_pwm >= 120) /* 限制外环输出（目标速度） */
            {
                g_motor_data.motor_pwm = 120;
            }
            else if (g_motor_data.motor_pwm <= -120)
            {
                g_motor_data.motor_pwm = -120;
            }
            /* 设置目标速度，外环输出作为内环输入 */
            g_speed_pid.SetPoint = g_motor_data.motor_pwm;

            g_motor_data.motor_pwm = increment_pid_ctrl(&g_speed_pid,
                   g_motor_data.speed);         /* 速度环 PID 控制（次外环） */

            if ( g_motor_data.motor_pwm > 0)    /* 判断速度环输出值是否为正数 */
            {
                dcmotor_dir(0);                  /* 输出为正数，设置电机正转 */
            }
            else
            {
                /* 输出取反 */
                g_motor_data.motor_pwm = -g_motor_data.motor_pwm;
                dcmotor_dir(1);                  /* 设置电机反转 */
            }

            if (g_motor_data.motor_pwm >= 100) /* 限制外环输出（目标电流） */
            {
                g_motor_data.motor_pwm = 100;
            }
            /* 设置目标电流，外环输出作为内环输入 */
            g_current_pid.SetPoint = g_motor_data.motor_pwm;

            g_motor_data.motor_pwm = increment_pid_ctrl(&g_current_pid,
                     g_motor_data.current);        /* 电流环 PID 控制（内环） */

            if (g_motor_data.motor_pwm >= 8200)   /* 限制占空比 */
            {
                g_motor_data.motor_pwm = 8200;
            }
            else if (g_motor_data.motor_pwm <= 0) /* 滤掉无效输出 */
            {
                g_motor_data.motor_pwm = 0;
            }

#if DEBUG_ENABLE  /* 发送基本参数*/

            /* 选择通道 1，发送实际位置（波形显示）*/
            debug_send_wave_data( 1 ,g_motor_data.location);
            /* 选择通道 2，发送目标位置（波形显示）*/
            debug_send_wave_data( 2 ,g_location_pid.SetPoint);
#endif
            dcmotor_speed(g_motor_data.motor_pwm); /* 设置占空比 */
        }
        val = 0;
    }
    val ++;
  }
}
```

该函数我们只介绍定时器 6 相关的程序，进入更新中断回调函数后，所执行的代码逻辑如下：

第一步，判断是不是定时器 6 的寄存器基地址，如果是则获取编码器的计数总值并存入变量 Encode_now 中，接着计算电机速度。

第二步，每隔 50ms 进行一次 PID 计算，在计算 PID 之前，需要判断 g_run_flag 是否为 1，如果是则说明电机已经启动，可以开始 PID 计算。

第三步，把变量 Encode_now 的值存入 g_motor_data.location 这个成员中，然后对三环的积分进行限幅并设置闭环死区。当系统偏差值进入死区范围之内，我们直接让偏差等于 0，并清除累计偏差，此时 PID 系统不再参与控制，这样可以避免小幅度偏差带来的位置抖动。

第四步，进行位置环 PID 计算并限制其输出，然后将位置环的输出作为速度环的目标值；速度环 PID 计算完后，我们先判断其输出，如果输出是正数，则设置电机正转，如果输出是负数，则将输出取反，设置电机反转。因为电流环 PID 的目标值和输出都是正数，我们无法通过电流环的输出来确定电机的转向，所以这里需要通过速度环的输出来确定电机的转向，并保证电流环的目标值是正数。

第五步，将速度环的输出作为电流环的目标值，然后进行电流环 PID 计算并限制其输出。

第六步，发送实际位置、目标位置的波形数据到上位机，最后设置 PWM 的占空比，进而控制电机的位置。

在 main.c 里面编写如下代码：

```
int main(void)
{
    uint8_t key;
    uint16_t t;
    uint8_t debug_cmd = 0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口 1 初始化，用于上位机调试 */
    led_init();                                 /* 初始化 LED */
    lcd_init();                                 /* 初始化 LCD */
    key_init();                                 /* 初始化按键 */
    pid_init();                                 /* 初始化 PID 参数 */
    atim_timx_cplm_pwm_init(8400 - 1 , 0);      /* 168Mhz 的计数频率 */
    dcmotor_init();                             /* 初始化电机 */
    gtim_timx_encoder_chy_init(0XFFFF, 0);      /* 编码器定时器初始化 */
    btim_timx_int_init(1000 - 1 , 84 - 1);      /* 基本定时器初始化，1ms 计数周期 */
    adc_nch_dma_init();

#if DEBUG_ENABLE                                /* 开启调试 */

    debug_init();                               /* 初始化调试 */
    debug_send_motorcode(DC_MOTOR);             /* 上传电机类型（直流有刷电机） */
    debug_send_motorstate(IDLE_STATE);          /* 上传电机状态（空闲） */

    /* 同步数据 PID 参数到上位机 ，无论同步哪一组数据，目标值地址只能是外环 PID 的 */
    debug_send_initdata(TYPE_PID1, (float *)(&g_location_pid.SetPoint),
                        L_KP, L_KI, L_KD);  /* 位置环 PID 参数（PID1）*/
    debug_send_initdata(TYPE_PID2, (float *)(&g_location_pid.SetPoint),
                        S_KP, S_KI, S_KD);  /* 速度环 PID 参数（PID2）*/
    debug_send_initdata(TYPE_PID3, (float *)(&g_location_pid.SetPoint),
                        C_KP, C_KI, C_KD);  /* 电流环 PID 参数（PID3）*/

#endif

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10, 10, 200, 16, 16, "DcMotor Test", g_point_color);
```

```
    lcd_show_string(10, 30, 200, 16, 16, "KEY0:Start forward", g_point_color);
    lcd_show_string(10, 50, 200, 16, 16, "KEY1:Start backward", g_point_color);
    lcd_show_string(10, 70, 200, 16, 16, "KEY2:Stop", g_point_color);

    while (1)
    {
        key = key_scan(0);                              /* 按键扫描 */
        if(key == KEY0_PRES)                            /* 当 key0 按下 */
        {
            g_run_flag = 1;                             /* 标记电机启动 */
            dcmotor_start();                            /* 开启电机 */
            /* 正转一圈，电机旋转圈数 = 编码器总计数值 / 44 / 30 */
            g_location_pid.SetPoint += 1320;

            if (g_location_pid.SetPoint >= 6600)        /* 限制电机位置 */
            {
                g_location_pid.SetPoint = 6600;
            }
#if DEBUG_ENABLE
            debug_send_motorstate(RUN_STATE);           /* 上传电机状态（运行） */
#endif
        }

        else if(key == KEY1_PRES)                       /* 当 key1 按下 */
        {
            g_run_flag = 1;                             /* 标记电机启动 */
            dcmotor_start();                            /* 开启电机 */
            g_location_pid.SetPoint -= 1320;            /* 反转一圈 */

            if (g_location_pid.SetPoint <= -6600)       /* 限制电机位置 */
            {
                g_location_pid.SetPoint = -6600;
            }
#if DEBUG_ENABLE
            debug_send_motorstate(RUN_STATE);           /* 上传电机状态（运行） */
#endif
        }

        else if(key == KEY2_PRES)                       /* 当 key2 按下 */
        {
            g_location_pid.SetPoint = 0;                /* 恢复初始位置 */
        }

#if DEBUG_ENABLE

        /* 接收 PID 助手设置的位置环 PID 参数 */
        debug_receive_pid(TYPE_PID1,(float *)&g_location_pid.Proportion,
        (float *)&g_location_pid.Integral,(float *)&g_location_pid.Derivative);

        /* 接收 PID 助手设置的速度环 PID 参数 */
        debug_receive_pid(TYPE_PID2,(float *)&g_speed_pid.Proportion,
        (float *)&g_speed_pid.Integral,(float *)&g_speed_pid.Derivative);

        /* 接收 PID 助手设置的电流环 PID 参数 */
        debug_receive_pid(TYPE_PID3,(float *)&g_current_pid.Proportion,
        (float *)&g_current_pid.Integral,(float *)&g_current_pid.Derivative);

        debug_set_point_range(6600, -6600, 6600);   /* 设置目标调节范围 */

        debug_cmd = debug_receive_ctrl_code();      /* 读取上位机指令 */

        if (debug_cmd == HALT_CODE)                 /* 电机停机 */
        {
```

```
            g_location_pid.SetPoint = 0;                /* 恢复初始位置 */
        }
        else if (debug_cmd == RUN_CODE)                 /* 电机运行 */
        {
            g_run_flag = 1;                             /* 标记电机启动 */
            dcmotor_start();                            /* 开启电机 */
            g_location_pid.SetPoint = 1320;             /* 设置目标位置 */
            debug_send_motorstate(RUN_STATE);           /* 上传电机状态（运行） */
        }
#endif
        t++;
        if(t % 20 == 0)
        {
            lcd_dis();
            LED0_TOGGLE();                                  /* LED0(红灯) 翻转 */
#if DEBUG_ENABLE
            debug_send_speed(g_motor_data.speed);           /* 发送速度 */
            g_debug.encode_p = g_motor_data.location;       /* 传入编码器当前总计数值 */
            debug_upload_data(&g_debug, TYPE_HAL_ENC);      /* 发送编码器当前总计数值 */
#endif
        }
        delay_ms(10);
    }
}
```

main.c 的代码逻辑如下：

第一步，初始化相关的外设，例如定时器、串口以及 ADC 等。

第二步，初始化 PID 参数、上位机调试，它们分别调用的是 pid_init 和 debug_init 函数。

第三步，同步电机的状态（空闲）、类型（直流有刷电机）、PID 参数到上位机。需要注意的是，在多环控制中，只有最外环的目标值可以手动设置，因此，无论同步哪一组 PID 参数，目标值地址只能是最外环 PID 的（本实验最外环为位置环）。

第四步，在 while 循环里面检测按键是否按下，如果 key0 按下，则目标位置增加 1320，也就是目标计数值增加 1320，根据公式：电机旋转圈数 = 目标计数值变化量 /44/30，电机将会正转一圈；如果 key1 按下，则目标位置减小 1320，电机将会反转一圈，然后上传电机状态（运行）；如果 key2 按下，电机将回到初始位置。

第五步，接收上位机下发的 PID 参数、设置目标位置的调节范围。

第六步，调用 debug_receive_ctrl_code 函数，接收上位机下发的命令。如果上位机的命令为 HALT_CODE（停机），则让电机回到初始位置；如果上位机的命令是 RUN_CODE，则开启电机，设置目标位置为 1320，上传电机状态（运行）到上位机。

第七步，每隔 200ms 更新一次数据到屏幕，然后调用 debug_upload_data 函数发送编码器位置值（计数总值）到上位机。

注意：我们可以通过串口来打印系统各环的累计偏差，以此确定各个环的积分限幅范围，切记不能用串口 1 打印，否则可能会影响 PID 数据。

根据经验，在只有电流环或者速度环组成的系统中，限幅的范围需要比累计偏差的最大值稍大一点，否则系统可能存在静态误差；在纯位置环或者最外环是位置环的系统中，限幅的范围可以比累计偏差最大值要小，大家可以去实际感受一下。需要注意的是，不同的 PID 系数对累计偏差的影响可能很大，因此，我们在重新调整 PID 系数之后，需要查看积分限幅范围是否合理。

13.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 上显示按键功能、占空比、位置、速度以及电流信息，当我们按下 KEY0，目标位置将增大，电机正转；按下 KEY1，目标位置将减小，电机反转；按下 KEY2，电机将回到初始位置。我们再打开 PID 调试助手，选择对应的串口端口，我这边是 COM7，接着选择通道 1 和 2，点击“开始”按钮，即可开始显示波形，如下图 13.4.1 所示：

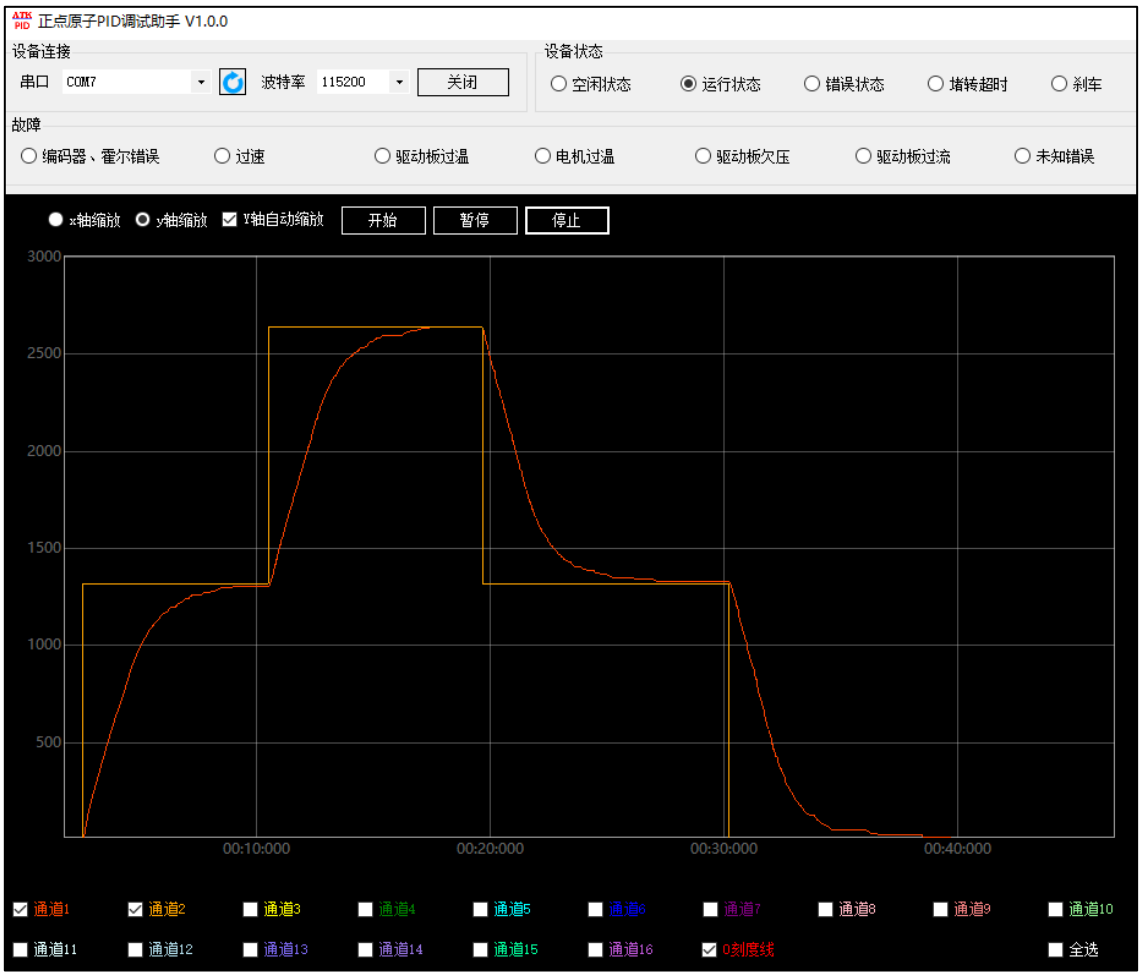


图 13.4.1 三环 PID 控制波形

图 13.4.1 中，橙线代表目标位置，红线代表实际位置，当我们按下 KEY0，目标位置增大，橙线先发生变化，而红线（实际位置）会逐渐靠近橙线（目标位置）；按下 KEY1，目标速度将减小，曲线的变化同理；按下 KEY2，电机将回到初始位置，目标位置将为 0。

注意：1、本实验需要使用 USB 数据线连接开发板的串口 1 到电脑，并启动电机之后才会有波形变化；2、如果发现波形不对，请检查电机的 M+和 M-接线；3、PID 系数并不是通用的，如果 PID 曲线不理想，大家需要根据自己的实际系统去调节。

第 14 章 舵机控制

本章我们的学习目的主要是了解舵机的驱动原理以及结构组成，并使用 STM32 将舵机驱动起来。 本章分为如下几个小节：

14.1 舵机的简介以及分类
14.2 舵机的结构
14.3 舵机的工作原理以及控制原理
14.4 舵机参数介绍
14.5 舵机的控制实现

14.1 舵机的简介以及分类

舵机（Servo）实际上可以看作一个伺服电机，其拥有驱动电路、控制电路，只不过一般舵机工作在一定的角度范围内，最常见的就是 180 度的舵机，它适用于那些需要角度不断变化并可以保持的控制系统中。目前在高档遥控玩具，如飞机、潜艇模型、遥控机器人中已经得到了普遍应用。下图为常见舵机的实物图：

图 14.1.1 SG90 舵机

舵机按照控制电路分类，可以分为**数字舵机**和**模拟舵机**，它们的机械结构是完全一模一样的，其最大的区别体现在控制电路上，数字舵机的伺服控制器采用了数字电路（拥有 MCU 和晶振），而模拟舵机的控制器采用的是模拟电路。它们最主要的区别在于：模拟舵机需要给它不停的发送 PWM 信号，才能让它保持在规定的位置或者让它按照某个速度转动，数字舵机则只需要发送一次 PWM 信号就能保持在规定的某个位置。

按照旋转角度可分为 **180°与 360°**的舵机，180 度舵机只能在 0 度到 180 度（±90 度）之间运动； 360°舵机转动的方式和普通的电机类似，可以连续的转动，不过我们只可以控制它转动的方向和速度，不能调节转动角度。**本章我们主要讲解 180°的模拟舵机**。

14.2 舵机的结构

舵机的结构主要由以下几个部分组成：直流电机、控制电路、角度传感器、输出轴以及减速齿轮组。

角度传感器：用来实现位置反馈的，它会将其旋转后产生的电阻变化信号发送回控制电路，从而监控当前轴旋转的角度；

控制电路：用来驱动电机和接收 PWM 控制信号以及电位器反馈信号；

减速齿轮组：用来增大直流电机的扭矩。齿轮的材质一般有塑料齿轮、混合材料齿轮和金属齿轮。下面我们来看下舵机结构实物图，如下图所示

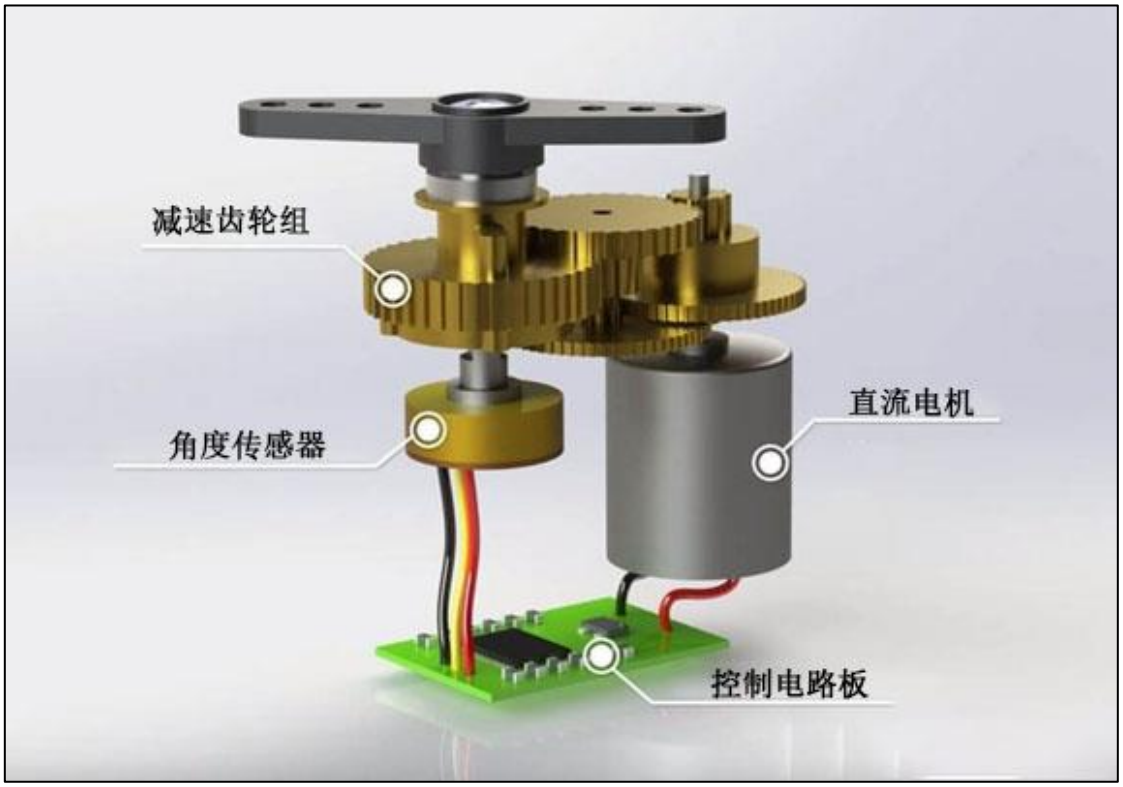


图 14.2.1 舵机的结构图

14.3 舵机的工作原理以及控制原理

1、**舵机的工作原理：**

①首先由控制电路板接收到信号线的控制信号（一般为 PWM 信号），进而控制直流电机转动；

②电机的输出轴和减速齿轮组的轴是相连的，所以只要电机一转动就会带动减速齿轮组转动；

③减速齿轮组又和角度传感器连接在一起，所以同时也会带动角度传感器转动；

④接着角度传感器会将当前的电压信号反馈到控制电路板，控制电路板会根据反馈的比例电压调节舵机的转动位置，当达到目标位置后停止，形成一个**闭环控制**系统。工作流程如下图所示：

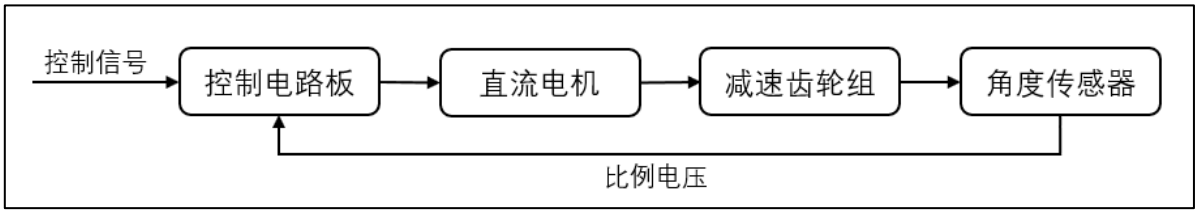


图 14.3.1 舵机的工作原理图

2、 舵机的控制原理：

舵机的输入线共有三条：如下图 14.3.2 所示，一般**棕色**表示 GND，**红色**表示 VCC，**橙色**表示信号脚，工作电压一般在 4.8～6V，不同的工作电压对应不同的转矩标准，即输出力矩不同，6V 的工作电压对应的力矩要大一些；根据不同的扭矩负载，工作电流可达 2A 以上。

图 14.3.2 舵机接口定义

其中信号线就是控制电路用于接受外部控制信号的，该信号一般周期为 20ms（频率：50HZ）的脉宽调制（PWM）信号。并且该脉冲的高电平持续时间为 0.5～2.5ms，对应舵盘旋转角度的 0～180°，所对应的控制关系如下图所示：

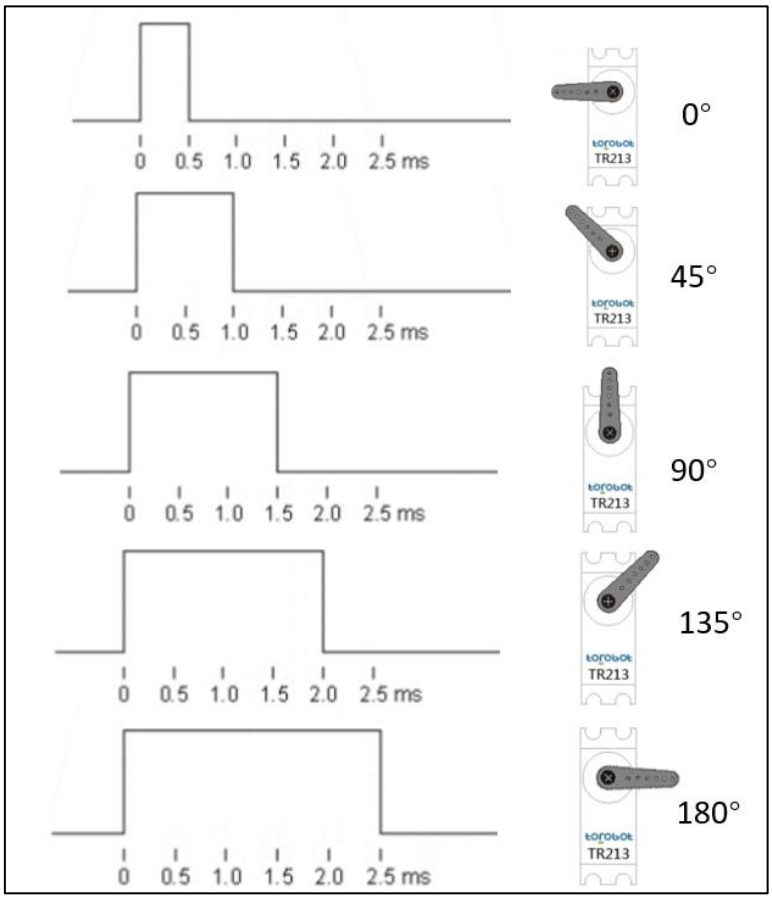


图 14.3.3 信号脉宽与舵盘位置关系图

从图 14.3.3 可以看出脉冲宽度与舵机旋转角度呈线性变化关系。也就是说我们只需要不断的提供一个脉冲周期为 20ms 的 PWM 信号给舵机，并且控制该脉冲信号的高电平持续时间占比，就可以控制输出轴保持在对应的角度上，注意角度范围不可超过 180°。

14.4 舵机的参数介绍

舵机的几个重要指标：

①响应速度：即在接收到目标位置之后多久才可以转到目标角度，一般舵机为 60 度每 0.17 秒，另外根据负载扭矩相应速度也不等。

②工作死区，死区数值越小，控制精度越高；数值越大，精度越小。

③材质：舵机材质一般有塑料齿轮和金属齿轮，金属材质相对的质量更好，价格也更高些。

④舵机的扭矩：根据应用场合可从飞行器模型舵机的 1.2KG/cm 到大扭矩舵机的 500~2000KG/cm 不等，不过一般扭矩越大舵机消耗电流也越大。

由此可见这类舵机具有以下这些特点：输出力矩大，稳定性好；体积小，易安装；控制简单等。所以现在不仅仅应用于航模运动中，在机器人的关节，飞机的舵面，以及机械手等应用也越来越广泛。

14.5 舵机的控制实现

本小节我们将使用 STM32 控制舵机转动到指定的角度位置。关于定时器如何输出 PWM 波的知识，请大家回顾高级定时器 PWM 输出实验的介绍。

14.5.1 硬件设计

1. 例程功能

F407 电机开发板全部有 3 个舵机接口，分别由 TIM8_CH1、TIM8_CH2、TIM8_CH3 控制，本实验需要将舵机接入对应的舵机接口，通过按键 KEY0 设置控制哪个接口的舵机，KEY1 增加舵机角度，最大 180°，KEY2 减小舵机，角度最小 0°，LED0 作为程序运行状态指示灯，闪烁周期 200ms。

2. 硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器 8 通道 1，将 TIM8_CH1 输出到 PI5。
定时器 8 通道 2，将 TIM8_CH2 输出到 PI6。
定时器 8 通道 3，将 TIM8_CH3 输出到 PI7。
4）舵机

3. 原理图

开发板设计了 3 组舵机接口，支持板载 5V 供电（注意该 5V 为 DC 口 12V 提供的，所以必须要接上 DC 口供电），原理图如下图：

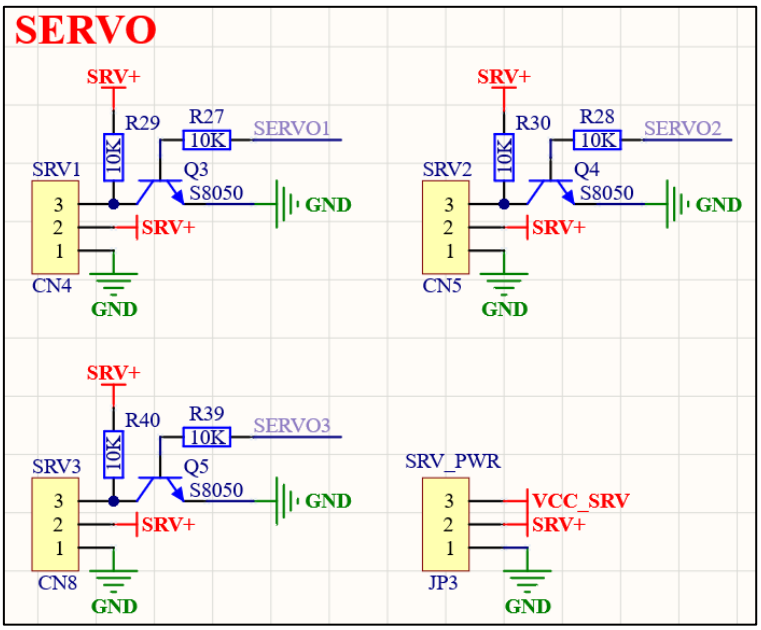


图 14.5.1.1 开发板舵机接口原理图

图 14.5.1.1 中的 JP3 的 SR+，是为舵机供电用的，它的来源可以选择板载 VCC_SRV（5V）供电，也可选择外部电源接入供电（针对驱动电压比较高的舵机，比如一些 12V 驱动的舵机），一般我们选择板载电源供电，所以首先我们**需要使用跳线帽将 SRV+与 VCC_SRV 短接一起**，这里要注意 JP3 跳帽处 VCC_SRV 的电源由 DC 电路提供的，所以 DC 口必须供电。开发板全部有三个舵机接口，其中 SERVOx 接舵机的信号线（通常信号线为橙色线），SRV+接舵机的电源正极（通常是红色线），GND 接电源负极（通常为棕色线）。

主要特别注意的点是舵机控制信号 SERVO1~3 并不是直接控制舵机，而是经过电平转换来控制舵机，这样既隔离了 IO，又可以控制各种不同电平的舵机。**电平转换电路实际就是一个反向器，因为是反向器，所以控制电平需要反过来**，比如要给舵机一个 2ms 高电平，周期 20ms 的信号，控制信号实际需要输出 2ms 低电平+18ms 高电平。

定时器属于 STM32F407 的内部资源，只需要软件设置好即可正常工作。外部需要将舵机接入对应开发板的舵机接口，同时还使用按键 KEY0、KEY1、KEY2 进行控制。我们这里接了三个舵机（只有一个舵机的话可接到其中一个接口）实物连接图如下：

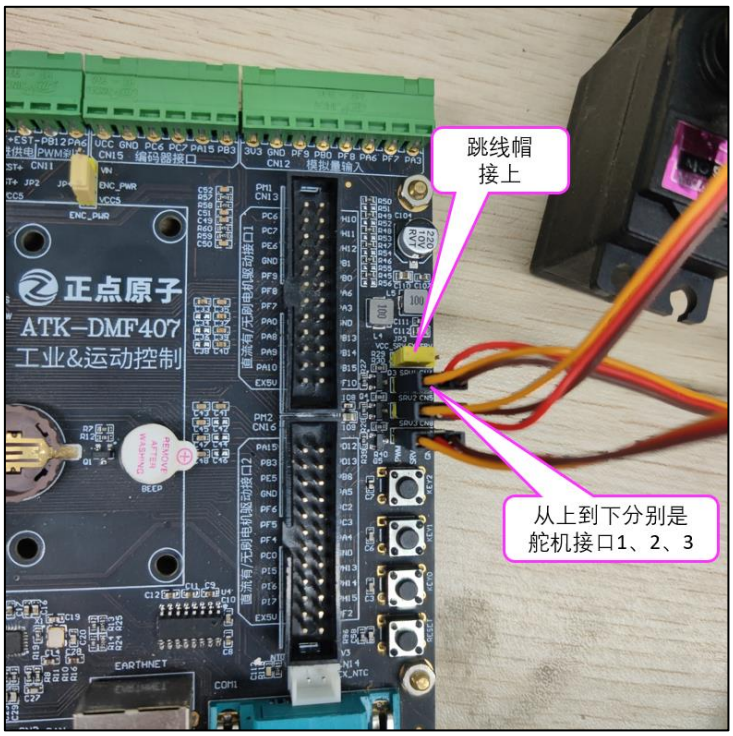


图 14.5.1.2 实物连接图

14.5.2 程序设计

14.5.2.1 定时器的 HAL 库驱动

本实验用到的 HAL 库驱动请回顾高级定时器 PWM 输出实验的介绍。下面介绍一下舵机控制的配置步骤。

舵机控制的配置步骤

1）开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出。

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1、通道 2 和通道 3，对应 IO 分别是 PI5、PI6、PI7，它们的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE();            /* 使能定时器 8 */
__HAL_RCC_GPIOI_CLK_ENABLE();           /* 开启 GPIOI 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数。

使用定时器的 PWM 模式功能时，我们调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数来完成对定时器底层以及其输出通道 IO 的初始化，包括：定时器及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

3）设置定时器为 PWM 模式，输出比较极性，比较值等参数。

在 HAL 库中，通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为 PWM1 模式或者 PWM2 模式，根据需求设置输出比较的极性，设置比较值（控制占空比）等。

本实验我们设置 TIM8 的通道 1、2、3 为 PWM2 模式，比较值我们设置为：0.5/20*(arr+1)，即默认设置 PWM 波的占空比为 2.5%，也就是 0.5ms 的脉宽，对应的舵机舵盘位置为 0° 。

14.5.2.2 程序流程图

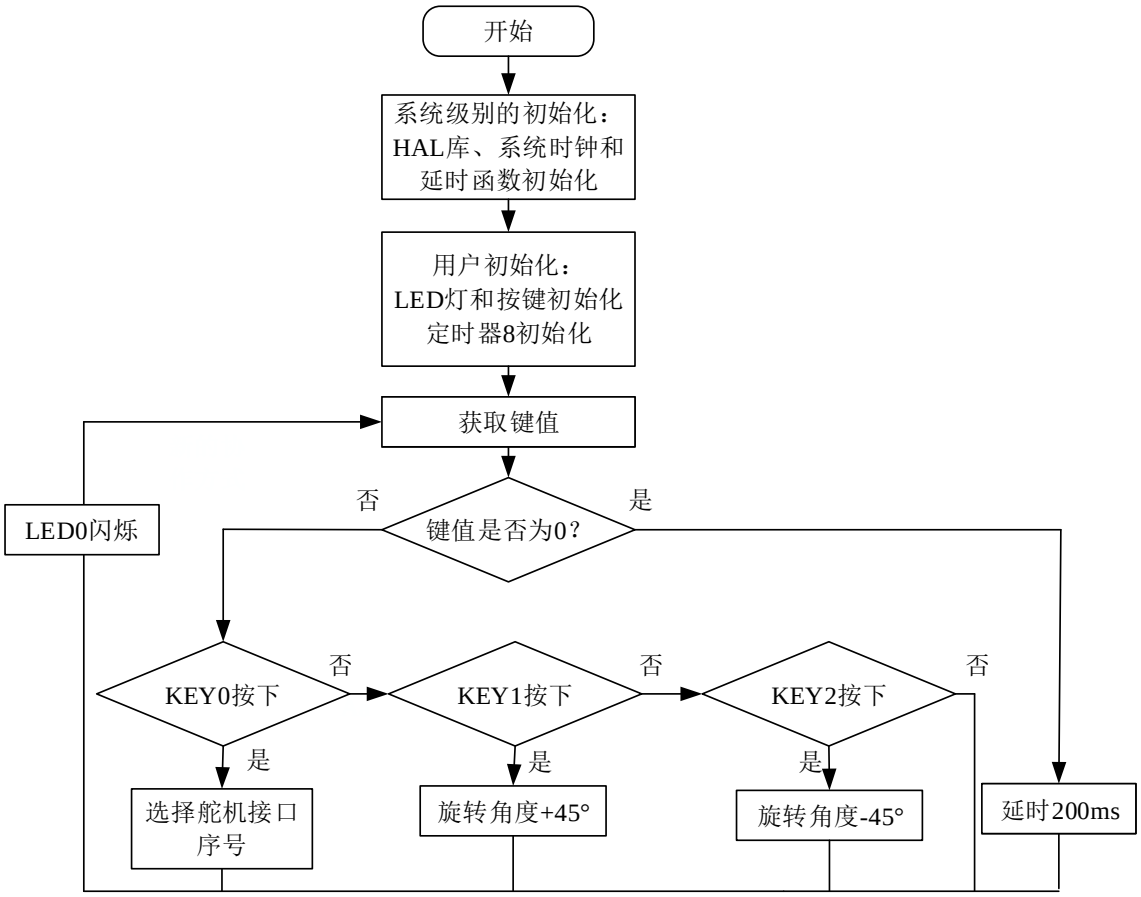


图 14.5.2.2 舵机控制流程图

14.5.2.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。舵机的控制源码主要有：高级定时器驱动源码和舵机角度设置源码四个文件：atim.c 和 atim.h 以及 steering_engine.c 和 steering_engine.h。源码中有明确的注释。

首先看 atim.h 头文件的几个宏定义：

```
/* 高级定时器 定义 */

 /* TIMX PWM 定义
 * 注意：通过修改这几个宏定义，可以支持 TIM1/TIM8 定时器
 */
#define ATIM_TIMX_PWM_CH1_GPIO_PORT             GPIOI
#define ATIM_TIMX_PWM_CH1_GPIO_PIN              GPIO_PIN_5
#define ATIM_TIMX_PWM_CH1_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)    /* PI 口时钟使能 */

#define ATIM_TIMX_PWM_CH2_GPIO_PORT             GPIOI
#define ATIM_TIMX_PWM_CH2_GPIO_PIN              GPIO_PIN_6
#define ATIM_TIMX_PWM_CH2_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)    /* PI 口时钟使能 */

#define ATIM_TIMX_PWM_CH3_GPIO_PORT             GPIOI
#define ATIM_TIMX_PWM_CH3_GPIO_PIN              GPIO_PIN_7
#define ATIM_TIMX_PWM_CH3_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)    /* PI 口时钟使能 */
#define ATIM_TIMX_PWM_CHY_GPIO_AF               GPIO_AF3_TIM8
```

```
#define ATIM_TIMX_PWM                           TIM8
#define ATIM_TIMX_PWM_IRQn                      TIM8_UP_TIM13_IRQn
#define ATIM_TIMX_PWM_IRQHandler                TIM8_UP_TIM13_IRQHandler
/* 通道 1 */
#define ATIM_TIMX_PWM_CH1                       TIM_CHANNEL_1
/* 通道 2 */
#define ATIM_TIMX_PWM_CH2                       TIM_CHANNEL_2
/* 通道 3 */
#define ATIM_TIMX_PWM_CH3                       TIM_CHANNEL_3
#define ATIM_TIMX_PWM_CHY_CLK_ENABLE()
do{ __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)      /* TIM8 时钟使能 */
```

可以把上面的宏定义分成两部分，第一部分是定时器 8 输入通道 1、2、3 对应的 IO 口的宏定义，第二部分则是定时器 8 输入通道 1、2、3 的相应宏定义。下面我们来看下 atim.c 程序。

```
/**
 * @brief       高级定时器 TIMX PWM 初始化函数
 * @note
 *              高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void atim_timx_pwm_chy_init(uint16_t arr, uint16_t psc)
{
    ATIM_TIMX_PWM_CHY_CLK_ENABLE();                     /* TIMX 时钟使能 */
    ATIM_TIMX_PWM_CH1_GPIO_CLK_ENABLE();                /* IO 时钟使能 */
    ATIM_TIMX_PWM_CH2_GPIO_CLK_ENABLE();                /* IO 时钟使能 */
    ATIM_TIMX_PWM_CH3_GPIO_CLK_ENABLE();                /* IO 时钟使能 */

    g_atimx_pwm_chy_handle.Instance = ATIM_TIMX_PWM;    /* 定时器 x */
    g_atimx_pwm_chy_handle.Init.Prescaler = psc;        /* 定时器分频 */
    g_atimx_pwm_chy_handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    g_atimx_pwm_chy_handle.Init.Period = arr;           /* 自动重装载值 */
    g_atimx_pwm_chy_handle.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    g_atimx_pwm_chy_handle.Init.AutoReloadPreload =
    TIM_AUTORELOAD_PRELOAD_DISABLE;                     /*使能 TIMx_ARR 进行缓冲*/
    g_atimx_pwm_chy_handle.Init.RepetitionCounter = 0;  /* 开始时不计数*/
    HAL_TIM_PWM_Init(&g_atimx_pwm_chy_handle);          /* 初始化 PWM */

    g_atimx_oc_pwm_chy_handle.OCMode = TIM_OCMODE_PWM2;/* 模式选择 PWM2 */
    g_atimx_oc_pwm_chy_handle.Pulse = 0.5/20*(arr+1);
    g_atimx_oc_pwm_chy_handle.OCPolarity = TIM_OCPOLARITY_HIGH;
    g_atimx_oc_pwm_chy_handle.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    g_atimx_oc_pwm_chy_handle.OCFastMode = TIM_OCFAST_DISABLE;
    g_atimx_oc_pwm_chy_handle.OCIdleState = TIM_OCIDLESTATE_RESET;
    g_atimx_oc_pwm_chy_handle.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    HAL_TIM_PWM_ConfigChannel(&g_atimx_pwm_chy_handle,
    &g_atimx_oc_pwm_chy_handle, ATIM_TIMX_PWM_CH1);     /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_pwm_chy_handle,
    &g_atimx_oc_pwm_chy_handle, ATIM_TIMX_PWM_CH2);     /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_pwm_chy_handle,
    &g_atimx_oc_pwm_chy_handle, ATIM_TIMX_PWM_CH3);     /* 配置 TIMx 通道 y */
   /* 开启对应 PWM 通道 */
    HAL_TIM_PWM_Start(&g_atimx_pwm_chy_handle, ATIM_TIMX_PWM_CH1);
    HAL_TIM_PWM_Start(&g_atimx_pwm_chy_handle, ATIM_TIMX_PWM_CH2);
    HAL_TIM_PWM_Start(&g_atimx_pwm_chy_handle, ATIM_TIMX_PWM_CH3);
```

```
}
```

HAL_TIM_PWM_Init 初始化 TIM8 并设置 TIM8 的 ARR 和 PSC 等参数，然后通过调用函数 HAL_TIM_PWM_ConfigChannel 设置 TIM8_CH1\2\3 的 PWM 模式以及比较值等参数，最后通过调用函数 HAL_TIM_PWM_Start 来使能 TIM8 以及使能 PWM 通道 TIM8_CH1\2\3 输出。通道的 GPIO 口初始化和定时器使能等程序放到回调函数 HAL_TIM_PWM_MspInit 中这里就不贴代码了，感兴趣打开源码看看。接着我们来看下 steering_engine.c，首先看角度转比较值函数。

```
/**
 * @brief       舵机角度转定时器比较值
 * @param       angle:角度
 * @retval      定时器装比较值
 */
uint16_t angle_to_tim_val(float angle)
{
    uint16_t ret;
    if((angle < 0)||(angle > 180))
    {
        ret = 0;
    }
    else
    {
        /* 0° -- 500, 45° -- 1000, 90° -- 1500, 135° -- 2000, 180° -- 2500 */
        ret = 1500 + (int)((float)((angle - 90) * 100 / 9));
        if((ret < 500)||(ret > 2500))
        {
            ret = 0;
        }
    }
    return ret;
}
```

这个函数主要功能就是把角度值转换成定时器的比较值，我们前面介绍讲过舵机的旋转角度和脉宽成线性关系，比较值决定脉宽时间，所以我们这里把要转动的角度转成比较值控制脉宽时间，从而达到控制舵机旋转角度。接着看下一个舵机角度的设置函数。

```
/**
 * @brief       舵机角度设定
 * @param       id:舵机编号对应舵机的接口：1~3; angle:角度
 * @retval      0:成功
 */
uint8_t servo_angle_set(uint8_t id,float angle)
{
    uint16_t val;
    switch(id)
    {
        case 1:
            val = angle_to_tim_val(angle);  /* 得到角度转换的比较值 */
            if(val != 0)
            {
                /* 设置比较值 */
                __HAL_TIM_SetCompare(&g_atimx_pwm_chy_handle,TIM_CHANNEL_1,val);
            }
            break;
        case 2:
            val = angle_to_tim_val(angle);
            if(val != 0)
            {
                __HAL_TIM_SetCompare(&g_atimx_pwm_chy_handle,TIM_CHANNEL_2,val);
            }
            break;
        case 3:
            val = angle_to_tim_val(angle);
            if(val != 0)
            {
```

```
            __HAL_TIM_SetCompare(&g_atimx_pwm_chy_handle,TIM_CHANNEL_3,val);
        }
        break;
    default:
        break;
  }
  return 0;
}
```

由于我们的开发板有三个舵机接口，所以我们写的这个函数有一个 id 判断，id:1、2、3，分别控制舵机接口 1、2、3，想要控制哪个接口的舵机就选择哪个 id 即可，然后第二个参数设置我们要舵机旋转的角度值，里边会调用到角度转比较值函数，然后我们将返回的数值，设置到定时器的比较值，即可完成脉宽时间的设置，控制舵机旋转对应角度。

在 main 函数里面编写如下代码：

```
int main(void)
{
    uint8_t key,t,id = 1;
    char buf[32];
    float angle[3] = {0};                   /* 初始化角度 0° */

    HAL_Init();                             /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);     /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    key_init();                             /* 初始化按键 */
    lcd_init();                             /* 初始化 LCD */
    /* 168 000 000 / 168 = 1000 000 1Mhz 的计数频率， 1us 计数一次，20K 次为 20ms */
    atim_timx_pwm_chy_init(20000 - 1, 168 - 1);

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"Servo Test",g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:ID + +",g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:IAngle +   ",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:Angle -",g_point_color);
    lcd_show_string(10,90,200,16,16,"Servo ID: 1",g_point_color);

    while (1)
    {
        if(t % 10 == 1)
        {
            sprintf(buf,"Servo 1: %.1f",angle[0]);
            lcd_show_string(10,110,200,16,16,buf,g_point_color);
            sprintf(buf,"Servo 2: %.1f",angle[1]);
            lcd_show_string(10,130,200,16,16,buf,g_point_color);
            sprintf(buf,"Servo 3: %.1f",angle[2]);
            lcd_show_string(10,150,200,16,16,buf,g_point_color);
        }

        key = key_scan(0);
        if(key == KEY0_PRES)
        {
            id++;
            if(id == 4)
            {
                id = 1;
            }
            /* 按下 KEY0：选择控制哪个舵机，并显示当前 ID */
            sprintf(buf,"Servo ID: %1d",id);
            lcd_show_string(10,90,200,16,16,buf,g_point_color);
        }
        if(key == KEY1_PRES)
        {
```

```
            angle[id-1] += 45;
            if(angle[id-1] > 180)
            {
                angle[id-1] = 180;
            }
            servo_angle_set(id,angle[id-1]);      /* 控制该 ID 的舵机，并设置角度值 */
        }
        if(key == KEY2_PRES)
        {
            angle[id-1] -= 45;
            if(angle[id-1] < 0)
            {
                angle[id-1] = 0;
            }
            servo_angle_set(id,angle[id-1]);      /* 控制该 ID 的舵机，并设置角度值 */
        }

        t++;
        if(t % 20 == 0)
        {
            LED0_TOGGLE();                        /* LED0(红灯) 翻转*/
        }
        delay_ms(10);
    }
}
```

先看 atim_timx_pwm_chy_init(20000 - 1, 168 - 1)这个语句，这两个形参分别设置自动重载寄存器的值为 20000，以及定时器预分频系数为 168。先看预分频系数，我们设置为 168 分频，然后定时器 8 的时钟是 2 倍的 APB2，即 168MHZ，可以得到计数器的计数频率是 1MHZ，也就是计数一次 1us，我们计数 20000 次就是 20ms，符合我们控制舵机的脉冲周期要求，通过按键设置比较值控制脉宽时间；然后 LCD 显示一些提示信息包括当前 id 以及当前控制舵机的角度值；注意需提前将舵机接入对应的舵机接口，按键 KEY0 设置控制哪个接口的舵机，KEY1 增加舵机旋转角度，最大到 180°；KEY2 减小舵机旋转角度，最小到 0°。LED0 周期闪烁提示程序正在运行。

14.5.3 下载验证

下载代码后，可以看到 LED0 每隔 200ms 左右闪烁一次，LCD 显示当前所控制的舵机接口号以及舵机旋转角度，当我们按下 KEY0 选择控制哪个舵机接口，按下 KEY1 将舵机旋转角度每次加 45°，最大到 180°；按下 KEY2 将舵机旋转角度每次减 45°，最小到 0°。同时我们也可以不接舵机，使用示波器查看输出波形的脉宽时间。

下面我们使用正点原子 DS100 手持数字示波器，把舵机接口一的控制信号引脚连接至示波器通道，看波形截获，具体如下：

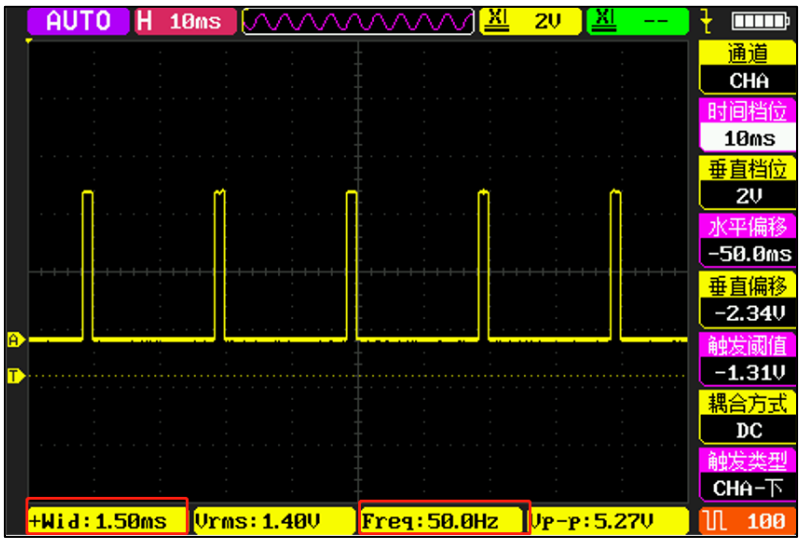


图 14.5.3.1 舵机接口一旋转 90°的波形图

可以看到当控制舵机旋转 90°时，脉冲频率 50HZ（20ms），脉宽 1.5ms 和我们前面介绍的完全一致，大家可以自己尝试测量看看。

第 15 章 步进电机

本章，我们主要来了解下什么是步进电机以及如何使用 STM32 来控制步进电机。希望大家通过本章的学习，对步进电机有个深入的认识，并掌握步进电机的基本驱动原理以及实现步进电机的定位控制。

本章分为如下几个小节：

15.1 步进电机的简介
15.2 步进电机的分类
15.3 步进电机的工作原理
15.4 步进电机的指标参数
15.5 步进电机的驱动器介绍
15.6 步进电机旋转控制实现
15.7 步进电机定位控制实现

15.1 步进电机的简介

步进电机是一种将电脉冲信号转换成相应角位移或线位移的电动机。每输入一个脉冲信号，转子就转动一个角度或前进一步，其输出的角位移或线位移与输入的脉冲数成正比，转速与脉冲频率成正比。因此，步进电动机又称脉冲电动机。**在非超载且不超频的情况下，电机的旋转位置只取决于脉冲个数，转速只取决脉冲信号的频率。**所以说步进电机它的开环能力非常优秀，目前也被广泛的应用于各种开环系统上。

图 15.1.1 各种各样的步进电机

步进电机通过控制脉冲频率可以控制电机转动的速度和加速度，从而达到调速的目的。控制方便，但想玩转也并非易事，在本章内容中主要涉及的是步进电机的驱动控制，其中包括：基础的旋转和换向控制，定位控制等。

15.2 步进电机的分类

步进电机的分类方式有很多种，我们这里不一一列举，一般我们可从其磁激励的方式进

行区分，将步进电机区分为三大类：**永磁式、反应式（磁阻式）、混合式步进电机**。他们之间的区别如下：

1.反应式：定子上有绕组、转子由软磁材料组成。结构简单、成本低、步距角小，可达 1.2°、但噪声和震动大，动态性能较低。

2.永磁式：永磁式步进电机的转子用永磁材料制成，转子的极数与定子的极数相同。其特点是动态性能好、输出力矩大，但这种电机步矩角大（一般为 7.5°或 15°）。精度也就一般。

3.混合式：混合式步进电机综合了反应式和永磁式的优点，其定子上有多相绕组、转子上采用永磁材料，转子和定子上均有多个小齿以提高步矩精度。其特点是输出力矩大、动态性能好，步距角小，但结构复杂、成本也相对较高，但这类步进电机的应用最为广泛，我们后面的教程也是主要以混合式的步进电机来讲解。

总结如下看表 15.2.1：

类别	优点	缺点
永磁式	动态性能好，输出力矩大	步距角过大(一般 7.5°或 15°)，精度较差
磁阻式	结构简单，成本低，步距角小可达 1.2°	噪音和震动过大，动态性能较低
混合式	力矩大，动态性能好，步距角小，精度高	结构复杂，成本相对较高

表 15.2.1 各类步进电机的优缺点

除了以磁激励的方式进行区分之外，还可以以极性的方式作为区分，可分为：单极性步进电机、双极性步进电机，如下图:

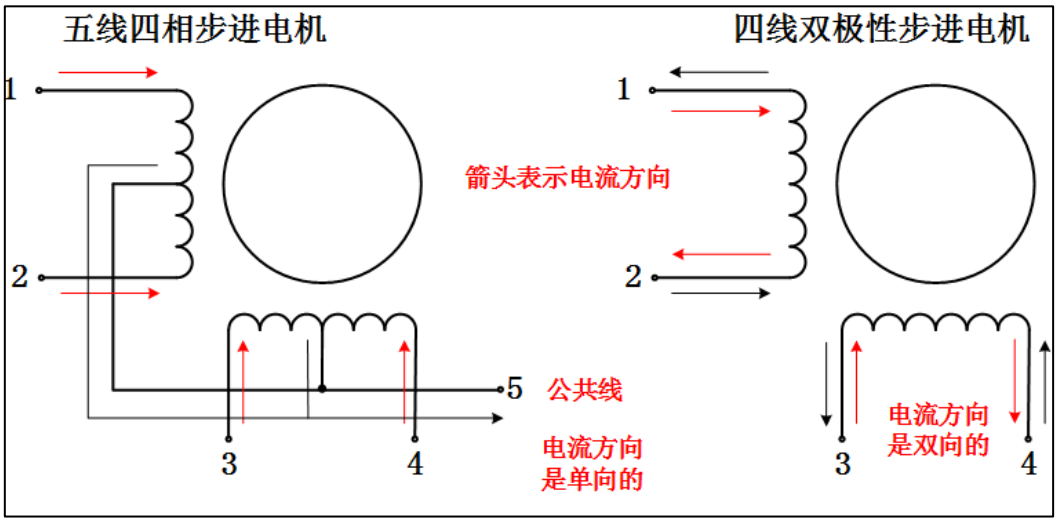


图 15.2.1 单极性与双极性简易示意图

图 15.2.1 中左侧为单机性步进电机，右侧为双极性的步进电机，它们的区别就是电流的流向，单极性步进电机，特点是有一个公共端，电流的流向固定且只有一个方向；双极性步进电机的特点：电流流向是双向的，电机中有两个电流回路。单极性步进电机较双极性步进电机驱动起来较为简单，但是单极性的输出力矩较小。

15.3 步进电机的工作原理

步进电机的核心部件主要有：定子、转子、定子上的绕组。我们来看下结构示意图：

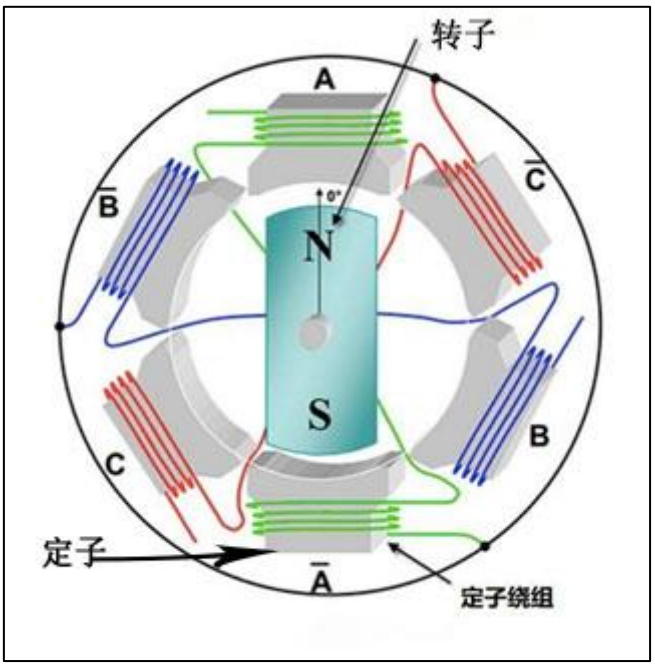


图 15.3.1 步进电机核心结构图

步进电机的工作原理就是控制定子绕组周期性、交替得电，使定子产生磁场，进而控制步进电机转子一步一步的运动起来。下面我们开始来分析单极性以及双极性步进电机它们的驱动原理，了解其究竟是如何旋转起来的。

15.3.1 单极性步进电机驱动原理

首先我们先来看下单极性步进电机驱动原理的示意图，如下图 15.3.1.1：

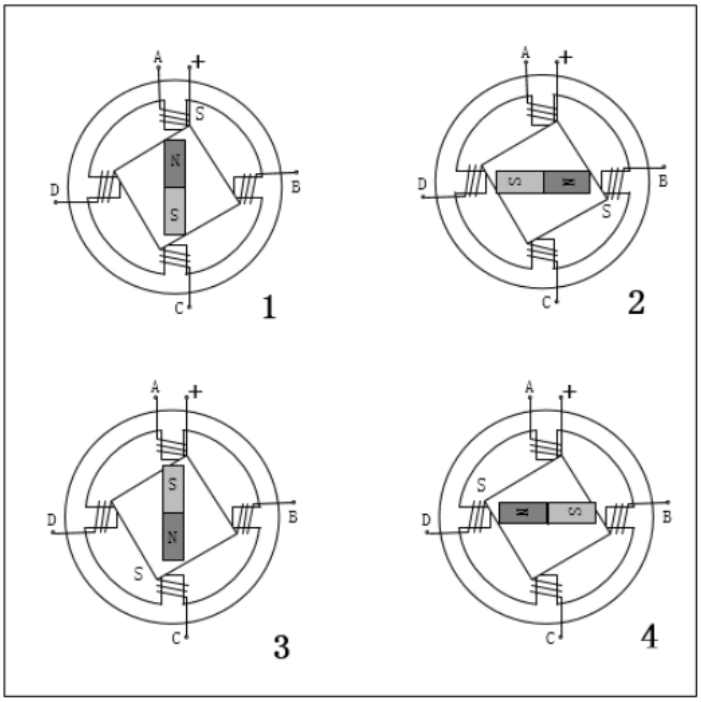


图 15.3.1.1 单极性步进电机整步驱动

我们把图 15.3.1.1 的驱动方式称为：单极性整步驱动，图示中有 5 根线，分别是 A、B、C、D 以及公共端，公共端是一直通电的，ABCD 相是交替得电的。我们来看下它的通电顺序步骤，第一步：A 相通电，通过电磁原理，电生磁，然后通过磁力把转子吸引到 A 处；第二步：A 相关闭，B 相通电，转子就会旋转 90° 到 B 处；第三步 B 相关闭，C 相通电，

转子会再次旋转 90°，到 C 处，接着第四步：C 相关闭，D 相通电，就会旋转到 D 处，重复该顺序就可以以 90°的步距角顺时针旋转起来。这种属于单相激励步进，意思就是每步进一步都只受到了一个磁场的力，但是更为常见的是双相激励的方式，因为力矩相对会更大，具体看下图：

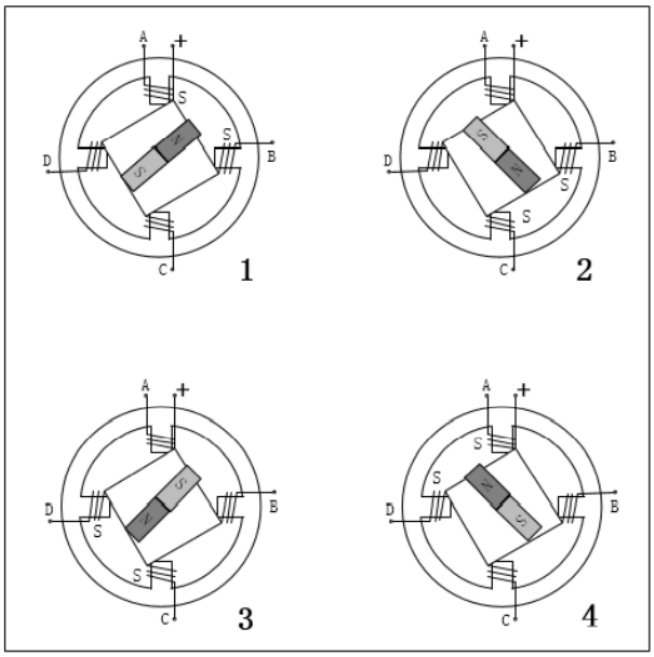


图 15.3.1.2 单极性整步驱动双相激励方式

图 15.3.1.2 为单极性的双相激励方式驱动。它的工作方式是同时两相一起通电，它与上一种驱动方式的区别是固定在两相中间位置，因为两相同时通入相同的电流此时磁力矢量在两相中间就会把转子吸引到两相的中间位置，还有一个区别就是这种方式它是受到了两个磁场的力所以他的输出力矩相对上一种会更大。接着我们看下它的通电顺序：AB—BC—CD—DA，重复该顺序就可以使电机顺时针旋转起来。像这种方式，相邻的线圈的角度为 90°，然后转子直接一步到位旋转 90°的方式称为整步驱动。还有一种半步驱动我们来看下是如何实现的，具体看下图：

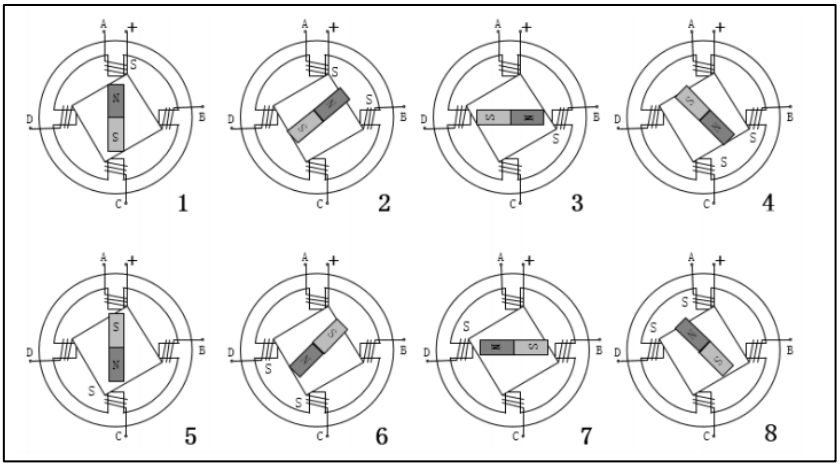


图 15.3.1.3 单极性半步驱动方式

图 15.3.1.3 就为半步驱动的方式了，它每步步进的距离为 45°，我们来看下它的通电顺序：A—AB—B—BC—C—CD—D—DA，重复该步骤，电机就可以以 45°的步距角顺时针旋转了，同理想要逆时针旋转的话就将通电顺序反过来即可。

15.3.2 双极性步进电机驱动原理

双极性与单极性类似，都可分为整步与半步驱动方式，不同的是，单极性的电流方向是单

向的，而双极性的步进电机可以通过改变电流的方向来改变每相磁场的方向，详细见下图：

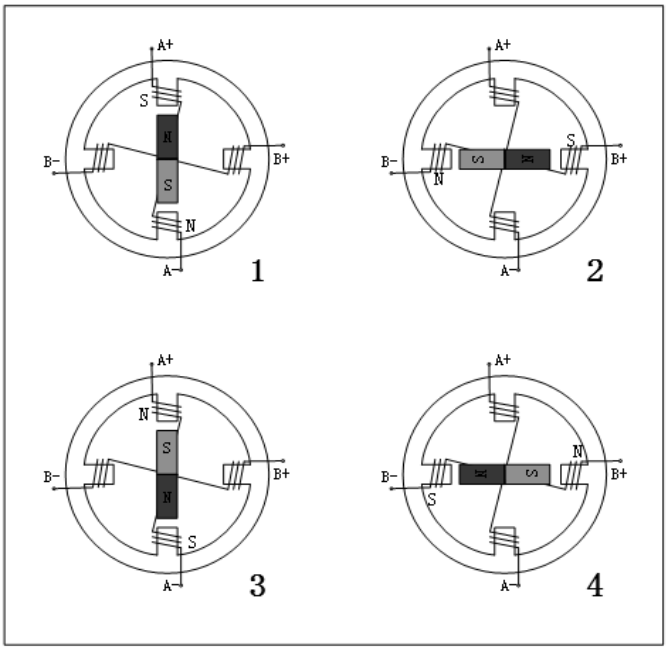


图 15.3.2.1 双极性步进电机驱动原理图

图 15.3.2.1 为双极性的整步驱动，它没有公共端，所以他的电流可以从 A+进去也可以从 A-进去，B 相同理。我们看下它的通电顺序第一步：A 相通电，此时 A+方向接电源正极，A-接电源负极，那么此时根据电磁原理，产生磁场，异性相吸，就会将转子吸引到第一步的位置；接着第二步：A 相关闭，B 相通电（B+接正，B-接负），就会吸引转子旋转 90°；第三步：B 相关闭，A 相通电但是极性与第一步相反（A+接负，A-接正），此时就会吸引转子继续旋转 90°；第四步：A 相关闭，B 相通电，极性与第 2 步相反，重复该步骤就可以使其以 90°的步距角顺时针旋转了。

同样的上面的驱动方式属于单相激励，双极性也有双相激励的我们来看下:

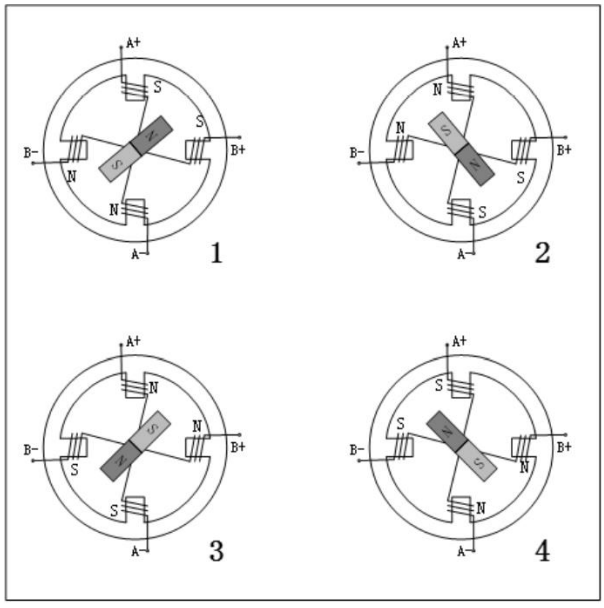


图 15.3.2.2 双极性双相激励的驱动方式

图 15.3.2.2 为双极性的双相激励驱动方式，力矩相较于单相会更大，它的通电顺序为：AB（A+B+接正）—AB（A-B+接正）—AB（A-B-接正）—AB（A+B-接正）重复该顺序即可顺时针旋转起来。前面介绍的两个方式都是整步驱动接着看下半步驱动的方式：

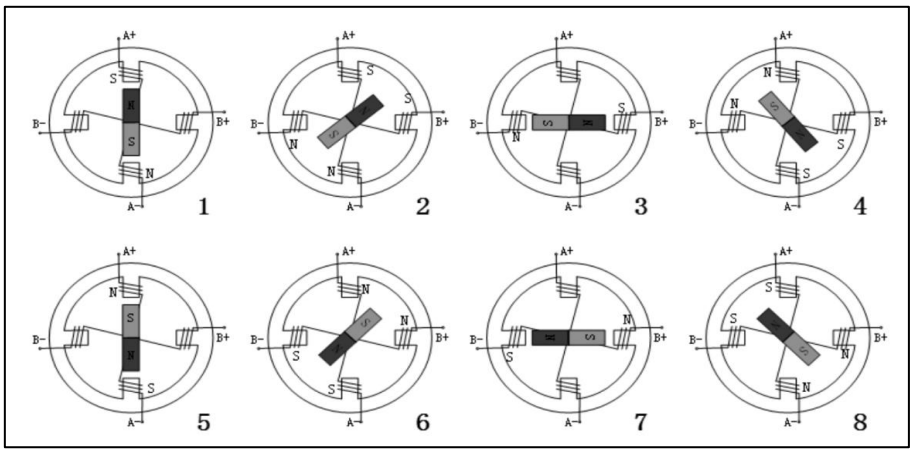


图 15.3.2.3 双极性半步驱动

我们直接看下它的通电顺序：

第一步：A 相通电（A+接正），B 相不通电；

第二步：AB 相通电（A+B+接正），通入相同电流，产生磁力相同；

第三步：B 相通电（B+接正），A 相不通电

第四步：AB 相通电（A-B+接正），通入相同电流，产生磁力相同；

第五步：A 相通电（A-接正），B 相不通电；

第六步：AB 相通电（A-B-接正），通入相同电流，产生磁力相同；

第七步：B 相通电（B-接正），A 相不通电

第八步：AB 相通电（A+B-接正）通入相同电流，产生磁力相同；

重复上面步骤就可以以 45° 的步距角顺时针旋转，与整步相比半步的驱动方式旋转起来更加的顺滑。

但是在实际的应用中，步距角 45° 还是过于大，在很多高精度的场合是不合适的，那我们有没有什么方法可以减小步距角呢？当然有的，大致可以分为两种：

硬件改进：通过增加线圈的组数或者增加转子的齿数，进而减小步进的角度。不过线圈越多需要的成本就越大，并且电机的大小是有限的，所以线圈的数量是有限制的，很显然这不是一个好方法；如果是增加转子的齿数，它的缺点也很显著，转子的齿数越多对加工工艺的要求就越大，而且硬件上的改进都有一个物理极限，如果想要一个很小的步距角很显然采用硬件改进不是一个明智的方法。

软件改进：使用细分驱动的方式，减小步距角，何谓细分驱动？我们来看下细分驱动的原理。

15.3.3 细分驱动原理

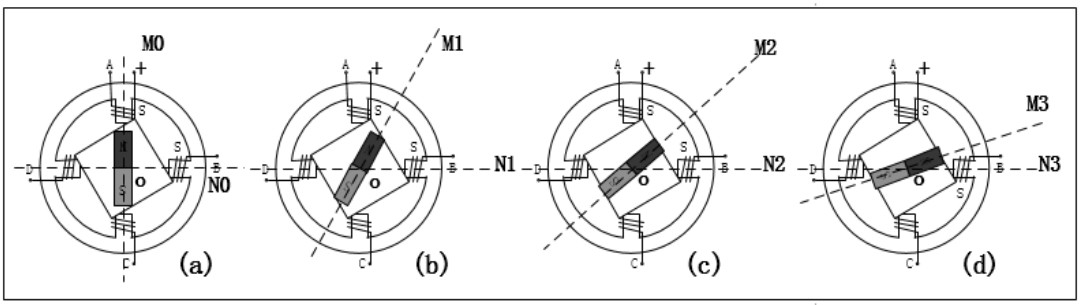


图 15.3.3.1 细分驱动原理示意图

细分驱动，单双极步进电机都同理，所以不分单双极电机，图 15.3.3.1 以单极为例。

在上图中均为双相激励；其中图（a）为 A 相电流很大，B 相的电流极其微弱，接近 0；图 (c)为 A 相和 B 相的电流相同，电流决定磁场，所以说 A 相和 B 相的磁场也是相同的。(a) 和（c）可以是极限特殊的情况，再看图（b）和图（d）这两个是由于 A 相和 B 相的电

流不同产生位置情况；由此可以得出改变定子的电流比例就可以使得转子在任意角度停住。所以细分的原理就是：**通过改变定子的电流比例，进而控制转子在一个整步中的不同位置，即可以将一个整步分成多个小步来运行**。在上图中就是一个整步分成了 4 步来跑，从（a）~（d）是 A 相的电流逐渐减小，B 相电流逐渐增大的过程，如果驱动器的细分能力很强，可以将其分成 32 细分、64 细分等；这不仅提高了步进电机旋转的顺畅度而且提高了每步的精度。

15.4 步进电机的指标参数

15.4.1 静态指标术语

（1）相数：产生不同对极 N、S 磁场的激磁线圈对数，也可以理解为步进电机中线圈的组数，其中两相步进电机步距角为 1.8°，三相的步进电机步距角为 1.5°，相数越多的步进电机，其步距角就越小。

（2）拍数：完成一个磁场周期性变化所需脉冲数或导电状态用 n 表示，或指电机转过一个齿距角所需脉冲数，以四相电机为例，有四相四拍运行方式即 AB-BC-CD-DA-AB，四相八拍运行方式即 A-AB-B-BC-C-CD-D-DA-A。

（3）步距角：一个脉冲信号所对应的电机转动的角度，可以简单理解为一个脉冲信号驱动的角度，电机上都有写，一般 42 两相混合式步进电机的步距角为 1.8°。

（4）定位转矩：电机在不通电状态下，电机转子自身的锁定力矩（由磁场齿形的谐波以及机械误差造成的）。

（5）静转矩：电机在额定静态电压作用下，电机不作旋转运动时，电机转轴的锁定力矩。此力矩是衡量电机体积的标准，与驱动电压及驱动电源等无关。

15.4.2 动态指标术语

（1）步距角精度：步进电机转动一个步距角度的理论值与实际值的误差。用百分比表示：误差/步距角 *100%。

（2）失步：电机运转时运转的步数，不等于理论上的步数。也可以叫做丢步，一般都是因负载太大或者是频率过快。

（3）失调角：转子齿轴线偏移定子齿轴线的角度，电机运转必存在失调角，由失调角产生的误差，采用细分驱动是不能解决的。

（4）最大空载起动频率：在不加负载的情况下，能够直接起动的最大频率。

（5）最大空载的运行频率：电机不带负载的最高转速频率。

（6）运行转矩特性：电机的动态力矩取决于电机运行时的平均电流（而非静态电流），平均电流越大，电机输出力矩越大，即电机的频率特性越硬。

（7）电机正反转控制：通过改变通电顺序而改变电机的正反转。

15.5 步进电机驱动器介绍

为了便于控制步进电机，将 MOS 管、控制器、电源、保护电路、散热器等等集成在一起，就做成了步进电机驱动器，驱动器的作用就是将控制器的信号放大或者进行转换。如下图所示，我们的电机开发板输出脉冲信号给步进电机驱动器，驱动器将其功率放大控制步进电机，有些驱动器支持细分功能可以调节电流大小达到细分驱动，使其步距角更小，精度更高，旋转起来更加的顺滑。

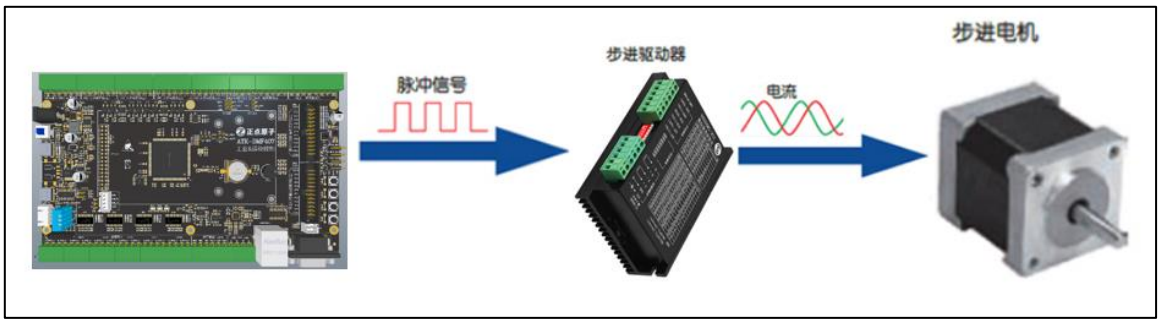


图 15.5.1 步进电机控制流程图

15.5.1 正点原子步进电机细分驱动器介绍

正点原子推出的一款高性能细分型 2 相混合式步进电机驱动器——ATK-2MD5050，采用 12~50VDC 供电，输出峰值电流可达 5.0A，适合驱动常用的 42mm、57mm、86mm 两相混合式步进电机。此驱动器采用了行业领先的功能集，包括高分辨率微步进，无感机械负载测量，负载自适应功率优化和低谐振斩波器等技术，这些功能使得电机在应用中实现更高额精度，更高的能源效率，更高的可靠性，更平稳的运行，广泛适用于 3D 打印机、雕刻机、数控机床、包装机械等分辨率要求较高的设备上。该驱动器特点包括：

1， 采用先进高精度斩波算法，降低电机运行噪音和振动。
2， 超高分辨率微步进，最大支持 256 细分，提高微步进平滑度。
3， 负载自适应功率优化，节能同时降低电机发热。
4， 支持自动半流功能，减小电机发热。
5， 灵活的输出电流设定，适用不同型号的电机。
6， 直流 12~50V 供电，输出峰值电流高达 5.0A。
7， 支持欠压保护、短路保护、过热保护等功能。
8， 控制信号全部光耦隔离，抗干扰能力强。

步进电机驱动器实物如下图 15.5.1.1 所示：

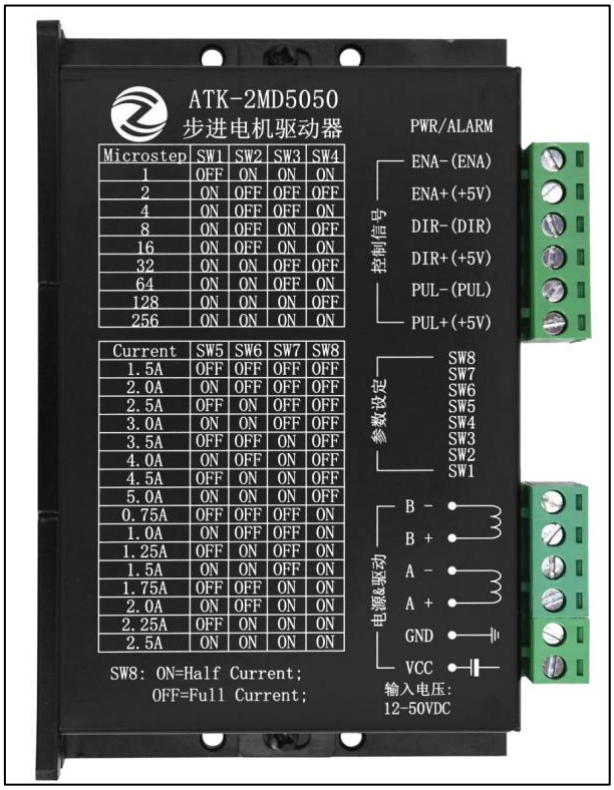


图 15.5.1.1 正点原子步进电机驱动器实物图

接着看下该步进电机驱动器的典型参数表：

项目	说明
输入电压	直流 12~50V 输入
输出电流	全流（SW8=OFF）：1.5~5.0A，8 档可调，分辨率 0.5A 半流（SW8=ON）：0.75~2.5A，8 档可调，分辨率 0.25A
驱动方式	双极恒流 PWM 驱动输出
温度	工作温度：-10℃~60℃；存储温度-40℃~80℃
冷却方式	自然冷却
使用场合	避免粉尘、油雾以及腐蚀性气体;避开过热潮湿环境
安装尺寸	118mm*76mm*33mm（长*宽*高）
重量	250 克

表 15.5.1 ATK-2MD5050 典型参数

15.5.2 驱动器控制接口

ATK-2MD5050 驱动器留有红绿 2 个指示灯（上绿下红），绿灯（PWR）为驱动器电源指示灯，驱动器供电正常，绿灯亮起；红灯（ALARM）为错误指示灯，当驱动器发生短路时，红灯快闪，驱动器过热时，红灯间隔 1S 快闪 3 次，当出现这些情况，请断开驱动器电源，排除故障后再操作。

ATK-2MD5050 驱动器控制信号接口描述如表 15.5.2.1 所示：

接口名称	引脚定义	功能说明
ENA-	输出使能负端	驱动器的输出通过该组合信号使能，通常说的脱机信号。此信号有效，输出关闭，电机线圈电流为零，电机处于无力矩状态，可自由转动。
ENA+(+5V)	输出使能正端	
DIR-	方向控制负端	电机转动方向控制信号，若信号有效，电机顺时针转动，若信号无效，电机则逆时针转动
DIR+(+5V)	方向控制正端	
PUL-	脉冲信号负端	电机转动脉冲信号，驱动器接收到信号后，按照设定的方向转动，转动角度和脉冲数量成正比，转速和脉冲频率成正比，输入的频频率<=200KHz
PUL+	脉冲信号正端	

表 15.5.2.1 驱动器控制信号描述

ATK-2MD5050 驱动器的控制信号可以高电平有效，也可以低电平有效。把所有控制信号的负端连接一起到地时，高电平有效；当把所有控制信号的正端连接一起作为信号公共端时，低电平有效。

15.5.2.1 驱动器细分参数设置

注意：拨码开关按下时为 ON，抬起时为 OFF，下同 。

用户可以通过调整驱动器面板上拨码开关（SW1 SW2 SW3 SW4）的状态来设定 9 种细分模式。一般 2 相 4 线步进电机的步距角都是 1.8°，所以在没有细分的情况下，转动一圈需 要脉冲数为 360/1.8°=200，2 细分转动一圈需要的脉冲数则为 200*2=400。细分设置见表 15.5.2.1.1：

细分微步	脉冲数/圈	SW1	SW2	SW3	SW4
1	200	OFF	ON	ON	ON
2	400	ON	OFF	OFF	OFF
4	800	ON	OFF	OFF	ON
8	1600	ON	OFF	ON	OFF
16	3200	ON	OFF	ON	ON
32	6400	ON	ON	OFF	OFF
64	12800	ON	ON	OFF	ON
128	25600	ON	ON	ON	OFF
256	51200	ON	ON	ON	ON

表 15.5.2.1.1 驱动器细分设置表

15.5.2.2 驱动器电流参数设置

本驱动器设计 8x2 档输出电流选择，由驱动器上的拨码开关（SW5 SW6 SW7 SW8）设定，SW8 用于设置半流：当 SW8 设置为 ON，驱动器输出电流为 SW5、SW6、SW7 所设置电流的一半，当 SW8 设置为 OFF，输出电流则为 SW5、SW6、SW7 所设置的电流，电流大小以最大值(Peak)标称。该驱动器最大可提供 5.0A 的输出电流，电流设定分率为 0.5A（SW8=OFF）或者 0.25A（SW8=ON）。拨码开关（SW5 SW6 SW7 SW8）组合的每一种状态代表一个电流值输出设定，参见表 15.5.2.1.2：

Peak	SW5	SW6	SW7	SW8
1.5A	OFF	OFF	OFF	OFF
2.0A	ON	OFF	OFF	OFF
2.5A	OFF	ON	OFF	OFF
3.0A	ON	ON	OFF	OFF
3.5A	OFF	OFF	ON	OFF
4.0A	ON	OFF	ON	OFF
4.5A	OFF	ON	ON	OFF
5.0A	ON	ON	ON	OFF
0.75A	OFF	OFF	OFF	ON
1.0A	ON	OFF	OFF	ON
1.25A	OFF	ON	OFF	ON
1.5A	ON	ON	OFF	ON
1.75A	OFF	OFF	ON	ON
2.0A	ON	OFF	ON	ON
2.25A	OFF	ON	ON	ON
2.5A	ON	ON	ON	ON

表 15.5.2.1.2 驱动器电流设置表

注意：严禁在驱动器驱动电机转动过程中更改拨码开关!!!

15.5.2.3 接线方式

1、共阳极接法如图所示：

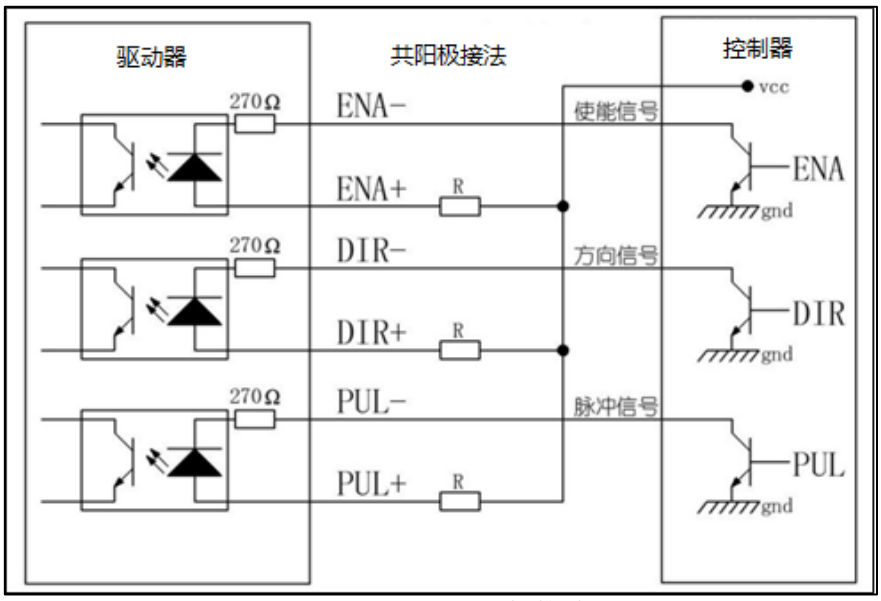


图 15.5.2.3.1 共阳极接法

2、共阴极接法如图所示：

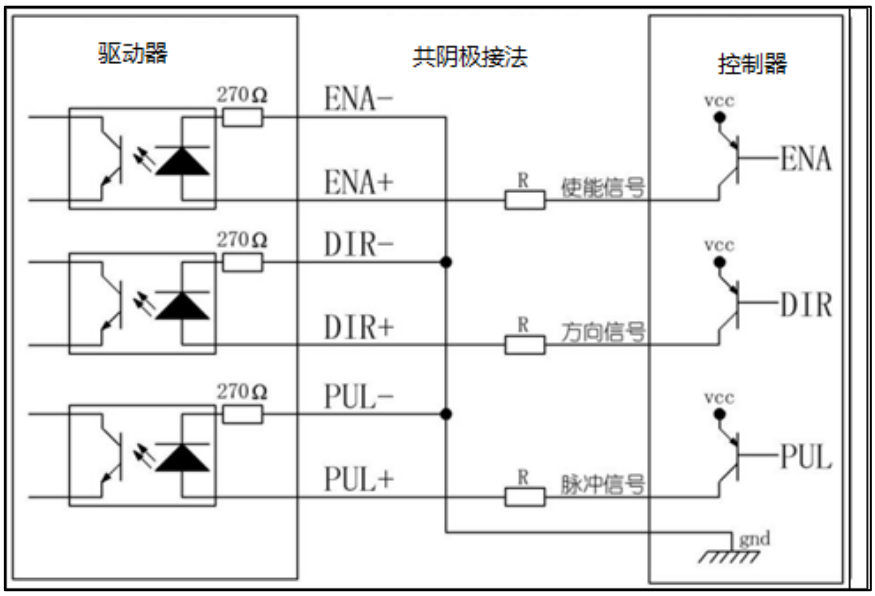


图 15.5.2.3.2 共阴极接法

实验中所使用到的是共阳极接法整体接线如下图所示：

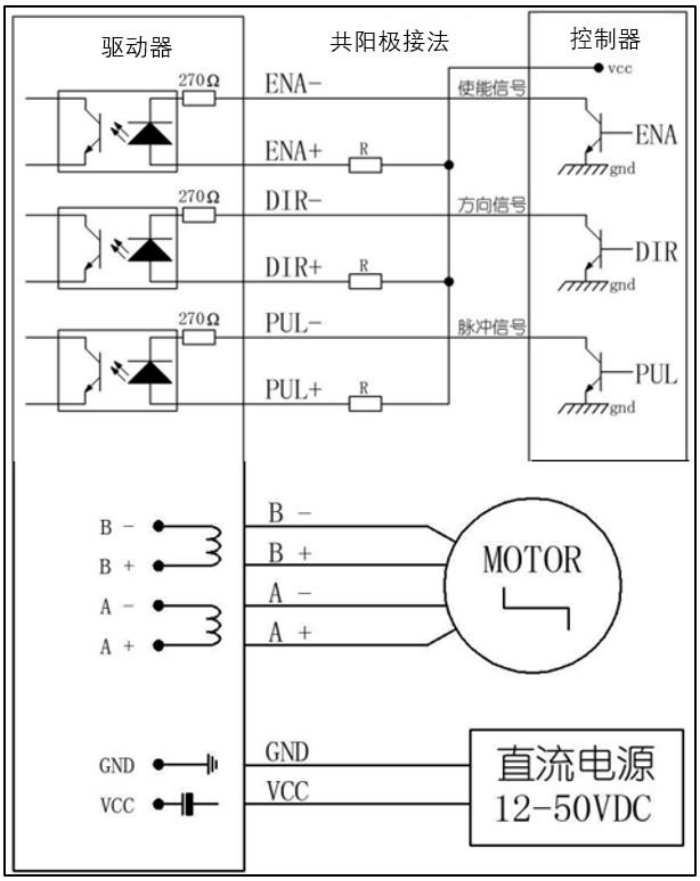


图 15.5.2.3.3 共阳极整体接线图

注意：接线顺序，请先接控制信号线，其次连接步进电机动力线，最后接驱动器电源线！

注意：步进电机专题下的实验如未特殊说明，则步进电机驱动器拨码开关均设置为：**8 细分、1.0A 电流。**

15.5.3 步进电机驱动器实物接线

下面我们将手把手地教大家搭建步进电机驱动的硬件平台，这一部分内容十分重要，大家一定要掌握。首先需要准备一些材料，如图 15.5.3.1 所示：

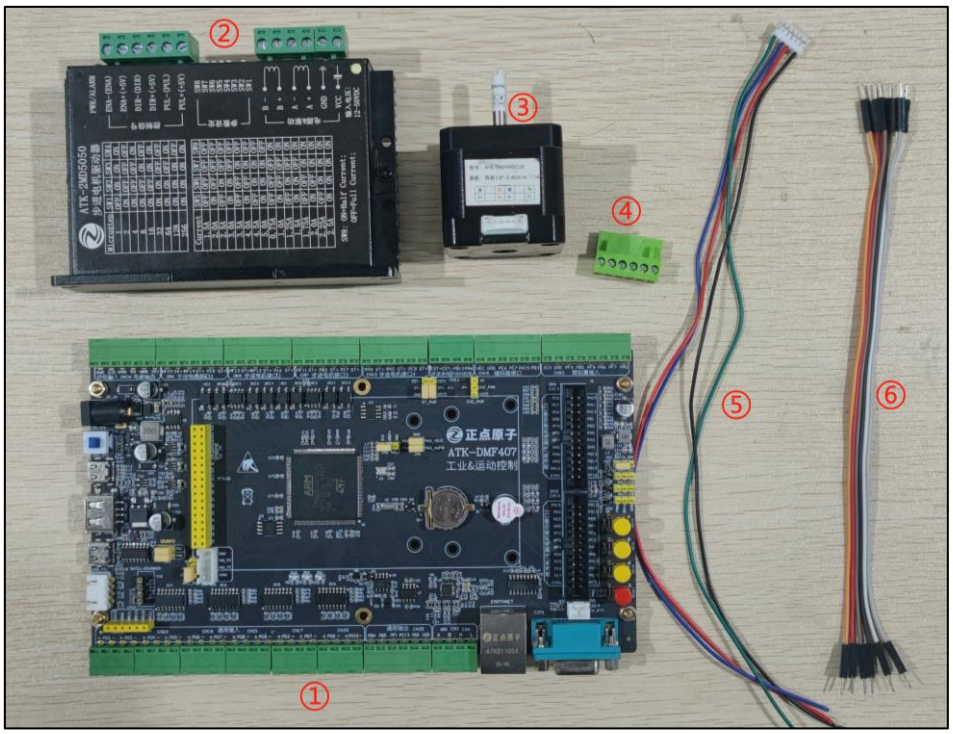


图 15.5.3.1 硬件平台搭建所需材料

图 15.5.3.1 中，不同的序号对应的材料如下：

① DMF407 电机开发板；
② 步进电机驱动器；
③ 步进电机；
④ 接口 6pin 端子；
⑤ 电机配套的 4p 排线；
⑥ 六根公头杜邦线；

除了之外，我们还需要准备一个 24V 的直流电源，给步进电机驱动器供电。

有了这些材料之后，我们还要了解下电机的引脚定义，实验用的 42 步进电机实物如下：

图 15.5.3.2 无刷电机实物

图 15.5.3.2 引脚定义从左往右数，1 脚为 A+，3 脚为 A-，4 脚为 B-，6 脚为 B+，2 和 5 为空脚。

了解接口的线序之后，就可以开始连接驱动器和步进电机了。我们只需要根据驱动器的丝印和电机的接口线序说明来连接它们即可，如图 15.5.3.3 所示：

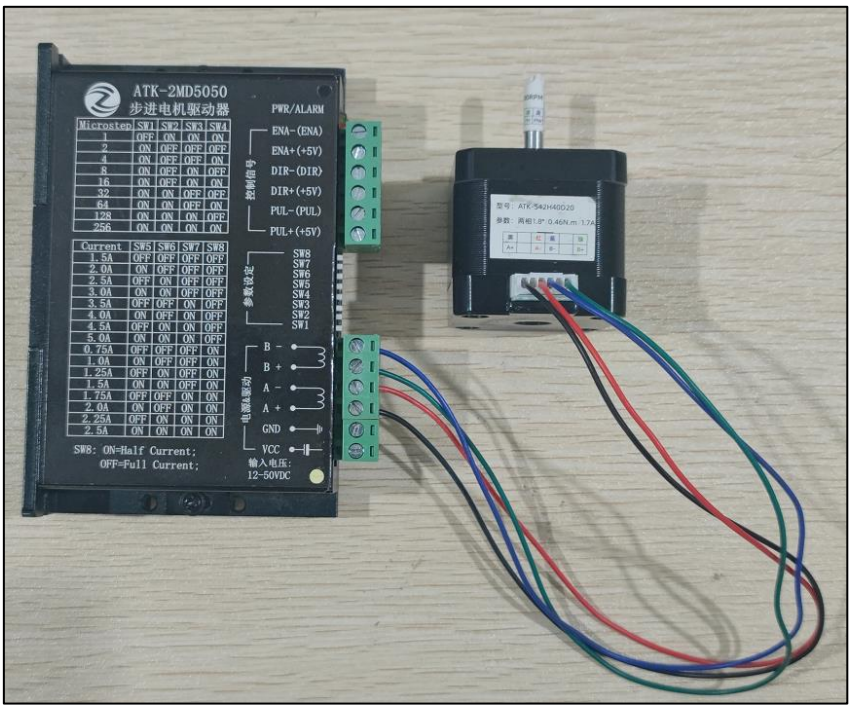


图 15.5.3.3 连接驱动器和电机

接好驱动板和电机之后，接着将驱动器与 DMF407 电机开发板进行连接（使用共阳极接法），这里以开发板上的步进电机接口一为例，连接关系如下表 15.5.3.1：

开发板丝印 （步进电机接口一为例）	驱动器丝印
PF15	ENA-
ST+	ENA+
PF14	DIR-
ST+	DIR+
PI5	PUL-
ST+	PUL+

表 15.5.3.1 步进电机与驱动器的连接关系

接完电机与驱动器以及驱动器与开发板之后，最后在接上驱动器电源即可大功告成，整体实物连接如下图 15.5.3.4 所示：

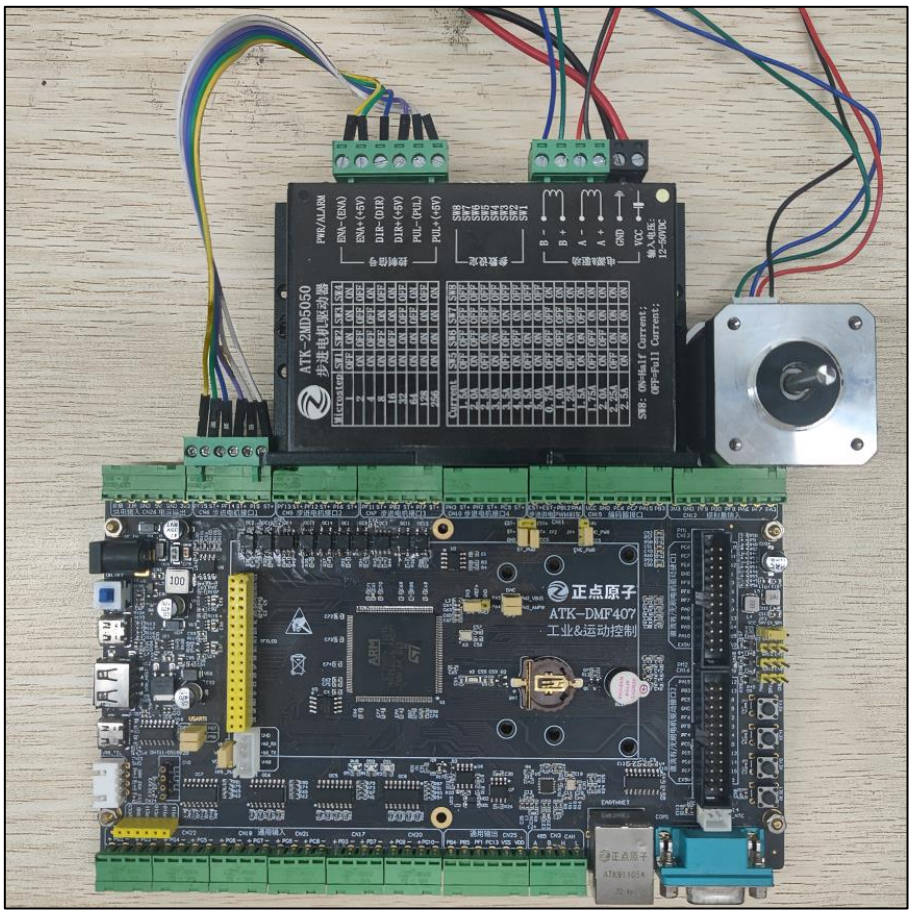


图 15.5.3.4 整体连接示意图

注意：接线顺序，请先接控制信号线以及连接步进电机动力线，最后接驱动器电源线！

至此，步进电机的整体实物连接就介绍完了，有了这些基础之后，我们就可以开始学习步进电机的基础驱动了。

15.6 步进电机旋转控制实现

前面已经介绍了步进电机的驱动原理以及步进电机驱动器功能特点等，接着我们开始来实现使用 STM32 控制步进电机的基础旋转。

由于接入了驱动器，控制步进电机就无需关注相的转换细节了，比如驱动电流、细分、过流保护等等全都交给驱动器控制了，STM32 只需要给驱动器电信号即可，一般一个完整的脉冲驱使步进电机旋转一步，如果不使用细分的话，对于 1.8° 步距角的步进电机，一次脉冲即可旋转 1.8° 。那我们本小节需要让它不断旋转起来就给他源源不断的脉冲信号即可。需要注意：一个驱动器只能驱动一个步进电机。

15.6.1 硬件设计

1、例程功能

我们的电机开发板全部有 4 路步进电机接口，分别由 TIM8_CH1、TIM8_CH2、TIM8_CH3 TIM8_CH4 控制，本实验需要将步进电机接入对应的步进电机接口，按键 KEY0 设置控制哪个接口的步进电机，KEY1 开启电机旋转°，KEY2 关闭电机旋转°，LED0 作为程序运行状态指示灯，闪烁周期 200ms。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器 8： TIM8_CH1：PI5，对应步进电机接口一
TIM8_CH2：PI6，对应步进电机接口二
TIM8_CH3：PI7，对应步进电机接口三
TIM8_CH4：PC9，对应步进电机接口四
方向引脚： DIR1: PF14
DIR2: PF12
DIR3: PB2
DIR4: PH2
脱机引脚： EN1: PF15
EN2: PF13
EN3: PF11
EN4: PH3
4）步进电机
5）步进电机驱动器
6）12-50V 的 DC 电源

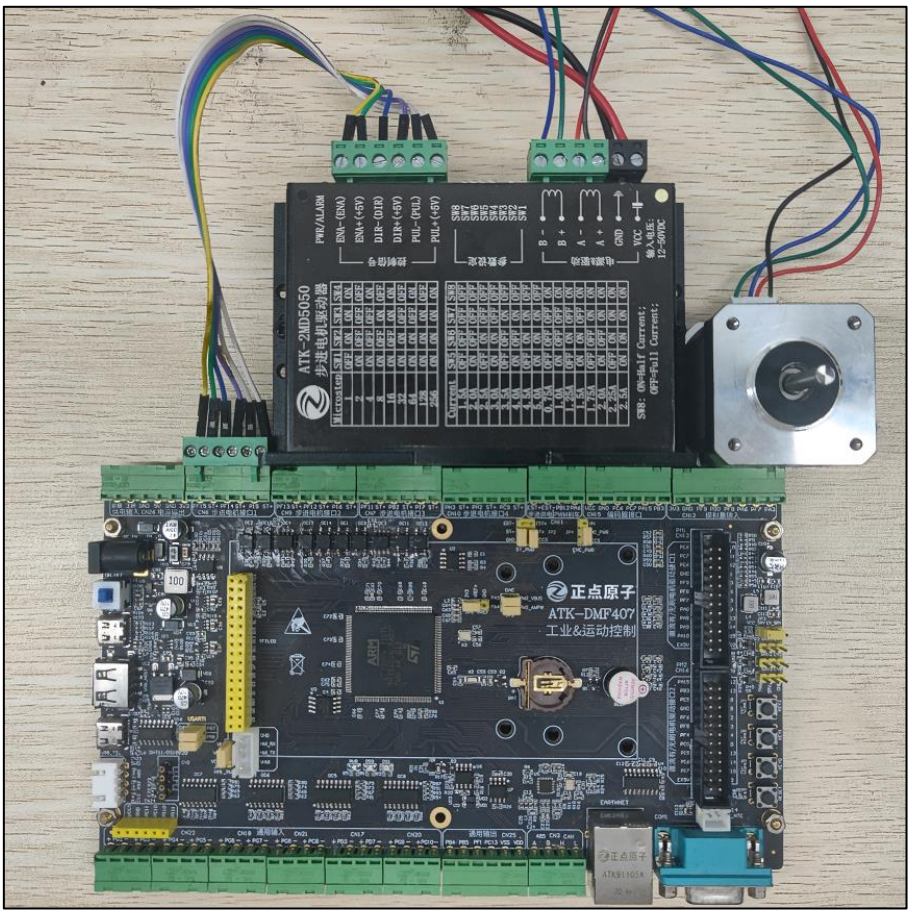


图 15.6.1.1 硬件准备（本图只用到一个电机，接至步进电机接口一）

3、原理图

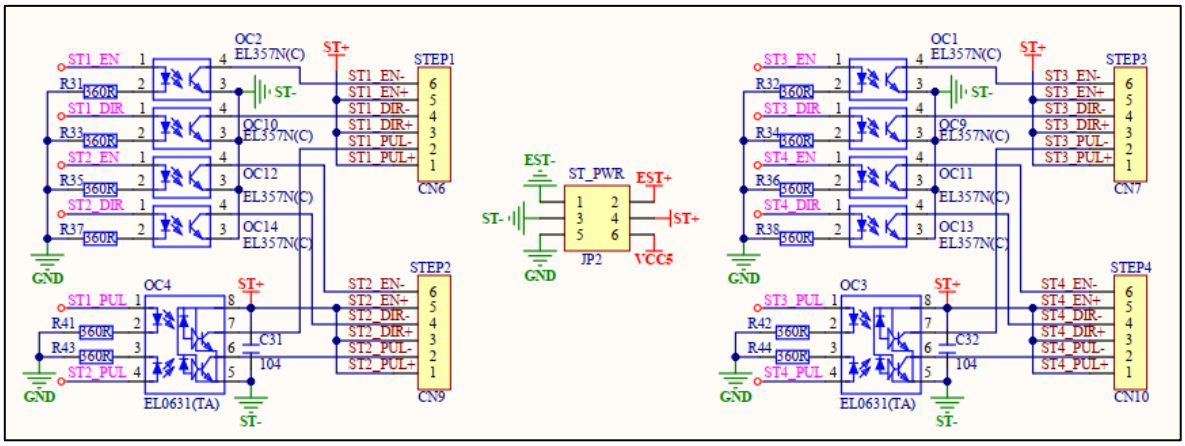


图 15.6.1.2 开发板步进电机接口原理图

图 15.6.1.1 是 DMF407 电机开发板四个步进电机接口的原理图，方向、脉冲、使能接口都是通过光耦隔离的，JP2 端子用于选择光耦输出信号的电平，一般连接到 GND 和 5V，EL0631(TA)为两路高速光耦，用于隔离高速脉冲信号；4 路使能以及方向引脚则使用两个 4 路普通光耦隔离。**注意：脱机使能引脚和方向使能引脚的电平转换电路实际就是一个反向器，因为是反向器，所以控制电平需要反过来。**

15.6.2 程序设计

15.6.2.1 定时器的 HAL 库驱动

本节将会用到两个方式驱动步进电机，即有两个实验。实验中将用到的定时器脉冲输出模式，其中一个是 **PWM1 模式**，另外一个是**输出比较翻转模式**来控制电机旋转。用到的 HAL 库驱动请回顾高级定时器 PWM 输出实验以及输出比较实验的介绍。下面介绍一下步进电机 PWM 模式控制以及翻转模式控制的配置步骤。

1、PWM 模式驱动步进电机的配置步骤

1）开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出。

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1、2、3、4，对应 IO 分别是 PI5、PI6、PI7，PC9 它们的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE();            /* 使能定时器 8 */
__HAL_RCC_GPIOI_CLK_ENABLE();           /* 开启 GPIOI 时钟 */
__HAL_RCC_GPIOC_CLK_ENABLE();           /* 开启 GPIOC 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数。

使用定时器的 PWM 模式功能时，我们调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数来完成对定时器底层以及其输出通道 IO 的初始化，包括：定时器及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

3）设置定时器为 PWM 模式，输出比较极性，比较值等参数。

在 HAL 库中，通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为 PWM1 模式或者 PWM2 模式，根据需求设置输出比较的极性，设置比较值（控制占空比）等。
本实验我们设置 TIM8 的通道 1、2、3、4 为 PWM1 模式。

4）初始化方向 DIR 引脚，脱机使能 EN 引脚

分别初始化四个电机接口的方向引脚以及使能引脚。

2、翻转模式驱动步进电机的配置步骤

1）开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出。

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 1、2、3、4，对应 IO 是 PI5\PI6\PI7\PC9，它们的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE();            /* 使能定时器 8 */
__HAL_RCC_GPIOI_CLK_ENABLE();           /* 开启 GPIOI 时钟 */
__HAL_RCC_GPIOC_CLK_ENABLE();           /* 开启 GPIOC 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数。

使用定时器的输出比较模式时，我们调用的是 HAL_TIM_OC_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_OC_MspInit 函数来完成对定时器底层以及其输出通道 IO 的初始化，包括：定时器及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

3）设置定时器为输出比较模式，输出比较极性，输出比较值、翻转功能等参数。

在 HAL 库中，通过 HAL_TIM_OC_ConfigChannel 函数来设置定时器为输出比较模式，根据需求设置输出比较的极性，设置输出比较值、翻转功能等。

最后我们通过__HAL_TIM_ENABLE_OCxPRELOAD 函数使能通道的预装载。

4）编写定时器输出中断服务函数

在中断函数里边设置新的比较值使其源源不断输出脉冲。

5）开启定时器并输出 PWM

通过 HAL_TIM_OC_Start_IT 函数使能定时器并开启输出。

15.6.2.2 程序流程图

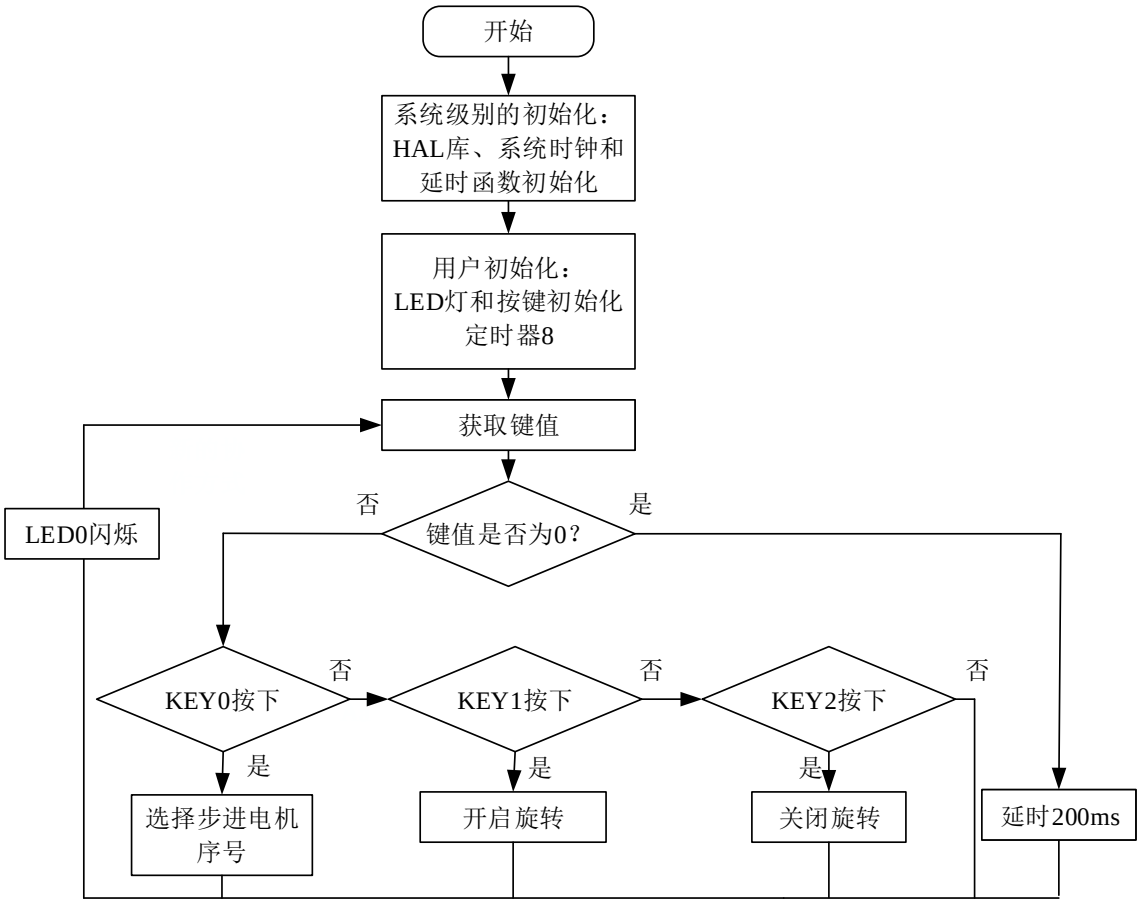


图 15.6.2.2.1 步进电机控制流程图

15.6.2.3 程序解析

1、PWM 模式驱动步进电机

这里我们只讲解核心代码，详细的源码请大家参考光盘中本实验所对应的源码。步进电机的控制源码主要有：定时器驱动源码和步进电机驱动源码四个文件：stepper_tim.c 和 stepper_tim.h 以及 stepper_motor.c 和 stepper_motor.h。源码中都有明确的注释。

首先看 stepper_tim.h 头文件的几个宏定义：

```
/* 高级定时器 定义 */

 /* TIMX PWM 定义
 * 默认使用的是 TIM8_CH1.
 * 注意：通过修改这几个宏定义，可以支持 TIM1/TIM8 定时器
 */
#define ATIM_TIMX_PWM_CH1_GPIO_PORT             GPIOI
#define ATIM_TIMX_PWM_CH1_GPIO_PIN              GPIO_PIN_5
#define ATIM_TIMX_PWM_CH1_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)    /* PI 口时钟使能 */

#define ATIM_TIMX_PWM_CH2_GPIO_PORT             GPIOI
#define ATIM_TIMX_PWM_CH2_GPIO_PIN              GPIO_PIN_6
#define ATIM_TIMX_PWM_CH2_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)    /* PI 口时钟使能 */

#define ATIM_TIMX_PWM_CH3_GPIO_PORT             GPIOI
#define ATIM_TIMX_PWM_CH3_GPIO_PIN              GPIO_PIN_7
```

```
#define ATIM_TIMX_PWM_CH3_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOI_CLK_ENABLE(); }while(0)    /* PI 口时钟使能 */

#define ATIM_TIMX_PWM_CH4_GPIO_PORT            GPIOC
#define ATIM_TIMX_PWM_CH4_GPIO_PIN             GPIO_PIN_9
#define ATIM_TIMX_PWM_CH4_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOC_CLK_ENABLE(); }while(0)    /* PC 口时钟使能 */

#define ATIM_TIMX_PWM_CHY_GPIO_AF              GPIO_AF3_TIM8

#define ATIM_TIMX_PWM                          TIM8
#define ATIM_TIMX_PWM_IRQn                     TIM8_UP_TIM13_IRQn
#define ATIM_TIMX_PWM_IRQHandler               TIM8_UP_TIM13_IRQHandler
#define ATIM_TIMX_PWM_CH1                      TIM_CHANNEL_1
/* 通道 Y,  1<= Y <=4 */
#define ATIM_TIMX_PWM_CH2                      TIM_CHANNEL_2
/* 通道 Y,  1<= Y <=4 */
#define ATIM_TIMX_PWM_CH3                      TIM_CHANNEL_3
/* 通道 Y,  1<= Y <=4 */
#define ATIM_TIMX_PWM_CH4                      TIM_CHANNEL_4
/* 通道 Y,  1<= Y <=4 */

#define ATIM_TIMX_PWM_CHY_CCRX                 TIM8->CCR1
/* 通道 Y 的输出比较寄存器 */
#define ATIM_TIMX_PWM_CHY_CLK_ENABLE()
do{  __HAL_RCC_TIM8_CLK_ENABLE(); }while(0)     /* TIM8 时钟使能 */
```

可以把上面的宏定义分成两部分，第一部分是定时器 8 输出通道 1、2、3、4 对应的 IO 口的宏定义，第二部分则是定时器 8 输出通道 1、2、3、4 的相应宏定义。下面我们来看下 stepper_tim.c 程序

```
/**
 * @brief       高级定时器 TIMX 通道 Y 输出指定个数 PWM 初始化函数
 * @note
 *              高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void atim_timx_oc_chy_init(uint16_t arr, uint16_t psc)
{
    ATIM_TIMX_PWM_CHY_CLK_ENABLE();       /* TIMX 时钟使能 */


    g_atimx_handle.Instance = ATIM_TIMX_PWM;                /* 定时器 x */
    g_atimx_handle.Init.Prescaler = psc;                    /* 定时器分频 */
    g_atimx_handle.Init.CounterMode = TIM_COUNTERMODE_UP;   /* 向上计数模式 */
    g_atimx_handle.Init.Period = arr;                       /* 自动重装载值 */
    g_atimx_handle.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;/* 分频因子 */
    g_atimx_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    g_atimx_handle.Init.RepetitionCounter = 0;              /* 开始时不计数*/
    HAL_TIM_PWM_Init(&g_atimx_handle);                      /* 初始化 PWM */

    g_atimx_oc_chy_handle.OCMode = TIM_OCMODE_PWM1;         /* 模式选择 PWM1 */
    g_atimx_oc_chy_handle.Pulse = arr/2;
    g_atimx_oc_chy_handle.OCPolarity = TIM_OCPOLARITY_HIGH;/* 输出比较极性为低 */
    g_atimx_oc_chy_handle.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    g_atimx_oc_chy_handle.OCFastMode = TIM_OCFAST_DISABLE;
    g_atimx_oc_chy_handle.OCIdleState = TIM_OCIDLESTATE_RESET;
```

```
    g_atimx_oc_chy_handle.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH1); /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH2); /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH3); /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH4); /* 配置 TIMx 通道 y */

}
```

HAL_TIM_PWM_Init 初始化 TIM8 并设置 TIM8 的 ARR 和 PSC 等参数，然后通过调用函数 HAL_TIM_PWM_ConfigChannel 设置 TIM8_CH1、2、3、4 的 PWM 模式以及比较值等参数，最后通过调用函数 HAL_TIM_PWM_Start 来使能 TIM8 以及使能 PWM 通道 TIM8_CH1、2、3、4 输出。其中定时器四个通道的 GPIO 口初始化和定时器使能等程序放到回调函数 HAL_TIM_PWM_MspInit 中这里就不贴代码了，感兴趣打开源码看看。接着我们来看下 stepper_motor.h:

```
/* 步进电机引脚定义*/

#define STEPPER_MOTOR_1       1                 /* 步进电机接口序号 */
#define STEPPER_MOTOR_2       2
#define STEPPER_MOTOR_3       3
#define STEPPER_MOTOR_4       4
/*     步进电机方向引脚定义     */

#define STEPPER_DIR1_GPIO_PIN                   GPIO_PIN_14
#define STEPPER_DIR1_GPIO_PORT                  GPIOF
#define STEPPER_DIR1_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PF 口时钟使能 */

#define STEPPER_DIR2_GPIO_PIN                   GPIO_PIN_12
#define STEPPER_DIR2_GPIO_PORT                  GPIOF
#define STEPPER_DIR2_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PF 口时钟使能 */

#define STEPPER_DIR3_GPIO_PIN                   GPIO_PIN_2
#define STEPPER_DIR3_GPIO_PORT                  GPIOB
#define STEPPER_DIR3_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)    /* PB 口时钟使能 */

#define STEPPER_DIR4_GPIO_PIN                   GPIO_PIN_2
#define STEPPER_DIR4_GPIO_PORT                  GPIOH
#define STEPPER_DIR4_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOH_CLK_ENABLE(); }while(0)    /* PH 口时钟使能 */

/*     步进电机脱机引脚定义     */

#define STEPPER_EN1_GPIO_PIN                    GPIO_PIN_15
#define STEPPER_EN1_GPIO_PORT                   GPIOF
#define STEPPER_EN1_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PF 口时钟使能 */

#define STEPPER_EN2_GPIO_PIN                    GPIO_PIN_13
#define STEPPER_EN2_GPIO_PORT                   GPIOF
#define STEPPER_EN2_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PF 口时钟使能 */

#define STEPPER_EN3_GPIO_PIN                    GPIO_PIN_11
#define STEPPER_EN3_GPIO_PORT                   GPIOF
#define STEPPER_EN3_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PF 口时钟使能 */
```

```
#define STEPPER_EN4_GPIO_PIN                    GPIO_PIN_3
#define STEPPER_EN4_GPIO_PORT                   GPIOH
#define STEPPER_EN4_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOH_CLK_ENABLE(); }while(0)    /* PH 口时钟使能 */

/*----------------------- 方向引脚控制 ------------------------------------*/
/* 由于我们使用的是共阳极解法所以当 x = 1 有效，x = 0 时无效*/
#define ST1_DIR(x)    do{ x ? \
                          HAL_GPIO_WritePin(STEPPER_DIR1_GPIO_PORT,
STEPPER_DIR1_GPIO_PIN, GPIO_PIN_SET) : \
                          HAL_GPIO_WritePin(STEPPER_DIR1_GPIO_PORT,
STEPPER_DIR1_GPIO_PIN, GPIO_PIN_RESET); \
                      }while(0)

#define ST2_DIR(x)    do{ x ? \
                          HAL_GPIO_WritePin(STEPPER_DIR2_GPIO_PORT,
STEPPER_DIR2_GPIO_PIN, GPIO_PIN_SET) : \
                          HAL_GPIO_WritePin(STEPPER_DIR2_GPIO_PORT,
STEPPER_DIR2_GPIO_PIN, GPIO_PIN_RESET); \
                      }while(0)

#define ST3_DIR(x)    do{ x ? \
                          HAL_GPIO_WritePin(STEPPER_DIR3_GPIO_PORT,
STEPPER_DIR3_GPIO_PIN, GPIO_PIN_SET) : \
                          HAL_GPIO_WritePin(STEPPER_DIR3_GPIO_PORT,
STEPPER_DIR3_GPIO_PIN, GPIO_PIN_RESET); \
                      }while(0)

#define ST4_DIR(x)    do{ x ? \
                          HAL_GPIO_WritePin(STEPPER_DIR4_GPIO_PORT,
STEPPER_DIR4_GPIO_PIN, GPIO_PIN_SET) : \
                          HAL_GPIO_WritePin(STEPPER_DIR4_GPIO_PORT,
STEPPER_DIR4_GPIO_PIN, GPIO_PIN_RESET); \
                      }while(0)

/*----------------------- 脱机引脚控制 ------------------------------------*/
/* 由于我们使用的是共阳极解法所以当 x = 1 有效，x = 0 时无效*/
#define ST1_EN(x)     do{ x ? \
                      HAL_GPIO_WritePin(STEPPER_EN1_GPIO_PORT,
STEPPER_EN1_GPIO_PIN, GPIO_PIN_SET) : \
                      HAL_GPIO_WritePin(STEPPER_EN1_GPIO_PORT,
STEPPER_EN1_GPIO_PIN, GPIO_PIN_RESET); \
                    }while(0)

#define ST2_EN(x)     do{ x ? \
                      HAL_GPIO_WritePin(STEPPER_EN2_GPIO_PORT,
STEPPER_EN2_GPIO_PIN, GPIO_PIN_SET) : \
                      HAL_GPIO_WritePin(STEPPER_EN2_GPIO_PORT,
STEPPER_EN2_GPIO_PIN, GPIO_PIN_RESET); \
                    }while(0)

#define ST3_EN(x)     do{ x ? \
                      HAL_GPIO_WritePin(STEPPER_EN3_GPIO_PORT,
STEPPER_EN3_GPIO_PIN, GPIO_PIN_SET) : \
                      HAL_GPIO_WritePin(STEPPER_EN3_GPIO_PORT,
STEPPER_EN3_GPIO_PIN, GPIO_PIN_RESET); \
                    }while(0)

#define ST4_EN(x)     do{ x ? \
                      HAL_GPIO_WritePin(STEPPER_EN4_GPIO_PORT,
STEPPER_EN4_GPIO_PIN, GPIO_PIN_SET) : \
                      HAL_GPIO_WritePin(STEPPER_EN4_GPIO_PORT,
STEPPER_EN4_GPIO_PIN, GPIO_PIN_RESET); \
                    }while(0)
```

这里可以主要分为三部分，第一部分是方向引脚和脱机引脚的IO宏定义以及时钟定义等；

第二部分是四个步进电机接口的脱机控制以及方向控制宏定义；第三部分是四个接口的 id 宏定义。接着来看下 stepper_motor.c，首先看下步进电机的初始化函数：

```
/**
 * @brief       初始化步进电机相关 IO 口，并使能时钟
 * @param       arr: 自动重装值
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void stepper_init(uint16_t arr, uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct;

    STEPPER_DIR1_GPIO_CLK_ENABLE();                         /* DIR1 时钟使能 */
    STEPPER_DIR2_GPIO_CLK_ENABLE();                         /* DIR2 时钟使能 */
    STEPPER_DIR3_GPIO_CLK_ENABLE();                         /* DIR3 时钟使能 */
    STEPPER_DIR4_GPIO_CLK_ENABLE();                         /* DIR4 时钟使能 */

    STEPPER_EN1_GPIO_CLK_ENABLE();                          /* EN1 时钟使能 */
    STEPPER_EN2_GPIO_CLK_ENABLE();                          /* EN2 时钟使能 */
    STEPPER_EN3_GPIO_CLK_ENABLE();                          /* EN3 时钟使能 */
    STEPPER_EN4_GPIO_CLK_ENABLE();                          /* EN4 时钟使能 */


    gpio_init_struct.Pin = STEPPER_DIR1_GPIO_PIN;           /* DIR1 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;            /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLDOWN;                  /* 下拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_LOW;           /* 低速 */
    HAL_GPIO_Init(STEPPER_DIR1_GPIO_PORT, &gpio_init_struct);  /* 初始化 DIR1 */

    gpio_init_struct.Pin = STEPPER_DIR2_GPIO_PIN;               /* DIR2 引脚 */
    HAL_GPIO_Init(STEPPER_DIR2_GPIO_PORT, &gpio_init_struct);  /* 初始化 DIR2 */

    gpio_init_struct.Pin = STEPPER_DIR3_GPIO_PIN;               /* DIR3 引脚 */
    HAL_GPIO_Init(STEPPER_DIR3_GPIO_PORT, &gpio_init_struct);  /*初始化 DIR3 */

    gpio_init_struct.Pin = STEPPER_DIR4_GPIO_PIN;               /* DIR4 引脚 */
    HAL_GPIO_Init(STEPPER_DIR4_GPIO_PORT, &gpio_init_struct);  /* 初始化 DIR4 */

    /*   脱机引脚初始化   */
    gpio_init_struct.Pin = STEPPER_EN1_GPIO_PIN;                /* EN1 引脚 */
    HAL_GPIO_Init(STEPPER_EN1_GPIO_PORT, &gpio_init_struct);   /*初始化 EN1 引脚*/

    gpio_init_struct.Pin = STEPPER_EN2_GPIO_PIN;                /* EN2 引脚 */
    HAL_GPIO_Init(STEPPER_EN2_GPIO_PORT, &gpio_init_struct);   /*初始化 EN2 引脚*/

    gpio_init_struct.Pin = STEPPER_EN3_GPIO_PIN;                /* EN3 引脚 */
    HAL_GPIO_Init(STEPPER_EN3_GPIO_PORT, &gpio_init_struct);   /*初始化 EN3 引脚*/

    gpio_init_struct.Pin = STEPPER_EN4_GPIO_PIN;                /* EN4 引脚 */
    HAL_GPIO_Init(STEPPER_EN4_GPIO_PORT, &gpio_init_struct);   /*初始化 EN4 引脚*/

    atim_timx_oc_chy_init(arr, psc);                /* 初始化 PUL 引脚，以及脉冲模式等 */
}
```

这个函数主要就是初始化步进电机的方向 IO 和脱机 IO 等并调用了定时器的初始化函数，方便后续直接通过一个函数来实现步进电机的初始化。接着来看下步进电机相关的控制函数：开启旋转以及关闭旋转：

```
/**
 * @brief       开启步进电机
 * @param       motor_num: 步进电机接口序号
```

```
 * @retval     无
 */
void stepper_star(uint8_t motor_num)
{
    switch(motor_num)
    {
        case STEPPER_MOTOR_1 :
        {
            /* 开启对应 PWM 通道 */
            HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH1);
            break;
        }
        case STEPPER_MOTOR_2 :
        {
            /* 开启对应 PWM 通道 */
            HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH2);
            break;
        }
        case STEPPER_MOTOR_3 :
        {
            /* 开启对应 PWM 通道 */
            HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
            break;
        }
        case STEPPER_MOTOR_4 :
        {
            /* 开启对应 PWM 通道 */
            HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH4);
            break;
        }
        default : break;
    }
}

/**
 * @brief      关闭步进电机
 * @param      motor_num: 步进电机接口序号
 * @retval     无
 */
void stepper_stop(uint8_t motor_num)
{
    switch(motor_num)
    {
        case STEPPER_MOTOR_1 :
        {
            /* 关闭对应 PWM 通道 */
            HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH1);
            break;
        }
        case STEPPER_MOTOR_2 :
        {
            /* 关闭对应 PWM 通道 */
            HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH2);
            break;
        }
        case STEPPER_MOTOR_3 :
        {
            /* 关闭对应 PWM 通道 */
            HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
            break;
        }
        case STEPPER_MOTOR_4 :
        {
            /* 关闭对应 PWM 通道 */
            HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH4);
```

```
            break;
        }
        default : break;
    }
}
```

首先开启步进电机旋转函数主要调用的 HAL_TIM_PWM_Start 函数开启 PWM 的输出，然后通过 id 号选择控制哪个接口的 PWM 输出，这样就可以实现控制对应接口的步进电机旋转了。关闭函数，主要调用的就是 HAL_TIM_PWM_Stop 函数，用来关闭 PWM 输出的，通过 id 号选择关闭哪路接口的 PWM 输出，这样就可以实现控制关闭对应接口的步进电机了。

在 main 函数里面编写如下代码：

```
int main(void)
{
    uint8_t key,t,id = 1;
    char buf[32];

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    /* 168 000 000 / 168 = 1000 000 1Mhz 的计数频率，计数 1K 次为 1ms */
    stepper_init(1000 - 1, 168 - 1);
    g_point_color = WHITE;
    g_back_color  = BLACK;
    /* LCD 显示提示信息 */
    lcd_show_string(10,10,200,16,16,"Stepper Test",g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:ID + +",g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:star",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:stop",g_point_color);
    lcd_show_string(10,90,200,16,16,"Stepper ID: 1",g_point_color);
    /* 串口打印提示信息 */
    printf("按下 KEY0 选择控制哪个步进电机\r\n");
    printf("按下 KEY1 开启步进电机\r\n");
    printf("按下 KEY2 关闭步进电机\r\n");
    printf("当前控制步进电机接口%d\r\n",id);

    while (1)
    {
        key = key_scan(0);
        if(key == KEY0_PRES)
        {
            id++;
            if(id > 4)
            {
                id = 1;
            }
            printf("当前控制步进电机接口%d\r\n",id);          /* 串口打印提示信息 */
            sprintf(buf,"Stepper ID: %1d",id);
            lcd_show_string(10,90,200,16,16,buf,g_point_color);
        }
        else if(key == KEY1_PRES)                           /* 按下 KEY1 开启电机 */
        {
            stepper_star(id);
            printf("开启步进电机\r\n");
        }
        else if(key == KEY2_PRES)                           /* 按下 KEY2 关闭电机 */
        {
            stepper_stop(id);
            printf("关闭步进电机\r\n");
        }
```

```
        t++;
        if(t % 20 == 0)
        {
            LED0_TOGGLE();                                  /* LED0(红灯) 翻转 */
        }
        delay_ms(10);
    }
}
```

先看 stepper_init(1000 - 1, 168 - 1);这个语句，这两个形参分别设置自动重载寄存器的值为 1000，以及定时器预分频系数为 168。先看预分频系数，我们设置为 168 分频，然后定时器 8 的时钟是 2 倍的 APB2，即 168MHZ，可以得到计数器的计数频率是 1MHZ，也就是计数一次 1us，我们计数 1000 次就是 1kz 的频率脉冲，这是我们控制步进电机的初始速度，如果太快有可能出现丢步或堵转现象。在 LCD 上会显示一些提示信息。大家需要提前将步进电机接入对应的接口，按键 KEY0 设置控制哪个接口的步进电机，KEY1 开启电机旋转°，KEY2 关闭电机旋转°。这个就是 PWM 驱动步进电机的整个代码过程了。接着看下第二种驱动方式。

2、翻转模式驱动步进电机

同样的我们只讲解核心代码，详细的源码请大家参考光盘中本实验所对应的源码。此代码在 PWM 驱动方式上添加而来。步进电机的控制源码主要有：定时器驱动源码和步进电机驱动源码四个文件：stepper_tim.c 和 stepper_tim.h 以及 stepper_motor.c 和 stepper_motor.h。源码中有明确的注释。其中与 PWM 的驱动方式，同样的部分就不再赘述了，只讲解不同部分。来看下定时器的驱动 stepper_tim.c：

```
/**
 * @brief       高级定时器 TIMX 输出比较翻转模式初始化函数
 * @note
 *              高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void atim_timx_oc_chy_init(uint16_t arr, uint16_t psc)
{
    ATIM_TIMX_PWM_CHY_CLK_ENABLE();       /* TIMX 时钟使能 */


    g_atimx_handle.Instance = ATIM_TIMX_PWM;                    /* 定时器 x */
    g_atimx_handle.Init.Prescaler = psc;                        /* 定时器分频 */
    g_atimx_handle.Init.CounterMode = TIM_COUNTERMODE_UP;       /* 向上计数模式 */
    g_atimx_handle.Init.Period = arr;                           /* 自动重装载值 */
    g_atimx_handle.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;   /* 分频因子 */
    g_atimx_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    g_atimx_handle.Init.RepetitionCounter = 0;                  /* 开始时不计数*/
    HAL_TIM_OC_Init(&g_atimx_handle);                           /* 初始化 PWM */

    g_atimx_oc_chy_handle.OCMode = TIM_OCMODE_TOGGLE;           /*模式选择 PWM1*/
    g_atimx_oc_chy_handle.Pulse = 0;
    g_atimx_oc_chy_handle.OCPolarity = TIM_OCPOLARITY_HIGH;  /*输出比较极性为低 */
    g_atimx_oc_chy_handle.OCNPolarity = TIM_OCNPOLARITY_LOW;
    g_atimx_oc_chy_handle.OCFastMode = TIM_OCFAST_DISABLE;
    g_atimx_oc_chy_handle.OCIdleState = TIM_OCIDLESTATE_RESET;
    g_atimx_oc_chy_handle.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    HAL_TIM_OC_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH1); /* 配置 TIMx 通道 y */
```

```
    HAL_TIM_OC_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH2); /* 配置 TIMx 通道 y */
    HAL_TIM_OC_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH3); /* 配置 TIMx 通道 y */
    HAL_TIM_OC_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH4); /* 配置 TIMx 通道 y */
     HAL_TIM_Base_Start(&g_atimx_handle);
}
```

HAL_TIM_OC_Init 初始化 TIM8 并设置 TIM8 的 ARR 和 PSC 等参数，其次通过调用函数 HAL_TIM_OC_ConfigChannel 设置通道 1~通道 4 的配置参数（包括模式功能、输出比较寄存器的值，输出极性等），最后通过调用函数 HAL_TIM_OC_Start 来启动 TIM1 通道 1~通道 4 输出。此步骤过程跟通用定时器 PWM 输出实验基本相似。在回调函数 HAL_TIM_OC_MspInit 中多了一个输出中断使能，因为我们需要在中断函数里边不断修改比较值让其源源不断输出脉冲，我们看下中断函数的代码：

```
/**
 * @brief       高级定时器 TIMX NPWM 中断服务函数
 * @param       无
 * @retval      无
 */
void ATIM_TIMX_INT_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&g_atimx_handle); /* 定时器共用处理函数 */
}

/**
  * @brief  定时器比较中断
  * @param  htim：定时器句柄指针
    *   @note       无
  * @retval 无
  */
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim)
{

    /*获取当前计数*/
    g_count_val = __HAL_TIM_GET_COUNTER(&g_atimx_handle);

    /*设置比较数值*/
    if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
    __HAL_TIM_SET_COMPARE(&g_atimx_handle, ATIM_TIMX_PWM_CH1, (g_count_val +
                          500)%0XFFFF);

    if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2)
    __HAL_TIM_SET_COMPARE(&g_atimx_handle, ATIM_TIMX_PWM_CH2, (g_count_val +
                          500)%0XFFFF);

    if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_3)
    __HAL_TIM_SET_COMPARE(&g_atimx_handle, ATIM_TIMX_PWM_CH3, (g_count_val +
                          500)%0XFFFF);

    if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_4)
    __HAL_TIM_SET_COMPARE(&g_atimx_handle, ATIM_TIMX_PWM_CH4, (g_count_val +
                          500)%0XFFFF);
}
```

ATIM_TIMX_INT_IRQHandler 函数为定时器输出中断服务函数，而这个函数实际上就是调用 HAL 库中的定时器公用中断处理函数 HAL_TIM_IRQHandler。逻辑程序在定时器的中断回调函数 HAL_TIM_OC_DelayElapsedCallback 中处理。主要操作就是不断修改比较值，让计数值每次与之匹配就翻转电平，达到不断输出脉冲的目的。

接着看下 stepper_motor.c 源码，它跟 PWM 驱动方式不一样的地方在于启动旋转函数以及关闭旋转函数的区别，具体如下：

```
/**
```

```
 * @brief       开启步进电机
 * @param       motor_num: 步进电机接口序号
 * @retval      无
 */
void stepper_star(uint8_t motor_num)
{
    switch(motor_num)
    {
        case STEPPER_MOTOR_1 :
        {
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM1||
              g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM2)
            {
              /* 开启对应 PWM 通道 */
                HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH1);
            }
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_TOGGLE)
            {
              /* 开启对应 PWM 通道 */
                HAL_TIM_OC_Start_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH1);
            }
            break;
        }
        case STEPPER_MOTOR_2 :
        {
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM1||
              g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM2)
            {
                HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH2);
            }
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_TOGGLE)
            {
                HAL_TIM_OC_Start_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH2);
            }
            break;
        }
        case STEPPER_MOTOR_3 :
        {
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM1||
              g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM2)
            {
                HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
            }
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_TOGGLE)
            {
                HAL_TIM_OC_Start_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
            }
            break;
        }
        case STEPPER_MOTOR_4 :
        {
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM1||
              g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM2)
            {
                HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH4);
            }
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_TOGGLE)
            {
                HAL_TIM_OC_Start_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH4);
            }
            break;
        }
        default : break;
    }
}
```

```
/**
 * @brief       关闭步进电机
 * @param       motor_num: 步进电机接口序号
 * @retval      无
 */
void stepper_stop(uint8_t motor_num)
{
    switch(motor_num)
    {
        case STEPPER_MOTOR_1 :
        {
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM1||
              g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM2)
            {
                HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH1);
            }
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_TOGGLE)
            {
                HAL_TIM_OC_Stop_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH1);
            }
            break;
        }
        case STEPPER_MOTOR_2 :
        {
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM1||
              g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM2)
            {
                HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH2);
            }
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_TOGGLE)
            {
                HAL_TIM_OC_Stop_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH2);
            }
            break;
        }
        case STEPPER_MOTOR_3 :
        {
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM1||
              g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM2)
            {
                HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
            }
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_TOGGLE)
            {
                HAL_TIM_OC_Stop_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
            }
            break;
        }
        case STEPPER_MOTOR_4 :
        {
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM1||
              g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_PWM2)
            {
                HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH4);
            }
            if(g_atimx_oc_chy_handle.OCMode == TIM_OCMODE_TOGGLE)
            {
                HAL_TIM_OC_Stop_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH4);
            }
            break;
        }
        default : break;
    }
}
```

由于输出比较用到了输出中断所以需要调用 HAL_TIM_OC_Start_IT 函数开启脉冲输出，

关闭输出函数同理，输出比较使用的是 HAL_TIM_OC_Stop_IT 来关闭输出中断。main 函数部分，和 pwm 驱动模式完全一模一样的这里就不贴出来了。

15.6.3 下载验证

两种驱动模式的下载现象一致，所以这里统一进行讲解。将代码下载进去开发板，可以看到 LED0 每隔 200ms 左右闪烁一次，代表程序正在运行。此时复位开发板可以看到串口打印提示信息如图 15.6.3.1 所示，且 LCD 以及串口都会显示当前所控制的步进电机接口号，当我们按下 KEY0 选择控制哪个步进电机接口，按下 KEY1 开启电机旋转，KEY2 关闭电机旋转。

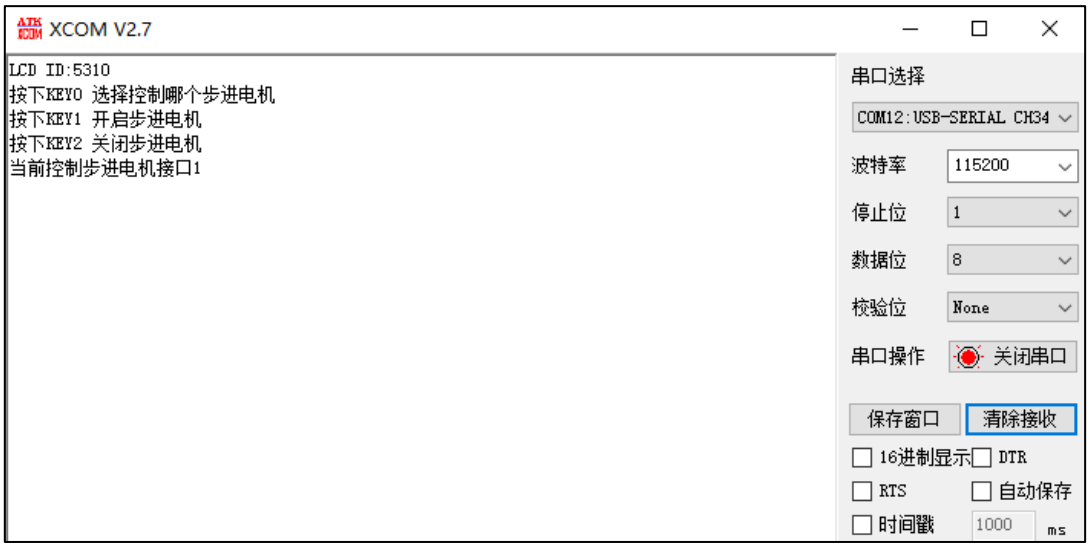


图 15.6.3.1 串口提示信息

15.7 步进电机定位控制实现

我们知道步进电机的最大特点是能够精确的步进一定的角度，而在前面的基础控制中，因为我们施加的脉冲是连续不断的脉冲，所以步进电机的旋转也和直流电机一样在持续的转动，这种控制其实并不能很好的展现步进电机的运行特点，所以为了更好体现步进电机的特性，接下来这一内容将通过控制脉冲的输出数量使步进电机步进到一定的角度后停止，即实现了所说的定位控制。

15.7.1 硬件设计

1、例程功能

本节我们将实现步进电机的定位控制，即让电机转动到指定角度之后停止，我们本实验使用步进电机接口一，也就是使用 TIM8_CH1（PI5）输出脉冲控制，按键 KEY0 设置旋转角度正向加 90°，按键 KEY1 设置旋转角度反向加 90°，KEY2 开启电机旋转。同样的本节将会使用到两种驱动方式：PWM 模式和比较输出翻转模式。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器 8： TIM8_CH1：PI5，对应步进电机接口一
方向引脚： DIR1: PF14
脱机引脚： EN1: PF15

4）步进电机
5）步进电机驱动器
6）12-50V 的 DC 电源

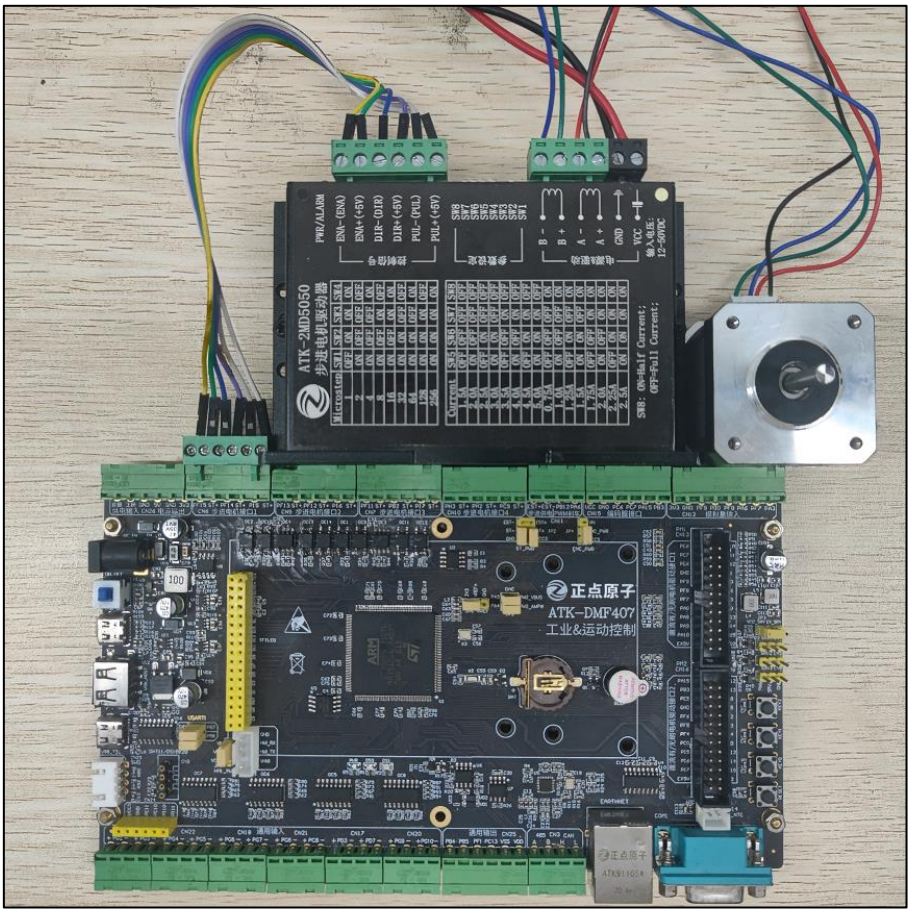


图 15.7.1.1 硬件连接（接口一）

3、原理图

与步进电机旋转控制实现的 15.6.1 小节一致，这里就不重复了。

15.7.2 程序设计

15.7.2.1 实现原理解析

原理介绍：本实验将驱动两相混合式步进电机，步距角为 1.8°，也就是说在不细分的情况下旋转一圈 360° 需要 200 个脉冲，我们为了旋转更加流畅，本实验使用 **8 细分**，实际输出步距角为：1.8° / 8 = 0.225°，所以说旋转 360° 就需要 360 / 0.225 = 1600 个脉冲数，根据这个特性我们可以将角度转换成脉冲数，然后通过控制脉冲个数，驱动步进电机旋转到指定位置停止。注意：PWM 模式，进入一次更新中断输出一个脉冲，即发生一次中断对应一个完整脉冲；而比较输出翻转模式需要进入两次输出中断才为一个完整脉冲。

15.7.2.2 步进电机定位控制配置步骤

1、PWM 驱动模式

1）初始化定时器，配置 PWM 模式 1

使用定时器的 PWM 模式功能时，我们调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数来完成对定时器底层以及其输出通道 IO 的初始化，包括：定时器及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

2）初始化接口一的方向引脚，使能引脚等

3）使能定时器更新中断，开启定时器计数，配置定时器中断优先级

通过 HAL_TIM_Base_Start_IT 函数使能定时器更新中断和开启定时器计数。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断，通过 HAL_NVIC_SetPriority 函数设置中断优先级。

4）编写角度转脉冲个数函数

实现把所需要转动的角度，转换成脉冲个数。

5）编写中断服务函数

实现脉冲计数，当达到指定的脉冲数时就关闭 PWM 输出。**注意进入一次更新中断，输出一个完整脉冲。**

2、翻转模式

1）初始化定时器，配置翻转模式

使用定时器的 PWM 模式功能时，我们调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数来完成对定时器底层以及其输出通道 IO 的初始化，包括：定时器及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

2）初始化接口一的方向引脚，使能引脚等

3）使能定时器输出中断，开启定时器计数，配置定时器中断优先级

通过 HAL_TIM_OC_Start_IT 函数使能定时器更新中断和开启定时器计数。

通过 HAL_NVIC_EnableIRQ 函数使能定时器中断，通过 HAL_NVIC_SetPriority 函数设置中断优先级。

4）编写角度转脉冲个数函数

实现把所需要转动的角度，转换成脉冲个数。

5）编写中断服务函数

实现脉冲计数，当达到指定的脉冲数时就关闭脉冲输出。**注意进入两次输出中断，输出一个完整脉冲。**

15.7.2.3 程序流程图

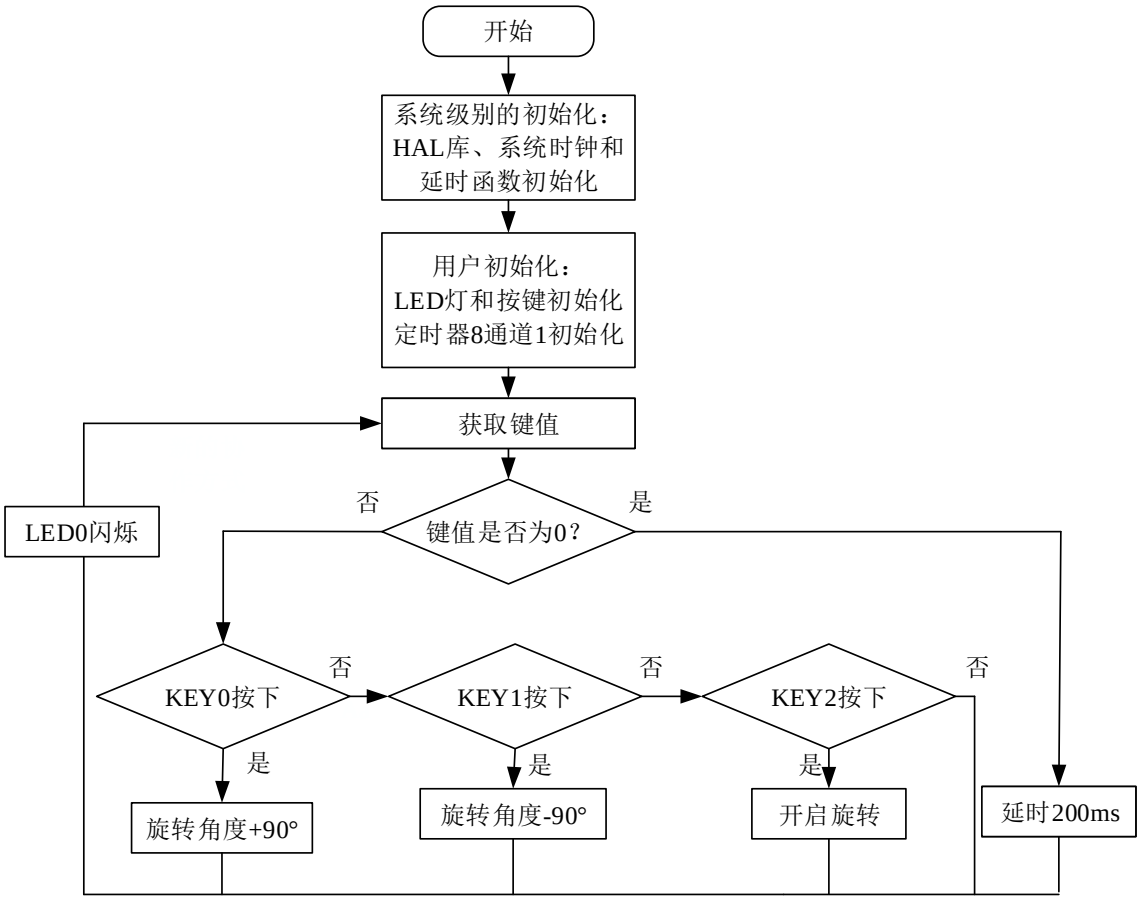


图 15.7.2.3.1 定位控制流程图

15.7.2.4 程序解析

1、PWM 模式驱动步进电机

这里我们只讲解核心代码，详细的源码请大家参考光盘中本实验对应源码。步进电机的控制源码主要有：定时器驱动源码和步进电机驱动源码四个文件：stepper_tim.c 和 stepper_tim.h 以及 stepper_motor.c 和 stepper_motor.h。定位控制代码在旋转控制的代码后面追加。

其中 stepper_tim.h 头文件与上一节课是完全一模一样的这里就不重复贴出来了，感兴趣看下旋转控制代码解析部分。

接着看下 stepper_tim.c 的定时器初始化：

```
/**
 * @brief       高级定时器 TIMX PWM 初始化函数
 * @note
 *              高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void atim_timx_oc_chy_init(uint16_t arr, uint16_t psc)
{
    ATIM_TIMX_PWM_CHY_CLK_ENABLE();                             /* TIMX 时钟使能 */
```

```
    g_atimx_handle.Instance = ATIM_TIMX_PWM;                    /* 定时器 x */
    g_atimx_handle.Init.Prescaler = psc;                        /* 定时器分频 */
    g_atimx_handle.Init.CounterMode = TIM_COUNTERMODE_UP;       /* 向上计数模式 */
    g_atimx_handle.Init.Period = arr;                           /* 自动重装载值 */
    g_atimx_handle.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;   /* 分频因子 */
    g_atimx_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    g_atimx_handle.Init.RepetitionCounter = 0;                  /* 开始时不计数*/
    HAL_TIM_PWM_Init(&g_atimx_handle);                          /* 初始化 PWM */

    g_atimx_oc_chy_handle.OCMode = TIM_OCMODE_PWM1;             /*模式选择 PWM1 */
    g_atimx_oc_chy_handle.Pulse = arr/2;
    g_atimx_oc_chy_handle.OCPolarity = TIM_OCPOLARITY_HIGH;  /*输出比较极性为低 */
    g_atimx_oc_chy_handle.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    g_atimx_oc_chy_handle.OCFastMode = TIM_OCFAST_DISABLE;
    g_atimx_oc_chy_handle.OCIdleState = TIM_OCIDLESTATE_RESET;
    g_atimx_oc_chy_handle.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH1); /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH2); /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH3); /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
     ATIM_TIMX_PWM_CH4); /* 配置 TIMx 通道 y */

    HAL_NVIC_SetPriority(ATIM_TIMX_UP_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(ATIM_TIMX_UP_IRQn);
}
```

可以看出来和上节课的配置基本一致，只多了一个中断以及使能中断优先级的设置，IO 初始化等都放在回调函数 HAL_TIM_PWM_MspInit 中，这里就不贴出来了，大家可以自行打开源码查看。接着看下关键的代码部分就是中断服务函数的处理：

```
/**
 * @brief       高级定时器 TIMX  PWM 中断服务函数
 * @param       无
 * @retval      无
 */
void ATIM_TIMX_UP_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&g_atimx_handle);    /* 定时器共用处理函数 */
}
/**
 * @brief       高级定时器 TIMX  PWM 中断回调函数
 * @param       htim : 定时器句柄
 * @retval      无
 */

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim->Instance==TIM8)
    {
        g_run_flag = 1;                         /* 标志位置一 */
        g_stepper.pulse_count--;                /* 每一个完整的脉冲就-- */
        if(g_stepper.dir == CW)
        {
          g_stepper.add_pulse_count++;          /* 绝对位置++ */
        }else
        {
          g_stepper.add_pulse_count--;          /* 绝对位置-- */
        }
         /*当脉冲数等于 0 的时候 代表需要发送的脉冲个数已完成，停止定时器输出 */
```

```
        if(g_stepper.pulse_count<=0)
        {
            stepper_stop(STEPPER_MOTOR_1);    /* 停止接口一输出 */
            g_run_flag = 0;
        }
    }
}
```

在中断服务函数里边主要把角度转换后的脉冲个数值进行计数，所需的脉冲个数值保存在 g_stepper.pulse_count，当每进入一次中断服务函数，脉冲个数就减一，当减到 0 时代表已达到指定位置了此时就会关闭定时器脉冲输出。而累计脉冲个数 g_stepper.add_pulse_count 用于保存整个过程相对于起点的绝对位置。

接着看下 stepper_motor.h，这里只贴出与上节课有区别的部分源码:

```
/* 步进电机参数相关宏 */
#define PULSE_REV 1600.0                    /* 每圈脉冲数（细分数 8） */
#define MAX_STEP_ANGLE  0.225               /* 最小步距(1.8/PULSE_REV) */

typedef struct
{
    int angle ;                             /* 设置需要旋转的角度 */
    uint8_t dir;                            /* 方向 */
    uint8_t en;                             /* 使能 */
    volatile uint32_t pulse_count;          /* 脉冲个数记录 */
    volatile int add_pulse_count;           /* 脉冲个数累计 */
} STEPPER_MOTOR;
extern  STEPPER_MOTOR g_stepper;
enum dir
{
    CCW = 0,                                /* 逆时针旋转 */
    CW ,                                    /* 顺时针旋转 */
};
```

宏 PULSE_REV 表示电机旋转一圈所需要的脉冲数，MAX_STEP_ANGLE 表示实际输出的步距角大小，在后面会用于角度值转换成脉冲个数的计算。然后就是 STEPPER_MOTOR 结构体，里边的结构体成员分别用于保存需旋转的角度值、旋转方向、脱机使能、脉冲个数、累计脉冲个数。接着定义了一个枚举类型，便于直观的表示电机是顺时针旋转还是逆时针旋转。

stepper_motor.c 的源码主要添加了**角度转脉冲数**的函数：

```
/**
 * @brief       将需要转动的角度转换成脉冲数
 * @param       angle    : 需要转动的角度值
 * @param       dir      : 旋转方向
 * @param       motor_num: 步进电机接口序号
 * @retval      无
 */
void stepper_set_angle(uint16_t angle, uint8_t dir,uint8_t motor_num)
{
    g_stepper.pulse_count = angle / MAX_STEP_ANGLE;
    if(g_stepper.pulse_count == 0)
    {
        stepper_stop(motor_num);
    }
    else stepper_star(motor_num,dir);
}
```

该函数有三个入口参数分别是：转动的角度值、旋转方向以及控制的步进电机接口序号，我这里用到的是步进电机接口一，然后我们会根据所需转动的角度值除以步距角就可以算出需要几个脉冲才可达到指定角度，之后通过判断脉冲个数决定开启还是关闭电机。除了添加这部分的代码，启动旋转以及关闭旋转添加了以下部分：

```
/**
 * @brief       开启步进电机
 * @param       motor_num: 步进电机接口序号
 * @param       dir      : 步进电机旋转方向
```

```
 * @retval      无
 */
void stepper_star(uint8_t motor_num, uint8_t dir)
{
    HAL_TIM_Base_Start_IT(&g_atimx_handle);
    switch(motor_num)
    {
        case STEPPER_MOTOR_1 :
        {
            ST1_DIR(dir);
         /* 开启对应 PWM 通道 */
            HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH1);
            break;
        }
        case STEPPER_MOTOR_2 :
        {
            ST2_DIR(dir);
            HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH2);
            break;
        }
        case STEPPER_MOTOR_3 :
        {
            ST3_DIR(dir);
            HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
            break;
        }
        case STEPPER_MOTOR_4 :
        {
            ST4_DIR(dir);
            HAL_TIM_PWM_Start(&g_atimx_handle, ATIM_TIMX_PWM_CH4);
            break;
        }
        default : break;
    }
}

/**
 * @brief       关闭步进电机
 * @param       motor_num: 步进电机接口序号
 * @retval      无
 */
void stepper_stop(uint8_t motor_num)
{
    HAL_TIM_Base_Stop_IT(&g_atimx_handle);
    switch(motor_num)
    {
        case STEPPER_MOTOR_1 :
        {
         /* 关闭对应 PWM 通道 */
            HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH1);
            break;
        }
        case STEPPER_MOTOR_2 :
        {
            HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH2);
            break;
        }
        case STEPPER_MOTOR_3 :
        {
            HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
            break;
        }
        case STEPPER_MOTOR_4 :
        {
            HAL_TIM_PWM_Stop(&g_atimx_handle, ATIM_TIMX_PWM_CH4);
            break;
```

```
        }
        default : break;
    }
}
```

主要添加了 HAL_TIM_Base_Start_IT 和 HAL_TIM_Base_Stop_IT 用于开启/关闭定时器以及更新中断，并且在启动旋转函数里添加了 STx_DIR(dir)函数用于设置电机旋转方向。

在 main 函数里面编写如下代码：

```
int main(void)
{
    uint8_t key,t;
    int angle = 0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    /* 168 000 000 / 168 = 1000 000 1Mhz 的计数频率，计数 1K 次为 1ms */
    stepper_init(1000 - 1, 168 - 1);

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"Stepper Test",g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:angle ++",g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:angle --",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:star",g_point_color);
    lcd_show_string(10,90,200,16,16,"Stepper ID: 1",g_point_color);
    printf("KEY0 往正方向增加旋转角度，每次加 90°\r\n");
    printf("KEY1 往反方向增加旋转角度，每次加 90°\r\n");
    printf("KEY2 开启电机\r\n");

    while (1)
    {
        key = key_scan(0);
        if(key == KEY0_PRES)                    /* 按下 KEY0 增加旋转角度 */
        {
            if(g_run_flag == 0)
            {
                angle += 90;
                if(angle >= 0)
                {
                    g_stepper.angle = angle;
                    g_stepper.dir = CW;
                }else
                {
                    g_stepper.angle = -angle;
                    g_stepper.dir = CCW;
                }
                printf("旋转的角度为：%d\r\n",angle);
            }
        }
        if(key == KEY1_PRES)                    /* 按下 KEY1 增加旋转角度 */
        {
            if(g_run_flag == 0)
            {
               angle -= 90;
                if(angle >= 0)
                {
                    g_stepper.angle = angle;
                    g_stepper.dir = CW;
                }else
```

```
            {
                g_stepper.angle = -angle;
                g_stepper.dir = CCW;
            }
            printf("旋转的角度为：%d\r\n",angle);
        }
    }
    if(key == KEY2_PRES)                            /* 按下 KEY2 关闭电机 */
    {
        if(g_run_flag == 0)
        {
            angle = 0;                              /* 角度清 0，以便下次设置 */
            stepper_set_angle(g_stepper.angle, g_stepper.dir,
                              STEPPER_MOTOR_1);     /* 开启旋转 */
            printf("开启旋转\r\n");
        }
    }
    t++;
    if(t % 20 == 0)
    {
        LED0_TOGGLE();                              /*LED0(红灯) 翻转*/
    }
    delay_ms(10);
  }
}
```

主要就是调用 stepper_init 函数初始化步进电机，然后串口打印提示信息，LCD 显示提示信息，我们需要先把步进电机接到驱动器在接到步进电机接口一，通过按键 KEY0 设置旋转角度正向加 90°，按键 KEY1 设置旋转角度反向加 90°，KEY2 调用 stepper_set_angle 将需要转动的角度转换成脉冲数并开启更新中断以及脉冲输出，当旋转到指定位置关闭脉冲输出及中断，即达到了定位的目的。接着看下第二种驱动方式。

2、翻转模式驱动步进电机

同样的我们只讲解核心代码，详细的源码请大家参考光盘中本实验对应源码。此代码在驱动旋转比较输出翻转模式方式上添加而来。源码中有明确的注释。和基本旋转驱动相同的部分就不再赘述了，只讲解不同部分。来看下定时器的驱动 stepper_tim.c 里边的输出中断回调函数：

```
/**
  * @brief  定时器比较中断
  * @param  htim：定时器句柄指针
  * @note   无
  * @retval 无
  */
uint8_t i = 0;
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim)
{
    i++;
    if(i % 2 == 0)
    {
        i = 0;
        g_run_flag = 1;
        g_stepper.pulse_count --;               /* 每一个完整的脉冲就-- */
        if(g_stepper.dir == CW)                 /* 正转 */
        {
            g_stepper.add_pulse_count++;        /* 绝对位置++ */
        }else
        {
            g_stepper.add_pulse_count--;        /* 绝对位置-- */
        }
       /*当脉冲数等于 0 的时候 代表需要发送的脉冲个数已完成，停止定时器输出 */
        if(g_stepper.pulse_count <= 0)
```

```
        {
            printf("累计旋转的角度:%d\r\n",
                  (int)(g_stepper.add_pulse_count*MAX_STEP_ANGLE));
            stepper_stop(STEPPER_MOTOR_1);    /* 停止接口一输出 */
            g_run_flag = 0;
        }
    }

    /*获取当前计数*/
    g_count_val = __HAL_TIM_GET_COUNTER(&g_atimx_handle);

    /*设置比较数值*/
    if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
    __HAL_TIM_SET_COMPARE(&g_atimx_handle, ATIM_TIMX_PWM_CH1, (g_count_val +
                          g_ccr_val)%0XFFFF);

    if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2)
    __HAL_TIM_SET_COMPARE(&g_atimx_handle, ATIM_TIMX_PWM_CH2, (g_count_val +
                          g_ccr_val)%0XFFFF);

    if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_3)
    __HAL_TIM_SET_COMPARE(&g_atimx_handle, ATIM_TIMX_PWM_CH3, (g_count_val +
                          g_ccr_val)%0XFFFF);

    if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_4)
    __HAL_TIM_SET_COMPARE(&g_atimx_handle, ATIM_TIMX_PWM_CH4, (g_count_val +
                          g_ccr_val)%0XFFFF);
}
```

在中断回调函数里边同样的也是进行脉冲计数，但是注意的点是输出中断需要翻转两次才是一个完整脉冲，也就是进入两次中断输出一个完整脉冲，所以当每次进入中断 i++，当 i 等于 2 了代表进入两次中断了，此时重新设置为 0；使脉冲个数减一，直到减到 0 停止脉冲输出。接着看下 stepper_motor.c 源码里边也是添加了角度转脉冲数函数：

```
/**
 * @brief       将需要转动的角度转换成脉冲数
 * @param       angle    : 需要转动的角度值
 * @param       dir      : 旋转方向
 * @param       motor_num: 步进电机接口序号
 * @retval      无
 */
void stepper_set_angle(uint16_t angle, uint8_t dir,uint8_t motor_num)
{
    g_stepper.pulse_count = angle / MAX_STEP_ANGLE;
    if(g_stepper.pulse_count == 0)
    {
        stepper_stop(motor_num);
    }
    else stepper_star(motor_num,dir);
}
```

该函数有三个入口参数分别是：转动的角度值、旋转方向以及控制的步进电机接口序号，我这里用到的是步进电机接口一，然后我们会根据所需转动的角度值除以步距角就可以算出需要几个脉冲才可达到指定角度，之后通过判断脉冲个数决定开启还是关闭电机，旋转角度的正负号决定旋转方向。main 函数部分和 pwm 驱动模式完全一模一样的这里就不贴出来了。

15.7.3 下载验证

两种驱动模式的下载现象一致，所以这里统一进行讲解。将代码下载进去开发板，可以看到 LED0 每隔 200ms 左右闪烁一次，代表程序正在运行。此时复位开发板可以在串口打印提示信息如图 15.7.3.1 所示，且 LCD 显示当前所控制的步进电机接口号，当我们按下按键 KEY0 设置旋转角度正向加 90°，按键 KEY1 设置旋转角度反向加 90°，KEY2 开启电机旋转。下图为复位后按下两次 KEY0 再接着按下 KEY2 打印信息，步进电机就会顺时针旋转 180°之

后停止。

图 15.7.3.1 串口打印结果

第 16 章 步进电机梯形加减速

在前面的章节，我们已经学习了步进电机的基础控制以及定位控制，相信你现在已经对步进电机的控制有了基本的认识，但是这远远不够，步进电机在开环控制的情况下也具有高精度，无积累误差等优点，被广泛应用于数控系统中，但是我们要实现步进电机的快速准确的定位，就要防止失步和过冲的现象发生，并以最快的速度到达指定的位置，所以说步进电机的加减速控制是数控系统的关键技术之一。那么今天我们就来学习下步进电机梯形加减速。

本章分为如下几个小节：

16.1 梯形加减速的简介
16.2 梯形加减速的原理分析
16.3 梯形加减速整体流程
16.4 梯形加减速的控制实现

16.1 梯形加减速的简介

步进电机因其无需反馈就能对位置和速度进行控制而在工业自动化设备中的应用极为广泛，如下图所示，假设该装置使用步进电机实现物体 X 的移动，系统要求从 A 点出发，到 B 点停止，移动的时间越短越好且系统稳定。

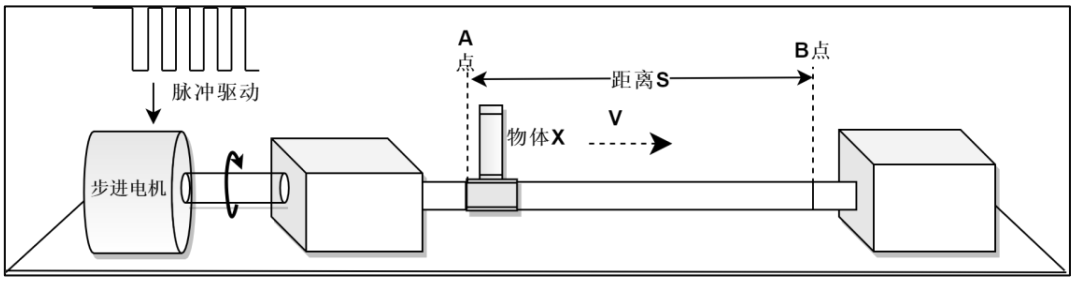


图 16.1.1 加减速实例图解

根据步进电机的特性，最大程度加大电机转速（提高脉冲频率），则到达 B 点的时间就越短，但是如果施加脉冲频率太高，超过了步进电机的最小启动频率，则会造成电机内部的反向电动势的阻尼作用，转子与定子之间的磁反应将跟随不上电信号的变化，导致堵转或者丢步，滑块连动都没动，非常尴尬！

所以要求在电机启动时需要一个低速，但为了实现系统要求，在启动之后慢慢的升高速度，实现一个加速的过程，那如果在达到终点前一直在加速，就又有另外一个问题，到达终点时速度太快就会导致刹不住，出现过冲现象，为了防止这个现象我们需要在达到终点前的一段距离提前减速，使得滑块到达终点时速度刚好为 0，即从最高速度一直减至停止。如果在加速的过程中，从启动速度到目标速度的加减速是以一定的比例进行加速/减速，则可以将这个过程描述为如下图所示：

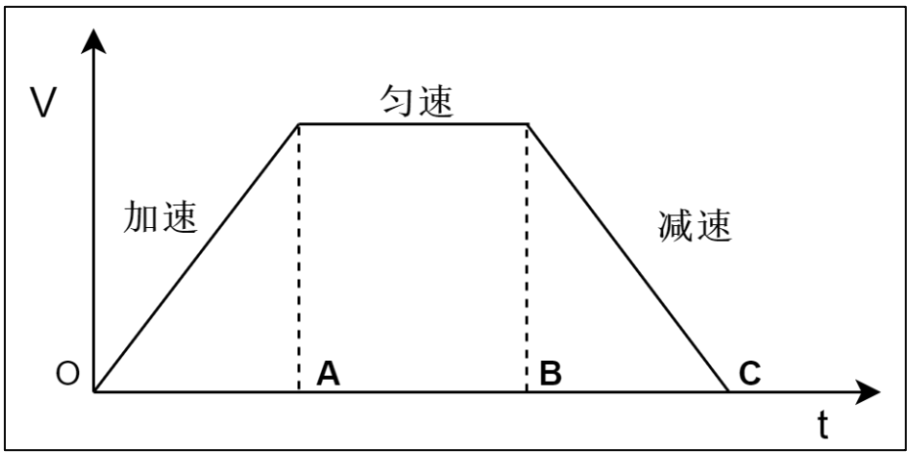


图 16.1.2 加减速模型解析

由图 16.1.2 可知，整个运动过程将滑块的速度变化一共分为三个阶段，OA 段其实就是滑块的加速部分、AB 则是匀速部分，BC 则是减速部分。因其图像呈现梯形，所以我们把这种运动变化称之为梯形加减速。

- 在 OA 加速过程中，由低于步进电机的启动频率开始启动(模型中由 0 启动)，以固定的加速度增加速度到目标值;
- 在 AB 匀速过程中，以最大速度匀速运动;
- 在 BC 减速部分中，以加速度不变的速度递减到 0;

这种算法是一种在加速过程和减速过程中加速度不变的匀变速控制算法。

16.2 梯形加减速的原理分析

那接着就是我们的重头戏了，如何实现梯形加减速算法？步进电机的速度由脉冲频率决定，在匀速运动的过程中，脉冲的频率是固定不变的，根据步进电机特性，每给一个脉冲，电机就会步进固定的距离（将角度转换为距离），知道了 A 点到 B 的距离，就可以知道步进电机需要走多少步，然后让控制器输出所需个数且固定频率的脉冲就行。

在加减速段的过程中，速度是不断的改变的，所以需改变脉冲的频率，让电机在加速时给高频率的脉冲，减速时给低频率脉冲。那具体下一个变化的脉冲的周期是多少？虽然从 A 点到 B 点所需要的总步数（脉冲）不变，但在加速阶段时需要多少步数？多少步之后需要减速？减速阶段要步进多少步？所以接下来我们一个个解答这些疑问，前方高能，请做好心理准备！

16.2.1 确定未知量和已知量

要解答上述问题，我们需要先明确要求解的**未知量**有哪些？

（1）**加速阶段时需要多少脉冲数**？使用 n1 进行表示

（2）**减速阶段要步进多少脉冲数**？使用 n2 进行表示

（3）**下一个变化的脉冲的周期是多少**？使用 T 进行表示

因为要实现加减速效果，所以输送给电机的脉冲的周期是变化的，只要确定下一个脉冲的周期，并且确定加速阶段和减速阶段需要提供多少个脉冲，然后让控制器一个一个把脉冲送出去即可。如下图所示：

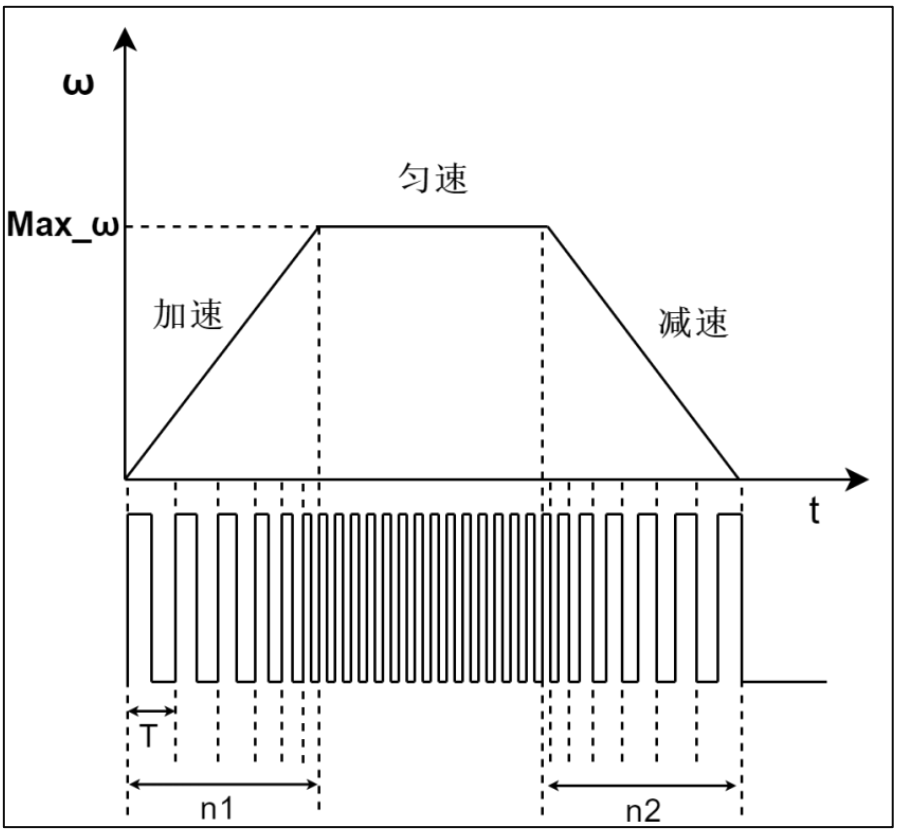


图 16.2.1.1 梯形加减速各阶段和脉冲的关系图解

在进行加减速前我们必须要确定的参数量有如下：

（1）电机需要步进的距离，使用 step 表示

（2）加速度大小，使用 accel 表示

（3）减速度大小，使用 decel 表示

（4）最大速度，决定电机匀速时的速度值，用 Max_ω 表示

这些参数在步进电机做加减速之前就要求用户来设置好，也就是我们的已知量。根据设置的参数不同，系统将有如下两种运动情况：

16.2.2 三角形运动变化的加减速阶段所需脉冲数

第一种设置情况：最大速度设置偏大，而加速度设置偏小，导致电机步进了很多步数依然没有达到最大速度，但此时距离终点 Step 可走的步速越来越少，再不减速将无法在终点时使速度降为 0。所以这种情况，其加速模型并不呈现梯形，加速不到最大速度将要被迫降速。

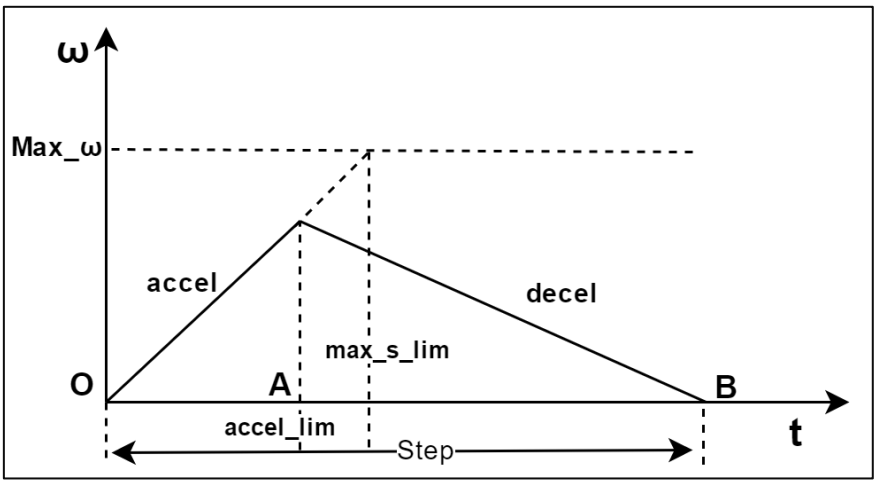


图 16.2.2.1 梯形加减速运动场景 1

运动变化图形类似于三角形，上图全部有以下几个参数：

Max_w ：用户设置的最大速度；
accel ：加速度大小；
decel ：减速度大小；
step ：总步数；

这些都是已知量，不需要求解，接着看下需要求解的变量：

max_s_lim ：速度从 0 加速到 Max_w 所需要的步数，也就是与最大速度的交点；
acc_lim ：加速度曲线与减速度曲线的相交点；
decel ：实际减速的步数；

此时 n1 = accel_lim，加速段延长线做数学模型，得出只要满足：**acc_lim ≤ max_s_lim，运动变化状态就呈现三角形，**首先我们通过求解 accel_lim 则可获知加速段的步数，不难看出，这里有一个等式条件：加速段的末速度 ω1 和开始减速的速度 ω2 是相等的，所以：

$$\omega_1 = \omega_2 \quad \text{[式 1-1]}$$

$$\dot{\omega}_1 * t_1 = \dot{\omega}_2 * t_2 \quad \text{[式 1-2]}$$

此时的 t1 为加速段步进消耗的时间，t2 则是减速段的时间，从上面的模型中可知，梯形加减速运动规律符合物理学中匀加速运动，既然我们的疑问是关于加速过程中走过的距离问题，想必大家都已经想到了经典的位移公式，根据这个公式我们就获知时间 t，然后将时间 t 转换为我们想要的步数

$$S = V_0 t - \frac{1}{2} a t^2 \quad \text{[式 1-3]}$$

没错，位移 S 就是步进电机的步进距离，等于步距角（a）和脉冲个数(n)的乘积，在初速度为 0 则有以下公式，其中加速度 a 为角加速度 w：

$$n\alpha = \frac{1}{2}\dot{\omega} t^2 \quad \text{[式 1-4]}$$

简单变换一下，将时间 t 转化为已知量可得：

$$t_n = \sqrt{\frac{2\alpha * n}{\dot{\omega}}} \quad \text{[式 1-5]}$$

结合[式 1-2]和[式 1-5]，即可得出我们迫切想要求解的加速段的脉冲数 **n1**：

$$\dot{\omega}_1 * n_1 = \dot{\omega}_2 * n_2 \quad \text{[式 1-6]}$$

因为 n1 和 n2 都是未知的，但是 n1+n2=step 是可知的，所以这里还需要做一个技巧性处理，

$$\dot{\omega}_1 * n_1 = \dot{\omega}_2 * n_2$$
$$\dot{\omega}_1 * n_1 + \dot{\omega}_2 * n_1 = \dot{\omega}_2 * n_2 + \dot{\omega}_2 * n_1$$

$$(\dot{\omega}_1 + \dot{\omega}_2) * n_1 = \dot{\omega}_2 * (n_2 + n_1)$$

$$n_1 = \frac{\dot{\omega}_2 * (n_2 + n_1)}{(\dot{\omega}_1 + \dot{\omega}_2)} \quad \text{[式 1-7]}$$

由上求解出了加速段脉冲数 n1，已知总的脉冲数 step 不变，则可知减速段 n2

$$n_2 = -(Step - n_1) \quad \text{[式 1-8]}$$

其中负号是因为减速阶段的运动方向和减速度的方向是相反的。这就是第一种情况下的加减速阶段的脉冲个数求解方法，接着看下第二种情况。

16.2.3 梯形运动变化的加减速阶段所需脉冲数

第二种设置情况：最大速度设置偏小，而加速度设置偏大，步进电机很快就加速到了设置的最大速度，距离设置的终点 max_w 还有剩余很长的步数可以走，此时，电机以最大速度匀速步进，直到满足减速条件才开始减速，并刚好使得速度到终点时降至 0。

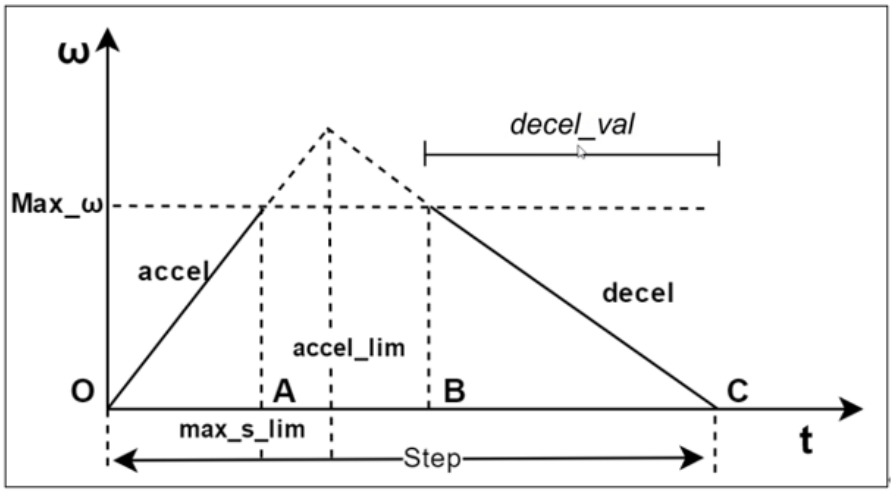


图 16.2.3.1 梯形加减速运动场景 2

由上图得出只要满足：**acc_lim >max_s_lim，运动变化状态就呈现梯形，**那么此时加速段步数 n1=max_s_lim，由最大速度已知我们可以直接得到以下等式：

$$n_1 = \frac{Max_\omega^2}{2 * \alpha * \dot{\omega}} \quad \text{[式 1-9]}$$

我们知道了加速段的步数，也知道总步数，但是减速度段不单单只总步数减去加速度段，因为，中间需要一个匀速步进段，但是从图中可以看出，减速前的速度等于最大速度的条件是依然不变的，我们同样可以使用前面 1--6 的公式，求解减速段的步数。

$$\dot{\omega}_1 * n_1 = \dot{\omega}_2 * n_2$$

$$decel_val = n2 = -max_s_lim \times \frac{accel}{decel} \quad \text{[式 1-10]}$$

注意：decel_val 是减速段的步数，而不是开始减速的步数，开始减速的步数等于总步数减掉减速段的步数（实际减速段步数为负值，所以直接将总步数与减速段步数相加即可）。

解决了加速和减速阶段时需要多少脉冲数的问题，接着还需要求出 accel_lim 的大小，以便在程序中比较 accel_lim 与 max_s_lim 的大小判断当前运动状态属于哪种情况。同样的交点满足速度相同所以：

$$accel_lim = \frac{\dot{\omega}_2 (n_2 + n_1)}{(\dot{\omega}_1 + \dot{\omega}_2)} = \frac{Step * decel}{accel + decel} \quad \text{[式 1-11]}$$

求解完之后，接下来我们要还有一个大麻烦，求解加减速段以及匀速段的脉冲间隔时

间。

16.2.4 脉冲时间间隔的精确计算

根据我们所知道的，步进电机速度的快慢是由脉冲频率的高低决定的，而脉冲频率的高低是由定时器的计算频率 ft 和计数值 cn 决定的，在计数频率不变的情况下，脉冲周期只由定时器计数值决定，即在加速段/减速度的时候改变 cn 的值提升/降低步进电机速度，而在匀速时，保持最高速度对应下的 cn 值即可，

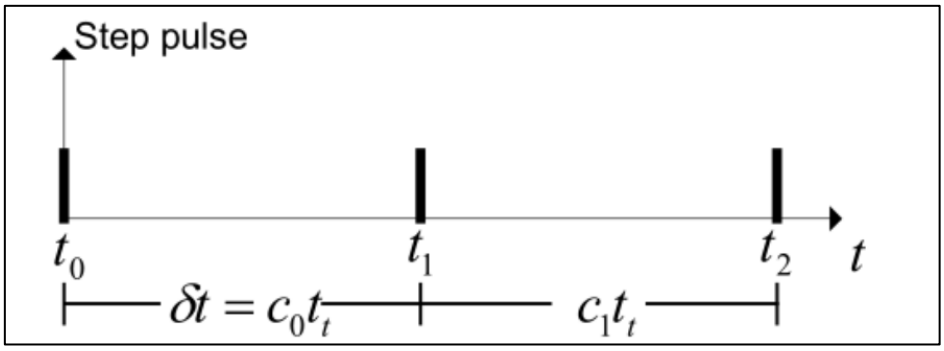


图 16.2.4 脉冲间隔时间示意图

- t1 ：发送第二个脉冲的时刻
- t2 ：发送第三个脉冲的时刻
- &t ：两个脉冲之间的间隔时间
- c0 ：定时器从 t0~t1 时刻的定时器计数值
- c1 ：定时器从 t1~t2 时刻的定时器计数值
- tt ：定时器计数周期

以 stm32 的高级定时器 8 为例，定时器 8 的时钟频率为 168MHZ，如果将分频值设置为 83，那么定时器的计数频率则为: ft=168/(83+1)=2MHZ，每过 1/ft=1/2Mhz=0.5us 则 cn 增加 1，如果 Cn=10，则输出的脉冲周期则是 T=Cn*1/ft=5us，可知 Cn 越小，脉冲周期越小，电机速度越快，反之则越慢。

在求解之前先了解下我们已知的一些基本公式：

$$\delta t = Ct_t = C/f_t \quad \text{[式 1-12]}$$

$$\alpha = \frac{2\pi}{spr}[rad] \quad \text{[式 1-13]}$$

$$\theta = n\alpha[rad] \quad \text{[式 1-14]}$$

$$w = \frac{\alpha}{\delta t}[rad/sec] = \frac{\alpha f_t}{c}[rad/sec] \quad \text{[式 1-15]}$$

其中：

ft ：定时器的计数频率；
α ：步距角；
Spr ：步进电机旋转一圈脉冲数， 电机相关常数；
θ ：位置；
ω ：速度；
n ：脉冲数；
rad ：弧度单位，1rad/sec = 60/2π ≈ 9.55rpm；
rpm：转每分钟（rounds per minutes）， 常用转速单位；
接着我们继续看下梯形加减速的数学模型，如下图：

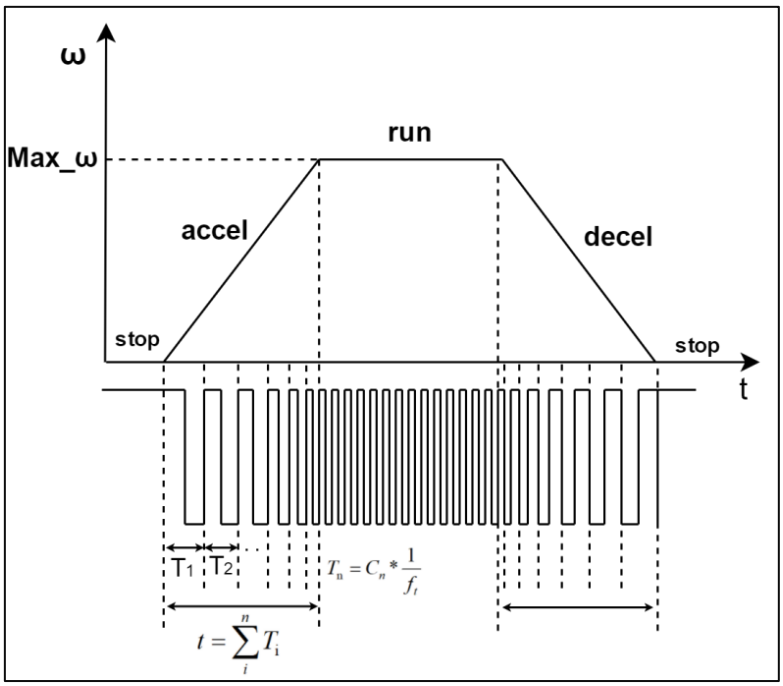


图 16.2.4.1 梯形加减速数学模型

结合上述基本公式与下图的数学模型，我们可以看出在加速阶段，步进电机做的匀加速运动，并且初速度为 0，所以可得以下关系式：

$$S = v0t + \frac{1}{2}at^2 = \frac{1}{2}\dot{w}t_n^2 \quad \text{[式 1-16]}$$

其中：S 表示位移；$\dot{w}$表示加速度，tn 表示时间点，整理得：

$$t_n = \sqrt{\frac{2a}{\dot{w}}} \quad \text{[式 1-17]}$$

可以将 tn 理解为某个时间点，结合基础公式[式 1-12]，得：

$$\delta t = c_n t_t = t_n - t_{n-1} \quad \text{[式 1-18]}$$

将[式 1-17]和[式 1-18]整理化简得：

$$c_n t_t = \sqrt{\frac{2a}{\dot{w}}}(\sqrt{n+1} - \sqrt{n}) \quad \text{[式 1-19]}$$

继续化简得 Cn：

$$c_n = \frac{1}{t_t}\sqrt{\frac{2a}{\dot{w}}}(\sqrt{n+1} - \sqrt{n}) \quad \text{[式 1-20]}$$

由此可知，当 n=0 时，代入上式，得到第 1 个脉冲的定时计数器值 C0 为：

$$c_0 = \frac{1}{t_t}\sqrt{\frac{2a}{\dot{w}}} \quad \text{[式 1-21]}$$

再继续把[式 1-21]代入[式 1-20]得到：

$$c_n = c_0(\sqrt{n+1} - \sqrt{n}) \quad \text{[式 1-22]}$$

算到这差不多已经大功告成了，不过从上面的式子可知，定时器计数值需要连续两次计算开方根才能得出结果，而我们面对的现实问题是：一般控制器的计算能力是有限的，连续两次计算开方根在实时运动的过程中比较费时，因此这里还可做一些技巧性处理，帮控制器“减负”，但是带来的运算也会稍微复杂些，当然了，你也可以选择直接使用上面的式子。那么如何“减负”？，我们这里考虑使用多项式来展开来减少运算。**根据泰勒公式的一个特例麦克劳林公式**:

$$\sqrt{1\pm\frac{1}{\mathrm{n}}}=1\pm\frac{1}{2n}-\frac{1}{8n^2}+O\left(\frac{1}{n^3}\right) \quad \text{[式 1-23]}$$

根据上面的式子，可以对 Cn 进行一些转换，以使用简单的计算方法得出 Cn，下面是其推导过程：

$$\frac{C_n}{C_{n-1}}=\frac{C_0\left(\sqrt{n+1}-\sqrt{n}\right)}{C_0\left(\sqrt{n}-\sqrt{n-1}\right)}$$

$$=\frac{C_0\sqrt{n}\left(\sqrt{1+\frac{1}{n}}-1\right)}{C_0\sqrt{n}\left(1-\sqrt{1-\frac{1}{n}}\right)}$$

$$=\frac{1+\frac{1}{2n}-\frac{1}{8n^2}+O\left(\frac{1}{n^3}\right)-1}{1-\left(1-\frac{1}{2n}-\frac{1}{8n^2}+O\left(\frac{1}{n^3}\right)\right)}$$

$$=\frac{\frac{1}{2n}-\frac{1}{8n^2}+O\left(\frac{1}{n^3}\right)}{\frac{1}{2n}+\frac{1}{8n^2}-O\left(\frac{1}{n^3}\right)}$$

$$\approx\frac{4n-1}{4n+1} \quad \text{[式 1-24]}$$

最后的结果显而易见，定时器计数值 Cn 变得更加简洁。

$$C_n=C_{\mathrm{n-1}}\left(\frac{4\mathrm{n}-1}{4n+1}\right)=C_{\mathrm{n-1}}\frac{(4n+1)-2}{4n+1}=C_{\mathrm{n-1}}-\frac{2C_{\mathrm{n-1}}}{4n+1}\text{-} \quad \text{[式 1-25]}$$

实验结果表明，这个公式比连续开两次方的计算方式快许多，

$$C_n=C_0\left(\sqrt{\mathrm{n}+1}-\sqrt{n}\right) \quad \text{原式}$$

$$C_n=C_{\mathrm{n-1}}-\frac{2C_{\mathrm{n-1}}}{4n+1} \quad \text{化简式}$$

但是由于在化简时舍弃了无穷小余项，发现当 n=1 时有 0.4142 的偏差，我们可以将 C0 乘一个系数 0.69 来解决这个误差。接着就是确定下一步定时器计数值的值。

在加速和减速过程中的每一步新的时间间隔必须计算得出。而这个新的脉冲周期的确定需要使用前面费劲心思推导的下个脉冲周期：

$$C_n=C_{\mathrm{n-1}}-\frac{2C_{\mathrm{n-1}}}{4n+1}$$

这里我们将使用 new_step_delay 进行表示下一个脉冲周期 Cn，上一个脉冲周期使用 step_delay 进行表示，而 n 是记录第几个脉冲，使用 accel_count 表示
则有以下表达式：

$$new_step_delay=step_delay-\frac{rest}{4*accel_count+1} \quad \text{[式 1-26]}$$

因为得到的结果 new_step_delay 会包括一个系数和一个余数，为了提高精度余数保留并包含在下一次计算当中如下所示：

$$\mathrm{new}_rest=(2*step_delay+rest)(mod(4*accel_count+1))$$

其中 rest 为余数，首次计算值为 0。new_ rest 就是保存除不尽的余数参与下一次计算。

我们这里求解的是加减速段的 c 值。还需要求解匀速段的 c 值，这个比较简单我们已知最大速度 w，且匀速段的脉冲频率不变，所以可以由[式 1-15]得到以下公式

$$w = \frac{\alpha}{\delta t}[rad/sec] = \frac{\alpha f t}{c}[rad/sec]$$

$$c = \frac{\alpha f_t}{w} \quad [式子\ 1\text{-}27]$$

到这里还有个问题，数值中有可能出现浮点数，而浮点数对我们的运算速度是有影响的，虽然 F4 集成硬件 FPU，但是为了兼容其他低性能的芯片，此算法会将系数放大，但是不影响最终运算结果。

比如我们在计算匀速段的 c 值时，将会在代码中把af_t的值放大 10 倍，即：

$$A_T_x10 = af_t * 10 \quad [式子\ 1\text{-}28]$$

注意为了保持匀速段的 c 值大小不变，最大速度$speed$也会放大 10 倍（假设最大速度设置 100，实际为 10，该值由用户设置），即得到 c 等于：

$$c = \frac{A_T_x10}{speed} \quad [式子\ 1\text{-}29]$$

其他的数值求解同样使用了此方法来解决浮点数带来的影响，在代码中有明确体现！

16.3 梯形加减速整体流程

16.3.1 不同速度段的处理方法

通过上面的论述我们的思路大概已经清晰，接下来就是软件设计层面，其实就是使用定时器产生脉冲，并按梯形规律加速即可，使用定时器中断产生步脉冲并且只有在步进电机移动时进入。根据我们前面的长篇大论概述，需要在这个中断中处理四个不同速度的状态，分别是 stop——accel——run——decel——stop， 如图 16.3.1.1 所示。

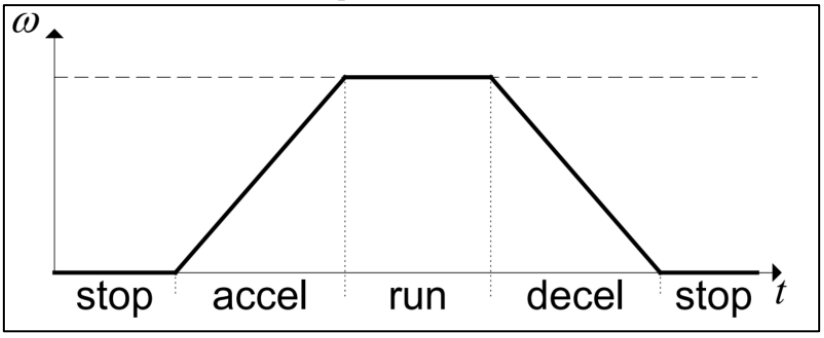


图 16.3.1.1 梯形加减速时间-速度关系

每一个速度变化的阶段的划分是前面计算得出的对应脉冲个数，当脉冲个数到达限制则启动切换即可，这种操作可以通过我们熟悉的状态机在定时器中断中实现。如图 16.3.1.2 所示。STOP 为停止状态，ACCEL 为加速状态，RUN 为匀速状态(对应最大速度 speed)，DECEL 为减速状态。

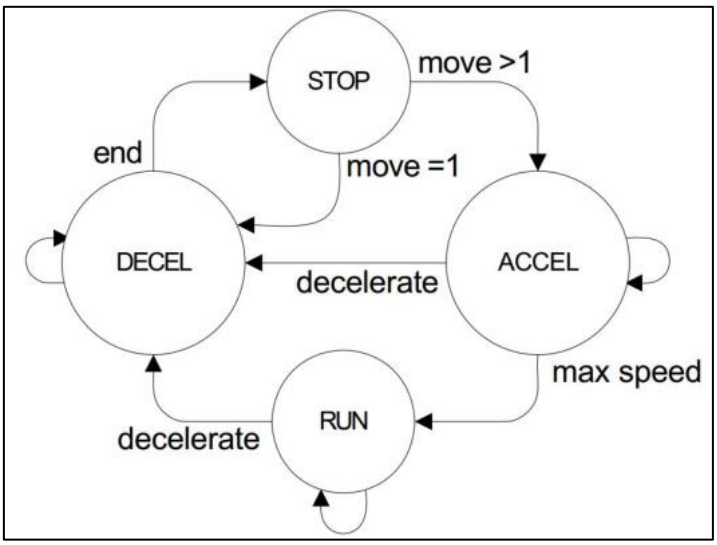


图 16.3.1.2 状态表切换

简单的解析一下：当应用程序启动或者步进电机停止状态机处于 STOP 状态。当输入移动步数建立计算完成，一个新状态被置位同时定时器的中断被使能。当运行的步数超过 1 步状态机进入 ACCEL 状态，如果只移动 1 步，那么状态直接变为 DECEL，因为只移动一步是不需要加速的。当状态变为 ACCEL，应用程序一直加速步进电机达到期望的最大速度，这时状态变为 RUN, 或者减速必须开始，状态变为 DECEL。当状态为 RUN 时候，步进电机会持续保持最大速度 speed 旋转，直到必须开始减速然后把状态改变为 DECEL。它会一直保持 DECEL 状态并一直减速到期望的步数并且速度为 0。然后状态变为 STOP。

16.4 梯形加减速的控制实现

16.4.1 硬件设计

1、例程功能

本实验我们将使用步进电机接口三控制电机实现梯形加减速运动，**驱动器设置：1.0A 电流，8 细分**，也就是旋转一圈所需要 1600 个脉冲数，按键 KEY0 控制开启步进电机梯形加减速运动，KEY1 增加总步数，KEY2 减少总步数，LED0 作为程序运行状态指示灯，闪烁周期 200ms。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器 8： TIM8_CH3：PI7，对应步进电机接口三
方向引脚： DIR3: PB2
脱机引脚： EN3: PF11
4）步进电机
5）步进电机驱动器
6）12-50V 的 DC 电源

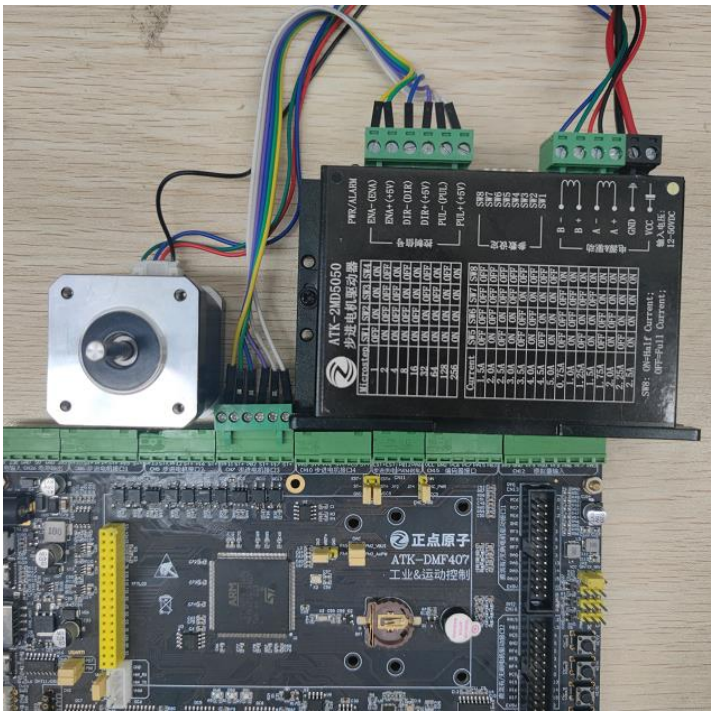


图 16.4.1.1 硬件连接（接口三）

16.4.2 程序设计

16.4.2.1 梯形加减速控制的配置步骤

1）开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出。

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 3 对应 IO 为 PI7 它的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE();                /* 使能定时器 8 */
__HAL_RCC_GPIOI_CLK_ENABLE();               /* 开启 GPIOI 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数。

使用定时器的 PWM 模式功能时，我们调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数来完成对定时器底层以及其输出通道 IO 的初始化，包括：定时器及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

3）设置定时器为比较输出翻转模式，输出比较极性，比较值等参数。

在 HAL 库中，通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为翻转模式，根据需求设置输出比较的极性，设置比较值（控制占空比）等。

本实验我们设置 TIM8 的通道 3 为翻转模式。

4）初始化方向 DIR 引脚，脱机使能 EN 引脚

初始化电机接口三的的方向引脚以及使能引脚。

5）编写步进电机运动状态判断函数

通过对步进电机的步数、加减速度和最大速度的设置来决定步进电机的运动

6）编写中断服务函数

在中断里边对步进电机状态进行决策

16.4.2.2 程序流程图

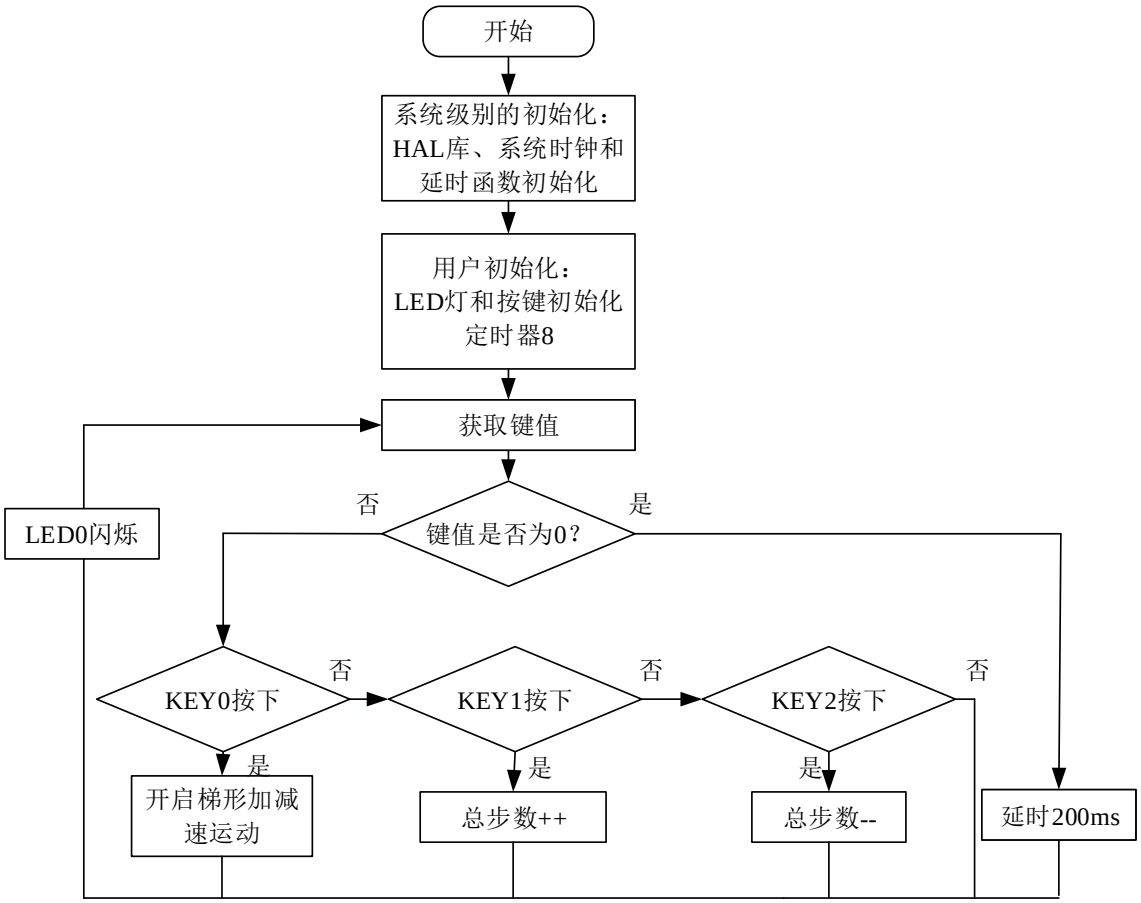


图 16.4.2.2.1 梯形加减速流程图

16.4.2.3 程序解析

不同速度段切换——梯形加减控制参数生成函数解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。在初始化时，需要生成梯形加减速控制参数，也就是我们前面说不同速度段确定的脉冲个数，获取不同速度段的切换条件限制，这个函数的形参需要我们提供前面所说的已知量，Step 是电机需要步进的步数，accel 为加速度，单位是弧度/秒，decel 很明显是减速度，单位也和加速度一致，最后的 speed 就是我们说的最大速度。首先先看下 stepper_motor.h 头文件：

```
#define TIM_FREQ              168000000U              /* 定时器主频 */
#define MAX_STEP_ANGLE        0.225                   /* 最小步距(1.8/MICRO_STEP)*/
#define PAI                   3.1415926               /* 圆周率*/
#define FSPR                  200                     /* 步进电机单圈步数*/
#define MICRO_STEP            8                       /* 步进电机驱动器细分数*/
#define T1_FREQ               (TIM_FREQ/84)           /* 频率 ft 值*/
#define SPR                   (FSPR*MICRO_STEP)       /* 旋转一圈需要的脉冲数*/

// 数学常数

#define ALPHA                 ((float)(2*PAI/SPR))              /* α= 2*pi/spr*/
#define A_T_x10               ((float)(10*ALPHA*T1_FREQ))
#define T1_FREQ_148           ((float)((T1_FREQ*0.69)/10))      /* 0.69 为误差修正值*/
#define A_SQ                  ((float)(2*100000*ALPHA))
#define A_x200                ((float)(200*ALPHA))              /* 2*10*10*a/10 */
```

因为涉及很多计算公式，所以这里为了方便计算，以及代码规范，所以定义了这些宏，主

要分成两部分，第一部分关于步进电机相关参数，第二部分关于计算公式所用到的一些数字常数。接着继续看下 stepper_motor.c:

```
/*
 * @brief       生成梯形运动控制参数
 * @param       step：移动的步数（正数为顺时针，负数为逆时针）.
 * @param       accel  加速度,实际值为 accel*0.1*rad/sec^2  10 倍并且 2 个脉冲算一个完整
                的周期
 * @param       decel  减速度,实际值为 decel*0.1*rad/sec^2
 * @param       speed  最大速度,实际值为 speed*0.1*rad/sec
 * @retval      无
 */
void create_t_ctrl_param(int32_t step, uint32_t accel, uint32_t decel,
uint32_t speed)
{
    __IO uint16_t tim_count;     /*达到最大速度时的步数*/
    __IO uint32_t max_s_lim;     /*必须要开始减速的步数（如果加速没有达到最大速度）*/
    __IO uint32_t accel_lim;
    if(g_motion_sta != STOP)     /*只允许步进电机在停止的时候才继续*/
        return;
    if(step < 0)                 /*步数为负数*/
    {
        g_srd.dir = CCW;         /*逆时针方向旋转*/
        ST3_DIR(CCW);
        step =-step;             /*获取步数绝对值*/
    }
    else
    {
        g_srd.dir = CW;          /*顺时针方向旋转*/
        ST3_DIR(CW);
    }

    if(step == 1)                /*步数为 1*/
    {
        g_srd.accel_count = -1;  /*只移动一步*/
        g_srd.run_state = DECEL; /*减速状态.*/
        g_srd.step_delay = 1000; /*默认速度*/
    }
    else if(step != 0)           /*如果目标运动步数不为 0*/
    {
/*设置最大速度极限，得到 min_delay 用于定时器计数器的值 min_delay = (alpha / t)/ w*/
        g_srd.min_delay = (int32_t)(A_T_x10 /speed); //匀速运行时的计数值
/* 通过计算第一个(c0) 的步进延时来设定加速度，其中 accel 单位为 0.1rad/sec^2
 * step_delay = 1/tt * sqrt(2*alpha/accel)
 * step_delay = ( tfreq*0.69/10 )*10 * sqrt( (2*alpha*100000) /
 * (accel*10) )/100
 */
     g_srd.step_delay = (int32_t)((T1_FREQ_148 * sqrt(A_SQ / accel))/10); //c0
       /*计算多少步之后达到最大速度的限制 max_s_lim = speed^2 / (2*alpha*accel)*/
      max_s_lim = (uint32_t)(speed*speed/(A_x200*accel/10));
       /*达到最大速度小于 0.5 步，我们将四舍五入为 0,必须移动至少一步才能达到想要的速度*/
       if(max_s_lim == 0)
       {
           max_s_lim = 1;
       }
       /*计算多少步之后我们必须开始减速 n1 = (n1+n2)decel / (accel + decel)*/
       accel_lim = (uint32_t)(step*decel/(accel+decel));
       if(accel_lim == 0)        /*不足一步 按一步处理*/
       {
           accel_lim = 1;
       }
       /*加速阶段到不了最大速度就得减速。。。使用限制条件我们可以计算出减速阶段步数*/
       if(accel_lim <= max_s_lim)
```

```
        {
            g_srd.decel_val = accel_lim - step;
        }
        else
        {
            g_srd.decel_val = -(max_s_lim*accel/decel);
        }
        if(g_srd.decel_val == 0)                    /*当只剩下一步则必须减速*/
        {
            g_srd.decel_val = -1;
        }
        g_srd.decel_start = step + g_srd.decel_val; /*计算开始减速时的步数*/

        /*如果一开始 c0 的速度比匀速段速度还大，们就不需要进行加速运动，直接进入匀速*/
        if(g_srd.step_delay <= g_srd.min_delay)
        {
            g_srd.step_delay = g_srd.min_delay;
            g_srd.run_state = RUN;
        }
        else
        {
            g_srd.run_state = ACCEL;
        }
        g_srd.accel_count = 0;                      /*复位加减速计数值*/
    }
    g_motion_sta = 1;                               /*电机为运动状态*/
    ST3_EN(EN_ON);
    tim_count=__HAL_TIM_GET_COUNTER(&g_atimx_handle);
    __HAL_TIM_SET_COMPARE(&g_atimx_handle,ATIM_TIMX_PWM_CH3,tim_count+
     g_srd.step_delay/2);                                   /* 设置定时器比较值*/
    HAL_TIM_OC_Start_IT(&g_atimx_handle,ATIM_TIMX_PWM_CH3);/* 使能定时器通道 */
}
```

这里根据用户设置的总步数 step、加速度 accel、减速度 decel、最大速度 speed，然后计算出 C0 的值，以及 max_s_lim 和 accel_lim，通过这两个值判断当前的运动属于哪种运动变化。并判断总步数的值，如果仅有 1 步，直接做减速运动，否则做匀速或加速运动，之后设置电机状态，设置第一步的比较值，并开启定时器通道输出。接着就来到我们的定时器比较中断回调函数部分，控制脉冲生成以及脉冲频率。

```
/**
  * @brief  定时器比较中断
  * @param  htim：定时器句柄指针
  * @note   无
  * @retval 无
  */
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim)
{
    __IO uint32_t tim_count=0;
    __IO uint32_t tmp = 0;
    uint16_t new_step_delay=0;                  /* 保存新（下）一个延时周期 */
    __IO static uint16_t last_accel_delay=0;    /* 加速过程中最后一次延时*/
    __IO static uint32_t step_count = 0;        /* 总移动步数计数器*/
    __IO static int32_t rest = 0;               /* 记录余数，提高下一步计算的精度*/
    __IO static uint8_t i=0;                    /* 需要进入两次中断才是一个完整脉冲*/
    if(htim->Instance==TIM8)
    {

        tim_count=__HAL_TIM_GET_COUNTER(&g_atimx_handle);
        tmp = tim_count+g_srd.step_delay/2;     /* 需要翻转两次的所以需要除以 2 */
        __HAL_TIM_SET_COMPARE(&g_atimx_handle,ATIM_TIMX_PWM_CH3,tmp);
        i++;                                    /* 定时器中断次数计数值*/
        if(i==2)                                /* 2 次，说明已经输出一个完整脉冲*/
        {
            i=0;                                /* 清零定时器中断次数计数值*/
```

```
        switch(g_srd.run_state)                 /*加减速曲线阶段*/
        {
        case STOP:
            step_count = 0;                     /* 清零步数计数器*/
            rest = 0;                           /* 清零余值*/
            /* 关闭通道*/
            HAL_TIM_OC_Stop_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
            ST3_EN(EN_OFF);
            g_motion_sta = 0;                   /*  电机为停止状态  */
            break;
        case ACCEL:
            g_add_pulse_count++;                /* 只用于记录相对位置 */
            step_count++;                       /* 步数加 1*/
            if(g_srd.dir==CW)
            {
                g_step_position++;              /* 绝对位置加 1，记录绝对位置*/
            }
            else
            {
                g_step_position--;              /* 绝对位置减 1*/
            }
            g_srd.accel_count++;                /* 加速计数值加 1*/
            new_step_delay = g_srd.step_delay - (((2 *g_srd.step_delay) +
            rest)/(4 * g_srd.accel_count + 1));/* 计算新一步脉冲周期(时间间隔) */
            /* 计算余数，下次计算补上余数，减少误差 */
            rest = ((2 * g_srd.step_delay)+rest)%(4 * g_srd.accel_count + 1);
            if(step_count >= g_srd.decel_start)/* 检查是否到了需要减速的步数 */
            {
              /* 加速计数值为减速阶段计数值的初始值 */
               g_srd.accel_count = g_srd.decel_val;
               g_srd.run_state = DECEL;          /* 下个脉冲进入减速阶段 */
            }
            /* 检查是否到达期望的最大速度 */
            else if(new_step_delay <= g_srd.min_delay)
            {
              /* 保存加速过程中最后一次延时（脉冲周期）*/
               last_accel_delay = new_step_delay;
              /* 使用 min_delay（对应最大速度 speed）*/
               new_step_delay = g_srd.min_delay;
               rest = 0;                        /* 清零余值 */
               g_srd.run_state = RUN;           /* 设置为匀速运行状态 */
            }
            break;
        case RUN:
            g_add_pulse_count++;
            step_count++;                       /* 步数加 1 */
            if(g_srd.dir==CW)
            {
                g_step_position++;              /* 绝对位置加 1 */
            }
            else
            {
                g_step_position--;              /* 绝对位置减 1*/
            }
            new_step_delay = g_srd.min_delay;
            if(step_count >= g_srd.decel_start)  /* 需要开始减速 */
            {
                g_srd.accel_count = g_srd.decel_val;  /* 减速步数做为加速计数值 */
              /* 加阶段最后的延时做为减速阶段的起始延时(脉冲周期) */
               new_step_delay = last_accel_delay;
               g_srd.run_state = DECEL;         /* 状态改变为减速 */
            }
```

```
            break;

        case DECEL:
            step_count++;                           /* 步数加 1 */
            g_add_pulse_count++;
            if(g_srd.dir==CW)
            {
                g_step_position++;                  /* 绝对位置加 1 */
            }
            else
            {
                g_step_position--;                  /* 绝对位置减 1 */
            }
            g_srd.accel_count++;
            new_step_delay = g_srd.step_delay - (((2 * g_srd.step_delay) +
            rest)/(4 * g_srd.accel_count + 1));   /*计算新(下)一步脉冲周期*/
            /*计算余数*/
            rest = ((2 * g_srd.step_delay)+rest)%(4 * g_srd.accel_count + 1);
            /* 检查是否为最后一步 */
            if(g_srd.accel_count >= 0)  /*减速步数是否从负值加到 0 是的话减速完成*/
            {
                g_srd.run_state = STOP;
            }
            break;
        }
        g_srd.step_delay = new_step_delay;    /* 为下个(新的)延时(脉冲周期)赋值 */
    }
  }
}
```

为了精确控制步进电机的每一步，需要严格控制每一个脉冲的生成，所以这里设置定时器（TIM8）方式是输出比较模式，意思就是当比较值与计数器数值相等的时候就翻转电平，不是使用 PWM 模式，改变脉冲频率不需要改变 ARR 寄存器（自动重装载值）。注意：该模式需要进入两次中断才会输出一个脉冲。

该函数也有两个重点作用: 1.设置定时器比较值；2.计算下个定时器比较值。该函数先调用_HAL_TIM_GET_IT_SOURCE 确保是发生通道比较中断，然后清除该中断标志。然后调用_HAL_TIM_GET_COUNTER 函数获取当前定时器计数值，保存在 tim_ count 中，然后调用_HAL_TIM_ SET_COMPARE 函数设置通道的比较值。剩下就是对每个状态下的一个运动判断以及计算，公式都在前面有推导这里就不重复了。

在 main 函数里面编写如下代码：

```
/*
    加速度和减速度选取一般根据实际需要，值越大速度变化越快，加减速阶段比较抖动
    所以加速度和减速度值一般是在实际应用中多尝试出来的 结果
*/
__IO uint32_t g_set_speed  = 2000;  /*速度 单位为 0.1rad/sec */
__IO uint32_t g_step_accel = 25;    /*加速度 单位为 0.1rad/sec^2    0.025*/
__IO uint32_t g_step_decel = 20;    /*减速度 单位为 0.1rad/sec^2     0.025*/
__IO uint16_t g_step_angle=1;       /*设置的步数*/
extern __IO uint32_t g_add_pulse_count;        /*脉冲个数累计*/

int main(void)
{
    uint8_t key,t;
    char buf[32];

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
```

```
    lcd_init();                                     /* 初始化 LCD */
    stepper_init(0xFFFF, 84 - 1);
    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"Stepper Motor Test",g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:Start", g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:Set++",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:Set--",g_point_color);
    printf("KEY0 开启梯形加减速\r\n");
    printf("KEY1 增加步数\r\n");
    printf("KEY2 减少步数\r\n");

    while (1)
    {
        t++;
        if(t % 20 == 0)
        {
            sprintf(buf,"Set_Aangle:%d     ",g_step_angle); /* 旋转位置（角度）*/
            lcd_show_string(10,90,200,16,16,buf,g_point_color);
           /*累计旋转的角度*/
            sprintf(buf,"Add_Aangle:%.2f    ",g_add_pulse_count*0.225);
            lcd_show_string(10,110,200,16,16,buf,g_point_color);
            LED0_TOGGLE();                          /*LED0(红灯) 翻转*/
        }
        delay_ms(10);
        key = key_scan(0);
        if(key == KEY0_PRES)                        /* 开启梯形加减速 */
        {
            Create_T_Ctrl_Param(SPR*g_step_angle, g_step_accel, g_step_decel,
                                g_set_speed);
            g_add_pulse_count=0;
        }
        if(key == KEY1_PRES)                        /* 增加步数 */
        {
            g_step_angle=g_step_angle+1;
            if(g_step_angle>100)  g_step_angle=1;
        }
        if(key == KEY2_PRES)                        /* 减少步数 */
        {
            g_step_angle=g_step_angle-1;
            if(g_step_angle<1)  g_step_angle=100;
        }
    }
}
```

首先我们先提前设置好总步数：g_step_angle、最大速度：g_set_speed、加速度：g_step_accel、减速度：g_step_decel，之后在 main.c 初始化定时器以及步进电机接口等，在串口以及 LCD 显示提示信息，当按下按键 KEY0 控制开启步进电机做梯形加减速运动，注意在入口参数总步数会乘以步进电机旋转一圈所需的脉冲数 SPR=1600，KEY1 增加总步数，KEY2 减少总步数；LED0 周期闪烁提示程序正在运行。

16.4.3 下载验证

下载代码后，可以看到 LED0 每隔 200ms 左右闪烁一次，LCD 及串口都会显示提示信息，如下图，根据屏幕和串口打印信息提示，点击按键 0 即可使步进电机做梯形加减速运动，如果现象不明显，请通过按键 KEY1 增加总步数，当步数足够多，可以明显感觉到步进电机由缓慢加速，再到最大速度，最后缓慢减速直到停止。

图 16.4.3.1 串口打印提示信息

第 17 章 步进电机 S 形加减速

上一章学习的梯形加减速算法其特点是：算法简便，占用时少、响应快、效率高，实现方便。但匀加速和匀减速阶段不符合步进电机速度变化规律，在变速和匀速转折点不能平滑过渡。启动、停止、高速运动段会产生较大的冲击和振动及噪音所以这种算法主要应用在对升降速过程要求不高的场合，如简单的定长送料。也就是说在一些精密的场合，梯形加减速算法并不适用。而是用到我们本章所学习的内容：S 形加减速算法！

本章分为如下几个小节：

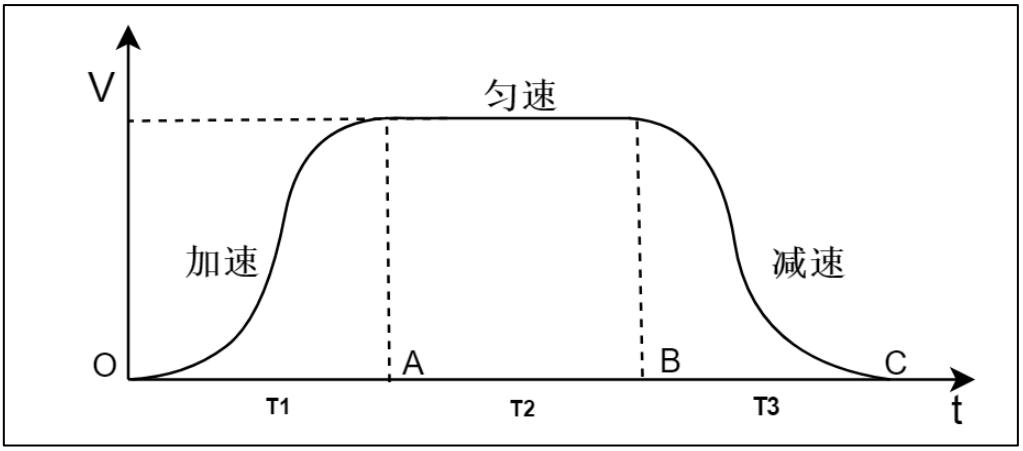
17.1 S 形曲线加减速的简介

17.2 S 形曲线加减速的的原理分析

17.3 S 型曲线加减速的控制实现

17.1 S 形曲线加减速的简介

还是以梯形加减速章节提到的模型为例，如果滑块从启动速度到目标速度的加减速不是以固定的比例进行加速/减速，而在加减速的变化过程中速度曲线呈现一个英文字母“S”形的，我们称之为 S 形加减速算法。则上述将这个过程描述为如下图所示，



17.1.1 S 曲线加减速模型

可以获知 OA 段其实就是滑块的加速部分、AB 则是匀速部分，BC 则是减速部分。

- 在 OA 加速过程中，速度刚开始是缓慢增加，后来增加得越来越快，而在中点时刻，增加又有所放慢，但依然继续增加逼近设定的速度。实际这一阶段又分成了三个阶段
- 在 AB 匀速过程中，加速到设定速度之后，以设定速度匀速步进;
- 在 BC 减速部分中，以设定的速度开始按照加速度段的变化规律做减速变化，直到速度降至 0 后停止。

前面我们有提到梯形加减速的缺点，梯形加减速在启动、停止和高速运动的过程中会产生很大的冲击力振动和噪声，所以多数会应用于简单的定长送料的应用场合中，例如常见的 3D 打印机使用的就是梯形加减速算法；但是相比较 S 形加减速在启动停止以及高速运动时的速度变化的比较慢，导致冲击力噪音就很小，但这也决定了他在启动停止时需要较长的时间，所以多数适用于精密的工件搬运与建造。

17.2 S 形曲线加减速的的原理分析

实际上要实现 S 型可以采用的方法有很多，在传统的 S 形曲线加减速算法中,它包括七个运动阶段：加加速阶段，恒加速阶段，减加速阶段，恒速阶段，加减速阶段，恒定减速阶段和减减速阶段。

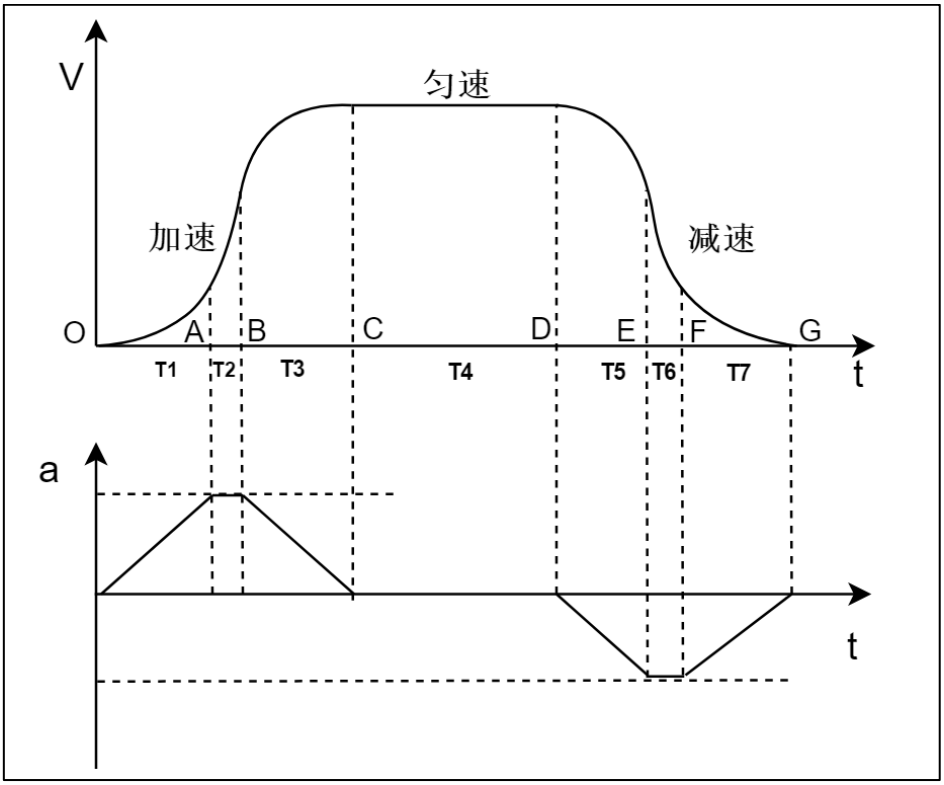


图 17.2.1 S 曲线加减速七段式模型

虽然该算法具有平稳、精度高的特点，但是也可以感受到该算法的参数复杂，大大降低到了工作效率，而且对硬件的要求也比较高。因此，本文介绍的是一种五段 S 曲线加减速算法并通过查表的方式进行实现，这是一种简单但具有实时性和高精度的加减速控制算法，非常适合资源紧凑的小型嵌入式系统。

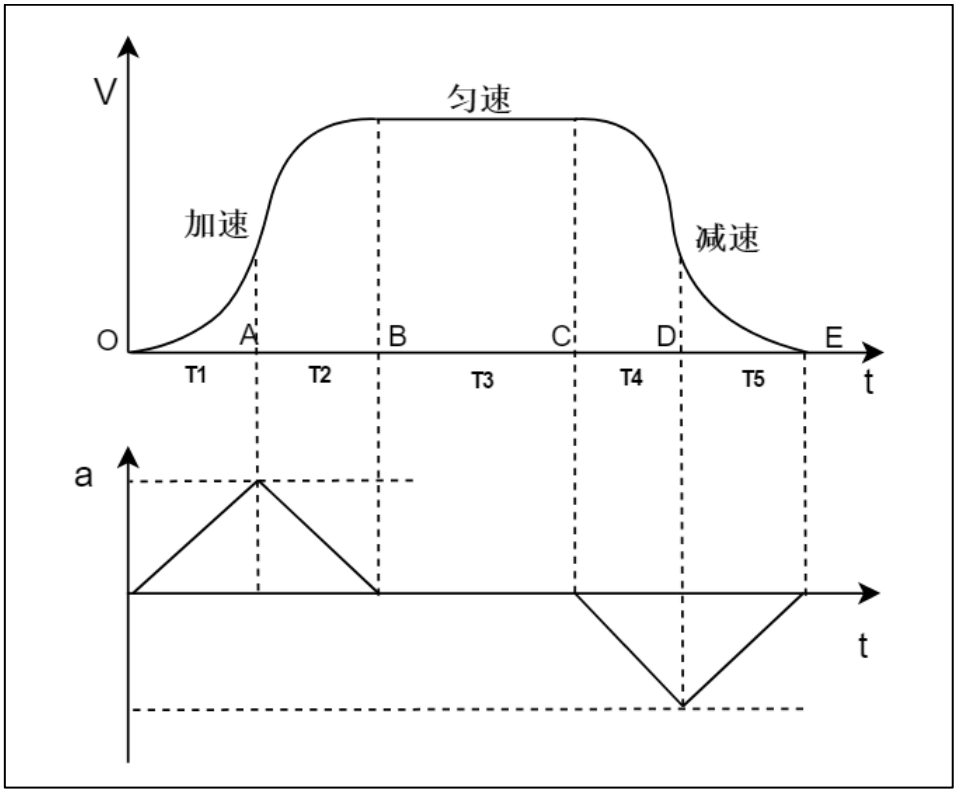


图 17.2.2　S 曲线加减速五段式模型

在图 17.2.1 中 T1、T2、T3、T4、T5 分别代表加加速阶段、减加速阶段、恒速阶段、加减速阶段和减减速阶段的时间。为了使该开始点和结束点的加加速度在减加速阶段为零，加加速阶段的时间必须等于减加速阶段的时间，即 T1=T2，T4=T5=T。

以加速段为例，第一分为加加速段，这部分是加速度 a 从 0 增大到最大值 Amax，第二部分是减加速度段，Amax 从最大值减少至 0（注意是加速度的变化！在加速度段，速度是一直在增加的！）也就是说两部的划分是看加速度 a 的最大值,减速度段也同理，只不过加速度是相反变化的。

17.2.1 确定已知量和未知量

S 形加减速我们需要求解的未知量如下：

（1）加速阶段时的加加速度段需要多少脉冲数？使用 n1 进行表示

（2）加速阶段时的减加速度段需要多少脉冲数？使用 n2 进行表示

（3）下一个变化的脉冲的周期是多少？使用 Cn 进行表示

同梯形算法一样，无论速度怎么变化，给定的距离是不变的，只要知道加速度段的距离和减速段的距离，确定加速阶段和减速阶段需要提供多少个脉冲，然后让控制器一个一个把脉冲送出去即可。但是这里需要说明的是，因为本文采用五段 S 曲线加减速算法模型使用对称结构，**所以当我们知道了加速度段距离求解算法，同理就知道了减速度距离的求解算法，它俩算法一样，只是速度排序为倒序关系。所以这里不用另外求解减速度段的距离，这也是五段式的优点**，如下图所示：

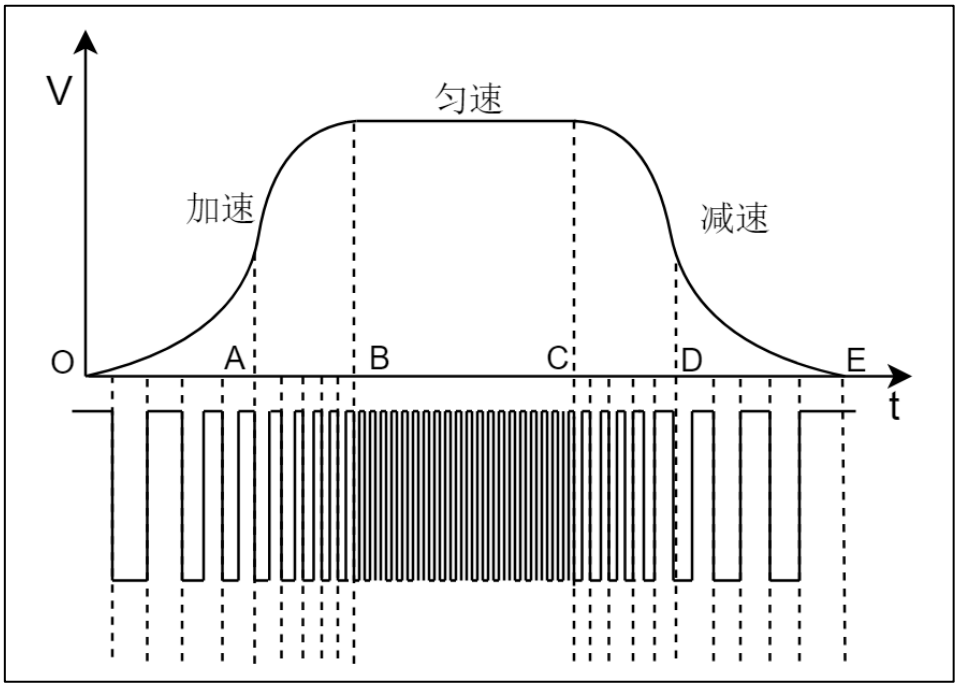


图 17.2.1.1 S 曲线加减速各阶段和脉冲的关系图解

在进行 S 形加减速前我们必须要确定的目标量有如下：

（1）电机需要步进的距离，即用户必须要明确需要电机步进的步数，使用 step 表示

（2）加速度所需时间，使用 ts1 表示

（3）减速度所需时间，使用 ts2 表示

（4）最大速度大小，决定电机匀速时的速度值，用 v 表示

这几个参数一起决定电机将以多快的速度接进目标值，需要注意的是，因为 s 型加减速的加速度和减速度是随时间变化的，这一点和梯形的加速度是固定的有所不同，所以不能确定，但是只要我们给出确定的时间和末速度就可以算出要步进的步数。确定了目标量，下面将对未知量进行具体求解。

17.2.2 获取加加速度段的步数 S1

因为加速度段被分成了加加速度段和减加速度段，并且对应着两个不同的加速度，一个以加加速的方式逼近目标值，一个以减加速的方式逼近目标值，所以两部分所需要的步数也是不相同的，因为求解加速度段所需要的步数，可以通过求解加加速度段与减加速度段的和进行得出：

要获取位移（步数），可知先获取速度，对速度 v 的积分就可以得到位移，如下所示：

$$S_1 = \int v_1 \, dt \quad \text{[式 2-1]}$$

要先求取速度，可对加速度 a 积分则得速度的变化量，如下：

$$\Delta v_1 = \int a_1 \, d\,t \quad \text{[式 2-2]}$$

实际的速度 V1 为初始速度加速度的变化量，如下：

a 为加速度，加速度 a 从 0 变化到最大值，有如下：（其中 J 是加加速度（jerk）,即速度的变化率，）因为这里为加加速度段，对应的时间 t 也是在一定范围内(t=[0,t1])

$$a_1 = J_1 t \quad \text{[式 2-3]}$$

结合两式可得位移公式：

$$S_1 = \int \frac{1}{2} * J_1 t^2 \; dt = \frac{1}{6} J_1 t^3 \quad \text{[式 2-4]}$$

可知，如果我们知道加加速度，并带入我们之前设置好的时间，就可以知道加加速度段所需要的距离，遗憾的是我们还不知道加加速度，所以需要先求解加加速度，根据图形模型可知，加速度和时间的比值就是加加速度(斜率)，速度是等于加速度的积分，则有如下：

$$v_1 = \int J_1 \mathrm{t} \, d\, t = \frac{1}{2} J_1 t^2 \quad \text{[式 2-5]}$$

而加速度在加速度段呈现两种变化，也就是在加速段有两种加速度斜率，我们知道在减加速度部段的加速斜率是以加加速度部段为基础反向变化的，所以只要知道加加速度段斜率就可知减加速度斜率，那么如何求解第一部分的加速度斜率？

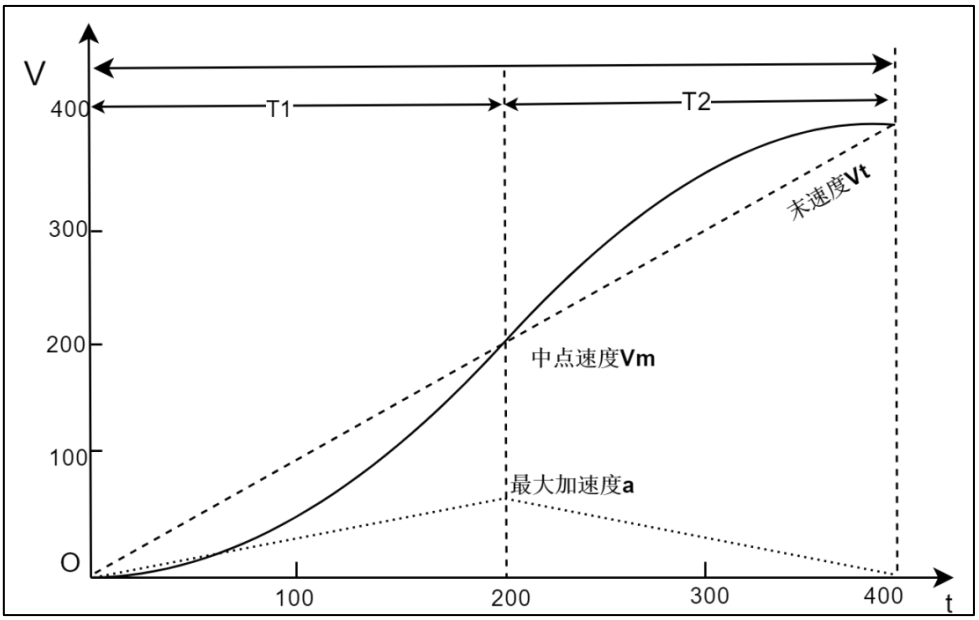


图 17.2.2.1 S 曲线加减速各阶段速度解析

如上图所示：第一部分和第二部分的加速是以加速度最大值作为划分的，而加速度最大值下对应着从初速度速度到末速度的中点速度，当加速度 a 随着时间变化到最大值时速度 V=Vm，由于初速度为 0，中点速度即为末速度的一半，这里的时间为 t=T1,可得加加速度：

$$v_m = \frac{v_0 + v_t}{2} \quad \text{[式 2-6]}$$

由于 Vt 是用户设置的已知的，所以就可以求出 J1：

$$J_1 = \frac{2 * v_m}{t_1^2} \quad [式\ 2\text{-}7]$$

将就得的值 J1 代入[式 2-4]，可得加加速度段的距离为：

$$S_1 = \frac{1}{6} J_1 t^3 \quad [式\ 2\text{-}8]$$

需要注意的是：为了使 S1 的数值直接对应步数，即 S1 的单位为 Step，（如 S1=2，则为 2 步，即需要两个脉冲），在设定速度时，Vm 的单位应该设置为 Step/s，表示 1 秒内可步进多少步数，这样设置还有一个好处是，速度 v 的倒数 1/V（每一步对应的时间）直接对应要输出脉冲的周期，于是乎我们的问题 3（下一个变化的脉冲的周期 Cn 是多少？）就迎刃而解，不用和梯形那样进行繁琐的推导，只要知道速度即可，在后面论述中我们会重点介绍。

那么知道了加加速度段的距离，再求解减加速度段的距离就能得到加速段的距离。

17.2.3 获取减加速度段的步数 S2

要获取减加速度段的位移，同样要知道速度和加速度，而减加速度段的加加速度 J 是和加加速度段是相反的，即为-J，所以有加速度为 a2，其中时间 t 在初始时 t=t1，t1 为加加速度段的所用时间，t2-t1 为减加速段所用时间，则有（t=[t1,t2]）：

$$\alpha_2 = -J(t - t_1) \quad [式\ 2\text{-}9]$$

对加速度进行积分则可得到对应的速度为（t=[t1,t2]）：

$$\Delta v = \int \alpha dt = -\int J(t - t_1)\, dt = -\frac{1}{2} J(t - t_1)^2 \quad [式\ 2\text{-}10]$$

其中 Vt 为这个阶段的末速度：（t=[t1,t2]）

$$v = v_t + \Delta v = v_t - \frac{1}{2} J(t - t_1)^2 \quad [式\ 2\text{-}11]$$

则对速度进行积分求得位移：

$$S_2 = \int v dt = \int v_t - \frac{1}{2} J(t - t_1)^2\, dt = v_t t - \frac{1}{6} J(t - t_1)^3 \quad [式\ 2\text{-}12]$$

已知加加速阶段时间与减加速阶段时间相等，即 t2-t1 = t1，所以减加速度段的位移段 S2：

$$S_2 = v_t t - \frac{1}{6} J t^3 = v_t t - S_1 \quad [式\ 2\text{-}13]$$

因最终加速度段的位移等于：

$$S = S_1 + S_2 \quad [式\ 2\text{-}14]$$

17.2.4 确定下一个脉冲的周期

下一个脉冲的周期决定了电机速度的快慢，根据我们前面的论述，速度的倒数对应每一个脉冲的周期 Tn(Tn=Cn*1/Ft)。

$$C_n / f_t = \frac{1}{v_n}$$

所以对应的定时器计数值为：

$$C_n = \frac{f_t}{v_n}$$

因此要想求出每一步对应的速度，以加加速度段为例，加加速度 J 已经确定，根据位移公式 $v_n = \frac{1}{2} J_1 t^2$，必须要知道每一步速度下对应的时间 t，而这个 t 可以通过位移（$t = \left(\frac{6 * S_1}{J}\right)^{\frac{1}{3}}$）求得，如当 S1=1，即第一步的对应时间为 t1，同理第二步对应的时间为 t2，而 t2 等于第一步时间和第二步脉冲周期的和(t2=t1+T2)，因此 t2-t1 的时间就等于第二步脉冲的周期 T2，但是我们不采用这种方法计算。

我们只需要知道第一步所需的时间 t1，通过 t1 计算出对应的速度 V，速度的倒数 1/V，就是第二步脉冲的周期，然后将第一步时间 t1 加上第二步周期 T2 就是位移公式中步进到第二步的时间总和，以此循环即可得出每一步的脉冲周期，当然了，如果是减加速度段，就使用减加速度段的公式即可。

17.3 S 型曲线加减速的控制实现

17.3.1 硬件设计

1、例程功能

本实验我们将使用步进电机接口三控制电机实现 S 形加减速运动，驱动器设置的 1.0A，8 细分，也就是旋转一圈所需要 1600 个脉冲数，按键 KEY0 控制开启步进电机 S 形加减速运动，KEY1 增加总步数，KEY2 减少总步数，LED0 作为程序运行状态指示灯，闪烁周期 200ms。

2、硬件资源

本实验的硬件设计与前面的步进电机梯形加减速控制章节中的硬件设计完全相同，可查看前面的连接，在此不再赘述

17.3.2 程序设计

17.3.2.1 S 形加减速控制的配置步骤

1）开启 TIMx 和通道输出的 GPIO 时钟，配置该 IO 口的复用功能输出。

首先开启 TIMx 的时钟，然后配置 GPIO 为复用功能输出。本实验我们默认用到定时器 8 通道 3 对应 IO 为 PI7 它的时钟开启方法如下：

```
__HAL_RCC_TIM8_CLK_ENABLE();            /* 使能定时器 8 */
__HAL_RCC_GPIOI_CLK_ENABLE();           /* 开启 GPIOI 时钟 */
```

IO 口复用功能是通过函数 HAL_GPIO_Init 来配置的。

2）初始化 TIMx，设置 TIMx 的 ARR 和 PSC 等参数。

使用定时器的 PWM 模式功能时，我们调用的是 HAL_TIM_PWM_Init 函数来初始化定时器 ARR 和 PSC 等参数。

注意：该函数会调用：HAL_TIM_PWM_MspInit 函数来完成对定时器底层以及其输出通道 IO 的初始化，包括：定时器及 GPIO 时钟使能、GPIO 模式设置、中断设置等。

3）设置定时器为比较输出翻转模式，输出比较极性，比较值等参数。

在 HAL 库中，通过 HAL_TIM_PWM_ConfigChannel 函数来设置定时器为翻转模式，根据需求设置输出比较的极性，设置比较值（控制占空比）等。

本实验我们设置 TIM8 的通道 3 为翻转模式。

4）初始化方向 DIR 引脚，脱机使能 EN 引脚

初始化电机接口三的的方向引脚以及使能引脚。

5）编写 S 加减速的速度表函数

计算 S 加减速的速度表

6）编写步进电机运动状态判断函数

通过对步进电机的步数、加减速度和最大速度的设置来决定步进电机的运动

7）编写中断服务函数

在中断里边对步进电机状态进行决策

17.3.2.2 程序流程图

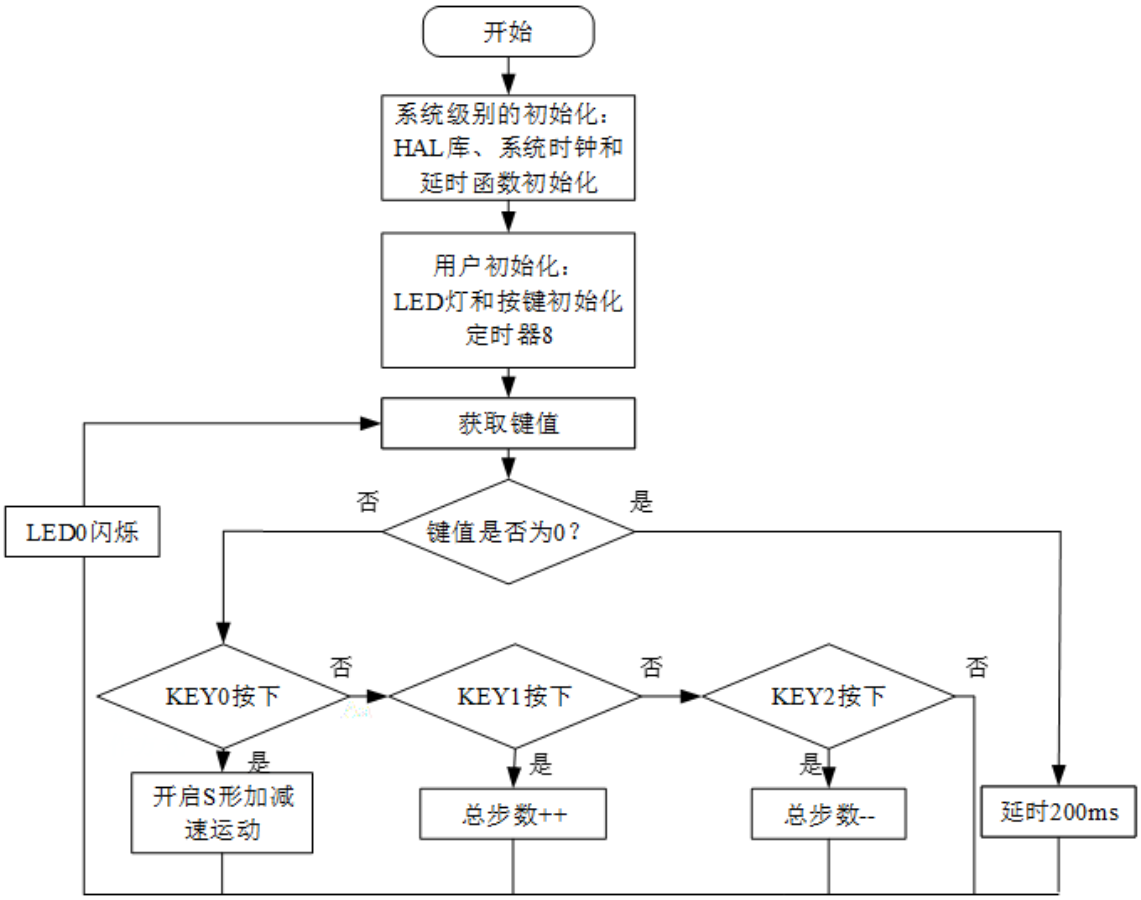


图 17.3.2.2.1 S 形加减速流程图

17.3.2.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。首先就是表格查找法，同梯形加减速不同的是，这里的 S 曲线加减速使用的查表法直接读取速度值。

（1）加减速度表的生成，在 stepper_motor.c 文件中：

```
/**
 * @brief       速度表计算函数
 * @param       vo,初速度;vt,末速度;time,加速时间
 * @retval      TRUE：成功；FALSE：失败
 */
uint8_t calc_speed(int32_t vo, int32_t vt, float time)
{
    uint8_t is_dec = FALSE;
    int32_t i = 0;
    int32_t vm =0;                  /*  中间点速度 */
    int32_t inc_acc_stp;            /*  加加速所需的步数 */
    int32_t dec_acc_stp;            /*  减加速所需的步数 */
    int32_t accel_step;             /*  加速或减速需要的步数 */
    float jerk = 0;                 /*  加加速度 */
    float ti = 0;                   /*  时间间隔 dt */
    float sum_t = 0;                /*  时间累加量 */
    float delta_v = 0;              /*  速度的增量 dv */
    float ti_cube = 0;              /*  时间间隔的立方 */
    float *velocity_tab = NULL; /*  速度表格指针 */
```

```
    if(vo > vt )                    /* 初速度比末速度大,做减速运动,数值变化跟加速运动相同*/
    {                               /*  只是建表的时候注意将速度倒序 */
        is_dec = TRUE;
        g_calc_t.vo = ROUNDPS_2_STEPPS(vt);    /*  起速:step/s */
        g_calc_t.vt = ROUNDPS_2_STEPPS(vo);    /*  末速:step/s */
    }
    else
    {
        is_dec = FALSE;
        g_calc_t.vo = ROUNDPS_2_STEPPS(vo);
        g_calc_t.vt = ROUNDPS_2_STEPPS(vt);
    }

    time = ACCEL_TIME(time);                        /* 得到加加速段的时间 */
    printf("time=%f\r\n",time);
    vm =  (g_calc_t.vo + g_calc_t.vt) / 2 ;         /* 计算中点速度 */

    jerk = fabs(2.0f * (vm - g_calc_t.vo) /  (time * time));/* 计算加加速度 */
     /*  加加速需要的步数 */
    inc_acc_stp = (int32_t)(g_calc_t.vo * time + INCACCELSTEP(jerk,time));
     /*  减加速需要的步数 S = vt * time - S1 */
    dec_acc_stp = (int32_t)((g_calc_t.vt + g_calc_t.vo) * time - inc_acc_stp);

    /* 申请内存空间存放速度表 */
    accel_step = dec_acc_stp + inc_acc_stp;     /* 加速需要的步数 */
    if( accel_step  % 2 != 0)  /* 由于浮点型数据转换成整形数据带来了误差,所以这里加 1 */
        accel_step  += 1;
    /* mallo 申请内存空间,记得释放 */
    velocity_tab = (float*)(mymalloc(SRAMIN,((accel_step + 1) *
                             sizeof(float))));
    if(velocity_tab == NULL)
    {
        printf("内存不足!请修改参数\r\n");
        return FALSE;
    }
/*
 * 目标的 S 型速度曲线是对时间的方程,但是在控制电机的时候则是以步进的方式控制,所以这里对 V-t
 * 曲
 * 线做转换
 * 得到 V-S 曲线,计算得到的速度表是关于步数的速度值.使得步进电机每一步都在控制当中
 */
     /*  计算第一步速度,根据第一步的速度值达到下一步的时间 */
    ti_cube  = 6.0f * 1.0f / jerk;       /*  第 1 步的时间:ti^3 = 6 * 1 / jerk */
    ti = pow(ti_cube,(1 / 3.0f));                       /* ti */
    sum_t += ti;
    delta_v = 0.5f * jerk * pow(sum_t,2);               /* 第一步的速度 */
    velocity_tab[0] = g_calc_t.vo + delta_v;

/*****************************************************/
    if( velocity_tab[0] <= SPEED_MIN )  /* 以当前定时器频率所能达到的最低速度 */
        velocity_tab[0] = SPEED_MIN;
/*****************************************************/

    for(i = 1; i < accel_step; i++)
    {
        /* 步进电机的速度就是定时器脉冲输出频率,可以计算出每一步的时间 */
        /* 得到第 i-1 步的时间 */
        ti = 1.0f / velocity_tab[i-1];   /*  电机每走一步的时间 ti = 1 / Vn-1 */
        /* 加加速段速度计算 */
        if( i < inc_acc_stp)
        {
            sum_t += ti;                                /* 从 0 开始到 i 的时间累积 */
```

```
            delta_v = 0.5f * jerk * pow(sum_t,2);    /* V = 1/2 * jerk * ti^2 */
            velocity_tab[i] = g_calc_t.vo + delta_v; /* 加加速段每一步对应的速度 */
        /* 当最后一步的时候,时间并不严格等于 time,所以这里要稍作处理,作为减加速段的时间 */
            if(i == inc_acc_stp - 1)
                sum_t  = fabs(sum_t - time );
        }
        /* 减加速段速度计算 */
        else
        {
            sum_t += ti;                                        /*  时间累计 */
           /*  dV = 1/2 * jerk *(T-t)^2 */
            delta_v = 0.5f * jerk * pow(fabs( time - sum_t),2);
            velocity_tab[i] = g_calc_t.vt - delta_v;     /*  V = vt - delta_v */
            if(velocity_tab[i] >= g_calc_t.vt)
            {
                accel_step = i;
                break;
            }
        }
    }
    if(is_dec == TRUE)                                          /* 减速 */
    {
        float tmp_Speed = 0;
        g_calc_t.decel_tab = velocity_tab;
        g_calc_t.decel_step = accel_step;
        /* 倒序排序 */
        for(i = 0; i< (accel_step / 2); i++)
        {
            tmp_Speed = velocity_tab[i];
            velocity_tab[i] = velocity_tab[accel_step-1 - i];
            velocity_tab[accel_step-1 - i] = tmp_Speed;
        }
    }
    else                                                        /* 加速 */
    {
        g_calc_t.accel_tab = velocity_tab;
        g_calc_t.accel_step = accel_step;
    }
    return TRUE;
}
```

初始化时需先生成 S 曲线加减速段的速度表，根据前面思路的推导将每一个速度段的速度计算出来后进行存储，读取速度表部分则放置在定时器中断服务函数里面即可，接着看下 S 形加减速控制代码：

```
/**
 * @brief       S 型加减速运动
 * @param       vo:初速度;vt:末速度;AcTime:加速时间;DeTime:减速时间;step:步数;
 * @retval      无
 */
void stepmotor_move_rel(int32_t vo, int32_t vt, float AcTime,float
DeTime,int32_t step)
{
    if(calc_speed(vo,vt,AcTime)==FALSE) /*  计算出加速段的速度 */
        return;
    if(calc_speed(vt,vo,DeTime)==FALSE) /*  计算出减速段的速度 */
        return;

    if(step < 0)
    {
        step = -step;
        ST3_DIR(CCW);
    }
    else
    {
        ST3_DIR(CW);
```

```
    }

    if(step >= (g_calc_t.decel_step+g_calc_t.accel_step) )
    {
        g_calc_t.step = step;
        g_calc_t.dec_point = g_calc_t.step - g_calc_t.decel_step;
    }
    else                                    /* 步数不足以进行足够的加减速 */
    {
        printf("参数设置错误!\r\n");
        return;
    }
    g_calc_t.step_pos = 0;
    g_motor_sta = STATE_ACCEL;              /*  电机为运动状态 */

    g_calc_t.ptr = g_calc_t.accel_tab;  /* 把加速段的速度表存储到 ptr 里边 */
    Toggle_Pulse  = (uint32_t)(T1_FREQ/(*g_calc_t.ptr));
    g_calc_t.ptr++;
    __HAL_TIM_SET_COUNTER(&g_atimx_handle,0);

__HAL_TIM_SET_COMPARE(&g_atimx_handle,TIM_CHANNEL_3,(uint16_t)(Toggle_Pulse));
     /* 设置定时器比较值 */
    HAL_TIM_OC_Start_IT(&g_atimx_handle,ATIM_TIMX_PWM_CH3); /* 使能定时器通道 */
    ST3_EN(EN_ON);
}
```

根据传入的参数判断是加速段还是减速段；如果初始速度小于末速度那么就是加速段，如果初始速度大于末速度那么就是减速段，然后此函数会调用到前面的速度表函数，将用户设定值计算出相应速度参数，并判断符合哪种运动状态，之后判断设定的步数参数是否合理，如果步数不足以实现 S 形加减速运动则直接返回并打印提示信息。当步数合理时，开启定时器工作，比较值设置为速度表第一个参数，即第一步，然后在定时器调用到速度表参数。接着看下中断处理函数：

```
/**
  * @brief  定时器比较中断
  * @param  htim：定时器句柄指针
  * @note   无
  * @retval 无
*/
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim)
{
    volatile uint32_t Tim_Count = 0;
    volatile uint32_t tmp = 0;
    volatile float Tim_Pulse = 0;
    volatile static uint8_t i = 0;
    if(htim->Instance==TIM8)
    {
        i++;                        /*  定时器中断次数计数值 */
        if(i == 2)                  /*  2 次，说明已经输出一个完整脉冲 */
        {
            i = 0;                  /*  清零定时器中断次数计数值 */
            g_step_pos ++;          /* 当前位置 */
            if((g_motor_sta!=STATE_IDLE)&&(g_motor_sta != STATE_STOP))
            {
                g_calc_t.step_pos ++;
            }
            switch(g_motor_sta)
            {
                case STATE_ACCEL:
                    g_add_pulse_count++;
                   /*  由速度表得到每一步的定时器计数值 */
                    Tim_Pulse = T1_FREQ / (*g_calc_t.ptr);
                    g_calc_t.ptr++;                             /* 取速度表的下一位 */
                    Toggle_Pulse = (uint16_t) (Tim_Pulse / 2);/* 翻转模式 C 除以 2 */
```

```
                /* 大于加速段步数就进入匀速 */
                if(g_calc_t.step_pos >= g_calc_t.accel_step)
                {
                    myfree(SRAMIN,g_calc_t.accel_tab);      /* 运动完要释放内存 */
                    g_motor_sta = STATE_AVESPEED;
                }
                break;
            case STATE_DECEL:
                g_add_pulse_count++;
                /* 由速度表得到每一步的定时器计数值 */
                Tim_Pulse = T1_FREQ / (*g_calc_t.ptr);
                g_calc_t.ptr++;
                Toggle_Pulse = (uint16_t) (Tim_Pulse / 2);
                if(g_calc_t.step_pos >= g_calc_t.step )
                {
                    myfree(SRAMIN,g_calc_t.decel_tab);      /* 运动完要释放内存 */
                    g_motor_sta = STATE_STOP;
                }
                break;
            case STATE_AVESPEED:
                g_add_pulse_count++;
                Tim_Pulse  = T1_FREQ /g_calc_t.vt;
                Toggle_Pulse = (uint16_t) (Tim_Pulse / 2);
                if(g_calc_t.step_pos >= g_calc_t.dec_point )
                {
                   /* 将减速段的速度表赋值给 ptr */
                    g_calc_t.ptr = g_calc_t.decel_tab;
                    g_motor_sta = STATE_DECEL;
                }
                break;
            case STATE_STOP:
                HAL_TIM_OC_Stop_IT(&g_atimx_handle, ATIM_TIMX_PWM_CH3);
                g_motor_sta = STATE_IDLE;
                break;
            case STATE_IDLE:
                break;
        }
    }
    /*  设置比较值 */
    Tim_Count=__HAL_TIM_GET_COUNTER(&g_atimx_handle);
    tmp = 0xFFFF & (Tim_Count + Toggle_Pulse);
    __HAL_TIM_SET_COMPARE(&g_atimx_handle,TIM_CHANNEL_3,tmp);
  }
}
```

同梯形定时器输出设置相同，将定时器（TIM8）输出方式设置为输出比较模式，在中断服务函数里使用状态机进行不同速度段的切换，然后根据速度表查找每一步对应的速度值使用 $C_n = f_t / v_n$ 公式即可计算出每一步对应的定时器计数值，该值就是每一步的脉冲周期。需要注意的是，因为使用的是查表方式，所以会消耗一定的存储空间，所以在加减速完成后要释放内存。

在 main 函数里面编写如下代码：

```
int main(void)
{
    uint8_t key,t;
    char buf[32];

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
```

```
    stepper_init(0xFFFF, 168 - 1);

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"Stepper Motor Test",   g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:Run Once",       g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:STEP ++",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:STEP --",  g_point_color);
    printf("KEY0 开启 S 形加减速\r\n");
    printf("KEY1 增加步数\r\n");
    printf("KEY2 减少步数\r\n");

    while (1)
    {
        t++;
        if(t % 20 == 0)
        {
            sprintf(buf,"Set_Aangle:%d ",g_step_angle);  /*设置的旋转位置（角度）*/
            lcd_show_string(10,90,200,16,16,buf,g_point_color);
            sprintf(buf,"Add_Aangle:%.2f ",g_add_pulse_count*0.225);
            lcd_show_string(10,110,200,16,16,buf,g_point_color);
            LED0_TOGGLE();                              /*LED0(红灯) 翻转*/
        }
        delay_ms(10);
        key = key_scan(0);
        if(key == KEY0_PRES)                            /* 开启电机 S 型加减速 */
        {
            if(g_motor_sta == STATE_IDLE)
            {
                g_add_pulse_count=0;
                stepmotor_move_rel(V_START,V_END,ACCELTIME,DECEELTIME,
                                   g_step_angle*SPR );
            }
        }
        if(key == KEY1_PRES)                            /* 步数加 */
        {
            g_step_angle=g_step_angle+1;
            if(g_step_angle>=50)  g_step_angle=1;
        }
        if(key == KEY2_PRES)                            /* 步数减 */
        {
            g_step_angle=g_step_angle-1;
            if(g_step_angle<=1)  g_step_angle=50;
        }
    }
}
```

主函数的内容比较简单，通过按键判断逻辑实现，当按下 KEY0 控制开启步进电机 S 形加减速运动，KEY1 增加总步数，KEY2 减少总步数。LED0 周期闪烁提示程序正在运行。

17.3.3 下载验证

下载代码后，可以看到 LED0 每隔 200ms 左右闪烁一次，LCD 和串口都会显示提示信息，根据屏幕提示，点击按键 0 即可使步进电机做 S 形加减速运动，如果现象不明显，请通过按键 KEY1 增加总步数，当步数足够多，可以明显感觉到步进电机由缓慢加速，再到最大速度，最后缓慢减速直到停止。

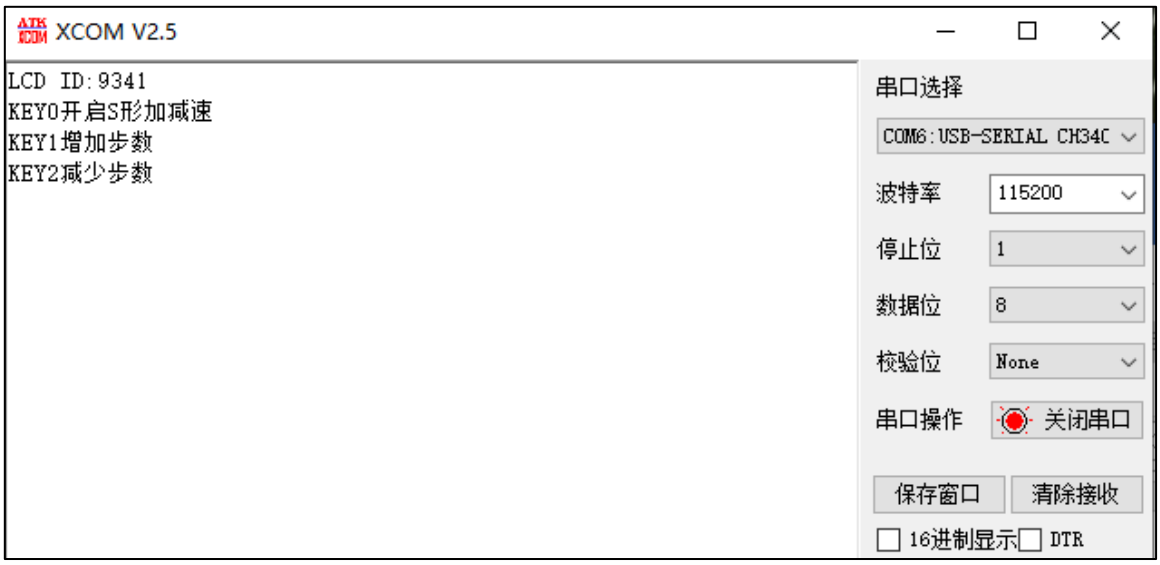


图 17.3.3.1 串口打印提示信息

注意驱动器设置 8 细分，使用到的是步进电机接口三。实验现象大家可自行测试看下效果。

第 18 章 步进电机直线插补

前面我们所学习的单一步进电机只能完成和结构相关的简单直线运动(例如丝杆)，也称为“一维运动”，但在实际应用中往往需要工件做斜线甚至圆弧运动，这时就需要通过两个步进电机连轴做同步运动才能实现，这种控制方式称为“多轴联动”。所以本章我们将学习控制两个直线位移的步进电机实现双轴联动，即：步进电机的直线插补运动。

本章分为如下几个小节：

18.1 插补的简介
18.2 插补算法
18.3 逐点比较法直线插补原理
18.4 任意象限直线插补原理
18.5 任意象限直线插补实验

18.1 插补的简介

插补：机床数控系统依照一定方法确定刀具运动轨迹的过程。也可以说，已知曲线上的某些数据，数控装置根据输入的零件程序的信息，将程序段所描述的曲线的起点、终点之间的空间进行数据密化，从而形成要求的轮廓轨迹，这种“数据密化”机能就称为“插补”。

在数控机床中，刀具不能严格地按照要求加工的曲线运动，只能用折线轨迹逼近所要加工的曲线。 机床数控系统依照一定方法确定刀具运动轨迹的过程。 也可以说，已知曲线上的某些数据，按照某种算法计算已知点之间的中间点的方法，也称为**“数据点的密化”**。

常见的插补方式有：直线插补、圆弧插补、抛物线插补、样条线插补等。而我们本章所要学习的就是直线插补，我们下面了解下什么是直线插补。

直线插补(Llne Interpolation)：是车床上常用的一种插补方式，通常用于走斜线或直线轨迹，在此方式中，两点间的插补沿着直线的点群来逼近。在实际应用中一个零件的轮廓可以是多样的，有直线、圆弧，也有可能是任意曲线、样条线等，而数控机床的刀具往往是不能以曲线的实际轮廓去走刀的，而是近似地以若干条很小的直线去走刀逼近，走刀的方向一般是 x 和 y 方向。

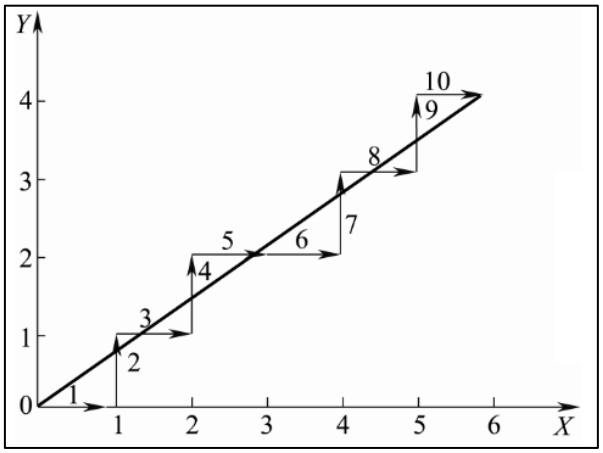


图 18.1.1 直线插补图示

18.2 插补算法

插补算法的好坏，直接影响系统的控制速度和零件加工精度，那常见的插补算法有哪些？常见的插补算法有：脉冲增量插补法、数字增量插补法这两大类，下面我们来介绍下这两种算法。

- 脉冲增量插补法：该插补为行程标量插补，常用于开环系统，每次插补结束产生一个行程

增量，以脉冲的方式输出，一个脉冲所产生的坐标轴移动量叫做脉冲当量，也可以说，通过向**各个运动轴分配驱动脉冲**来控制机床坐标轴协调运动，从而加工出一定轮廓形状的算法

- 数字增量插补法：该插补为时间标量插补，分两步进行。首先计算出插补周期内各坐标轴的增量值，称为粗插补；然后在根据采样得到的实际位置增量计算跟随误差，得到速度指令输出给伺服驱动系统，称为精插补。使用于闭环或者半闭环系统。

由上述两种插补算法，其中脉冲增量算法非常适用于以步进电机为驱动装置的开环系统。脉冲增量算法也分为多种：**逐点比较法**、数字积分法、最小偏差法、目标点跟踪法、单步追踪法等，我们这里只介绍我们最为常用的逐点比较法。

逐点比较法：顾名思义，就是每走一步都要将加工点的瞬时坐标同规定的图形轨迹相比较，判断其偏差，然后决定下一步的走向，如果加工点走到图形外面去了，那么下一步就要向图形里边走，如果加工点在图形里边，那下一步就往图形外面走，以缩小偏差，这样就能得到一个非常接近规定图形的轨迹。

18.3 逐点比较法直线插补原理

18.3.1 逐点比较法整体流程图

逐点比较法不管是直线插补还是圆弧插补步骤都分为以下四步：

（1）偏差判别：根据偏差函数值判别加工点的相对直线位置

（2）坐标进给：沿减小误差的方向进给一步

（3）偏差计算：进给后，计算新加工点相对直线的位置

（4）终点判别：判别是否到达终点，未到达则继续第一步步骤，继续插补，到终点则停止

整体流程图如下：

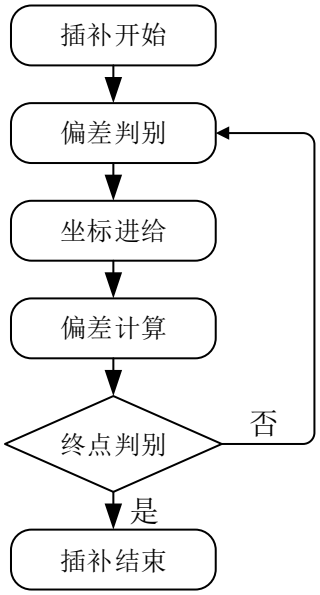


图 18.3.1.1 逐点比较法流程图

18.3.2 偏差判别

设有斜线 OA，A 点坐标为（Xe,Ye），有一动点 P（Xi.Yi），他可能在 OA 线上，也可能在线的上方或下方，所以就有以下几种可能：

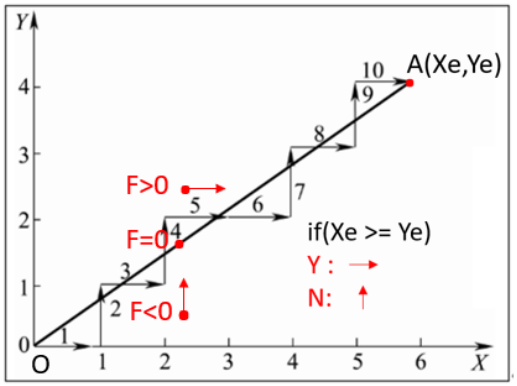


图 18.3.2.1 第一象限直线插补示意图

- 当 P 点在斜线上时，由于斜率相等就有：
  Yi / Xi = Ye / Xe
  取偏差判别式：**F= Xe*Yi – Xi*Ye = 0**
- 当 P 点在直线上方，此时 OP 直线的斜率比 OA 大，那么有下述关系：
  Yi / Xi > Ye / Xe
  偏差判别式： **F = Xe*Yi – Xi*Ye > 0**
- 当 P 点在直线下方，此时 OP 直线的斜率比 OA 小，那么有下述关系：
  Yi / Xi < Ye / Xe
  偏差判别式： **F= Xe*Yi – Xi*Ye <0**
  综上所述，可知 F 的大小反映了动点 P 与斜线 OA 的相对位置
  当 F = 0，表示动点 P 在直线 OA 上；
  当 F > 0，表示动点 P 在直线 OA 上方；
  当 F < 0，表示动点 P 在直线 OA 下方。

18.3.3 坐标进给

当我们通过偏差判别知道动点与直线的相对位置后，接着就是坐标进给，需往误差减小的方向进给：

当 F = 0 时，在直线上，需判断终点坐标 x、y 的值，当 x >= y 时，动点 P 往+X 方向进给一步，否则往+Y 方向进给一步；

当 F > 0 时，在直线上方，应该向+X 方向进给一步；

当 F < 0 时，在直线下方，应该向+Y 方向进给一步；

按照上述方法进行步进，就可以走出逼近目标直线的轮廓出来。

18.3.4 偏差计算

接着就是偏差判别式的化简:

由上面可知，偏差判别式为：F(Xi , Yi) = **Xe*Yi – Xi*Ye;**

假设下一步向+X 步进，那么此时:

F(Xi+1 , Yi) = Xe*Yi –(Xi+1)Ye = **Xe*Yi - Xi*Ye** - Ye = **F(Xi,Yi) –Ye**

假设下一步向+Y 步进，那么此时:

F(Xi , Yi+1) = Xe(Yi+1) – Xi*Ye = **Xe*Yi -Xi*Ye** +Xe = **F(Xi,Yi) +Xe**

经过上述化简后，判别式简单很多，只需要知道终点坐标及上一步的偏差值，便可求出下一步的偏差值。

18.3.5 终点判别

最后就是终点判别，插补到何时停止呢？常用的终点判别法有三种，分别是：总步长法、终点坐标法、投影法。

- 总步长法：插补前，将终点位置的 x,y 坐标绝对值相加求和，得出总步数，开始插补时，

无论是往哪个轴给进一步，总步数都减一，直到为 0 时，代表已到终点，就停止插补。

- 终点坐标法：插补前，先定义 x、y 轴两个方向的步长计数器，分别存放两个坐标轴方向上应该走的总步数。开始插补后当 x、y 方向每进给一步，就在相应的计数器中减 1，直到两个计数器的值都减为 0 时，刀具抵达终点，停止插补。
- 投影法：在插补处理开始之前，先确定所走步数较大的那根轴，并求出该轴运动的总步数，然后存放在总步长计数器中，当对应的轴有进给时，计数器减 1，直到计数器为 0。相当于终点坐标向值较大的轴做投影，所以叫投影法。

由于总步长法实现简单且易于理解，所以在我们本实验中，终点判别使用的就是该方法。

18.4 任意象限直线插补的原理

前面讲解的都是以第一象限为例，实际上为了完成任意直线插补还需要考虑运动方向在 4 个象限内的情况，原理与第一象限同理，所以理解起来较为简单。

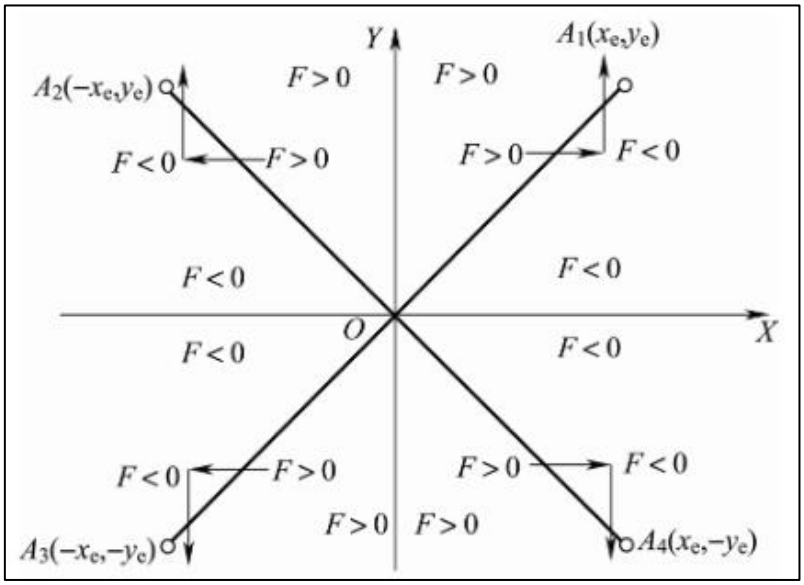


图 18.4.1 四个象限直线插补图示

这里以第二象限为例，此时的坐标点 X 轴方向都是小于 0，与原本第一象限+X 相反，所以此时步进电机运动方向需要反向，Y 轴都为正，电机运动方向无需改变，这就可以插补出第二象限的直线了，第三、四象限同理，只要与第一象限坐标符号相反，所对应的轴的步进电机就往反方向运动，否则同向。总结下表：

直线坐标	F > 0	偏差计算公式	F < 0	偏差计算公式
第一象限	+X	$F_{i+1} = Fi - Ye$	+Y	$F_{i+1} = Fi + Xe$
第二象限	-X		+Y	
第三象限	-X		-Y	
第四象限	+X		-Y	

表 18.4.1 四个象限的直线插补计算

这样我们就已将直线插补的原理部分讲解完毕，接下来让我们来到实战编程看看如何实现直线插补运动。

18.5 任意象限直线插补实验

18.5.1 硬件设计

1、例程功能

本实验使用步进电机接口三控制 X 轴步进电机和接口四控制 Y 轴步进电机，当按下 KEY0 控制步进电机开始直线插补运动。注意驱动器设置为 1.0A，8 细分，也就是旋转一圈所需要

1600 个脉冲数；LED0 作为程序运行状态指示灯，闪烁周期 200ms。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键：
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器 8： TIM8_CH3：PI7，对应步进电机接口三
TIM8_CH4：PC9，对应步进电机接口四
方向引脚： DIR3: PB2
DIR4: PH2
脱机引脚： EN3: PF11
EN4: PH3
4）丝杆滑台步进电机 2 个
5）步进电机驱动器 2 个
6）12-50V 的 DC 电源

3、原理图

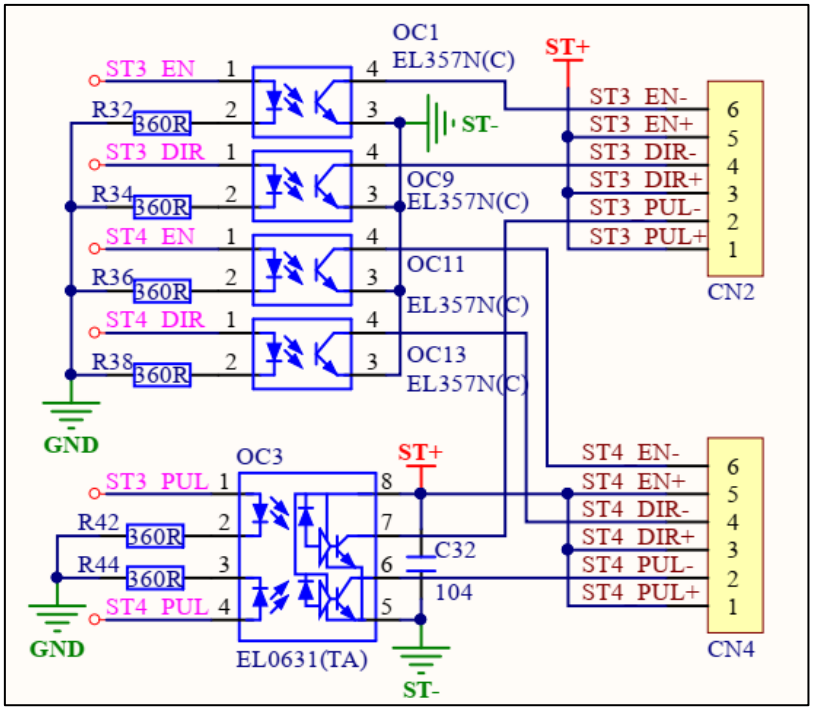


图 18.5.1.1 步进电机接口 3、4 原理图

本章将用到两个步进电机接口，分别是接口三以及接口四，如上图所示，接口三对应控制 X 轴的步进电机，接口四对应控制 Y 轴的步进电机。

18.5.2 程序设计

18.5.2.1 任意象限直线插补的配置步骤

1）初始化定时器

初始化定时器通道 IO，设置 ARR、PSC，计数方式以及脉冲输出模式等

2）电机旋转方向判定

实现任意象限直线插补，需提前判断所属象限，设置电机旋转方向

3）编写直线插补算法

实现插补第一步，偏差判别，坐标进给，偏差计算并开启脉冲输出与中断使能

4）编写中断服务函数

实现除第一步以外的其他步数的偏差判别、坐标进给、偏差计算、终点判别

18.5.2.2 程序流程图

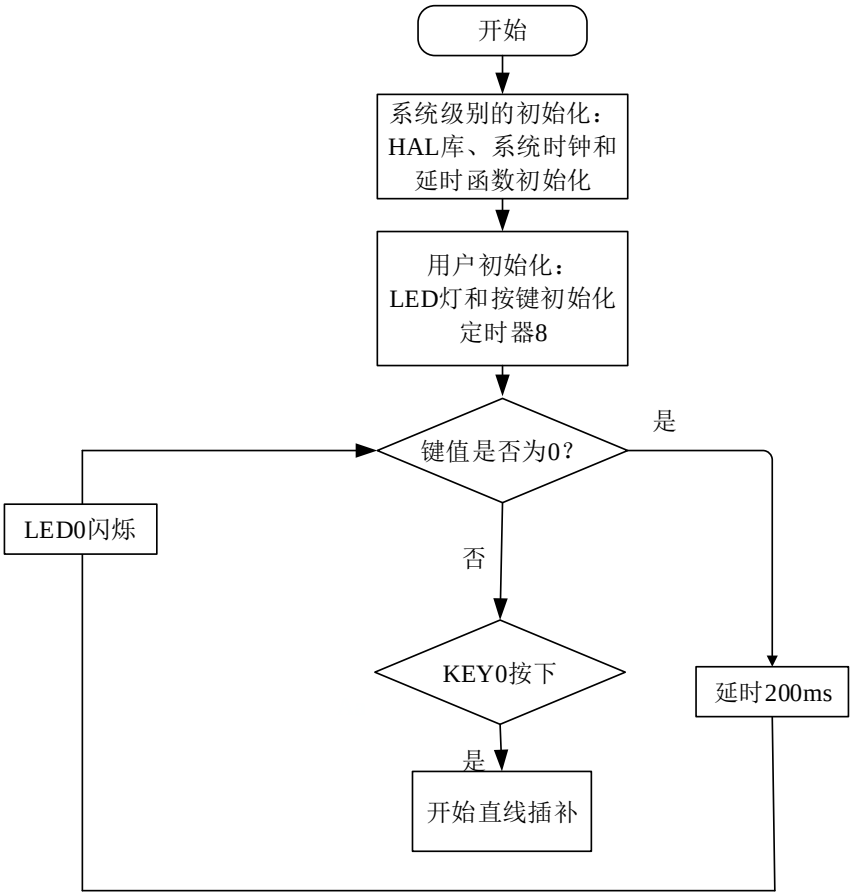


图 18.5.2.2.1 直线插补流程图

18.5.2.3 程序解析

本实验在步进电机的基础驱动下添加代码，这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。源码我们主要分成定时器初始化、直线插补算法函数以及中断逻辑处理三大部分讲解，首先看下定时器，本次使用的 PWM 模式驱动，详解看 stepper_tim.c

```
/**
 * @brief       高级定时器 PWM 初始化函数
 * @note
 *              高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 * @param       arr: 自动重装值
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void atim_timx_oc_chy_init(uint16_t arr, uint16_t psc)
{
```

```
    ATIM_TIMX_PWM_CHY_CLK_ENABLE();                                 /* TIMX 时钟使能
*/


    g_atimx_handle.Instance = ATIM_TIMX_PWM;                        /* 定时器 x */
    g_atimx_handle.Init.Prescaler = psc;                            /* 定时器分频 */
    g_atimx_handle.Init.CounterMode = TIM_COUNTERMODE_UP;           /* 向上计数模式 */
    g_atimx_handle.Init.Period = arr;                               /* 自动重装载值 */
    g_atimx_handle.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;       /* 分频因子 */
    g_atimx_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    g_atimx_handle.Init.RepetitionCounter = 0;                      /* 开始时不计数*/
    HAL_TIM_PWM_Init(&g_atimx_handle);                              /* 初始化 PWM */

    g_atimx_oc_chy_handle.OCMode = TIM_OCMODE_PWM2;                 /* 模式选择 PWM1*/
    g_atimx_oc_chy_handle.Pulse = arr/2;
    g_atimx_oc_chy_handle.OCPolarity = TIM_OCPOLARITY_HIGH;  /* 输出比较极性为低*/
    g_atimx_oc_chy_handle.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    g_atimx_oc_chy_handle.OCFastMode = TIM_OCFAST_DISABLE;
    g_atimx_oc_chy_handle.OCIdleState = TIM_OCIDLESTATE_RESET;
    g_atimx_oc_chy_handle.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
                              ATIM_TIMX_PWM_CH1);                   /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
                              ATIM_TIMX_PWM_CH2);                   /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
                              ATIM_TIMX_PWM_CH3);                   /* 配置 TIMx 通道 y */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
                              ATIM_TIMX_PWM_CH4);                   /* 配置 TIMx 通道 y */
    HAL_NVIC_SetPriority(TIM8_UP_TIM13_IRQn, 2, 2);                 /* 设置中断优先级分组*/
    HAL_NVIC_EnableIRQ(TIM8_UP_TIM13_IRQn);
    /* 直线插补提前使能主输出以及清除更新中断标志位 */
    __HAL_TIM_MOE_ENABLE(&g_atimx_handle);                          /* 主输出使能 */
    __HAL_TIM_CLEAR_IT(&g_atimx_handle,TIM_IT_UPDATE);              /* 清除更新中断标志位*/

}
```

HAL_TIM_PWM_Init 初始化 TIM8 并设置 TIM8 的 ARR 和 PSC 等参数，然后通过调用函数 HAL_TIM_PWM_ConfigChannel 设置 TIM8_CH3、4 的 PWM 模式以及比较值等参数，最后通过调用函数 HAL_TIM_PWM_Start 来使能 TIM8 以及使能 PWM 通道 TIM8_CH3、4 输出。通道的 GPIO 口初始化和定时器使能等程序放到回调函数 HAL_TIM_PWM_MspInit 中这里就不贴代码了，感兴趣打开源码看看。接着我们来看下 stepper_motor.h 核心部分:

```
/* 直线插补定义 */

#define AXIS_X              0   /* X 轴标号 */
#define AXIS_Y              1   /* Y 轴标号 */
#define LINE                0

enum dir
{
    CCW = 0,            /* 逆时针旋转 */
    CW ,                /* 顺时针旋转 */
};

typedef enum            /* 电机状态 */
{
    STATE_STOP = 0,
    STATE_RUN = 1,
} st_motor_status_def;

typedef struct {
    uint16_t        pulse_pin;      /* 定时器脉冲输出引脚 */
    uint32_t        pulse_channel;  /* 定时器脉冲输出通道 */
```

```
    uint16_t       en_pin;          /* 电机使能引脚编号 */
    uint16_t       dir_pin;         /* 电机方向引脚编号 */
    GPIO_TypeDef    *dir_port;      /* 电机方向引脚端口 */
    GPIO_TypeDef    *en_port;       /* 电机使能引脚端口 */
} st_motor_ctr_def;

/*  插补算法类型定义 */
typedef struct {
    __IO uint8_t    moving_mode;    /* 运动模式 */
    __IO uint8_t    inter_dir;      /* 插补方向 */
    __IO uint8_t    qua_points;     /* 象限点 */
    __IO uint8_t    x_dir;          /* X 轴方向 */
    __IO uint8_t    y_dir;          /* Y 轴方向 */
    __IO int32_t    end_x;          /* 终点坐标 X */
    __IO int32_t    end_y;          /* 终点坐标 Y */
    __IO uint32_t   end_pulse;      /* 终点位置总的脉冲数 */
    __IO uint32_t   act_axis;       /* 活动轴 */
    __IO int32_t    f_e;            /* 函数方程 */
} inter_pol_def;

/* X 轴电机相关引脚定义 */
#define STEPMOTOR_TIM_CHANNEL1      TIM_CHANNEL_3   /* 定时器 8 通道 3 X 轴 */
#define STEPMOTOR_TIM_PULSE_PIN_X   GPIO_PIN_7      /* 输出脉冲给 X 轴步进驱动器 */

#define STEPMOTOR_X_DIR_PORT        GPIOB           /* X 轴方向脚 */
#define STEPMOTOR_X_DIR_PIN         GPIO_PIN_2      /* X 轴方向脚 */

#define STEPMOTOR_X_ENA_PORT        GPIOF           /* X 轴使能脚 */
#define STEPMOTOR_X_ENA_PIN         GPIO_PIN_11     /* X 轴使能脚 */

/* Y 轴电机相关引脚定义 */
#define STEPMOTOR_TIM_CHANNEL2      TIM_CHANNEL_4   /* 定时器 8 通道 4 Y 轴 */
#define STEPMOTOR_TIM_PULSE_PIN_Y   GPIO_PIN_9      /* 输出脉冲给 Y 轴步进驱动器 */

#define STEPMOTOR_Y_DIR_PORT        GPIOH           /* Y 轴方向脚 */
#define STEPMOTOR_Y_DIR_PIN         GPIO_PIN_2      /* Y 轴方向脚 */

#define STEPMOTOR_Y_ENA_PORT        GPIOH           /* Y 轴使能脚 */
#define STEPMOTOR_Y_ENA_PIN         GPIO_PIN_3      /* Y 轴使能脚 */
```

这里可以分为三部分，第一部分是步进电机运动相关的一些宏定义比如方向，运动状态等；第二部分 inter_pol_def 该结构体主要与直线插补算法相关；第三部分是关于步进电机 X 和 Y 轴上步进电机相关的引脚宏定义以及结构体定义。接着看下 stepper_motor.c 关于直线插补的部分源码：

```
/**
 * @brief       直线增量插补函数实现直线插补功能,两个步进电机分别向 X 轴和 Y 轴步进
IncX,IncY 步
 * @param       IncX    : 终点 X 轴坐标
 * @param       IncY    : 终点 Y 轴坐标
 * @param       Speed   : 进给速度
 * @retval      无
 */
void line_incmove(uint32_t IncX,uint32_t IncY,uint32_t Speed)
{
    /* 偏差方程置零 */
    g_pol_par.f_e = 0;

    /* 计算起点到终点坐标对应的脉冲数位置*/
    g_pol_par.end_x = IncX;
    g_pol_par.end_y = IncY;
```

```
    g_pol_par.end_pulse = g_pol_par.end_y + g_pol_par.end_x;

    /* 根据终点判断在直线上的进给方向,减少偏差 */
    if(g_pol_par.end_y > g_pol_par.end_x)
    {
        g_pol_par.act_axis = AXIS_Y;              /* 第一步进给 Y 轴 */
        g_pol_par.f_e = g_pol_par.f_e + g_pol_par.end_x;
    }
    else
    {
        g_pol_par.act_axis = AXIS_X;              /* 第一步进给 X 轴 */
        g_pol_par.f_e = g_pol_par.f_e - g_pol_par.end_y;
    }
    /* 设置通道的比较值 */
     __HAL_TIM_SET_COMPARE(&g_atimx_handle,st_motor[AXIS_X].pulse_channel,
                           Speed);
     __HAL_TIM_SET_COMPARE(&g_atimx_handle,st_motor[AXIS_Y].pulse_channel,
                           Speed);
    /* ARR 设置为比较值 2 倍，这样输出的波形就是 50%的占空比 */
    __HAL_TIM_SET_AUTORELOAD(&g_atimx_handle,Speed*2);

    TIM_CCxChannelCmd(TIM8, st_motor[g_pol_par.act_axis].pulse_channel,
                      TIM_CCx_ENABLE);        /* 使能通道输出 */
    HAL_TIM_Base_Start_IT(&g_atimx_handle);   /* 使能定时器以及开启更新中断 */
    g_motor_sta = STATE_RUN;                  /* 标记电机正在运动 */
}
/**
 * @brief       实现任意象限直线插补
 * @param       coordsX    : 终点 X 轴坐标
 * @param       coordsY    : 终点 Y 轴坐标
 * @param       Speed      : 进给速度
 * @retval      无
*/
void line_inpolation( int32_t coordsX,int32_t coordsY, int32_t Speed)
{
    if(g_motor_sta != STATE_STOP)         /* 当前电机正在运转 */
       return ;
    /* 其他象限的直线跟第一象限是一样,只是电机运动方向不一样 */
    g_pol_par.moving_mode = LINE;
    if(coordsX < 0)                       /* 当 x 轴小于 0 时，电机方向设为反向*/
    {
        g_pol_par.x_dir = CCW;
        coordsX = -coordsX;               /* 取绝对值 */
        ST3_LINE_DIR(CCW,AXIS_X);
    }
    else
    {
        g_pol_par.x_dir = CW;
        ST3_LINE_DIR(CW,AXIS_X);
    }
    if(coordsY < 0)                       /*当 y 轴小于 0 时，电机方向设为反向*/
    {
        g_pol_par.y_dir = CCW;
        coordsY = -coordsY;               /* 取绝对值 */
        ST3_LINE_DIR(CCW,AXIS_Y);
    }
    else
    {
        g_pol_par.y_dir = CW;
        ST3_LINE_DIR(CW,AXIS_Y);
    }
    line_incmove(coordsX,coordsY,Speed);
}
```

这里主要有两个函数，其中函数 Line_InPolation 用来实现任意象限直线插补的功能，有三个入口参数分别是：直线终点 X 坐标和 Y 坐标以及步进电机运行速度。首先该函数会判断终点坐标正负符号，设置对应的步进电机旋转方向，然后调用函数 Llne_IncMove，该函数是我们实现插补算法的核心代码，首先会通过终点坐标计算出总步数，然后进行偏差判别、第一步的进给方向以及计算下一步的偏差值，然后设置比较值以及重装载值进而控制电机速度，最后使能定时器输出。到这里只是实现第一步的进给，后面的步数都是在中断处理逻辑实现，接着我们看下中断回调函数：

```
/**
 * @brief       定时器中断回调函数
 * @param       htim :  定时器句柄
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    __IO uint32_t axis = 0;          /* 进给轴 */
    axis = g_pol_par.act_axis;       /* 当前进给轴 */

    /* 判断是否到达终点或者还没开始运动 */
    if(g_pol_par.end_pulse == 0)
        return;
    /* 根据进给方向 更新坐标值 */
    if(g_pol_par.moving_mode == LINE)
    {
      /* 偏差方程 > 0 ,位置位于直线上方,应向 X 轴进给 */
       if(g_pol_par.f_e > 0)
       {
           g_pol_par.act_axis = AXIS_X;
          /* 第一象限的 X 轴进给时,偏差计算 */
           g_pol_par.f_e = g_pol_par.f_e - g_pol_par.end_y;
       }
       else if(g_pol_par.f_e < 0)   /* 偏差方程 < 0 ,位置位于直线下方,应向 Y 轴进给 */
       {
           g_pol_par.act_axis = AXIS_Y;
          /* 第一象限的 Y 轴进给时,偏差计算 */
           g_pol_par.f_e = g_pol_par.f_e+g_pol_par.end_x;
       }
       /* 偏差为 0 的时候,判断 x,y 轴终点的大小决定进给方向 */
       else if(g_pol_par.f_e == 0) /* 偏差方程 = 0 ,直线上,应判断终点坐标再进给 */
       {
           if(g_pol_par.end_y > g_pol_par.end_x) /* 当 Y 轴更长的话，应向 Y 轴进给 */
           {
               g_pol_par.act_axis = AXIS_Y;
               g_pol_par.f_e = g_pol_par.f_e+g_pol_par.end_x;
           }
           else
           {
               g_pol_par.act_axis = AXIS_X;
               g_pol_par.f_e = g_pol_par.f_e - g_pol_par.end_y;
           }
       }
    }
    /* 判断是否需要跟换进给轴 */
    if(axis != g_pol_par.act_axis)
    {
        TIM_CCxChannelCmd(TIM8, st_motor[axis].pulse_channel, TIM_CCx_DISABLE);
        TIM_CCxChannelCmd(TIM8, st_motor[g_pol_par.act_axis].pulse_channel,
                          TIM_CCx_ENABLE);
    }
    /* 终点判别:总步长 */
    g_pol_par.end_pulse--;
    if(g_pol_par.end_pulse == 0)
    {
```

```
        g_motor_sta = STATE_STOP;                   /* 到达终点 */
        TIM_CCxChannelCmd(TIM8, st_motor[g_pol_par.act_axis].pulse_channel,
                          TIM_CCx_DISABLE);   /* 关闭当前轴输出 */
        HAL_TIM_Base_Stop_IT(&g_atimx_handle);   /* 停止定时器 */
    }
}
```

在中断回调函数中，首先保存上一次的活动轴是 x 还是 y 的步进电机，然后继续进行除第一步以外，其他步数的偏差判别、坐标进给、新的偏差计算，然后就是终点判别，并比较上一步进给轴与当前进给轴是不是一致，如果不是就切换当前轴。终点判别只要总步数值不等于 0 就会一直重复进行以上步骤，当等于 0 时才会关闭定时器结束插补运动。

在 main 函数里面编写如下代码：

```
int main(void)
{
    uint8_t key,t;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    stepper_init(0xFFFF, 8 - 1);                /* 定时器初始化 */

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"Stepper Motor Test",   g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:Run Once",        g_point_color);
    printf("直线插补实验，按 KEY0 开始插补\r\n");

    while (1)
    {
        key = key_scan(0);
        if(key == KEY0_PRES)                            /* 按下 KEY0 开始插补 */
        {
            line_inpolation(5*1600,5*1600, 2000);    /* 第一象限直线类似图形'/'  */
            while(g_motor_sta);
            line_inpolation(5*1600, -5*1600, 2000); /* 第四象限直线类似图形'\'  */
            while(g_motor_sta);
            line_inpolation(-5*1600,-5*1600, 2000); /* 第三象限直线类似图形'/'  */
            while(g_motor_sta);
            line_inpolation(-5*1600, 5*1600,2000);   /* 第二象限直线类似图形'\'  */
            while(g_motor_sta);
        }
        t++;
        if(t % 20 == 0)
        {
            LED0_TOGGLE();                              /* LED0(红灯) 翻转*/
        }
        delay_ms(10);
    }
}
```

mian 函数主要就是初始化一些外设包括串口、定时器初、LCD 等等，串口和 LCD 都会显示提示信息，当按下 KEY0 之后就会分别开启四个象限的直线插补运动。最后可以看出走的轨迹是一个正方形。大家如果要增加走的长度，可以自己调整 Line_InPolation 入口参数的 x 与 y 的坐标。

18.5.3 下载验证

需使用两个丝杆滑台组成一个标准的两轴 X-Y 平台，硬件连接好之后下载程序到开发板，液晶屏和串口都会显示提示信息，当我们按下按键 KEY0，将会开启直线插补运动，最后可以看出我们的源码实现一个走正方形的轨迹，大家可自行下载程序，查看现象。

图 18.5.3.1 串口提示信息

第 19 章 步进电机圆弧插补

上一章我们学习了直线插补，但直线插补针对的是走直线或者斜线轮廓形状轨迹，而在实际的数控机床上，不单单只有走斜线或直线，如果需要走的目标轮廓是弧线呢？很显然，直线插补满足不了，所以这时候就需要用到今天所学习的内容——圆弧插补。

本章分为如下几个小节：

19.1 圆弧插补的简介
19.2 逐点比较法圆弧插补原理
19.3 任意象限圆弧插补原理
19.4 任意象限圆弧插补实验

19.1 圆弧插补的简介

在圆弧起点与终点间，计算逼近实际圆弧的点群，控制刀具沿点运动，加工出圆弧曲线。它的思想与直线插补的类似，并且我们都是使用的逐点比较法来实现，所以它们的插补步骤也一样都分为：偏差判别、坐标进给、偏差计算、终点判别。我们这里为了降低难度，先从第一象限逆圆弧一步一步来进行分析。

19.2 逐点比较法圆弧插补原理

19.2.1 偏差判别

设要加工图形为第一象限逆时针走向的圆弧 AE，以原点为圆心，半径为 R，动点 **P(Xi,Yi)**，动点 P 可能在圆弧外也可能在圆弧内，还有可能在圆弧上，所以就有三种可能，下面我们逐一来分析下：

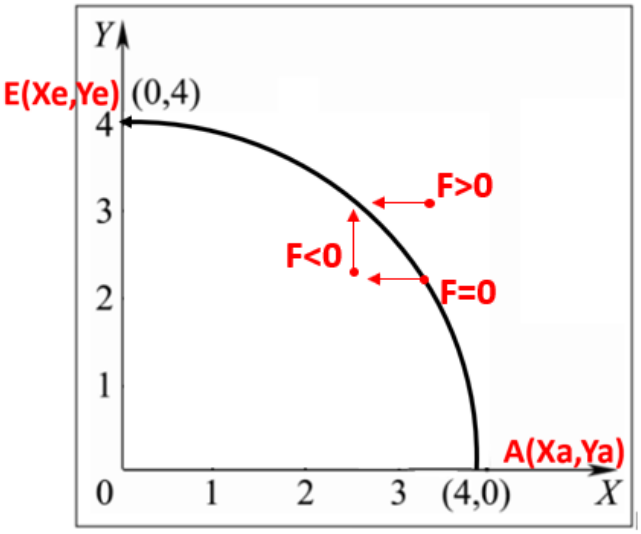


图 19.2.1.1 第一象限逆圆弧插补示意图

- 当 P 点在**圆弧上**时，那么由勾股定理可得：
  $Xi^2+Yi^2 = R^2$
  即偏差判别式：**$F = Xi^2+Yi^2-R^2 = 0$**
- 当 P 点在**圆弧外**的话，就有以下关系：
  $Xi^2+Yi^2 > R^2$
  偏差判别式： **$F = Xi^2+Yi^2-R^2 > 0$**
- 当 P 点在**圆弧内**的话，就有以下关系：
  $Xi^2+Yi^2 < R^2$

偏差判别式：**$F = Xi^2+Yi^2-R^2 < 0$**

综上所述，可知 F 的大小反映了动点 P 与圆弧 AE 的相对位置

当 F = 0，表示动点 P 在圆弧 AE 上；

当 F > 0，表示动点 P 在圆弧 AE 外部；

当 F < 0，表示动点 P 在圆弧 AE 内部。

19.2.2 坐标进给

当我们知道动点与圆弧的相对位置后，接着就是坐标进给，需往误差减小的方向进给。

当 F = 0 时，在圆弧上，此时不管是往 x 轴步进还是往 y 轴步进它们之间的误差是差不多的，所以这里我们为了方便计算统一将其运动方向与 F>0 一样，也就是-X 的方向步进。 但是这里要注意 F=0 表示动点 P 是在圆弧上的任意一点，所以也可能在 x 轴上，此时注意需要往+Y 方向步进才可减少误差，其他的位置均与 F>0 的步进方向一致即可。

当 F > 0 时，在圆弧外部，应该向圆内-X 方向进给一步；

当 F < 0 时，在圆弧内部，应该向圆外+Y 方向进给一步；

按照上述方法步进，就可以走出类似目标圆弧轮廓出来。整个过程如下图所示：

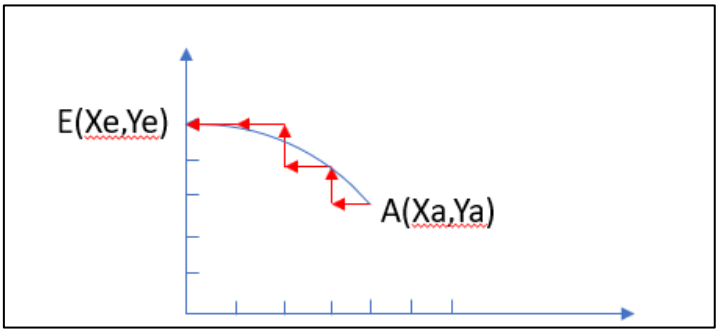


图 19.2.2.1 第一象限逆圆弧插补的过程图

图中所走的一步看着是比较夸张的，实际步进电机走的一步单位向量是非常小的，比头发丝的直径都还要小，所以实际运动中看着是非常顺滑的，并不会像图中这样棱角分明。

19.2.3 偏差计算

坐标进给之后得到新的动点坐标，接着就是计算新的偏差值，这里我们会对偏差判别式进行化简，由已知偏差判别式为： **$F = Xi^2+Yi^2-R^2$** ，假设：

- 当动点 P(Xi,Yi)在圆弧上或圆弧外，此时 F ≥ 0，需向 X 轴负方向进给一步(-X),

此时新加工点 P 的位置:X = Xi-1，Y = Yi，此时加工偏差为：

$Fi+1 = (Xi-1)^2+Yi^2-R^2$ → $Fi+1 = Xi^2+Yi^2-R^2 -2Xi+1$ → **$Fi+1 = F-2Xi+1$**

- 当动点 P 在圆弧内侧的话，此时 F < 0，需向 y 轴正向进给一步(+Y)，

此时新加工点 P 的位置:X = Xi，Y = Yi+1，此时加工偏差为：

$Fi+1 = Xi^2+(Yi+1)^2-R^2$ → $Fi+1 = Xi^2+Yi^2-R^2 +2Yi+1$ → **$Fi+1 = F-2Yi+1$**

经过上述化简后的判别式比原式更简单，运算速度也更快；从式中可以看出，偏差 Fi+1 的值只跟上一步进给的偏差 Fi 和上一步的动点坐标值 Pi 有关。注意：直线插补的偏差计算是不需要动点坐标的，这是圆弧插补与直线插补的区别。

19.2.4 终点判别

最后就是终点判别，插补到何时停止呢？常用的终点判别有三种，分别是：总步长法、终点坐标发。投影法，与直线插补章节介绍一致，这里就不再赘述。

我们这里使用的是总步长法。所以在插补处理开始之前，先设置一个**总步长计数器 n** ：

即： **$n = |Xe - Xa| + |Ye - Ya|$**

其中，圆弧终点坐标 E(Xe,Ye)，起点坐标 A(Xa,Ya)，每当步进一步计数器 n 就减一，直至 n = 0 时，插补结束。

19.3 任意象限圆弧插补原理

前面讲解的都是以第一象限逆圆弧为例，推导的偏差计算公式也只适合第一象限逆圆弧，实际应用中还需要考虑运动方向在 4 个象限内的顺逆圆弧情况，这个就比较复杂，所以接着我们来讲解下任意象限顺逆圆弧的情况。

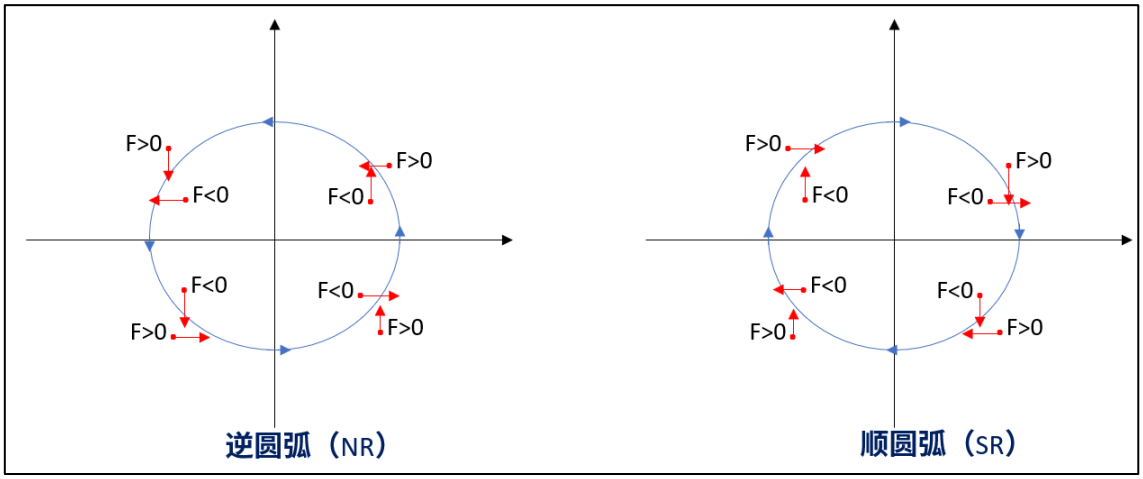


图 19.3.1 任意象限顺逆圆弧的情况

根据前面的逆圆弧差插补思路，我们可以推导第一象限顺圆弧的偏差判别，坐标进给以及偏差计算公式，如下表:

线形	$F_i \geq 0$			$F_i < 0$		
SR1	进给	新动点坐标	偏差计算	进给	新动点坐标	偏差计算
	-Y	$X_{i+1}= X_i$ $Y_{i+1}= Y_i-1$	$F_{i+1} = F_i-2Y_i+1$	+X	$X_{i+1}= X_i+1$ $Y_{i+1}= Y_i$	$F_{i+1} = F_i-2X_i+1$

表 19.3.1 第一象限顺圆弧推导

由表 19.3.1 可知，第一象限的顺圆弧进给方向与第一象限逆圆弧的相反，两个方向的偏差计算公式的形式是不变的，只是符号和对应的 X，Y 轴发生变化。

其他的象限推导方式也是很类似的，我可以依据上面的推导方法，推导出所有象限的顺逆圆弧的情况，我们将顺圆弧的 1、2、3、4 象限定义为：SR1、SR2、SR3、SR4，将逆圆弧的第 1、2、3、4 象限定义为：NR1、NR2、NR3、NR4，一共就有八种情况，我们这里直接进行归纳，如下表：

线形	Fi ≥ 0		Fi < 0	
	坐标进给	偏差计算	坐标进给	偏差计算
SR1	-Y	Fi+1 = Fi-|2Yi|+1 | Yi+1| = |Yi| - 1	+X	Fi+1 = Fi+|2Xi|+1 | Xi+1| = |Xi| + 1
NR2	-Y		-X	
SR3	+Y		-X	
NR4	+Y		+X	
NR1	-X	Fi+1 = Fi-|2Xi|+1 | Xi+1| = |Xi| - 1	+Y	Fi+1 = Fi+|2Yi|+1 | Yi+1| = |Yi| + 1
SR2	+X		+Y	
NR3	+X		-Y	
SR4	-X		-Y	

表 19.3.2 任意象限顺逆圆弧情况汇总表

可以将表中数据主要分为两组：第一组（SR1、NR2、SR3、NR4）和第二组（NR1、SR2、NR3、SR4），各组之间有共同点如下：

第一组的共同点：

当 Fi≥0 时， Y 轴进给， SR1、NR2 走-Y 方向， SR3、NR4 走 +Y 方向；

当 Fi＜0 时， X 轴进给， SR1、NR4 走 +X 方向， NR2、SR3 走-X 方向。

第二组的共同点：

当 Fi≥0 时， X 轴进给， NR1、SR4 走-X 方向， SR2、NR3 走 +X 方向；

当 Fi＜0 时， Y 轴进给， NR1、SR2 走 +Y 方向， NR3、SR4 走-Y 方向。

偏差的计算公式格式是不变的，只是符号的正负以及 X，Y 的区别，所以在代码实现部分需要去判断当前所要走的圆弧属于第几象限以及顺逆方向，在根据表 19.3.2 代入各自所对应的偏差判别式进行偏差计算即可。任意象限的原理部分就讲解完了，大家可以自行去推导看看！

19.4 任意象限圆弧插补实验

19.4.1 硬件设计

1、例程功能

本实验使用步进电机接口三控制 X 轴步进电机和接口四控制 Y 轴步进电机，当按下 KEY0 控制步进电机开始圆弧插补运动。注意驱动器设置：1.0A，8 细分，也就是旋转一圈所需要 1600 个脉冲数； LED0 作为程序运行状态指示灯，闪烁周期 200ms。

2、硬件资源

硬件连接部分与直线插补实验完全相同，这里就不再赘述，如不清楚可查看直线插补实验章节。

19.4.2 程序设计

19.4.2.1 任意象限圆弧插补的配置步骤

1）初始化定时器

初始化定时器通道 IO，设置 ARR、PSC，计数方式以及脉冲输出模式等

2）设置圆弧方向

需提前设置好是顺圆弧还是逆圆弧

3）电机旋转方向判定

实现任意象限圆弧插补，需提前判断所属象限，设置电机旋转方向

4）编写圆弧插补算法

实现插补第一步，偏差判别，坐标进给，偏差计算并开启脉冲输出与中断使能

5）编写中断服务函数

计算动点新坐标，实现除第一步以外的其他步数的偏差判别、坐标进给、偏差计算、终点判别

19.4.2.2 程序流程图

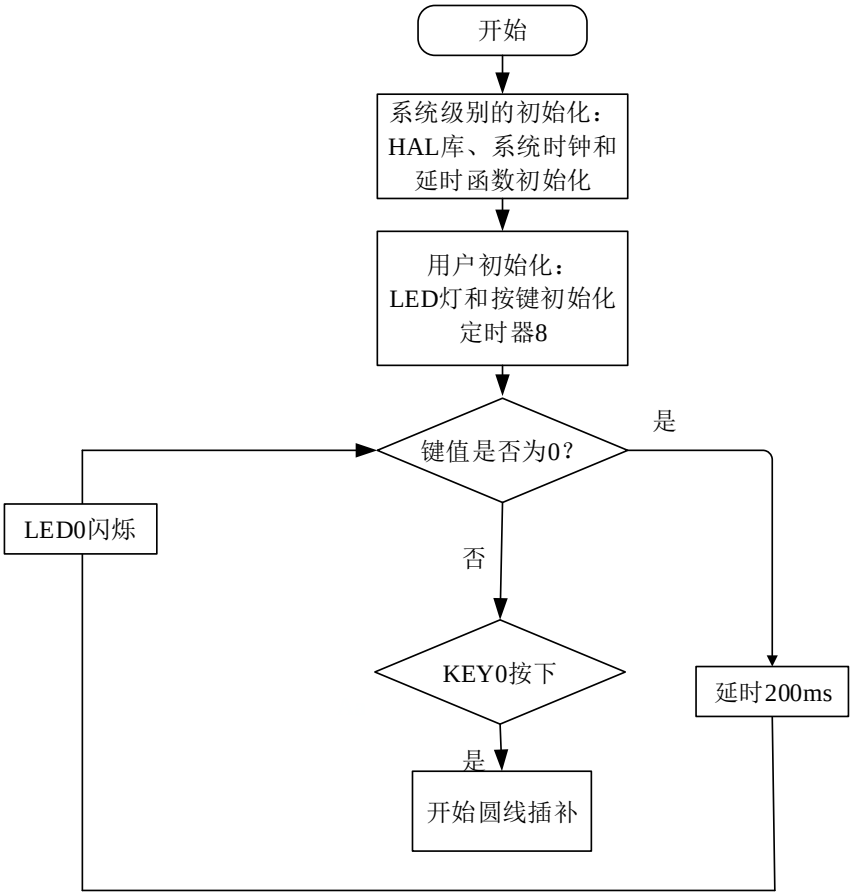


图 19.4.2.2.1 圆弧插补流程图

19.4.2.3 程序解析

本实验在步进电机直线插补实验的基础上添加代码，这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。与直线插补相同部分不再赘述，主要讲解圆弧插补相关部分，首先看 stepper_motor.h 中添加了哪部分：

```
#define AXIS_X                  0   /* X 轴标号 */
#define AXIS_Y                  1   /* Y 轴标号 */

#define EN                      0   /* 失能脱机信号 */
#define OFF                     1   /* 使能脱机信号 */

#define FIRST_QUADRANT          1   /* 第一象限 */
#define SECOND_QUADRANT         2   /* 第二象限 */
#define THIRD_QUADRANT          3   /* 第三象限 */
```

```
#define FOURTH_QUADRANT     4   /* 第四象限 */

#define ARC                 1   /* 圆弧插补 */
```

主要就是添加了第 1、2、3、4 象限的宏定义，用于判断当前圆弧属于哪个象限。接着看下 stepper_motor.c 中的方向设置函数：

```
/**
 * @brief       设置圆弧插补方向
 * @param       coordsX,coordsY:x,y 坐标,dir: 圆弧的插补方向
 * @retval      无
 */
void setarcdir(int32_t coords_x,int32_t coords_y,int8_t dir)
 {
    g_pol_par.inter_dir = dir;
    if(g_pol_par.inter_dir == CCW)
    {
        if(coords_x > 0)          /*x > 0 起点在 X 轴的正半轴,顺时针 Y 轴进给方向是负的 */
        {
            if(coords_y>=0)       /*  如果 y > 0,则是第一象限 */
            {
                g_pol_par.qua_points = FIRST_QUADRANT;
                g_pol_par.x_dir = CCW;
                g_pol_par.y_dir = CW;
                sign_dir[AXIS_X] = -1;
                sign_dir[AXIS_Y] = 1;
                ST3_LINE_DIR(CCW,AXIS_X);
                ST3_LINE_DIR(CW,AXIS_Y);
            }
            else
            {
                g_pol_par.qua_points = FOURTH_QUADRANT;
                g_pol_par.x_dir = CW;     /*  y <= 0, 在第四象限 */
                g_pol_par.y_dir = CW;
                sign_dir[AXIS_X] = 1;
                sign_dir[AXIS_Y] = 1;
                ST3_LINE_DIR(CW,AXIS_X);
                ST3_LINE_DIR(CW,AXIS_Y);
            }
        }
        else if(coords_x < 0)             /*  X 轴的负半轴 */
        {
            if(coords_y <= 0)             /*  y < 0,则是第三象限 */
            {
                g_pol_par.qua_points = THIRD_QUADRANT;
                g_pol_par.x_dir = CW;
                g_pol_par.y_dir = CCW;
                sign_dir[AXIS_X] = 1;
                sign_dir[AXIS_Y] = -1;
                ST3_LINE_DIR(CW,AXIS_X);
                ST3_LINE_DIR(CCW,AXIS_Y);
            }
            else                          /*  y >= 0,第二象限 */
            {
                g_pol_par.qua_points = SECOND_QUADRANT;
                g_pol_par.x_dir = CCW;
                g_pol_par.y_dir = CCW;
                sign_dir[AXIS_X] = -1;
                sign_dir[AXIS_Y] = -1;
                ST3_LINE_DIR(CCW,AXIS_X);
                ST3_LINE_DIR(CCW,AXIS_Y);
            }
        }
        else if(coords_x == 0)   /* x = 0,在 Y 轴上的特殊起点 */
        {
            if(coords_y>0)
```

```
        {
            g_pol_par.qua_points = SECOND_QUADRANT;
            g_pol_par.x_dir = CCW;
            g_pol_par.y_dir = CCW;
            sign_dir[AXIS_X] = -1;
            sign_dir[AXIS_Y] = -1;
            ST3_LINE_DIR(CCW,AXIS_X);
            ST3_LINE_DIR(CCW,AXIS_Y);
        }
        else
        {
            g_pol_par.qua_points = FOURTH_QUADRANT;
            g_pol_par.x_dir = CW;
            g_pol_par.y_dir = CW;
            sign_dir[AXIS_X] = 1;
            sign_dir[AXIS_Y] = 1;
            ST3_LINE_DIR(CW,AXIS_X);
            ST3_LINE_DIR(CW,AXIS_Y);
        }
    }
}
else /* CW 方向 */
{
    if(coords_x > 0)    /*  坐标 x > 0 起点在 X 轴的正半轴,顺时针 Y 轴进给方向是负的 */
    {
        if(coords_y >0) /*  如果 y > 0,则是第一象限 */
        {
            g_pol_par.qua_points = FIRST_QUADRANT;
            g_pol_par.x_dir = CW;
            g_pol_par.y_dir = CCW;
            sign_dir[AXIS_X] = 1;
            sign_dir[AXIS_Y] = -1;
            ST3_LINE_DIR(CW,AXIS_X);
            ST3_LINE_DIR(CCW,AXIS_Y);
        }
        else
        {
            g_pol_par.qua_points = FOURTH_QUADRANT;
            g_pol_par.x_dir = CCW;    /*  y <= 0, 在第四象限 */
            g_pol_par.y_dir = CCW;
            sign_dir[AXIS_X] = -1;
            sign_dir[AXIS_Y] = -1;
            ST3_LINE_DIR(CCW,AXIS_X);
            ST3_LINE_DIR(CCW,AXIS_Y);
        }
    }
    else if(coords_x < 0)            /*  X 轴的负半轴 */
    {
        if(coords_y < 0)             /*  y < 0,则是第三象限 */
        {
            g_pol_par.qua_points = THIRD_QUADRANT;
            g_pol_par.x_dir = CCW;
            g_pol_par.y_dir = CW;
            sign_dir[AXIS_X] = -1;
            sign_dir[AXIS_Y] = 1;
            ST3_LINE_DIR(CCW,AXIS_X);
            ST3_LINE_DIR(CW,AXIS_Y);
        }
        else                    /*  y >= 0,第二象限 */
        {
            g_pol_par.qua_points = SECOND_QUADRANT;
            g_pol_par.x_dir = CW;
            g_pol_par.y_dir = CW;
            sign_dir[AXIS_X] = 1;
            sign_dir[AXIS_Y] = 1;
```

```
            ST3_LINE_DIR(CW,AXIS_X);
            ST3_LINE_DIR(CW,AXIS_Y);
        }
    }
    else if(coords_x == 0)   /* x = 0,在 Y 轴上的特殊起点 */
    {
        if(coords_y>0)
        {
            g_pol_par.qua_points = FIRST_QUADRANT;
            g_pol_par.x_dir = CW;
            g_pol_par.y_dir = CCW;
            sign_dir[AXIS_X] = 1;
            sign_dir[AXIS_Y] = -1;
            ST3_LINE_DIR(CW,AXIS_X);
            ST3_LINE_DIR(CCW,AXIS_Y);
        }
        else
        {
            g_pol_par.qua_points = THIRD_QUADRANT;
            g_pol_par.x_dir = CCW;
            g_pol_par.y_dir = CW;
            sign_dir[AXIS_X] = -1;
            sign_dir[AXIS_Y] = 1;
            ST3_LINE_DIR(CCW,AXIS_X);
            ST3_LINE_DIR(CW,AXIS_Y);
        }
    }
  }
}
```

该函数有三个入口参数由用户设置，其中 coords_x 和 coords_y 为圆弧的起点坐标，通过该坐标判断当前圆弧属于哪一象限，dir 设置的是圆弧的顺逆方向，这样就可确定该圆弧属于哪个象限以及它的顺逆方向情况了，之后设置 XY 轴的步进电机方向，并且设置对应的偏差计算公式的符号位。所以该函数非常关键！接着看实现任意象限画圆弧函数：

```
/**
 * @brief       在 XOY 平面内画任意圆弧
 * @param       start_x_point,start_y_point:  分别是起点坐标 X,Y
 * @param       stop_x_point,stop_y_point:  分别是终点坐标 X,Y
 * @param       speed:  速度值
 * @param       dir:    圆弧的方向
 * @retval      无
 */
void arc_incmove(int32_t start_x_point,int32_t start_y_point,int32_t
stop_x_point,int32_t stop_y_point,uint32_t speed,int8_t dir)
{
    if(g_motor_sta == STATE_RUN)                /*  当前电机正在运动 */
        return ;

    /* 不符合圆的坐标方程 */
    if( (pow(start_x_point,2)+pow(start_y_point,2)) !=
    (pow(stop_x_point,2)+pow(stop_y_point,2)))  /* 需要满足半径一致才是一个圆 */
        return ;
    g_pol_par.moving_mode = ARC;                /* 圆弧标志 */

    g_pol_par.f_e = 0;                          /* 偏差方程置零 */
    g_pol_par.start_point[AXIS_X] = start_x_point;
    g_pol_par.start_point[AXIS_Y] = start_y_point;
    g_pol_par.end_x = stop_x_point;
    g_pol_par.end_y = stop_y_point;
     /* 设置电机做逆时针圆弧轨迹的运动方向 */
    setarcdir(g_pol_par.start_point[AXIS_X],g_pol_par.start_point[AXIS_Y],dir);
     /* 设置圆弧插补方向 */
     /* 计算总的步数 */
    g_pol_par.end_pulse = abs((stop_y_point-start_y_point))+ abs((stop_x_point-
```

```
    start_x_point));  /*  从起点到终点的脉冲数 */
    /* 起点坐标 x = 0,说明起点位于 y 轴上,此时往 X 轴进给误差会减少 */
    if(g_pol_par.start_point[AXIS_X] == 0)
    {
        g_pol_par.act_axis = AXIS_X;                              /*  第一步给 X 轴 */
        /* 根据圆弧方向决定向 X 轴进给的偏差方程 */
        g_pol_par.f_e = g_pol_par.f_e +
        sign_dir[AXIS_X]*g_pol_par.start_point[AXIS_X]*2 + 1; /* 偏差方程的计算 */
    }
    else
    {
        g_pol_par.act_axis = AXIS_Y;                              /* 第一步给 Y 轴 */
        g_pol_par.f_e = g_pol_par.f_e +
         sign_dir[AXIS_Y]*g_pol_par.start_point[AXIS_Y]*2 + 1;/* 偏差方程的计算 */
    }

    ST3_LINE_EN(EN,AXIS_X);
    ST3_LINE_EN(EN,AXIS_Y);
    __HAL_TIM_SET_COMPARE(&g_atimx_handle,st_motor[AXIS_X].pulse_channel,
                          speed);     /* 设置通道的比较值 */
    __HAL_TIM_SET_COMPARE(&g_atimx_handle,st_motor[AXIS_Y].pulse_channel,
                          speed);
    __HAL_TIM_SET_AUTORELOAD(&g_atimx_handle,speed*2);
    TIM_CCxChannelCmd(ATIM_TIMX_PWM,
    st_motor[g_pol_par.act_axis].pulse_channel,
     TIM_CCx_ENABLE);
    HAL_TIM_Base_Start_IT(&g_atimx_handle);     /* 使能定时器以及开启更新中断 */
    g_motor_sta = STATE_RUN;                    /*  标记电机正在运动 */
}
```

该函数实现计算总步长和判断入口参数起点与终点是否构成同一个圆的圆弧，为同一个圆才可继续否则直接退出；然后开始第一步的偏差判别，坐标进给，并会调用 SetArcDir 函数确定符号正负，进行偏差计算，最后就是开启定时器脉冲输出以及中断使能。然后其余步数的计算在中断回调函数里边实现，接着看下中断回调函数的逻辑：

```
/**
 * @brief       定时器中断回调函数
 * @param       htim :  定时器句柄
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    __IO uint32_t axis = 0;          /*  进给轴 */
    axis = g_pol_par.act_axis;       /*  当前进给轴 */

    /* 判断是否到达终点或者还没开始运动 */
    if(g_pol_par.end_pulse == 0)
        return;
    /* 计算新的动点坐标 */
    if(g_pol_par.act_axis == AXIS_X)
    {
        if(g_pol_par.x_dir == CCW)
        {
            g_pol_par.start_point[AXIS_X]--;
        }
        else
        {
            g_pol_par.start_point[AXIS_X]++;
        }
    }
    if(g_pol_par.act_axis == AXIS_Y)
    {
        if(g_pol_par.y_dir == CCW)
        {
            g_pol_par.start_point[AXIS_Y]--;
```

```
        }
        else
        {
            g_pol_par.start_point[AXIS_Y]++;
        }
    }

    /* 根据进给方向 更新坐标值 */
    if(g_pol_par.moving_mode == ARC)
    {
        /* 根据上一次的偏差判断下一步进给方向,同时计算下一次的偏差 */
        if(g_pol_par.inter_dir == CCW)       /* 插补方向:逆时针圆弧 */
        {
            if(g_pol_par.f_e < 0)  /* 偏差方程 < 0 ,当前位置位于圆弧内侧,应向圆外进给*/
            {
                /* 第二和第四象限,当偏差<0 时都是向 X 轴进给 */
                if( (g_pol_par.qua_points == SECOND_QUADRANT) ||
                      (g_pol_par.qua_points == FOURTH_QUADRANT) )
                {
                    g_pol_par.act_axis = AXIS_X;
                }
                else                /*  在第一和第三象限,偏差<0,都是向 Y 轴进给 */
                {
                    g_pol_par.act_axis = AXIS_Y;
                }
            }
            /* 偏差方程 >= 0 ,当前位置位于圆弧外侧,应向圆内进给 */
            else if(g_pol_par.f_e >= 0)
            {
                if( (g_pol_par.qua_points == SECOND_QUADRANT) ||
                   (g_pol_par.qua_points == FOURTH_QUADRANT) )
                {
                    g_pol_par.act_axis = AXIS_Y;
                }
                else
                {
                    g_pol_par.act_axis = AXIS_X;
                }
            }
        }
        else
        {
            if(g_pol_par.f_e < 0) /* 偏差方程 < 0 ,当前位置位于圆弧内侧,应向圆外进给 */
            {
                /* 第一和第三象限,当偏差<0 时都是向 X 轴进给 */
                if( (g_pol_par.qua_points == FIRST_QUADRANT) ||
                   (g_pol_par.qua_points == THIRD_QUADRANT) )
                {
                    g_pol_par.act_axis = AXIS_X;
                }
                else                /* 在第二和第四象限,偏差<0,都是向 Y 轴进给 */
                {
                    g_pol_par.act_axis = AXIS_Y;
                }
            }
            /* 偏差方程 >= 0 ,说明当前位置位于圆弧外侧,应向圆内进给 */
            else if(g_pol_par.f_e >= 0)
            {
                if( (g_pol_par.qua_points == FIRST_QUADRANT) ||
                   (g_pol_par.qua_points == THIRD_QUADRANT) )
                {
                    g_pol_par.act_axis = AXIS_Y;
                }
                else
                {
```

```
                g_pol_par.act_axis = AXIS_X;
            }
        }
    }
    /* 计算当前坐标与目标曲线的偏差 */
    g_pol_par.f_e = g_pol_par.f_e + 2*sign_dir[g_pol_par.act_axis]*
                    g_pol_par.start_point[g_pol_par.act_axis] + 1;
}
/* 判断是否需要跟换进给轴 */
if(axis != g_pol_par.act_axis)
{
    TIM_CCxChannelCmd(ATIM_TIMX_PWM, st_motor[axis].pulse_channel,
                      TIM_CCx_DISABLE);
    TIM_CCxChannelCmd(ATIM_TIMX_PWM,
     st_motor[g_pol_par.act_axis].pulse_channel, TIM_CCx_ENABLE);
}
/* 终点判别:总步长 */
g_pol_par.end_pulse--;
if(g_pol_par.end_pulse == 0)
{
    g_motor_sta = STATE_STOP;                    /*  到达终点 */
    /* 关闭当前轴输出 */
    TIM_CCxChannelCmd(ATIM_TIMX_PWM,
     st_motor[g_pol_par.act_axis].pulse_channel, TIM_CCx_DISABLE);
    HAL_TIM_Base_Stop_IT(&g_atimx_handle);       /*  停止定时器 */
}
}
```

中断回调函数里边首先记录上一步的进给活动轴，接着计算新的动点坐标用于下一步的偏差计算，接着进行除了第一步以外的其他步数的偏差判别、坐标进给、偏差计算，比较当前活动轴与上一步活动轴是否一致，若不一致则更换输出通道，然后总步数减一，最后判断总步长是否为 0，是的话则插补停止，否则重复以上步骤。

在 main 函数里面编写如下代码：

```
int main(void)
{
    uint8_t key,t;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    stepper_init(0xFFFF, 8 - 1);                /* 定时器初始化 */

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"Stepper Motor Test",   g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:Run Once",        g_point_color);
    printf("圆弧插补实验，按 KEY0 开始插补\r\n");

    while (1)
    {
        key = key_scan(0);
        if(key == KEY0_PRES)                    /* 按下 KEY0 开始插补 */
        {
            arc_incmove(800 * 5, 0, 0, -800 * 5, 2000, CW);      /* 第四象限圆弧 */
            while(g_motor_sta);
            arc_incmove(0, -800 * 5, -800 * 5, 0, 2000, CW);     /* 第三象限圆弧 */
            while(g_motor_sta);
            arc_incmove(-800 * 5, 0, 0, 800 * 5, 2000, CW);      /* 第二象限圆弧 */
            while(g_motor_sta);
```

```
            arc_incmove(0, 800 * 5, 800 * 5, 0, 2000, CW);          /* 第一象限圆弧 */
            while(g_motor_sta);
        }
        t++;
        if(t % 20 == 0)
        {
            LED0_TOGGLE();                                          /*LED0(红灯) 翻转*/
        }
        delay_ms(10);
    }
}
```

mian 函数主要就是初始化一些外设包括串口、定时器初、LCD 等等，串口和 LCD 都会显示提示信息，当按下 KEY0 之后就会分别开启四个象限的圆弧插补运动。最后可以看出走的轨迹是一个顺时针画的圆。大家如果要改变圆弧半径，可以自己调整 Arc_IncMove 入口参数的起点以及终点坐标，注意坐标点必须满足在同一个圆上。

19.4.3 下载验证

同直线插补的硬件连接完全一致，需使用两个丝杆滑台组成一个标准的两轴 X-Y 平台，硬件连接好之后下载程序到开发板，液晶屏和串口都会显示提示信息，当我们按下按键 KEY0，将会开启圆弧插补运动，最后可以看到我们的源码实现了一个顺时针画的圆。

图 19.4.3.1 串口提示信息

第 20 章 步进电机闭环系统（位置环）

本章我们将学习步进电机的闭环系统。经过我们前面的学习，我们知道步进电机它拥有非常优秀的开环能力，控制器不需要传感器反馈电机信号，只要在不超频且不超载的情况下，脉冲数与旋转位置成正比关系。那我们为什么还要引入闭环系统呢？因为在实际的应用中单独靠开环还是有很多不确定因素，比如步进电机的机械老化导致的丢步或堵转，还有一些外部因素比如：突然给电机加了一个比较大的负载或者快速启停都是有可能出现丢步、过冲甚至堵转现象，控制器是无法知晓并纠正偏差的，这些情况在一些要求高精度的场合是不允许的，否则有可能导致严重后果，所以我们引入闭环系统使控制器可以及时发现偏差并纠正偏差。

本章分为如下几个小节：

20.1 步进电机闭环系统组成
20.2 步进电机闭环系统原理
20.3 步进电机位置环控制实现（增量式/位置式）

20.1 步进电机闭环系统组成

同直流有刷电机的闭环系统类似，步进电机的闭环系统的组成由：**步进电机+编码器构成**，编码器反馈步进电机的实际旋转位置给控制器，这样就可以控制器知晓是否有按要求到达指定目标位置，当有偏差就可及时纠偏了。

接着来看总结的一些闭环系统与开环系统之间的优缺点：

系统	优点	缺点
步进电机开环系统	价格较低，控制简单	无反馈信号 无法纠偏 可靠性较低
步进电机闭环系统	精度更高可以消除位置误差 明显降低噪音	价格较贵， 控制较为复杂（PID 控制）

表 20.1.1 开/闭环系统优缺点对比

20.2 步进电机闭环系统原理

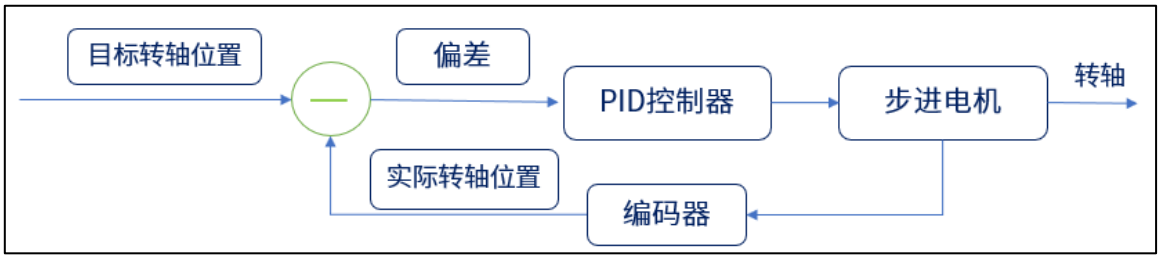


图 20.2.1 步进电机闭环系统控制原理

图 20.2.1 为步进电机闭环系统的整体控制流程，其中编码器作为反馈通道，反馈步进电机的实际位置，系统将实际位置与目标位置进行比较，计算差值，再把偏差值代入到 PID 控制器中，之后控制器输出期望值，最后作用于步进电机使其旋转至指定位置停止！这就是位置闭环控制。

20.3 步进电机位置闭环控制实现（增量式/位置式）

20.3.1 硬件设计

1、例程功能

- 本实验使用步进电机接口三控制
- 当按下 KEY0 一次启动步进电机转动到目标位置，在按一次 KEY0 关闭电机转动，按下 KEY1 增加目标位置，按下 KEY2 减少目标位置；
- 可通过串口 1 即 USB_TTL 接口连接正点原子 PID 调试助手，查看 PID 波形；
- 注意驱动器使用 8 细分，也就是旋转一圈所需要 1600 个脉冲数；
- LED0 作为程序运行状态指示灯，闪烁周期 200ms。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器： TIM8_CH3：PI7，对应步进电机接口三
TIM3_CH1 : PC6（编码器接口 1）
TIM3_CH2 : PC7（编码器接口 2）
TIM6 定时采样
方向引脚：DIR3: PB2
脱机引脚：EN3: PF11
4）步进电机
5）步进电机驱动器
6）12-50V 的 DC 电源

20.3.2 程序设计

20.3.2.1 步进电机位置闭环控制的配置步骤

1）初始化定时器
初始化定时器通道 IO，设置 ARR、PSC，计数方式以及脉冲输出模式等
2）初始化编码器接口
初始化编码器接口模式定时器三的通道一和通道二，用来采集反馈信号
3）PID 算法实现
实现增量式/位置式 PID 算法，调整好比例 P，积分 I，微分 D 的参数
4）位置闭环运动控制
设置电机旋转方向，以及将 PID 控制器输出的期望值，作用于电机运动
5）编写中断服务函数
编码器溢出计数，定时控制电机，设置比较值

20.3.2.2 程序流程图

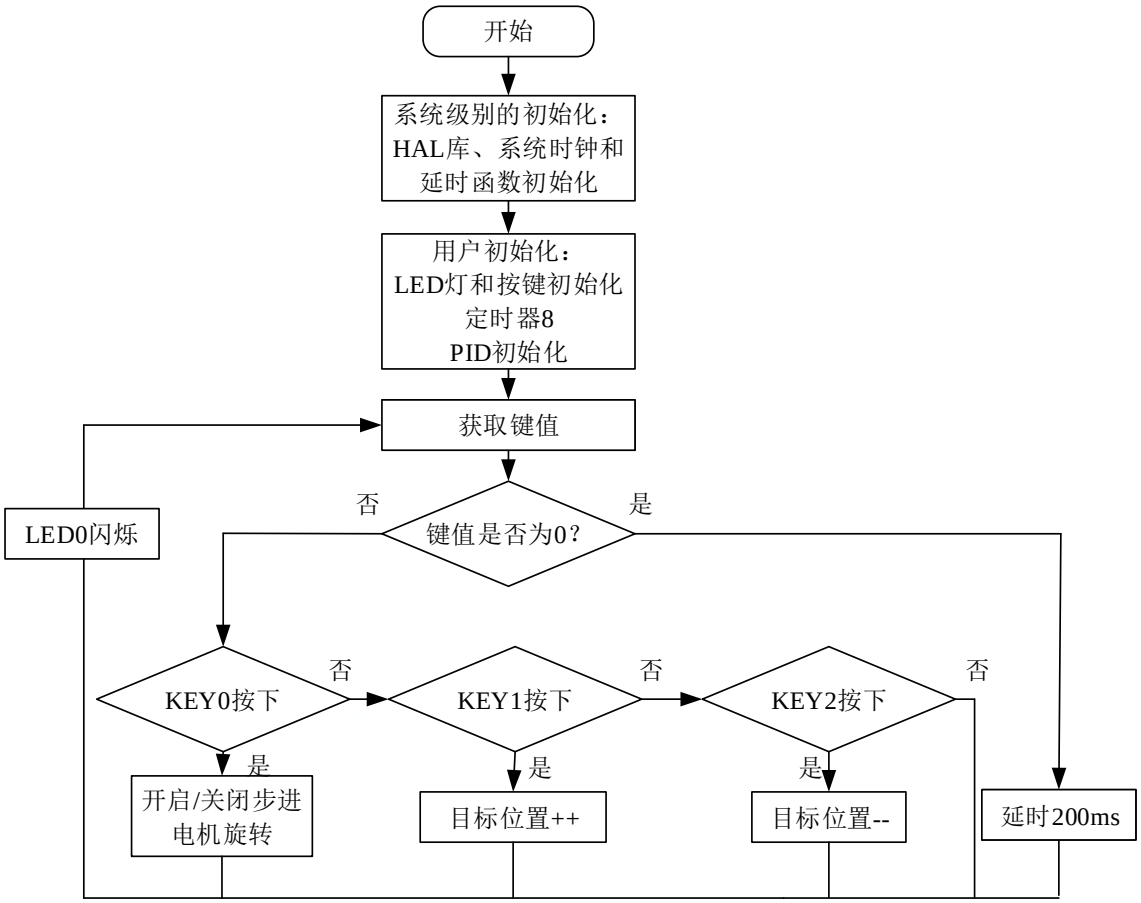


图 20.3.2.2.1 闭环控制流程图

19.4.2.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。首先先看下 stepper_motor.h 头文件：

```
#define PULSE_REV       1600                /* 每圈脉冲数*/
#define TIM_FREQ        168000000U          /* 定时器主频 */
#define T1_FREQ         (TIM_FREQ/168)      /* 频率 ft 值*/
/* 对定时器的频率做单位换算,避免数值太大溢出 */
#define FREQ_UINT       (T1_FREQ/(SECOND/SAMPLING_PERIOD))
#define ENCODER_SPR     (float)(600*4)    /* 编码器单圈线数*4 倍频 */
#define MPR             5                 /* 步进电机旋转一圈,丝杠的距离;单位：mm/r
*/
#define PPM             (ENCODER_SPR/MPR)/* 每 mm 内编码器的脉冲数;单位:Pules/mm */
#define MPP             ((float)(MPR)/ENCODER_SPR)  /* 编码器单步步进距离 */
/* 编码器和步进电机驱动器的比值（表示每个编码器输出线数对应的步进电机脉冲数） */
#define FEEDBACK_CONST  (float)(PULSE_REV/ENCODER_SPR)


//电机参数结构体
typedef struct
{
    uint8_t state;      /*电机状态*/
    uint8_t dir;        /*电机方向*/
    float speed;        /*电机实际速度*/
    float setspeed;     /*电机实际速度*/
    int32_t location;   /*电机位置*/
```

```
} Motor_TypeDef;

/*****************************************************************************
*************/
/* 步进电机引脚定义*/

#define STEPPER_MOTOR_1       1                 /* 步进电机接口序号 */
#define STEPPER_MOTOR_2       2
#define STEPPER_MOTOR_3       3
#define STEPPER_MOTOR_4       4
/*     步进电机方向引脚定义     */

#define STEPPER_DIR1_GPIO_PIN                   GPIO_PIN_14
#define STEPPER_DIR1_GPIO_PORT                  GPIOF
#define STEPPER_DIR1_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PC 口时钟使能 */

#define STEPPER_DIR2_GPIO_PIN                   GPIO_PIN_12
#define STEPPER_DIR2_GPIO_PORT                  GPIOF
#define STEPPER_DIR2_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PC 口时钟使能 */

#define STEPPER_DIR3_GPIO_PIN                   GPIO_PIN_2
#define STEPPER_DIR3_GPIO_PORT                  GPIOB
#define STEPPER_DIR3_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)    /* PC 口时钟使能 */

#define STEPPER_DIR4_GPIO_PIN                   GPIO_PIN_2
#define STEPPER_DIR4_GPIO_PORT                  GPIOH
#define STEPPER_DIR4_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOH_CLK_ENABLE(); }while(0)    /* PC 口时钟使能 */

/*     步进电机脱机引脚定义     */

#define STEPPER_EN1_GPIO_PIN                    GPIO_PIN_15
#define STEPPER_EN1_GPIO_PORT                   GPIOF
#define STEPPER_EN1_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PC 口时钟使能 */

#define STEPPER_EN2_GPIO_PIN                    GPIO_PIN_13
#define STEPPER_EN2_GPIO_PORT                   GPIOF
#define STEPPER_EN2_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PC 口时钟使能 */

#define STEPPER_EN3_GPIO_PIN                    GPIO_PIN_11
#define STEPPER_EN3_GPIO_PORT                   GPIOF
#define STEPPER_EN3_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)    /* PC 口时钟使能 */

#define STEPPER_EN4_GPIO_PIN                    GPIO_PIN_3
#define STEPPER_EN4_GPIO_PORT                   GPIOH
#define STEPPER_EN4_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOH_CLK_ENABLE(); }while(0)    /* PC 口时钟使能 */
```

上面主要包含两部分，第一部分的宏定义是步进电机本身的参数和闭环控制所需要用到的一些参数，包含步进电机单圈脉冲数，定时器频率，其中宏 **FEEDBACK_CONST** 是细分后步进电机单圈脉冲数与编码器单圈脉冲数的比值，因为在闭环系统中，反馈的 PID 期望值为编码器的脉冲数，我们需要转换成与步进电机相关脉冲数；第二部分为四个步进电机接口的相关宏定义。接着看下 PID 头文件 pid.c：

```
/* PID 相关参数 */

#define  INCR_LOCT_SELECT  0     /*0：选择位置式  1：增量式控制*/
#if INCR_LOCT_SELECT
/*定义 PID 参数相关宏*/
```

```
#define  L_KP      0.5f          /* P 参数*/
#define  L_KI      0.05f         /* I 参数*/
#define  L_KD      0.00f         /* D 参数*/

#define SMAPLSE_PID_SPEED  20   /*采样率 单位 ms*/
#else
#define  L_KP      0.5f          /* P 参数*/
#define  L_KI      0.05f         /* I 参数*/
#define  L_KD      0.00f         /* D 参数*/

#define SMAPLSE_PID_SPEED  20   /*采样率 单位 ms*/
#endif

/*定义位置 PID 参数相关宏*/
/*PID 结构体*/
typedef struct
{
    __IO float  SetPoint;        /*设定目标 */
    __IO float  ActualValue;     /*实际值*/
    __IO float  SumError;        /*误差累计*/
    __IO float  Proportion;      /*比例常数 P*/
    __IO float  Integral;        /*积分常数 I*/
    __IO float  Derivative;      /*微分常数 D*/
    __IO float  Error;           /*Error[-1]*/
    __IO float  LastError;       /*Error[-1]*/
    __IO float  PrevError;       /*Error[-2]*/
    __IO float  IngMin;
    __IO float  IngMax;
    __IO float  OutMin;
    __IO float  OutMax;
} PID_TypeDef;
```

首先通过宏 **INCR_LOCT_SELECT** 设置 0 或 1 选择位置式还是增量式 PID 算法，接着通过不同的宏定义选择不同的 P、I、D 参数以及采样时间，PID_TypeDef 定义的是 PID 的结构体，主要参与 PID 算法计算相关。接着看下 PID 初始化以及 PID 的控制算法：

```
/*
 * @brief       初始化 PID 参数
 * @param       无
 * @retval      无
 */
void pid_init(void)
{
       /*位置环初始化*/
    g_location_pid.SetPoint=(float)(50*PPM);     /* 设定目标 Desired Value*/
    g_location_pid.ActualValue=0.0;              /* 设定目标 Desired Value*/
    g_location_pid.SumError=0.0;                 /* 积分值*/
    g_location_pid.Error=0.0;                    /* Error[1]*/
    g_location_pid.LastError=0.0;                /* Error[-1]*/
    g_location_pid.PrevError=0.0;                /* Error[-2]*/
    g_location_pid.Proportion=L_KP;              /* 比例常数 Proportional Const*/
    g_location_pid.Integral= L_KI;               /* 积分常数 Integral Const*/
    g_location_pid.Derivative=L_KD;              /* 微分常数 Derivative Const*/
    g_location_pid.IngMax = 20;
    g_location_pid.IngMin = -20;
    g_location_pid.OutMax = 150;                 /* 输出限制 */
    g_location_pid.OutMin = -150;
}
/**
  * 函数名称：位置闭环 PID 控制设计
  * 输入参数：当前控制量
  * 返 回 值：目标控制量
  * 说    明：无
```

```
  */
int32_t increment_pid_ctrl(PID_TypeDef *PID,float Feedback_value)
{
     PID->Error = (float)(PID->SetPoint - Feedback_value);   /*速度档位偏差*/
#if  INCR_LOCT_SELECT
    /*E[k]项*/
    PID->ActualValue += (PID->Proportion * (PID->Error - PID->LastError))
                     + (PID->Integral * PID->Error)     /* E[k-1]项*/
                     + (PID->Derivative * (PID->Error - 2 * PID->LastError +
                                           PID->PrevError));  /* E[k-2]项*/
    PID->PrevError = PID->LastError;                    /* 存储误差，用于下次计算*/
    PID->LastError = PID->Error;
#else
    PID->SumError += PID->Error;
    if(PID->SumError > PID->IngMax)
    {
        PID->SumError = PID->IngMax;
    }
    else if(PID->SumError < PID->IngMin)
    {
        PID->SumError = PID->IngMin;
    }

    PID->ActualValue = (PID->Proportion * PID->Error)               /*E[k]项*/
                     + (PID->Integral * PID->SumError)              /*E[k-1]项*/
                     + (PID->Derivative * (PID->Error - PID->LastError));
                                                                    /*E[k-2]项*/
    PID->LastError = PID->Error;
#endif
    if(PID->ActualValue > PID->OutMax)
    {
        PID->ActualValue = PID->OutMax;
    }
    else if(PID->ActualValue < PID->OutMin)
    {
        PID->ActualValue = PID->OutMin;
    }
    return ((int32_t)(PID->ActualValue));               /*返回实际控制数值*/

}
```

初始化 PID 结构体成员，赋值初始值，其中 L_KP、L_KI、L_KD 需要不断调试，选择可以让电机最为稳定可靠的数据，increment_pid_ctrl 函数为 PID 控制器，他的入口参数第一个为目标值，第二个为反馈回来的实际值，然后通过宏 INCR_LOCT_SELECT 选择相应公式并计算输出期望值。接着来看步进电机的闭环控制代码：

```
/*PID 闭环系统代码*/

Motor_TypeDef  g_step_motor;            /*电机参数变量*/
volatile uint16_t step_angle = 65535;  /*设置的步进角度*/
/**
 * @brief 函数功能: 步进电机运动控制
 * @param 输入参数: Dir:步进电机运动方向 0:反转 1 正转
          Frequency:步进电机频率,0:停止
 * @retval 返 回 值: void
 */
/* location_m 如果这个值设置 100，那就是 20ms 的采样频率内要输出 100 个脉冲信号 */
void stepper_motion_ctrl(uint8_t Dir, uint16_t location_m)
{
    float Step_Delay = 0.0;  /* 步进延时 */

    if(location_m == 0)
    {
         /* 当期望值与目标值一致时，代表已到指定位置，停止输出 */
```

```
        HAL_TIM_OC_Stop_IT(&g_atimx_handle,ATIM_TIMX_PWM_CH3);
        step_angle = 0;          /* 清空数据 */
    }
    else
    {
        if(Dir==CW)
            ST3_DIR(CW);
        else
            ST3_DIR(CCW);

/* 经过 PID 计算得到的结果是编码器的输出期望值,
 * 将其转换为步进电机所需的脉冲数（ 编码器期望值 *  （步进电机一圈所需脉冲数 / 编码器一圈计数
 * 值））
 */
         /*单个脉冲的时间宽度，单位： ms*/
        Step_Delay=(float)(SMAPLSE_PID_SPEED    /(location_m*FEEDBACK_CONST));
        /*ms 转成 s Step_Delay =Step_Delay/1000;*/
        /*
           T : 单个脉冲所需的时间
           c : 需要求解的比较值
           f : 定时器计数频率
           T  = c *(1/f)
           Step_Delay/1000 = c*1/1000000 (单位是 s)
           c = Step_Delay*1000  求出整一个脉冲的计数值，需要除以 2 一个完整的脉冲翻转 2 次
           所要的计数值：c/2 = Step_Delay*1000/2 = Step_Delay*500
        */
        step_angle = (uint16_t)(Step_Delay*500);/* 这里除以 2,半周期翻转一次 */
        HAL_TIM_OC_Start_IT(&g_atimx_handle,ATIM_TIMX_PWM_CH3);
    }
}
```

上述代码是整个步进电机位置闭环控制的核心代码，首先函数 stepper_motion_ctrl 有两个入口参数，第一个为旋转方向，第二个为经过 PID 计算输出的期望值，也就是编码器的期望值，注意这个函数是每 20ms 在中断里边调用一次，即 20ms 执行一次该函数。函数内部操作如下:

① 判断期望值是否为 0。如果为 0，代表已到目标位置，关闭定时器脉冲输出；

② 不为 0 时, 代表还未到达目标位置，需继续驱动步进电机旋转，所以首先设置旋转方向，然后将编码器的期望值转换成步进电机的脉冲数期望值，并求出单个脉冲需要的时间，知道了单个脉冲时间就可以通过 T = c *(1/f)，求出比较值 c 的值，然后能定时器输出以及输出中断。接着将求得的比较值 c 作用到定时器上，在输出比较中断回调函数中实现调用：

```
/**
  * @brief  定时器比较中断
  * @param  htim：定时器句柄指针
  * @note   无
  * @retval 无
  */
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim)
{
    __IO uint32_t tim_count=0;
    __IO uint32_t tmp = 0;
    if(htim->Instance == ATIM_TIMX_PWM)
    {
        tim_count=__HAL_TIM_GET_COUNTER(&g_atimx_handle);
        tmp = (tim_count+step_angle)&0xFFFF;
        __HAL_TIM_SetCompare(&g_atimx_handle,TIM_CHANNEL_3, tmp);
    }
}
```

接着看下闭环控制的周期调用：

```
/* HAL 通用回调接口函数 */

/**
```

```
 * @brief       定时器更新中断回调函数
 * @param       htim:定时器句柄指针
 * @note        此函数会被定时器中断函数共同调用的
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    static uint16_t val=0;
    static uint16_t sum_pulse=0;
    if(htim->Instance == GTIM_TIMX_ENCODER)
    {
        if(__HAL_TIM_IS_TIM_COUNTING_DOWN(&g_timx_encode_chy_handle))
        {
            g_timx_encode_count--;
        }
        else
        {
            g_timx_encode_count++;
        }
    }
    else if(htim->Instance == BTIM_TIMX_INT)
    {
        g_encode_now = gtim_get_encode();            /* 获取编码器值，用于计算速度 */

        if(val == SMAPLSE_PID_SPEED)
        {
            g_speed=g_encode_now- g_encode_old;
            sum_pulse+=abs(g_speed);
            g_encode_old=g_encode_now;
     /* 实际速度（使用编码器运算）速度 = 一分钟的脉冲数/一圈所需要的脉冲数 就可以求出 RPM */
            g_step_motor.speed=g_speed * ((1000/SMAPLSE_PID_SPEED)*60.0) /
                                          (uint16_t)(ENCODER_SPR);
              g_step_motor.setspeed=((float)(1000000/
              (uint16_t)(step_angle*2))/PULSE_REV)*60;/*设定速度(脉冲运算转速)*/

            if(g_run_flag)/* 电机启动 */
            {
               g_step_motor.location=gtim_get_encode();
               g_motor_pwm=increment_pid_ctrl(&g_location_pid,
                                              g_step_motor.location);

               (g_motor_pwm > 0) ? (g_step_motor.dir=CW) :
                                   (g_step_motor.dir=CCW);
               g_motor_pwm=abs(g_motor_pwm);
               stepper_motion_ctrl(g_step_motor.dir,g_motor_pwm);
               g_send_flag=1;
            }
            val = 0;
        }
        val ++;
    }
}
```

闭环控制函数是在基本定时器 TIM6 的更新中断里边被循环调用，每 20ms 调用一次，也就是闭环控制的采样周期是 20ms。

在 main 函数里面编写如下代码：

```
int main(void)
{
    uint8_t key,t;
    uint8_t flag_stop=0;
    char buf[32];
    uint8_t debug_cmd=0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
```

```
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    rs232_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    btim_timx_int_init(1000-1 , 84-1);
    gtim_timx_encoder_chy_init(0xFFFF , 0);
    stepper_init(0xFFFF, 168 - 1);
    pid_init();

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"Stepper Motor Test",g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:Start/Stop",g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:SET++",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:SET--",g_point_color);
    printf("KEY0 翻转电机状态，开启/关闭\r\n");
    printf("KEY1 增加目标位置\r\n");
    printf("KEY2 减少目标位置\r\n");


    #if DEBUG_ENABLE /*开启调试*/
        debug_init();/*PID 调试初始化*/
        /* 初始化同步数据（选择第 x 组 PIDX,目标速度地址,P,I,D 参数）到上位机 */
        debug_send_initdata(TYPE_PID1,User_SetPoint,L_KP,L_KI,L_KD);
    #endif
    while (1)
    {
        #if DEBUG_ENABLE  /*Debug 发送部分*/
            if(g_send_flag)
            {
                g_send_flag=0;
               /* 选择通道 1 发送实际速度 */
                debug_send_wave_data(1,(int16_t)g_step_motor.location);
               /* 选择通道 2 发送设置的目标速度 */
                debug_send_wave_data(2,(int16_t)*User_SetPoint);
            }
        #endif
        t++;
        if(t % 20 == 0)
        {
            sprintf(buf,"Set_Locaion:%f",*User_SetPoint);
            lcd_show_string(10,90,200,16,16,buf,g_point_color);
            sprintf(buf,"Encode:%d    ",gtim_get_encode());
            lcd_show_string(10,110,200,16,16,buf,g_point_color);
            sprintf(buf,"Motor_Speed :%3d RPM ",(int16_t)g_step_motor.speed);
            lcd_show_string(10,130,200,16,16,buf,g_point_color);
            sprintf(buf,"Set_Speed :%3d RPM ",(int16_t)g_step_motor.setspeed);
            lcd_show_string(10,150,200,16,16,buf,g_point_color);
            LED0_TOGGLE();                              /*LED0(红灯) 翻转*/
        }
        delay_ms(10);
        /**********************************按键部分**************************/
        key = key_scan(0);
        if(key == KEY0_PRES)                            /* 电机状态翻转 */
        {
            flag_stop=!flag_stop;
            if(flag_stop)
            {
                g_run_flag=1;
                /* 启动电机 */
                HAL_TIM_OC_Start_IT(&g_atimx_handle,ATIM_TIMX_PWM_CH3);
```

```
            }
            else
            {
                g_run_flag=0;
                /* 关闭电机 */
                HAL_TIM_OC_Stop_IT(&g_atimx_handle,ATIM_TIMX_PWM_CH3);
            }
        }
        if(key == KEY1_PRES)                        /* 步数 ++ */
        {
            *User_SetPoint+=(20*PPM);
            if(*User_SetPoint>=60*PPM)
                *User_SetPoint=60*PPM;
            lcd_show_num(100,90,*User_SetPoint,3,16,g_point_color);
        }
        if(key == KEY2_PRES)                        /* 步数 --  */
        {
            *User_SetPoint-=(20*PPM);
            if(*User_SetPoint<=-60*PPM)
                *User_SetPoint=-60*PPM;
            lcd_show_num(100,90,*User_SetPoint,3,16,g_point_color );
        }
        /**********************************Debug 接收部分**********************/
        #if DEBUG_ENABLE                    /* Debug 接收部分 */
          debug_receive_pid(TYPE_PID1,(float*)&g_location_pid.Proportion,
          (float*)&g_location_pid.Integral,(float*)&g_location_pid.Derivative);
              /* 设置目标调节范围 */
              debug_set_point_range((float)(60*PPM),(float)(-60*PPM),120*PPM);
              debug_cmd = debug_receive_ctrl_code();    /* 读取命令 */
              if(debug_cmd == HALT_CODE)                /* 停机 */
              {
                  g_run_flag = 0;                       /* 标记停机 */
                  HAL_TIM_OC_Stop_IT(&g_atimx_handle,TIM_CHANNEL_3);
              }
              else if (debug_cmd == RUN_CODE)           /* 运行（解除刹车和停机） */
              {
                  g_run_flag = 1;                       /* 运行标记 */
                  HAL_TIM_OC_Start_IT(&g_atimx_handle,TIM_CHANNEL_3);/*启动电机*/
                  debug_send_motorstate(RUN_STATE);     /* 电机运行 */
              }
              else if (debug_cmd == BREAKED)            /* 刹车 */
              {
                  *User_SetPoint = g_step_motor.location;
                  debug_send_motorstate(BREAKED_STATE);/* 电机刹车 */
              }
        #endif
    }
}
```

mian 函数中主要初始化一些外设，包括 PID 初始化，定时器初始化、步进电机初始化等等，其中当按下按键 KEY0 控制步进电机状态开启/关闭，按下 KEY1 增加目标位置，按下 KEY2 减少目标位置。

20.3.3 下载验证

硬件连接好之后下载程序到开发板，液晶屏和串口都会显示提示信息，打开正点原子 PID 调试助手(注意接上串口 1 的 USB_TTL 连接至电脑)，当我们按下按键 KEY0 电机开始启动，通过调节按键 KEY1 和 KEY2，可以看到 PID 调试助手位置变化曲线：

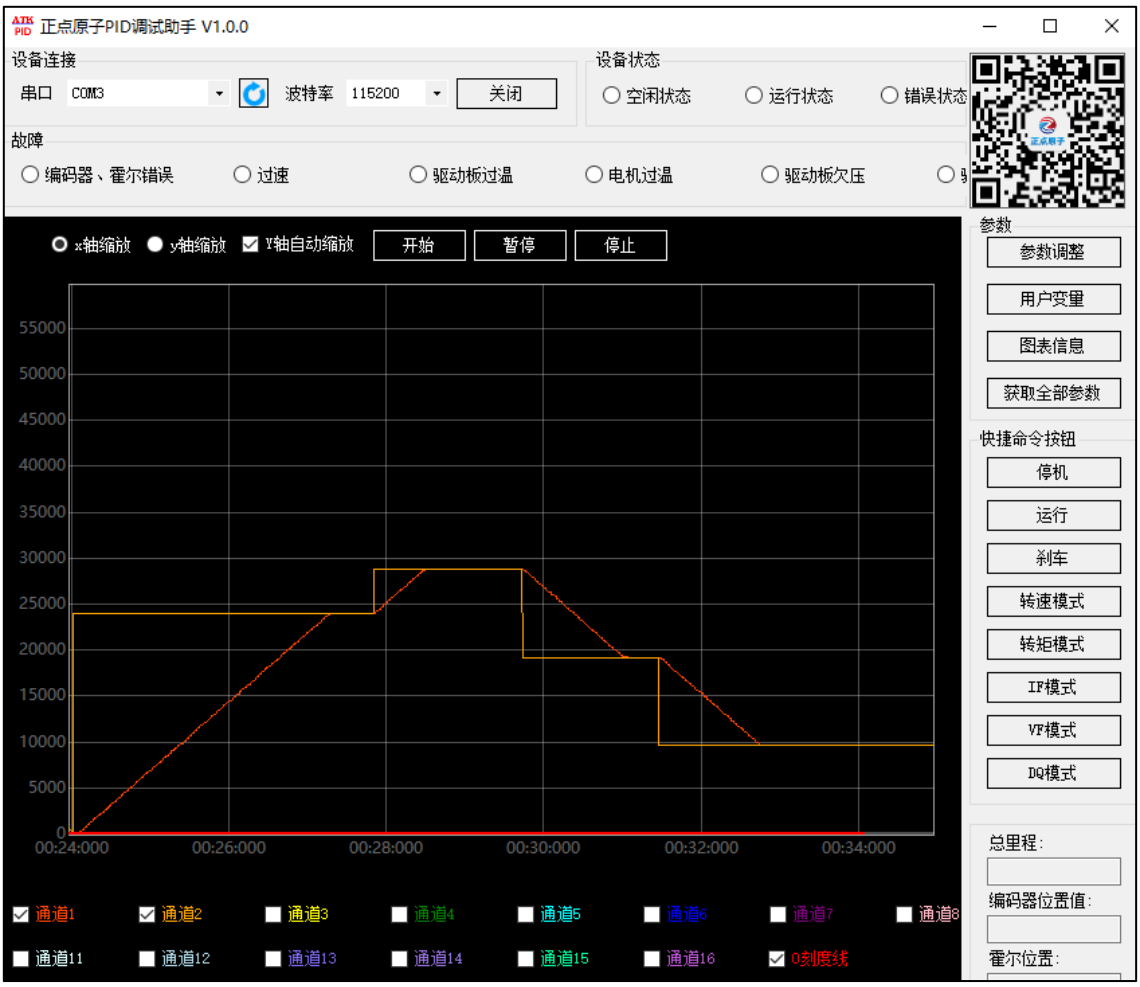


图 20.3.3.1 PID 调试助手波形图

由图 20.3.3.1 我们发现该曲线更像是一条斜线，是因为我们对步进电机电机做了限速，所以 PID 输出的期望值大小被限制了，导致每次位置增量值一致，所以看起来类似条斜线。

第 21 章 直流无刷电机

本章我们主要来学习直流无刷电机的简介以及直流无刷电机的驱动原理，然后对直流无刷电机的驱动硬件平台进行介绍。

本章分为如下几个小节：

21.1 直流无刷电机的简介
21.2 直流无刷电机的基本控制原理
21.3 直流无刷电机的霍尔传感器控制基本原理
21.4 直流无刷电机的驱动硬件平台简介

21.1 直流无刷电机的简介

直流无刷电机(Brushless Direct Current Motor，简称 BLDCM）顾名思义没有了直流有刷电机中的电刷和换向器等结构，因此线圈绕组不参与旋转，而是作为定子，永磁体作为转子，所以需要通过控制线圈电流方向来改变磁场方向从而使转子持续旋转，同步进电机不同的是，无刷电机绕组通常是 3 组，并且只有 3 个引出接线端子，一般为星形接法，3 组线圈的起始端通过电机内部连接到一起，剩余的一端引出作为电机的引线。由于无刷电机的换向需要根据转子位置确定，因此一般无刷电机有霍尔传感器，用于提供转子位置信息，当然也不是必需的，还可以通过无感的方式，用仅有的 3 根电机线来分析和确定转子的位置、转速等信息。

21.1.1 直流无刷电机与直流有刷电机的特点

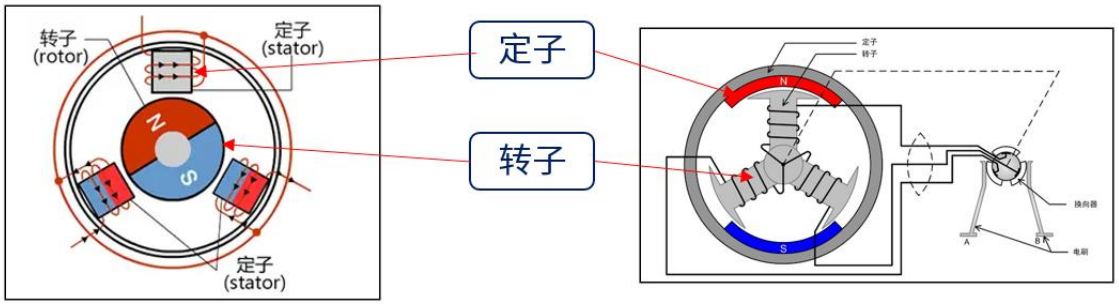


图 a 直流无刷电机结构图　　图 b 直流有刷电机结构图

可以从图 a 与图 b 中看出有刷电机与无刷电机之间最大的结构区别在于：无刷电机没有电刷以及换向器；并且无刷电机的线圈绕组不参与旋转，而是作为定子，永磁体作为转子，与有刷电机是相反的结构。

无刷电机其性能相比于有刷电机有很大优势，由于没有了碳刷结构，所以干扰小、噪音低、运转流畅、高速而且寿命更长。缺点就是控制较为复杂、成本比有刷电机贵。在四轴飞行器，汽车，工业工控上的应用非常广泛。下面我们总结了关于直流有刷电机以及无刷电机的特点，如下表：

电机类型	特点
直流有刷电机	使用寿命较短、噪音大、转速不高、但是价格便宜、控制、启动简单
直流无刷电机	使用寿命长、噪音低、转速快、但是价格贵、控制较复杂。

表 21.1.1.1 有刷电机与无刷电机特点

21.1.2 直流无刷电机的分类

无刷电机的分类方式有很多种，我们这里仅以驱动方式进行区分，如下图：

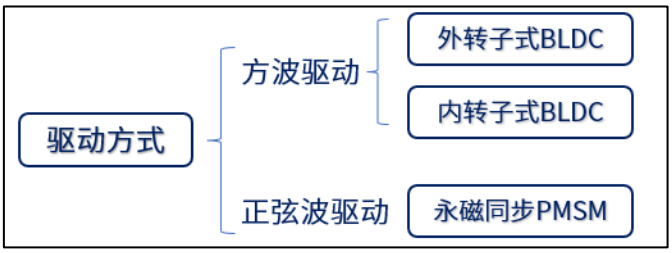


图 21.1.2.1 驱动方式分类示意图

图 21.1.2.1 中我们可以看到，直流无刷电机以驱动方式进行分类，可以分为：方波驱动以及正弦波驱动，其中方波驱动又分为：外转子式 BLDC、内转子式 BLDC；正弦波驱动分为：永磁同步电机（PMSM）。下面我们来看下这些电机的实物图：

图 21.1.2.2 外转式 BLDC

图 21.1.2.2 的无刷电机为外置式转子，即磁体做成的转子在外侧，内侧的定子是缠绕漆包线的硅钢叠片，主要应用于四轴飞行器等。

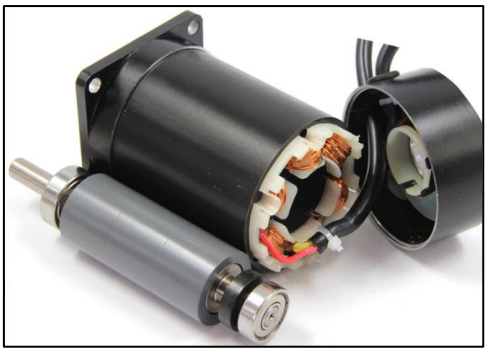


图 21.1.2.3 内转式 BLDC

图 21.1.2.3 的无刷电机类似步进电机的一种结构，即磁体转子在内侧，硅钢片定子在电机外侧。一般电机直径和长度越大，扭矩越大，有时还通过减速齿轮组来增大扭矩，在工业工控上的应用非常广泛。

图 21.1.2.4 永磁同步电机（PMSM）

图 21.1.2.4 为永磁同步电机（PMSM）该电机一般都配有高精度编码器，优点显著，精度高，效率高，运行可靠，损耗小，在精度要求较高的场合被广泛应用。适合高级控制方式——FOC。

21.1.3 BLDC 和 PMSM 的区别

BLDC 与 PMSM 电机在控制方式、运行方式、绕组方式等等，都有很大的区别，最主要的区别就是反电动势不同，BLDC 接近于方波，PMSM 接近于正弦波。下面我们总结了一张 BLDC 与 PMSM 之间的对比表，如下表：

不同点	BLDC	PMSM
反电动势	具有梯形反电动势	具有正弦波反电动势
运行电流	梯形波电流	正弦波电流
绕组方式	定子绕组为集中绕组，永磁转子形成方波磁场	定子绕组为短距分布绕组，永磁转子形成正弦磁场
控制方式	控制算法相对简单	控制算法复杂（FOC）
运行方式	绕组两两导通	绕组三相全部导通

表 21.1.3.1 BLDC 与 PMSM 之间的区别

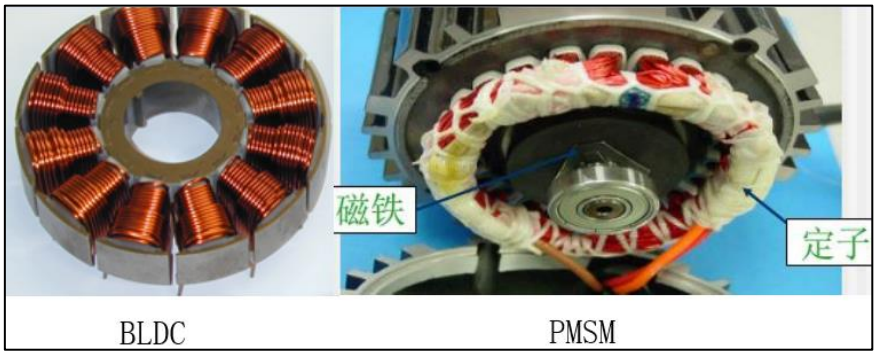


图 21.1.3.1 绕组方式区别

21.1.4 直流无刷电机的主要参数介绍

- **极对数**：转子磁铁 NS 级的对数，此参数和电机旋转速度有关：电子速度 = 电机实际速度 * 极对数。

- **KV 值**：电机的运转速度，值越大电机转速越大。电机转速 =KV 值 * 工作电压。同系列同外形尺寸的无刷电机，根据绕线匝数的多少，会表现出不同的 KV 特性。绕线匝数多的， KV 值低，最高输出电流小，扭力大；绕线匝数少的， KV 值高，最高输出电流大，扭力小；
- **额定转速**：额定的电流下的空载转速，通常单位用 RPM 表示；
- **转矩**：电机中转子产生的可以带动机械负载的驱动力矩。通常单位为：N.M；

21.2 直流无刷电机的基本控制原理

首先我们回忆下高中知识——安培定则，也叫右手螺旋定则，是表示电流和电流激发磁场的磁感线方向间关系的定则。我们这里利用定则之一：当用右手握住通电螺线管，让四指指向电流的方向，那么大拇指所指的那一端是通电螺线管的 N 极。如下图所示：

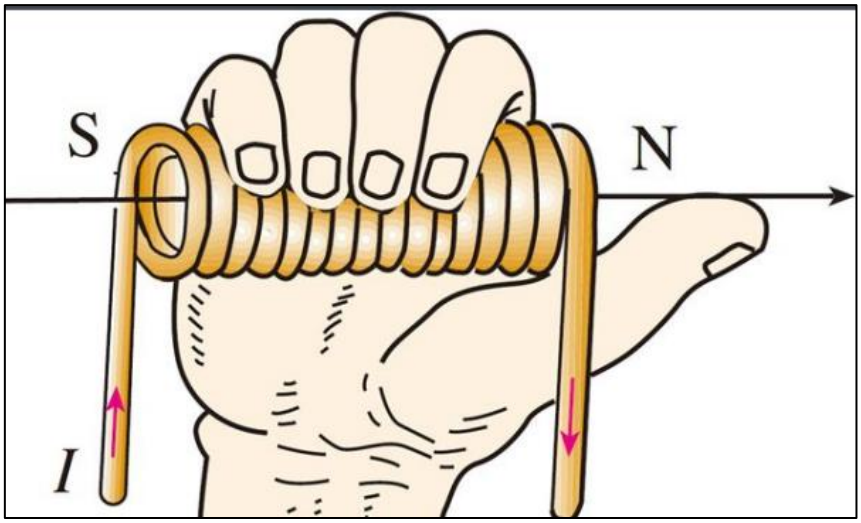


图 21.2.1 右手螺旋定则示意图

所以通电线圈会产生磁场，我们可以把通电线圈的磁场看成一个磁体，如下图：

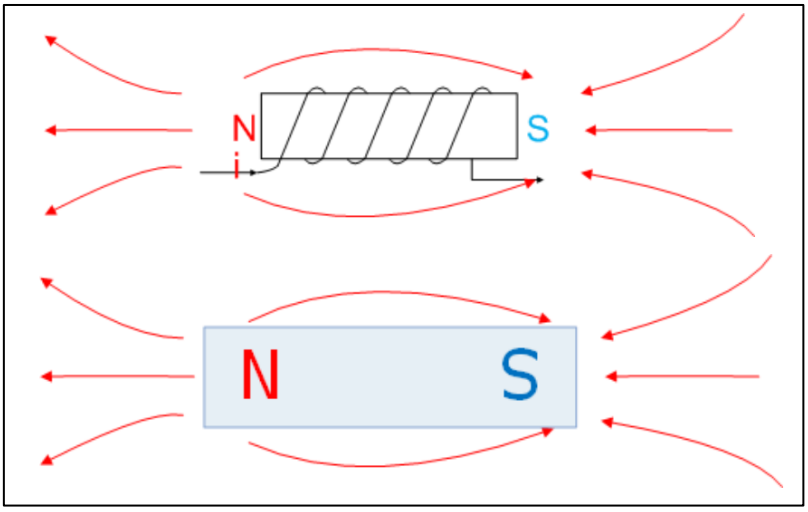


图 21.2.2 通电线圈与磁体的等效图

磁体之间，存在异性相吸，同性相斥的原理，通电线圈和永磁体之间同样存在，而无刷电机就是利用了通电线圈和永磁体的相互作用原理，下面看下 BLDC 的内部结构图：

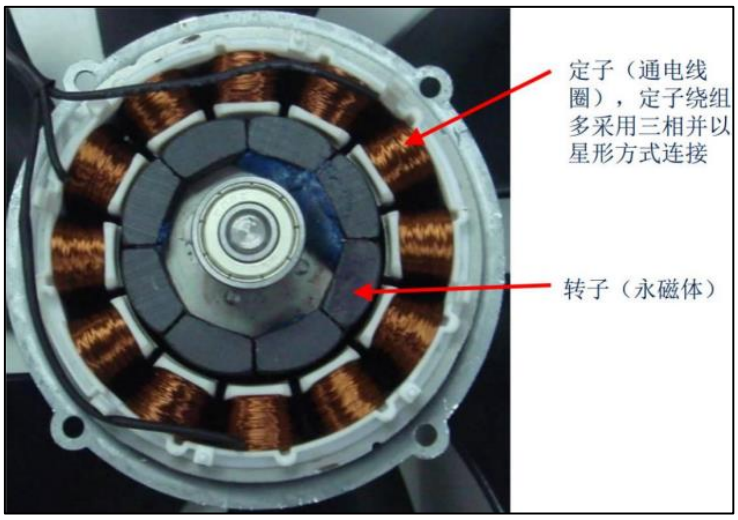


图 21.2.3 内置转子式无刷电机

无刷电机的 3 根电机线按顺序依次叫 U 相线(一般为黄色)，V 相线(一般为绿色)和 W 相线(一般为蓝色)，3 组漆包线绕组的一端连接在一起，另外一端引出即为 UVW 相线，所以任意两根相线通电都可以导通这两个线圈，除此之外，一般无刷电机还有另外一组引出线，即霍尔传感器线，一般有 5 根分别是：

1：红色霍尔电源线正极；
2：黑色霍尔电源线负极；
3：黄色霍尔 U 相输出；
4：绿色霍尔 V 相输出；
5：蓝色霍尔 W 相输出，

霍尔传感器的 3 相电平输出主要用于指示无刷电机转子的位置。

接着来分析直流无刷电机的驱动原理，为了方便分析，将上述无刷电机实物图的结构简化为以下形式，如图 21.2.4：

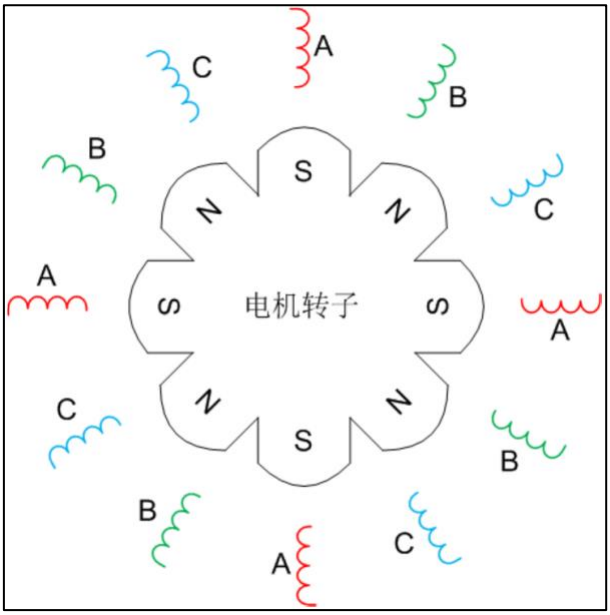


图 21.2.4 无刷电机简化结构图

极对数过多不易于分析所以进一步简化为如下：

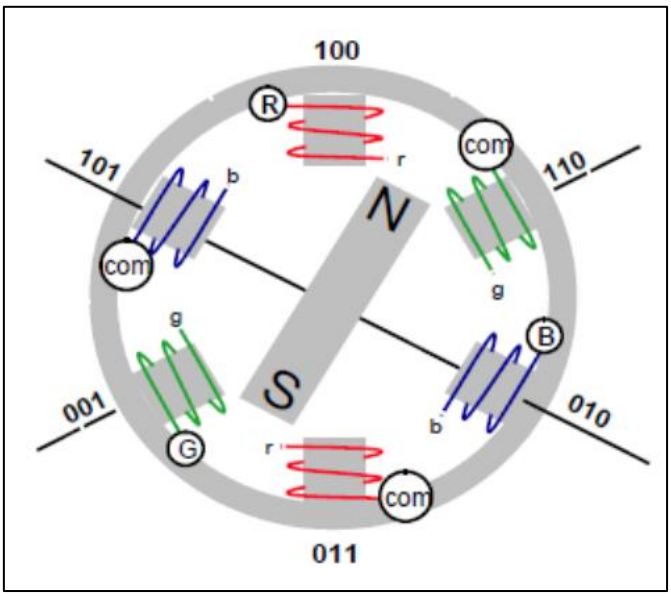


图 21.2.5 无刷电机进一步简化图示

其中 COM 为公共结点，就是将 UVW 三相的一端短接一起形成的节点，另外一端引出，所以可以看作 3 个有公共端的线圈和一个永磁体结构，如下：

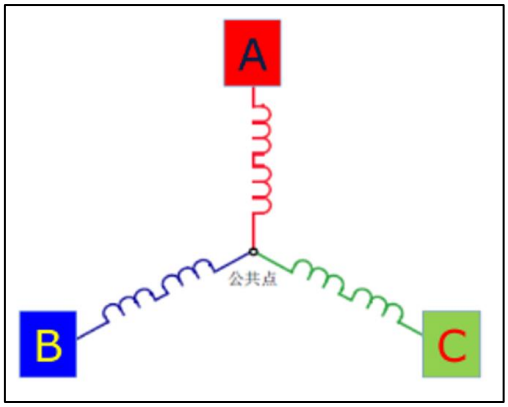


图 21.2.6 无刷电机简化结构

此时将转子放上，并将 A、B、C 改为 U、V、W 三相方便理解，如下图：

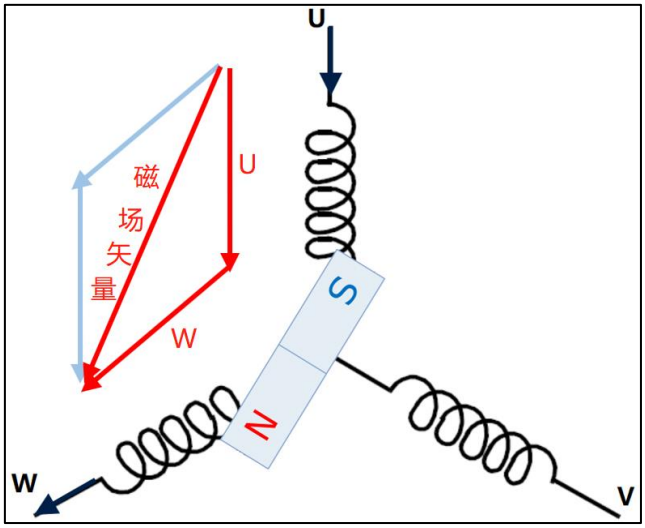


图 21.2.7 无刷电机简化图

图 21.2.7 反映了 U 相接正极，W 相接负极，V 相开路，那么此时电流由 U 相流到 W 相，同时 U 相和 W 相产生的磁场合成的磁场矢量方向即为转子磁场方向。

由于 BLDC 的运行方式为绕组两两导通，所以三相线圈的导通组合只有 6 种通电情况，根据合理的顺序依次切换通电顺序即可让转子跟着磁场转起来。如下图：

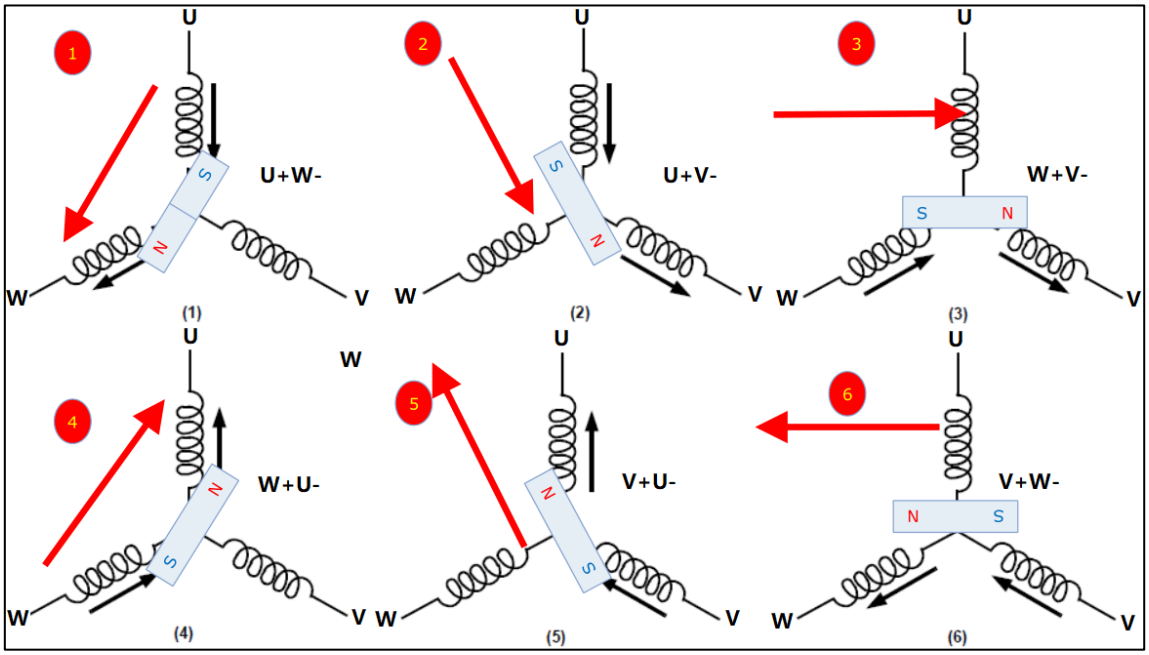


图 21.2.8 无刷电机定子顺序换向步骤

由上图可知，想要控制 BLDC 旋转，根本的问题就是产生这 6 拍的工作方式的电压信号（称为 BLDC 的六步控制），假设 BLDC 的额定电压为 24V，电机的三根相线定义为 U，V，W：

1. 给 U 接 24V、V 悬空、W 接 GND，此时电机的转轴对应上图(1)的转子位置。
2. 在上一步的基础上修改接线方式，给 U 接 24V、V 接 GND、W 悬空，此时电机的转轴对应上图(2)的转子位置，相较于(1)旋转了一个角度。
3. 在 2.的基础上继续修改接线方式，U 悬空，V 接 GND，W 接 24V，此时对应(3)。
4. 在 3.的基础上继续修改接线方式，U 接 GND，V 悬空，W 接 24V，此时对应(4)。
5. 在 4.的基础上继续修改接线方式，U 接 GND，V 接 24V，W 悬空，此时对应(5)。
6. 在 5.的基础上继续修改接线方式，U 悬空，V 接 24V，W 接 GND，此时对应(6)。

同样的，如果顺序相反则无刷电机会反向旋转。同步进电机类似，实际上步进电机也算是无刷电机，所以无刷电机也会存在步进电机那样的问题，即丢步，原因也是定子产生的磁场没有来得及改变定子的位置，一般也可以增大驱动电流来缓解。一般步进电机没有转子位置传感器，但是无刷电机一般是有的，传感器需要独立供电，一般直流 5~24V 即可，传感器有 3 根输出信号线，同样排除全部高、低电平情况也是 6 种状态。

我们可以从图 21.2.8 得知，不管是 U 相、V 相、W 相在 6 种状态里边有时需要接正极，有时需要接负极，所以就有个问题了，如何简便的控制三相极性的切换？我们使用三相逆变电路来实现，如下图：

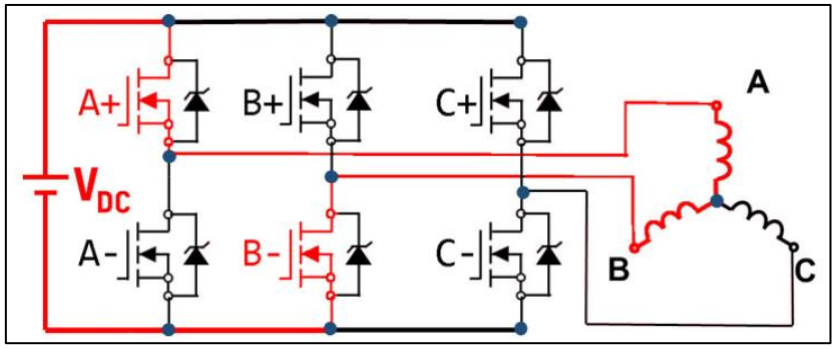


图 21.2.9 三相逆变电路

所谓的三相逆变电路就是由三个半桥构成的电路，图 21.2.9 中的 A+与 A-为一个半桥，B+与 B-以及 C+与 C-各自又为一个半桥，共三个半桥；这三个半桥各自控制对应的 A、B、C 三相绕组；当控制 A 的上桥臂 A+导通时，此时 A 相绕组接到电源正，当控制 B 的下桥臂 B-导通时，此时 B 相绕组接到电源负，所以此时电流由 A 流向 B。

所以想要控制绕组的极性，只需要控制绕组对应半桥的“上桥臂导通”或者“下桥臂导通”就可以实现控制该相连接至“正极”或者“负极”了，但是要注意不可以同侧半桥上下桥臂同时导通，负责会短路，烧毁电机！那我们要实现图 21.2.8 中的 6 步控制，就可以通过三相逆变电路来实现，如下图 21.2.10：

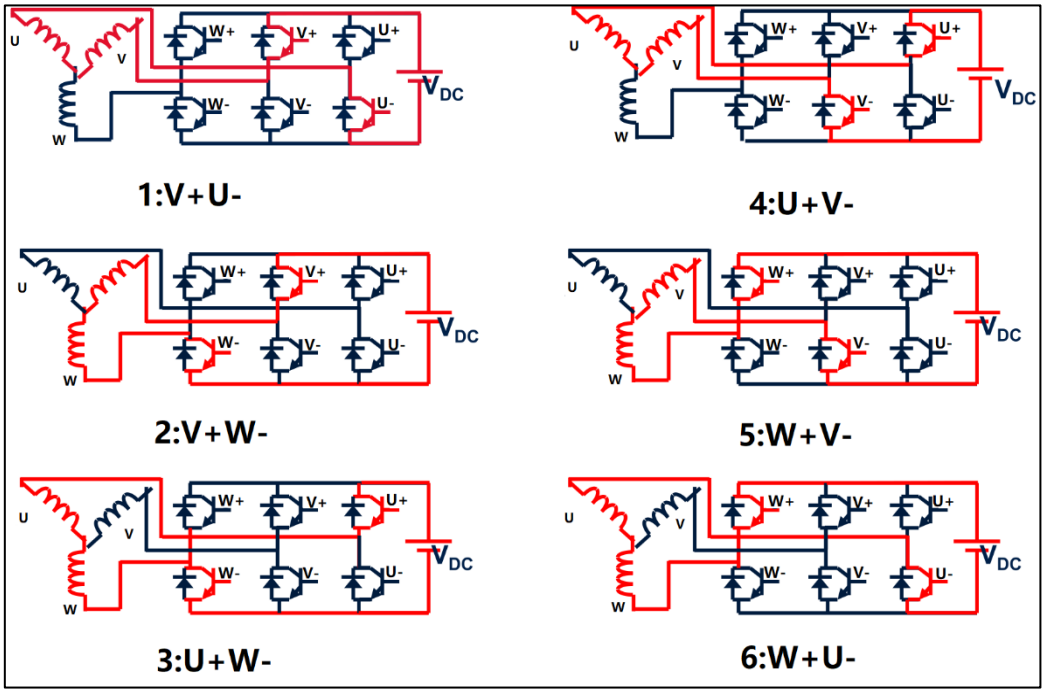


图 21.2.10 六步换向导通情况

从图 21.2.10 我们发现以上方式直接把电源加载到线圈上，这样会直接使电机很快飙到很高的速度，这样不利于我们控制，所以一般工作时都是将高低电平用 PWM 来代替，这样可以方便的控制线圈电流，从而控制转子扭矩及转速。接着我们就来看下 PWM 控制方式的示意图：

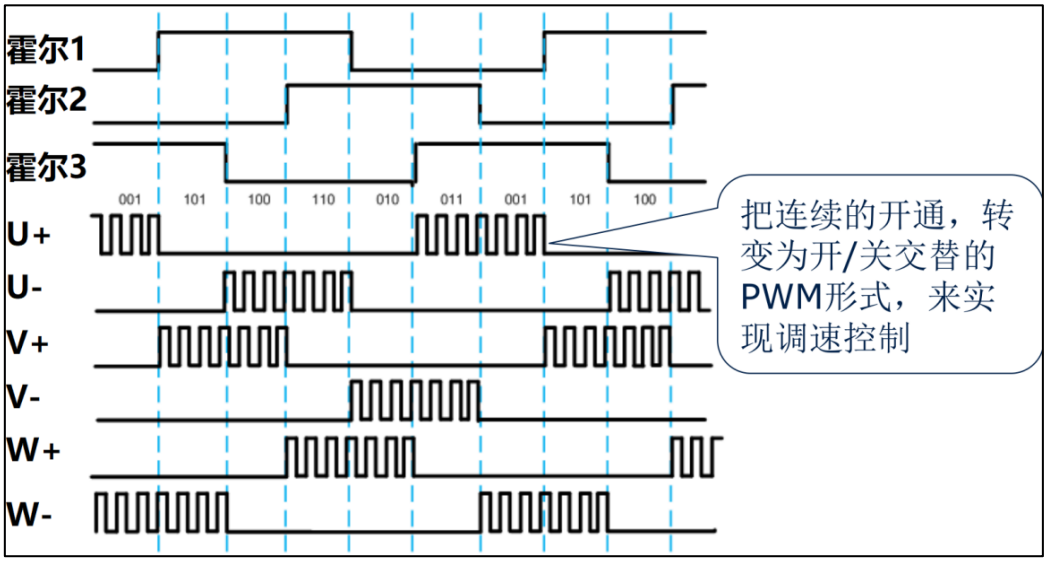


图 21.2.11 PWM 方式驱动无刷电机

通常使用 PWM 控制直流无刷电机的常见方法有 5 种，分别是： PWM – ON、ON–

PWM、H_ON – L_PWM 、H_PWM – L_ON、H_PWM – L_PWM。且均是电机处于 120°的运行方式下进行的。如下图：

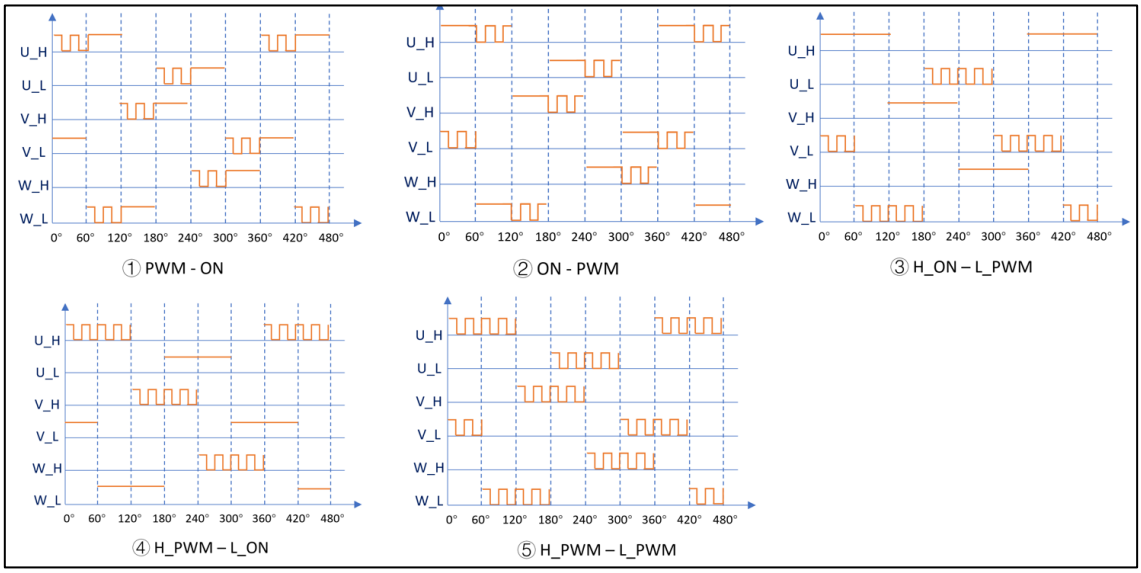


图 21.2.12 PWM 调制方式

（1）采用 PWM-ON 方式时，下桥换相和上桥换相的换相转矩脉动相等，且最小；非换向相电流脉动也是最小的；

（2）采用 ON-PWM 方式时，下桥和上桥换相转矩脉动相等且比 PWM-ON 方式大，非换向相电流脉动也比 PWM-ON 方式时大。

（3）采用 H_ON - L_PWM 方式时，下桥换相转矩脉动和非换向相电流脉动小且与 PWM-ON 方式时的转矩脉动和电流脉动相等，上桥换相转矩脉动和非换向相电流脉动大且与 ON - PWM 方式时的转矩脉动和电流脉动相等。

（4）采用 H_PWM - L_ON 方式时，下桥换相转矩脉动和非换向相电流脉动大且与 ON-PWM 方式时的转矩脉动和电流脉动相等，上桥换相转矩脉动和非换向相电流脉动小且与 PWM-ON 方式时的转矩脉动和电流脉动相等。

（5）采用 H_PWM - L_PWM 方式时，换相转矩脉动最大且非换向相电流脉动也最大。

不同的控制方式在性能上有不同的效果，针对实际的应用场合可以多尝试多种调制方式，然后选择最优调制方式，一般认为：单极性调制转矩波动更小，双极性调制转矩波动较大。我们例程依据我们的驱动硬件所使用的是 **H_PWM – L_ON** 的驱动方式。

21.3 直流无刷电机的霍尔传感器控制基本原理

我们虽然已经知道了控制转子的六个节拍的方法。但是你有没有发现一个新的问题：如果不知道转子的位置，就不知道何时该驱动哪个绕组？所以驱动无刷电机的前提是我们必须要知道转子的**当前位置**！

无刷电机依靠传感器提供转子位置信息进行驱动的方式我们称之为**有感驱动**方式，既然有有感驱动，那么是不是也同样存在**无感驱动**？是的，在无感驱动章节会重点介绍，这里先从有感驱动开始。

无刷电机的传感器一般为霍尔传感器，根据霍尔器件可检测磁场的变化的特性再搭配一定的电路将磁场方向变化信号转化成不同的高低电平信号输出，如下图 21.3.1 所示

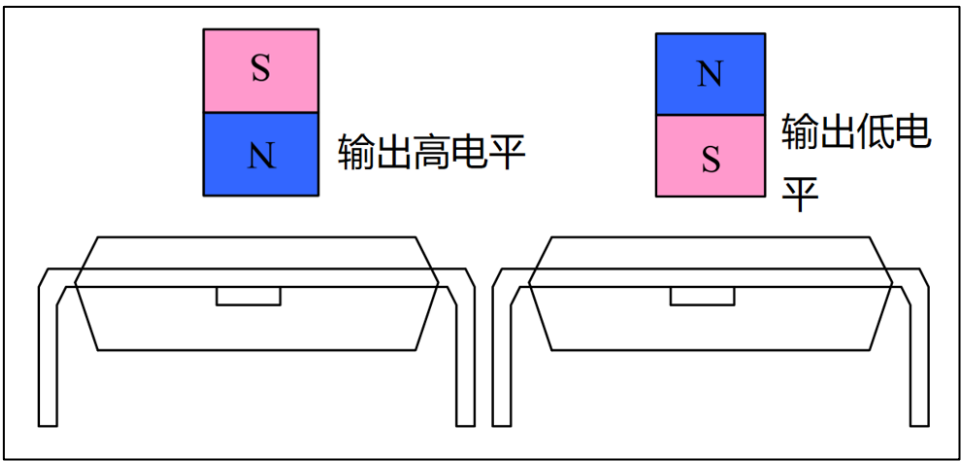


图 21.3.1 霍尔传感器磁场检测示意图

以霍尔传感器为参照物，定子旋转时，霍尔传感器检测到的磁场变化及输出信号如下所示。

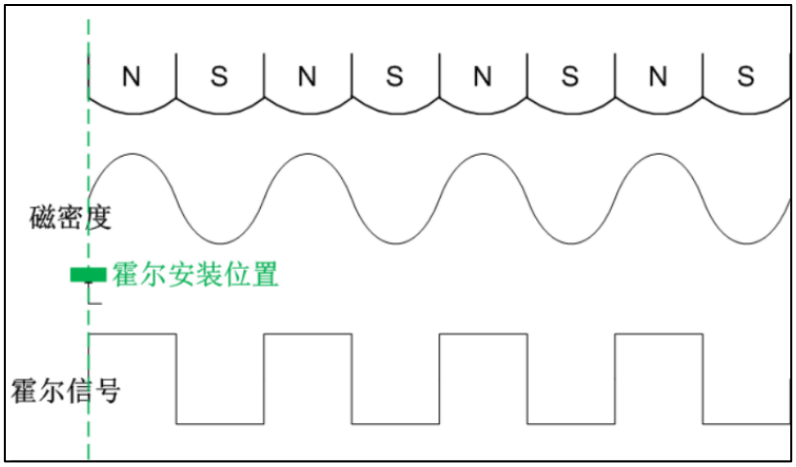


图 21.3.2 磁场变化及霍尔输出信号图示

同无刷电机均匀分布的定子一样，用于输出 3 路磁场信号的 3 个霍尔传感器也是均匀分布在无刷电机的一周的，每相邻两个传感器**电角度**相差 120°。电机按一定方向转动时，3 个霍尔的输出会按照 6 步的规律变化，如下所示。

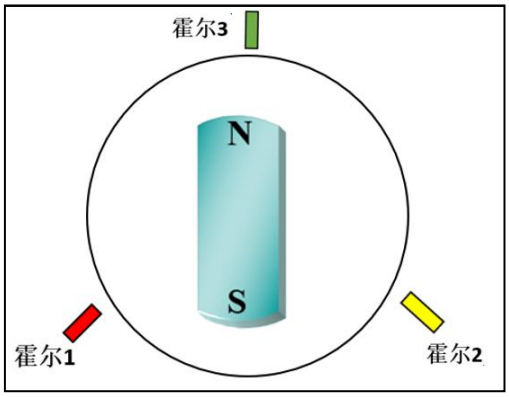


图 21.3.3 霍尔传感器安装位置 120° 电角度

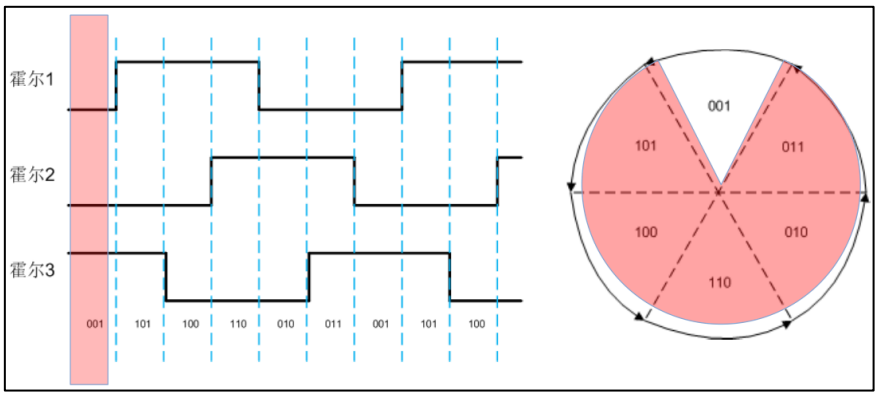


图 21.3.4 霍尔传感器信号变化图示

补充，**机械角度**是指电机转子的旋转角度，**电角度**是指磁场的旋转角度，它们的关系式满足：**电角度 = 机械角度 * 极对数。**

图 21.3.3 为霍尔传感器安装的以 120° 电角度所安装的示意图；图 21.3.4 为电机旋转时三个霍尔传感器输出的波形，所对应的扇区组合。通过三个霍尔传感器输出的波形就可以判断当前转子的具体位置，同样满足 6 步一周期。

6 步换向需要依赖霍尔传感器反馈的转子位置，其相对应的就是三相逆变电路的上下桥臂导通情况，如下表所示。

方向	霍尔 W	霍尔 V	霍尔 U	U+	U-	V+	V-	W+	W-	顺序
正转	1	0	1	打开	关闭	关闭	打开	关闭	关闭	↑
	0	0	1	打开	关闭	关闭	关闭	关闭	打开	↑
	0	1	1	关闭	关闭	打开	关闭	关闭	打开	↑
	0	1	0	关闭	打开	打开	关闭	关闭	关闭	↑
	1	1	0	关闭	打开	关闭	关闭	打开	关闭	↑
	1	0	0	关闭	关闭	关闭	打开	打开	关闭	↑
反转	1	0	1	关闭	打开	打开	关闭	关闭	关闭	↓
	0	0	1	关闭	打开	关闭	关闭	打开	关闭	↓
	0	1	1	关闭	关闭	关闭	打开	打开	关闭	↓
	0	1	0	打开	关闭	关闭	打开	关闭	关闭	↓
	1	1	0	打开	关闭	关闭	关闭	关闭	打开	↓
	1	0	0	关闭	关闭	打开	关闭	关闭	打开	↓

表 21.3.1 不同转向时霍尔状态及 MOS 管状态

如表 21.3.1 所示，当我们需要无刷电机**正转**时，如检测霍尔输出的组合值为“**101**”，那么此时我们需要导通 **U 相的上桥臂“U+”**，和 **V 相的下桥臂“V-”**即对应当前转子所在的扇区位置。其他霍尔组合情况同理。

特别注意，一般厂家都会给出一个霍尔传感器和绕组得电情况的对应关系表，不一定和表 21.3.1 完全对应一致，但是原理分析都是一致的，表 21.3.1 为我们店铺的无刷电机正反转控制逻辑表。

最后我们可以得到有感驱动无刷电机整体框图如下：

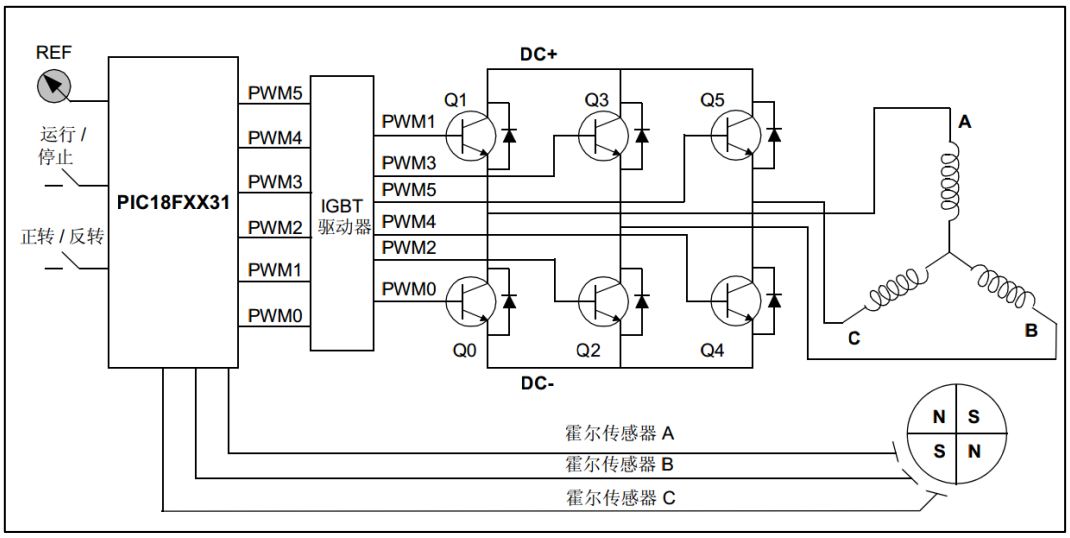


图 21.3.5 有感驱动整体框图

21.4 直流无刷电机的驱动硬件平台简介

21.4.1 店铺 BLDC 的技术参数

关于店铺无刷电机的技术参数可查看下表：

型号 Model	ATK-BL57H95D20E
磁级数	4
相数	3
额定电压（VDC）	24
额定转速 （RPM）	3000
额定扭矩 （N-m）	0.33
额定电流 （Amps）	6.6
输出功率 （W）	105
峰值扭矩 （N-m）	4.2
峰值电流（Amps）	19.8
力矩常数（N-m/A）	0.063
反电势（V/kRPM）	6.6
线电阻（Ω）	0.4
线电感（mH）	0.8
转动惯量 （g·cm²）	173
机身长度（mm）	95
重量 （kg）	1.0

表 21.4.1.1 无刷电机参数

21.4.2 无刷电机驱动板介绍

无刷电机驱动板同直流驱动板类似，相对多了一组驱动桥用于控制无刷电机 UVW 三相的通电，无刷电机驱动板有 3 路的电流及反电动势采集电路、还有驱动板电压及温度采集电

路。这里我们重点介绍直流无刷驱动板的三相逆变电路原理及驱动板的接口作用。更多关于 ATK-PD6010B 驱动板原理介绍请查看《ATK-PD6010B 直流无刷驱动板用户手册.pdf》。

1）直流无刷驱动板接口

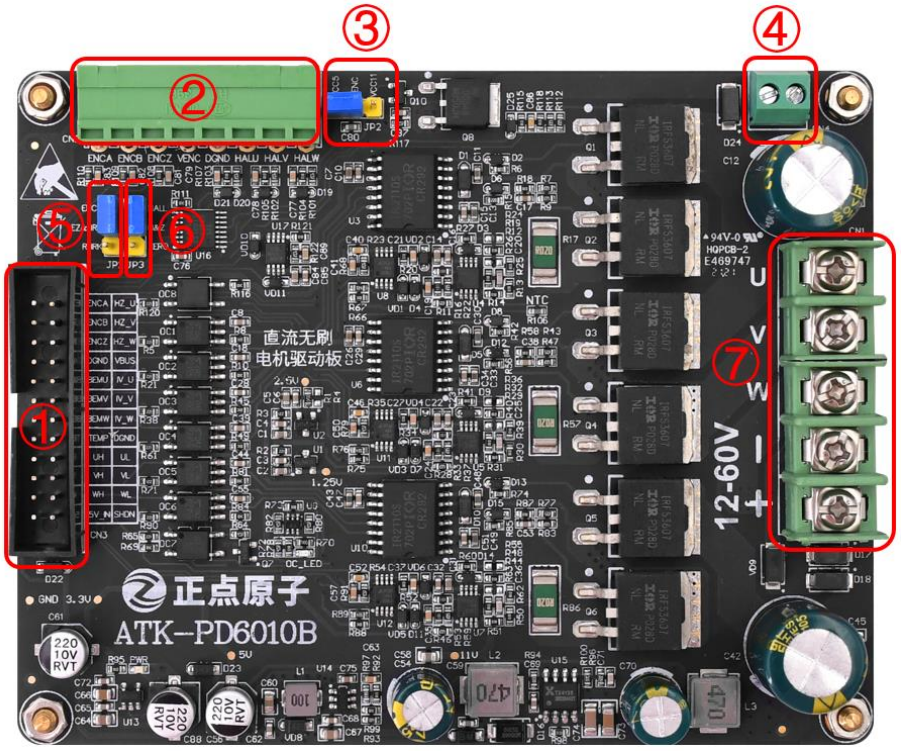


图 21.4.2.1 直流无刷驱动板

为了方便用户接线使用，我们对 ATK-PD6010B 驱动板接口和跳线帽作一个说明：

① 控制信号和采集信号接口 CN3，这是一个 2*12P 2.54 间距的牛角座接口，通过一根 24P 牛角排线直连我们 ATK-DMF407 电机开发板的直流有/无刷驱动器接口，实现开发板对驱动板的控制以及信号的采集。

② 编码器和霍尔传感器接口 CN2，这是一个 8P 的 HT396R 端子，包括 3 相编码器接口、 3 相霍尔传感器接口以及编码器供电接口。这个接口用于连接电机的编码器和霍尔传感器，从而实现电机转速的测量以及电机转子位置的检测等。

③ 编码器供电选择接口 JP2，驱动板板载了 5V 电源和 11V 电源，用户可根据电机编码器供电电压选择合适的电源，默认使用 5V 电源。

④ 制动电阻接口 CN4，用于外接制动电阻， 结合制动电路实现制动控制。

⑤ Z 相编码器和制动控制信号选择接口 JP1，有时候我们可能只用到编码器的 A B 相，那 Z 相就可以用作制动控制，通过制动控制电路实现电机制动。

⑥ 霍尔检测和过零检测选择接口 JP3。为了提高驱动板对更多无刷电机的适配性，我们的驱动板不仅板载了编码器接口和霍尔传感器接口，还板载了过零检测电路。当我们要驱动一个既没有编码器又没有霍尔传感器的直流无刷电机时，这个过零检测电路就起作用了。默认使用霍尔检测的方式。

⑦ 电机和电源接口 CN1，电机接口就是用来连接直流无刷电机的动力线了，包括 U/V/W 这 3 相线；电源接口则是连接外部电源，范围 DC12~60V。

ATK-PD6010B 驱动板 3 相 H 桥电路， 如图 21.4.1.2 所示：原理图如下。

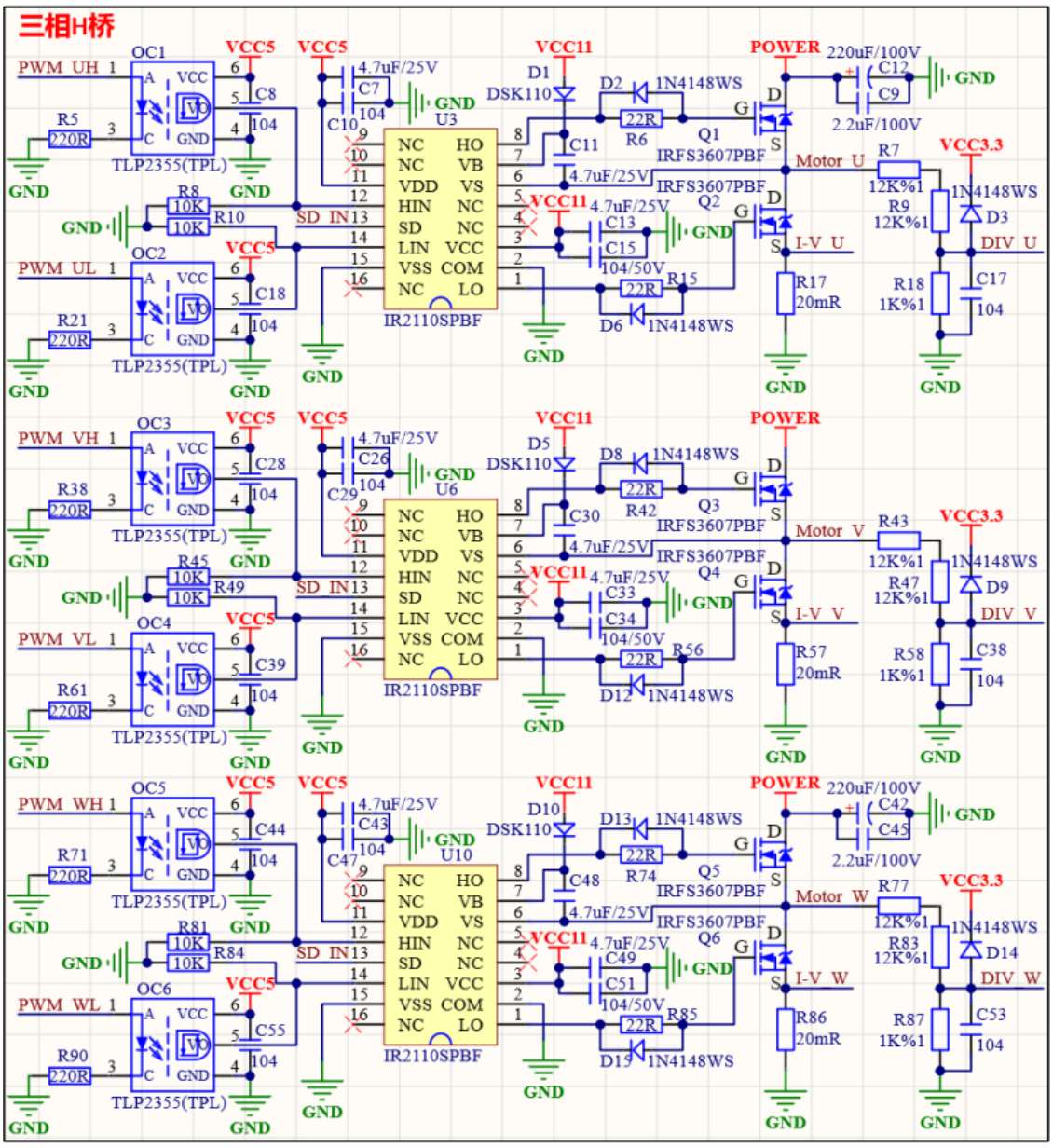


图 21.4.2.2 驱动板三路驱动桥原理图

可以看到，这是 3 个半桥电路(U/V/W 三相)， 主要包括 6 路 PWM 控制信号、 6 路高速光耦 TLP2355、3 颗高低侧栅极驱动芯片 IR2110S、驱动芯片关断控制信号，6 颗 MOS 功率输出管 IRFS3607、 3 路相电流信号、 3 相反向电动势信号、逻辑供电 VCC5、驱动 IC 栅极供电 VCC11 以及功率管母线电压 POWER（DC12~60V）。

因为 U/V/W 三相驱动电路相同，我们就拿 U 相驱动电路来分析：

首先半桥芯片 IR2110S 它由 SD_IN 决定是否能正常输出，当 SD_IN=0 时输出正常，当 SD_IN=1 时芯片输出关闭。类似一个刹车引脚。

U 相高低侧 PWM 信号 PWM_UH 和 PWM_UL 分别通过图腾输出型光耦，进入栅极驱动逻辑控制输入端 HIN 和 LIN，在关断控制信号 SD_IN=0 的时候， HIN 信号通过 HO 直接控制上管 Q1 的开关，HIN=1 则 Q1 导通，HIN=0 则 Q1 关闭；LIN 信号则通过 LO 控制下管 Q2 开和关。 Q1 和 Q2 的输出 Motor_U 就连接电机的 U 相线。

我们这里选择图腾输出型高速光耦，也是经过慎重选择的，图腾输出类似推挽输出，无需上下拉电阻，输出能力强，输出信号跟随输入信号， 当光耦输入端悬空时，默认输出低电平。如果把图腾输出型换成开集输出型光耦，会有什么现象呢？开集输出类似开漏输出，需要强上拉电阻才能保证足够的驱动能力和速度， 最重要一点是，当输入悬空时，输出信号被上拉

电阻拉高，这个时候， HIN 和 LIN 都为高，恰好栅极驱动 IC 的 SD 引脚默认内部下拉，所以都为高电平的 HIN 和 LIN 会同时打开上下管，上下管同时导通，主电源 POWER 直接接地，那后果不堪设想，轻则烧毁驱动器，重则连同输入电源一同烧毁。所以我们这里选择图腾型输出光耦，当输入悬空的时候，上下管都是关闭的。

并且我们驱动板将需要控制器参与的部分接口统一做到一起，使用 24P 的 DC3 插座连接开发板，十分方便。

1） 直流无刷驱动板实物接线

下面我们将手把手地教大家搭建直流无刷电机驱动的硬件平台，这一部分内容十分重要，大家一定要掌握。首先我们需要准备一些材料，如图 21.4.2.3 所示：

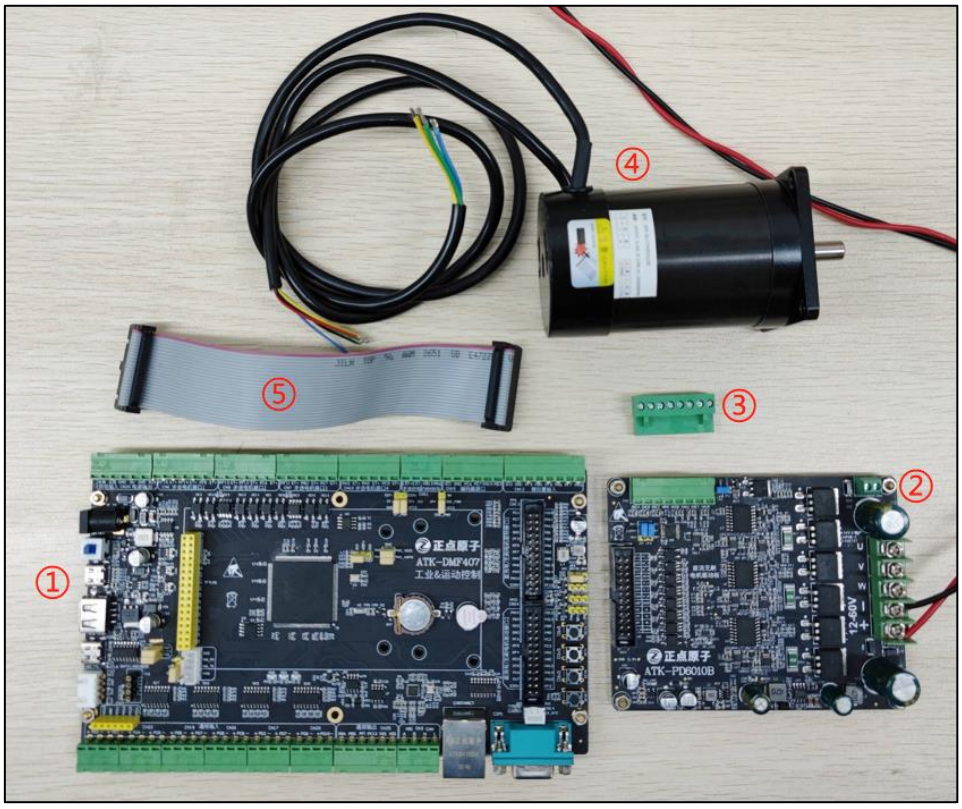


图 21.4.2.3 硬件平台搭建所需材料

图 21.4.2.3 中，不同的序号对应的材料如下：

① 电机开发板；
② 无刷驱动板；
③ 传感器接口 8pin 端子；
④ 无刷电机；
⑤ 专用 24pin 排线（排线颜色不定）；

除了之外，我们还需要准备一个符合电机工作电压的直流电源，例如我们教程中所使用的直流无刷电机的工作电压是 24V，这里就准备一个 24V 的直流电源即可。

有了这些材料之后，我们还要了解下电机的引脚定义，实验用的 57 无刷电机实物如下：

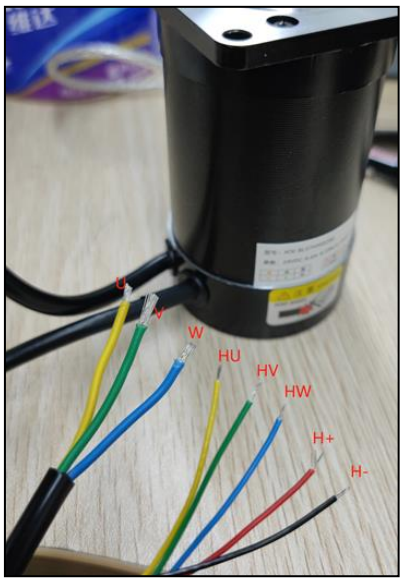


图 21.4.2.4 无刷电机实物

其中引脚定义，在电机标签有注明，如图：

图 21.4.2.5 无刷电机标签

得知电机的引脚定义之后，接着就是将电机与无刷电机驱动板进行连接，连接关系如下表：

电机线	引脚定义	连接至	驱动板丝印
黄色粗线	U 相		U
绿色粗线	V 相		V
蓝色粗线	W 相		W
黄色细线	霍尔 U	→	HALU
绿色细线	霍尔 V		HALV
蓝色细线	霍尔 W		HALW
红色细线	VCC		VENC(5V)
黑色细线	GND		DGND

表 21.4.2.1 电机与无刷驱动板的连接关系

无刷驱动板的传感器接口电源电压有 5V 和 11V 两个选择，大家一定要根据实际传感器的工作电压来选择，例如我们教程中所使用的霍尔传感器（装在直流无刷电机内部）的工作电压是 5V，这里就接 VCC5 即可。驱动板和开发板连接只需要使用排线连接即可，电机与驱动板通过接线端子进行连接，整体实物连接如下图 21.4.2.6：：

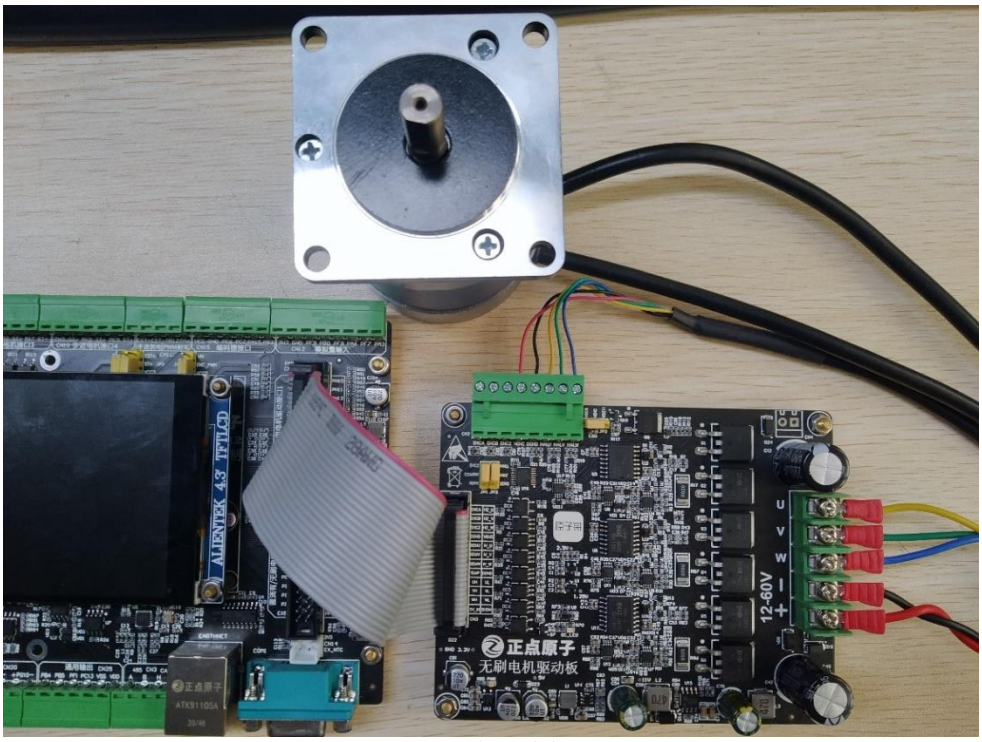


图 21.4.2.6 端子连接驱动板

注意：后续的直流无刷专题教程中，如果没有特别说明实物接线的，均按照上述接线方式（开发板需要单独供电）。至此，直流无刷驱动板的实物连接就介绍完了，有了这些基础之后，我们就可以开始学习直流无刷电机的基础驱动了

第 22 章 有感方波驱动

经过前面的理论分析，是时候一展身手让电机转起来了。需要注意的是关于本书中的无刷章节的代码是同时兼容直流无刷电机（BLDC）和永磁同步电机（PMSM）的。驱动无刷电机我们使用到最简单的方式就是本章的有感方波驱动。

本章分为如下几个小节：

22.1 硬件设计
22.2 程序设计
22.3 下载验证

22.1 硬件设计

1、例程功能

- 本实验使用无刷电机接口一连接无刷电机驱动板
- 当按下 KEY0 一次加速旋转，按一次 KEY1 减速旋转，按下 KEY2 则停止旋转。
- LED0 作为程序运行状态指示灯，闪烁周期 200ms。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3） 定时器：
TIM1_CH1：PA8
TIM1_CH2：PA9
TIM1_CH3：PA10
IO：PB13\14\15
使能引脚：SHDN: PF10
4）无刷电机
5）无刷电机驱动板
6）12-60V 的 DC 电源（建议使用电机额定电压 24V）

3、原理图

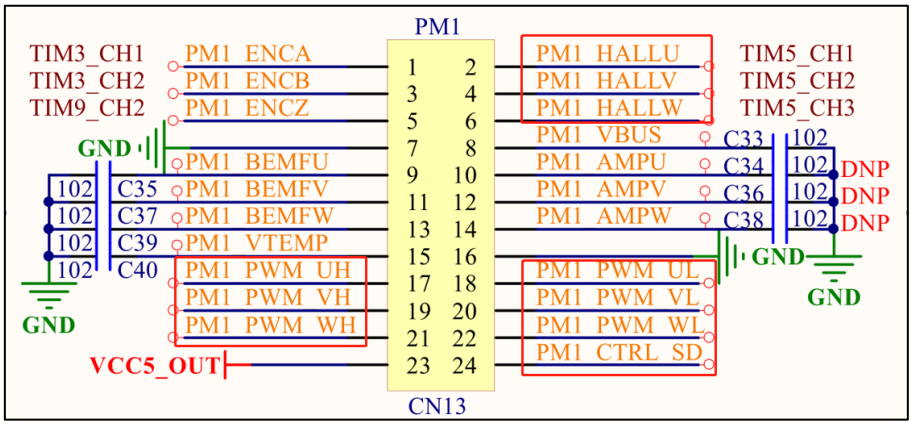


图 22.1.2 无刷电机接口 1 原理图

图 22.1.2 就是我们 DMF407 电机开发板的直流无刷电机接口 1 原理图，本实验所涉及的 IO 如下：

直流无刷驱动板	电机开发板
驱动板 U 相	**PA8、PB13**
驱动板 V 相	**PA9、PB14**
驱动板 W 相	**PA10、PB15**
霍尔 U 相	**PH10**
霍尔 V 相	**PH11**
霍尔 W 相	**PH12**
输出使能	**PF10**

表 22.1.1 无刷相关 IO 口说明

电机与驱动板之间的连接请查看 21.4.2 小节。

驱动板和开发板连接只需要使用排线连接即可，电机与驱动板通过接线端子进行连接，**注意：由于本实验使用霍尔传感器读取转子位置，所以需将无刷驱动板的 JP3 跳帽把 H&Z 与 HALL 短接在一起。**实物连接如下：

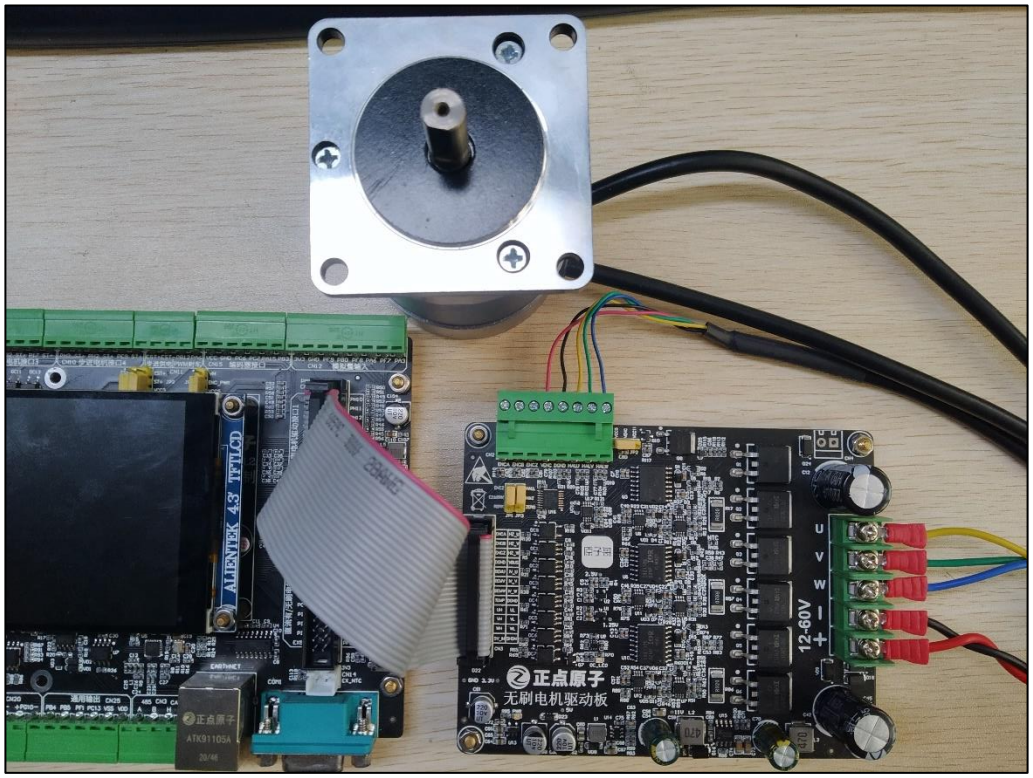


图 22.1.3 开发板&驱动板连接图

22.2 程序设计

22.2.1 无刷电机基本控制的配置步骤

1）初始化定时器以及相关 IO

初始化三相上下桥臂六路 IO、霍尔状态读取三路 IO、SHUTDOWN 引脚以及定时器通道 IO，设置 ARR、PSC，计数方式以及脉冲输出模式等

2）霍尔状态读取

编写霍尔传感器状态读取函数

3）BLDC 相关函数实现

包含电机启停函数，6 步换相控制函数等

4）控制转速

设置旋转方向以及脉冲占空比

5）编写中断服务函数

在中断里边读取霍尔状态，根据方向以及霍尔状态依次导通上下桥臂

22.2.2 程序流程图

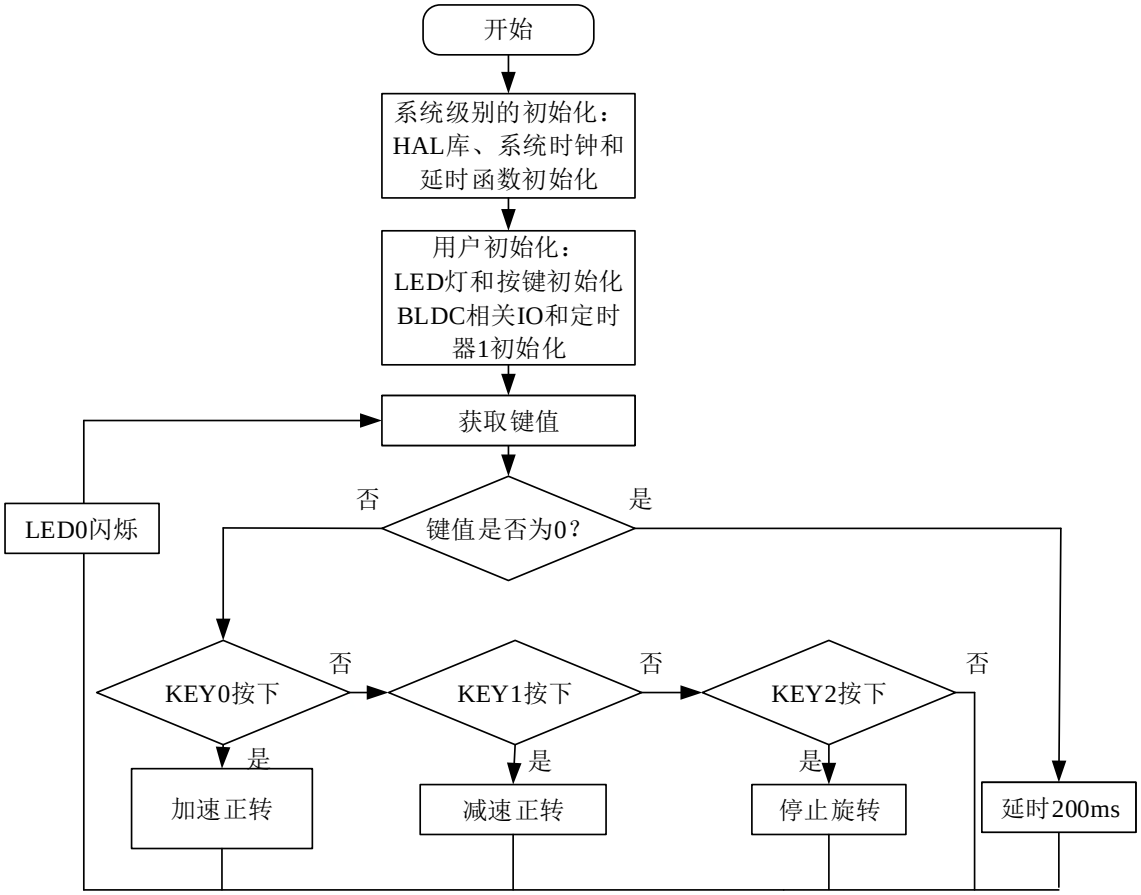


图 22.2.2.1 无刷电机基本控制流程图

22.2.3 程序解析

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。无刷方波有感的控制源码主要有：定时器驱动源码和无刷电机驱动源码四个文件：bldc_tim.c 和 bldc_tim.h 以及 bldc.c 和 bldc.h。源码中都有明确的注释。

首先来看下 bldc_tim.h 头文件的内容：

```
/* 高级定时器 定义 */

/* TIM_CHx 通道 上桥臂 IO 定义 */
#define ATIM_TIMX_PWM_CH1_GPIO_PORT                 GPIOA
#define ATIM_TIMX_PWM_CH1_GPIO_PIN                  GPIO_PIN_8
#define ATIM_TIMX_PWM_CH1_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)        /* PC 口时钟使能 */

#define ATIM_TIMX_PWM_CH2_GPIO_PORT                 GPIOA
#define ATIM_TIMX_PWM_CH2_GPIO_PIN                  GPIO_PIN_9
```

```
#define ATIM_TIMX_PWM_CH2_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)        /* PC 口时钟使能 */

#define ATIM_TIMX_PWM_CH3_GPIO_PORT                 GPIOA
#define ATIM_TIMX_PWM_CH3_GPIO_PIN                  GPIO_PIN_10
#define ATIM_TIMX_PWM_CH3_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)        /* PC 口时钟使能 */

/* 下桥臂 IO 定义 */
#define M1_LOW_SIDE_U_PORT                          GPIOB
#define M1_LOW_SIDE_U_PIN                           GPIO_PIN_13
#define M1_LOW_SIDE_U_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)        /* PC 口时钟使能 */

#define M1_LOW_SIDE_V_PORT                          GPIOB
#define M1_LOW_SIDE_V_PIN                           GPIO_PIN_14
#define M1_LOW_SIDE_V_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)        /* PC 口时钟使能 */

#define M1_LOW_SIDE_W_PORT                          GPIOB
#define M1_LOW_SIDE_W_PIN                           GPIO_PIN_15
#define M1_LOW_SIDE_W_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)        /* PC 口时钟使能 */

/* 复用到 TIM1 */
#define ATIM_TIMX_PWM_CHY_GPIO_AF                   GPIO_AF1_TIM1

#define ATIM_TIMX_PWM                               TIM1
#define ATIM_TIMX_PWM_IRQn                          TIM1_UP_TIM10_IRQn
#define ATIM_TIMX_PWM_IRQHandler                    TIM1_UP_TIM10_IRQHandler
#define ATIM_TIMX_PWM_CH1                           TIM_CHANNEL_1
/* 通道 Y,  1<= Y <=4 */
#define ATIM_TIMX_PWM_CH2                           TIM_CHANNEL_2
/* 通道 Y,  1<= Y <=4 */
#define ATIM_TIMX_PWM_CH3                           TIM_CHANNEL_3
/* 通道 Y,  1<= Y <=4 */
#define ATIM_TIMX_PWM_CHY_CLK_ENABLE()
do{  __HAL_RCC_TIM1_CLK_ENABLE(); }while(0)         /* TIM8 时钟使能 */
```

可以把上面的宏定义分成三部分：第一部分是定时器 1 输出通道 1、2、3 对应的 IO 口的宏定义，同时也是控制逆变器的三个上桥臂的 IO 宏定义；第二部分则是控制逆变器的三个下桥臂的 IO 宏定义，第三部分是关于定时器 1 的通道、时钟等相关宏定义。

下面我们来看下 bldc_tim.c 代码。

```
/*
 * @brief       高级定时器 TIMX  PWM 输出初始化函数
 * @note
 *              高级定时器的时钟来自 APB2, 而 PCLK2 = 168Mhz, 我们设置 PPRE2 不分频, 因此
 *              高级定时器时钟 = 168Mhz
 *              定时器溢出时间计算方法: Tout = ((arr + 1) * (psc + 1)) / Ft us.
 *              Ft=定时器工作频率,单位:Mhz
 *
 * @param       arr: 自动重装值
 * @param       psc: 时钟预分频数
 * @retval      无
 */
void atim_timx_oc_chy_init(uint16_t arr, uint16_t psc)
{
    ATIM_TIMX_PWM_CHY_CLK_ENABLE();       /* TIMX 时钟使能 */


    g_atimx_handle.Instance = ATIM_TIMX_PWM;                    /* 定时器 x */
    g_atimx_handle.Init.Prescaler = psc;                        /* 定时器分频 */
    g_atimx_handle.Init.CounterMode = TIM_COUNTERMODE_UP;       /* 向上计数模式 */
```

```
    g_atimx_handle.Init.Period = arr;                                   /* 自动重装载值 */
    g_atimx_handle.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;   /* 分频因子 */
    g_atimx_handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    g_atimx_handle.Init.RepetitionCounter = 0;                          /* 开始时不计数*/
    HAL_TIM_PWM_Init(&g_atimx_handle);                                  /* 初始化 PWM */

    g_atimx_oc_chy_handle.OCMode = TIM_OCMODE_PWM1;                     /* 模式选择 PWM1*/
    g_atimx_oc_chy_handle.Pulse = 0;
    g_atimx_oc_chy_handle.OCPolarity = TIM_OCPOLARITY_HIGH;     /* 比较极性为低 */
    g_atimx_oc_chy_handle.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    g_atimx_oc_chy_handle.OCFastMode = TIM_OCFAST_DISABLE;
    g_atimx_oc_chy_handle.OCIdleState = TIM_OCIDLESTATE_RESET;
    g_atimx_oc_chy_handle.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
                              ATIM_TIMX_PWM_CH1);                       /* 配置通道 1 */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
                              ATIM_TIMX_PWM_CH2);                       /* 配置通道 2 */
    HAL_TIM_PWM_ConfigChannel(&g_atimx_handle, &g_atimx_oc_chy_handle,
                              ATIM_TIMX_PWM_CH3);                       /* 配置通道 3 */

    /* 开启定时器通道 1 输出 PWM */
    HAL_TIM_PWM_Start(&g_atimx_handle,TIM_CHANNEL_1);

    /* 开启定时器通道 2 输出 PWM */
    HAL_TIM_PWM_Start(&g_atimx_handle,TIM_CHANNEL_2);

    /* 开启定时器通道 3 输出 PWM */
    HAL_TIM_PWM_Start(&g_atimx_handle,TIM_CHANNEL_3);
}
```

函数 atim_timx_oc_chy_init 的主要功能，就是初始化定时器 1 的 ARR 和 PSC 等参数，然后通过调用函数 HAL_TIM_PWM_ConfigChannel 设置 TIM1_CH1、2、3 的 PWM 模式以及比较值等参数，最后通过调用函数 HAL_TIM_PWM_Start 来使能 TIM1 以及使能 PWM 通道 TIM1_CH1、2、3 输出。定时器通道的 GPIO 口初始化和定时器使能等程序放到回调函数 HAL_TIM_PWM_MspInit 中，注意：我们为了程序易读，将下桥臂的 IO 初始化一起放在该回调函数里实现了，如下：

```
/**
 * @brief       定时器底层驱动，时钟使能，引脚配置
 *              此函数会被 HAL_TIM_PWM_Init()调用
 * @param       htim:定时器句柄
 * @retval      无
 */
void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == ATIM_TIMX_PWM)
    {
        GPIO_InitTypeDef gpio_init_struct;
        ATIM_TIMX_PWM_CHY_CLK_ENABLE();                 /* 开启通道 y 的 CPIO 时钟 */
        ATIM_TIMX_PWM_CH1_GPIO_CLK_ENABLE();            /* IO 时钟使能 */
        ATIM_TIMX_PWM_CH2_GPIO_CLK_ENABLE();            /* IO 时钟使能 */
        ATIM_TIMX_PWM_CH3_GPIO_CLK_ENABLE();            /* IO 时钟使能 */

        M1_LOW_SIDE_U_GPIO_CLK_ENABLE();                /* IO 时钟使能 */
        M1_LOW_SIDE_V_GPIO_CLK_ENABLE();                /* IO 时钟使能 */
        M1_LOW_SIDE_W_GPIO_CLK_ENABLE();                /* IO 时钟使能 */

        /* 下桥臂的 IO 初始化 */
        gpio_init_struct.Pin = M1_LOW_SIDE_U_PIN;
        gpio_init_struct.Pull = GPIO_NOPULL;
        gpio_init_struct.Speed = GPIO_SPEED_HIGH;
        gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;    /* 推挽输出模式 */
        HAL_GPIO_Init(M1_LOW_SIDE_U_PORT, &gpio_init_struct);
```

```
        gpio_init_struct.Pin = M1_LOW_SIDE_V_PIN;
        HAL_GPIO_Init(M1_LOW_SIDE_V_PORT, &gpio_init_struct);

        gpio_init_struct.Pin = M1_LOW_SIDE_W_PIN;
        HAL_GPIO_Init(M1_LOW_SIDE_W_PORT, &gpio_init_struct);


        /* 上桥臂即定时器 IO 初始化 */
        gpio_init_struct.Pin = ATIM_TIMX_PWM_CH1_GPIO_PIN;   /* 通道 y 的 CPIO 口 */
        gpio_init_struct.Mode = GPIO_MODE_AF_PP;             /* 复用推完输出 */
        gpio_init_struct.Pull = GPIO_NOPULL;                 /* 上拉 */
        gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;       /* 高速 */
        gpio_init_struct.Alternate = ATIM_TIMX_PWM_CHY_GPIO_AF;/* 端口复用 */
        HAL_GPIO_Init(ATIM_TIMX_PWM_CH1_GPIO_PORT, &gpio_init_struct);

        gpio_init_struct.Pin = ATIM_TIMX_PWM_CH2_GPIO_PIN;   /* 通道 y 的 CPIO 口 */
        HAL_GPIO_Init(ATIM_TIMX_PWM_CH2_GPIO_PORT, &gpio_init_struct);

        gpio_init_struct.Pin = ATIM_TIMX_PWM_CH3_GPIO_PIN;   /* 通道 y 的 CPIO 口 */
        HAL_GPIO_Init(ATIM_TIMX_PWM_CH3_GPIO_PORT, &gpio_init_struct);

        HAL_NVIC_SetPriority(TIM1_UP_TIM10_IRQn, 2, 2);
        HAL_NVIC_EnableIRQ(TIM1_UP_TIM10_IRQn);

    }
}
```

程序使用 H_PWM - L_ON 方式驱动，因此只有 3 路上桥臂使用 PWM 通道的初始化，三路下桥臂使用的普通 IO 输出。相应的 IO 也在头文件进行了定义。

接着看下 bldc.h 的内容：

```
/* 半桥芯片的刹车引脚 */
#define SHUTDOWN_PIN                    GPIO_PIN_10     /* PF10 */
#define SHUTDOWN_PIN_GPIO               GPIOF
#define SHUTDOWN_PIN_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOF_CLK_ENABLE(); }while(0)            /* PF 口时钟使能 */

#define SHUTDOWN_EN
HAL_GPIO_WritePin(SHUTDOWN_PIN_GPIO,SHUTDOWN_PIN,GPIO_PIN_SET);
#define SHUTDOWN_OFF
HAL_GPIO_WritePin(SHUTDOWN_PIN_GPIO,SHUTDOWN_PIN,GPIO_PIN_RESET);

/**********************************************************************/
/* 霍尔传感器接口 */

#define HALL1_TIM_CH1_PIN               GPIO_PIN_10     /* U */
#define HALL1_TIM_CH1_GPIO              GPIOH
#define HALL1_U_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOH_CLK_ENABLE(); }while(0)            /* PH 口时钟使能 */

#define HALL1_TIM_CH2_PIN               GPIO_PIN_11     /* V */
#define HALL1_TIM_CH2_GPIO              GPIOH
#define HALL1_V_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOH_CLK_ENABLE(); }while(0)            /* PH 口时钟使能 */

#define HALL1_TIM_CH3_PIN               GPIO_PIN_12     /* W */
#define HALL1_TIM_CH3_GPIO              GPIOH
#define HALL1_W_GPIO_CLK_ENABLE()
do{  __HAL_RCC_GPIOH_CLK_ENABLE(); }while(0)            /* PH 口时钟使能 */


/**********************************************************************/
#define MAX_PWM_DUTY    (((10000) - 1)*0.96)            /* 最大占空比限制 */
```

上述代码主要分为三部分：第一部分为逆变电路的半桥芯片刹车 IO 的宏定义；第二部分

为读取三路霍尔输出状态的 IO 宏定义；第三部分为 PWM 占空比的最大限制宏定义，用于对速度进行限制。

接着看下 bldc.c 文件，该文件主要包含了 BLDC 初始化函数、霍尔传感器状态读取函数、无刷电机的启停函数、以及基础的六步换向控制函数。函数如下：

```
/**
 * @brief       初始化 BLDC 相关 IO 口, 并使能时钟
 * @param       无
 * @retval      无
 */
void bldc_init(uint16_t arr, uint16_t psc)
{
    GPIO_InitTypeDef gpio_init_struct;

    SHUTDOWN_PIN_GPIO_CLK_ENABLE();

    gpio_init_struct.Pin = SHUTDOWN_PIN;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;
    gpio_init_struct.Pull = GPIO_NOPULL;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(SHUTDOWN_PIN_GPIO, &gpio_init_struct);

    hall_gpio_init();                       /* 霍尔接口初始化 */
    atim_timx_oc_chy_init(arr,  psc);       /* 定时器初始化 */
}
```

该函数主要实现刹车引脚和霍尔接口以及定时器的初始化。内容比较简单，就不多说了，接着来看下霍尔状态读取函数：

```
/**
 * @brief       获取霍尔传感器引脚状态
 * @param       无
 * @retval      霍尔传感器引脚状态
 */
uint32_t hallsensor_get_state(uint8_t motor_id)
{
    __IO static uint32_t State ;
    State  = 0;
    if(motor_id == MOTOR_1)
    {
        if(HAL_GPIO_ReadPin(HALL1_TIM_CH1_GPIO,HALL1_TIM_CH1_PIN) !=
           GPIO_PIN_RESET)  /* 霍尔 1 传感器状态获取 */
        {
            State |= 0x01U;
        }
        if(HAL_GPIO_ReadPin(HALL1_TIM_CH2_GPIO,HALL1_TIM_CH2_PIN) !=
           GPIO_PIN_RESET)  /* 霍尔 2 传感器状态获取 */
        {
            State |= 0x02U;
        }
        if(HAL_GPIO_ReadPin(HALL1_TIM_CH3_GPIO,HALL1_TIM_CH3_PIN) !=
           GPIO_PIN_RESET)  /* 霍尔 3 传感器状态获取 */
        {
            State |= 0x04U;
        }
    }
    return State;
}
```

该函数有 1 个入口参数，该参数代表当前控制哪个电机接口，方便大家控制多个电机时使用。接着是霍尔状态读取，将霍尔 1、2、3 各自的读取值，组合在一起。注意霍尔 3 最高位，霍尔 2 次之，霍尔 1 最低位。然后返回该霍尔组合值。

接着看下电机的启停函数

```
/* 关闭电机运转 */
void stop_motor1(void)
```

```
{
    /* 关闭半桥芯片输出 */
    SHUTDOWN_OFF;
    /* 关闭 PWM 输出 */
    HAL_TIM_PWM_Stop(&g_atimx_handle,TIM_CHANNEL_1);
    HAL_TIM_PWM_Stop(&g_atimx_handle,TIM_CHANNEL_2);
    HAL_TIM_PWM_Stop(&g_atimx_handle,TIM_CHANNEL_3);
    /* 上下桥臂全部关断 */
    g_atimx_handle.Instance->CCR2 = 0;
    g_atimx_handle.Instance->CCR1 = 0;
    g_atimx_handle.Instance->CCR3 = 0;
    HAL_GPIO_WritePin(M1_LOW_SIDE_U_PORT,M1_LOW_SIDE_U_PIN,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_V_PORT,M1_LOW_SIDE_V_PIN,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_W_PORT,M1_LOW_SIDE_W_PIN,GPIO_PIN_RESET);
}

/* 开启电机运转 */
void start_motor1(void)
{
    SHUTDOWN_EN;
    /* 使能 PWM 输出 */
    HAL_TIM_PWM_Start(&g_atimx_handle,TIM_CHANNEL_1);
    HAL_TIM_PWM_Start(&g_atimx_handle,TIM_CHANNEL_2);
    HAL_TIM_PWM_Start(&g_atimx_handle,TIM_CHANNEL_3);
}
```

电机开启函数，开启半桥芯片正常输出，即使能 shutdown 引脚，然后开启 TIM1 的三个通道的 PWM 输出。

电机关闭函数，关闭半桥芯片正常输出，即失能 shutdown 引脚，然后关闭 TIM1 通道的 PWM 输出，并关断所有上下桥臂。

接着是我们的六步换相控制函数：

```
/* 上下桥臂的导通情况，共 6 种，也称为 6 步换向 */
void m1_uhvl(void)
{
    g_atimx_handle.Instance->CCR2 = 0;
    g_atimx_handle.Instance->CCR1 = g_bldc_motor1.pwm_duty;   /* U 相上桥臂 PWM */
    g_atimx_handle.Instance->CCR3 = 0;
    /* V 相下桥臂导通 */
    HAL_GPIO_WritePin(M1_LOW_SIDE_V_PORT,M1_LOW_SIDE_V_PIN,GPIO_PIN_SET);
    /* U 相下桥臂关闭 */
    HAL_GPIO_WritePin(M1_LOW_SIDE_U_PORT,M1_LOW_SIDE_U_PIN,GPIO_PIN_RESET);
    /* W 相下桥臂关闭 */
    HAL_GPIO_WritePin(M1_LOW_SIDE_W_PORT,M1_LOW_SIDE_W_PIN,GPIO_PIN_RESET);
}

void m1_uhwl(void)
{
    g_atimx_handle.Instance->CCR2 = 0;
    g_atimx_handle.Instance->CCR1 = g_bldc_motor1.pwm_duty;
    g_atimx_handle.Instance->CCR3 = 0;
    HAL_GPIO_WritePin(M1_LOW_SIDE_W_PORT,M1_LOW_SIDE_W_PIN,GPIO_PIN_SET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_U_PORT,M1_LOW_SIDE_U_PIN,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_V_PORT,M1_LOW_SIDE_V_PIN,GPIO_PIN_RESET);
}

void m1_vhwl(void)
{
    g_atimx_handle.Instance->CCR1=0;
    g_atimx_handle.Instance->CCR2 = g_bldc_motor1.pwm_duty;
    g_atimx_handle.Instance->CCR3=0;
    HAL_GPIO_WritePin(M1_LOW_SIDE_W_PORT,M1_LOW_SIDE_W_PIN,GPIO_PIN_SET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_U_PORT,M1_LOW_SIDE_U_PIN,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_V_PORT,M1_LOW_SIDE_V_PIN,GPIO_PIN_RESET);
}
```

```
void m1_vhul(void)
{
    g_atimx_handle.Instance->CCR1 = 0;
    g_atimx_handle.Instance->CCR2 = g_bldc_motor1.pwm_duty;
    g_atimx_handle.Instance->CCR3 = 0;
    HAL_GPIO_WritePin(M1_LOW_SIDE_U_PORT,M1_LOW_SIDE_U_PIN,GPIO_PIN_SET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_V_PORT,M1_LOW_SIDE_V_PIN,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_W_PORT,M1_LOW_SIDE_W_PIN,GPIO_PIN_RESET);
}


void m1_whul(void)
{
    g_atimx_handle.Instance->CCR2 = 0;
    g_atimx_handle.Instance->CCR3 = g_bldc_motor1.pwm_duty;
    g_atimx_handle.Instance->CCR1 = 0;
    HAL_GPIO_WritePin(M1_LOW_SIDE_U_PORT,M1_LOW_SIDE_U_PIN,GPIO_PIN_SET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_V_PORT,M1_LOW_SIDE_V_PIN,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_W_PORT,M1_LOW_SIDE_W_PIN,GPIO_PIN_RESET);
}

void m1_whvl(void)
{
    g_atimx_handle.Instance->CCR2 = 0;
    g_atimx_handle.Instance->CCR3 = g_bldc_motor1.pwm_duty;
    g_atimx_handle.Instance->CCR1 = 0;
    HAL_GPIO_WritePin(M1_LOW_SIDE_V_PORT,M1_LOW_SIDE_V_PIN,GPIO_PIN_SET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_U_PORT,M1_LOW_SIDE_U_PIN,GPIO_PIN_RESET);
    HAL_GPIO_WritePin(M1_LOW_SIDE_W_PORT,M1_LOW_SIDE_W_PIN,GPIO_PIN_RESET);
}
```

上述代码为 6 步换相控制代码，由于我们使用的驱动方式为 H_PWM－L_ON，即上桥臂由定时器通道输出的 PWM 控制，下桥臂使用的普通 IO 直接控制，所以上桥臂通过修改对应定时器通道的比较值，进而修改占空比，既可以控制 MOS 的开关也可以控制电机速度；下桥臂直接通过 IO 拉高拉低控制开关。

讲到这里，bldc.c 文件里的函数就讲得差不多了，接着来看下我们的核心内容，定时器的中断回调函数：

```
/**
 * @brief       定时器中断回调
 * @param       无
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    uint8_t bldc_dir=0;
    if(htim->Instance == ATIM_TIMX_PWM)         /* 55us */
    {
#ifdef H_PWM_L_ON
        if(g_bldc_motor1.run_flag == RUN)
        {
            if(g_bldc_motor1.dir == CW)         /* 正转 */
            {
                /* 顺序 6,2,3,1,5,4 */
                g_bldc_motor1.step_sta = hallsensor_get_state(MOTOR_1);
            }
            else                                /* 反转 */
            {
                /* 顺序 5,1,3,2,6,4 */
                g_bldc_motor1.step_sta = 7 - hallsensor_get_state(MOTOR_1);
            }

            if((g_bldc_motor1.step_sta <= 6)&&(g_bldc_motor1.step_sta >= 1))
            {
```

```
            pfunclist_m1[g_bldc_motor1.step_sta-1]();

        }
        else                            /* 霍尔传感器错误、接触不良、断开等情况 */
        {
            stop_motor1();
            g_bldc_motor1.run_flag = STOP;
        }
        /* 检测到霍尔值变换，代表换向一次  */
        if(g_bldc_motor1.step_last != g_bldc_motor1.step_sta)
        {
            g_bldc_motor1.hall_keep_t=0;
            bldc_dir = check_hall_dir(&g_bldc_motor1);
            if(bldc_dir == CCW)
            {
                g_bldc_motor1.pos -=1;
            }
            else if(bldc_dir == CW)
            {
                g_bldc_motor1.pos +=1;
            }
            g_bldc_motor1.step_last = g_bldc_motor1.step_sta;
        }
        else if(g_bldc_motor1.run_flag == RUN)   /* 运行且霍尔保持时 */
        {
            g_bldc_motor1.hall_keep_t++ /* 换向一次所需计数值（时间） 单位 1/18k */

        }
    }

#endif
    }
}
```

在中断回调函数里，主要根据霍尔状态调用不同的换相函数，霍尔值与绕组的导通真值表可以查看表 21.3.2，我们也是依据这个表来实现该代码的。通过读取到的霍尔状态组合值，导通对应的上下桥臂，我们这里使用函数指针的方式，指向函数列表 pfunclist_m1 的成员，这样可以使代码更加简洁，当然大家也可以使用 switch 语句实现。

有了上述的逻辑，操作起来无刷电机就比较简单了。最后来看下 main 函数。

```
int main(void)
{
    uint8_t key,t;
    char buf[32];
    int16_t pwm_duty_temp=0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    bldc_init(10000-1,0);
    bldc_ctrl(MOTOR_1,CCW,0);                   /* 初始无刷电机接口 1 速度 */
    HAL_TIM_Base_Start_IT(&g_atimx_handle); /* 启动高级定时器 1*/

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10, 10, 200, 16, 16, "BLDC Motor Test",g_point_color);
    lcd_show_string(10, 30, 200, 16, 16, "KEY0:Start forward", g_point_color);
    lcd_show_string(10, 50, 200, 16, 16, "KEY1:Start backward", g_point_color);
    lcd_show_string(10, 70, 200, 16, 16, "KEY2:Stop", g_point_color);

    while (1)
```

```
    {
        t++;
        if(t % 20 == 0)
        {
            sprintf(buf,"PWM_Duty:%.1f%%",(float)((g_bldc_motor1.pwm_duty/
                    MAX_PWM_DUTY)*100));/* 显示控制 PWM 占空比 */
            lcd_show_string(10,110,200,16,16,buf,g_point_color);
            LED0_TOGGLE();                /* LED0(红灯) 翻转 */
        }

        key = key_scan(0);
        if(key == KEY0_PRES)
        {
            pwm_duty_temp+=500;
            if(pwm_duty_temp>=MAX_PWM_DUTY/2)
                pwm_duty_temp=MAX_PWM_DUTY/2;
            if(pwm_duty_temp>0)
            {
                g_bldc_motor1.pwm_duty= pwm_duty_temp;
                g_bldc_motor1.dir=CCW;
            }
            else
            {
                g_bldc_motor1.pwm_duty=-pwm_duty_temp;
                g_bldc_motor1.dir=CW;
            }
            g_bldc_motor1.run_flag=RUN;  /* 开启运行 */
            start_motor1();              /* 开启运行 */
        }
        if(key == KEY1_PRES)             /* 按下 KEY1 开启电机 */
        {
            pwm_duty_temp-=500;
            if(pwm_duty_temp<=-MAX_PWM_DUTY/2)
                pwm_duty_temp=-MAX_PWM_DUTY/2;
            if(pwm_duty_temp < 0)
            {
                g_bldc_motor1.pwm_duty=-pwm_duty_temp;
                g_bldc_motor1.dir=CW;
            }
            else
            {
                g_bldc_motor1.pwm_duty=pwm_duty_temp;
                g_bldc_motor1.dir=CCW;
            }
            g_bldc_motor1.run_flag=RUN;  /* 开启运行 */
            start_motor1();              /* 运行电机 */
        }
        if(key == KEY2_PRES)             /* 按下 KEY2 关闭电机 */
        {
            stop_motor1();               /* 停机 */
            g_bldc_motor1.run_flag=STOP; /* 标记停机 */
            pwm_duty_temp=0;
            g_bldc_motor1.pwm_duty=0;
        }
        delay_ms(10);
    }
}
```

主函数首先调用 bldc_init 函数初始化定时器的 PWM 参数，然后调用 bldc_ctrl 函数控制无刷电机的初始化速度和方向，接着启动定时器，接下来就可以使用无刷电机了。主函数循环通过检测按键来控制无刷电机的旋转和停止，当按下 KEY0 增大占空比，按下 KEY1 减小占空比，按下 KEY2 关闭电机。

22.3 下载验证

硬件连接好之后下载程序到开发板，液晶屏和串口都会显示提示信息，点击按键 KEY0 一次可以看到定时器比较值递增 500（加速），电机开始旋转，点击一次按键 KEY1 则可以看到定时器比较值递减 500（减速），点击按键 KEY2 则可以随时停止无刷电机。大家可以自己下载程序并点击按键测试效果。

第 23 章 无刷电机电压温度电流采集

前面我们已经学会了无刷电机的基本驱动，本章我们将利用无刷驱动板的电路功能，来检测电机的三相电流以及驱动板的电源电压和温度。

本章分为如下几个小节：

23.1 驱动板硬件电路原理分析
23.2 硬件设计
23.3 程序设计
23.4 下载验证

23.1 驱动板硬件电路原理分析

下面我们主要分为三部分来讲解驱动板的硬件电路，包括：三相电流采集电路分析、电源电压采集电路分析、温度采集电路分析共三部分。

23.1.1 三相电流采集电路分析

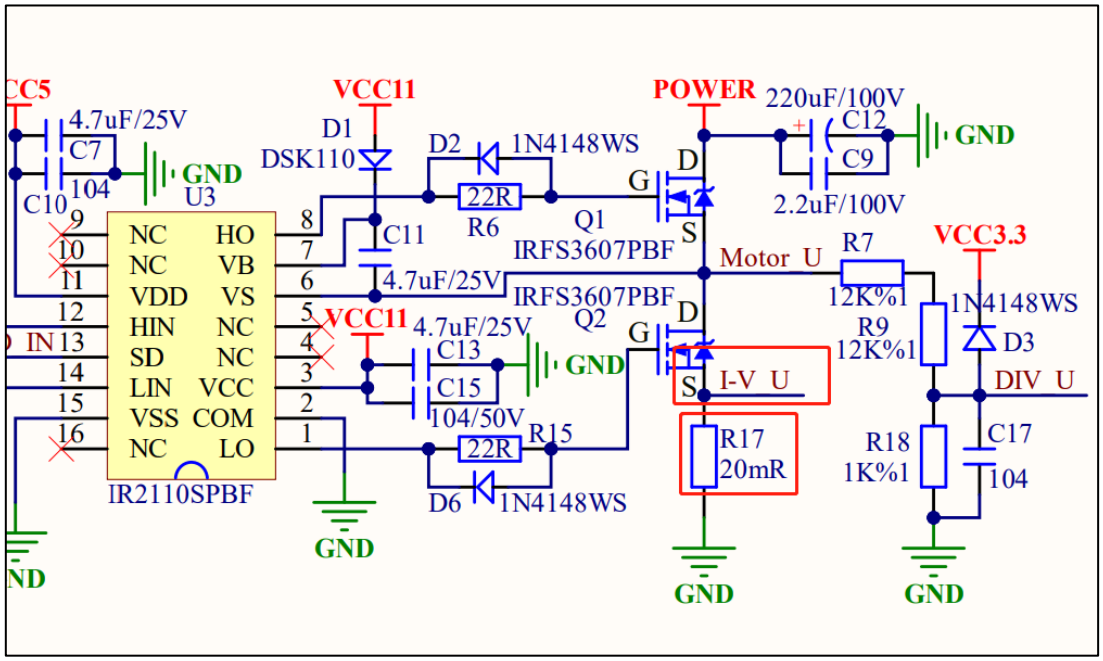


图 23.1.1.1 U 相电流采集电路

图 23.1.1.1 为 U 相电流采集电路（U、V、W 三相同理，这里只以 U 相为例），其中 R17 为采样电阻（20mR），当有电流 I 流过采样电阻时，采样电阻上就会产生一个电压，电压大小为： I*0.02Ω（20mR）。我们使用 I-V 表示电流 I 经过采样电阻后形成的电压，所以 U 相的电压，我们使用 I-V_U 表示。

那么现在只需要测出电压 I-V_U 的值，就可以换算出 U 相电流，但是这里我们发现一个问题，假设 U 相电流为 10A，此时电压 I-V_U = 10 * 0.02 V = 0.2V，同理当电流为 20A 时，求得的电压值 I-V_U = 0.4V，可以看到当电流比较大的情况下，求得的电压值是非常小的，不利于我们的 ADC 采集，考虑到 ADC 的精度问题，所以我还需将电压经过差分放大电路放大后在来采集，如下图：

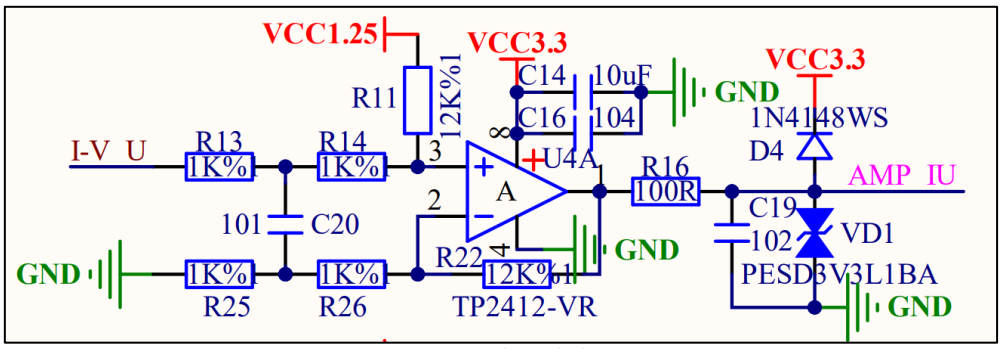


图 23.1.1.2 差分放大电路

可以从图 23.1.1.2 中得出该差分放大电路的放大倍数为：Diff = 12K/（1K+1K）= 6 倍，并有一抬升电压 1.25V，所以电压放大后的输出为： AMP_IU = 6 * (0.02*I) + 1.25V。由于 AMP_IU 是接到电机开发板的 ADC 引脚的，所以该值是已知的，此时就可以计算出 U 相电流 I 的值。

23.1.2 电源电压采集电路分析

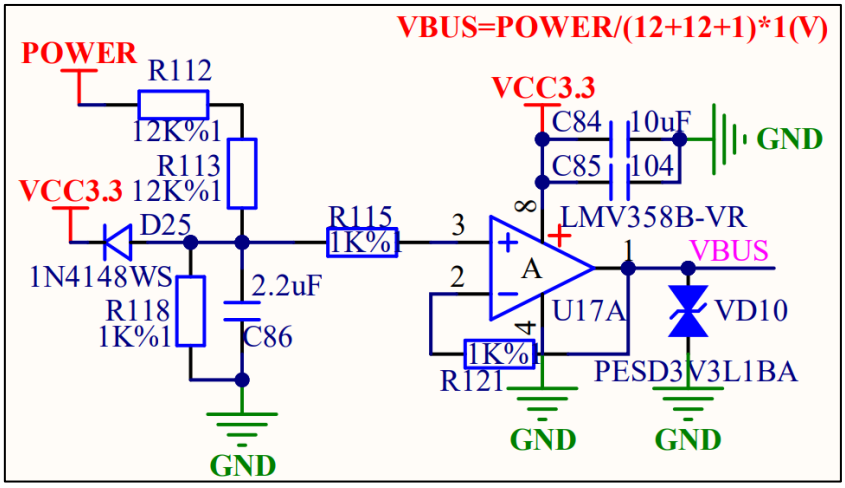


图 23.1.2.1 电源电压采集电路

图 23.1.2.1 为驱动板板载的电源电压检测电路。驱动板的供电电压 POWER 经过电阻分压后进入 LMV358 的 A 路运放，该运放为电压跟随电路，所以 1 脚输出电压 VBUS 和电阻分压的值相同，即 VBUS = POWER/(12K+12K+1K)*1K，当 ADC 采集到 VBUS 电压后就可以计算出 POWER 电压。

23.1.3 温度采集电路分析

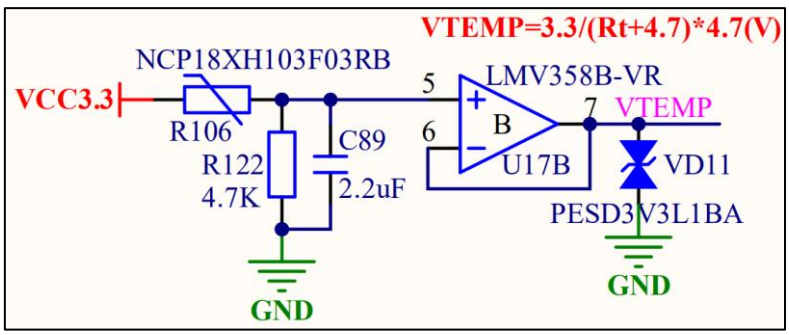


图 23.1.3.1 温度采集电路

图 23.1.3.1 为温度检测电路，可以看到，NTC 电阻和固定的 4.7K 电阻分压，然后接入 B 路运放，B 路运放也是电压跟随电路，所以此时 VTMEP = 3.3V/(Rt+4.7K)*4.7K，通过

ADC 采集 VTMEP 的电压后就可以算出 Rt 的值，其中 Rt 就表示 NTC 电阻在当前温度下的电阻值。从 NTC 电阻的数据手册可以找到温度计算公式：T1=1 / (In (Rt/R0) / B + 1/T2)，这里的 T1 和 T2 指的是 K 度即开尔文温度，K 度 = 273.15(绝对温度) + 摄氏度；其中：

- T2 ：273.15 + 25（25 摄氏度下的 K 度值）
- T1 ：实际温度
- Rt ：热敏电阻在 T1 温度下的阻值；
- R0 ：热敏电阻在 T2 常温下的标称阻值；
- B 值：热敏电阻的重要恒定参数；

其中 T1 是我们需要求解的温度值，Rt 的大小由前面的温度采集电路可以算出，其他的参数都是可以在数据手册中查找得出，代入温度求解公式即可算出 T1 温度值，注意该值为 K 度。

注意：我们所使用的温度换算公式以及相关参数均来自《NCP18XH103F03RB 热敏电阻.pdf》这个数据手册，路径：A 盘→7，硬件资料→2，芯片资料文件夹下可以找到。

23.2 硬件设计

1、例程功能

- 本实验使用无刷电机接口一连接无刷电机驱动板
- 当按下 KEY0 一次加速旋转，按一次 KEY1 减速旋转，按下 KEY2 则停止旋转。
- LED0 作为程序运行状态指示灯，闪烁周期 200ms。
- 测量并在 LCD 和串口显示无刷电机电压、电流和温度

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
4） 定时器：
TIM1_CH1：PA8
TIM1_CH2：PA9
TIM1_CH3：PA10
IO：PB13\14\15
使能引脚：SHDN: PF10
4）无刷电机
5）无刷电机驱动板
6）12-60V 的 DC 电源
7）ADC：ADC1_CH9：PB1
ADC1_CH0：PA0
ADC1_CH8：PB0
ADC1_CH6：PA6
ADC1_CH3：PA3

3、原理图

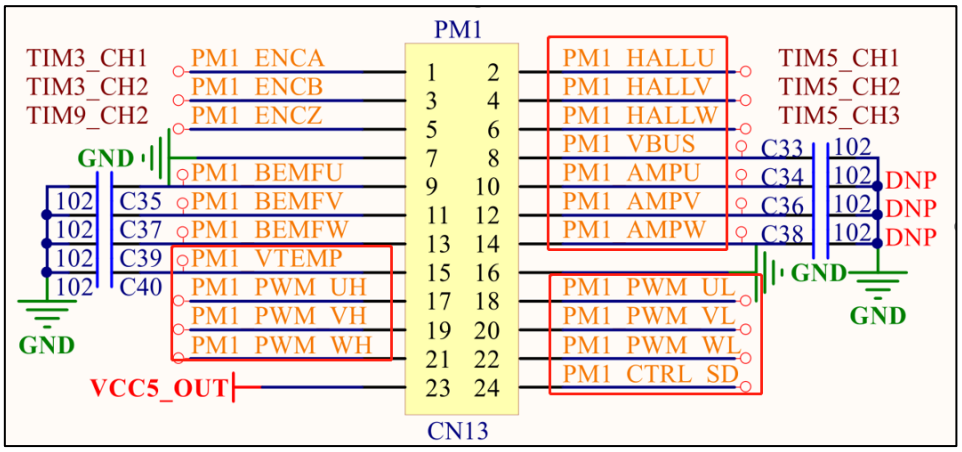


图 23.2.1 无刷电机接口 1 原理图

相较于上一章，本次实验多了 5 个模拟采集引脚，分别是：**电源电压（ADC1_CN9/PB1）、温度（ADC1_CN0/PA0）、三相电流 UVW（ADC1_CN8/PB0、ADC1_CN6/PA6、ADC1_CN3/PA3）**。接口所涉及的 IO 如下：

直流无刷驱动板	IO 口
驱动板 U 相	PA8、PB13
驱动板 V 相	PA9、PB14
驱动板 W 相	PA10、PB15
霍尔 U 相	PH10
霍尔 V 相	PH11
霍尔 W 相	PH12
输出使能	PF10
电压	**PB1**
温度	**PA0**
电流 U 相	**PB0**
电流 V 相	**PA6**
电流 W 相	**PA3**

表 23.2.1 无刷相关 IO 口说明

本章节的硬件接线部分和基础驱动一模一样，注意：驱动板的 JP3 跳线帽短接 H&Z 与 HALL，实物连接如下。

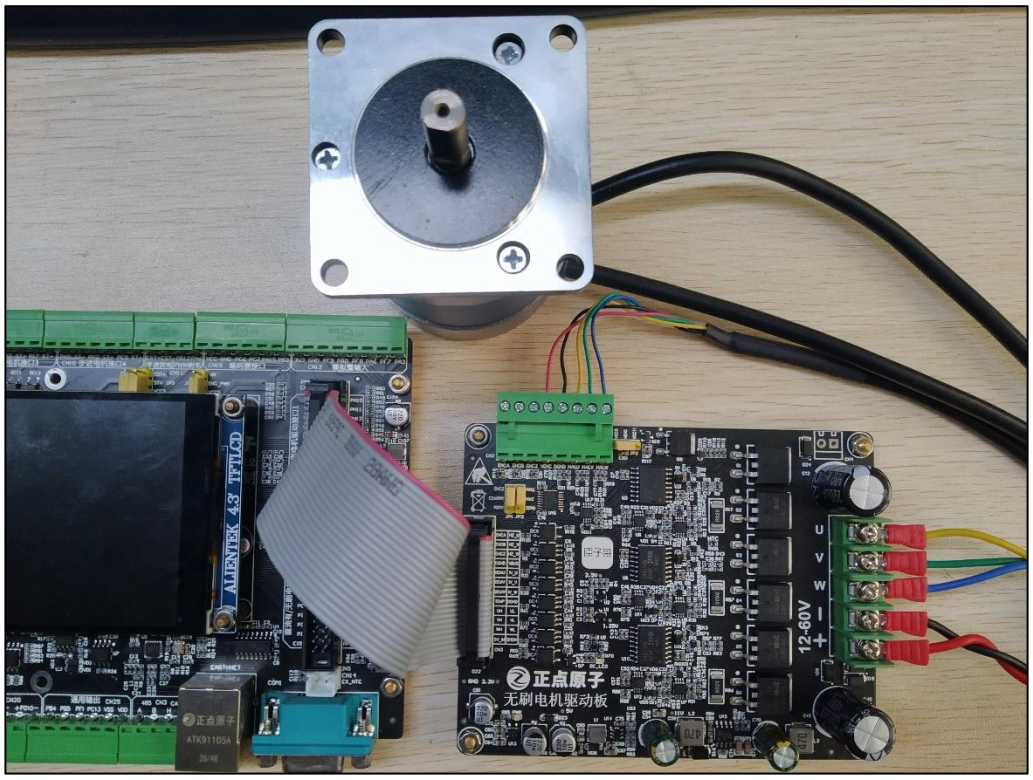


图 23.2.2 开发板&驱动板连接图

23.3 程序设计

23.3.1 无刷电机参数采集的配置步骤

1）实现电机基本驱动

实现电机的基础驱动函数，启停、6 步换向组合等等

2）初始化 ADC&DMA

初始化 ADC 通道的 IO，设置 ADC 工作方式、DMA 等等

3）转速控制

设置电机旋转方向及 PWM 占空比

4）编写中断服务函数

PWM 中断用于换向控制、采集电机未开始转时的基准电压，在将旋转后得到的电压进行相减得到实际电压等等

23.3.2 程序流程图

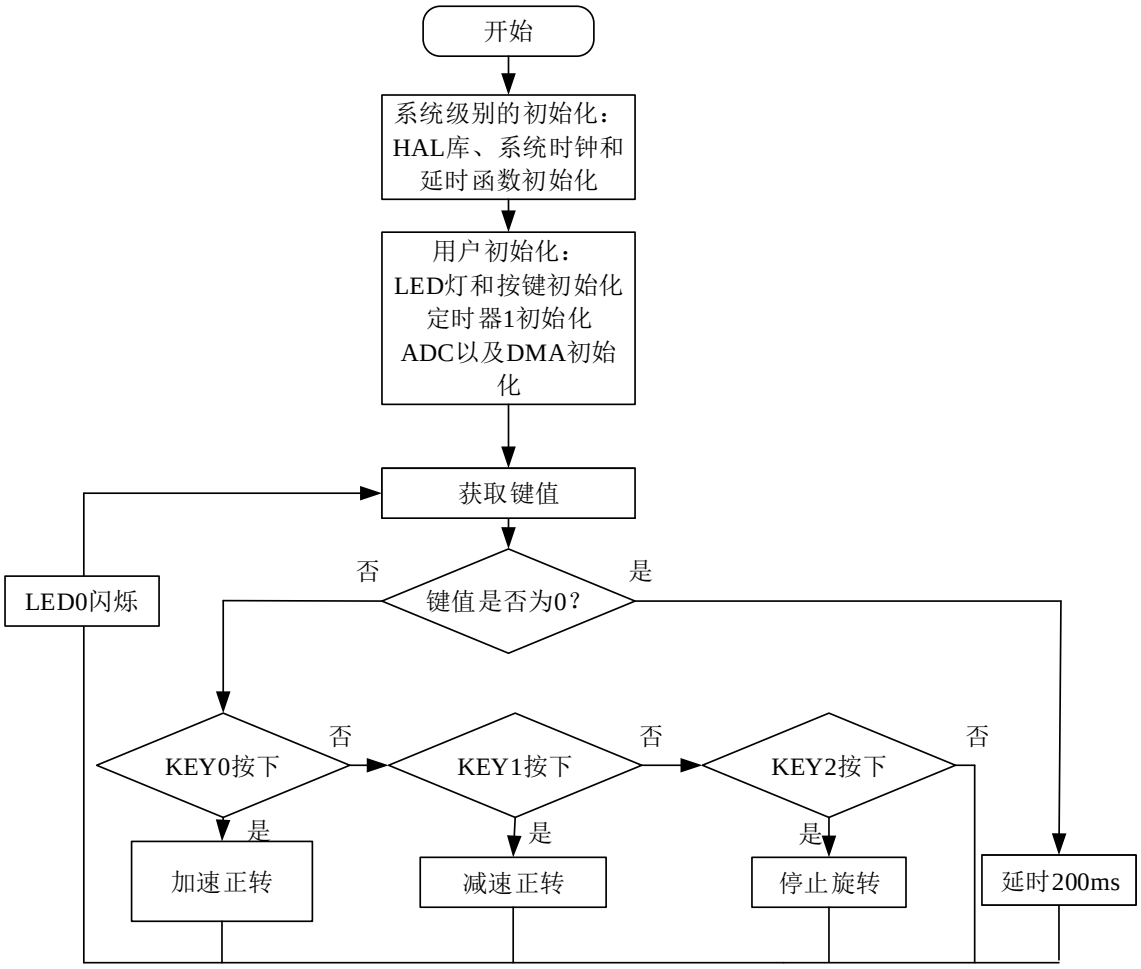


图 23.3.2.1 无刷电机基本控制流程图

23.3.3 程序解析

该程序是在上一章的工程基础上添加的，大部分相同，这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。定时器驱动相关源码与上一章节一致，这里不再赘述。我们先看 ADC 相关的程序，ADC 驱动源码包括两个文件：adc.c 和 adc.h。

首先看 adc.h 头文件的内容：

```
/* ADC 及引脚 定义 */

#define ADC_ADCX_CH0_GPIO_PORT                  GPIOB
#define ADC_ADCX_CH0_GPIO_PIN                   GPIO_PIN_1
#define                                         ADC_ADCX_CH0_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)     /* PB 口时钟使能 */

#define ADC_ADCX_CH1_GPIO_PORT                  GPIOA
#define ADC_ADCX_CH1_GPIO_PIN                   GPIO_PIN_0
#define                                         ADC_ADCX_CH1_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)     /* PA 口时钟使能 */

#define ADC_ADCX_CH2_GPIO_PORT                  GPIOB
#define ADC_ADCX_CH2_GPIO_PIN                   GPIO_PIN_0
#define                                         ADC_ADCX_CH2_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)     /* PB 口时钟使能 */
```

```
#define ADC_ADCX_CH3_GPIO_PORT                  GPIOA
#define ADC_ADCX_CH3_GPIO_PIN                   GPIO_PIN_6
#define                                         ADC_ADCX_CH3_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)       /* PA 口时钟使能 */

#define ADC_ADCX_CH4_GPIO_PORT              GPIOA
#define ADC_ADCX_CH4_GPIO_PIN               GPIO_PIN_3
#define                                     ADC_ADCX_CH4_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOA_CLK_ENABLE(); }while(0)       /* PA 口时钟使能 */

#define ADC_ADCX                            ADC1
#define ADC_ADCX_CH0                        ADC_CHANNEL_9
#define ADC_ADCX_CH1                        ADC_CHANNEL_0
#define ADC_ADCX_CH2                        ADC_CHANNEL_8
#define ADC_ADCX_CH3                        ADC_CHANNEL_6
#define ADC_ADCX_CH4                        ADC_CHANNEL_3

#define                                     ADC_ADCX_CHY_CLK_ENABLE()
do{ __HAL_RCC_ADC1_CLK_ENABLE(); }while(0)        /* ADC1 时钟使能 */

#define ADC_CH_NUM                          5       /* 需要转换的通道数目 */
#define ADC_COLL                            50      /* 单采集次数 */
#define ADC_SUM                             ADC_CH_NUM  *   ADC_COLL
/* 总采集次数 */

/* ADC 单通道/多通道 DMA 采集 DMA 数据流相关 定义
 * 注意：这里我们的通道还是使用上面的定义.
 */
#define ADC_ADCX_DMASx                      DMA2_Stream4
#define ADC_ADCX_DMASx_Chanel               DMA_CHANNEL_0 /* ADC1_DMA 请求源 */
#define ADC_ADCX_DMASx_IRQn                 DMA2_Stream4_IRQn
#define ADC_ADCX_DMASx_IRQHandler           DMA2_Stream4_IRQHandler

#define                                     ADC_ADCX_DMASx_IS_TC()
( __HAL_DMA_GET_FLAG(&g_dma_nch_adc_handle, DMA_FLAG_TCIF0_4) )
/* 判断 DMA2 Stream4 传输完成标志，这是一个假函数形式，

* 不能当函数使用，只能用在 if 等语句里面

*/
#define                                             ADC_ADCX_DMASx_CLR_TC()
do{ __HAL_DMA_CLEAR_FLAG(&g_dma_nch_adc_handle, DMA_FLAG_TCIF0_4); }while(0)
/* 清除 DMA2 Stream4 传输完成标志 */
```

这里主要分为三部分，第一部分 ADC 通道所对应的 IO 以及时钟使能的宏定义；第二部分为本例程所使用的 ADC 通道宏定义，第三部分为 ADC 相关的 DMA 宏定义以及 ADC 的转换次数的宏定义。接着看下 ADC 的初始化程序，如下：

```
void adc_init(void)
{
    ADC_ChannelConfTypeDef sConfig = {0};

    g_adc_nch_dma_handle.Instance = ADC_ADCX;
    g_adc_nch_dma_handle.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
    /* 4 分频，ADCCLK = PCLK2/4 = 84/4 = 21Mhz */
    g_adc_nch_dma_handle.Init.Resolution = ADC_RESOLUTION_12B;
    /* 12 位模式 */
    g_adc_nch_dma_handle.Init.ScanConvMode = ENABLE;
    /* 扫描模式 多通道使用 */
    g_adc_nch_dma_handle.Init.ContinuousConvMode = ENABLE;
    /* 连续转换模式，转换完成之后接着继续转换 */
    g_adc_nch_dma_handle.Init.DiscontinuousConvMode = DISABLE;
    /* 禁止不连续采样模式 */
    g_adc_nch_dma_handle.Init.ExternalTrigConvEdge =
    ADC_EXTERNALTRIGCONVEDGE_NONE; /* 使用软件触发 */
```

```
    g_adc_nch_dma_handle.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    /* 软件触发 */
    g_adc_nch_dma_handle.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    /* 右对齐 */
    g_adc_nch_dma_handle.Init.NbrOfConversion = ADC_CH_NUM;
    /* 使用转换通道数，需根据实际转换通道去设置 */
    g_adc_nch_dma_handle.Init.DMAContinuousRequests = ENABLE;
    /* 开启 DMA 连续转换 */
    g_adc_nch_dma_handle.Init.EOCSelection = ADC_EOC_SEQ_CONV;
    HAL_ADC_Init(&g_adc_nch_dma_handle);

    /* 配置使用的 ADC 通道，采样序列里的第几个转换，增加或者减少通道需要修改这部分 */
    sConfig.Channel = ADC_ADCX_CH0;                         /* 电源电压采集 */
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);

    sConfig.Channel = ADC_ADCX_CH1;                         /* 温度采集 */
    sConfig.Rank = 2;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);

    sConfig.Channel = ADC_ADCX_CH2;                         /* U 相电压采集 */
    sConfig.Rank = 3;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);

    sConfig.Channel = ADC_ADCX_CH3;                         /* V 相电压采集 */
    sConfig.Rank = 4;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);

    sConfig.Channel = ADC_ADCX_CH4;                         /* W 相电压采集 */
    sConfig.Rank = 5;
    HAL_ADC_ConfigChannel(&g_adc_nch_dma_handle, &sConfig);
}
```

该函数将 ADC 设置为软件触发连续转换，并配置了 ADC 的转换顺序，1~5 分别对应：电源电压、温度、U 相电压、V 相电压、W 相电压，而且使用 DMA 采集，DMA 初始化程序，内容如下。

```
/**
 * @brief       ADC DMA 读取 初始化函数
 * @note      本函数还是使用 adc_init 对 ADC 进行大部分配置,有差异的地方再单独配置
 * @param       par         : 外设地址
 * @param       mar         : 存储器地址
 * @retval      无
 */
void adc_nch_dma_init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    ADC_ADCX_CHY_CLK_ENABLE();              /* 使能 ADCx 时钟 */
    ADC_ADCX_CH0_GPIO_CLK_ENABLE();         /* 开启 GPIO 时钟 */
    ADC_ADCX_CH1_GPIO_CLK_ENABLE();
    ADC_ADCX_CH2_GPIO_CLK_ENABLE();
    ADC_ADCX_CH3_GPIO_CLK_ENABLE();
    ADC_ADCX_CH4_GPIO_CLK_ENABLE();

    /* AD 采集引脚模式设置,模拟输入 */
    GPIO_InitStruct.Pin = ADC_ADCX_CH0_GPIO_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(ADC_ADCX_CH0_GPIO_PORT, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = ADC_ADCX_CH1_GPIO_PIN;
    HAL_GPIO_Init(ADC_ADCX_CH1_GPIO_PORT, &GPIO_InitStruct);
```

```
    GPIO_InitStruct.Pin = ADC_ADCX_CH2_GPIO_PIN;
    HAL_GPIO_Init(ADC_ADCX_CH2_GPIO_PORT, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = ADC_ADCX_CH3_GPIO_PIN;
    HAL_GPIO_Init(ADC_ADCX_CH3_GPIO_PORT, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = ADC_ADCX_CH4_GPIO_PIN;
    HAL_GPIO_Init(ADC_ADCX_CH4_GPIO_PORT, &GPIO_InitStruct);

    adc_init();

    if ((uint32_t)ADC_ADCX_DMASx > (uint32_t)DMA2)
    /* 大于 DMA1_Channel7，则为 DMA2 的通道了 */
    {
        __HAL_RCC_DMA2_CLK_ENABLE();     /* DMA2 时钟使能 */
    }
    else
    {
        __HAL_RCC_DMA1_CLK_ENABLE();     /* DMA1 时钟使能 */
    }

    /* DMA 配置 */
    g_dma_nch_adc_handle.Instance = ADC_ADCX_DMASx;                 /* 设置 DMA 通道 */
    g_dma_nch_adc_handle.Init.Channel = DMA_CHANNEL_0;
    g_dma_nch_adc_handle.Init.Direction = DMA_PERIPH_TO_MEMORY;
    /* DIR = 1 ，  外设到存储器模式 */
    g_dma_nch_adc_handle.Init.PeriphInc = DMA_PINC_DISABLE;     /*外设非增量模式*/
    g_dma_nch_adc_handle.Init.MemInc = DMA_MINC_ENABLE;         /*存储器增量模式*/
    g_dma_nch_adc_handle.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
    /* 外设数据长度:16 位 */
    g_dma_nch_adc_handle.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
    /* 存储器数据长度:16 位 */
    g_dma_nch_adc_handle.Init.Mode = DMA_CIRCULAR;                  /* 外设流控模式 */
    g_dma_nch_adc_handle.Init.Priority = DMA_PRIORITY_MEDIUM;   /* 中等优先级 */
    HAL_DMA_Init(&g_dma_nch_adc_handle);

    __HAL_LINKDMA(&g_adc_nch_dma_handle,DMA_Handle,g_dma_nch_adc_handle);

    HAL_NVIC_SetPriority(ADC_ADCX_DMASx_IRQn, 2, 1);
    HAL_NVIC_EnableIRQ(ADC_ADCX_DMASx_IRQn);

    HAL_ADC_Start_DMA(&g_adc_nch_dma_handle,(uint32_t *)g_adc_value,
                      ADC_CH_NUM * ADC_COLL);
}
```

首先初始化 ADC 通道管脚，并初始化 DMA。DMA 存储的地址就是数组 g_adc_value，传输的数据量大小为：ADC_CH_NUM * ADC_COLL ，如果对于 ADC 与 DMA 的配置部分还不熟悉，大家可以回顾下 4.4 小节。

接着来到 ADC 转换完成中断回调函数用于计算 ADC 平均值，程序如下。

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    if (hadc->Instance == ADC1)                 /* 大约 2.6ms 采集完成进入中断 */
    {
        HAL_ADC_Stop_DMA(&g_adc_nch_dma_handle);/* 关闭 DMA 转换 */
        calc_adc_val(g_adc_val);                /* ADC 数值转换 */
        HAL_ADC_Start_DMA(&g_adc_nch_dma_handle, (uint32_t *)&g_adc_value,
                          (uint32_t)(ADC_SUM));  /* 再启动 DMA 转换 */
    }
}
```

在 ADC 中断回调函数中，先关闭 DMA 转换，然后调用 calc_adc_val 函数计算 ADC 平均值（滤波），calc_adc_val 函数我们后面再介绍。滤波后的 ADC 值存储到 g_adc_val 这个数组当中，我们将通过滤波后的 ADC 值来进行电压、温度以及电流的计算。计算完 ADC 平均值

后，再次启动 DMA 转换。

接下来看电压、电流以及温度计算相关的程序，包括两个文件：bldc.c 和 bldc.h，我们这里只介绍本实验新增的内容。

首先看 bldc.h 头文件的几个宏定义：

```
/* ADC 采集值 * 3.3/4.096 （mv） = 6 * ( 0.02*I ) */
#define ADC2CURT    (float)(3.3f / 4.096f / 0.12f)

/* ADC 采集值 * 3.3/4096 （V）= POWER/(12K+12K+1K)*1K */
#define ADC2VBUS    (float)(3.3f * 25 / 4096)
```

这里定义的是电流、电压计算相关的宏，其中涉及的公式是根据我们前面介绍的电流、电压采集电路原理得出的，当我们需要计算电流和电压的时候，直接调用相应的宏定义即可，具体的公式推导过程大家可以直接看代码的注释。

下面看 bldc.c 的程序，首先看 ADC 平均值计算函数。

平均值计算函数 calc_adc_val 在 bldc.c 原文件中实现，内容如下。

```
/**
 * @brief       计算 ADC 的平均值（滤波）
 * @param       * p ：存放 ADC 值的指针地址
 * @note        此函数对电压、温度、电流对应的 ADC 值进行滤波
 * @retval      无
 */
void calc_adc_val(uint16_t * p)
{
    uint32_t temp[ADC_CH_NUM] = {0,0,0};    /* 定义一个缓存数组 */
    int i,j;
    for(i = 0; i < ADC_COLL; i++)           /* 循环采集 ADC_COLL 次数 */
    {
        for(j = 0; j < ADC_CH_NUM; j++)     /* 根据 ADC 通道数循环获取，并累加 */
        {
            temp[j] += g_adc_value[j+i*ADC_CH_NUM];  /* 各通道进行累加 */
        }
    }
    for(j = 0; j < ADC_CH_NUM; j++)
    {
        temp[j] /= ADC_COLL;                        /* 获取平均值 */
        p[j] = temp[j];                             /* 存到*p */
    }
}
```

该函数将 5 个通道采集的数据，进行软件滤波取平均并将滤波后的数据保留在入口参数指针 p 的地址中。

接下来看温度值计算函数，其定义如下：

```
/*
    Rt = Rp *exp(B*(1/T1-1/T2))

    Rt 是热敏电阻在 T1 温度下的阻值；
    Rp 是热敏电阻在 T2 常温下的标称阻值；
    exp 是 e 的 n 次方，e 是自然常数，就是自然对数的底数，近似等于 2.7182818；
    B 值是热敏电阻的重要参数，教程中用到的热敏电阻 B 值为 3380；
    这里 T1 和 T2 指的是开尔文温度，T2 是常温 25℃，即(273.15+25)K
    T1 就是所求的温度
*/
const float Rp = 10000.0f;          /* 10K */
const float T2 = (273.15f + 25.0f); /* T2 */
const float Bx = 3380.0f;           /* B */
const float Ka = 273.15f;

/**
 * @brief       计算温度值
 * @param       para: 温度采集对应 ADC 通道的值（已滤波）
 * @note        计算温度分为两步：
```

```
                1.根据 ADC 采集到的值计算当前对应的 Rt
                2.根据 Rt 计算对应的温度值
 * @retval      温度值
 */
float get_temp(uint16_t para)
{
    float Rt;
    float temp;

    /*
        第一步:
        Rt = 3.3 * 4700 / VTEMP - 4700 ,其中 VTEMP 就是温度检测通道采集回来的电压
        值,VTEMP = ADC 值* 3.3/4096
        由此我们可以计算出当前 Rt 的值: Rt = 3.3f * 4700.0f / (para * 3.3f /
        4096.0f ) - 4700.0f;
    */
    Rt = 3.3f * 4700.0f / (para * 3.3f / 4096.0f ) - 4700.0f;

    /*
        第二步:
        根据当前 Rt 的值来计算对应温度值: Rt = Rp *exp(B*(1/T1-1/T2))
    */

    temp = Rt / Rp;  /* 解出 exp(B*(1/T1-1/T2)) , 即 temp = exp(B*(1/T1-1/T2)) */
    temp = log(temp);        /* 解出 B*(1/T1-1/T2) , 即 temp = B*(1/T1-1/T2) */
    temp /= Bx;              /* 解出 1/T1-1/T2 , 即 temp = 1/T1-1/T2 */
    temp += (1.0f / T2);     /* 解出 1/T1 , 即 temp = 1/T1 */
    temp = 1.0f / (temp);    /* 解出 T1 , 即 temp = T1 */
    temp -= Ka;              /* 计算 T1 对应的摄氏度 */
    return temp;             /* 返回温度值 */
}
```

代码中首先定义了一些关于热敏电阻的参数，各参数的含义我们在代码注释中已经详细介绍，它们的值是根据实际的热敏电阻型号来确定的，详见《NCP18XH103F03RB 热敏电阻.pdf》这个数据手册，路径：A 盘→7，硬件资料→2，芯片资料 文件夹下可以找到。

温度计算函数的实现分为两个步骤：

第一步：根据采集到的 ADC 值计算出当前热敏电阻阻值 Rt，公式如下：

$$Rt = 3.3 * 4700 / VTEMP - 4700$$

公式中 VTEMP 是温度检测通道对应的电压值，结合 ADC 换算公式 VTEMP = ADC 值* 3.3/4096 可得 Rt= 3.3 * 4700/ (ADC 值 * 3.3 / 4096) - 4700。

第二步：根据当前 Rt 的值来计算对应温度值，公式如下：

$$Rt = Rp * exp(B * (1/T1 - 1/T2))$$

公式中的 T1 就是所求的温度，它的默认单位是华氏度，我们最后还需要将其转换成摄氏度。温度计算函数的实现本质上就是这两个公式的代码体现，大家一定要跟着公式去理解代码。

接着是定时器的中断回调函数，如下：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    uint8_t bldc_dir=0;
    uint8_t i;
    static uint8_t times_count=0;        /* 定时器时间记录 */
    int16_t temp_speed=0;                /* 临时速度存储 */
    if(htim->Instance == ATIM_TIMX_PWM) /* 55us */
    {
#ifdef H_PWM_L_ON
        if(g_bldc_motor1.run_flag == RUN)
        {
            g_bldc_motor1.count_j++;
            if(g_bldc_motor1.dir == CW)   /* 顺时针旋转 */
            {
```

```
            g_bldc_motor1.step_sta = hallsensor_get_state(MOTOR_1);
        }
        else
        {
            g_bldc_motor1.step_sta = 7 - hallsensor_get_state(MOTOR_1);
        }
        if((g_bldc_motor1.step_sta <= 6)&&(g_bldc_motor1.step_sta >= 1))
        {
            pfunclist_m1[g_bldc_motor1.step_sta-1]();
        }
        else    /* 编码器错误、接触不良、断开等情况 */
        {
            stop_motor1();
            g_bldc_motor1.run_flag = STOP;
        }
        g_bldc_motor1.hall_sta_edge =
        uemf_edge(g_bldc_motor1.hall_single_sta);/* 检测单个霍尔信号的变化 */
        if(g_bldc_motor1.hall_sta_edge == 0) /* 统计单个霍尔信号的高电平时间 */
        {
            /*计算速度*/
            if(g_bldc_motor1.dir == CW)
                temp_speed = (SPEED_COEFF/g_bldc_motor1.count_j);
            else
                temp_speed = -(SPEED_COEFF/g_bldc_motor1.count_j);
            FirstOrderRC_LPF(g_bldc_motor1.speed, temp_speed, 0.2379);
            g_bldc_motor1.no_single = 0;
            g_bldc_motor1.count_j = 0;
        }
        if(g_bldc_motor1.hall_sta_edge== 1)
        {
            g_bldc_motor1.no_single = 0;
            g_bldc_motor1.count_j = 0;
        }
        if(g_bldc_motor1.hall_sta_edge== 2)
        {
            g_bldc_motor1.no_single++;     /* 不换相时间累计 超时则判定速度为 0 */

            if(g_bldc_motor1.no_single>15000)
            {

                g_bldc_motor1.no_single = 0;
                g_bldc_motor1.speed = 0;   /* 超时换向 判定为停止 速度为 0*/
            }
        }
        if(g_bldc_motor1.step_last != g_bldc_motor1.step_sta)
        {
            g_bldc_motor1.hall_keep_t = 0;
            bldc_dir = check_hall_dir(&g_bldc_motor1);
            if(bldc_dir == CCW)
            {
                g_bldc_motor1.pos -= 1;
            }
            else if(bldc_dir == CW)
            {
                g_bldc_motor1.pos += 1;
            }
            g_bldc_motor1.step_last = g_bldc_motor1.step_sta;
        }
        else if(g_bldc_motor1.run_flag == RUN)      /* 运行且霍尔保持时 */
        {
            g_bldc_motor1.hall_keep_t++;
        }

        for(i=0; i<3; i++)
        {
```

```
            adc_val_m1[i] = g_adc_val[i+2];
            adc_amp[i] = adc_val_m1[i] -
            adc_amp_offset[i][ADC_AMP_OFFSET_TIMES];
            if(adc_amp[i]>=0)                /* 去除反电动势引起的负电流数据 */
                adc_amp_un[i]=adc_amp[i];
        }
        /*运算母线电流（母线电流为任意两个有开关动作的相电流之和）*/
        if(g_bldc_motor1.step_sta == 0x05)
        {
            adc_amp_bus= (adc_amp_un[0]+ adc_amp_un[1])*ADC2CURT;   /* UV */
        }
        else if(g_bldc_motor1.step_sta== 0x01)
        {
            adc_amp_bus= (adc_amp_un[0]+ adc_amp_un[2])*ADC2CURT;   /* UW */
        }
        else if(g_bldc_motor1.step_sta== 0x03)
        {
            adc_amp_bus= (adc_amp_un[1]+ adc_amp_un[2])*ADC2CURT;   /* VW */
        }
        else if(g_bldc_motor1.step_sta== 0x02)
        {
            adc_amp_bus= (adc_amp_un[0]+ adc_amp_un[1])*ADC2CURT;   /* UV */
        }
        else if(g_bldc_motor1.step_sta == 0x06)
        {
            adc_amp_bus= (adc_amp_un[0]+ adc_amp_un[2])*ADC2CURT;   /* WU */
        }
        else if(g_bldc_motor1.step_sta == 0x04)
        {
            adc_amp_bus= (adc_amp_un[2]+ adc_amp_un[1])*ADC2CURT;   /* WV */
        }
    }
#endif
    }
    else if(htim->Instance == TIM6)
    {
        /* 计算未开始启动时的基准电压 */
        times_count++;
        if(g_bldc_motor1.run_flag == STOP)
        {
            uint8_t i;
            uint32_t avg[3] = {0,0,0};
            /* 获取电机停机状态下的三相电流 */
            adc_amp_offset[0][adc_amp_offset_p] = g_adc_val[2]; /* U */
            adc_amp_offset[1][adc_amp_offset_p] = g_adc_val[3]; /* V */
            adc_amp_offset[2][adc_amp_offset_p] = g_adc_val[4]; /* W */
            adc_amp_offset_p ++;
            /* 如果溢出，从头开始计数 */
            NUM_CLEAR(adc_amp_offset_p,ADC_AMP_OFFSET_TIMES);
            for(i = 0; i < ADC_AMP_OFFSET_TIMES; i++)
            {
                avg[0] += adc_amp_offset[0][i];                 /* 各相数值累加 */
                avg[1] += adc_amp_offset[1][i];
                avg[2] += adc_amp_offset[2][i];
            }
            for(i = 0; i < 3; i++)
            {
                avg[i] /= ADC_AMP_OFFSET_TIMES;                 /* 取平均 */
                adc_amp_offset[i][ADC_AMP_OFFSET_TIMES] = avg[i];/* 赋值 */
            }
        }
    }
}
```

关于 BLDC 的基础驱动部分与上一章一致，不在赘述；这里主要讲下母线电流的采集。

注意：在电流采集电路的原理介绍中，我们有提到抬升电压的值为 1.25V。在实际的应用

中，为了使电流采集的值更加精确，我们不是直接减掉该抬升电压，因为驱动板一上电，本身是带有基准电压的，所以会造成电压不准确，所以需在电机还未开始旋转时检测当前的电压值，即在定时器 6 更新中断里所操作的内容：首先判断电机是否停止，是的话将此时的三相电流所对应的 ADC 通道采集的 ADC 值进行累加取平均，即软件滤波，此时滤波后的值才是电机未启动时的基准电压；

接着当电机旋转时采集此时的三相电压，实际的相电压 = 旋转时电压 – 基准电压；通过该算法求得的即为相电压。母线电压为任意导通的两相电压之和，在乘以前面原理介绍的系数即可换算出电流值。

最后是 main 函数。

```
int main(void)
{
    uint8_t key,t;
    char buf[32];
    int16_t pwm_duty_temp=0;
    float current[3]= {0.0f};
    float current_lpf[4]= {0.0f};

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    bldc_init(10000-1,0);
    bldc_ctrl(MOTOR_1,CCW,0);                   /* 初始无刷电机接口 1 速度 */

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10, 10, 200, 16, 16, "BLDC Motor Test",g_point_color);
    lcd_show_string(10, 30, 200, 16, 16, "KEY0:Start forward", g_point_color);
    lcd_show_string(10, 50, 200, 16, 16, "KEY1:Start backward", g_point_color);
    lcd_show_string(10, 70, 200, 16, 16, "KEY2:Stop", g_point_color);
    adc_nch_dma_init();

    while (1)
    {
        t++;
        if(t % 20 == 0)
        {
            sprintf(buf,"PWM_Duty:%.1f%%",(float)((g_bldc_motor1.pwm_duty
            /MAX_PWM_DUTY)*100));                   /* 显示控制 PWM 占空比 */
            lcd_show_string(10,110,200,16,16,buf,g_point_color);

            sprintf(buf,"Power:%.3fV ",g_adc_val[0]*ADC2VBUS);
            lcd_show_string(10,190,200,16,16,buf,g_point_color);
            sprintf(buf,"Temp:%.1fC ",get_temp(g_adc_val[1]));
            lcd_show_string(10,210,200,16,16,buf,g_point_color);
            LED0_TOGGLE();                          /* LED0(红灯) 翻转 */


            current[0] = adc_amp_un[0]* ADC2CURT;       /* 计算出三相电流值，U */
            current[1] = adc_amp_un[1]* ADC2CURT;       /* 计算出三相电流值，V */
            current[2] = adc_amp_un[2]* ADC2CURT;       /* 计算出三相电流值，W */

            /*一阶数字滤波 滤波系数 0.1 用于显示*/
            FirstOrderRC_LPF(current_lpf[0],current[0],0.1f);   /* U 相电流 */
            FirstOrderRC_LPF(current_lpf[1],current[1],0.1f);   /* V 相电流 */
            FirstOrderRC_LPF(current_lpf[2],current[2],0.1f);   /* W 相电流 */
            FirstOrderRC_LPF(current_lpf[3],adc_amp_bus,0.1f);  /* 母线电流 */
```

```
        if(g_bldc_motor1.run_flag == STOP)                /* 停机的电流显示 */
        {
            current_lpf[0] = 0;
            current_lpf[1] = 0;
            current_lpf[2] = 0;
            current_lpf[3] = 0;
        }
        /* LCD 显示提示信息 */
        sprintf(buf,"Amp U:%.3fmA ",(float)current_lpf[0]);
        lcd_show_string(10,230,200,16,16,buf,g_point_color);
        sprintf(buf,"Amp V:%.3fmA ",(float)current_lpf[1]);
        lcd_show_string(10,250,200,16,16,buf,g_point_color);
        sprintf(buf,"Amp W:%.3fmA ",(float)current_lpf[2]);
        lcd_show_string(10,270,200,16,16,buf,g_point_color);
        sprintf(buf,"Amp Bus:%.3fmA ",(float)current_lpf[3]);
        lcd_show_string(10,290,200,16,16,buf,g_point_color);

        /* 串口打印信息 */
        printf("Valtage:%.1fV \r\n", g_adc_val[0]*ADC2VBUS);
        printf("Temp:%.1fC \r\n", get_temp(g_adc_val[1]));
        printf("U 相电流为: %.3fmA\r\n", (current_lpf[0]));
        printf("V 相电流为: %.3fmA\r\n", (current_lpf[1]));
        printf("W 相电流为: %.3fmA\r\n", (current_lpf[2]));
        printf("母线电流为: %.3fmA\r\n", (current_lpf[3]));
        printf("\r\n");
    }
    key = key_scan(0);
    if(key == KEY0_PRES)
    {
        pwm_duty_temp+=500;
        if(pwm_duty_temp>=MAX_PWM_DUTY/2)
            pwm_duty_temp=MAX_PWM_DUTY/2;
        if(pwm_duty_temp>0)
        {
            g_bldc_motor1.pwm_duty= pwm_duty_temp;
            g_bldc_motor1.dir=CCW;
        }
        else
        {
            g_bldc_motor1.pwm_duty=-pwm_duty_temp;
            g_bldc_motor1.dir=CW;
        }
        g_bldc_motor1.run_flag=RUN;  /* 开启运行 */
        start_motor1();              /* 开启运行 */
    }
    if(key == KEY1_PRES)             /* 按下 KEY1 开启电机 */
    {
        pwm_duty_temp-=500;
        if(pwm_duty_temp<=-MAX_PWM_DUTY/2)
            pwm_duty_temp=-MAX_PWM_DUTY/2;
        if(pwm_duty_temp < 0)
        {
            g_bldc_motor1.pwm_duty=-pwm_duty_temp;
            g_bldc_motor1.dir=CW;
        }
        else
        {
            g_bldc_motor1.pwm_duty=pwm_duty_temp;
            g_bldc_motor1.dir=CCW;
        }
        g_bldc_motor1.run_flag=RUN;  /* 开启运行 */
        start_motor1();              /* 运行电机 */
    }
    if(key == KEY2_PRES)             /* 按下 KEY2 关闭电机 */
```

```
        {
            stop_motor1();                  /* 停机 */
            g_bldc_motor1.run_flag=STOP; /* 标记停机 */
            pwm_duty_temp=0;
            g_bldc_motor1.pwm_duty=0;
        }
        delay_ms(10);
    }
}
```

主函数主要添加了 adc 初始化函数 adc_nch_dma_init，对 ADC 以及 DMA 进行初始化。

在 while 循环里面，除了有感方波驱动的内容，我们还利用 LDC 以及串口 1 来打印电压、温度以及电流的值，计算过程以及在原理部分讲的很详细了，大家可以原理分析部分。

23.4 下载验证

硬件连接好之后下载程序到开发板，液晶屏和串口都会显示程序计算的电压、电流等信息，点击按键 KEY0 一次可以看到定时器比较值递增 500（加速），电机开始旋转，点击一次按键 KEY1 可以看到定时器比较值递减 500（减速），点击按键 KEY2 可以随时停止无刷电机旋转，并在 LCD 以及串口显示 ADC 所采集的电源电压、驱动板温度、以及三相电流。大家可以自行下载程序并点击按键测试效果。

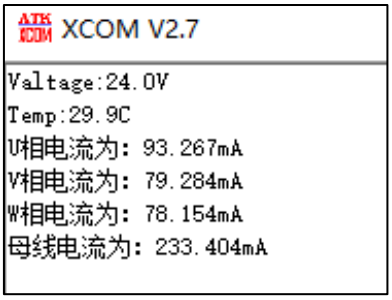


图 23.4.1 模拟信号数据

第 24 章 无刷电机有感方波闭环控制

前面已经实现了无刷电机的基本控制和电机的模拟信号采集，本章将利用 PID 来实现**速度环闭环控制**以及**速度+电流双闭环控制**。

本章分为如下几个小节：

24.1 速度闭环控制

24.2 速度&电流双闭环控制

24.1 速度闭环控制

要实现速度闭环，首先需要先了解下无刷有感驱动的测速原理，这样才可以知道当前转速是否符合我们的设定值。

24.1.1 无刷有感测速原理

从前面的学习得知，当转子磁极只有一对级时，转子旋转一圈三个霍尔信号都会输出相应波形，这些波形的特点：高低电平的持续时间是一样的，均为 180°电角度（对于店铺 BLDC 的霍尔传感器来说，N 级靠近输出：1，S 级靠近输出：0）。如下：

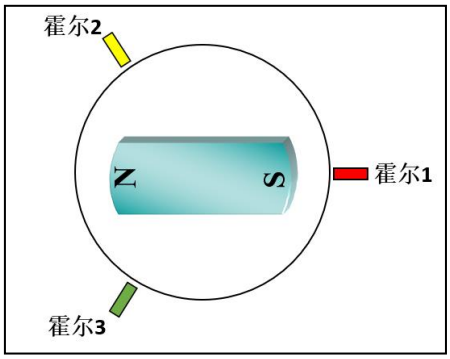


图 24.1.1.1 无刷电机与霍尔位置示意图

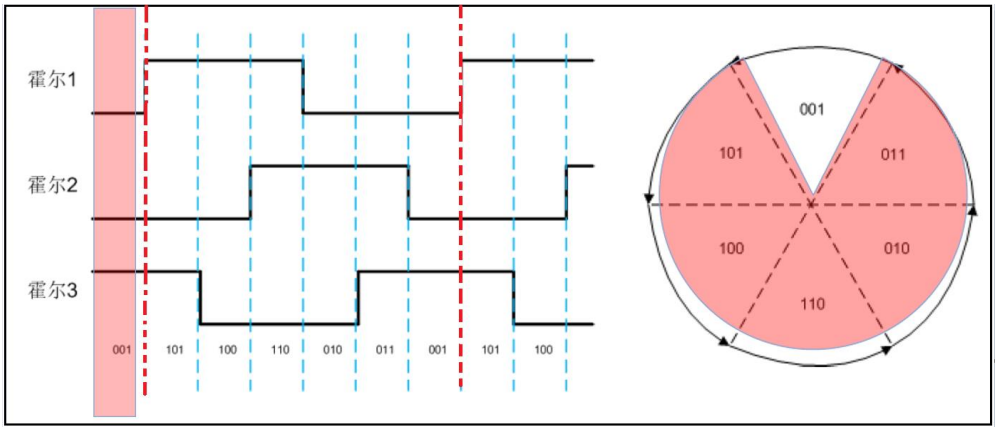


图 24.1.1.2 霍尔输出波形示意图

第一步：当只有一对级时，转子旋转一圈，霍尔输出一个完整脉冲（其中高电平与低电平持续时间均为 180°电角度）；

第二步：计算其中高电平的持续时间，即：t = C / Ft （其中 t 为 180°电角度所代表的时间，Ft 是霍尔脉冲的频率，C 为计数次数）

第三步：所以旋转一圈，需要的总时间为 T = 2*C/Ft；

第四步：所得出的结果单位为：s/圈 ，倒数即为：圈/s ，需将其单位转化为 RPM 即：Ft/（2*C） *60 。

第五步：当转子为 2 对级时，霍尔输出的高低电平时间均为 360°电角度，所以速度公式为：Ft/(4*C)*60。

24.1.2 硬件设计

1、例程功能

1、本实验以电机开发板的直流有刷/无刷电机驱动接口 1 为例，基于无刷电机参数采集实验，加入速度环 PID 控制算法，对电机的速度进行闭环控制。

2、当按键 0 按下，就增大目标速度值；当按键 1 按下，就减小目标速度值。目标速度的绝对值大小决定电机的速度，它的正负决定电机的正反转方向。按下按键 2 则马上停止电机。

3、屏幕显示按键功能、占空比、目标转速以及实际转速以及显示无刷电机电压、电流和温度、速度等等信息。

4、可通过串口 1 即 USB_TTL 接口连接正点原子 PID 调试助手，查看 PID 波形。

5、LED0 闪烁指示程序运行。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器：
TIM1_CH1：PA8
TIM1_CH2：PA9
TIM1_CH3：PA10
IO：PB13\14\15
使能引脚：SHDN: PF10
4）无刷电机
5）无刷电机驱动板
6）12-60V 的 DC 电源
7）ADC：ADC1_CH9：PB1
ADC1_CH0：PA0
ADC1_CH8：PB0
ADC1_CH6：PA6
ADC1_CH3：PA3

3、原理图

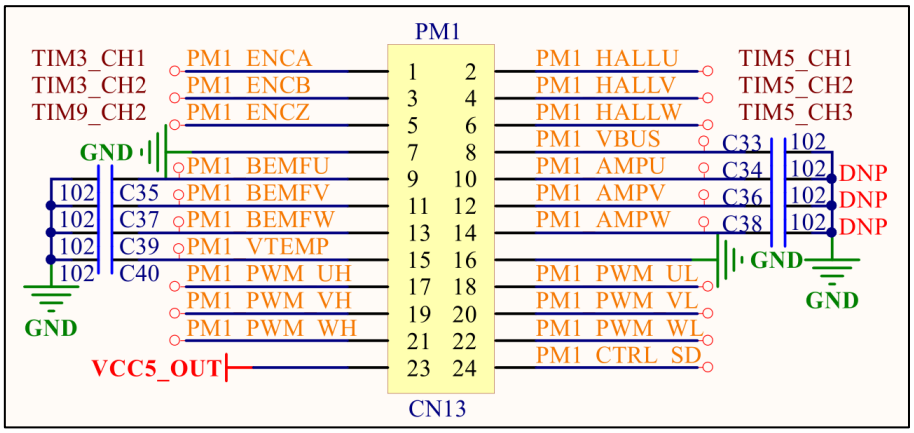


图 24.1.2.1 无刷电机接口 1 原理图

接口涉及的 IO 如下。

直流无刷驱动板	IO 口
驱动板 U 相	PA8、PB13
驱动板 V 相	PA9、PB14
驱动板 W 相	PA10、PB15
霍尔 U 相	PH10
霍尔 V 相	PH11
霍尔 W 相	PH12
输出使能	PF10
电压	PB1
温度	PA0
电流 U 相	PB0
电流 V 相	PA6
电流 W 相	PA3

表 24.1.2.1 无刷相关 IO 口说明

驱动板和电机的连接方式之前已经介绍过了，驱动板和开发板连接只需要使用排线连接即可(注意如使用 PID 调试助手查看波形，需将开发板串口 1 的 USB_TTL 连接至电脑)，实物连接如下。

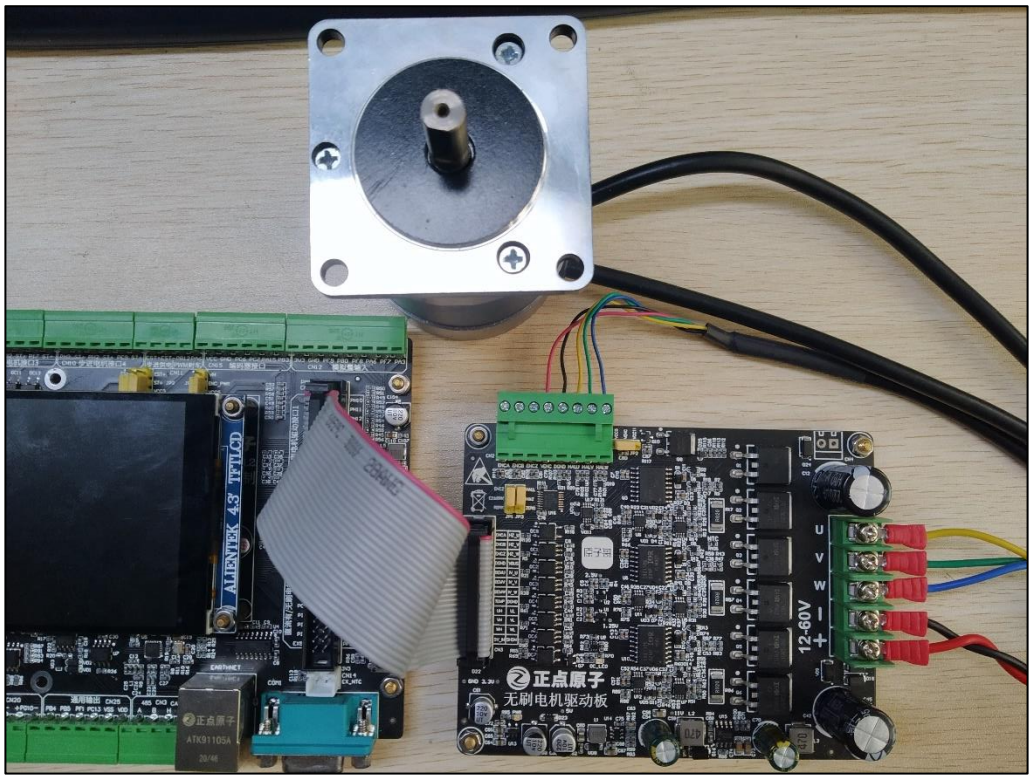


图 24.1.2.2 开发板&驱动板连接图

24.1.3 程序设计

24.1.3.1 无刷电机速度闭环的配置步骤

1）实现电机基本驱动

实现电机的基础驱动函数，启停、6 步换向组合等等

2）初始化 ADC（保留上节课的参数采集功能）

初始化 ADC 通道的 IO，设置 ADC 工作方式、DMA 等等

3）PID 闭环控制实现

实现 PID 的初始参数赋值、PID 计算等等

4）上位机通信协议

编写上位机通信代码，可在上位机上实时显示当前速度与目标速度的波形变化

5）编写中断服务函数

PWM 中断用于换向控制、堵转检测等等，添加 PID 的周期计算调用

24.1.3.2 程序流程图

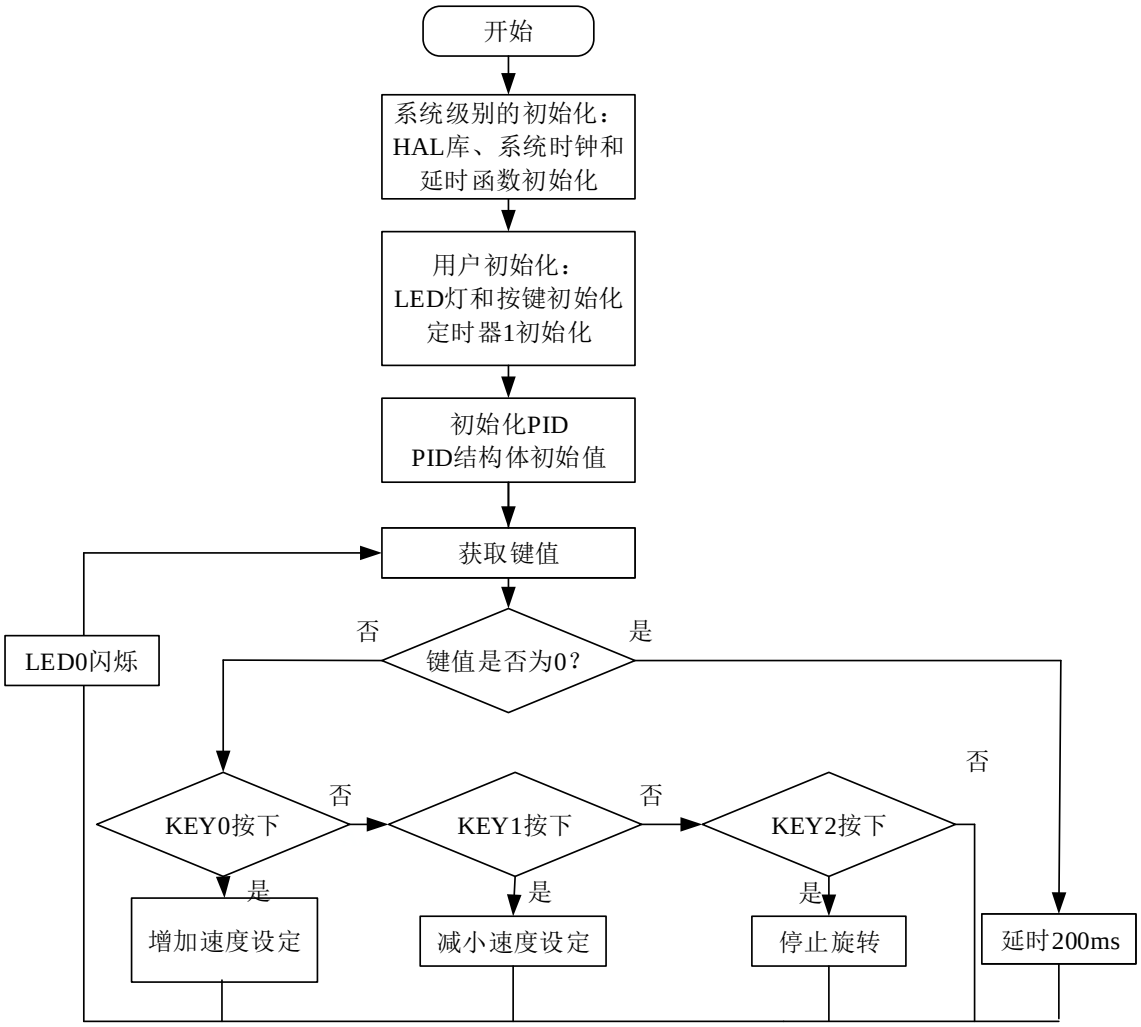


图 24.1.3.2.1 无刷电机速度闭环控制流程图

24.1.3.3 程序解析

这里我们只讲解核心代码，定时器及 ADC 的相关程序都和之前一致，这里就不重复列出了，首先需要关注的是 pid.h 头文件中 PID 结构体和默认的 P、I、D 参数的定义。如下所示。

```
/* PID 相关参数 */

#define  INCR_LOCT_SELECT  1          /*0 选择位置式 1：增量式控制*/
#if INCR_LOCT_SELECT
#define  S_KP      0.00800f          /* P 参数*/
#define  S_KI      0.00025f          /* I 参数*/
#define  S_KD      0.00020f          /* D 参数*/
#else

#define  S_KP      0.00800f          /* P 参数*/
#define  S_KI      0.00025f          /* I 参数*/
#define  S_KD      0.00020f          /* D 参数*/
#endif
#define SMAPLSE_PID_SPEED  40        /* 采样率 单位 ms*/

/*定义位置 PID 参数相关宏*/
/*PID 结构体*/
typedef struct
```

```
{
    __IO float  SetPoint;             /* 设定目标 */
    __IO float  ActualValue;          /* 实际值*/
    __IO float  SumError;             /* 误差累计*/
    __IO float  Proportion;           /* 比例常数 P*/
    __IO float  Integral;             /* 积分常数 I*/
    __IO float  Derivative;           /* 微分常数 D*/
    __IO float  Error;                /* Error[-1]*/
    __IO float  LastError;            /* Error[-1]*/
    __IO float  PrevError;            /* Error[-2]*/
    __IO float  IngMin;
    __IO float  IngMax;
    __IO float  OutMin;
    __IO float  OutMax;
} PID_TypeDef;
```

上述代码可以通过宏 INCR_LOCT_SELECT 选择使用哪种 PID 控制算法，当为 0 的时候选用位置式 PID 算法；为 1 时，选用增量式 PID 算法。然后接着是关于 PID 算法相关的结构体 PID_TypeDef，在后续的 PID 控制算法中使用到。

我们接着看下 pid.c 的内容：

```
PID_TypeDef  g_speed_pid;       /*位置 PID 参数结构体*/
/**
 * @brief       初始化 LED 相关 IO 口, 并使能时钟
 * @param       无
 * @retval      无
 */
void pid_init(void)
{
    g_speed_pid.SetPoint = 0;         /* 设定目标 Desired Value*/
    g_speed_pid.ActualValue = 0;      /* 设定目标 Desired Value*/
    g_speed_pid.SumError = 0;         /* 积分值*/
    g_speed_pid.Error = 0;            /* Error[1]*/
    g_speed_pid.LastError = 0;        /* Error[-1]*/
    g_speed_pid.PrevError = 0;        /* Error[-2]*/
    g_speed_pid.Proportion = KP;      /* 比例常数 Proportional Const*/
    g_speed_pid.Integral = KI;        /* 积分常数 Integral Const*/
    g_speed_pid.Derivative = KD;      /* 微分常数 Derivative Const*/
    g_speed_pid.IngMax = 4000;
    g_speed_pid.IngMin = -4000;
    g_speed_pid.OutMax = 4000;        /* 输出限制 */
    g_speed_pid.OutMin = -4000;
}

/**
  * 函数名称：位置闭环 PID 控制设计
  * 输入参数：当前控制量
  * 返 回 值：目标控制量
  * 说    明：无
  */
int32_t increment_pid_ctrl(PID_TypeDef *PID,float Feedback_value)
{
    PID->Error = (float)(PID->SetPoint - Feedback_value);   /* 速度档位偏差*/
#if  INCR_LOCT_SELECT
    PID->ActualValue += (PID->Proportion * (PID->Error - PID->LastError))
                                                            /* E[k]项*/
                      + (PID->Integral * PID->Error)        /* E[k-1]项*/
                      + (PID->Derivative * (PID->Error - 2 * PID->LastError +
PID->PrevError));                                           /* E[k-2]项*/
    PID->PrevError = PID->LastError;                        /* 存储误差，下次计算*/
    PID->LastError = PID->Error;
#else
    PID->SumError += PID->Error;
```

```
    PID->ActualValue = (PID->Proportion * PID->Error)       /* E[k]项*/
                     + (PID->Integral * PID->SumError)      /* E[k-1]项*/
                     + (PID->Derivative * (PID->Error - PID->LastError));
                                                            /* E[k-2]项*/
    PID->LastError = PID->Error;
#endif
    if(PID->ActualValue > PID->OutMax)
    {
        PID->ActualValue = PID->OutMax;
    }
    else if(PID->ActualValue < PID->OutMin)
    {
        PID->ActualValue = PID->OutMin;
    }
    return ((int32_t)(PID->ActualValue));                   /*返回实际控制数值*/
}
```

上述代码可以分为 PID 初始化及 PID 控制算法两部分。第一部分主要是将 PID 算法相关结构体变量进行初始赋值；第二部分是 PID 控制算法的计算过程，这部分与之前介绍的 PID 算法一致，这里不多赘述。

重点内容是在定时器中断服务函数里边定时调用 PID 控制以及速度的计算，这部分内容在 bldc_tim.c 中实现，程序如下：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    uint8_t i;
    uint8_t bldc_dir=0;
    static uint8_t times_count=0;       /* 定时器时间记录 */
    int16_t temp_speed=0;               /* 临时速度存储 */
    if(htim->Instance == ATIM_TIMX_PWM) /* 55us */
    {
        /******************* 六步换向 ********************/
        if(g_bldc_motor1.run_flag == RUN)
        {
            if(g_bldc_motor1.dir == CW)
            {
                g_bldc_motor1.step_sta = hallsensor_get_state(MOTOR_1);
            }
            else
            {
                g_bldc_motor1.step_sta = 7 - hallsensor_get_state(MOTOR_1);
            }
            if((g_bldc_motor1.step_sta <= 6)&&(g_bldc_motor1.step_sta >= 1))
            {
                pfunclist_m1[g_bldc_motor1.step_sta-1]();
            }
            else    /* 编码器错误、接触不良、断开等情况 */
            {
                stop_motor1();
                g_bldc_motor1.run_flag = STOP;
            }

            /****************** 速度计算 *****************/
            g_bldc_motor1.count_j++;                        /* 计算速度专用计数值 */
            g_bldc_motor1.hall_sta_edge =
              uemf_edge(g_bldc_motor1.hall_single_sta);  /* 检测单个霍尔信号的变化 */
            if(g_bldc_motor1.hall_sta_edge == 0)
            {
                /*计算速度*/
                if(g_bldc_motor1.dir == CW)
                    temp_speed = (SPEED_COEFF/g_bldc_motor1.count_j);
                else
                    temp_speed = -(SPEED_COEFF/g_bldc_motor1.count_j);
                FirstOrderRC_LPF(g_bldc_motor1.speed,temp_speed,0.2379f);
                g_bldc_motor1.no_single = 0;
```

```
            g_bldc_motor1.count_j = 0;
        }
        if(g_bldc_motor1.hall_sta_edge == 1)
        {
            g_bldc_motor1.no_single = 0;
            g_bldc_motor1.count_j = 0;
        }
        if(g_bldc_motor1.hall_sta_edge == 2)  /* 霍尔值一直不变代表未换向 */
        {
            g_bldc_motor1.no_single++;          /*不换相时间累计超时则判定速度为 0*/

            if(g_bldc_motor1.no_single > 15000)
            {

                g_bldc_motor1.no_single = 0;
                g_bldc_motor1.speed = 0;        /* 超时换向 判定为停止 速度为 0 */
            }
        }
        /****************** 位置记录 ******************/
        if(g_bldc_motor1.step_last != g_bldc_motor1.step_sta)
        {
            bldc_dir = check_hall_dir(&g_bldc_motor1);
            if(bldc_dir == CCW)
            {
                g_bldc_motor1.pos -=1;
            }
            else if(bldc_dir == CW)
            {
                g_bldc_motor1.pos +=1;
            }
            g_bldc_motor1.step_last = g_bldc_motor1.step_sta;
        }
        /****************** PID 控制 ******************/
            temp_pwm1 = increment_pid_ctrl(&g_speed_pid,g_bldc_motor1.speed);
            FirstOrderRC_LPF(motor_pwm_s,temp_pwm1,0.085);
            if(motor_pwm_s < 0)
            {
                g_bldc_motor1.pwm_duty = -motor_pwm_s;
            }
            else
            {
               g_bldc_motor1.pwm_duty = motor_pwm_s;
            }
        /****************** 三相电流计算 ****************/
        for(i=0; i<3; i++)
        {
            adc_val_m1[i] = g_adc_val[i+2];
            adc_amp[i] = adc_val_m1[i] -
              adc_amp_offset[i][ADC_AMP_OFFSET_TIMES];

            if(adc_amp[i]>=0)   /* 去除反电动势引起的负电流数据 */
             adc_amp_un[i]=adc_amp[i];
        }
        /* 运算母线电流（母线电流为任意两个有开关动作的相电流之和）*/
        if(g_bldc_motor1.step_sta == 0x05)
        {
            adc_amp_bus= (adc_amp_un[0]+ adc_amp_un[1])*ADC2CURT;/* UV */
        }
        else if(g_bldc_motor1.step_sta== 0x01)
        {
            adc_amp_bus= (adc_amp_un[0]+ adc_amp_un[2])*ADC2CURT;/* UW */
        }
        else if(g_bldc_motor1.step_sta== 0x03)
        {
            adc_amp_bus= (adc_amp_un[1]+ adc_amp_un[2])*ADC2CURT;/* VW */
```

```
            }
            else if(g_bldc_motor1.step_sta== 0x02)
            {
                adc_amp_bus= (adc_amp_un[0]+ adc_amp_un[1])*ADC2CURT;/* UV */
            }
            else if(g_bldc_motor1.step_sta == 0x06)
            {
                adc_amp_bus= (adc_amp_un[0]+ adc_amp_un[2])*ADC2CURT;/* WU */
            }
            else if(g_bldc_motor1.step_sta == 0x04)
            {
                adc_amp_bus= (adc_amp_un[2]+ adc_amp_un[1])*ADC2CURT;/* WV */
            }
        }
    }
    else if(htim->Instance == TIM6)
    {
        /************ 采集电机停机状态下的偏置电压 **************/
        times_count++;
        if(g_bldc_motor1.run_flag == STOP)
        {
            uint8_t i;
            uint32_t avg[3] = {0,0,0};
            adc_amp_offset[0][adc_amp_offset_p] = g_adc_val[2];
            adc_amp_offset[1][adc_amp_offset_p] = g_adc_val[3];

            adc_amp_offset[2][adc_amp_offset_p] = g_adc_val[4];

            adc_amp_offset_p ++;
            NUM_CLEAR(adc_amp_offset_p,ADC_AMP_OFFSET_TIMES);
            for(i=0; i<ADC_AMP_OFFSET_TIMES; i++)
            {
                avg[0] += adc_amp_offset[0][i];
                avg[1] += adc_amp_offset[1][i];
                avg[2] += adc_amp_offset[2][i];
            }
            for(i=0; i<3; i++)
            {
                avg[i] /= ADC_AMP_OFFSET_TIMES;
                adc_amp_offset[i][ADC_AMP_OFFSET_TIMES] = avg[i];
            }
        }
    }
}
```

主要看下速度计算部分，通过函数 uemf_edge 实现对霍尔状态变化的判定，当返回值会 0 时，开始计算电机转速，其中 g_bldc_motor1.count_j 代表霍尔的计数值，而进入定时器更新中断的频率 18KHz 即为霍尔的脉冲频率，我们所使用的无刷电机为 2 对级，所以根据前面所介绍的测速原理即可算出电机的实际转速，然后根据旋转方向设置速度正负，接着使用滤波算法过滤速度就得到了实际速度值，经过计数分频，进入 PID 计算函数，将上述计算的速度值代入 PID 控制算法，得到 PWM 输出值，最终赋值到定时器，在下一次中断时使用新的 PWM 用于驱动无刷电机。相应的主函数的 PWM 设定只需要换成 PID 的目标值设定即可，主函数 main.c 内容如下。

```
int main(void)
{
    uint16_t adc_vbus,adc_temp;
    uint8_t debug_cmd=0;
    float current_lpf[4]= {0.0f};
    uint8_t key,t;
    char buf[32];
    float current[3]= {0.0f};

    HAL_Init();                                 /* 初始化 HAL 库 */
```

```
    sys_stm32_clock_init(336, 8, 2, 7);     /* 设置时钟,168Mhz */
    delay_init(168);                        /* 延时初始化 */
    usart_init(115200);                     /* 串口初始化为 115200 */
    led_init();                             /* 初始化 LED */
    key_init();                             /* 初始化按键 */
    lcd_init();                             /* 初始化 LCD */
    bldc_init(168000/18-1,0);
    bldc_ctrl(MOTOR_1,CCW,0);               /* 初始无刷电机接口 1 速度 */
    pid_init();
    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"BLDC Motor Test",g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:Step++",g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:Step--",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:Stop",g_point_color);
    adc_nch_dma_init();

#if DEBUG_ENABLE                            /* 开启调试 */
    debug_init();                           /* PID 调试初始化 */
    debug_send_motorcode(BLDC_MOTOR );      /* 直流无刷电机 */
    debug_send_motorstate(IDLE_STATE);      /* 电机空闲 */
    /* 初始化同步数据（选择第 x 组 PIDX,目标速度地址,P,I,D 参数）到上位机 */
    debug_send_initdata(TYPE_PID1,User_SetPoint,S_KP,S_KI,S_KD);
    /*位置环 PID 参数（PID1）*/
#endif

    while (1)
    {
        t++;
        if(t % 20 == 0)
        {
            sprintf(buf,"PWM_Duty:%.1f%%",(float)((g_bldc_motor1.pwm_duty/
                    MAX_PWM_DUTY)*100));            /* 显示控制 PWM 占空比 */
            lcd_show_string(10,110,200,16,16,buf,g_point_color);
            sprintf(buf,"SetSpeed:%4d   ",(int16_t)*User_SetPoint);
            /* 显示设置速度 */
            lcd_show_string(10,110,200,16,16,buf,g_point_color);
            sprintf(buf,"M1 Speed:%4d   ",(int16_t)g_bldc_motor1.speed);
            lcd_show_string(10,130,200,16,16,buf,g_point_color);
            sprintf(buf,"M1 pos:%4d",g_bldc_motor1.pos); /* 显示测量速度 */
            lcd_show_string(10,150,200,16,16,buf,g_point_color);
            sprintf(buf,"Power:%.3fV ",g_adc_value[0]*ADC2VBUS);
            lcd_show_string(10,190,200,16,16,buf,g_point_color);
            sprintf(buf,"Temp:%.1fC ",get_temp(g_adc_value[1]));
            lcd_show_string(10,210,200,16,16,buf,g_point_color);

            LED0_TOGGLE();                  /* LED0(红灯) 翻转 */

            printf("PB1 数字原始值为: %d\r\n",(g_adc_value[0]));
            printf("PA0 数字原始值为: %d\r\n",(g_adc_value[1]));
            printf("PB0 数字原始值为: %d\r\n",(g_adc_value[2]));
            printf("PA6 数字原始值为: %d\r\n",(g_adc_value[3]));
            printf("PA3 数字原始值为: %d\r\n",(g_adc_value[4]));

            printf("Valtage:%.1fV \r\n",    g_adc_value[0]*ADC2VBUS);
            printf("Temp:%.1fC \r\n",       get_temp(g_adc_value[1]));
            printf("\r\n");

            if(g_bldc_motor1.run_flag==STOP)/* 停机的电流显示 */
            {
                current_lpf[0]=0;
                current_lpf[1]=0;
                current_lpf[2]=0;
```

```
        }
        current[0]=adc_amp_un[0]* ADC2CURT;/* U */
        current[1]=adc_amp_un[1]* ADC2CURT;/* V */
        current[2]=adc_amp_un[2]* ADC2CURT;/* W */

        /* 一阶数字滤波 滤波系数 0.1 用于显示 */
        FirstOrderRC_LPF(current_lpf[0],current[0],0.1f);
        FirstOrderRC_LPF(current_lpf[1],current[1],0.1f);
        FirstOrderRC_LPF(current_lpf[2],current[2],0.1f);
        FirstOrderRC_LPF(current_lpf[3],adc_amp_bus,0.1f);
        sprintf(buf,"Amp U:%.3fmA ",(float)current_lpf[0]);
        lcd_show_string(10,230,200,16,16,buf,g_point_color);
        sprintf(buf,"Amp V:%.3fmA ",(float)current_lpf[1]);
        lcd_show_string(10,250,200,16,16,buf,g_point_color);
        sprintf(buf,"Amp W:%.3fmA ",(float)current_lpf[2]);
        lcd_show_string(10,270,200,16,16,buf,g_point_color);
        sprintf(buf,"Amp Bus:%.3fmA ",(float)adc_amp_bus);
        lcd_show_string(10,290,200,16,16,buf,g_point_color);

        printf("Amp U:%.3fmA \r\n",(float)current_lpf[0]);
        printf("Amp V:%.3fmA \r\n",(float)current_lpf[1]);
        printf("Amp W:%.3fmA \r\n",(float)current_lpf[2]);
        printf("Amp Bus:%.3fmA \r\n",(float)adc_amp_bus);
    }

    key = key_scan(0);
    if(key == KEY0_PRES)
    {
        g_bldc_motor1.run_flag=RUN;                 /* 开启运行 */
        start_motor1();                             /* 开启运行 */
        if(*user_setpoint==0 && g_bldc_motor1.dir == CW)
        {
            g_bldc_motor1.dir = CCW;
        }

        *user_setpoint+=400;                        /* 逆时针旋转下递增 */
        if(*user_setpoint >= 4000)
            *user_setpoint = 4000;
        if(*user_setpoint==0)
        {
            g_bldc_motor1.run_flag = STOP;          /* 标记停机 */
            stop_motor1();                          /* 停机 */
            g_bldc_motor1.speed = 0;
            motor_pwm_s = 0;
            g_bldc_motor1.pwm_duty = 0;
        }
    }
    if(key == KEY1_PRES)                            /* 按下 KEY1 开启电机 */
    {
        g_bldc_motor1.run_flag=RUN;                 /* 开启运行 */
        start_motor1();                             /* 开启运行 */
        if(*user_setpoint==0 && g_bldc_motor1.dir == CCW)
        {
            g_bldc_motor1.dir = CW;
        }
        *user_setpoint -= 400;                      /* 逆时针旋转下递增 */
        if(*user_setpoint <= -4000)
            *user_setpoint = -4000;
        if(*user_setpoint==0)
        {
            g_bldc_motor1.run_flag = STOP;          /* 标记停机 */
            stop_motor1();                          /* 停机 */
            g_bldc_motor1.speed = 0;
            motor_pwm_s = 0;
            g_bldc_motor1.pwm_duty = 0;
```

```
            }
        }
        if(key == KEY2_PRES)                            /* 按下 KEY2 关闭电机 */
        {
            pid_init();
            g_bldc_motor1.count_j=0;
            bldc_speed_stop();
        }

#if DEBUG_ENABLE
        /* Debug 发送部分 */
        /* 主要显示参数 */
        debug_send_valtage(adc_vbus * ADC2VBUS);                /* 发送电压 */
        debug_send_speed(g_bldc_motor1.speed);                  /* 发送速度 */
        debug_send_distance((uint64_t)(g_bldc_motor1.sum_pos)); /* 发送总圈数 */
        debug_send_temp(50,get_temp(adc_temp));/* 发送电机温度、驱动板温度 */
        debug_send_current((float)(current_lpf[0]/1000),(float)(current_lpf[0]\
                           /1000),(float)(current_lpf[0]/1000));/*发送电流*/
        /* 电流波形和速度波形 */
        debug_send_wave_data(1,(int16_t)g_bldc_motor1.speed);
        /* 选择通道 1 发送实际速度 */
        debug_send_wave_data(2,(int16_t)*user_setpoint);/*选择通道 1 发送实际速度*/
        debug_send_wave_data(3,current_lpf[0]);      /*选择通道 1 发送实际电流 u*/
        debug_send_wave_data(4,current_lpf[1]);      /*选择通道 2 发送实际电流 v*/
        debug_send_wave_data(5,current_lpf[2]);      /*选择通道 3 发送实际电流 w*/
        debug_send_wave_data(6,current_lpf[3]);      /*选择通道 3 发送实际母线电流*/
        /* debug 接收部分 */
        debug_receive_pid(TYPE_PID1,(float*)&g_speed_pid.Proportion,
        /* 查询接收 PID 助手的 PID1 参数 */
        (float*)&g_speed_pid.Integral,(float*)&g_speed_pid.Derivative);
        debug_cmd=debug_receive_ctrl_code();            /* 读取命令 */
        if(debug_cmd==HALT_CODE)                        /* 停机 */
        {
            stop_motor1();
            pid_init();
            g_bldc_motor1.run_flag=STOP;                /* 标记停机 */
            g_bldc_motor1.pwm_duty=0;
        }
        else if(debug_cmd==RUN_CODE)                    /* 运行 */
        {
            g_bldc_motor1.run_flag=RUN;                 /* 运行标记 */
            *user_setpoint=400;                         /* 自动设置目标 */
            debug_data_temp=*user_setpoint;
            start_motor1();                             /* 启动电机 */
            debug_send_motorstate(RUN_STATE);           /* 电机运行 */
        }
        else if (debug_cmd==BREAKED)/* 刹车（电机停止 点击电机运行才可解除）*/
        {
            *user_setpoint=0;                           /* 减速直至 0 */
            debug_send_motorstate(BREAKED_STATE);       /* 电机刹车 */
        }
#endif

        delay_ms(10);
    }
}

void bldc_speed_stop(void)
{
    pid_init();                         /* 重新初始化 PID，防止积分过大失控 */
    g_bldc_motor1.run_flag = STOP;      /* 标记停机 */
    stop_motor1();                      /* 停机 */
```

```
    g_bldc_motor1.speed = 0;
    motor_pwm_s = 0;
    g_bldc_motor1.pwm_duty = 0;
}
```

main 函数主要初始化基本外设以及 BLDC 等等，然后添加上位机通信协议，将实际转速数据发送给上位机的通道 1 上进行显示，目标转速数据发送给上位机的通道 2 上进行显示。按键判断逻辑：按下 KEY0 目标转速加 400RPM，按下 KEY1 目标转速减 400RPM，按下 KEY2 停止电机旋转。

24.1.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 上显示按键功能、占空比以及电机速度信息和显示程序计算的电压、电流等信息，当我们按下 KEY0，目标速度将增大；按下 KEY1，目标速度将减小；按下 KEY2，电机将停止。目标速度为正数时，电机正转，反之电机反转。我们再打开 PID 调试助手（注意接上串口 1 的 USB_TTL 连接至电脑），选择对应的串口端口，接着勾选通道 1 和 2，点击“开始”按钮，即可开始显示波形，如下图 24.1.3.1，大家可以自己下载程序并点击按键测试效果。

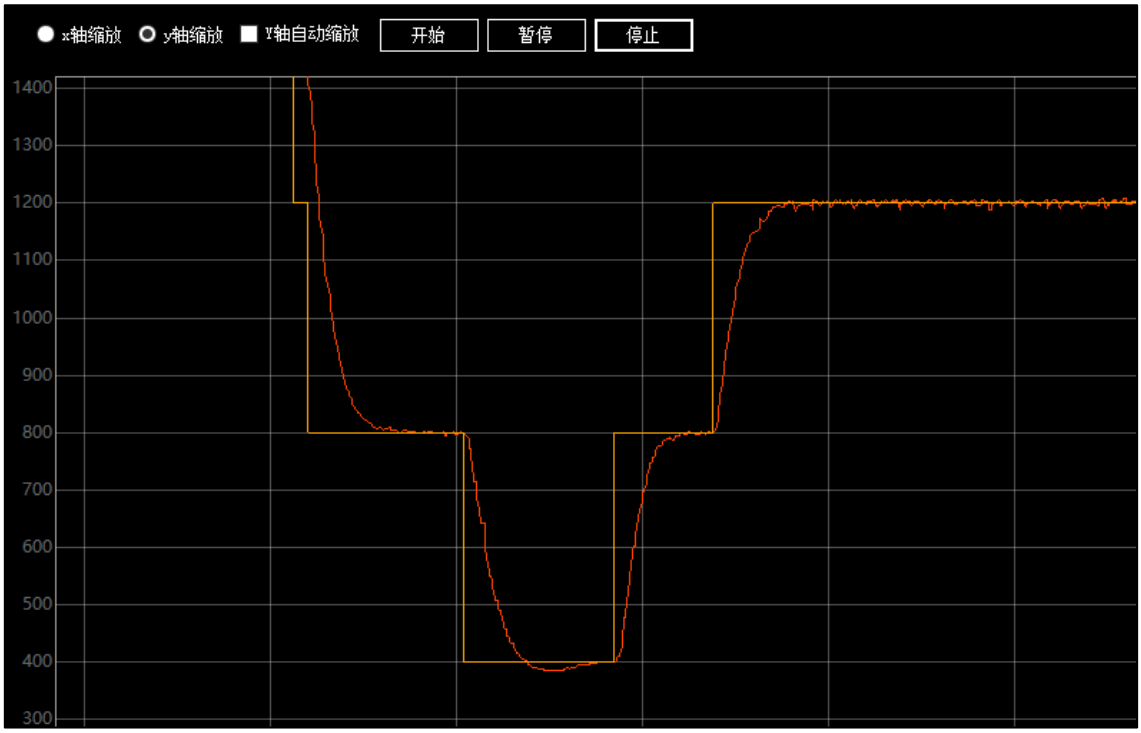


图 24.1.4.1 速度 PID 调节效果

图 24.1.4.1 中，橙线代表目标速度，红线代表实际速度，当我们按下 KEY0，目标速度增大，橙线先发生变化，而红线（实际速度）会逐渐靠近橙线（目标速度）；按下 KEY1，目标速度将减小，曲线的变化同理；按下 KEY2，电机将停止，目标速度将为 0。

注意：1、本实验需要使用 USB 数据线连接开发板的串口 1 到电脑，并启动电机之后才会有波形变化；2、如果发现波形不对，请检查电机接线；3、PID 系数并不是通用的，如果 PID 曲线不理想，大家需要根据自己的实际系统去调节。

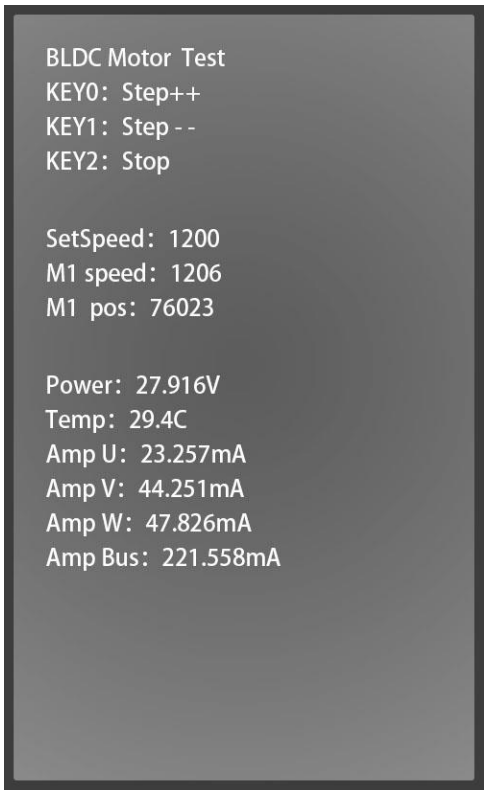


图 24.1.4.2 屏幕显示效果

24.2 速度&电流双闭环控制

要实现速度+电流双闭环控制，只需在速度环的基础上加上电流环部分即可。三相电流的采集已经在前面的课程（第 23 章）学习过了，所以下面我们直接进入代码实现。

24.2.1 硬件设计

1、例程功能

- 本实验以电机开发板的直流有刷/无刷电机驱动接口 1 为例，基于无刷速度闭环实验，加入电流环控制，对电机进行速度环+电流环双闭环控制。
- 通过 PID 电流环的初始化设置转矩大小，即电流大小。目标电流越大，转矩越大。
- 当按键 0 按下，就增大目标速度值；当按键 1 按下，就减小目标速度值。目标速度的绝对值大小决定电机的速度，它的正负决定电机的正反转方向。按下按键 2 则马上停止电机。
- 屏幕显示按键功能、占空比、目标转速以及实际转速以及显示无刷电机电压、电流和温度、速度等等信息。
- 可通过串口 1 即 USB_TTL 接口连接正点原子 PID 调试助手，查看 PID 波形。
- LED0 闪烁指示程序运行。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器：
TIM1_CH1：PA8
TIM1_CH2：PA9
TIM1_CH3：PA10
IO：PB13\14\15
使能引脚：SHDN: PF10
4）无刷电机
5）无刷电机驱动板
6）12-60V 的 DC 电源
7）ADC：ADC1_CH9：PB1
ADC1_CH0：PA0
ADC1_CH8：PB0
ADC1_CH6：PA6
ADC1_CH3：PA3

3、原理图

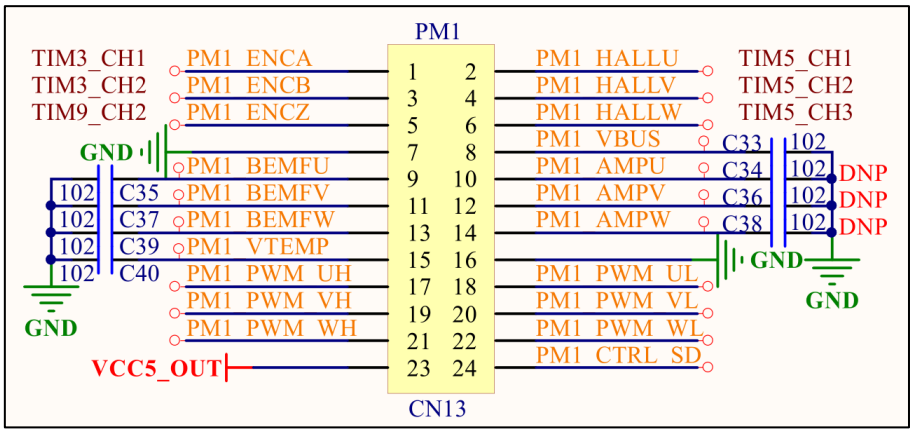


图 24.2.1.1 无刷电机接口 1 原理图

接口涉及的 IO 如下。

直流无刷驱动板	IO 口
驱动板 U 相	PA8、PB13
驱动板 V 相	PA9、PB14
驱动板 W 相	PA10、PB15
霍尔 U 相	PH10
霍尔 V 相	PH11
霍尔 W 相	PH12
输出使能	PF10
电压	PB1
温度	PA0
电流 U 相	PB0
电流 V 相	PA6
电流 W 相	PA3

表 24.2.1.1 无刷相关 IO 口说明

驱动板和电机的连接方式之前已经介绍过了，驱动板和开发板连接只需要使用排线连接即可，实物连接如下。

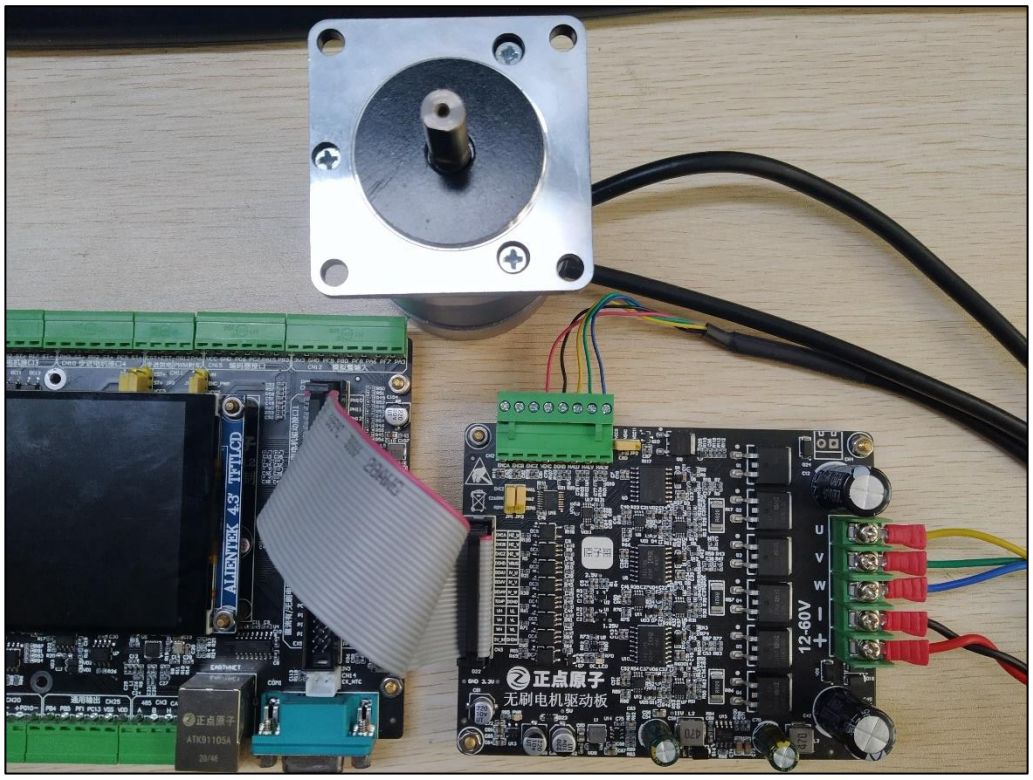


图 24.2.1.2 开发板&驱动板连接图

24.2.2 程序设计

24.2.2.1 无刷电机速度+电流双闭环的配置步骤

1）实现电机基本驱动

实现电机的基础驱动函数，启停、6 步换向组合等等

2）初始化 ADC

初始化 ADC 通道的 IO，设置 ADC 工作方式、DMA 等等

3）PID 闭环控制实现

实现 PID 的初始参数赋值、PID 计算等等

4）上位机通信协议

编写上位机通信代码，可在上位机上实时显示当前速度与目标速度的波形变化

5）编写中断服务函数

PWM 中断用于换向控制、堵转检测等等，添加 PID 的周期计算调用

24.2.2.2 程序流程图

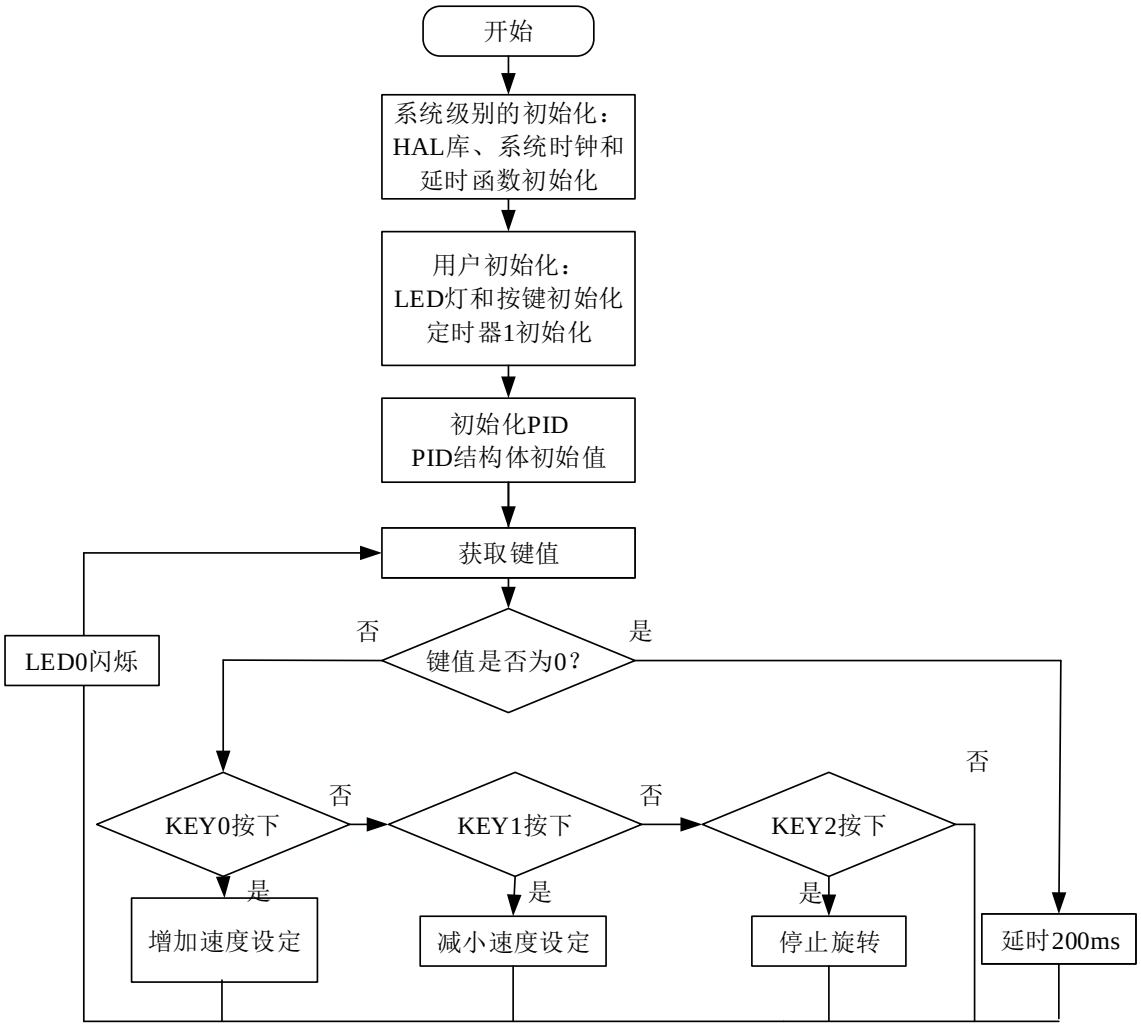


图 24.2.2.2.1 无刷电机速度+电流双闭环控制流程图

24.2.2.3 程序解析

这里我们只讲解核心代码，定时器及 ADC 的相关程序都和之前一致，这里就不重复列出了，速度+电流双闭环的 PID 参数的定义有不同。实际内容如下所示。

```
/* PID 相关参数 */

#define  C_KP      2.00f              /* P 参数 */
#define  C_KI      0.20f              /* I 参数 */
#define  C_KD      0.01f              /* D 参数 */

#define  S_KP      0.00800f           /* P 参数 */
#define  S_KI      0.00025f           /* I 参数 */
#define  S_KD      0.00020f           /* D 参数 */
#define SMAPLSE_PID_SPEED  40        /* 采样率 单位 ms */
```

其中 C_KP，C_KI，C_KD 为电流环的 PID 参数，S_KP，S_KI，S_KD 为速度环的 PID 参数，PID 的计算过程和之前的一致，这里就不贴出来了，大家感兴趣可以打开源码查看。重点内容放在定时器中断回调函数中，在 bldc_tim.c 中实现，贴出的程序删除了部分换向相关的代码避免篇幅太长，中断回调函数内容如下：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    uint8_t i;
```

```
    uint8_t bldc_dir=0;
    static uint8_t times_count=0;           /* 定时器时间记录 */
    int16_t temp_speed=0;                   /* 临时速度存储 */
    if(htim->Instance == ATIM_TIMX_PWM)     /* 55us */
    {
        if(g_bldc_motor1.run_flag == RUN)
        {
            … …
            /********************** 速度计算 **********************/
            g_bldc_motor1.count_j++;                    /* 计算速度专用计数值 */
            g_bldc_motor1.hall_sta_edge =
            uemf_edge(g_bldc_motor1.hall_single_sta);
            if(g_bldc_motor1.hall_sta_edge == 0)        /*统计单个霍尔信号的高电平时间*/
            {
                /* 计算速度 */
                if(g_bldc_motor1.dir == CW)
                    temp_speed = (SPEED_COEFF/g_bldc_motor1.count_j);
                else
                    temp_speed = -(SPEED_COEFF/g_bldc_motor1.count_j);
                FirstOrderRC_LPF(g_bldc_motor1.speed,temp_speed,0.2379);
                g_bldc_motor1.no_single = 0;
                g_bldc_motor1.count_j = 0;
            }
            if(g_bldc_motor1.hall_sta_edge == 1)
            {
                g_bldc_motor1.no_single = 0;
                g_bldc_motor1.count_j = 0;
            }
            if(g_bldc_motor1.hall_sta_edge == 2)        /* 霍尔值一直不变代表未换向 */
            {
                g_bldc_motor1.no_single++;              /* 不换相时间累计*/
                if(g_bldc_motor1.no_single > 15000)
                {
                    g_bldc_motor1.no_single = 0;
                    g_bldc_motor1.speed = 0;            /* 超时换向 判定为停止 */
                }
            }
            /******************* 位置记录以及堵转标记 **************/
            if(g_bldc_motor1.step_last != g_bldc_motor1.step_sta)
            {
                g_bldc_motor1.hall_keep_t = 0;
                bldc_dir = check_hall_dir(&g_bldc_motor1);
                if(bldc_dir == CCW)
                {
                    g_bldc_motor1.pos -=1;
                }
                else if(bldc_dir == CW)
                {
                    g_bldc_motor1.pos +=1;
                }
                g_bldc_motor1.step_last = g_bldc_motor1.step_sta;
            }
            else if(g_bldc_motor1.run_flag == RUN)      /* 运行且霍尔保持时 */
            {
                g_bldc_motor1.hall_keep_t++;            /* 换向一次所需计数值*/
                if(g_bldc_motor1.hall_keep_t > 15000)   /* 堵转 */
                {
                    g_bldc_motor1.hall_keep_t = 0;
#if LOCK_TAC
                    stop_motor1();
                    g_bldc_motor1.run_flag = STOP;;     /* 标记停机 */
                    g_bldc_motor1.pwm_duty = 0;
#endif
                    g_bldc_motor1.locked_rotor = 1;     /* 标记堵转 */
```

```
            }
        }
    /********************* PID 控制 **********************/
    if(g_bldc_motor1.run_flag == RUN)                /*进入 PID 闭环控制*/
    {
        pid_c_count++;
        pid_s_count++;

        if(pid_s_count > 2)
        {
/* 开启上位机调试 在 PID 执行之前对调节数值进行预先判断 */
#if DEBUG_ENABLE
            /* 控制目标调节范围(3000~-3000)并且最大步进值不超过 6000 RPM */
            debug_set_point_range(3000,-3000,6000);
            debug_data_temp = *user_setpoint;
            if(*user_setpoint < 0)                   /* 上位机指令切换旋转方向 */
            {
                /* 为简化控制逻辑 只以电机状态作为判依据（同步）*/
                if(g_bldc_motor1.speed == 0)
                {
                    g_bldc_motor1.dir = CCW;
                }
                else if(g_bldc_motor1.dir == CW &&g_bldc_motor1.speed != 0)
                {
                    *user_setpoint = 0;
                }
            }
            else if(*user_setpoint > 0)
            {
                if(g_bldc_motor1.speed == 0)
                {
                    g_bldc_motor1.dir = CW;
                }
                else if(g_bldc_motor1.dir == CCW&&g_bldc_motor1.speed != 0)
                    *user_setpoint = 0;
                }
            }
#endif
            /******************* PID 计算 *******************/
            temp_pwm1 = increment_pid_ctrl(&g_speed_pid,g_bldc_motor1.speed);
            FirstOrderRC_LPF(motor_pwm_s,temp_pwm1,0.085);
            if(motor_pwm_s < 0)
            {
                motor_pwm_sl = -motor_pwm_s;
            }
            else
            {
                motor_pwm_sl = motor_pwm_s;
            }

            *user_setpoint = debug_data_temp;     /* 重新保持上位机指令要求 */
            pid_s_count = 0;
        }

        if(pid_c_count > 1)                          /* 电流环 */
        {
            /* 换向尖峰电流大于设定的电流值将导致 PID 调节转至电流环调节 */
            if(adc_amp_bus > (g_current_pid.SetPoint - 20))
            {
                cf_count++;                          /* 滤除换向尖峰电流的影响 */
                if(cf_count > 4)
                {
                    cf_count = 0;
                    temp_pwm2 =
                     increment_pid_ctrl(&g_current_pid,adc_amp_bus);
```

```
                    FirstOrderRC_LPF(motor_pwm_c,temp_pwm2,0.085);
                }
            }
            else
            {
                cf_count = 0;
                temp_pwm2 = increment_pid_ctrl(&g_current_pid,adc_amp_bus);
                FirstOrderRC_LPF(motor_pwm_c,temp_pwm2,0.085);
            }
            pid_c_count = 0;
        }
        /* 电流环输出值大于速度环输出则使用速度环调节 */
        if(motor_pwm_c > motor_pwm_sl)
        {
            g_bldc_motor1.pwm_duty = motor_pwm_sl;
            if(motor_pwm_s < 0)                    /* 正反转积分控制 */
                g_current_pid.Ui = -g_speed_pid.Ui;
            else
                g_current_pid.Ui = g_speed_pid.Ui;
        }
        else  /* 速度环输出值大于电流环输出则使用电流环调节 */
        {
            g_bldc_motor1.pwm_duty = motor_pwm_c;
            if(motor_pwm_s < 0)
                g_speed_pid.Ui = -g_current_pid.Ui;
            else
                g_speed_pid.Ui = g_current_pid.Ui;
        }
    }
    }
}
if(htim->Instance == TIM6)
{
    /***************** 采集电机停机状态下的偏置电压 *****************/
    times_count++;
    if(g_bldc_motor1.run_flag == STOP)
    {
        uint8_t i;
        uint32_t avg[3] = {0,0,0};
        /* 得到还未开始运动时三相的基准电压 */
        adc_amp_offset[0][adc_amp_offset_p] = g_adc_val[2];
        adc_amp_offset[1][adc_amp_offset_p] = g_adc_val[3];
        adc_amp_offset[2][adc_amp_offset_p] = g_adc_val[4];

        adc_amp_offset_p++;
        /* 最大采集 ADC_AMP_OFFSET_TIMES 次，超过即从 0 开始继续采集 */
        NUM_CLEAR(adc_amp_offset_p,ADC_AMP_OFFSET_TIMES);
        /* 将采集的每个通道值累加 */
        for(i = 0; i < ADC_AMP_OFFSET_TIMES; i++)
        {
            avg[0] += adc_amp_offset[0][i];
            avg[1] += adc_amp_offset[1][i];
            avg[2] += adc_amp_offset[2][i];
        }
        for(i = 0; i < 3; i++)                /* 取平均即软件滤波 */
        {
            avg[i] /= ADC_AMP_OFFSET_TIMES;
            adc_amp_offset[i][ADC_AMP_OFFSET_TIMES] = avg[i];
        }
    }
    /****************** 定时判断电机是否发生堵塞 ********************/
    if(times_count == SMAPLSE_PID_SPEED)
    {

        if(g_bldc_motor1.locked_rotor == 1)          /* 堵转处理*/
```

```
            {
                clc++;
                if(clc > 50)                                    /* 延迟 2s 后重新启动 */
                {
#if DEBUG_ENABLE /*开启调试*/
                    debug_send_motorstate(RUN_STATE);           /* 电机运行*/
#endif
                    clc = 0;
                    pid_init();
                    stop_motor1();
                    g_speed_pid.SetPoint = 400.0;               /* 400PRM */
                    g_bldc_motor1.pwm_duty = 500;               /* 加速启动速度 */
                    g_bldc_motor1.run_flag = RUN;               /* 开启运行 */
                    start_motor1();                             /* 运行电机 */
                    g_bldc_motor1.locked_rotor = 0;
                }
            }

            times_count = 0;
        }

    }
}
```

在定时器 1 中断里首先优先进行电流环的 PID 计算，这里需要进行滤波，避免换向尖峰对 PID 的影响，接着通过霍尔跳变数量计算电机转速，经过计数分频，进入速度环 PID 计算函数，带入上述计算的速度，得到速度环计算出的 PWM 输出值，然后将两个环的输出做比较，如果电流环较大则使用速度环输出，否则使用电流环输出，最终赋值到定时器，在下一次中断时使用新的 PWM 用于驱动无刷电机。这里我们还在中断中加上了堵转处理，当电机因意外或过流导致停机时，当堵转时间超过 2s，此时会重新初始化 PID 并以 400RPM 的初始速度重新开启电机旋转。接着看来先下主函数 main.c 内容如下。

```
int main(void)
{
    uint16_t adc_vbus,adc_temp;
    uint8_t debug_cmd=0;
    float current_lpf[4]= {0.0f};
    uint8_t key,t;
    char buf[32];
    float current[3]= {0.0f};
    uint16_t strar_sf=0;//启动速度标识
    int16_t speed_diplay=0;
    float User_SetPoint_temp=0.0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    bldc_init(168000/18-1,0);
    bldc_ctrl(MOTOR_1,CCW,0);                   /* 初始无刷电机接口 1 速度 */
    pid_init();
    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"BLDC Motor Test",g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:Step++",g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:Step--",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:Stop",g_point_color);
    adc_nch_dma_init();

#if DEBUG_ENABLE    /* 开启调试 */
    debug_init();   /* PID 调试初始化 */
```

```
    debug_send_motorcode(BLDC_MOTOR );/* 直流电机 */
    debug_send_motorstate(IDLE_STATE);/* 电机空闲 */
    /* 初始化同步数据（选择第 x 组 PIDX,目标速度地址,P,I,D 参数）到上位机 */
    debug_send_initdata(TYPE_PID1,User_SetPoint,C_KP,C_KI,C_KD);
    /* 位置环 PID 参数（PID1） */
    debug_send_initdata(TYPE_PID2,User_SetPoint,S_KP,S_KI,S_KD);
    /* 速度环 PID 参数（PID2） */
#endif
    while (1)
    {
        t++;
        if(t % 20 == 0)
        {
             sprintf(buf,"PWM_Duty:%.1f%%  ",
            (float)((g_bldc_motor1.pwm_duty/MAX_PWM_DUTY)*100));
            lcd_show_string(10,110,200,16,16,buf,g_point_color);

            user_setpoint_temp = (*user_setpoint);
            speed_diplay = g_bldc_motor1.speed;
            /* 显示设置速度 */
            sprintf(buf,"SetSpeed:%4d   ",(int16_t)user_setpoint_temp);
            lcd_show_string(10,110,200,16,16,buf,g_point_color);
            /* 显示转速 */
            sprintf(buf,"M1 speed:%4d   ",speed_diplay);
            lcd_show_string(10,130,200,16,16,buf,g_point_color);
            /* 显示位置变化 */
            sprintf(buf,"M1 pos:%4d",g_bldc_motor1.pos);
            lcd_show_string(10,150,200,16,16,buf,g_point_color);
            sprintf(buf,"PWM_Duty:%.1f%%  ",
            (float)((g_bldc_motor1.pwm_duty/(float)MAX_PWM_DUTY)*100));
            /* 显示控制 PWM 占空比 */
            lcd_show_string(10,170,200,16,16,buf,g_point_color);


            sprintf(buf,"Power:%.3fV ",g_adc_value[0]*ADC2VBUS);
            lcd_show_string(10,190,200,16,16,buf,g_point_color);
            sprintf(buf,"Temp:%.1fC ",get_temp(g_adc_value[1]));
            lcd_show_string(10,210,200,16,16,buf,g_point_color);
            LED0_TOGGLE();                                      /* LED0(红灯) 翻转 */
            /* 三相电流计算 */
            current[0] = adc_amp_un[0]* ADC2CURT;/*U*/
            current[1] = adc_amp_un[1]* ADC2CURT;/*V*/
            current[2] = adc_amp_un[2]* ADC2CURT;/*W*/

            /*一阶数字滤波 滤波系数 0.1 用于显示*/
            FirstOrderRC_LPF(current_lpf[0],current[0],0.1f);
            FirstOrderRC_LPF(current_lpf[1],current[1],0.1f);
            FirstOrderRC_LPF(current_lpf[2],current[2],0.1f);
            FirstOrderRC_LPF(current_lpf[3],adc_amp_bus,0.1f);
            if(g_bldc_motor1.run_flag == STOP)                  /* 停机的电流显示 */
            {
                current_lpf[0]=0;
                current_lpf[1]=0;
                current_lpf[2]=0;
                current_lpf[3]=0;
            }

            sprintf(buf,"Amp U:%.3fmA ",(float)current_lpf[0]);
            lcd_show_string(10,230,200,16,16,buf,g_point_color);
            sprintf(buf,"Amp V:%.3fmA ",(float)current_lpf[1]);
            lcd_show_string(10,250,200,16,16,buf,g_point_color);
            sprintf(buf,"Amp W:%.3fmA ",(float)current_lpf[2]);
            lcd_show_string(10,270,200,16,16,buf,g_point_color);
            sprintf(buf,"Amp Bus:%.3fmA ",(float)adc_amp_bus);
```

```
            lcd_show_string(10,290,200,16,16,buf,g_point_color);
        }

        key = key_scan(0);
        if(key == KEY0_PRES)                                /* 按下 KEY0 目标速度值++*/
        {
            g_bldc_motor1.run_flag = RUN;                   /* 开启运行 */
            start_motor1();                                 /* 开启运行 */

            if(g_bldc_motor1.dir == CCW && *user_setpoint == 0) /* 切换方向条件*/
            {
                g_bldc_motor1.dir = CW;
            }
            *user_setpoint += 400;                          /* 逆时针旋转下递增 */
            if(*user_setpoint >= 3000)                      /* 最高不超过 3000PRM */
                *user_setpoint = 3000;
            if(*user_setpoint == 0)
            {
                pid_init();                                 /* 初始化 PID */
                g_bldc_motor1.run_flag = STOP;              /* 标记停机 */
                stop_motor1();                              /* 停机 */
                g_bldc_motor1.speed = 0;
                motor_pwm_s = 0;
                g_bldc_motor1.pwm_duty = 0;
            }
            debug_data_temp = *user_setpoint;
        }
        else if(key == KEY1_PRES)                           /* 按下 KEY1 目标速度值--*/
        {
            g_bldc_motor1.run_flag = RUN;                   /* 开启运行 */
            start_motor1();                                 /* 运行电机 */
            /* 切换方向条件 */
            if(g_bldc_motor1.dir == CW && *user_setpoint == 0)
            {
                g_bldc_motor1.dir = CCW;
            }
            *user_setpoint -= 400;                          /* 逆时针旋转下递增 */
            if(*user_setpoint <= -3000)                     /* 最高不超过 300PRM */
                *user_setpoint = -3000;
            if(*user_setpoint == 0)
            {
                pid_init();                                 /* 初始化 PID */
                g_bldc_motor1.run_flag = STOP;              /* 标记停机 */
                stop_motor1();                              /* 停机 */
                g_bldc_motor1.speed = 0;
                motor_pwm_s = 0;
                g_bldc_motor1.pwm_duty = 0;
            }

            debug_data_temp = *user_setpoint;
        }
        else if(key == KEY2_PRES)                               /* 按下 KEY2 关闭电机 */
        {
            bldc_speed_stop();
        }

#if DEBUG_ENABLE
        /* Debug 发送部分 */
        /* 主要显示参数 */
        debug_send_valtage(adc_vbus * ADC2VBUS);                /* 发送电压 */
        debug_send_speed(g_bldc_motor1.speed);                  /* 发送速度 */
        debug_send_distance((uint64_t)(g_bldc_motor1.sum_pos)); /* 发送总圈数 */
        debug_send_temp(50,get_temp(adc_temp));/* 发送电机温度、驱动板温度 */
```

```
        debug_send_current((float)(current_lpf[0]/1000),(float)(current_lpf[0]/
                           1000),(float)(current_lpf[0]/1000));/* 发送电流 */
        /* 选择通道 1 发送实际速度 */
        debug_send_wave_data(1,(int16_t)g_bldc_motor1.speed);
        /* 选择通道 2 发送目标速度 */
        debug_send_wave_data(2,(int16_t)*user_setpoint);
        /* 选择通道 3 发送实际电流 U */
        debug_send_wave_data(3,current_lpf[0]);
        /* 选择通道 4 发送实际电流 V */
        debug_send_wave_data(4,current_lpf[1]);
        /* 选择通道 5 发送实际电流 W */
        debug_send_wave_data(5,current_lpf[2]);
        /* 选择通道 7 发送实际电流 */
        debug_send_wave_data(7,current_lpf[3]);
        /* 选择通道 8 发送目标电流 */
        debug_send_wave_data(8,g_current_pid.SetPoint);
        /* Debug 接收部分 */
        debug_receive_pid(TYPE_PID1,(float*)&g_current_pid.Proportion,
        /* 查询接收 PID 助手的 PID1 参数 */
        (float*)&g_current_pid.Integral,(float*)&g_current_pid.Derivative);
        debug_receive_pid(TYPE_PID2,(float*)&g_speed_pid.Proportion,
        /* 查询接收 PID 助手的 PID2 参数 */
        (float*)&g_speed_pid.Integral,(float*)&g_speed_pid.Derivative);
        debug_cmd=debug_receive_ctrl_code();/* 读取命令 */
        if(debug_cmd==HALT_CODE)/* 停机 */
        {
            stop_motor1();
            pid_init();
            g_bldc_motor1.run_flag=STOP;      /* 标记停机 */
            g_bldc_motor1.pwm_duty=0;
        }
        else if(debug_cmd==RUN_CODE)          /* 运行 */
        {
            g_bldc_motor1.run_flag=RUN;       /* 运行标记 */
            *user_setpoint=400;               /* 自动设置目标 */
            debug_data_temp=*user_setpoint;
            start_motor1();                   /* 启动电机 */
            debug_send_motorstate(RUN_STATE);/* 电机运行 */
        }
        else if (debug_cmd==BREAKED)          /* 刹车（电机停止 点击电机运行才可解除）*/
        {
            *user_setpoint=0;                 /* 减速直至 0*/
            debug_send_motorstate(BREAKED_STATE);/*电机刹车*/
        }
#endif

        delay_ms(10);
    }
}

void bldc_speed_stop(void)
{
    pid_init();
    g_bldc_motor1.run_flag = STOP;  /* 标记停机 */
    stop_motor1();                  /* 停机 */
    g_bldc_motor1.speed = 0;
    motor_pwm_s = 0;
    g_bldc_motor1.pwm_duty = 0;
}
```

main 函数主要初始化基本外设以及 BLDC 等等，然后添加上位机通信协议，将实际转速数据、目标转速数据、实际电流、目标电流分别发送给上位机的通道 1、2、7、8 上进行显示。按键判断逻辑：按下 KEY0 目标转速加 400RPM，按下 KEY1 目标转速减 400RPM，按下 KEY2

停止电机旋转

24.2.3 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 上显示按键功能、占空比以及电机速度信息和显示程序计算的电压、电流等信息，当我们按下 KEY0，目标速度将增大；按下 KEY1，目标速度将减小；按下 KEY2，电机将停止。目标速度为正数时，电机正转，反之电机反转。我们再打开 PID 调试助手（注意接上串口 1 的 USB_TTL 连接至电脑），选择对应的串口端口，接着勾选通道 1 和 2 以及通道 7 和 8，点击“开始”按钮，即可开始显示波形。大家可以使用手给电机增加点负载，可以明显感觉到电机一开始会因为负载增加速度减慢，后续逐渐加速上去直至和目标速度一致，如下图 24.2.3.1，大家可以自己下载程序并点击按键测试效果。

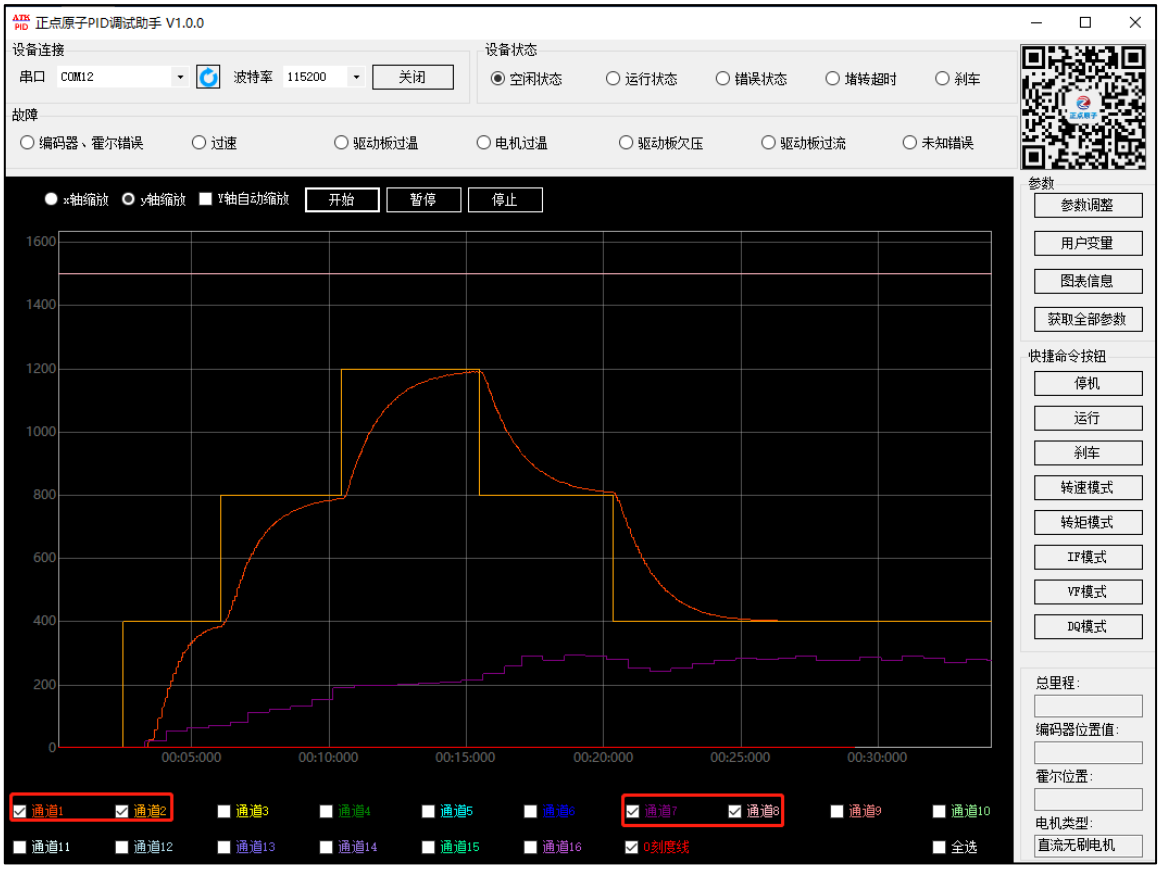


图 24.2.3.1 PID 调节效果

图 24.2.3.1 中，橙线代表目标速度，红线代表实际速度，紫线代表实际母线电流，粉线代表目标电流（由于 PID 初始化时已经设定了电流环的目标值，所以粉线为直线），当我们按下 KEY0，目标速度增大，橙线先发生变化，而红线（实际速度）会逐渐靠近橙线（目标速度）；可以看到紫线（实际电流）并未靠近粉线（目标电流），是由于代码中将两个环的输出做比较，如果电流环较大则使用速度环输出，否则使用电流环输出导致的。当我们手动给电机加负载，就能明显看到实际电流会在目标电流附近波动；按下 KEY1，目标速度将减小，曲线的变化同理；按下 KEY2，电机将停止，目标速度将为 0。

注意：1、电流环的波形存在小幅振荡属于正常现象，如果希望波形更稳定，可以适当地调整 PID 系数以及增大滤波次数；2、滤波次数越多则系统的响应越慢，大家需要在系统的响应速度和稳定性之间寻找平衡点；3、PID 系数并不是通用的，如果 PID 曲线不理想，大家需要根据自己的实际系统去调节。

第 25 章 直流无刷电机无感控制

前面的课程我们学习了无刷电机的**有感控制**，即通过霍尔传感器检测转子位置，实现六步换相，从而使无刷电机旋转起来。本章我们开始来学习**无感控制**，即不通过位置传感器实现无刷电机的基本驱动。

本章分为如下几个小节：

25.1 直流无刷电机的无感驱动介绍
25.2 硬件设计
25.3 程序设计
25.4 下载验证

25.1 直流无刷电机的无感驱动介绍

如前面有感控制 21.3 小节所述，无刷直流电机的工作原理必须有转子磁场位置的信息，以控制逆变器功率器件的开/关实现绕组的换相。例如，三相六状态运行的无刷电机在内部安放三个转子位置传感器确定六个换相点时刻。传统的无刷直流电机转子位置信息是采用机电式或电子式传感器直接检测，如霍尔传感器、光电传感器等，然而，在实际应用中发现，在电机内部安放转子位置传感器有以下问题:

1) 在某些高温、低温、高振动、潮湿、污浊空气和高干扰等恶劣的工作环境下由于位置传感器的存在使系统的可靠性降低。
2) 位置传感器电气连接线多，不便于安装，而且易引入电磁干扰。
3) 传感器的安装精度直接影响电机运行性能。特别是在多极电机安装精度难以保证
4) 位置传感器占用电机结构空间，限制了电机的小型化。

所以无传感器驱动的方式已成为一个新的技术发展方向，虽然无传感器的方式会降低控制精度，但好处是使系统能够在恶劣的工作环境下可靠运行，同时使电机的结构变得简单，体积减小、安装更方便、成本降低。

无感检测的主要方法有：反电动势过零点检测方法、反电动势积分范有电压比较法、反电动势积分及领相环法、续流二极管法、反电动势三次谱波检测法、感侧量法、C(0) 函数法、扩展卡尔曼滤波法、状态观测器法等。

这里我们主要使用**反电动势过零点检测方法**。它的主要核心就是通过检测定子绕组的反电动势过零点来判断转子当前的位置。与有霍尔的方案相比，最明显的优点就是降低了成本、减小了体积。并且电机引线仅 8 根变为 3 根，使接线调试都大为简化。另外，霍尔传感器容易受到温度和磁场等外界环境的影响，故障率较高。因此，无霍尔 BLDC 得到越来越多的应用，在很多场合正逐步取代有霍尔 BLDC。介绍三相 BLDC 电机的无霍尔控制理论。根据特定的应用场合，具体的实现方法会有所不同。

25.1.1 反电动势法控制 BLDC 电机的原理

三相六状态 120° 通电方式运行的无刷电机在任意时刻总是两相通电工作，另相绕组是浮地不导通的。这时候非导通绕组的端电压(从绕组端部到直流地之间)或相电压(从绕组端部到三相绕组中心点之间)就反映出该相绕组的感应电动势（BEMF，BackElectromotiveForce）。BLDC 电机的 BEMF 波形随转子的位置和速度变化，整体上呈现为梯形。

图 25.1.1.1 给出了电机旋转一个电周期中电流和反电电势的波形，其中实线代表电流，虚线代表反电动势，横坐标为电机旋转的电气角度，根据 BLDC 的“六步换向”控制理论，我们知道在任意时刻三相 BLDC 另有两相通电，另一相开路，三相两两导通，共有 6 种组合，以一定的顺序每 60° 变化一次，这样产生旋转的磁场，拉动永磁体转子跟着转动。这里的 60° 指的是电气角度，一个电周期可能并不是对应一个完整的转子机械转动周期。完成一圈机械转动要重复的电周期数取决于转子的磁极对数。每对转子磁极需要完成一个电周期，因此，电周期数/转数等于转子磁极对数。

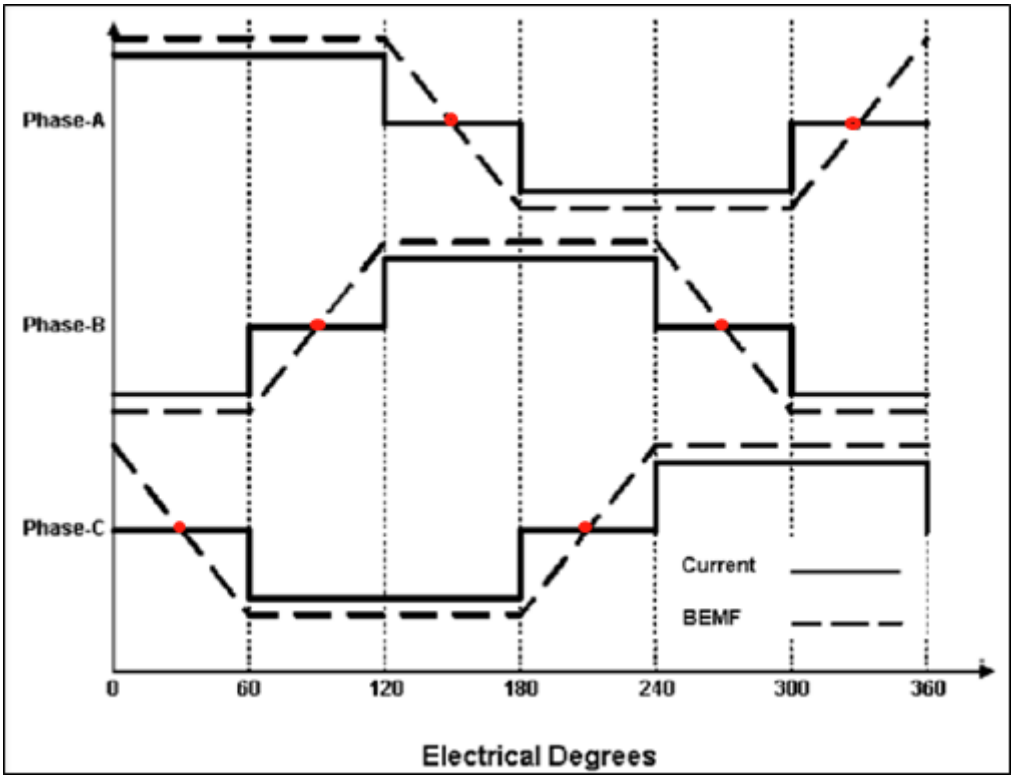


图 25.1.1.1 电流和反电动势波形

控制 BLDC 的关键就是确定换相的时刻。由上图可以看出，在每两个换相点的中间都对应一个反电动势的磁极改变的点，即反电点势从正变化为负或者从负变化为正的点，称为过零点。利用反电动势的这个特性，只要能够准确检测出反电动势的过零点，将其延迟 30°，即为需要换相的时刻。

25.1.2 反电动势的检测方法

从图 25.1.1.1 中可以看到每次的反电动势过零点都发生在不通电的那一相。例如图 25.1.1.1 中第一个 60° 内，A 相电流为正，B 相电流为负，C 相电流为零，这说明电机 AB 相通电，电流从 A 相流入 B 相，C 相为开路。反电动势的过零点正好出现在 C 相。而且由于 C 相不通电，没有电流，其相电压就与反电动势有直接的对应关系。

在众多检测转子的位置方法中，反电动势检测法最为成熟、应用最广泛的方法，该方法简单可靠、容易实现，但是这种方法也存在一些缺点：

1、 低速或转子静止时不适用，这是所有反电动势法的共同缺点。

2、 电压比较器对被检测信号中的毛刺、噪声非常敏感等等。

不过目前**反电动势过零点检测方法**依旧是当前无感控制中应用最为成熟的一种技术，我们例程同样也是使用的该技术实现的无感驱动。说了这么多，那我们应该如何检测过零点呢？又如何通过过零信号判断当前转子位置呢？接下来我们一步一步来解答。

25.1.3 比较器模式采样过零信号

由于 BLDC 电机的 Y 形连接，三相都接到公共的中性点，相电压无法直接测量。只能测量各相的端电压，通常将非通电绕组的端电压用于无传感器控制时，称为端电压法，即各相对地的电压，然后与中性点电压比较，当端电压从大于中性点电压变为小于中性点电压，或者从小于中性点电压变为大于中性点电压，即为过零点。如图所示。

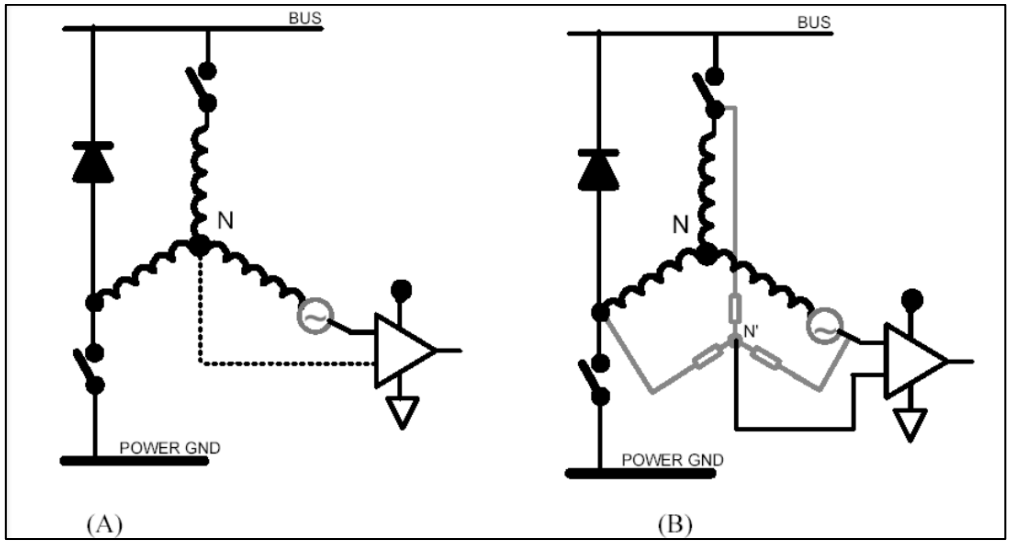


图 25.1.3.1.检测反电动势过零点

但是一般的 BLDC 电机都没有中性点的外接引线，所以无法直接测量中性点电压。解决这个问题最直接的办法就是重构一个"虚拟中性点”，通过将三相绕组分别通过阻值相等的电压连接到一个公共点而成，这个公共点就是虚拟中性点，如图 25.1.3.1(B)所示。将中性点信号和 UVW 信号通过比较器进行比较就可以获得过零信号。由于使用比较器进行外部硬件比较的方式较为稳定，因此得到广泛应用。

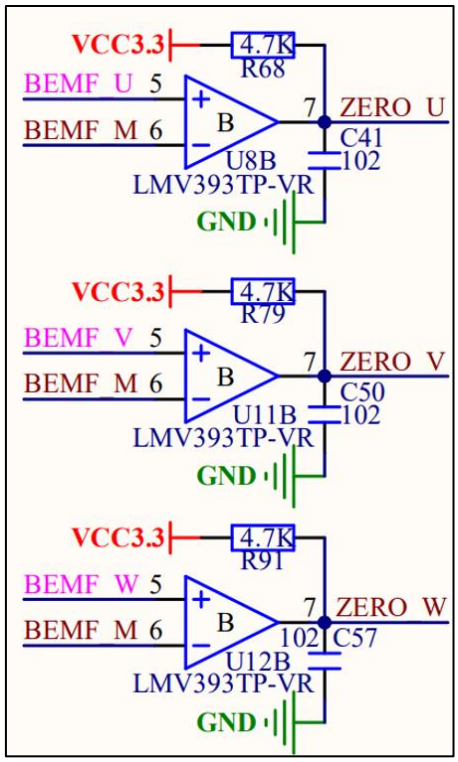


图 25.1.3.2 实际过零信号原理图

图 25.1.3.2 是我们无刷驱动板的比较器采样过零信号的电路，我们以 U 相来说明：U 相反向电动势和中性点信号通过 U8B 比较器输出过零点信号 ZERO_U。

25.1.4 闭环的建立

每一相的反电动势都有两种过零情况：从正变为负和从负变为正。三相共有六种过零情况对应六种换相状态，并且这种对应关系是固定不变的。所以我们可以将这个对应关系写入一个表中，程序中每检测到一个过零点，就通过查表来决定相应 IO 输出状态，控制下一步哪

两相通电；然后切换到当前断开相继续检测反电动势过零点，如此循环，直至建立稳定的闭环。下图为我们店铺无刷电机的过零信号组合所对应的绕组导通情况真值表：

方向	ZERO W	ZERO V	ZERO U	U+	U-	V+	V-	W+	W-	顺序
正转	0	0	1	打开	关闭	关闭	打开	关闭	关闭	↑
	0	1	1	打开	关闭	关闭	关闭	关闭	打开	↑
	0	1	0	关闭	关闭	打开	关闭	关闭	打开	↑
	1	1	0	关闭	打开	打开	关闭	关闭	关闭	↑
	1	0	0	关闭	打开	关闭	关闭	打开	关闭	↑
	1	0	1	关闭	关闭	关闭	打开	打开	关闭	↑
反转	0	1	0	关闭	打开	打开	关闭	关闭	关闭	↓
	1	1	0	关闭	打开	关闭	关闭	打开	关闭	↓
	1	0	0	关闭	关闭	关闭	打开	打开	关闭	↓
	1	0	1	打开	关闭	关闭	打开	关闭	关闭	↓
	0	0	1	打开	关闭	关闭	关闭	关闭	打开	↓
	0	1	1	关闭	关闭	打开	关闭	关闭	打开	↓

表 25.1.4.1 过零信组合对应的绕组导通情况

理论上，过零点总是超前换相点 30° 电角度。因此在检测到过零点后，要先延迟 30° 电角度再换相。但是在闭环调速过程中，电机旋转一个电气周期的时间不是固定不变的，因此无法预测在检测到过零点后接下来的这 30° 电角度是多长时间。那么在检测到过零点之后，怎样决定延时时间呢？

虽然无法预测接下来的 30° 电角度是多长，但是刚刚过去的上一个换相周期，即两个换相点之间 60° 电角度的长度是可以测量的。于是可以采用近似的办法，用上一个换相周期，即 60° 电角度的时间减半，作为接下来的 30° 电角度延时时间。这种办法是可行的，因为电机的转速是渐变的，相邻两个换相周期的时间相差不会很大。

由于定子绕组的反电动势与电机的转速成正比，所以电机在静止时反电动势为零或低速时反电动势很小，此时无法根据反电动势信号确定转子磁极的位置，因此反电动势法需要采用特殊起动技术，从静止开始加速，直至转速足够大，当反电势能检测到过零信号时，再切换至无刷直流电机运行状态。这个过程称为“三段式”起动，主要包括转子预定位、加速和运行状态切换三个阶段。这样既可以使电机转向可控,又可以保证电机达到一定转速后再进行切换，保证了起动的可靠性。下面就让我们介绍 **BLDC 方波启动技术**。

25.1.5 BLDC 方波启动技术

25.1.5.1 电机转子预定位

要保证无刷直流电机能够正常起动，首先要确定转子在静止时的位置。在轻载条件下，对于具有梯形反电势波形的无刷直流电机来说，一般采用磁制动转子定位方式。系统起动时，任意给定一组触发脉冲，在气隙中形成一个幅值恒定、方向不变的磁通，只要保证其幅值足够大，那么这一磁通就能在一定时间内将电机转子强行定位这个方向上。在应用中，可以在任意一组绕组上通电一定时间，其中预定位的 PWM 占空比和预定位时间的长短设定值可由具体电机特性和负载决定，在实际应用中调试而得。在预定位成功后，转子在起动前可达到预定的位置，为电机起动做好准备。

25.1.5.2 电机的外同步加速

确定了电机转子的初始位置后，由于此时定子绕组中的反电动势仍为零，所以必须人为的改变电机的外施电压和换相信号，使电机由静止逐步加速运动，这一过程称为外同步加速。对于不同的外施电压调整方法和换相信号调整方法，外同步加速可以划分为三类：换相信号频率不变，逐步增大外施电压使电机加速，称为恒频升压法。保持外施电压不变，逐渐增高换相信号的频率，使电机逐步加速,称为恒压升频法。在逐步增大外施电压的同时，增高换相的频率，称为**升频升压法**。

25.1.5.3 电机运行状态的转换

各个方法都有其优点和缺点。如升频升压法是人为地给电机施加一个由低频到高频不断加速的可控同步切换信号，而且电压也在不断地增加。通过调整电机换相频率，即可调整电机起动的速度。调整方法比较简单。但是这个过程比较难实现，切换信号的频率的选择要根据电机的极对数和其它参数来确定，太低电机无法加速，太高电机转速达不到会有噪声甚至无法启动，算法比较困难。无论哪种方法,该过程都是在未检测到反电动势信号时进行，因此对于控制系统来说，此段电机控制是盲区。参数在调试好的时候，可以快速切换至正常运行状态;而参数不理想时，电流可能不稳甚至电机会抖动。因此，在应用中，应根据电机及负载特性设定合理的升速曲线，并在尽可能短的时间内完成切换。

这一步是关键也是比较难实现的一步，有时软件或者硬件设计的不合理都可能导致起动失败。通常是采用估算的方式来选择切换速度。通过上面的分析可知，无位置传感器无刷直流电机控制系统的难点就是加速及切换阶段，当电机顺利起动后，就可以对电机调速操作。其中，无位置传感器无刷直流电机和有位置传感器电机调速原理一致。但由于无感三段式起动过程，转子位置检测无效，因此，对电机进行的速度 PID 闭环控制，需在电机起动顺利完成后进行。

至此，我们可以得到一个无感驱动的整体框图：

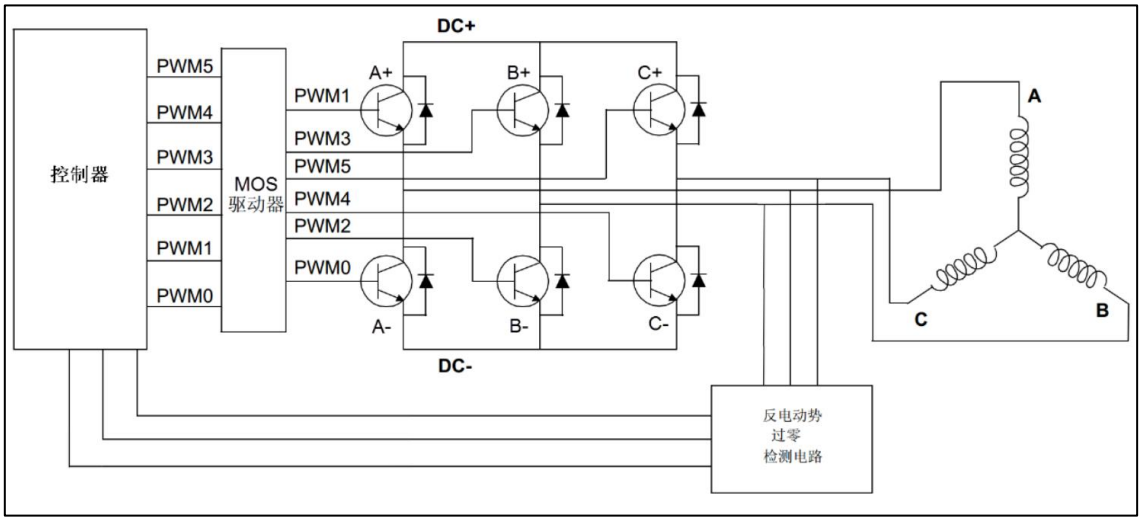


图 25.1.5.3.1 无感驱动的整体框图

25.2 硬件设计

1、例程功能

- 本实验使用无刷电机接口一连接无刷电机驱动板
- 当按下 KEY0 一次电机正转，按一次 KEY1 电机反转，按下 KEY2 则停止旋转。
- LED0 作为程序运行状态指示灯，闪烁周期 200ms。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器：
TIM1_CH1：PA8
TIM1_CH2：PA9
TIM1_CH3：PA10
IO：PB13\14\15
使能引脚：SHDN: PF10
4）过零信号检测引脚
过零 U 相 – PH10
过零 V 相 – PH11
过零 W 相 – PH12
5）无刷电机
6）无刷电机驱动板
7）12-60V 的 DC 电源

3、原理图

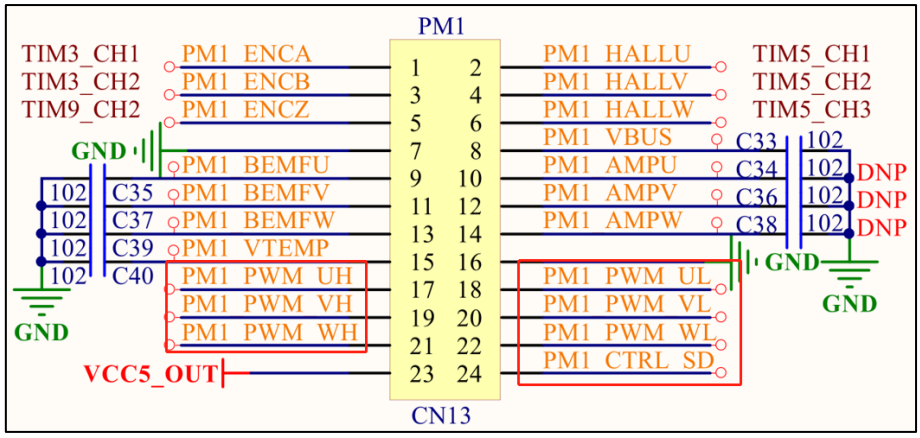


图 25.2.1 无刷电机接口 1 原理图

接口涉及的 IO 如下。

直流无刷驱动板	IO 口
驱动板 U 相	PA8、PB13
驱动板 V 相	PA9、PB14
驱动板 W 相	PA10、PB15
过零 U 相	PH10
过零 V 相	PH11
过零 W 相	PH12
输出使能	PF10

表 25.2.1 无刷相关 IO 口说明

驱动板和开发板连接只需要使用排线连接即可，电机与驱动板通过接线端子进行连接，实物连接如下。**另外需要将 JP3 跳线帽连接 H&Z 和 ZERO**。

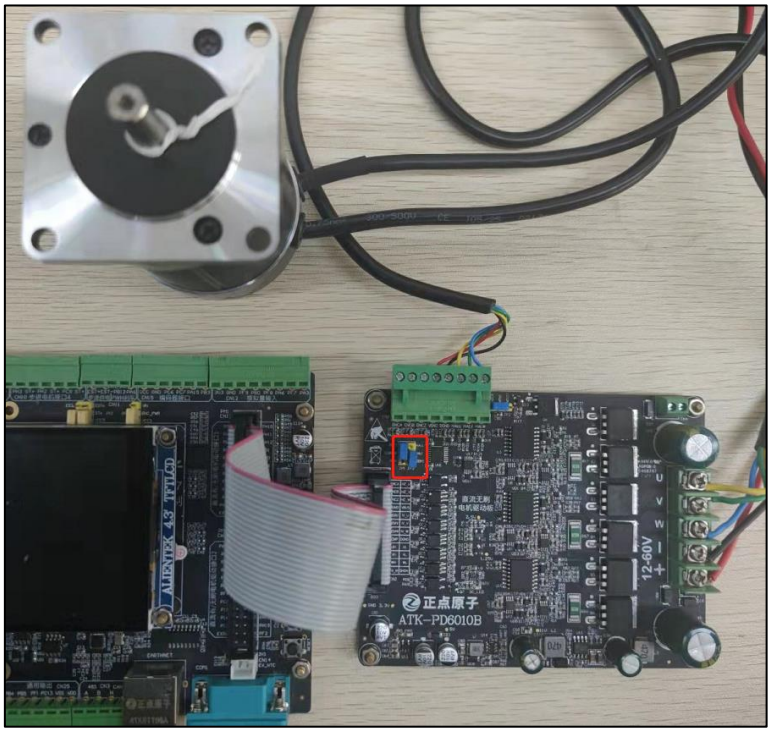


图 25.2.2 开发板&驱动板连接图

25.3 程序设计

25.3.1 无刷电机无感控制的配置步骤

1）实现电机基本驱动

实现电机的基础驱动函数，启停、6 步换向组合等等

2）预定位

固定转子在某一位置，然后以一定的速度按顺序换向旋转，实现方波启动

3）过零处理函数

过滤尖峰电压，判断过零信号发生改变，延迟计数，当延迟 30°之后开始换相

4）编写中断服务函数

定时调用过零控制函数

25.3.2 程序流程图

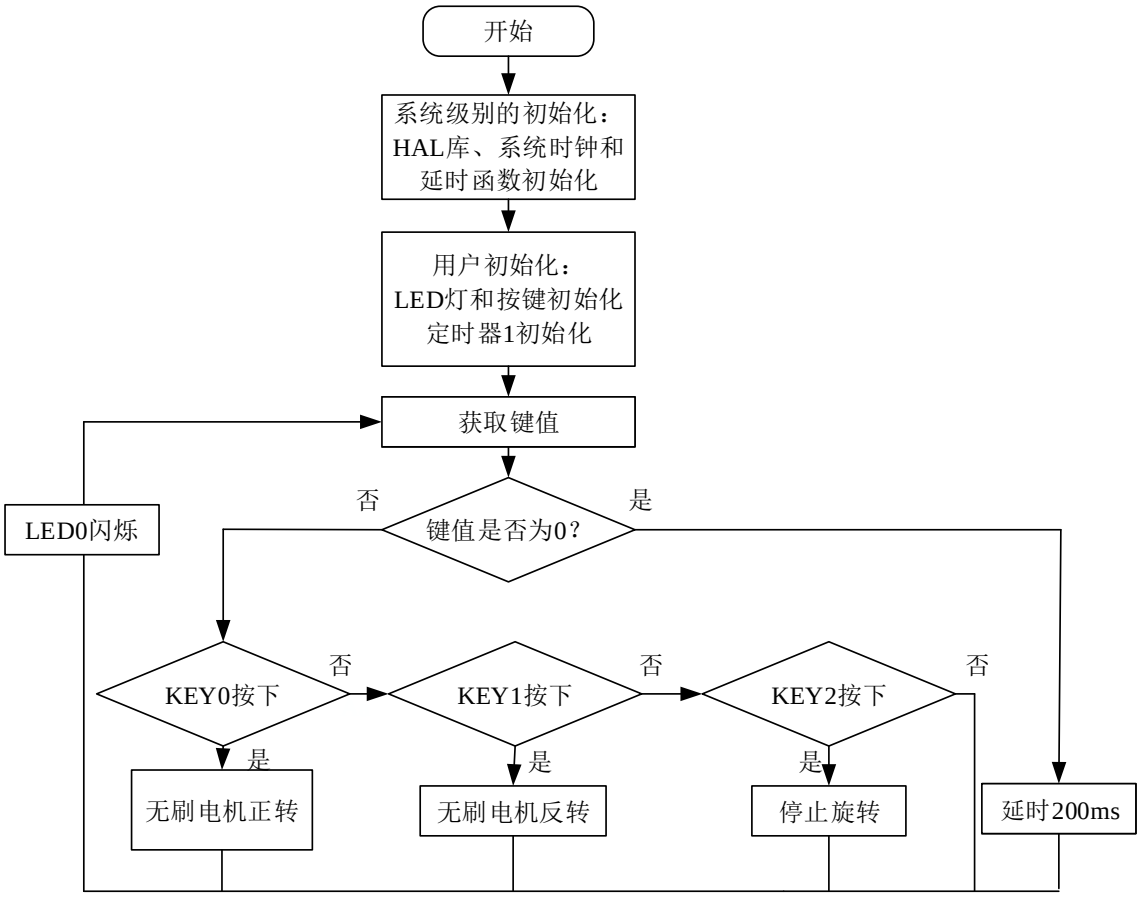


图 25.2.2.1 无刷电机基本控制流程图

25.3.3 程序解析

这里我们只讲解关于无感驱动的核心代码，首先是过零控制函数，在源文件 zero_ctr.c 中，内容如下：

```
/**
 * @brief       实现定位并以一定速度旋转之后，切换至过零控制
 * @param       无
 * @retval      无
 */
void zero_ctr_loop(void)
{
    if(g_bldc_motor1.run_flag == RUN)
    {
        switch(g_zero_ctr_status)
        {
            case 0:                             /* 定位：固定一相*/
            {
                /* 启动速度不可太快，否则容易过流 */
                vvvf_sp.Voilage_Ref = (uint16_t)(MAX_PWM_DUTY/20);
                g_bldc_motor1.pwm_duty = vvvf_sp.Voilage_Ref;
                m1_uhvl();                      /* U+V- */
                delay_temp = 0;
                vvvf_sp.Freq_T_Ref = 1100;      /* 初始延迟时间为：55*1100us */
                g_zero_ctr_status = 1;          /* 切换至延迟计数 */
                count_i = 0;
                switch_hallless = 0;
                vvvf_sp.Count = 0;
```

```
            }
            break;
            case 1:/*定时延迟*/
            {
                delay_temp++;
                if(delay_temp >= vvvf_sp.Freq_T_Ref)
                {
                    delay_temp = 0;
                    g_zero_ctr_status = 2;     /* 切换至换相操作 */
                }
#if (DRIVE_MODE)                               /* 过零闭环控制 为 0 开环控制 */
                switch_hallless = hallless_sw();/* 检测过零信号 稳定后切换至闭环 */

                if (switch_hallless == 1)
                {
                    g_zero_ctr_status = 3;
                }
#endif
            }
            break;
            case 2:                            /* 加速换相（提高换向频率）*/
            {
                vvvf_sp.Freq_T_Ref -= vvvf_sp.Freq_T_Ref/15+1;/*改变换向频率 */
                vvvf_sp.Count++;
                change_voltage();              /* 改变换向所需电压 */
                if(vvvf_sp.Freq_T_Ref < 180)
                {
                    vvvf_sp.Freq_T_Ref = 180; /* 固定次频率（延迟时间） */
                    g_zero_ctr_status = 1;     /* 切换至延迟计数 */
                }
                else
                {
                    g_zero_ctr_status = 1;     /* 切换至延迟计数 */
                }
                vvvf_sp.VvvF_Count++;          /* 换向计数 */

                if(vvvf_sp.VvvF_Count == 6)
                {
                    vvvf_sp.VvvF_Count = 0;
                }
                anwerfen_sw();
            }
            break;
            case 3:
            {
                hallless_sw();                 /* 切换至闭环过零检测状态 */
            }
            break;
            default: break;
        }
    }
    else if(g_bldc_motor1.run_flag == STOP)
    {
        zero_ctr_init();
    }
}
```

首先初始状态标志位 g_zero_ctr_status 为 0，此时使能旋转的话会进入 case 0，这时程序默认给 UV 相通电，实现转子定位，然后下一次进入时切换到 case 1，经过延迟后切换到 case 2，在这里将变量 vvvf_sp.Freq_T_Ref 递减，实现增频，通过函数 change_voltage 实现增压，即实现了**升频升压法。**之后逐渐加速，再回到状态 1 进行延迟，直至检测到稳定的过零信号，最后切换到状态 3，调用函数 hallless_sw 实现过零控制。接着看下函数 hallless_sw 的内容：

```
uint8_t hallless_sw(void)
{
```

```
count_j++;/* 运行过程时间计数 18K 的中断频率为单位 */
/* 过零信号滤波 */
hallless_three.Queue_Status[0] = hallless_three.Queue_Status[0] << 1;
hallless_three.Queue_Status[1] = hallless_three.Queue_Status[1] << 1;
hallless_three.Queue_Status[2] = hallless_three.Queue_Status[2] << 1;
/* 检测过零点 */
hallless_three.Queue_Status[0] |= HAL_GPIO_ReadPin(HALL1_TIM_CH1_GPIO,
HALL1_TIM_CH1_PIN); /* U 相过零点 */
hallless_three.Queue_Status[1] |= HAL_GPIO_ReadPin(HALL1_TIM_CH2_GPIO,
HALL1_TIM_CH2_PIN); /* V 相过零点 */
hallless_three.Queue_Status[2] |= HAL_GPIO_ReadPin(HALL1_TIM_CH3_GPIO,
HALL1_TIM_CH3_PIN); /* W 相过零点 */
/* 过零信号 U */
hallless_three.Filter_Math = hallless_three.Queue_Status[0] & FILTER_LONG;
if (hallless_three.Filter_Math == FILTER_LONG)
{
    hallless_three.QFilter_Status[0] = 1;

}
else if (hallless_three.Filter_Math == 0x00)
{
    hallless_three.QFilter_Status[0] = 0;
}
else
{
    hallless_three.Filter_Count++;
    return 0;
}
/* 过零信号 V */
hallless_three.Filter_Math = hallless_three.Queue_Status[1] & FILTER_LONG;
if (hallless_three.Filter_Math == FILTER_LONG)
{
    hallless_three.QFilter_Status[1] = 1;
}
else if (hallless_three.Filter_Math == 0x00)
{
    hallless_three.QFilter_Status[1] = 0;
}
else
{
    hallless_three.Filter_Count++;
    return 0;
}
/* 过零信号 W */
hallless_three.Filter_Math = hallless_three.Queue_Status[2] & FILTER_LONG;
if (hallless_three.Filter_Math == FILTER_LONG)
{
    hallless_three.QFilter_Status[2] = 1;
}
else if (hallless_three.Filter_Math == 0x00)
{
    hallless_three.QFilter_Status[2] = 0;
}
else
{
    hallless_three.Filter_Count++;
    return 0;
}
/****************** 速度计算 ******************/
hallless_three.Filter_Edge = uemf_edge(hallless_three.QFilter_Status[0]);

if (hallless_three.Filter_Edge == 0) /* 统计过零信号的高电平时间 */
{
    /* 延迟 30°换相，因为硬件上低通滤波器和软件延迟的原因，实际延迟角度需小于 30°
```

```
        最优解可以通过示波器确定。通过 PE1 查看实际延迟角度 */
        if (count_i >= 4)  /* 稳定检测到过零信号才对速度进行测量 */
        {
            if (g_bldc_motor1.dir == CCW)
                hallless_three.Speed_RPM = ((uint32_t)SPEED_COEFF / count_j);
            else
                hallless_three.Speed_RPM = -((uint32_t)SPEED_COEFF / count_j);

            FirstOrderRC_LPF(g_bldc_motor1.speed, hallless_three.Speed_RPM,
0.2379);
        }

        hallless_three.Filter_Delay = count_j / 6;  /* 30°所需时间 */
        hallless_three.Filter_Count = 0;
        count_j = 0;
        count_i++;                                  /* 捕捉过零点，累计次数 */
    }

    if (hallless_three.Filter_Edge == 1)
    {
        hallless_three.Filter_Count = 0;
        count_j = 0;
    }

    if (hallless_three.Filter_Edge == 2)    /* 过零信号没变化 */
    {
        hallless_three.Filter_Count++;      /* 不换相时间累计 超时则判定速度为 0 */

        if (hallless_three.Filter_Count > 15000)
        {
            hallless_three.Filter_Count = 0;
            hallless_three.Speed_RPMF = 0;   /* 超时换向 判定为停止 速度为 0 */
        }
    }
    /********************* 过零控制 ************************/
    if (count_i >= 4)                       /* 稳定检测到过零信号并旋转 2 圈之后进入*/
    {
        count_i = 4;
        edge_flag++;
        if (edge_flag >= 2)
        {
            edge_flag = 2;
            hallless_three.Hallless_State = hallless_three.QFilter_Status[0] |
              (hallless_three.QFilter_Status[1] << 1) |
              (hallless_three.QFilter_Status[2] << 2);   /* 获取过零信号组合 */
           /* 检测到过零信号发生改变，但不立即换向 开始过零计时*/
            if (hallless_three.Hallless_State !=
              hallless_three.OldHallless_State)
            {
                hallless_three.Filter_Count_All++;
            }
            /************ 调试延迟 30°的时间之后，执行换相 **************/
            if (hallless_three.Filter_Count_All >=
              (hallless_three.Filter_Delay))       /* 等延迟 30°时间之后 在进行换相 */
            {
                hallless_three.Filter_Count_All = 0;
            /* 在次确认是否检测到过零信号发生改变，是的话立即换相 */
                if (hallless_three.Hallless_State !=
                  hallless_three.OldHallless_State)
                {
                    if (g_bldc_motor1.dir == CCW)
                    {
                        g_bldc_motor1.pos--;        /* 电机位置计数 */
                       /* 根据过零信号组合值，导通对应相的上下桥臂 */
```

```
                    pfunclist_m1[hallless_three.Hallless_State - 1]();
                }
                else
                {
                    g_bldc_motor1.pos++;    /* 电机位置计数 */
                    /* 根据过零信号组合值，导通对应相的上下桥臂 */
                    pfunclist_m2[hallless_three.Hallless_State - 1]();
                }
            }
            hallless_three.OldHallless_State =
            hallless_three.Hallless_State ;
        }
    }
    return 1;
  }
  else
    return 0;
}
```

该函数实现了过零信号控制，首先将采集到的过零信号进行滤波，防止换向产生的尖峰电压带来的干扰，接着通过读取过零信号组合值进行换向控制，注意读取到过零信号组合值时不是立即换相，而是需要延迟 30° 电角度的时间在进行换相。这样就可以实现电机的无感过零控制了。

在中断回调函数里只需定时调用过零控制函数即可，中断回调函数如下：

```
/**
 * @brief       定时器中断回调
 * @param       无
 * @retval      无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim->Instance == ATIM_TIMX_PWM)/* 55us */
    {
#ifdef H_PWM_L_ON
        zero_ctr_loop();
#endif
    }
}
```

主函数内容同六步换相类似，通过按键控制无刷电机的正反转及停止，具体函数内容如下：

```
int main(void)
{
    uint8_t key,t;
    char buf[32];

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    bldc_init(168000/18-1,0);
    bldc_ctrl(MOTOR_1,CW,1000);                 /* 初始无刷电机接口 1 速度 */

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"BLDC Motor Test",g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:Step++",g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:Step--",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:Stop",g_point_color);

    while (1)
```

```
    {
        t++;
        if(t % 20 == 0)
        {
            sprintf(buf,"PWM_Duty:%.1f%%",(float)((g_bldc_motor1.pwm_duty/
                    MAX_PWM_DUTY)*100)); /* 显示控制 PWM 占空比 */
            lcd_show_string(10,110,200,16,16,buf,g_point_color);
            LED0_TOGGLE();               /* LED0(红灯) 翻转 */
        }

        key = key_scan(0);
        if(key == KEY0_PRES)             /* 按下 KEY0 开启电机 */
        {
            g_bldc_motor1.dir=CCW;
            g_bldc_motor1.run_flag=RUN;  /* 开启运行 */
            start_motor1();              /* 开启运行 */
        }
        if(key == KEY1_PRES)             /* 按下 KEY1 开启电机 */
        {
            g_bldc_motor1.dir=CW;
            g_bldc_motor1.run_flag=RUN;  /* 开启运行 */
            start_motor1();              /* 运行电机 */
        }
        if(key == KEY2_PRES)             /* 按下 KEY2 关闭电机 */
        {
            stop_motor1();               /* 停机 */
            g_bldc_motor1.run_flag=STOP; /* 标记停机 */
            g_zero_ctr_status=0;
            g_bldc_motor1.pwm_duty=0;
        }
        delay_ms(10);
    }
}
```

25.4 下载验证

硬件连接好之后下载程序到开发板，点击按键 KEY0 可以看到电机开始正向旋转，点击一次按键 KEY1 则可以看到电机开始反向旋转，点击按键 KEY2 则可以随时停止无刷电机。大家可以自己下载程序并点击按键测试效果。

第 26 章 无刷电机无感方波闭环控制

前面我们已经学会了无感方波的基本控制，本章将使用 PID 来实现无刷电机无感控制的**速度闭环控制**和**速度+电流双闭环控制**。

本章分为如下几个小节：

26.1 无感速度闭环控制

26.2 无感速度&电流双闭环控制

26.1 无感速度闭环控制

要实现速度闭环，首先需要先了解下无刷无感驱动的测速原理，这样才可以知道当前转速是否符合我们的设定值。

26.1.1 无刷无感测速原理

无感测速原理与有感类似，有感通过霍尔信号测速，无感通过过零信号测速，两者输出的波形一致。当转子只有一对级时，电机旋转一圈，每一相都会出现两次过零点，只需检测其中一相过零信号高电平持续时间，即可求出旋转一圈所需时间。详情可以回顾下 24.1.1 小节。

26.1.2 硬件设计

1、例程功能

- 本实验以电机开发板的直流有刷/无刷电机驱动接口 1 为例。
- 当按键 0 按下，就增大目标速度值；当按键 1 按下，就减小目标速度值。目标速度的绝对值大小决定电机的速度，它的正负决定电机的正反转方向。按下按键 2 则马上停止电机。
- 屏幕显示按键功能、占空比、目标转速以及实际转速以及显示无刷电机电压、电流和温度、速度等等信息。
- 可通过串口 1 即 USB_TTL 接口连接正点原子 PID 调试助手，查看 PID 波形。
- LED0 闪烁指示程序运行。

2、硬件资源

1）LED 灯：LED0 – PE0

2）独立按键

KEY0 – PE2

KEY1 – PE3

KEY2 – PE4

3）定时器：

TIM1_CH1：PA8

TIM1_CH2：PA9

TIM1_CH3：PA10

IO：PB13\14\15

使能引脚：SHDN: PF10

4）过零信号检测引脚

过零 U 相 – PH10

过零 V 相 – PH11

过零 W 相 – PH12

5）无刷电机

6）无刷电机驱动板

7）12-60V 的 DC 电源
8）ADC：ADC1_CH9：PB1
ADC1_CH0：PA0
ADC1_CH8：PB0
ADC1_CH6：PA6
ADC1_CH3：PA3

3、原理图

原理图部分和开环控制完全一致，这里就不重复叙述了，可以参考之前的章节。

26.1.3 程序设计

26.1.3.1 无刷电机无感速度闭环的配置步骤

1）电机基本驱动
实现电机的基础驱动函数，启停、6 步换向组合、过零控制等等
2）初始化 ADC（保留）
初始化 ADC 通道的 IO，设置 ADC 工作方式、DMA 等等
3）PID 闭环控制实现
实现 PID 的初始参数赋值、PID 计算等等
4）上位机通信协议
编写上位机通信代码，可在上位机上实时显示当前速度与目标速度的波形变化
5）编写中断服务函数
PWM 中断用于无感驱动状态切换、换向控制、堵转检测、PID 闭环控制等等

26.1.3.2 程序流程图

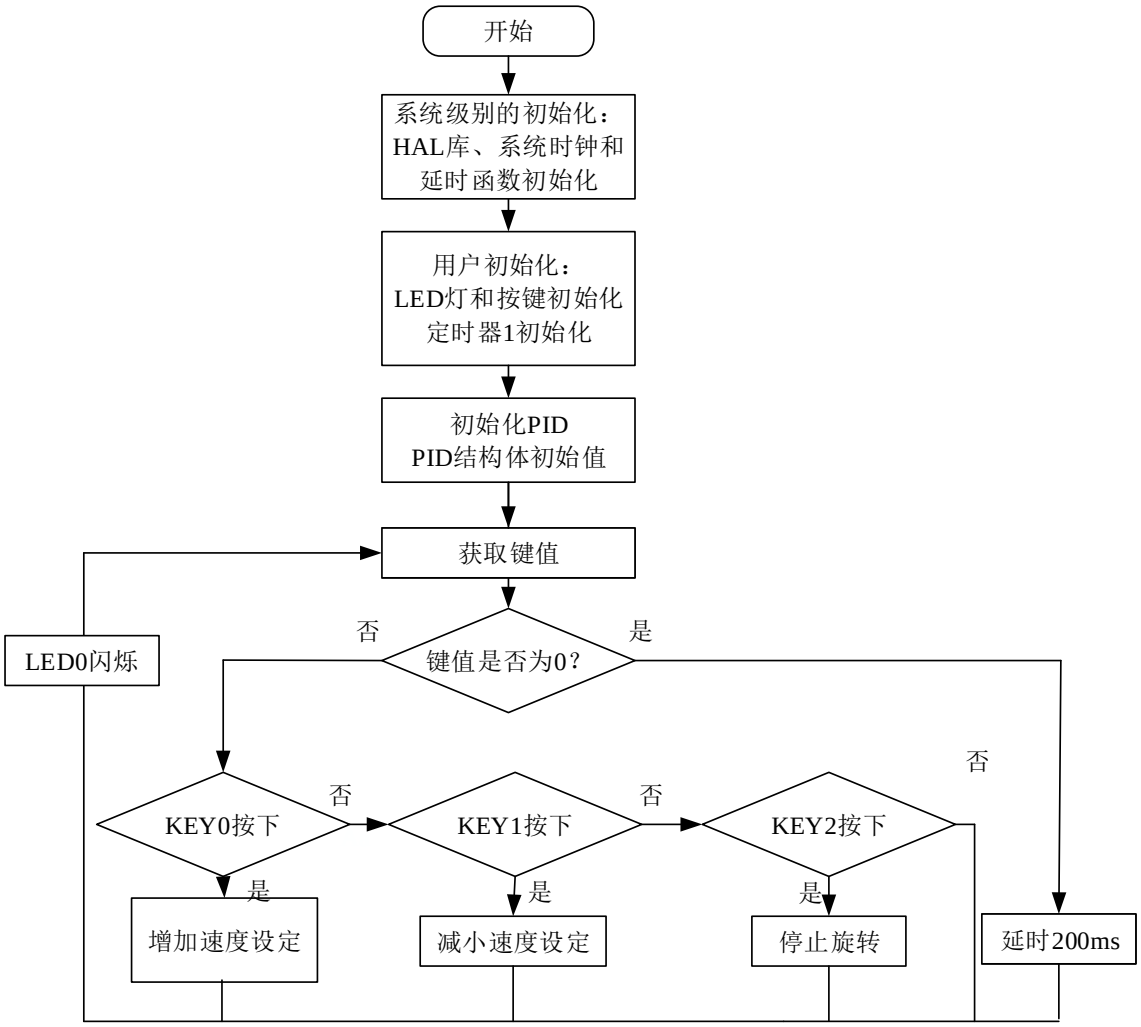


图 26.1.3.2.1 无刷电机无感速度闭环控制流程图

26.1.3.3 程序解析

这里我们只讲解核心代码，使用了 PID 闭环控制，因此需要 PID 部分的程序，计算过程一致，这里仅列出无感闭环的 PID 部分参数，如下。

```
/*定义 PID 参数相关宏*/
#define  KP      0.05000f           /* P 参数*/
#define  KI      0.00025f           /* I 参数*/
#define  KD      0.000150f          /* D 参数*/
#define SMAPLSE_PID_SPEED  40       /* 采样率 单位 ms*/
```

接着是定时器中断回调函数，在里边实现速度计算以及过零控制如下。

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    uint8_t i;
    static uint8_t times_count = 0;          /* 定时器时间记录 */
    if(htim->Instance == ATIM_TIMX_PWM)      /* 55us */
    {
        if(g_bldc_motor1.run_flag == RUN)
        {
            /******************* 三相电流采集 *******************/
            for(i=0; i<3; i++)
            {
                adc_val_m1[i] = g_adc_val[i+2];
```

```
            adc_amp[i] = adc_val_m1[i] -
              adc_amp_offset[i][ADC_AMP_OFFSET_TIMES];
            if(adc_amp[i] < 0)
                adc_amp_un[i] = 0;              /* 反电动势电压为悬空绕组直接清 0  */
            else if(adc_amp[i] >= 0)            /* 去除反电动势引起的负电流数据 */
                adc_amp_un[i] = adc_amp[i];
        }
        /*运算母线电流（母线电流为任意两个有开关动作的相电流之和）*/
        adc_amp_bus = (adc_amp_un[0] + adc_amp_un[1] +
          adc_amp_un[2])*ADC2CURT;
    }
#ifdef H_PWM_L_ON
    /* 过零控制 */
    zero_ctr_loop();
    /*************** 速度环 PID 控制 *****************/
    /* 具有一定速度（有速度测量说明已经进入过零闭环状态）后才能进入 PID 闭环控制 */
    if(g_bldc_motor1.run_flag == RUN && g_zero_ctr_status == 3)
    {
            temp_pwm1 = increment_pid_ctrl(&g_speed_pid,g_bldc_motor1.speed);
            FirstOrderRC_LPF(motor_pwm_s,temp_pwm1,0.085);
            /* 最低速度限制 */
            if(motor_pwm_s < 0)
            {
                /* 加速启动速度 */
                if(motor_pwm_s >= -600)
                    motor_pwm_s = -600;
                g_bldc_motor1.pwm_duty = -motor_pwm_s;
            }
            else
            {
                if(motor_pwm_s <= 600)
                    motor_pwm_s = 600;
                g_bldc_motor1.pwm_duty = motor_pwm_s;
            }
    }

#endif
    }
    if(htim->Instance == TIM6)
    {
        /******************基准电压测量 ********************/
        times_count++;
        if(g_bldc_motor1.run_flag == STOP)
        {
            uint8_t i;
            uint32_t avg[3] = {0,0,0};
            adc_amp_offset[0][adc_amp_offset_p] = g_adc_val[2];
            adc_amp_offset[1][adc_amp_offset_p] = g_adc_val[3];
            adc_amp_offset[2][adc_amp_offset_p] = g_adc_val[4];
            adc_amp_offset_p ++;
            NUM_CLEAR(adc_amp_offset_p,ADC_AMP_OFFSET_TIMES);
            for(i=0; i<ADC_AMP_OFFSET_TIMES; i++)
            {
                avg[0] += adc_amp_offset[0][i];
                avg[1] += adc_amp_offset[1][i];
                avg[2] += adc_amp_offset[2][i];
            }
            for(i=0; i<3; i++)
            {
                avg[i] /= ADC_AMP_OFFSET_TIMES;
                adc_amp_offset[i][ADC_AMP_OFFSET_TIMES] = avg[i];
            }
        }
        if(times_count == SMAPLSE_PID_SPEED)
        {
```

```
#if (LOCK_TAC==2)
          /********************* 堵塞处理 ******************/
          if(g_bldc_motor1.locked_rotor==1)     /* 堵塞 */
          {
              clc++;
              if(clc > 50)                      /* 延迟 2s 后重新启动*/
              {
                  clc = 0;
                  pid_init();
                  stop_motor1();
                  g_speed_pid.SetPoint = 400.0; /* 400PRM */
                  g_bldc_motor1.dir = CW;       /* 初始方向正转 */
                  g_bldc_motor1.pwm_duty = 600; /* 加速启动速度 */
                  g_bldc_motor1.run_flag = RUN; /* 开启运行 */
                  start_motor1();               /* 运行电机 */
                  g_bldc_motor1.locked_rotor = 0;
                  g_zero_ctr_status = 0;        /* 堵塞状态需要重新定位初始位置 */
              }
          }
#endif
          times_count=0;
      }
  }
}
```

在中断回调函数里边同样会调用过零控制函数 zero_ctr_loop，该函数内容与上节课完全一致，这里就不多赘述。主要来看下 PID 的控制环节，首先通过无感控制状态判断决定是否打开 PID 控制，当过零信号稳定就会进入 PID 控制，首先通过过零信号计算得出的电机转速，接着将电机转速进行滤波，然后带入 PID 计算，经过限幅将 PWM 带入到下一次中断时启用。这里我们还在中断中加上了堵转处理，当电机因意外或过流导致停机时，当堵转时间超过 2s，此时会重新初始化 PID 并以 400RPM 的初始速度重新开启电机旋转。

main 函数主要是通过按键设定闭环的目标速度值，内容如下。

```
int main(void)
{
    uint8_t key,t;
    char buf[32];
    int16_t pwm_duty_temp=0;
    float current[3]= {0.0f};
    float current_lpf[4]= {0.0f};
    uint16_t strar_sf=0;/* 启动速度标识 */
    uint8_t display_motor_offest=0;

    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
    delay_init(168);                            /* 延时初始化 */
    usart_init(115200);                         /* 串口初始化为 115200 */
    led_init();                                 /* 初始化 LED */
    key_init();                                 /* 初始化按键 */
    lcd_init();                                 /* 初始化 LCD */
    bldc_init(168000/18-1,0);
    bldc_ctrl(MOTOR_1,CW,1000);                 /* 初始无刷电机接口 1 速度 */
    adc_nch_dma_init();
    pid_init();

    g_point_color = WHITE;
    g_back_color  = BLACK;
    lcd_show_string(10,10,200,16,16,"BLDC Motor Test",g_point_color);
    lcd_show_string(10,30,200,16,16,"KEY0:Step++",g_point_color);
    lcd_show_string(10,50,200,16,16,"KEY1:Step--",g_point_color);
    lcd_show_string(10,70,200,16,16,"KEY2:Stop",g_point_color);

    while (1)
    {
```

```
        t++;
        if(t % 20 == 0)
        {
            sprintf(buf,"SetSpeed:%4d   ",(int16_t)(*User_SetPoint));
            /* 显示设置速度 */
            lcd_show_string(10,110+display_motor_offest,200,16,16,buf,
                            g_point_color);
            sprintf(buf,"M1 speed:%4d   ",g_bldc_motor1.speed);  /* 显示转速 */
            lcd_show_string(10,130+display_motor_offest,200,16,16,buf,
                            g_point_color);
            sprintf(buf,"M1 pos:%4d ",g_bldc_motor1.pos);        /* 显示测量速度 */
            lcd_show_string(10,150+display_motor_offest,200,16,16,buf,
                             g_point_color);
            sprintf(buf,"PWM_Duty:%.1f%%",(float)((g_bldc_motor1.pwm_duty/
                    (float)MAX_PWM_DUTY)*100));   /* 显示控制 PWM 占空比 */
            lcd_show_string(10,170+display_motor_offest,200,16,16,buf,
                            g_point_color);
            LED0_TOGGLE();                          /* LED0(红灯) 翻转 */

            if(g_bldc_motor1.run_flag==STOP)      /* 停机的电流显示 */
            {
                current_lpf[0]=0;
                current_lpf[1]=0;
                current_lpf[2]=0;
            }
            current[0]=adc_amp_un[0]* ADC2CURT;/* U */
            current[1]=adc_amp_un[1]* ADC2CURT;/* V */
            current[2]=adc_amp_un[2]* ADC2CURT;/* W */

            /* 一阶数字滤波 滤波系数 0.1 用于显示 */
            FirstOrderRC_LPF(current_lpf[0],current[0],0.1f);
            FirstOrderRC_LPF(current_lpf[1],current[1],0.1f);
            FirstOrderRC_LPF(current_lpf[2],current[2],0.1f);
            FirstOrderRC_LPF(current_lpf[3],adc_amp_bus,0.1f);
            sprintf(buf,"Amp U:%.3fmA ",(float)current_lpf[0]);
            lcd_show_string(10,230,200,16,16,buf,g_point_color);
            sprintf(buf,"Amp V:%.3fmA ",(float)current_lpf[1]);
            lcd_show_string(10,250,200,16,16,buf,g_point_color);
            sprintf(buf,"Amp W:%.3fmA ",(float)current_lpf[2]);
            lcd_show_string(10,270,200,16,16,buf,g_point_color);
            sprintf(buf,"Amp Bus:%.3fmA ",(float)adc_amp_bus);
            lcd_show_string(10,290,200,16,16,buf,g_point_color);
        }

        key = key_scan(0);
        if(key == KEY0_PRES)
        {
            step_up();
        }
        if(key == KEY1_PRES)                /* 按下 KEY1 开启电机 */
        {
            step_down();
        }
        if(key == KEY2_PRES)                /* 按下 KEY2 关闭电机 */
        {
            stop_motor1();                  /* 停机 */
            g_bldc_motor1.run_flag=STOP; /* 标记停机 */
            g_bldc_motor1.pwm_duty=0;
            pid_init();                     /* 初始化 PID */
            clc=0;                          /* 清零等待时间 */
        }
        delay_ms(10);
    }
}
```

```
void step_up(void)
{
    g_bldc_motor1.run_flag=RUN; /* 开启运行 */
    start_motor1();             /* 开启运行 */
    if(g_bldc_motor1.dir==CW&&*User_SetPoint==0)
    {
        g_bldc_motor1.dir=CCW;
    }
    *user_setpoint+=400;        /* 逆时针旋转下递增 */
    if(*user_setpoint==0)
    {
        stop_motor1();          /* 开启运行 */
        g_bldc_motor1.run_flag=STOP;/* 标记停机 */
        g_bldc_motor1.pwm_duty=0;
        pid_init();             /* 初始化 PID */
        g_bldc_motor1.speed=0;
        motor_pwm_s=0;
    }
    if(*user_setpoint>=3200)    /* 最高不超过 3200prm */
    {
        *user_setpoint=3200;
    }
    debug_data_temp=*user_setpoint;
}
void step_down(void)
{
    g_bldc_motor1.run_flag=RUN; /* 开启运行 */
    start_motor1();             /* 开启运行 */
    if(g_bldc_motor1.dir==CCW&&*user_setpoint==0)
    {
        g_bldc_motor1.dir=CW;
    }
    *user_setpoint-=400;        /* 逆时针旋转下递增 */
    if(*user_setpoint==0)
    {
        stop_motor1();          /* 开启运行 */
        g_bldc_motor1.run_flag=STOP;/* 标记停机 */
        g_bldc_motor1.pwm_duty=0;

        pid_init();             /* 初始化 PID */
        g_bldc_motor1.speed=0;
        motor_pwm_s=0;
    }
    if(*user_setpoint<=-3200)   /* 最高不超过 3200prm */
    {
        *user_setpoint=-3200;
    }
    debug_data_temp=*user_setpoint;
}
```

无感闭环切换方向时还需跳出 PID 过程，重新建立稳定的无感旋转过程，然后再次进入 PID 控制，因此按键控制逻辑相对多一点，并且 PID 还需要重新初始化，防止变量没有清零造成失控的现象。

26.1.4 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 上显示按键功能、占空比以及电机速度信息和显示程序计算的电压、电流等信息，当我们按下 KEY0，目标速度将增大；按下 KEY1，目标速度将减小；按下 KEY2，电机将停止。目标速度为正数时，电机正转，反之电机反转。我们再打开 PID 调试助手（注意接上串口 1 的 USB_TTL 连接至电脑），选择对应的串口端口，接着勾选通道 1 和 2，点击“开始”按钮，即可开始显示波形，如下图 26.1.4.1，大家可以自己下载程序并点击按键测试效果。

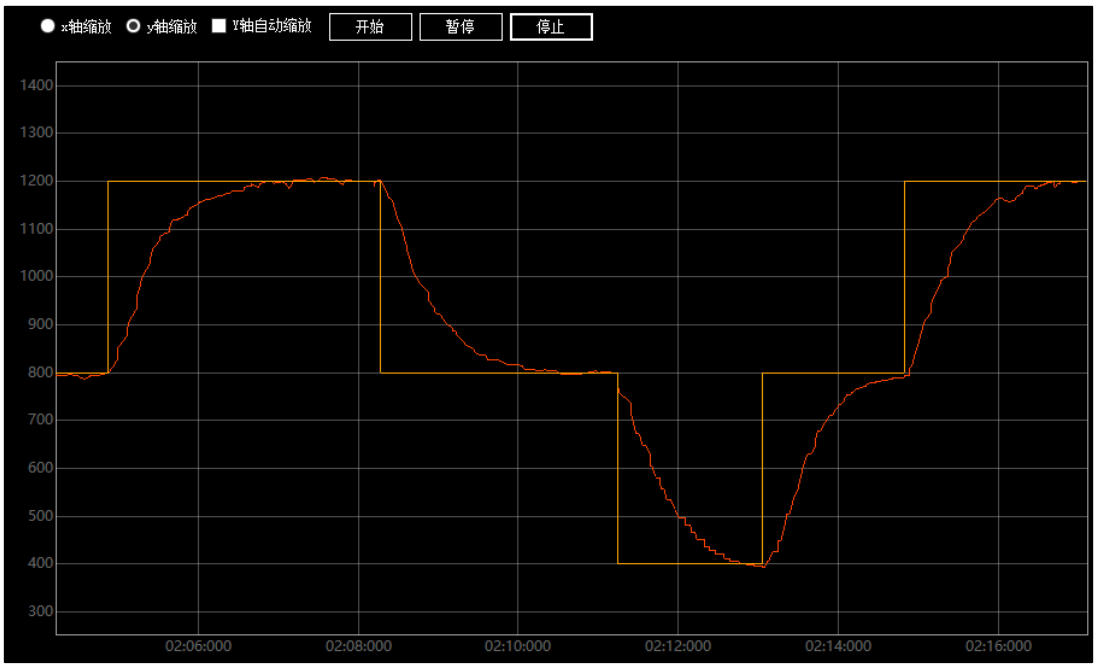


图 26.1.4.1 PID 调节效果

图 26.1.4.1 中，橙线代表目标速度，红线代表实际速度，当我们按下 KEY0，目标速度增大，橙线先发生变化，而红线（实际速度）会逐渐靠近橙线（目标速度）；按下 KEY1，目标速度将减小，曲线的变化同理；按下 KEY2，电机将停止，目标速度将为 0。

注意：1、本实验需要使用 USB 数据线连接开发板的串口 1 到电脑，并启动电机之后才会有波形变化；2、如果发现波形不对，请检查电机接线；3、PID 系数并不是通用的，如果 PID 曲线不理想，大家需要根据自己的实际系统去调节

26.2 无感速度+电流双闭环控制

要实现无感的速度+电流双闭环控制，只需在速度环的基础上加上电流环部分即可。三相电流的采集已经在前面的课程（第 23 章）学习过了，所以下面我们直接进入代码实现。

26.2.1 硬件设计

1、例程功能

- 本实验以电机开发板的直流有刷/无刷电机驱动接口 1 为例。
- 通过 PID 电流环的初始化设置转矩大小，即目标电流大小。目标电流越大，转矩越大。
- 当按键 0 按下，就增大目标速度值；当按键 1 按下，就减小目标速度值。目标速度的绝对值大小决定电机的速度，它的正负决定电机的正反转方向。按下按键 2 则马上停止电机。
- 屏幕显示按键功能、占空比、目标转速以及实际转速以及显示无刷电机电压、电流和温度、速度等等信息。
- 可通过串口 1 即 USB_TTL 接口连接正点原子 PID 调试助手，查看 PID 波形。
- LED0 闪烁指示程序运行。

2、硬件资源

1）LED 灯：LED0 – PE0
2）独立按键
KEY0 – PE2
KEY1 – PE3
KEY2 – PE4
3）定时器：
TIM1_CH1：PA8
TIM1_CH2：PA9
TIM1_CH3：PA10
IO：PB13\14\15
使能引脚：SHDN: PF10
4）过零信号检测引脚
过零 U 相 – PH10
过零 V 相 – PH11
过零 W 相 – PH12
5）无刷电机
6）无刷电机驱动板
7）12-60V 的 DC 电源
8）ADC：ADC1_CH9：PB1
ADC1_CH0：PA0
ADC1_CH8：PB0
ADC1_CH6：PA6
ADC1_CH3：PA3

3、原理图

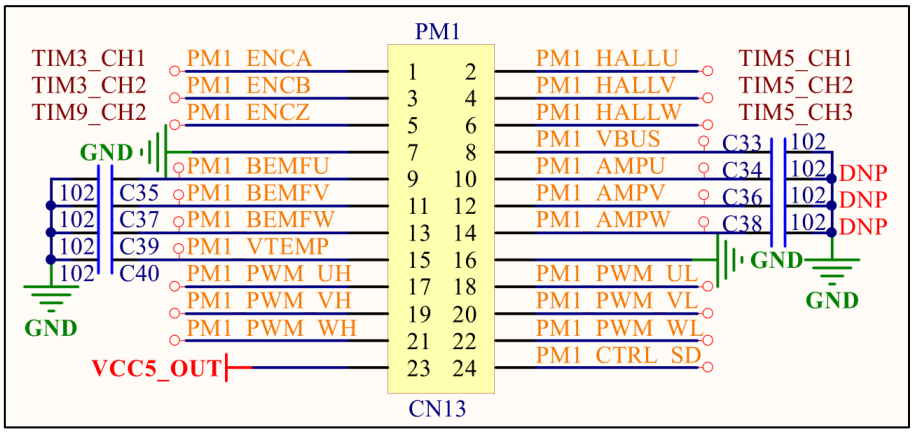


图 26.2.1.1 无刷电机接口 1 原理图

接口涉及的 IO 如下。

直流无刷驱动板	IO 口
驱动板 U 相	PA8、PB13
驱动板 V 相	PA9、PB14
驱动板 W 相	PA10、PB15
过零 U 相	PH10
过零 V 相	PH11
过零 W 相	PH12
输出使能	PF10
电压	PB1
温度	PA0
电流 U 相	PB0
电流 V 相	PA6
电流 W 相	PA3

表 26.2.1.1 无刷相关 IO 口说明

驱动板和电机的连接方式之前已经介绍过了，驱动板和开发板连接只需要使用排线连接即可（霍尔传感器接口可不接，JP3 跳线帽注意接 H&Z 与 ZERO 即可），实物连接如下。

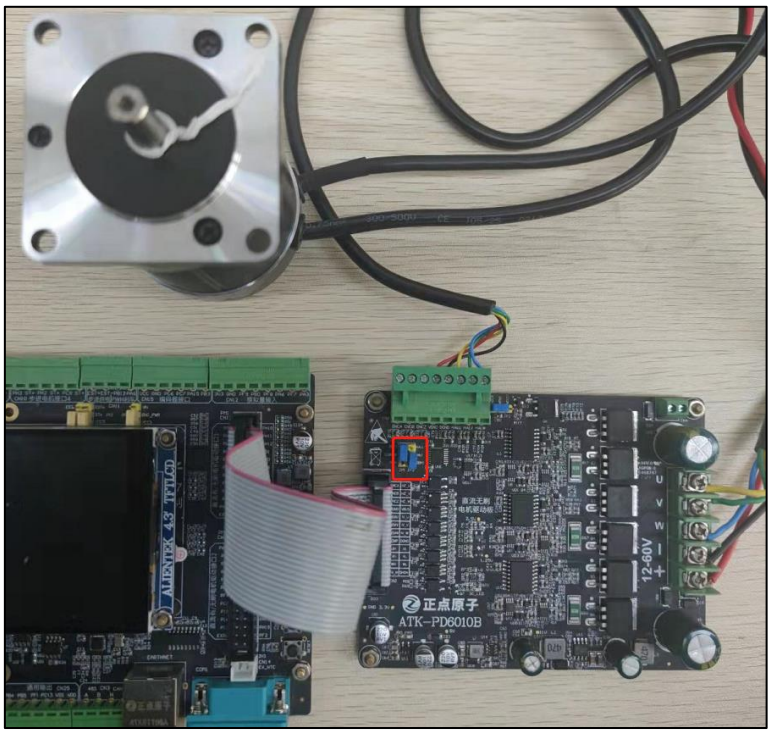


图 26.2.1.2 开发板&驱动板连接图

26.2.2 程序设计

26.2.2.1 无刷电机速度+电流双闭环的配置步骤

1）实现电机基本驱动

实现电机的基础驱动函数，启停、6 步换向组合等等

2）初始化 ADC

初始化 ADC 通道的 IO，设置 ADC 工作方式、DMA 等等

3）PID 闭环控制实现

实现 PID 的初始参数赋值、PID 计算等等

4）上位机通信协议

编写上位机通信代码，可在上位机上实时显示当前速度与目标速度的波形变化

5）编写中断服务函数

PWM 中断用于换向控制、堵转检测等等，添加 PID 的周期计算调用

26.2.2.2 程序流程图

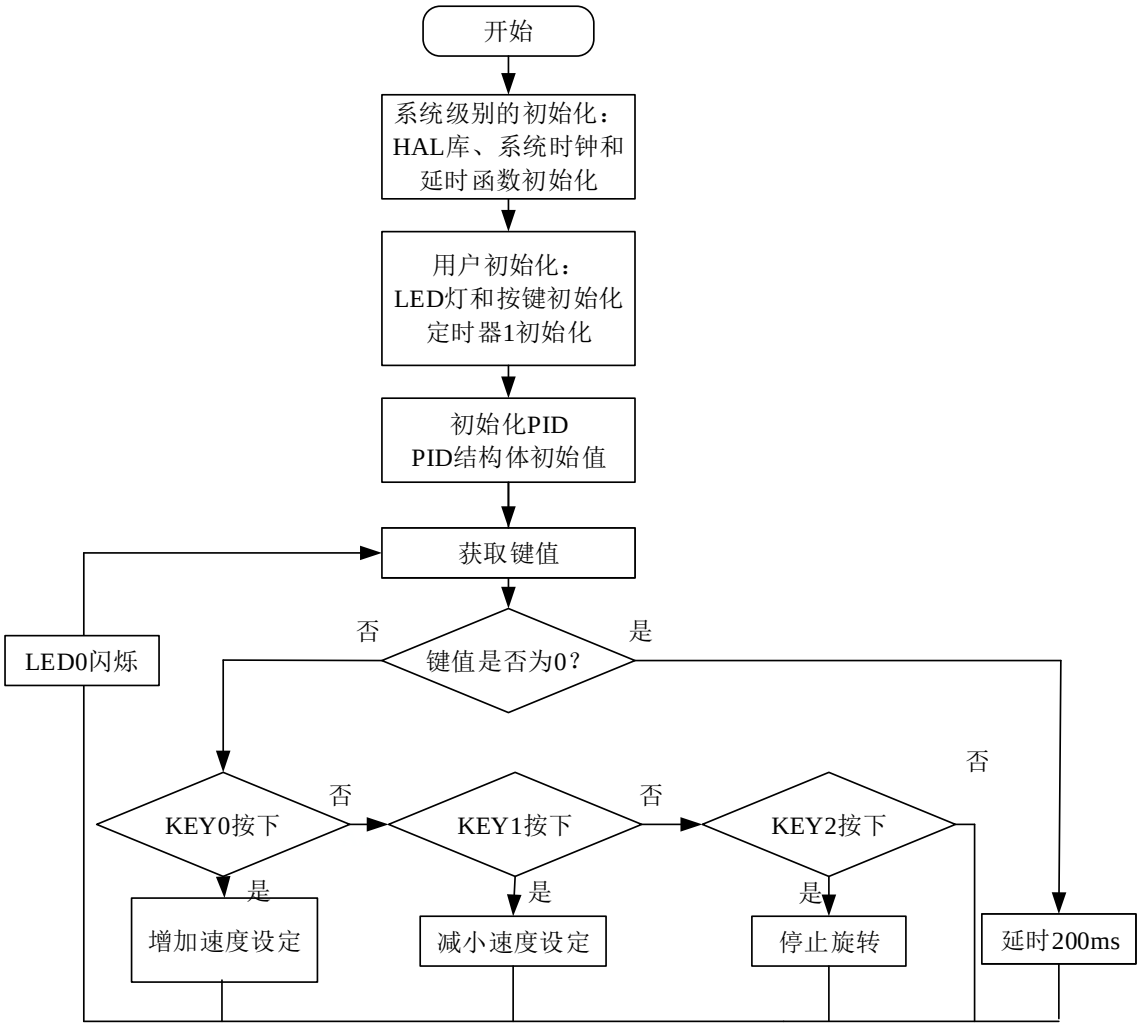


图 26.2.2.2.1 无刷电机速度+电流双闭环控制流程图

26.2.2.3 程序解析

这里我们只讲解核心代码，定时器及 ADC 的相关程序都和之前一致，这里就不重复列出了，速度+电流双闭环的 PID 参数的定义有不同。实际内容如下所示。

```
/*定义 PID 参数相关宏*/
#define  S_KP      0.05000f          /* 速度环的 P 参数需远小于电流环的 P */
#define  S_KI      0.00025f          /* I 参数*/
#define  S_KD      0.000150f         /* D 参数*/
/*定义 PID 参数相关宏*/
#define  C_KP      2.00f             /* P 参数*/
#define  C_KI      0.20f             /* I 参数*/
#define  C_KD      0.01f             /* D 参数*/
#define SMAPLSE_PID_SPEED  40       /* 采样率 单位 ms*/
```

其中 C_KP，C_KI，C_KD 为电流环的 PID 参数，S_KP，S_KI，S_KD 为速度环的 PID 参数，PID 的计算过程和之前的一致，这里就不贴出来了，大家感兴趣可以打开源码查看。重点内容 PID 控制在定时器的中断回调函数里进行，在 bldc_tim.c 中实现，贴出的程序删除了部分代码避免篇幅太长，中断服务函数内容如下。

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    uint8_t i;
```

```
    static uint8_t times_count=0;/*定时器时间记录*/
    if(htim->Instance == ATIM_TIMX_PWM)//55us
    {
    … …
#ifdef H_PWM_L_ON
        /* 过零控制 */
        zero_ctr_loop();
        /**************** PID 控制 ****************/
        if(g_bldc_motor1.run_flag == RUN && g_zero_ctr_status == 3)
        {
            /* 速度+电流双环控制 */
            pid_s_count++;
            pid_c_count++;
            /* 速度环 */
            if(pid_s_count>2)
            {
              … … /* 省略部分代码 */
            if(debug_switch == 0)
            {
                temp_pwm1 = increment_pid_ctrl(&g_speed_pid,g_bldc_motor1.speed);
                FirstOrderRC_LPF(motor_pwm_s,temp_pwm1,0.5);
                if(motor_pwm_s < 0)
                {
                    /*加速启动速度*/
                    if(motor_pwm_s >= -600)
                        motor_pwm_s = -600;
                        motor_pwm_sl = -motor_pwm_s;
                }
                else
                {
                    if(motor_pwm_s <= 600)
                        motor_pwm_s = 600;
                        motor_pwm_sl = motor_pwm_s;
                }
                *user_setpoint = debug_data_temp;/*重新保持上位机指令要求*/
                pid_s_count = 0;
            }
            }
            if(debug_switch == 0)
            {
                /* 电流环 */
                if(pid_c_count > 1)
                {
/*换向尖峰电流大于设定的电流值将导致 PID 调节转至电流环调节 速度环无法起作用，转速无法调节*/
                    if(adc_amp_bus > (g_current_pid.SetPoint - 20))
                    {
                        cf_count++;/* 滤除换向尖峰电流的影响 */
                        if(cf_count > 4)
                        {
                            cf_count = 0;
                            temp_pwm2 =
                            increment_pid_ctrl(&g_current_pid,adc_amp_bus);
                            FirstOrderRC_LPF(motor_pwm_c,temp_pwm2,0.085);
                        }
                    }
                    else
                    {
                        cf_count = 0;
                        temp_pwm2 =
                         increment_pid_ctrl(&g_current_pid,adc_amp_bus);
                        FirstOrderRC_LPF(motor_pwm_c,temp_pwm2,0.085);
                    }
                    pid_c_count = 0;
                }
                /* 电流环输出值大于速度环输出则使用速度环调节 */
```

```
            if(motor_pwm_c > motor_pwm_sl)
            {
                g_bldc_motor1.pwm_duty = motor_pwm_sl;
                if(motor_pwm_s < 0)                    /* 正反转积分控制 */
                    g_current_pid.Ui = -g_speed_pid.Ui;
                else
                    g_current_pid.Ui = g_speed_pid.Ui;
            }
            else  /* 速度环输出值大于电流环输出则使用电流环调节 */
            {
                g_bldc_motor1.pwm_duty = motor_pwm_c;
                if(g_bldc_motor1.dir == CCW)
                    g_speed_pid.Ui = -g_current_pid.Ui;
                else
                    g_speed_pid.Ui = g_current_pid.Ui;
            }
        }
    }
    /* 上位机 -> 开发板 方向变化处理 */
    if(debug_switch == 1 && abs((int)(*user_setpoint)) >= 400)
    {
        if(*user_setpoint > 0)
            g_bldc_motor1.dir = CW;
        else
            g_bldc_motor1.dir = CCW;
        start_motor1();                    /* 开启运行 */
        g_zero_ctr_status = 0;
        g_bldc_motor1.run_flag = RUN;      /* 开启运行 */
        debug_switch = 0;
    }

#endif
  }
  if(htim->Instance == TIM6)
  {
    /****************** 采集电机停机状态下的偏置电压 ******************/
    times_count++;
    if(g_bldc_motor1.run_flag == STOP)
    {
        uint8_t i;
        uint32_t avg[3] = {0,0,0};
        adc_amp_offset[0][adc_amp_offset_p] = g_adc_val[2];
        adc_amp_offset[1][adc_amp_offset_p] = g_adc_val[3];
        adc_amp_offset[2][adc_amp_offset_p] = g_adc_val[4];
        adc_amp_offset_p ++;
        NUM_CLEAR(adc_amp_offset_p,ADC_AMP_OFFSET_TIMES);
        for(i=0; i<ADC_AMP_OFFSET_TIMES; i++)
        {
            avg[0] += adc_amp_offset[0][i];
            avg[1] += adc_amp_offset[1][i];
            avg[2] += adc_amp_offset[2][i];
        }
        for(i=0; i<3; i++)
        {
            avg[i] /= ADC_AMP_OFFSET_TIMES;
            adc_amp_offset[i][ADC_AMP_OFFSET_TIMES] = avg[i];
        }
    }
    /****************** 定时判断电机是否发生堵塞 ******************/
    if(times_count == SMAPLSE_PID_SPEED)
    {
#if (LOCK_TAC == 2)
        if(g_bldc_motor1.locked_rotor == 1)  /* 到达一定速度后可进入闭环控制 */
        {
            clc++;
```

```
            if(clc > 50)                        /* 延迟 2s 后重新启动 */
            {
                clc = 0;
                pid_init();
                stop_motor1();
                g_speed_pid.SetPoint = 400.0; /* 400PRM */
                g_bldc_motor1.dir = CW;
                g_bldc_motor1.pwm_duty = 600; /* 加速启动速度 */
                g_bldc_motor1.run_flag = RUN; /* 开启运行 */
                start_motor1();               /* 运行电机 */
                g_bldc_motor1.locked_rotor = 0;
                g_zero_ctr_status = 0;        /* 堵塞状态需要重新定位初始位置 */
            }
        }
#endif
        times_count = 0;
    }
  }
}
```

在定时器 1 更新中断里首先优先进行电流环的 PID 计算，这里需要进行滤波，避免换向尖峰对 PID 的影响，接着通过过零跳变数量计算电机转速，经过计数分频，进入速度环 PID 计算函数，带入上述计算的速度，得到速度环计算出的 PWM 输出值，然后将两个环的输出做比较，如果电流环较大则使用速度环输出，否则使用电流环输出，最终赋值到定时器，在下一次中断时使用新的 PWM 用于驱动无刷电机。这里我们还在中断中加上了堵转处理，当电机因意外或过流导致停机时，当堵转时间超过 2s，此时会重新初始化 PID 并以 400RPM 的初始速度重新开启电机旋转。接着看来先下主函数 main.c 内容如下。

```
int main(void)
{
   uint16_t adc_vbus,adc_temp;
   uint8_t debug_cmd=0;
   float current_lpf[4]= {0.0f};
   uint8_t key,t;
   char buf[32];
   float current[3]= {0.0f};
   int16_t speed_diplay=0;
   float user_setpoint_temp=0.0;

   HAL_Init();                                 /* 初始化 HAL 库 */
   sys_stm32_clock_init(336, 8, 2, 7);         /* 设置时钟,168Mhz */
   delay_init(168);                            /* 延时初始化 */
   usart_init(115200);                         /* 串口初始化为 115200 */
   led_init();                                 /* 初始化 LED */
   key_init();                                 /* 初始化按键 */
   lcd_init();                                 /* 初始化 LCD */
   bldc_init(168000/18-1,0);
   bldc_ctrl(MOTOR_1,CW,1000);                 /* 初始无刷电机接口 1 速度 */
   adc_nch_dma_init();
   pid_init();

   g_point_color = WHITE;
   g_back_color  = BLACK;
   lcd_show_string(10,10,200,16,16,"BLDC Motor Test",g_point_color);
   lcd_show_string(10,30,200,16,16,"KEY0:Step++",g_point_color);
   lcd_show_string(10,50,200,16,16,"KEY1:Step--",g_point_color);
   lcd_show_string(10,70,200,16,16,"KEY2:Stop",g_point_color);

#if DEBUG_ENABLE                                                    /*开启调试*/
   debug_init();                                                    /*PID 调试初始化*/
   debug_send_motorcode(BLDC_MOTOR );                               /*直流无刷电机*/
   debug_send_motorstate(IDLE_STATE);                               /*电机空闲*/
   /* 初始化同步数据（选择第 x 组 PIDX,目标速度地址,P,I,D 参数）到上位机 */
```

```
    debug_send_initdata(TYPE_PID1,user_setpoint,C_KP,C_KI,C_KD);
    debug_send_initdata(TYPE_PID2,user_setpoint,S_KP,S_KI,S_KD);
#endif

    while (1)
    {
        … … /* 省略部分代码 */

        key = key_scan(0);
        if(key == KEY0_PRES)
        {
            step_up();
        }
        else if(key == KEY1_PRES)                    /* 按下 KEY1 开启电机 */
        {
            step_down();
        }
        else if(key == KEY2_PRES)                    /* 按下 KEY2 关闭电机 */
        {
            stop_motor1();                           /* 停机 */
            g_bldc_motor1.run_flag = STOP;           /* 标记停机 */
            g_bldc_motor1.pwm_duty = 0;
            motor_pwm_sl = 0;
            motor_pwm_c = 0;
            g_zero_ctr_status = 0;
            g_bldc_motor1.speed = 0;
            pid_init();                              /* 初始化 PID */
        }
        delay_ms(10);

#if DEBUG_ENABLE
        /* Debug 发送部分 */
        /* 主要显示参数 */
        debug_send_valtage(adc_vbus * ADC2VBUS);     /* 发送电压 */
        debug_send_speed(g_bldc_motor1.speed);       /* 发送速度 */
        debug_send_distance((uint64_t)(g_bldc_motor1.sum_pos));
        debug_send_temp(50,get_temp(adc_temp));      /* 发送电机温度、驱动板温度 */
        debug_send_current((float)(current_lpf[0]/1000),
                           (float)(current_lpf[0]/1000),
                           (float)(current_lpf[0]/1000));/*发送电流*/
        /* 电流波形和速度波形 */
        debug_send_wave_data(1,(int16_t)g_bldc_motor1.speed);  /*1 发送实际速度 */
        debug_send_wave_data(2,(int16_t)*user_setpoint);       /*2 发送目标速度 */
        debug_send_wave_data(3,current_lpf[0]); /* 选择通道 3 发送实际电流 U */
        debug_send_wave_data(4,current_lpf[1]); /* 选择通道 4 发送实际电流 V */
        debug_send_wave_data(5,current_lpf[2]); /* 选择通道 5 发送实际电流 W */
        debug_send_wave_data(6,current_lpf[3]); /* 选择通道 6 发送实际母线电流 */
        debug_send_wave_data(7,adc_amp_bus);     /* 选择通道 7 发送实际电流 W */
        debug_send_wave_data(8,g_current_pid.SetPoint);        /*8 实际母线电流 */
        /* Debug 接收部分 */
        debug_receive_pid(TYPE_PID1,(float*)&g_current_pid.Proportion,
        (float*)&g_current_pid.Integral,(float*)&g_current_pid.Derivative);
        debug_receive_pid(TYPE_PID2,(float*)&g_speed_pid.Proportion,
        (float*)&g_speed_pid.Integral,(float*)&g_speed_pid.Derivative);
        debug_cmd = debug_receive_ctrl_code();                    /* 读取命令 */
        if(debug_cmd == HALT_CODE)                                /* 停机 */
        {
            stop_motor1();
            pid_init();
            g_bldc_motor1.run_flag = STOP;                        /* 标记停机 */
            g_bldc_motor1.pwm_duty = 0;
        }
```

```
        else if(debug_cmd == RUN_CODE)                          /* 运行 */
        {
            g_bldc_motor1.run_flag = RUN;                       /* 运行标记 */
            *user_setpoint = 400;                               /* 自动设置目标 */
            debug_data_temp = *user_setpoint;
            start_motor1();                                     /* 启动电机 */
            debug_send_motorstate(RUN_STATE);                   /* 电机运行 */
        }
        else if (debug_cmd == BREAKED)                          /* 刹车*/
        {
            *user_setpoint = 0;                                 /* 减速直至 0 */
            debug_send_motorstate(BREAKED_STATE);               /* 电机刹车 */
        }
#endif
        delay_ms(10);
    }
}
void step_up(void)
{
    g_bldc_motor1.run_flag = RUN;                               /* 开启运行 */
    start_motor1();                                             /* 开启运行 */

    if(g_bldc_motor1.dir == CCW && *user_setpoint == 0)         /* 切换方向条件*/
    {
        g_bldc_motor1.dir = CW;
    }
    *user_setpoint += 400;                                      /* 递增 */
    if(*user_setpoint == 0)
    {
        g_bldc_motor1.run_flag = STOP;                          /* 标记停机 */
        stop_motor1();                                          /* 停机 */
        pid_init();                                             /* 初始化 PID */
        g_bldc_motor1.speed = 0;
        g_zero_ctr_status = 0;
        motor_pwm_sl = 0;
        motor_pwm_c = 0;
        g_bldc_motor1.pwm_duty = 0;
    }
    if(*user_setpoint >= 3200)                                  /* 限速 */
        *user_setpoint = 3200;
    debug_data_temp = *user_setpoint;
}
void step_down(void)
{
   g_bldc_motor1.run_flag = RUN;                                /* 开启运行*/
    start_motor1();                                             /* 运行电机*/
    /*切换方向条件*/
    if(g_bldc_motor1.dir == CW && *user_setpoint == 0)
    {
        g_bldc_motor1.dir = CCW;
    }
    *user_setpoint -= 400;                                      /* 递增 */
    if(*user_setpoint == 0)
    {
        g_bldc_motor1.run_flag = STOP;                          /* 标记停机*/
        stop_motor1();                                          /* 停机*/
        pid_init();                                             /* 初始化 PID*/
        g_bldc_motor1.speed = 0;
        g_zero_ctr_status = 0;
        motor_pwm_sl = 0;
        motor_pwm_c = 0;
        g_bldc_motor1.pwm_duty = 0;
    }
```

```
    if(*user_setpoint <= -3200)                                    /* 限速 */
    *user_setpoint = -3200;
    debug_data_temp = *user_setpoint;
}
```

main 函数主要初始化基本外设以及 BLDC 等等，然后添加上位机通信协议，将实际转速数据、目标转速数据、实际电流、目标电流分别发送给上位机的通道 1、2、7、8 上进行显示。按键判断逻辑：按下 KEY0 目标转速加 400RPM，按下 KEY1 目标转速减 400RPM，按下 KEY2 停止电机旋转

26.2.3 下载验证

下载代码后，可以看到 LED0 在闪烁，说明程序已经正常在跑了，LCD 上显示按键功能、占空比以及电机速度信息和显示程序计算的电压、电流等信息，当我们按下 KEY0，目标速度将增大；按下 KEY1，目标速度将减小；按下 KEY2，电机将停止。目标速度为正数时，电机正转，反之电机反转。我们再打开 PID 调试助手（注意接上串口 1 的 USB_TTL 连接至电脑），选择对应的串口端口，接着勾选通道 1 和 2 以及通道 7 和 8，点击“开始”按钮，即可开始显示波形。大家可以使用手给电机增加点负载，可以明显感觉到电机一开始会因为负载增加速度减慢，后续逐渐加速上去直至和目标速度一致，如下图 26.2.3.1，大家可以自己下载程序并点击按键测试效果。

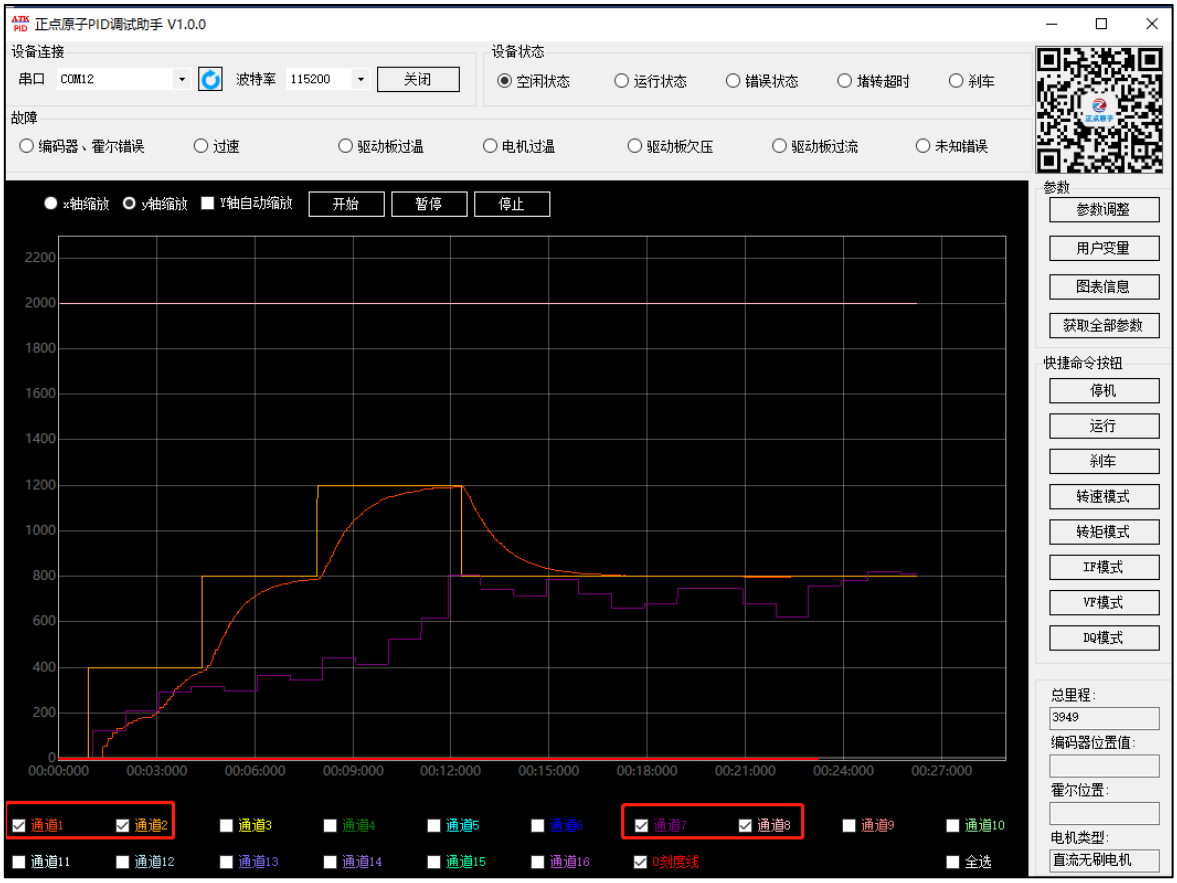


图 26.2.3.1 PID 调节效果

图 26.2.3.1 中，橙线代表目标速度，红线代表实际速度，紫线代表实际母线电流，粉线代表目标电流（由于 PID 初始化时已经设定了电流环的目标值，所以粉线为直线），当我们按下 KEY0，目标速度增大，橙线先发生变化，而红线（实际速度）会逐渐靠近橙线（目标速度）；可以看到紫线（实际电流）并未靠近粉线（目标电流），是由于代码中将两个环的输出做比较，如果电流环较大则使用速度环输出，否则使用电流环输出导致的。当我们手动给电机加负载，就能明显看到实际电流会在目标电流附近波动；按下 KEY1，目标速度将减小，曲线的变化同理；按下 KEY2，电机将停止，目标速度将为 0。

注意：1、电流环的波形存在小幅振荡属于正常现象，如果希望波形更稳定，可以适当地调整 PID 系数以及增大滤波次数；2、滤波次数越多则系统的响应越慢，大家需要在系统的响应速度和稳定性之间寻找平衡点；3、PID 系数并不是通用的，如果 PID 曲线不理想，大家需要根据自己的实际系统去调节。

第三篇 FOC 篇

现在我们已经学会如何使用 STM32 去驱动电机了，接下来我们将进入高级应用 FOC 篇的内容。此篇只适用于无刷电机（BLDC 和 PMSM）。从前面的无刷电机专题我们了解到控制无刷电机的基本方法——“六步换向”，但是我们可以发现一个缺点，使用六步换向的方式只能控制电机进行**段位移**，即每次驱动电机只能转动固定的角度，这种控制方式放到高精度的场景里面显然不适合（如工业机器人的机械臂），那么如何控制无刷电机丝滑的转动起来呢？这就需要用到本篇所学习的控制方式——“FOC”。FOC 能对无刷电机进行“**像素级**”控制，使其控制精度大大提高，相应的 FOC 控制算法会比较复杂，涉及到大量的数学、物理学等知识，掌握起来难度较大，需要大家认真学习，多多练习，做到举一反三。

第 27 章 FOC

为了进一步提升无刷电机的可控性，引入了 FOC 控制方式。本章主要以 PMSM 为例带领大家一起学习 FOC 的控制方式。

本章分为如下几个小节：

27.1 FOC 原理介绍
27.2 FOC 硬件平台
27.3 FOC 开发工具
27.4 FOC 例程创建
27.5 FOC 下载验证
27.6 ST MCSDK 库源码介绍
27.7 FOC 例程分析

27.1 FOC 原理介绍

FOC（field-oriented control）为磁场定向控制，又称为矢量控制（vectorcontrol），是目前无刷直流电机（BLDC）和永磁同步电机（PMSM）高效控制的最佳选择。FOC 可以精确地控制磁场大小与方向，使得电机转矩平稳、噪声小、效率高，并且具有高速的动态响应。目前已在很多应用上逐步替代传统的控制方式，在运动控制行业中备受瞩目。

我们都知道，电流可以产生磁场，并且磁场大小与电流大小成正比，因此为了使定子构造最合适的旋转磁场，需要精确控制线电流。想要磁场旋转就需要线电流做着正弦变化，而 3 组线圈绕组的角度差，就使得三相电流需要时刻做相位差为 120 度的正弦变化，这时可使定子构造最合适的旋转磁场，显然简单的 6 步换向无法控制三相电流做正弦变化，转矩在一定程度上会有跳变，这样无法输出稳定转矩，因此需要 FOC 来保持转子的扭矩时刻连续稳定可调。下面总结六步换向和 FOC 控制方式的对比表，如表 27.1.1：

控制方式	优点	缺点
六步换向	控制算法相较简单	精度差、运转不流畅、转矩波动大、存在一定的电流噪声、适用于对电机转动性能要求不高的场合
FOC	转矩平稳、效率高、噪声小、动态响应快	硬件成本较高、对 MCU 性能有较高要求、控制算法难度大。

表 27.1.1 控制方式对比表

27.1.1 FOC 框图介绍

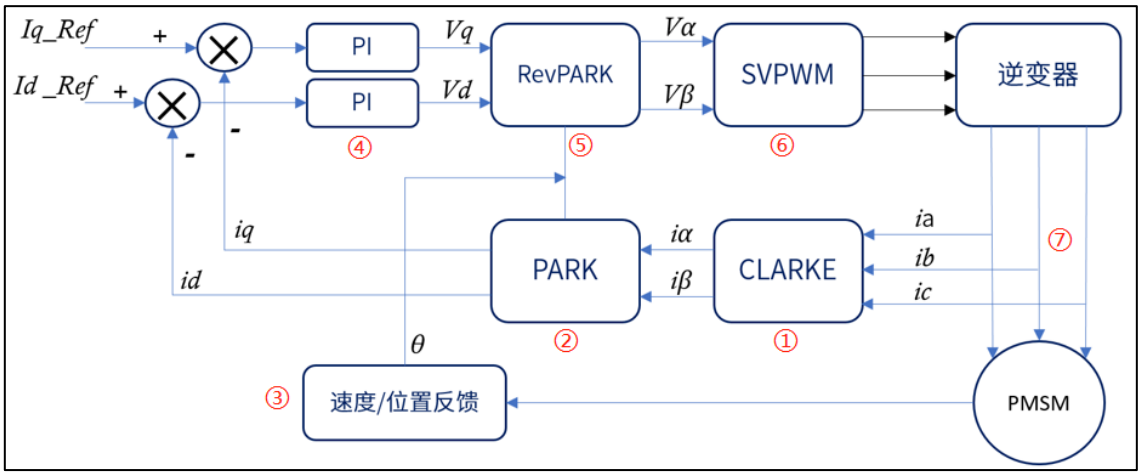


图 27.1.1.1 电流闭环控制 FOC 框图

图 27.1.1.1 是以电流闭环控制为例，也就是让电机始终保持一个恒定力矩（力矩与电流成正比）。

从上图可以看到最左边的 Iq_Ref 和 Id_Ref 两个变量经过 PID 控制器进行反馈调节，其中涉及到几个变换模块，包括：Clarke 变换、Park 变换以及反 Park 变换，最后是 SVPWM 模块作用到三相逆变器进而控制电机旋转。相信大家对上面这些过程不是很理解，没关系，我们先来大致的概括下 FOC 的整个控制框图，后面会对整体框架进行拆分讲解。

① 对电机三相电流进行采样得到：ia、ib、ic；
② 将 ia、ib、ic 经过 clarke 变换得到 iα、iβ；
③ 将 iα、iβ 经过 park 变换得到 iq、id；
④ 计算 iq、id 和其设定值 iq_Ref、id_Ref 的误差；
⑤ 将上述误差输入到两个 PID（只用到 PI）控制器，得到输出的控制电压 Vq、Vd；
⑥ 将 Vq、Vd 进行反 park 变换得到 Vα、Vβ；
⑦ 将 Vα、Vβ 输入 SVPWM 模块进行调制，合成电压空间矢量，输出该时刻三个半桥的开关状态进而控制电机旋转；
⑧ 循环上述步骤。

下面我们来分别介绍每一步的具体操作以及意义。

27.1.2 FOC 坐标变换

假设我们将一个 PMSM 电机手动让其匀速旋转，此时使用示波器观察它的三相输出电压(反电动势)，我们会发现示波器上会得到三组正弦波，并且三组正弦波之间两两相位差为 120° 。如图 27.1.2.1：

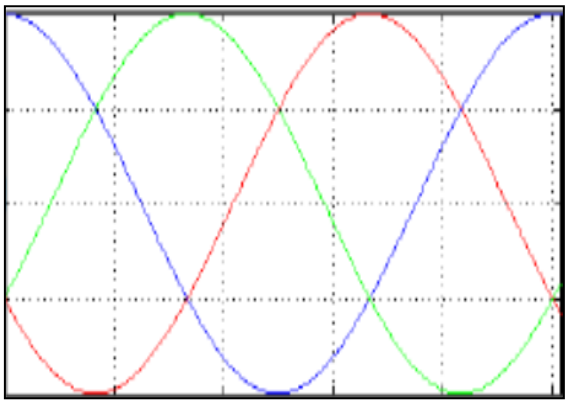


图 27.1.2.1 PMSM 电机运转时三相电流波形

所以反过来我们在三相无刷电机的三相线圈上输入上述三相正弦电压，那么就可以驱动

无刷电机平稳高效地旋转了。而这正是 FOC 驱动无刷电机的基本手段，但是从控制的角度来看，我们根本就不想跟三个正弦波打交道，因为对于非线性的信号进行准确控制就要使用复杂的高阶控制器，这对于建模成本、处理器算力、控制实时性等都是非常不利的，那么有没有什么方法可以将被控制量线性化呢？答案是当然有的，只需要应用一些数学技巧。

27.1.2.1 Clarke 变换

当 PMSM 电机匀速运转时，将采集到相位相差 120° 的三相电流 Ia、Ib、Ic 使用三相坐标系表示，如下图绿色线所示：

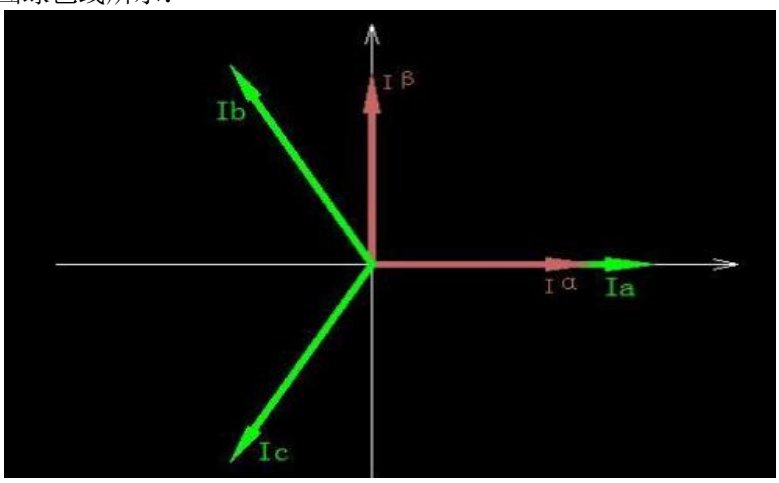


图 27.1.2.1.1 三相坐标系

我们可以利用一些数学小技巧，将三相坐标系变换成直角坐标系，我们把新的直角坐标系命名为 α-β 坐标系（如图 27.1.2.1.1 红色线所示），变换原则是电流产生的磁场相等。

Clarck 变换用于将三相静止坐标系变换到两相静止坐标系，变换前后在坐标系中产生的磁场等效。

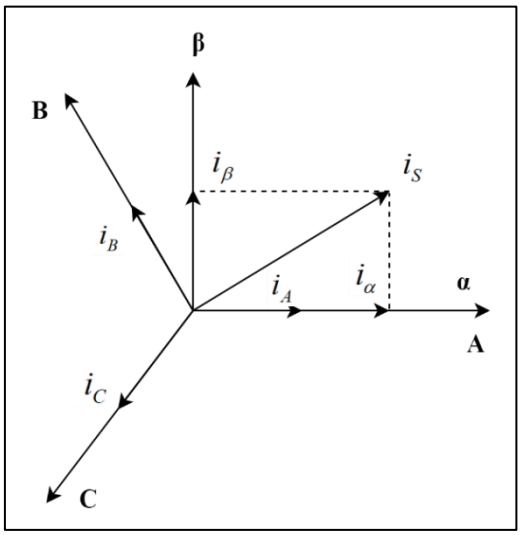


图 27.1.2.1.2 Clarck 变换

假设变换前三相定子绕组匝数为N_3，变换后的两相定子绕组匝数为N_2，根据磁动量守恒可得：

$$N_2\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = N_3\begin{bmatrix} i_A - i_B\cos 60° - i_C\cos 60° \\ sin0 \quad i_B\sin 60° - i_C\sin 60° \end{bmatrix}$$

提取i_α和i_β可以得到

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \frac{N_3}{N_2}\begin{bmatrix} 1 & \frac{-1}{2} & \frac{-1}{2} \\ 0 & \frac{\sqrt{3}}{2} & \frac{-\sqrt{3}}{2} \end{bmatrix}\begin{bmatrix} i_A \\ i_B \\ i_C \end{bmatrix}$$

遵循恒幅值不变原理可得

$$\frac{N_3}{N_2} = \frac{2}{3}$$

所以：

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \frac{2}{3}\begin{bmatrix} 1 & \frac{-1}{2} & \frac{-1}{2} \\ 0 & \frac{\sqrt{3}}{2} & \frac{-\sqrt{3}}{2} \end{bmatrix}\begin{bmatrix} i_A \\ i_B \\ i_C \end{bmatrix}$$

根据基尔霍夫定律（KCL）：

$$i_A + i_B + i_C = 0$$

所以归纳：

$$\begin{cases} i_A + i_B + i_C = 0 \\ i_\alpha = \frac{2}{3}i_A - \frac{1}{3}i_B - \frac{1}{3}i_C \\ i_\beta = 0 - \frac{1}{\sqrt{3}}i_B - \frac{1}{\sqrt{3}}i_C \end{cases}$$

最后化简得：

$$\begin{cases} i_\alpha = i_A \\ i_\beta = \frac{i_A + 2i_B}{\sqrt{3}} \end{cases}$$

经过 Clarke 变换后就变成了直角坐标系啦！变换前后的波形如图 27.1.2.1.3:

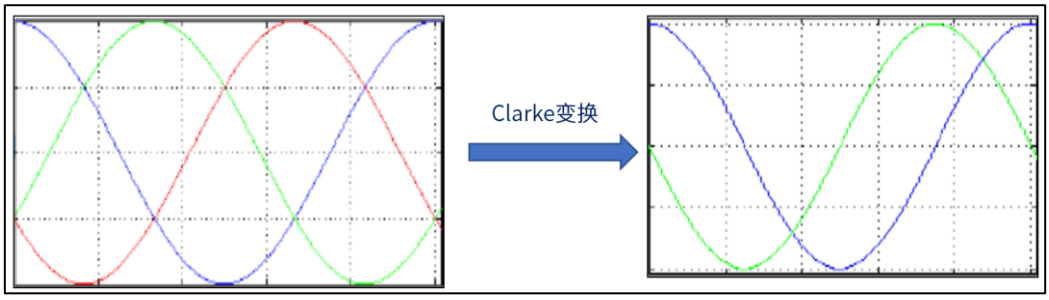


图 27.1.2.1.3 经过 Clarke 变换前后的波形

可以看到变换后还是正弦波，虽然少了一个需要控制的变量，但是新的变量还是非线性的（正弦），控制它依旧难度很大，那有没有办法把它们线性化呢，当然有的，接着就看下 Park 变换

27.1.2.2 Park 变换以及反 Park 变换

Park 变换可以将电机从两相静止坐标系变换到两相旋转坐标系（dq 坐标系），从而解耦出电机转矩分量和电机的励磁分量，这两个分量相互垂直、互不影响。反 Park 变换即为 Park 的逆变换。

首先我们来看下 Park 变换的原理，这里将 Clarke 变换后的 α—β 坐标系旋转 θ 度，其中 θ 为转子旋转的角度，如下图：

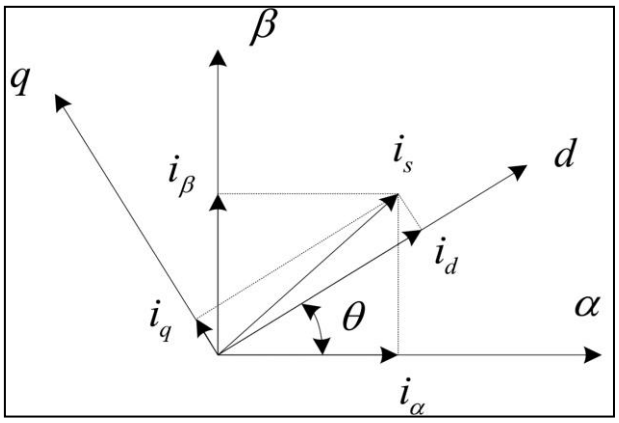


图 27.1.2.2.1 Park 变换

图 27.1.2.2.1 中，**d 轴方向与转子内磁场方向重合**，称为直轴；**q 轴方向与转子内磁场方向垂直**，称为交轴。如图 27.1.2.2.2 所示

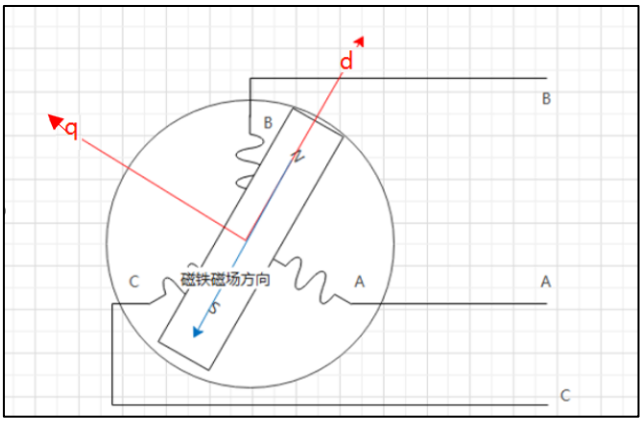


图 27.1.2.2.2 d-q 轴示意图

此时我们将 α—β 坐标系变换到 d-q 坐标系，即 Park 变换；依据坐标变换，Park 变换矩阵为：

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix}$$

变换公式如下：

$$\begin{cases} i_d = i_\alpha \cos(\theta) + i_\beta \sin(\theta) \\ i_q = -i_\alpha \sin(\theta) + i_\beta \cos(\theta) \end{cases}$$

在将 Park 变换后的结果在经过 PID 控制器，PI 运算后的输出结果在进行反 Park 变换，反 Park 变换矩阵为：

$$\begin{bmatrix} U_\alpha \\ U_\beta \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} U_d \\ U_q \end{bmatrix}$$

变换公式如下：

$$\begin{cases} U_\alpha = U_d \cos(\theta) - U_q \sin(\theta) \\ U_\beta = U_d \sin(\theta) + U_q \cos(\theta) \end{cases}$$

首先 α—β 坐标系经过 Park 变换后，即为 d-q 坐标系，该坐标系是始终跟着转子旋转的，旋转角度 θ 需通过编码器/霍尔传感器读取。经过这一步的变换，我们会发现，一个匀速旋转向量在这个坐标系下变成了一个**定值！**（因为参考系相对于该向量静止了），这个坐标系下**两个控制变量都被线性化了**！Park 变换前后波形如图 27.1.2.2.3 所示：

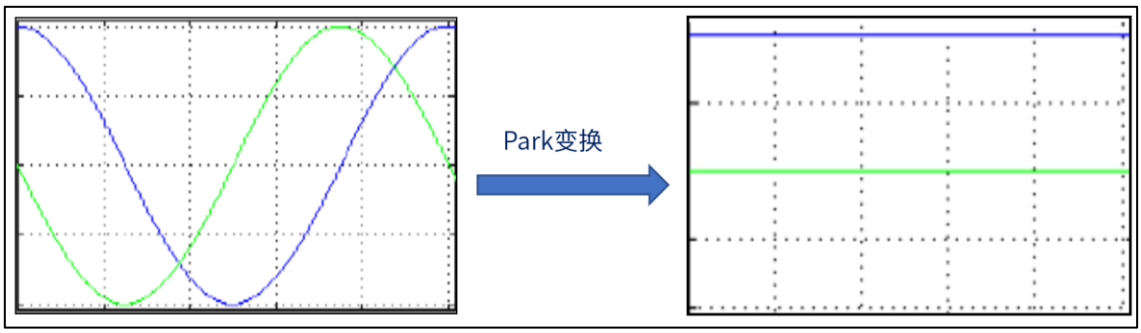


图 27.1.2.2.3 Park 变换前后波形

Park 变换后的控制量为 Iq、Id，很显然线性化后的控制量我们就可以使用 PID 来进行控制了；但是为什么我们还要进行反 Park 变换，将其又变换为 α—β 坐标系呢？因为 SVPWM 算法的实现需要用到静止的 α—β 坐标系，所以当我们完成了控制信号的 PID 运算后，还需进行反 Park 变换。关于 PID 控制部分这里就不赘述了，大家可以查看第八章的 PID 专题讲解。

27.1.3 FOC 的目的

经过前面的一通操作我们将转子磁链进行了解耦，分解为转子旋转的**径向**和**切向**这两个方向的变量：Iq 以及 Id，那这两个控制量到底代表什么含义呢？我们通过图 27.1.3.1 来进行讲解。

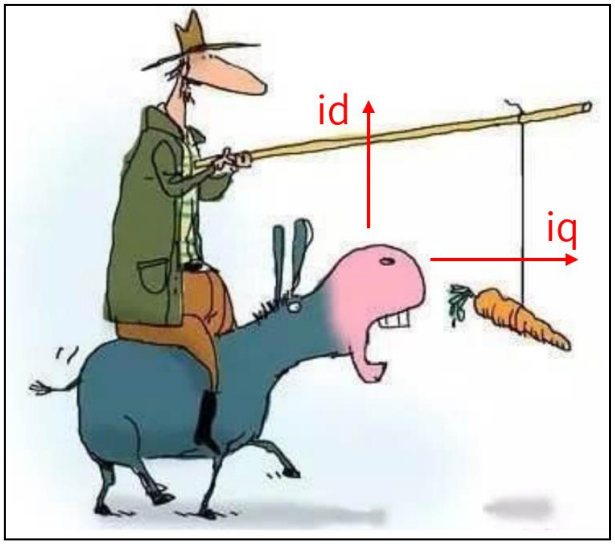


图 27.1.3.1 Iq、Id 的含义

我们可以看到对于图 27.1.3.1 中的人来说，他想驴能一直往前走，即 iq 是他需要的驱动力，iq 越大驴走得越快，iq 越小驴走得越慢，而 id 对于他来说无关紧要。对于电机来说也是一样的道理，即 FOC 的目的如图 27.1.3.2 所示：

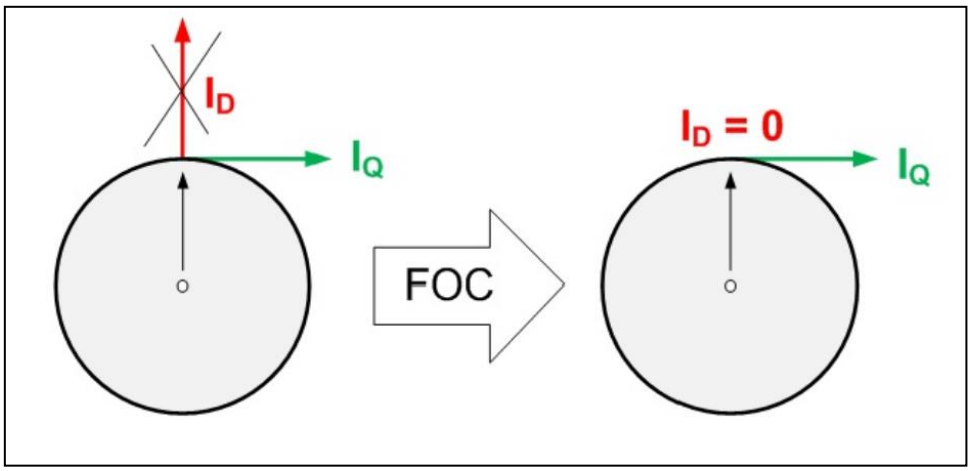


图 27.1.3.2 FOC 的目的

经过上述分析，我们得知：

1、Iq 是我们需要的。代表了期望的力矩输出；

2、Id 是我们不需要的。我们希望尽可能把它控制为 0；

至此整个 FOC 框架基本捋清楚了，只差最后一个步骤就是“SVPWM”，下面我们开始对 SVPWM 做一个详细的介绍。

27.1.4 SVPWM 解析

从前面的学习，我们知道如果要平稳地驱动三相电机转动，我们就需要生成三个相位相差 120 度的正弦波

为了使输出电流近乎于正弦波，可以使用 PWM 的方式，通过调整占空比使等效电流近似为正弦波，这种 PWM 也就是 SPWM。SPWM 是从电源的角度出发，构造出旋转所需的正弦电压，这种方式比较适合用作逆变器，但是对于无刷或者永磁同步电机，调节过的电源实际在定子线圈中产生的电流并不一定是所需的结果。这也就是所谓的开环，是没有反馈的。所以这种控制方式还是不能较为准确的控制定子电流。因此这种永磁无刷电机直接通三相交流电并不一定能很平稳的旋转起来。不然的话纯硬件就可以产生规则的电流，哪还需要微控制器做复杂运算。

隆重介绍，SVPWM，是的，它还是 PWM，既然用 MOS 管调制就离不开 PWM，它和 SPWM 的区别是**SVPWM 依靠的是 MOS 管的开关顺序和开关时间来调制有效电流**。而 SPWM 不依赖开关顺序，3 相独立调制。SVPWM 是依赖开关顺序的，需要把控整体的 3 相开关，让它们组合出不同顺序和时间的开关序列，以此模拟出正弦电流。SVPWM 乩优点主要有：优化谐波程度比较高，消除谐波效果要比 SPWM 好，实现容易，可以提高电压利用率。比较适合数字化控制系统。

SVPWM 的主要思想是以三相对称正弦波电压供电时三相对称电机定子理想磁链圆为参考标准，以三相逆变器不同开关模式作适当的切换，从而形成 PWM 波，以所形成的实际磁链矢量来追踪其准确磁链圆。传统的 SPWM 仅电源的角度出发生成一个可调频调压的正弦波电源，而 SVPWM 将逆变系统和异步电机看作一个整体来考虑，模型比较简单，也便于微处理器的实时控制。

普通的三相全桥是由六个开关器件构成的三个半桥。这六个开关器件组合起来（同一个桥臂的上下半桥的信号相反）共有 8 种安全的开关状态. 其中 000、111（这里是表示三个半桥的开关状态）这两种开关状态在电机驱动中都不会产生有效的电流。因此称其为**零矢量**。另外 6 种开关状态分别是六个**非零矢量**。它们将 360 度的电压空间分为 60 度一个扇区，共六个扇区，利用这六个基本非零矢量和两个零量，可以合成 360 度内的任何矢量。

当要合成某一矢量时先将这一矢量分解到离它最近的两个基本矢量，而后用这两个基本矢量去表示，而每个基本矢量的作用大小就利用作用时间长短去代表。用电压矢量按照不同的时间比例去合成所需要的电压矢量。从而保证生成电压波形近似于正弦波。

在变频电机驱动时，矢量方向是连续变化的，因此我们需要不断的计算矢量作用时间。为

了计算机处理的方便，在合成时一般是定时器计算（如每 0.1ms 计算一次）。这样我们只要算出在 0.1ms 内两个基本矢量作用的时间就可以了。由于计算出的两个时间的总和可能并不是 0.1ms(比这小)，而那剩下的时间就按情况插入合适零矢量。 由于在这样处理时，合成的驱动波形和 PWM 很类似。因此我们还叫它 PWM，又因这种 PWM 是基于电压空间矢量去合成的，所以就叫它 SVPWM 了。

SVPWM 的主要特点有：

1.在每个扇区虽有多次开关切换，但每次开关切换只涉及一个器件，所以开关损耗小。

2.利用电压空间矢量直接生成三相 PWM 波，计算简单。

3.逆变器输出线电压基波最大值为直流侧电压，比一般的 SPWM 逆变器输出电压高 15%

27.1.4.1 SVPWM 算法讲解

设直流母线侧电压为U_{dc}，MOS 管 3 相调制输出电压分别为U_{AO}、U_{BO}、U_{CO}，其分别加在空间上互差 120°的三相静止坐标系上，如图 27.1.4.1.1 所示：

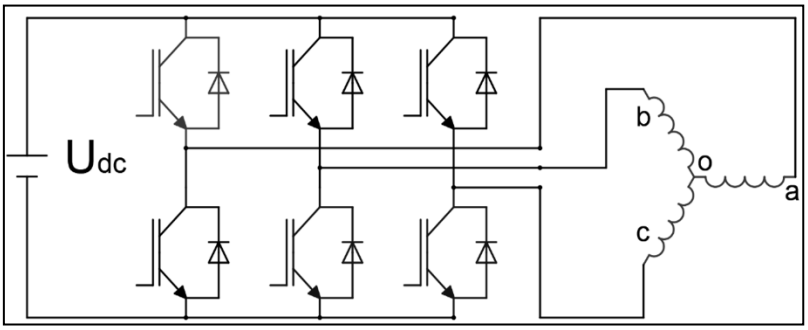


图 27.1.4.1.1 三相绕组示意图

定义三个电压空间矢量u_{AO}、u_{BO}、u_{CO}，方向分别在其轴线上，大小随时间按正弦规律变化，相位差 120°，设U_m为相电压基波峰值，f 为电源频率则：

$$\begin{cases} U_{AO}(t) = U_m\cos(\omega t) = \dfrac{U_m}{2}\left(e^{j\omega t} + e^{-j\omega t}\right) \\ U_{BO}(t) = U_m\cos\left(\omega t - \dfrac{2\pi}{3}\right) = \dfrac{U_m}{2}\left[e^{j\left(\omega t-\frac{2\pi}{3}\right)} + e^{-j\left(\omega t-\frac{2\pi}{3}\right)}\right] \\ U_{CO}(t) = U_m\cos\left(\omega t + \dfrac{2\pi}{3}\right) = \dfrac{U_m}{2}\left[e^{j\left(\omega t+\frac{2\pi}{3}\right)} + e^{-j\left(\omega t+\frac{2\pi}{3}\right)}\right] \end{cases}$$

在三相静止坐标系下：

$$\begin{cases} u_{AO}(t) = U_{AO}(t)e^{j0} \\ u_{BO}(t) = U_{BO}(t)e^{j2\pi/3} \\ u_{CO}(t) = U_{CO}(t)e^{-j2\pi/3} \end{cases}$$

则三相电压空间矢量的合成矢量$u_S(t)$为：

$$\begin{aligned} u_S &= U_{AO}(t) + U_{BO}(t) + U_{CO}(t) = U_{AO}(t)e^{j0} + U_{BO}(t)e^{j2\pi/3} + U_{CO}(t)e^{-j2\pi/3} \\ &= \frac{U_m}{2}\left(e^{j\omega t} + e^{-j\omega t}\right) + \frac{U_m}{2}\left[e^{j\left(\omega t-\frac{2\pi}{3}\right)} + e^{-j\left(\omega t-\frac{2\pi}{3}\right)}\right]e^{j2\pi/3} \\ &\quad + \frac{U_m}{2}\left[e^{j\left(\omega t+\frac{2\pi}{3}\right)} + e^{-j\left(\omega t+\frac{2\pi}{3}\right)}\right]e^{-j2\pi/3} \\ &= \frac{U_m}{2}\left[e^{j\omega t} + e^{-j\omega t} + e^{j\omega t} + e^{-j\left(\omega t-\frac{2\pi}{3}\right)} + e^{j\omega t} + e^{-j\left(\omega t-\frac{2\pi}{3}\right)}\right] = \frac{3}{2}U_m e^{j\omega t} \end{aligned}$$

可见$u_S(t)$是一个旋转的空间矢量，它的幅值不变，为相电压峰值的 1.5 倍，且以角频率 ω= 2π*f* 按逆时针方向匀速旋转的空间矢量。

由于驱动器三相桥臂依靠 6 只 MOS 管，而 MOS 管只有导通和非导通状态(PWM 信号输入)，因此三相开关全部组合有 8 种情况，这里定义一个开关函数$s_x(x = a、b、c)$，abc 分别代表三相绕组，当$s_a = 1$时，代表 A 相上桥臂导通，当$s_a = 0$时，代表 A 相下桥臂导通，其他

相同理。

那么(s_a、s_b、s_c)的全部可能组合共有八个，包括 6 个非零矢量 Ul(001)、U2(010)、U3(011)、U4(100)、U5(101)、U6(110)、和两个零矢量 U0(000)、U7(111)。下面我们以非零矢量 U4(100)为例来进行分析，即 (s_a、s_b、s_c)=(100)，代表 a 相上桥臂导通，b 相下桥臂导通，c 相下桥臂导通，如图 27.1.4.1.2 所示：

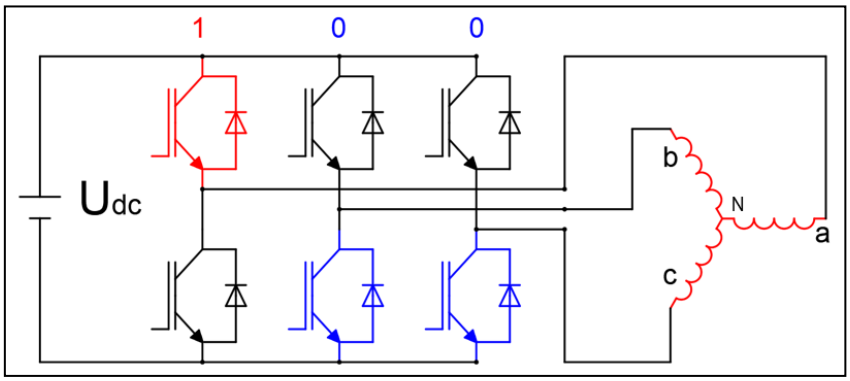


图 27.1.4.1.2　U4(100)导通情况

此时等效电路如图 27.1.4.1.3 所示：

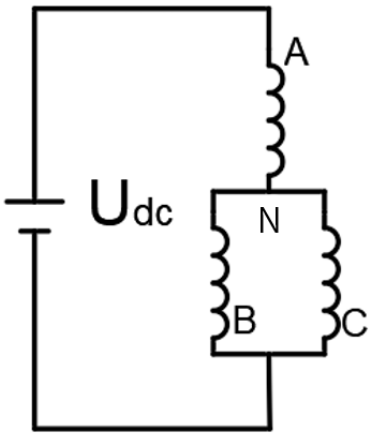


图 27.1.4.1.3　U4(100)等效电路

根据串联分压定理，得：

$$U_{aN}=\frac{2}{3}U_{dc}$$

$$U_{bN}=U_{cN}=-\frac{1}{3}U_{dc}$$

剩下的三相开关组合情况同理，可得各矢量开关状态、线电压、相电压如下：

S_a	S_b	S_c	矢量符号	线电压			相电压		
				U_{ab}	U_{bc}	U_{ca}	U_aN	U_bN	U_cN
0	0	0	U_0	0	0	0	0	0	0
1	0	0	U_4	U_{dc}	0	$-U_{dc}$	$\frac{2}{3}U_{dc}$	$-\frac{1}{3}U_{dc}$	$-\frac{1}{3}U_{dc}$
1	1	0	U_6	0	U_{dc}	$-U_{dc}$	$\frac{1}{3}U_{dc}$	$\frac{1}{3}U_{dc}$	$-\frac{2}{3}U_{dc}$
0	1	0	U_2	$-U_{dc}$	U_{dc}	0	$-\frac{1}{3}U_{dc}$	$\frac{2}{3}U_{dc}$	$-\frac{1}{3}U_{dc}$
0	1	1	U_3	$-U_{dc}$	0	0	$-\frac{2}{3}U_{dc}$	$\frac{1}{3}U_{dc}$	$\frac{1}{3}U_{dc}$

0	0	1	U_1	0	$-U_{dc}$	U_{dc}	$-\frac{1}{3}U_{dc}$	$-\frac{1}{3}U_{dc}$	$\frac{2}{3}U_{dc}$
1	0	1	U_5	U_{dc}	$-U_{dc}$	0	$\frac{1}{3}U_{dc}$	$-\frac{2}{3}U_{dc}$	$\frac{1}{3}U_{dc}$
1	1	1	U_7	0	0	0	0	0	0

表 27.1.4.1.1 各矢量线电压和相电压对比

6 个非零矢量幅值一致，均为$\frac{2}{3}U_{dc}$，从前面已知合成的电压向量的幅值是每个相所加相电压的 1.5 倍。我们用 SVPWM 重现该合成电压向量，希望重现的合成电压向量的幅值和所加相电压的幅值完全一样，所以在这里乘以$\frac{2}{3}$。

8 中开关状态可将 360 度的电压空间分为 60 度一个扇区，共六个扇区，利用这六个基本非零矢量和两个零量，可以合成 360 度内的任何矢量。如图图 27.1.4.1.4 所示：

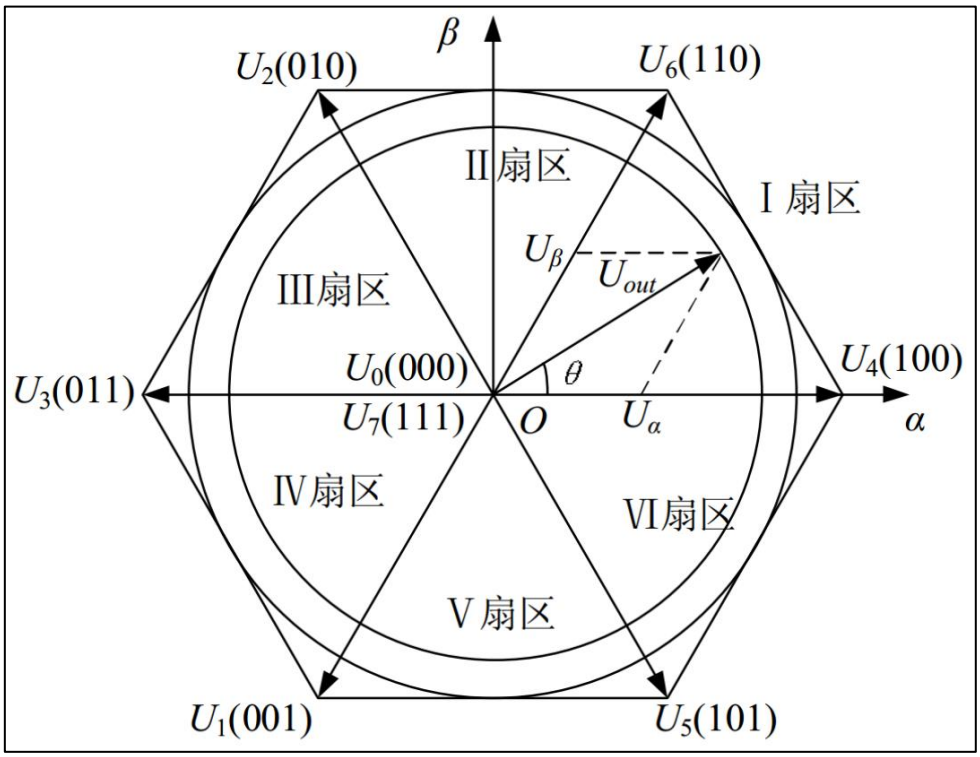


图 27.1.4.1.4 矢量扇区

在每一个扇区，用相邻俩电压矢量及零矢量按照伏秒平衡原则来合成每个扇区内的任意电压矢量。

$$U_{ref} * T_s = U_x * T_x + U_y * T_y + U_{deact} * T_0$$

其中，U_{ref}为期望电压矢量；T_s为开关周期；T_x、T_y、T_0分别为对应两个非零电压矢量U_x、U_y和零电压矢量U_0在一个采样周期的作用时间；其中U_{deact}可表示U_0或U_7两个零矢量。上式的意义是，矢量U_{ref}在T_s时间内所产生的积分效果值和U_x、U_y、U_0分别在时间T_x、T_y、T_0内产生的积分效果相加总和值相同。

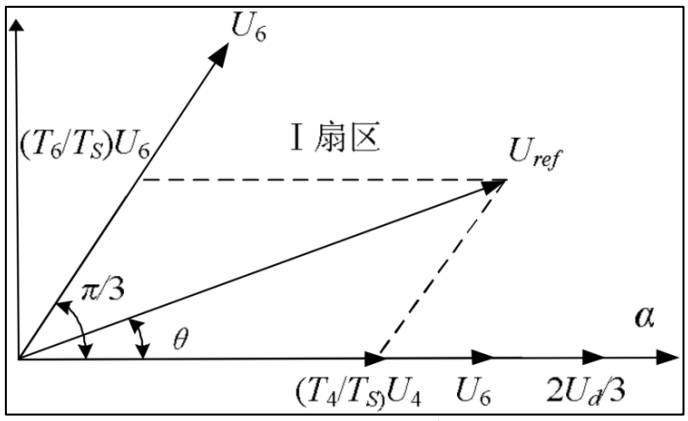


图 27.1.4.1.5 第一扇区

以第一扇区为例，则有：

$$U_{ref} * T_S = U_4 * T_4 + U_6 * T_6$$

令U_{ref}和U_4的夹角为θ，由正弦定理可得：

$$\begin{cases} |U_{ref}|\cos(\theta) = \dfrac{T_4}{T_S}|U_4| + \dfrac{T_6}{T_S}|U_6|\cos\left(\dfrac{\pi}{3}\right) = U_\alpha T_S \\ |U_{ref}|\sin(\theta) = \dfrac{T_6}{T_S}|U_6|\sin\left(\dfrac{\pi}{3}\right) = U_\beta T_S \end{cases}$$

整理得出

$$\begin{cases} U_\alpha T_S = \dfrac{2}{3}U_{dc}\left(T_4 + \dfrac{1}{2}T_6\right) \\ U_\beta T_S = \dfrac{2}{3}U_{dc}\left(\dfrac{\sqrt{3}}{2}T_6\right) \end{cases}$$

求得：

$$\begin{cases} T_4 = \dfrac{3U_\alpha T_S}{2U_{dc}} - \dfrac{1}{2}T_6 = \dfrac{3U_\alpha T_S}{2U_{dc}} - \dfrac{1}{2}\dfrac{\sqrt{3}U_\beta T_S}{U_{dc}} = \dfrac{\sqrt{3}T_S}{U_{dc}}\left(\dfrac{\sqrt{3}U_\alpha}{2} - \dfrac{U_\beta}{2}\right) \\ T_6 = \dfrac{\sqrt{3}U_\beta T_S}{U_{dc}} \\ T_7 = T_0 = \dfrac{T_S - T_4 - T_6}{2}(7periods) \end{cases}$$

现在一个周期内所有状态的持续时间我们都得到了，还差一个顺序，也就是各个状态切换的顺序。

以减少开关次数为目标，基本矢量作用顺序原则为：在每次开关状态转换时只改变其中一个相的开关状态，并且平均分配零矢量，使 PWM 对称、降低 PWM 的谐波分量。常用方式有 7 段式和 5 段式 SVPWM。我们这里主要讲解 7 段式 SVPWM。

以第Ⅰ扇区为例，三相调制波形在时间段 Ts 时段中如下图所示。

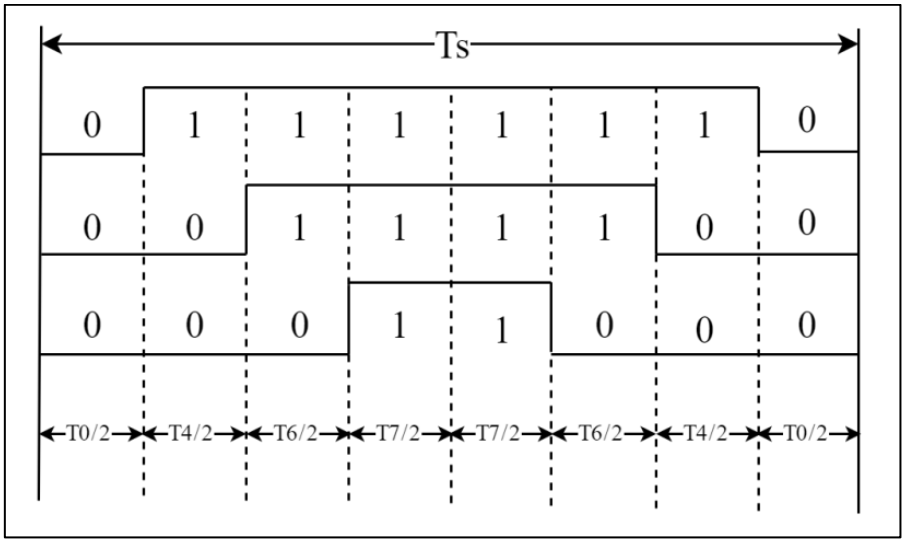


图 27.1.4.1.6 扇区 1 通电顺序

图中电压矢量出现先后顺序为U_0、U_4、U_6、U_7、U_6、U_4、U_0，各电压矢量的三相波形对应三相调制开关状态，零矢量过后U_{ref}角度增加γ，然后重新计算T_0、T_4、T_6、T_7，每一次载波周期合成一个新的矢量，角度逐渐增大，直至超过 60 度，然后进入Ⅱ扇区，开关顺序切换如下。

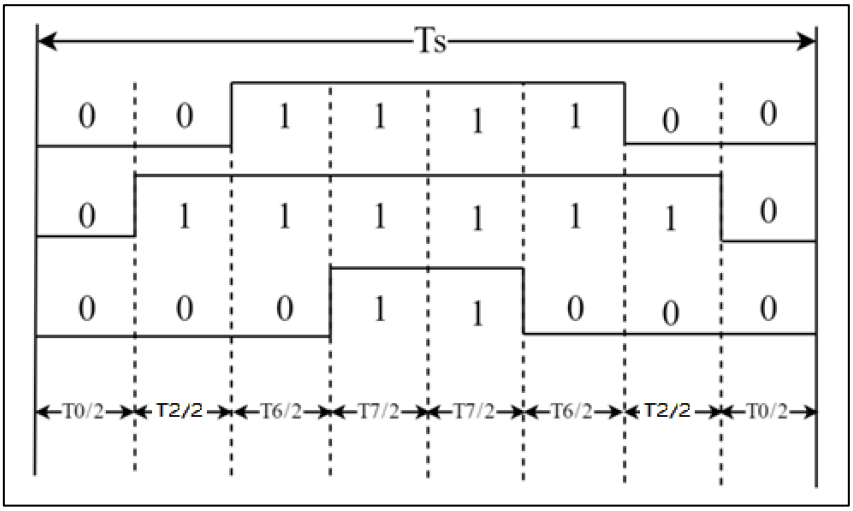


图 27.1.4.1.7 扇区 2 通电顺序

然后按照Ⅱ扇区开关切换顺序合成角度递增的新的矢量，直至超过 120° 范围，进入下一个扇区，直至旋转 360，然后依次循环。

通过上述推导可知，要实现信号调制，首先要知道参考电压矢量所在扇区位置，然后利用扇区相邻两个非零矢量和零矢量来合成，然后以旋转矢量为目标，在每个扇区内通过不同的开关顺序合成矢量来跟随目标旋转矢量。

空间矢量调制第一步需要判断空间电压矢量所处的扇区，假设合成电压矢量在Ⅰ扇区，则 0°<θ<60° ，由$tan(\theta) = U_\beta/U_a$，可知 θ= $arctan(U_\beta/U_\alpha)$，所以：

$$0° < arctan(U_\beta/U_\alpha) < 60°$$

且满足：

$$0 < tan\,\theta < \sqrt{3}$$

可求知，当合成电压矢量在Ⅰ扇区时的充要条件是：$U_\alpha > 0$, $U_\beta > 0, U_\beta/U_\alpha < \sqrt{3}$。其他扇区同理，可得：

扇区	落在扇区的充要条件
Ⅰ	$U_\alpha > 0$，$U_b > 0$且$U_\beta/U_\alpha < \sqrt{3}$
Ⅱ	$U_\alpha > 0$，且$U_\beta/\lvert U_\alpha\rvert > \sqrt{3}$
Ⅲ	$U_\alpha < 0$，$U_b > 0$且$-U_\beta/U_\alpha < \sqrt{3}$
Ⅳ	$U_\alpha < 0$，$U_b < 0$且$U_\beta/U_\alpha < \sqrt{3}$
Ⅴ	$U_b < 0$，且$-U_\beta/\lvert U_\alpha\rvert > \sqrt{3}$
Ⅵ	$U_\alpha > 0$，$U_b < 0$且$-U_\beta/U_\alpha < \sqrt{3}$

表 27.1.4.1.2 扇区判别条件

由上可以看出，电压矢量扇区由三个式子决定。

$$\begin{cases} U_1 = U_\beta \\ U_2 = \dfrac{\sqrt{3}}{2}U_\alpha - \dfrac{U_\beta}{2} \\ U_3 = -\dfrac{\sqrt{3}}{2}U_\alpha - \dfrac{U_\beta}{2} \end{cases}$$

整理得出

$$\begin{cases} U_\alpha T_S = \dfrac{2}{3}U_{dc}\left(T_4 + \dfrac{1}{2}T_6\right) \\ U_\beta T_S = \dfrac{2}{3}U_{dc}\left(\dfrac{\sqrt{3}}{2}T_6\right) \end{cases}$$

进行化简，得：

$$\begin{cases} T_4 = \dfrac{3U_\alpha T_S}{2U_{dc}} - \dfrac{1}{2}T_6 = \dfrac{3U_\alpha T_S}{2U_{dc}} - \dfrac{1}{2}\dfrac{\sqrt{3}U_\beta T_S}{U_{dc}} = \dfrac{\sqrt{3}T_S}{U_{dc}}\left(\dfrac{\sqrt{3}U_\alpha}{2} - \dfrac{U_\beta}{2}\right) = \dfrac{\sqrt{3}T_S}{U_{dc}}U_2 \\ T_6 = \dfrac{\sqrt{3}U_\beta T_S}{U_{dc}} = \dfrac{\sqrt{3}T_S}{U_{dc}}U_1 \\ T_7 = T_0 = \dfrac{T_S - T_4 - T_6}{2}(7periods) \\ T_7 = T_S - T_4 - T_6(5periods) \\ U_1 = U_\beta \\ U_2 = \dfrac{\sqrt{3}U_\alpha}{2} - \dfrac{U_\beta}{2} \end{cases}$$

至此我们得到了每一时刻所需要的空间电压矢量以及它们持续的时间，同理可计算得其他扇区基本空间矢量的作用时间。在处理器中赋值给对应通道的捕获比较寄存器产生相应的三个 PWM 波形，控制 MOS 管的开关，进而产生我们期望的电压、电流、力矩。

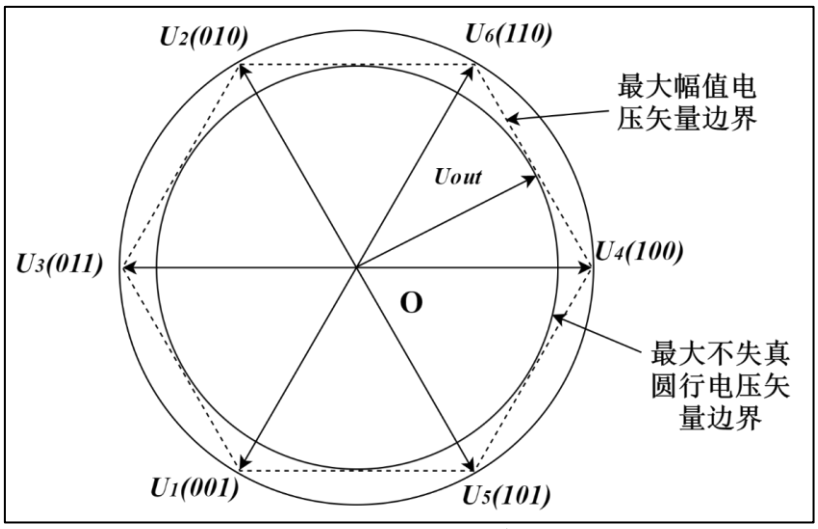


图 27.1.4.1.8 矢量边界

当俩零电压矢量作用时间为 0 时，一个 PWM 周期内非零电压作用时间最长，合成空间电压矢量幅值最大，其最大值不会超过正六边形边界，否则就会过调，输出电压波形就会失真，SVPWM 调制模式下能够输出的最大不失真旋转电压矢量为正六边形的内切圆，赋值为 $\frac{\sqrt{3}}{2}*\frac{2}{3}U_{dc}=\frac{\sqrt{3}}{3}U_{dc}$，相对 SPWM 最大为$\frac{1}{2}U_{dc}$，因此 SVPWM 方式电压利用率高出$\frac{\sqrt{3}}{3}/\frac{1}{2}-1=15.47\%$。

到目前为止，为了使电机旋转起来，还需要知道电机转子的位置，根据位置来合成旋转磁场矢量，使转子跟着旋转，对于一般的直流无刷电机，可以依据霍尔传感器来反馈转子位置，如果通过编码器来获取转子位置则更加精确，不过只使用编码器还需要启转前的校准，也就是找到转子对齐定子时刻的编码器位置，一般是通一段时间的小电流，将转子吸过去，然后将变化之后的编码器位置设置为零点，不过每次断电重启时都需要对齐一次，如果同时具备霍尔传感器和编码器，就可以相辅相成。而如果霍尔传感器和编码器都没有就有点麻烦了，和编码器方式一样，先通电流利用旋转磁场把转子带起来，通过电流的采集估测转子位置，转子位置采用观测器来估算，常用的有滑膜环观测器法、隆伯格观测器法、高频注入法等。

不同电流模式下的控制策略主要有：最大转矩电流比控制(保持转矩需求的情况下调节直轴交轴电流比例使电机定子输入电流最小)、id=0 控制(标贴式无刷电机常用策略)、弱磁控制(电机转速超过额定转速时通过减小直轴电流降低电枢电流从而减小反电动势以提供扭矩电流进一步加速)、cosφ=1 控制(功率因数恒为 1)等。

至此，FOC 的理论基础已经全部介绍完毕。下面接着介绍 FOC 所需的硬件平台。

27.2 FOC 硬件平台

27.2.1 驱动板硬件电路介绍

这部分内容跟无刷驱动板是完全一致的，所以这里不再赘述。简单介绍下驱动器板载功能，驱动器主要搭载了电流检测单元、过零检测单元、传感器反馈单元、反电动势检测单元、刹车单元等，可以有效的反馈驱动板电压、驱动板温度、三相电流及反电动势、过零信号、霍尔及编码器信号。大家如果想了解可以查看无刷专题驱动硬件介绍，更多关于 ATK-PD6010B 驱动板原理介绍请查看《ATK-PD6010B 直流无刷驱动板用户手册.pdf》。

1） 实物接线

下面我们将手把手地教大家搭建 FOC 硬件平台，这一部分内容十分重要，大家一定要掌握。首先我们需要准备一些材料，如图 27.2.1.1 所示：

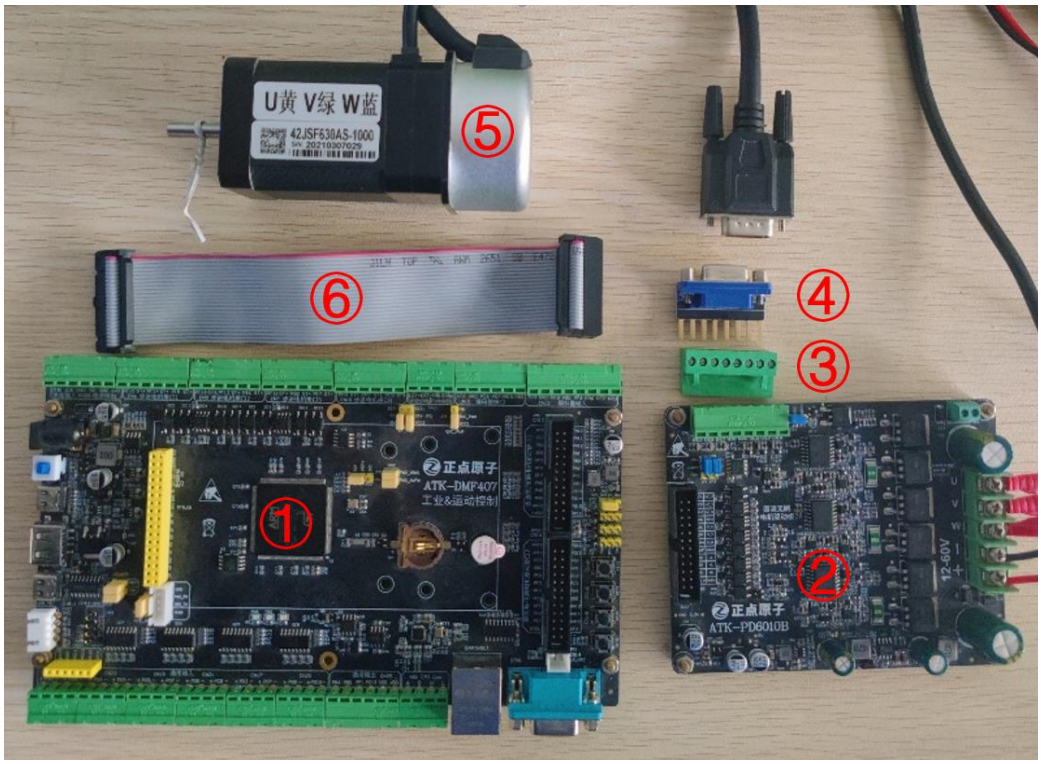


图 27.2.1.1 硬件平台搭建所需材料

图 27.2.1.1 中，不同的序号对应的材料如下：

① 电机开发板；
② 直流无刷驱动板；
③ 传感器接口 8pin 端子；
④ 电机插口转接板（PMSM 电机专用）；
⑤ PMSM 或者 BLDC 电机；
⑥ 专用 24pin 排线；

由于 PMSM 电机的传感器接口为 DB15，所以为了方便连接至驱动板我们使用转接板进行转接（如果是 BLDC 电机直接按照无刷专题的接法即可）。除了之外，我们还需要准备一个符合电机工作电压的直流电源，例如我们教程中所使用的无刷电机的工作电压是 24V，这里就准备一个 24V 的直流电源即可。

有了这些材料之后，我们接下来就进行实物的接线。这里以开发板的直流有刷/无刷电机驱动接口 1 为例，接线如图 27.2.1.2 所示：

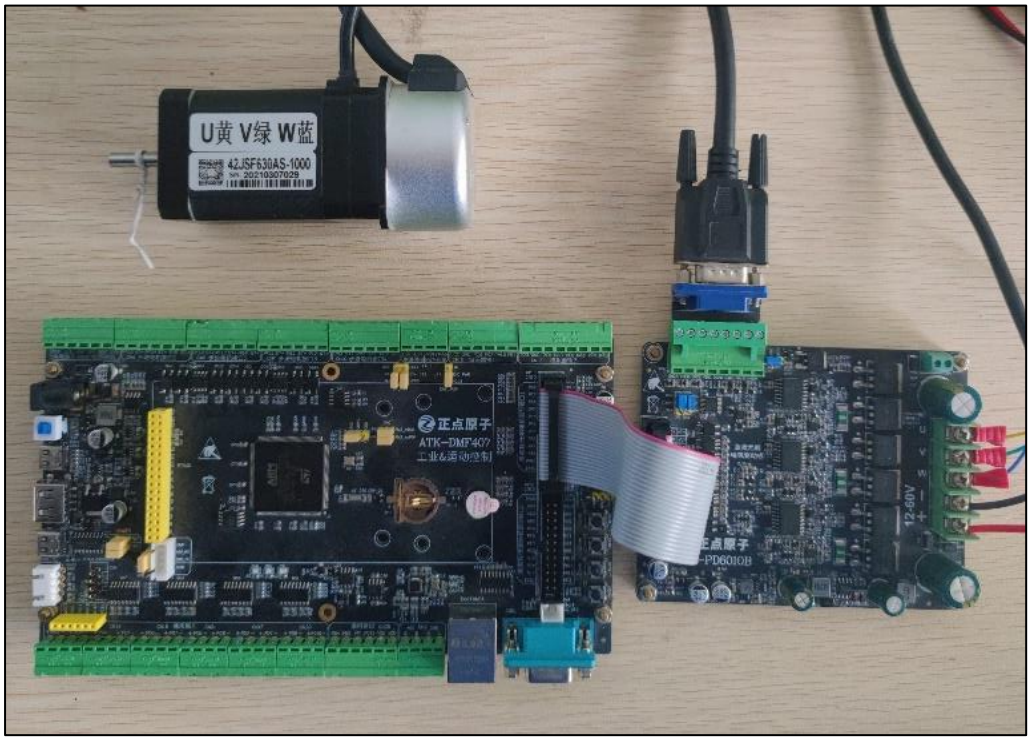


图 27.2.1.2 实物连接图

27.2.2 电机参数介绍

由于 BLDC 以及 PMSM 电机都是支持 FOC 控制的，所以下面我们列出了关于我们店铺这两款电机的一些技术参数，如下表：

电机类型	BLDC	PMSM
磁级数	4	8
相数	3	3
额定电压（VDC）	24	24
额定转速 （RPM）	3000	3000
额定电流 （Amps）	6.6	4
反电势（V/kRPM）	6.6	4.3
线电阻（Ohm）	0.4	1.02
线电感（mH）	0.8	0.59
编码器线	无	1000
霍尔安装角度	120°	120°
输出功率 （W）	105	64W
额定扭矩 （N-m）	0.33	0.2

表 27.2.2.1 店铺的 BLDC 与 PMSM 电机参数

27.3 FOC 开发工具

相信经过前面的学习，大家已经对 FOC 的理论基础部分有了一定的了解，但是想要实现 FOC 算法部分还是有一定的难度，所以我们可以借助 ST 提供的 FOC 开发套件——“X-CUBE-

MCSDK”，来帮助我们生成 FOC 控制代码 。

X-CUBE-MCSDK：ST 推出的电机控制软件开发套件。其中包括永磁同步电机（PMSM）固件库（FOC 控制）以及 STM32 电机控制工作台（用于配置 FOC 固件库参数）。如下：

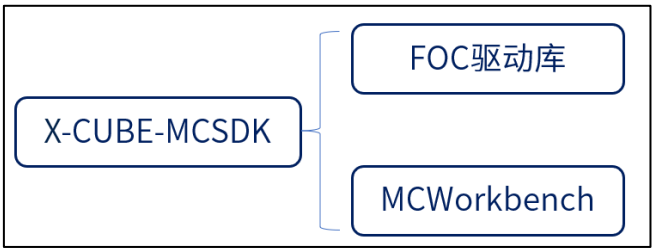


图 27.3.1 X-CUBE-MCSDK 组件

其中 FOC 驱动库就是 FOC 的核心算法实现，MCWorkbench 为一款 PC 软件，用于配置 FOC 固件库参数。

27.3.1 Motor Control Workbench 简介

Motor Control Workbench 是一款 PC 软件，它可以大大减少 STM32 PMSM FOC 固件配置所需的设计工作量和时间。用户通过 GUI 生成项目文件，并根据应用程序的需要，初始化对应的库。而且可以对所使用算法的某些变量实时监控和更改。

27.3.2 Motor Control Workbench 获取与安装

前面介绍了很多，接着我们开始来安装 FOC 开发所需的这些软件，首先安装“Motor Control Workbench”软件。它的软件安装包获取方法有两种：

1、在 ST 的官网：https://www.st.com 中直接搜索关键词“MC SDK”，如下图 27.3.2.1 所示：

2、可在光盘直接找到软件并安装，路径：“A 盘\6，软件资料\3，MCSDK”。

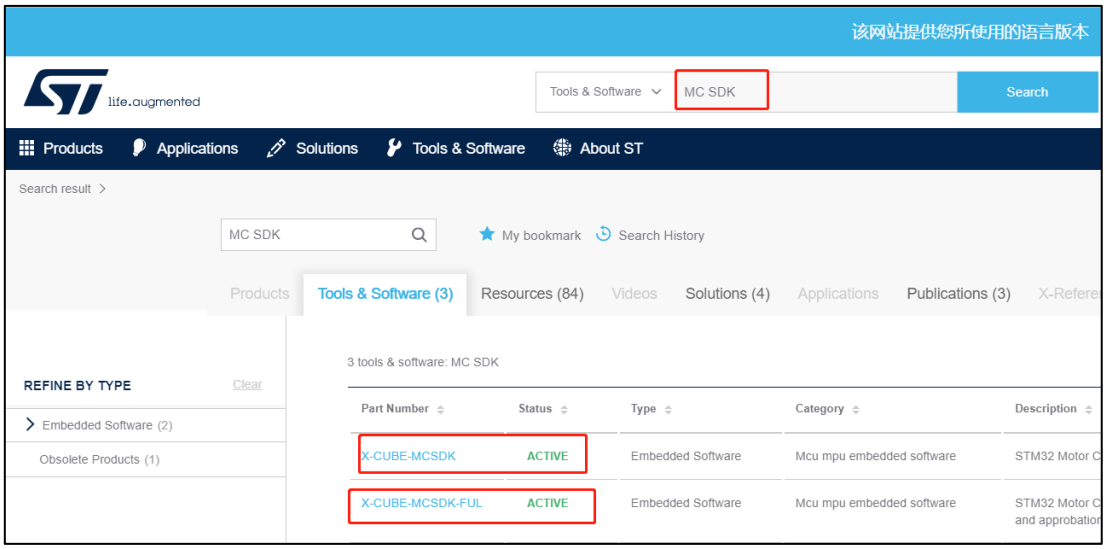


图 27.3.2.1 搜索结果

可以看到上图共有两种版本： X-CUBE-MCSDK 和 X-CUBE-MCSDK-FUL，其中 X-CUBE-MCSDK 在一些核心的算法使用 lib 的形式提供给用户，可以直接下载； X-CUBE-MCSDK-FUL 则是完整的源码，需要官方验证用户信息后才可下载。我们资料已提供完整版文件 X-CUBE-MCSDK-FUL，可在路径“电机 A 盘\6，软件资料\3，MCSDK”中获取。安装过程很简单双击 X-CUBE-MCSDK-FUL_5.4.4.exe 的安装包，之后只需要一直点击下一步即可。**注意安装路径一定不能有中文!!!**

安装完成后会在桌面生成 MotorControlWorkbench 5.4.4 和 Motor Profiler 5.4.4 两个软件， 其中 Motor Profiler 5.4.4 是用于自动测量电机参数的软件，不过该软件针对的是使用 ST 相关的主板和电机驱动板，这里就不介绍该软件的使用方法。 MotorControl Workbench

5.4.4 才是我们需要的软件，可以使用该软件配置电机驱动板等参数后并生成源代码。

完成了 X-CUBE-MCSDK-FUL 的安装，接着我们还需要安装 STM32CubeMX，因为使用 X-CUBE-MCSDK-FUL 配置完之后，在生成代码时需要使用到 STM32CubeMX。

27.3.3 STM32CubeMX 的获取与安装

由于 CubeMX 是运行在 JAVA 环境下的，所以安装 CubeMX 之前需要先安装 JAVA（JAVA8 及以上版本）。对于 Java 运行环境，大家可以到 Java 官网 www.java.com 下载最新的 Java 软件，也可以直接从我们光盘复制安装包，路径是“A 盘\6，软件资料\4，JAVA”，双击“chromeinstall-8u77.exe”即可进行安装。

图 27.3.3.1 java 安装包

这里大家需要注意，STM32CubeMX 的 Java 运行环境版本必须是 V1.7 及以上，如果你的电脑安装过 V1.7 以下版本，请先删掉后重新安装最新版本。

对于 Java 运行环境安装，我们这里就不做过多讲解，大家直接双击安装包，根据提示安装即可。安装完成之后提示界面如下图 27.3.3.2：

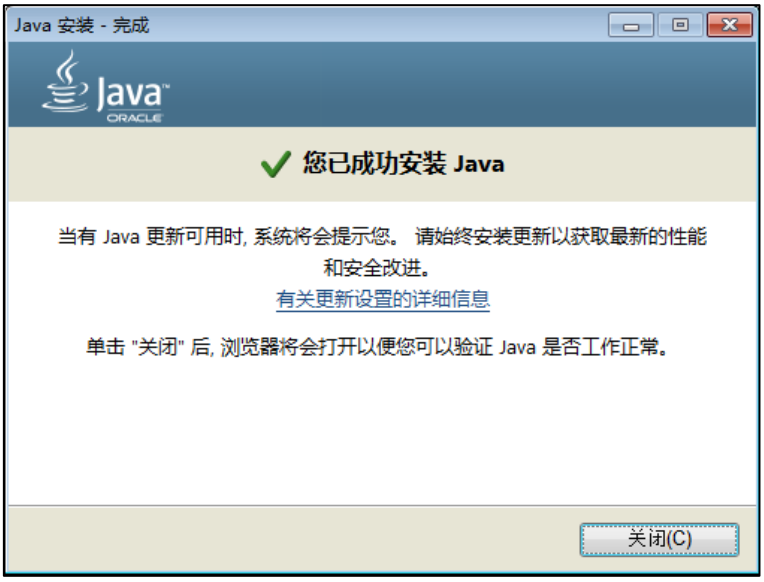


图 27.3.3.2 Java 安装成功提示界面

安装完 Java 运行环境之后，为了检测是否正常安装，我们可以打开 Windows 的命令输入框，输入：java –version 命令，如果显示 Java 版本信息，则安装成功。提示信息如图 27.3.3.3 所示。

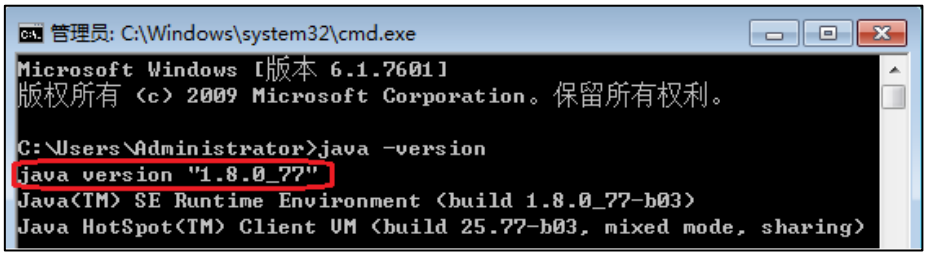


图 27.3.3.3 查看 Java 版本

JAVA 环境已经安装完成以后就可以安装 CubeMX，可以选择在线下载或者使用光盘软件资料的安装包直接安装，官网下载地址：https://www.st.com/en/development-

tools/stm32cubemx.html?sc=stm32cubemx，资料盘路径：“A 盘\6，软件资料\2，CubeMX”，双击“SetupSTM32CubeMX-5.5.0.exe”即可进行安装。

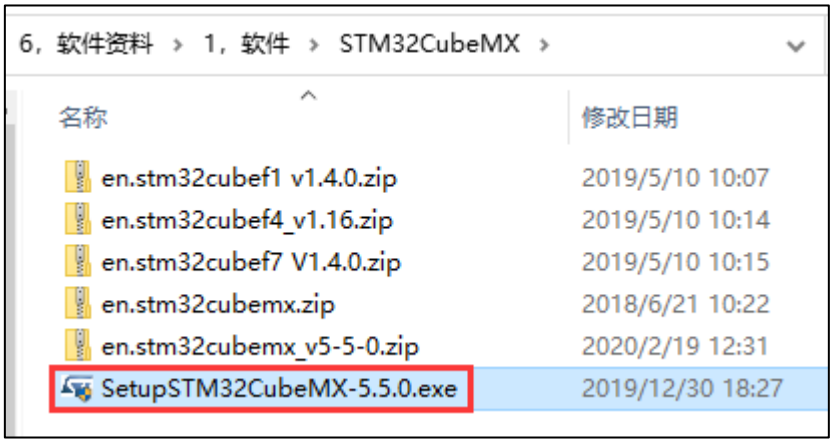


图 27.3.3.4 CubeMX 安装包

至此已经完成 CubeMX 软件的安装。**注意：请确保上述的三个软件安装路径均没有中文以及特殊字符！！**

27.4 FOC 例程创建

打开 ST Motor Control Workbench 软件，然后点击新建一个电机控制工程，如下。

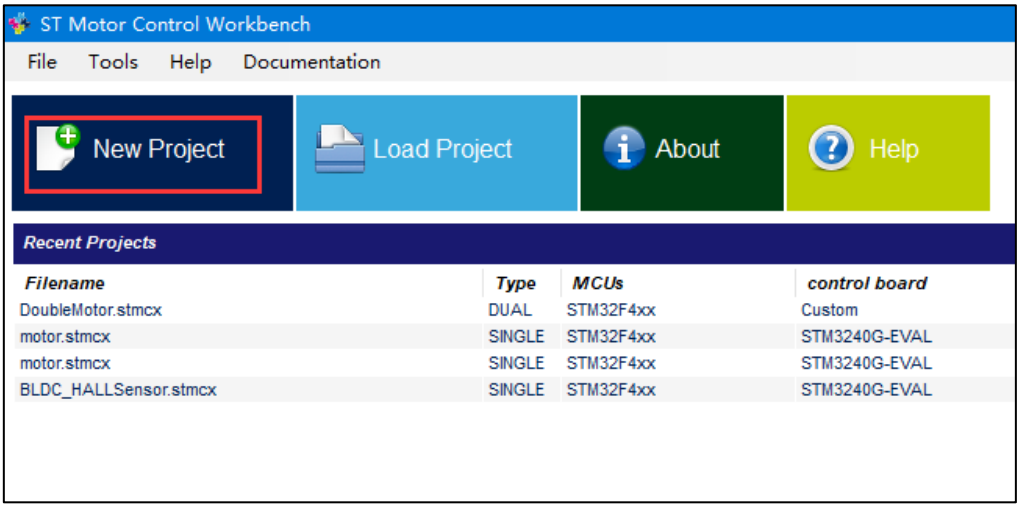


图 27.4.1 新建工程

选择应用类型，单/双电机控制以及硬件类型等，如下。

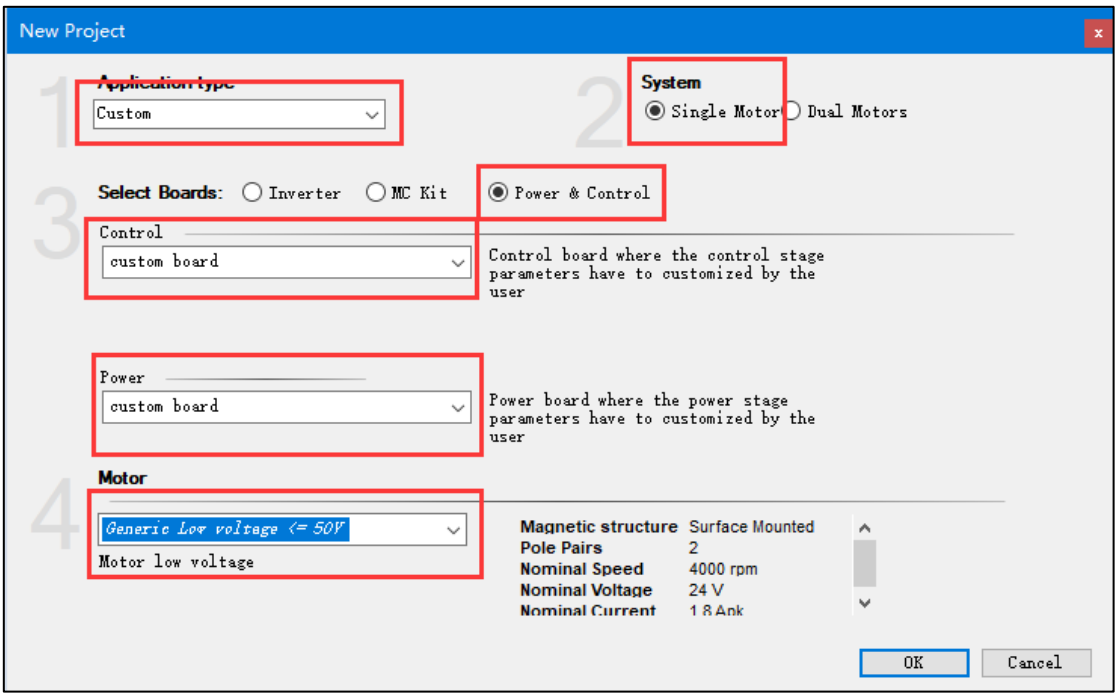


图 27.4.2 选择应用类型

针对不同负载类型的 FOC 可选应用类型，可选通用、泵、压缩机、空调、洗碗机、风扇、无人机，作为示例，我们直接选择通用类型即可。主设计界面如下。

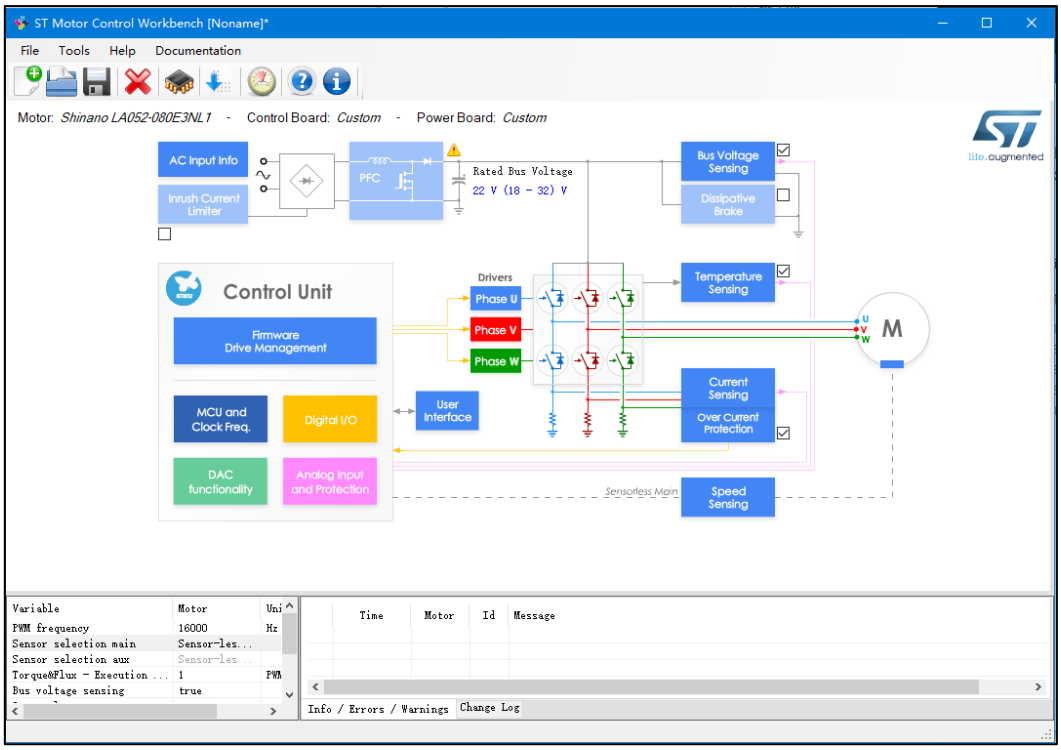


图 27.4.3 主界面

然后根据无刷驱动板输入电源范围设置母线电压，如下

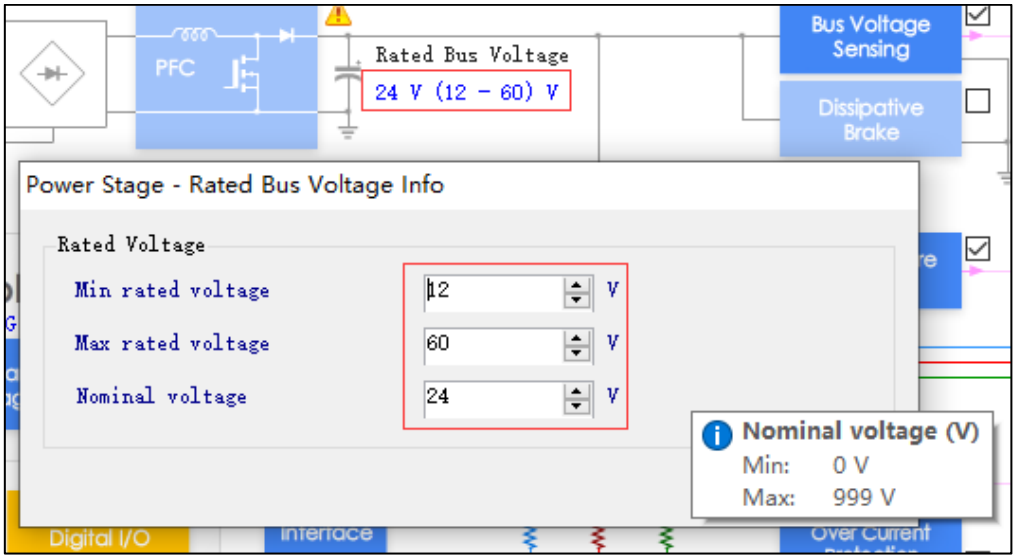


图 27.4.4 设置母线电压

然后修改控制器型号为板载的 STM32F407IG，如下。

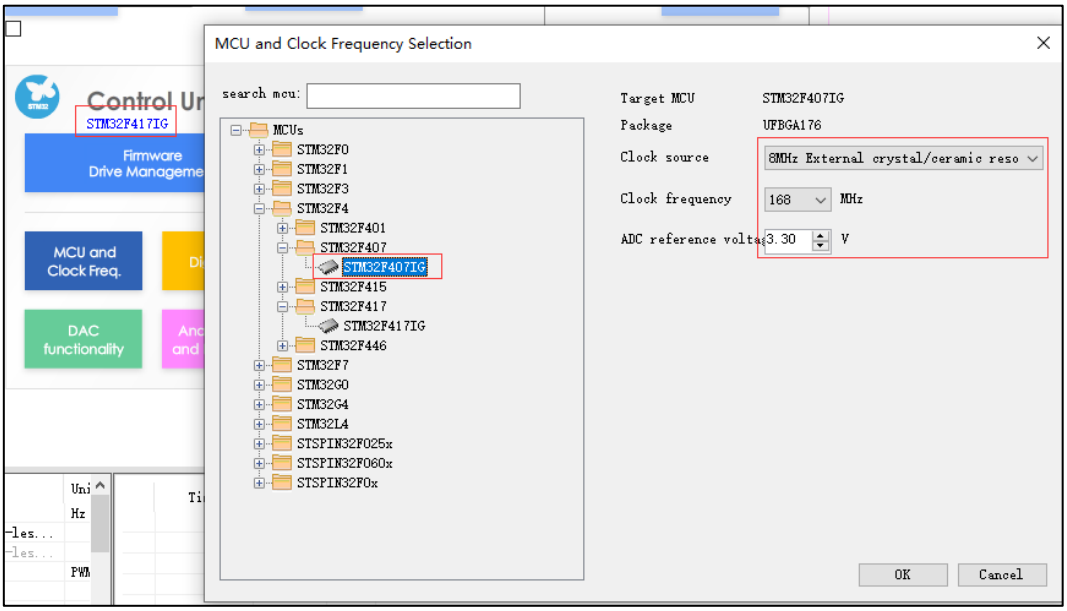


图 27.4.5 修改 MCU

接着点击电机参数配置，不同电机参数不同，需根据电机手册的参数填写，我们店铺的 PMSM/BLDC 电机技术参数可查看 27.2.2 小节。

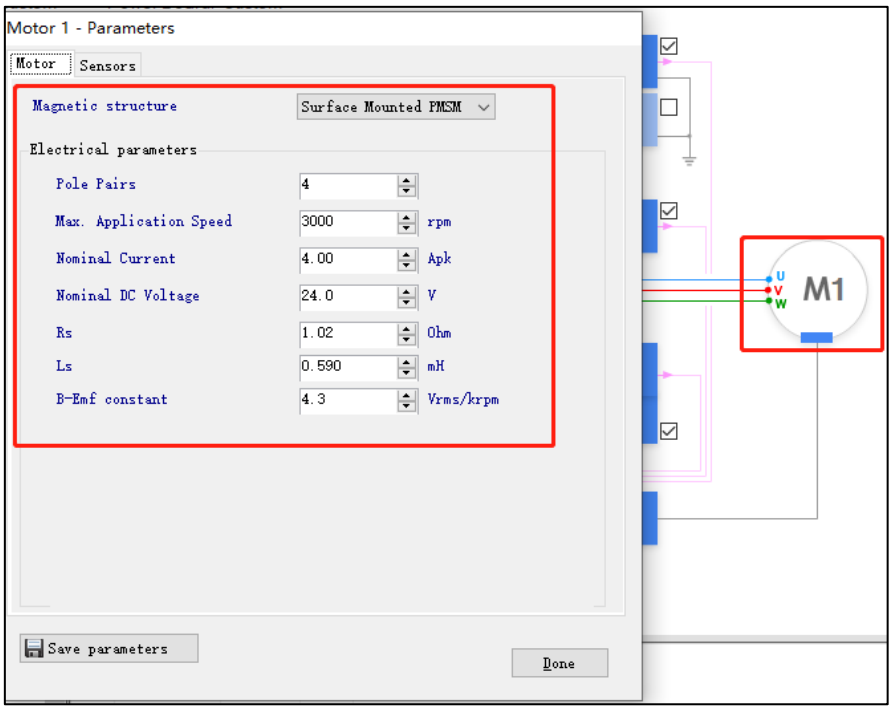


图 27.4.6 电机参数设置

传感器设置如下，这里使用的是霍尔传感器，大家可根据自己的电机硬件（传感器类型）选择编码器或者霍尔传感器，勾选对应选项即可。如需使用无感控制，则都不勾选。注意本店铺的 PMSM 电机内置霍尔传感器以及编码器；BLDC 内置霍尔传感器。

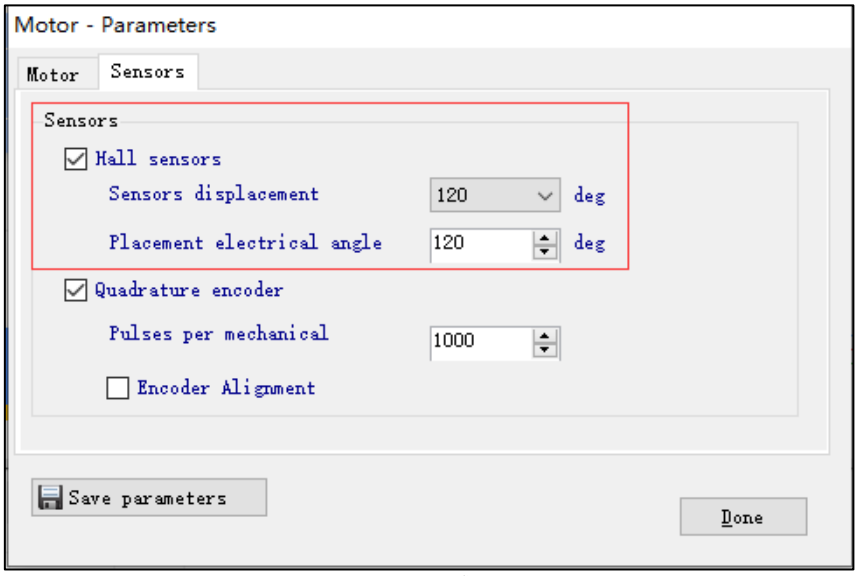


图 27.4.7 传感器设置

以霍尔反馈方式为例，速度反馈使用霍尔传感器采集，则速度传感器设置如下。

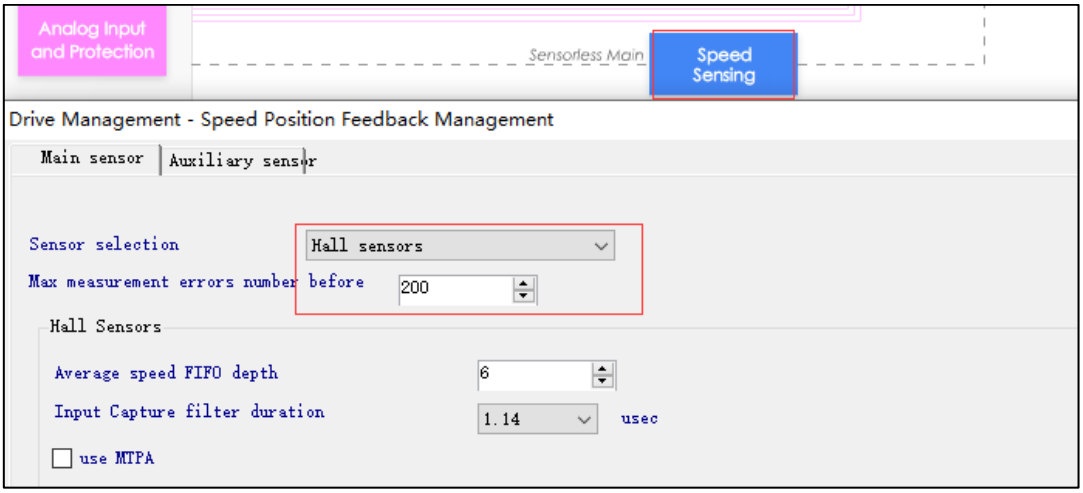


图 27.4.8 速度反馈设置

MOS 管设置，高电平有效，三相驱动配置完全一致，如下。

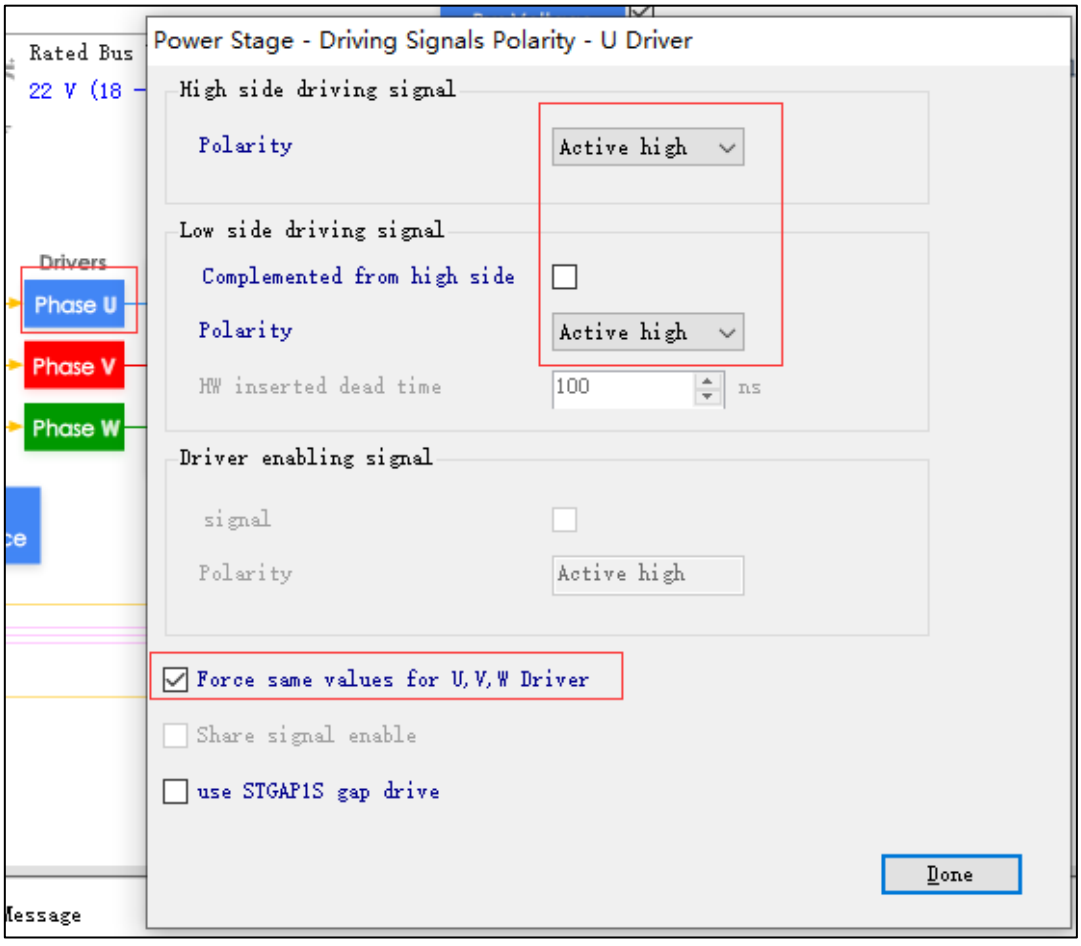


图 27.4.9 MOS 管设置

MOS 管最大开关频率和死区时间设置如下。

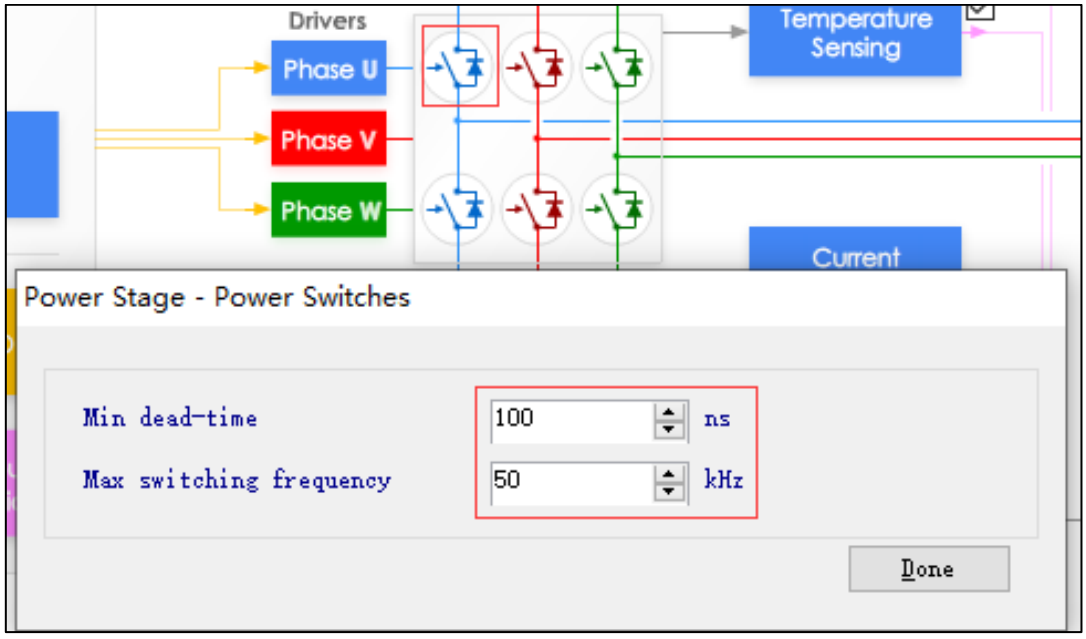


图 27.4.10 MOS 管频率设置

母线电压检测设置，根据无刷电机驱动板原理图设置如下。

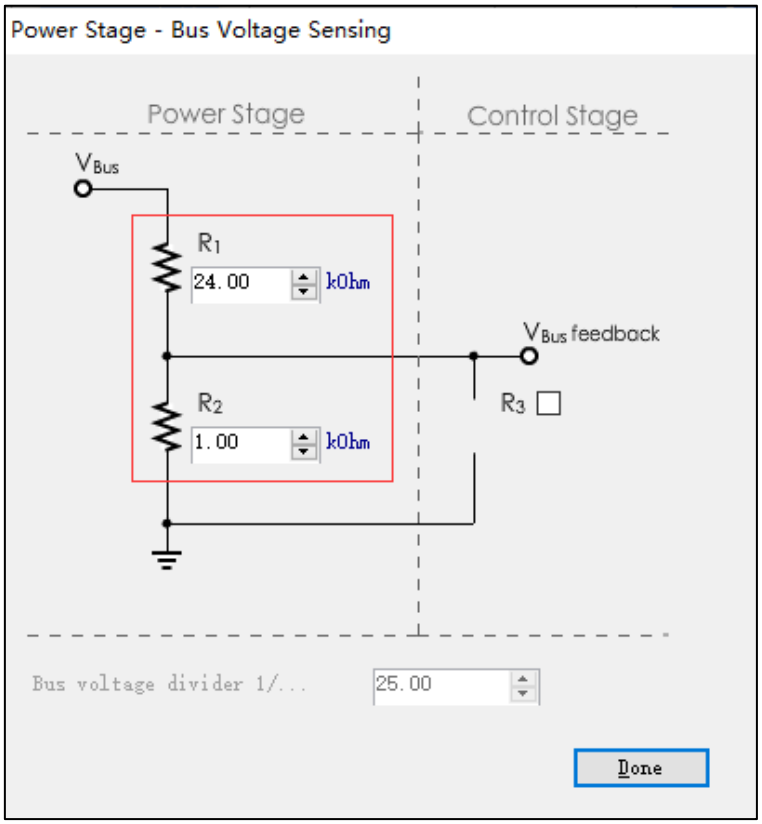


图 27.4.11 电压检测设置

温度传感器及保护设置如下，由于此设置只针对 ST 的温度传感器，所以直接默认设置即可。

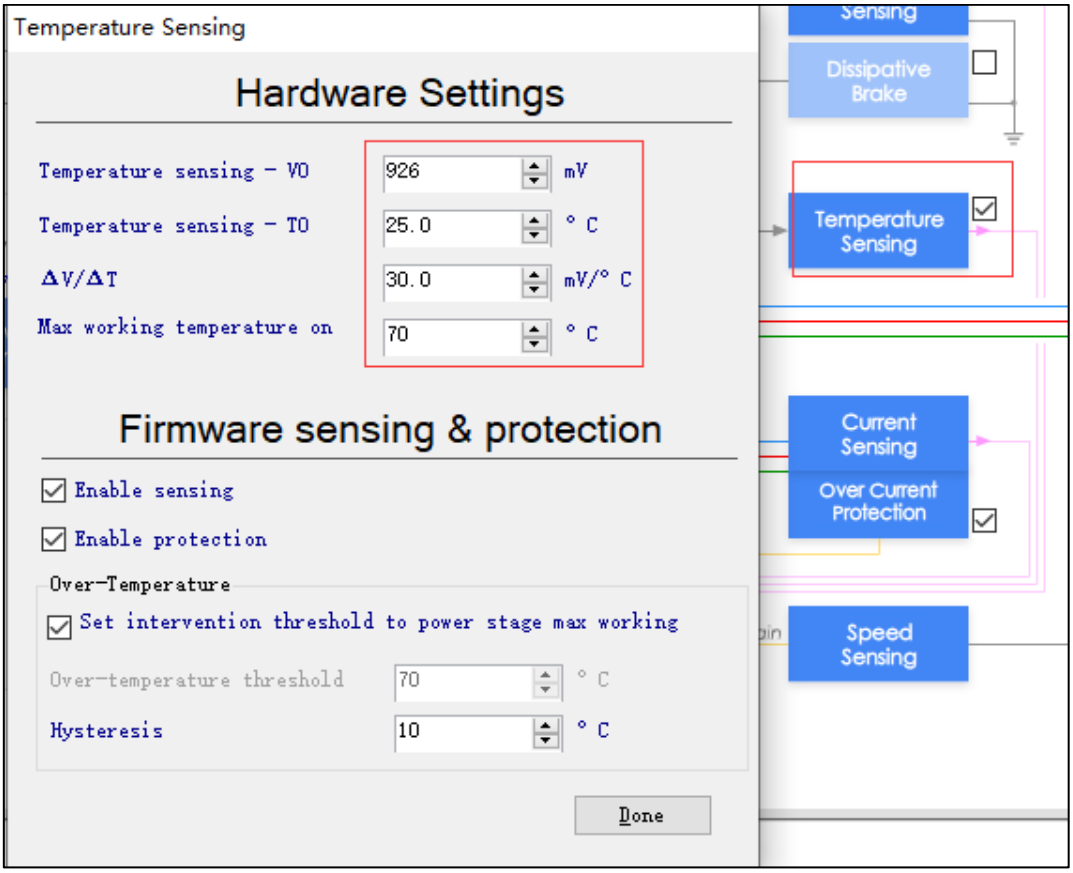


图 27.4.12 温度保护设置

电流传感器设置，需根据驱动板的电流采集硬件电路设置对应参数，我们驱动板使用的是 0.02Ω的采样电阻，差分电路的放大倍数为 6 倍，所以设置如下。

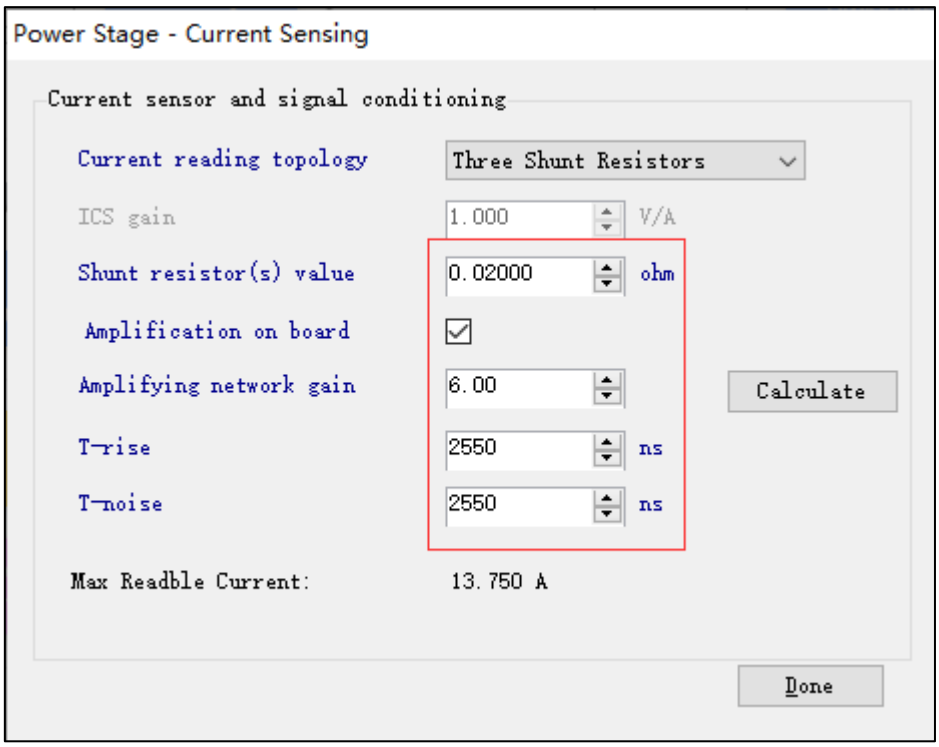


图 27.4.13 电流传感器设置

电流采集运放可按照驱动板参数设置，Calculate 选框主要协助计算放大倍数，鉴于我们

已知放大倍数，所以我们直接填写倍数值即可，Calculate 选框可以不设置，默认参数如下。

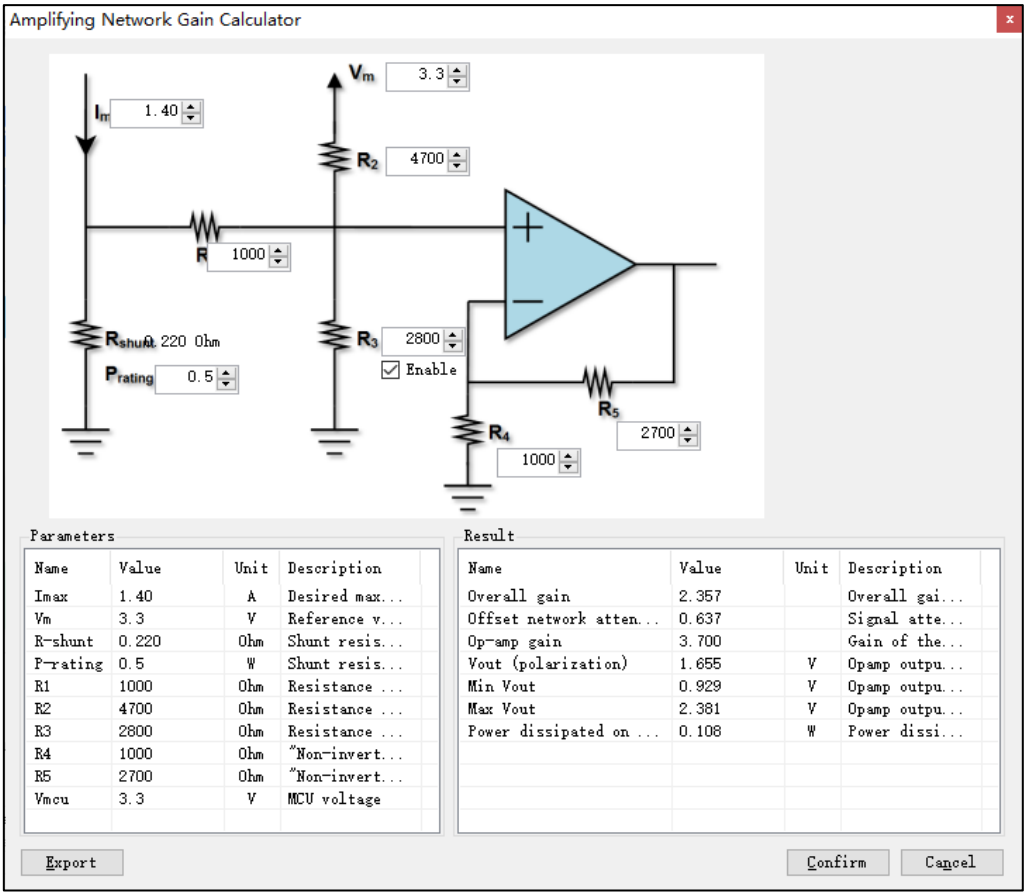


图 27.4.14 运放设置

右击 Fireware Driver Management 选择驱动设置，设置驱动参数设置如下。

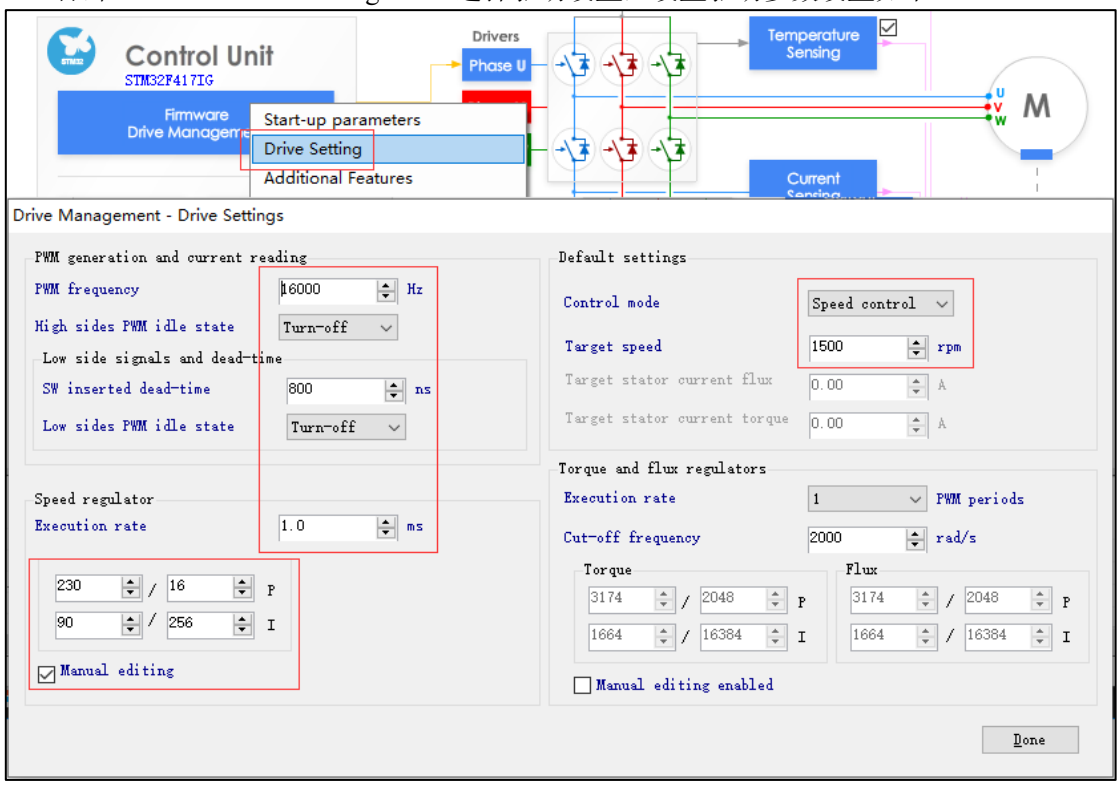


图 27.4.15 驱动设置

控制模式选择速度模式作为演示，PWM 频率选择 16KHz，左下侧速度环 PID 参数是我们测试过的数据，运转情况良好，为了直接生成即可运行我们就直接输入调试测试过的 PID 参数。接着设置电压保护如下。

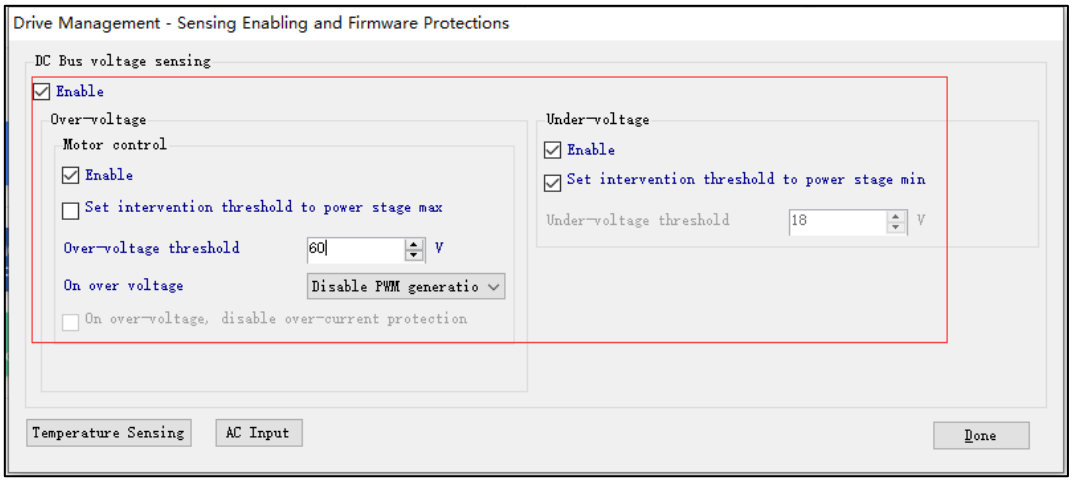


图 27.4.16 电压保护设置

用户接口设置如下。

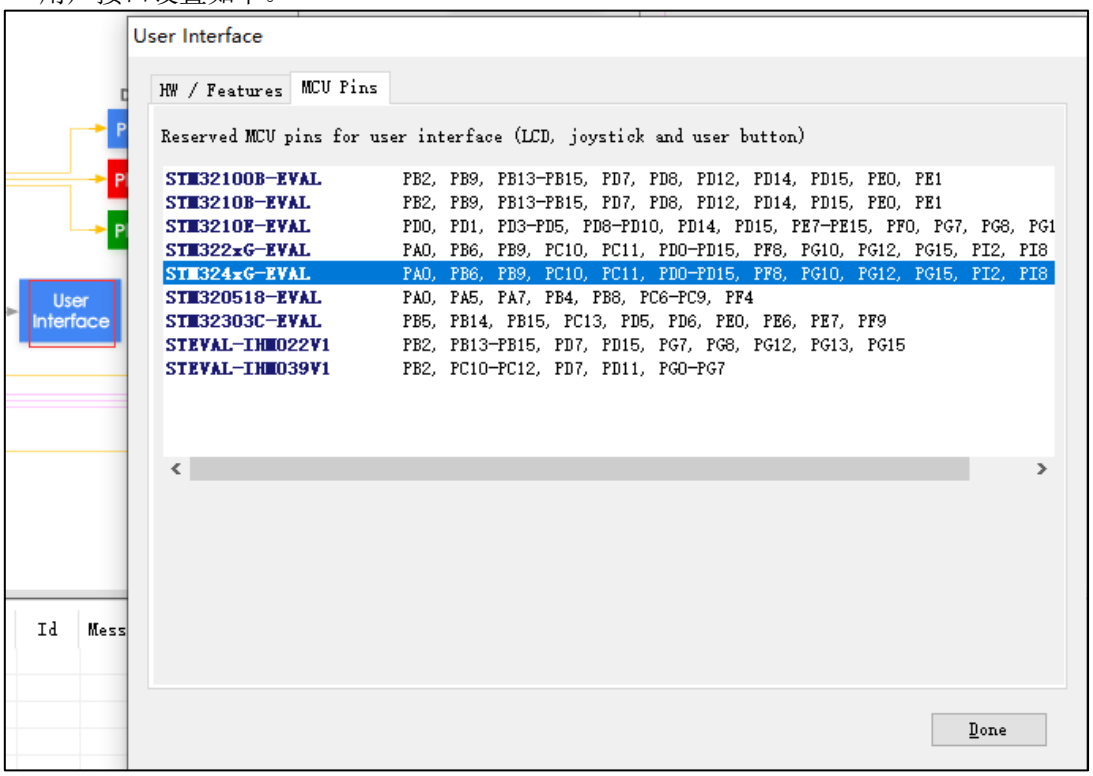


图 27.4.17 用户接口设置

数字输入输出设置根据开发板连接关系进行选择，以电机接口 1 为例，PWM 使用定时器 1，霍尔传感器使用定时器 5，调试串口使用串口 1，使用按键 KEY2 用于启停开发板，相应管脚设置如下。

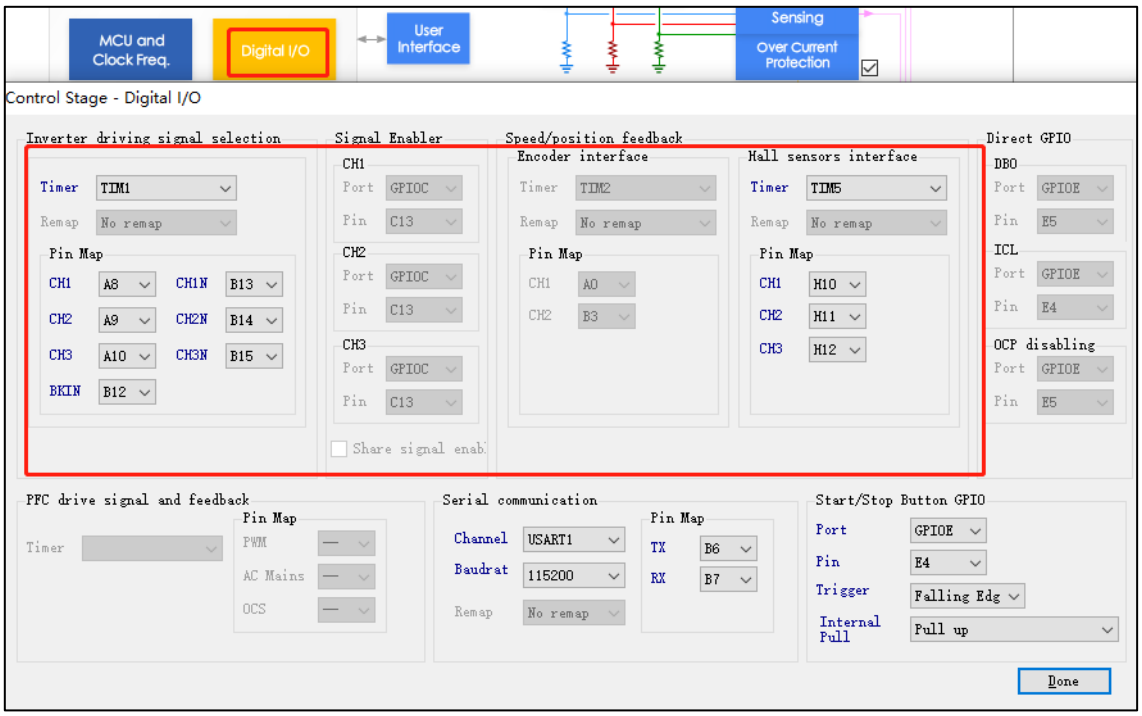


图 27.4.18 数字输入输出接口设置

电流采集模拟接口设置如下。

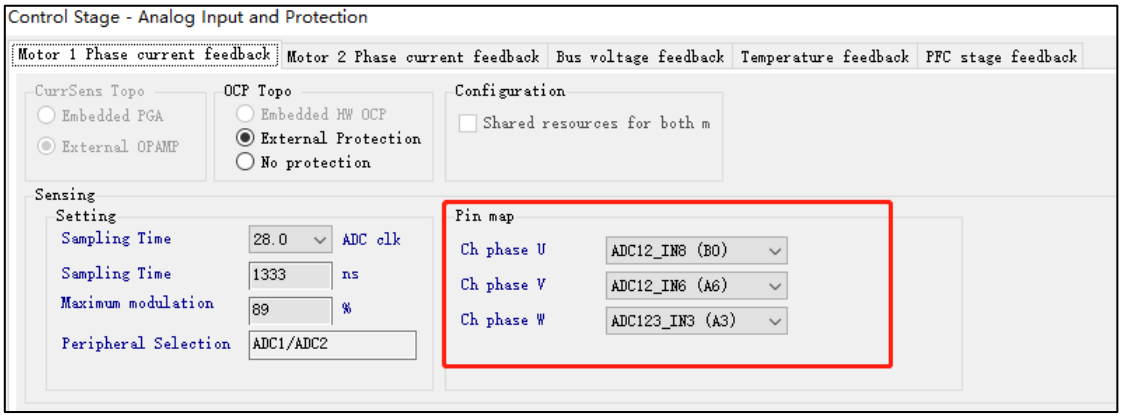


图 27.4.19 电流采集设置

电源电压采集模拟接口设置如下。

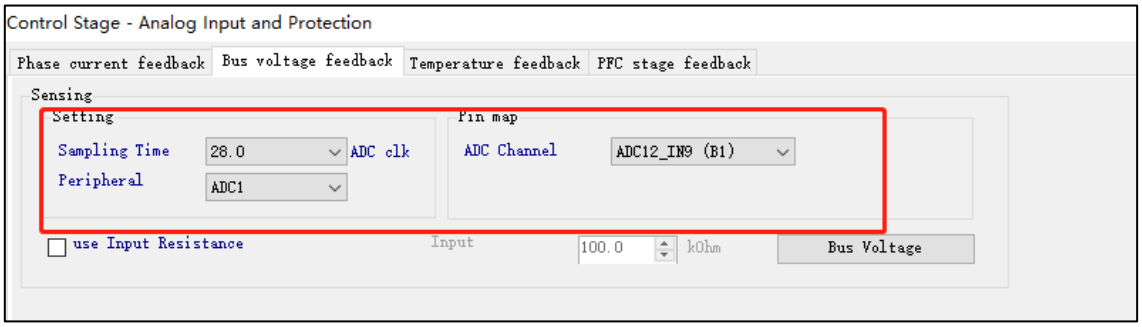


图 27.4.20 母线电压采集设置

温度采集模拟接口设置如下。

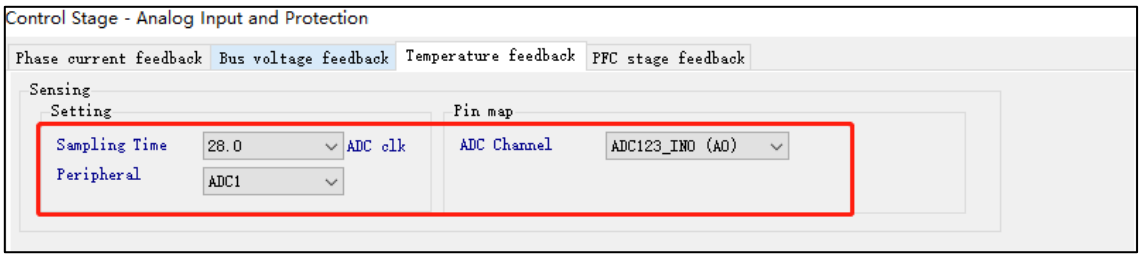


图 27.4.21 温度采集设置

DAC 设置用于调试时候观测 FOC 数据，例如将电机的电角度、交轴电流通过 DAC 方式输出，这样就可以使用示波器实时观测数据的变化，一般无需使用时直接选择不用即可，如下。

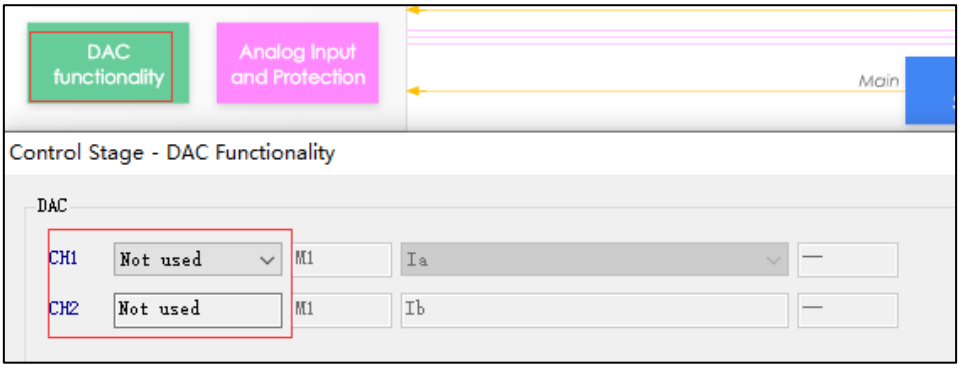


图 27.4.22 DAC 设置

配置完之后，就要检测 IO 分配是否有误，点击选项 ，IO 分配无误后显示 check ok。如下图：

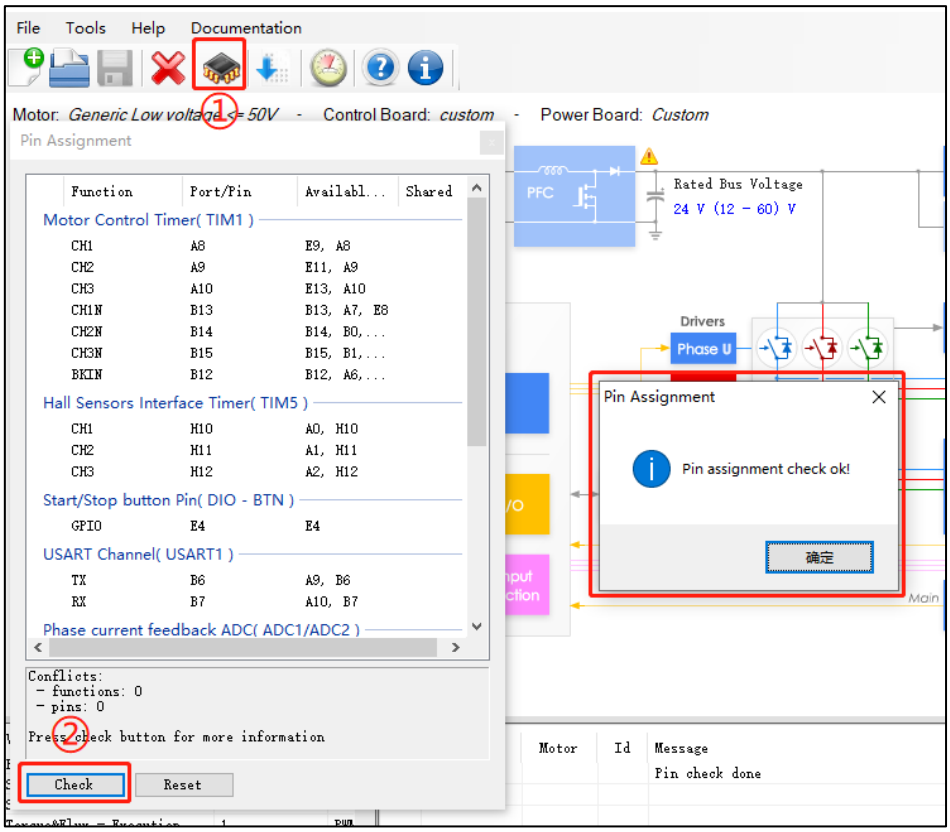


图 27.4.23 引脚检测

最后就可以保存工程，注意路径不能含有中文。

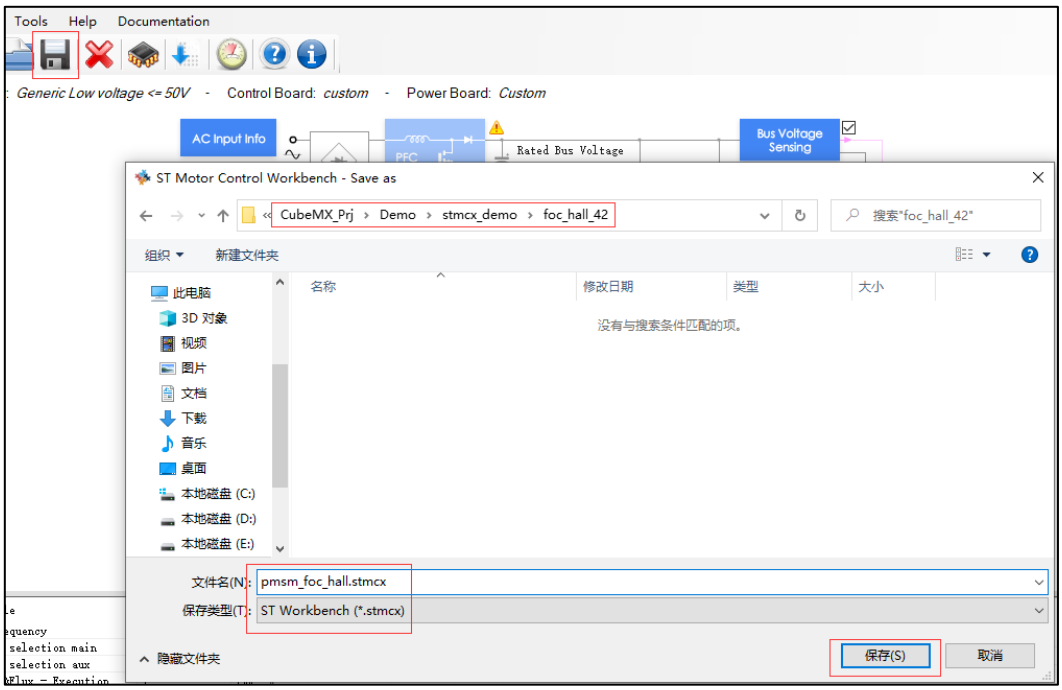


图 27.4.24 保存工程

点击 按钮，生成 CubeMX 工程，第一次点击生成，选择编译器平台，后续在该工程下进行的修改只需要更新即可。

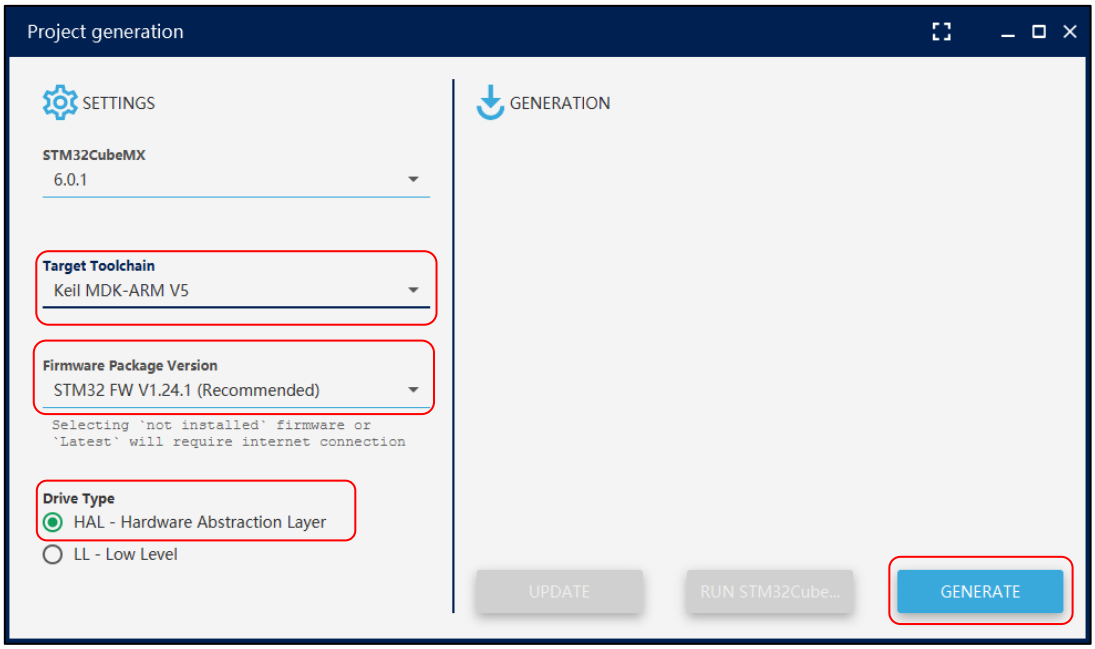


图 27.4.25 生成过程

开始生成工程，如下。

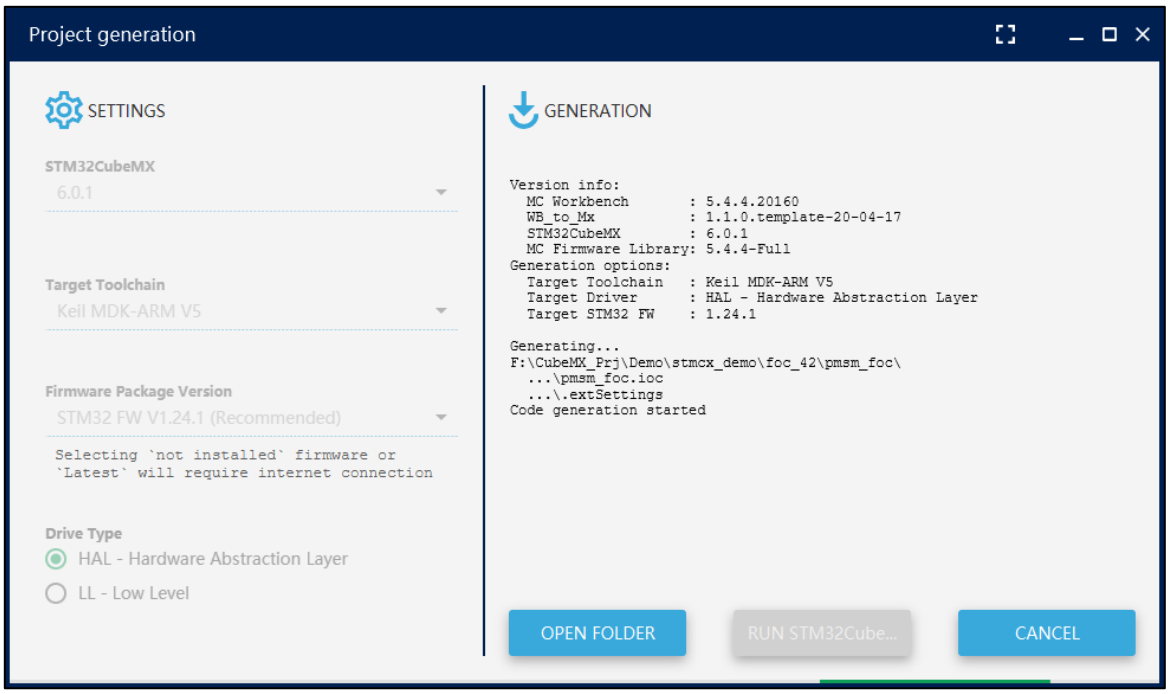


图 27.4.26 工程生成中

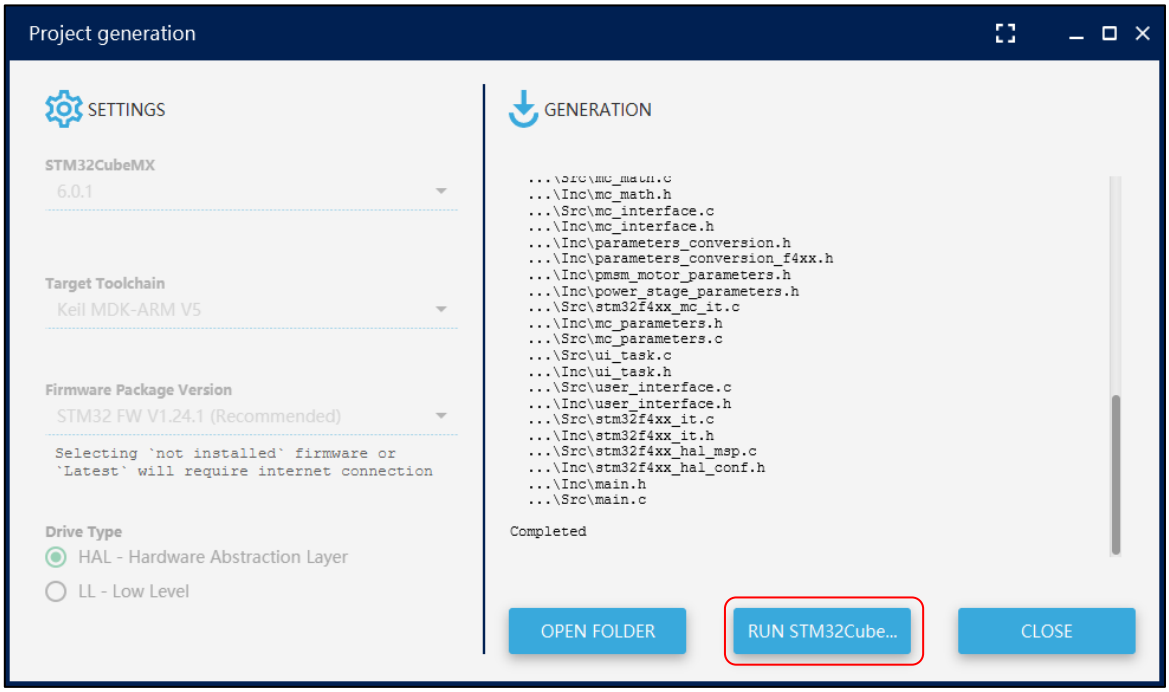


图 27.4.27 工程生成完成

生成结束后可以打开 CubeMX 添加自己的设置，由于无刷驱动板默认不输出，因此需要通过 CubeMX 打开 SHUTDOWN 引脚，并且配置为输出功能，点击 RUN STM32Cube…打开 CubeMX，设置关断引脚，如下。

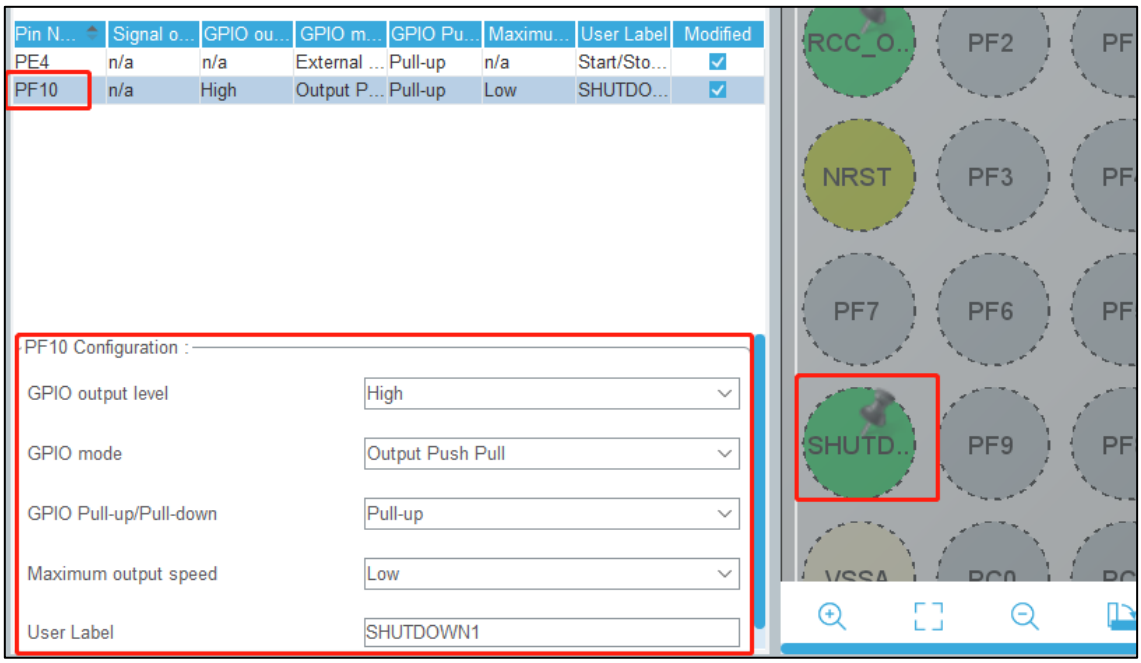


图 27.4.28 使能关断引脚功能

直接将 SHUTDOWN 引脚默认状态设置成拉高，即使能 SHUTDOWN 引脚，半桥芯片可正常输出，然后点击保存即可。

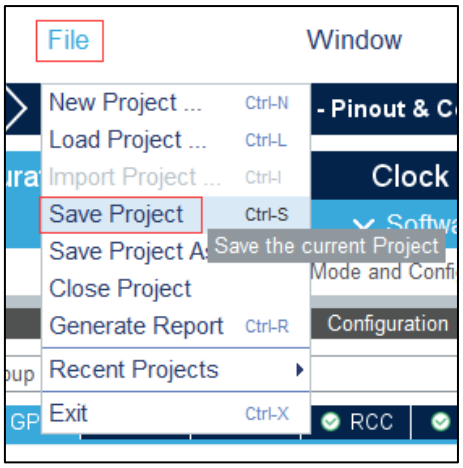


图 27.4.29 保存 CubeMX 工程

点击 CLOSE 关闭生成完成窗口，重新回到 MCWorkbench，点击 按钮，生成工程，由于工程文件已经存在，为了避免覆盖用户配置，直接点击 UPDATE 更新即可。

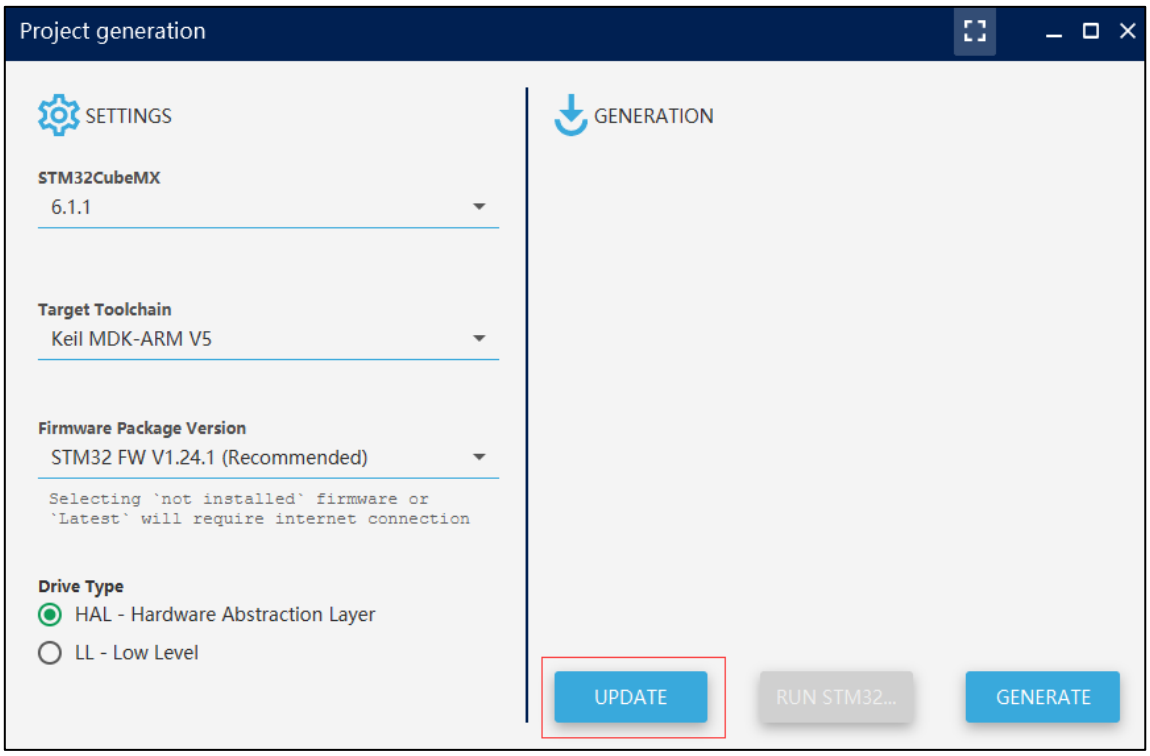


图 27.4.30 更新工程

更新完成后关闭窗口，打开前面保存项目路径的文件夹找到 MDK 工程并打开，可直接编译并将程序下载进开发板。

27.5 FOC 下载验证

编译上述创建好的 FOC 控制工程并下载程序到开发板，连接开发板和驱动板及电机。由于永磁同步电机传感器接口为 DB15 接口，可以使用转接模块连接，比较方便，实物连接如下。

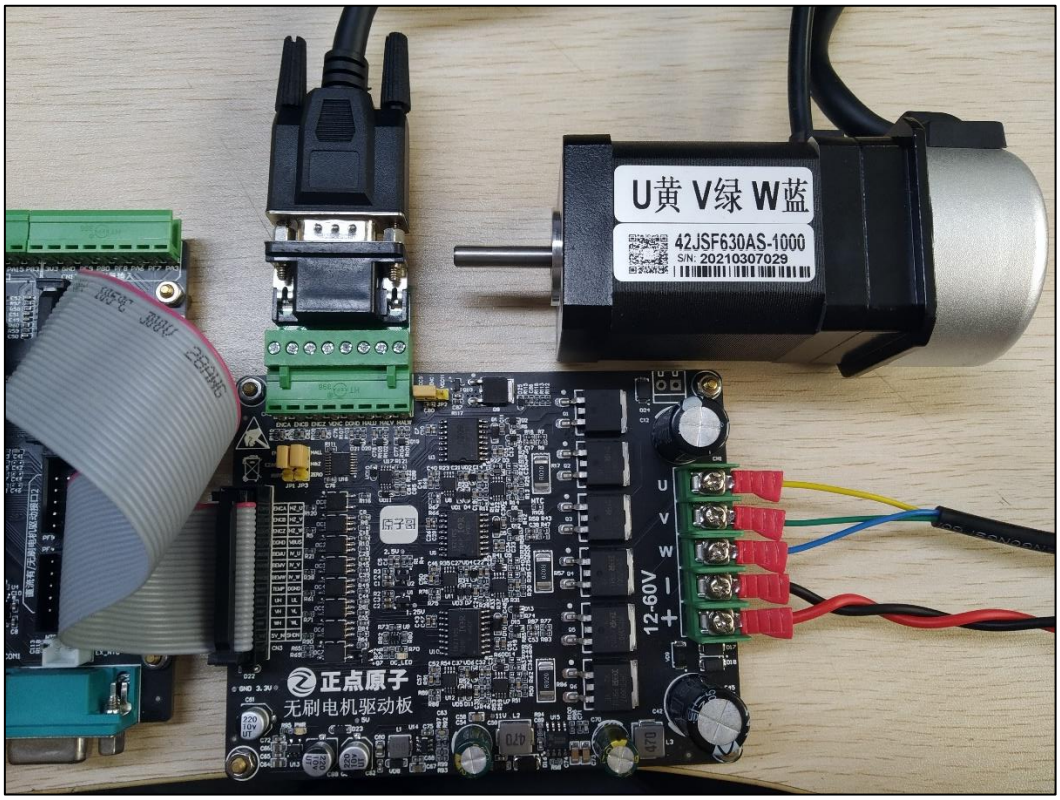


图 27.5.1 FOC 验证实物连接图

将开发板串口 1 用 USB 线连接至电脑。

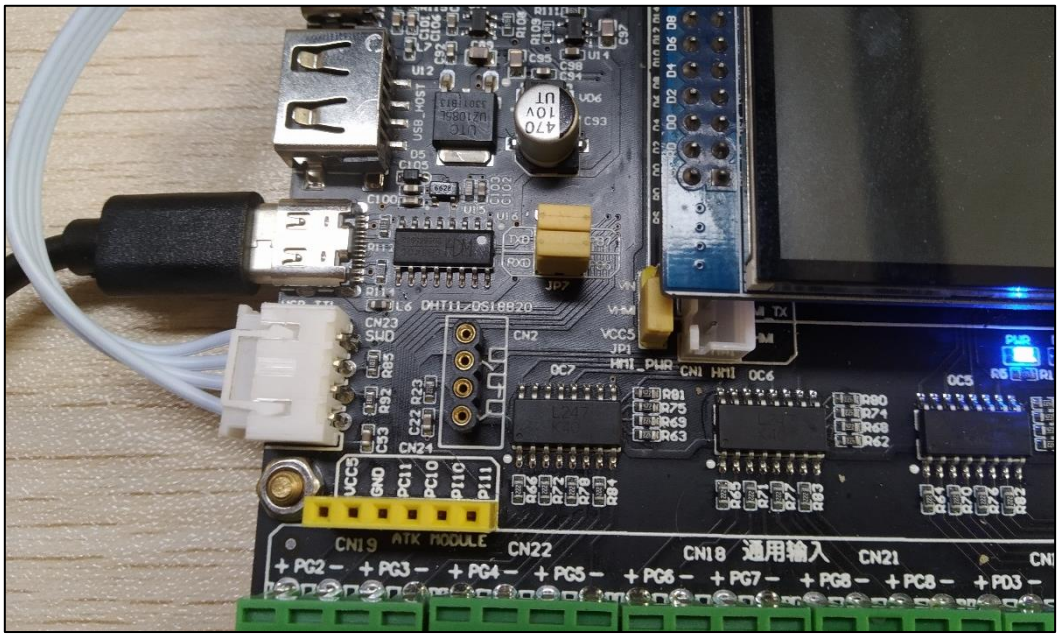


图 27.5.2 串口连接至电脑 USB

打开上位机的监控功能，如下。

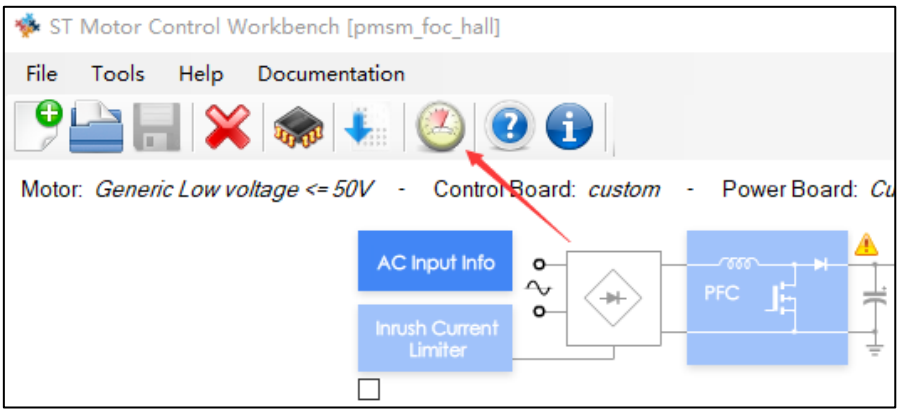


图 27.5.3 打开监控界面

选择正确的串口号并点击插头按钮连接到开发板，如下：

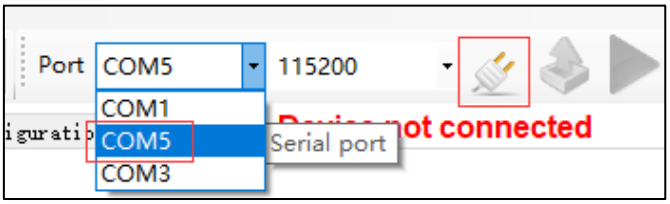


图 27.5.4 连接开发板

此时显示已连接状态，并且显示 MCSDK 版本，由于未连接驱动板电压，这时左侧状态栏可能会报低压警告，给驱动板通电并且点击右侧的 Fault Ask 按钮即可消除警告。

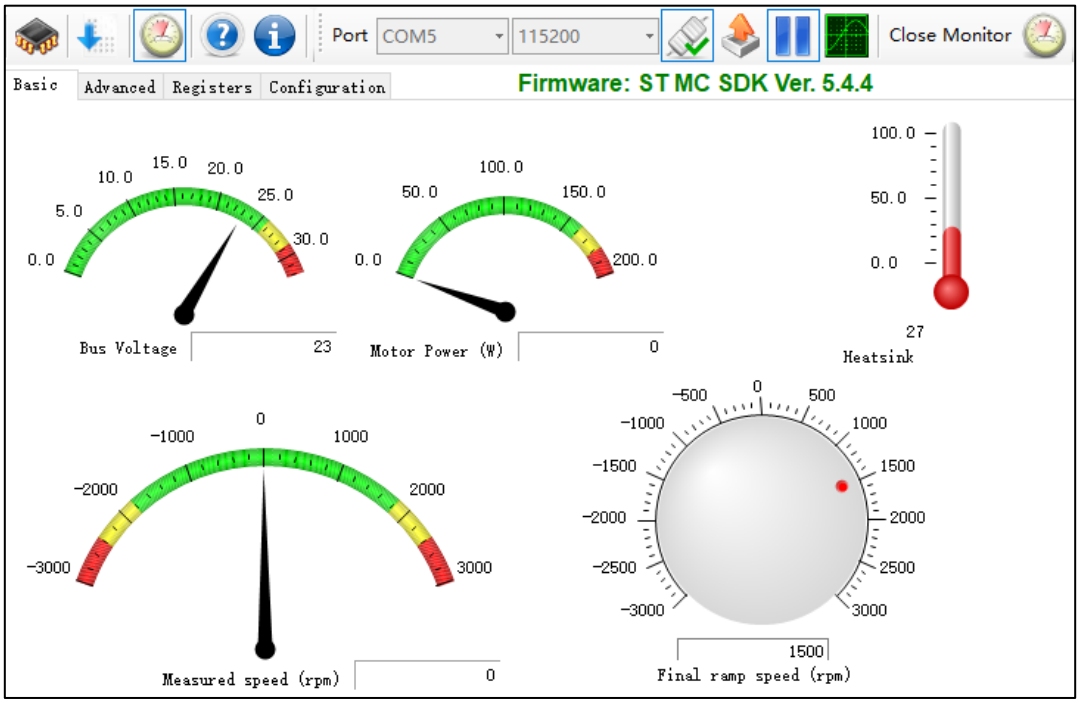


图 27.5.5 串口连接成功

点击右侧的 Start Motor 按钮即可启动电机，电机开始旋转，拖动界面中间的圆形旋钮即可调节转速，同时可以看到测量到的母线电压、温度、实际转速等信息。通过板子上的 KEY2 按键也可以控制电机的启动和停止。

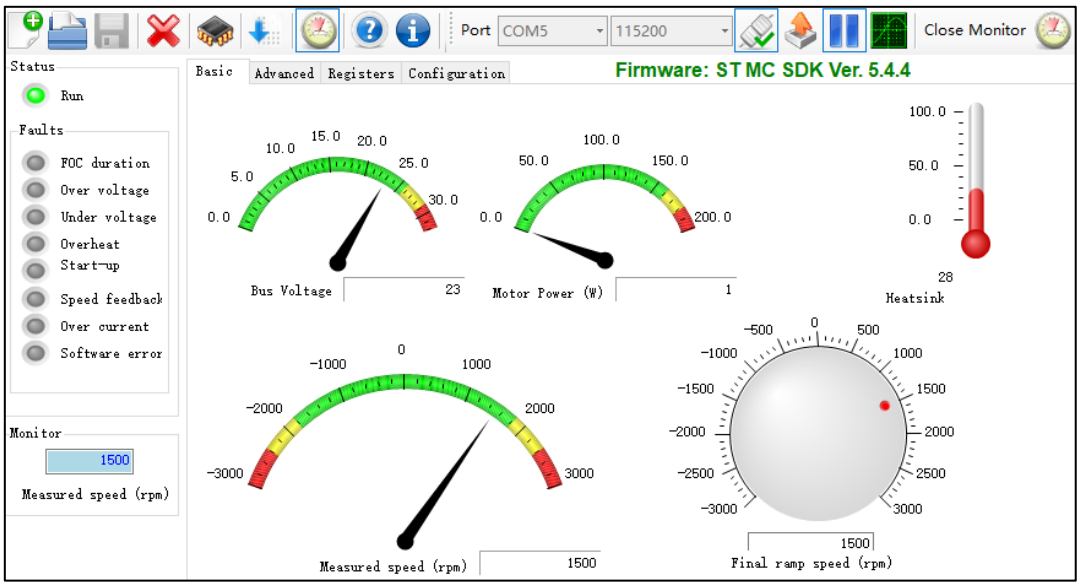


图 27.5.6 FOC 运转

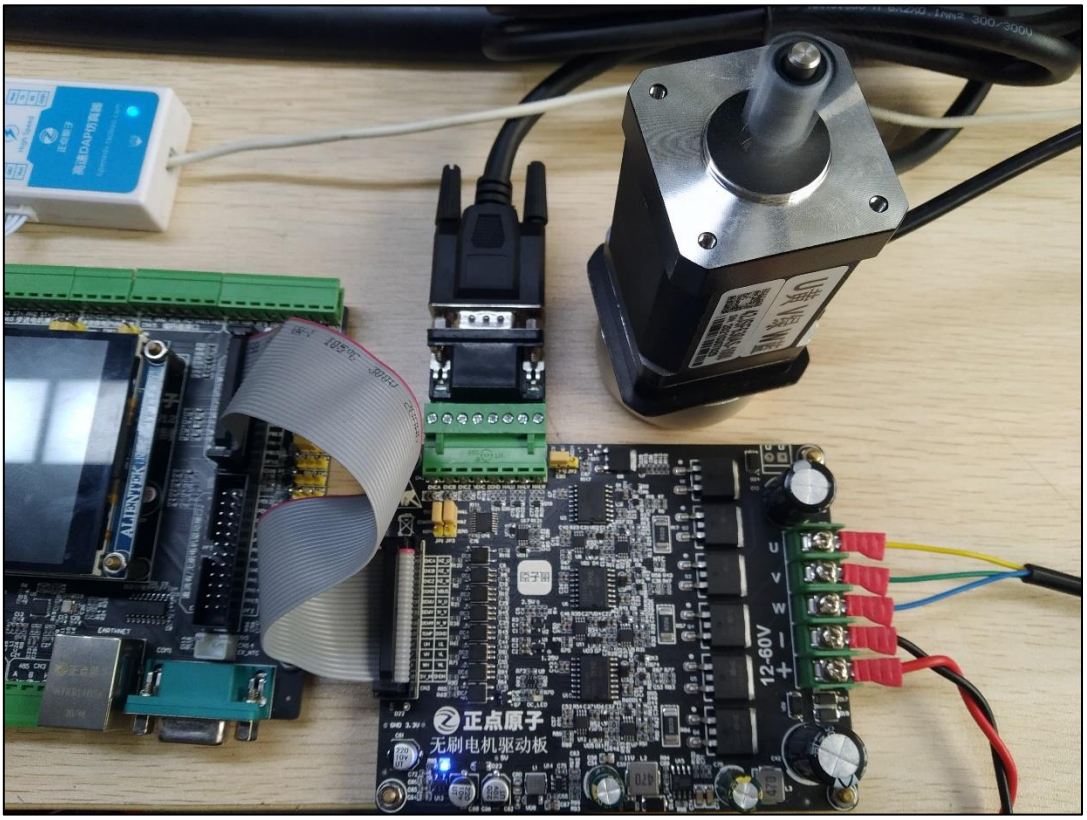


图 27.5.7 电机运行情况

点击 按钮可以打开折线图窗口，用于观测速度变化，如下。

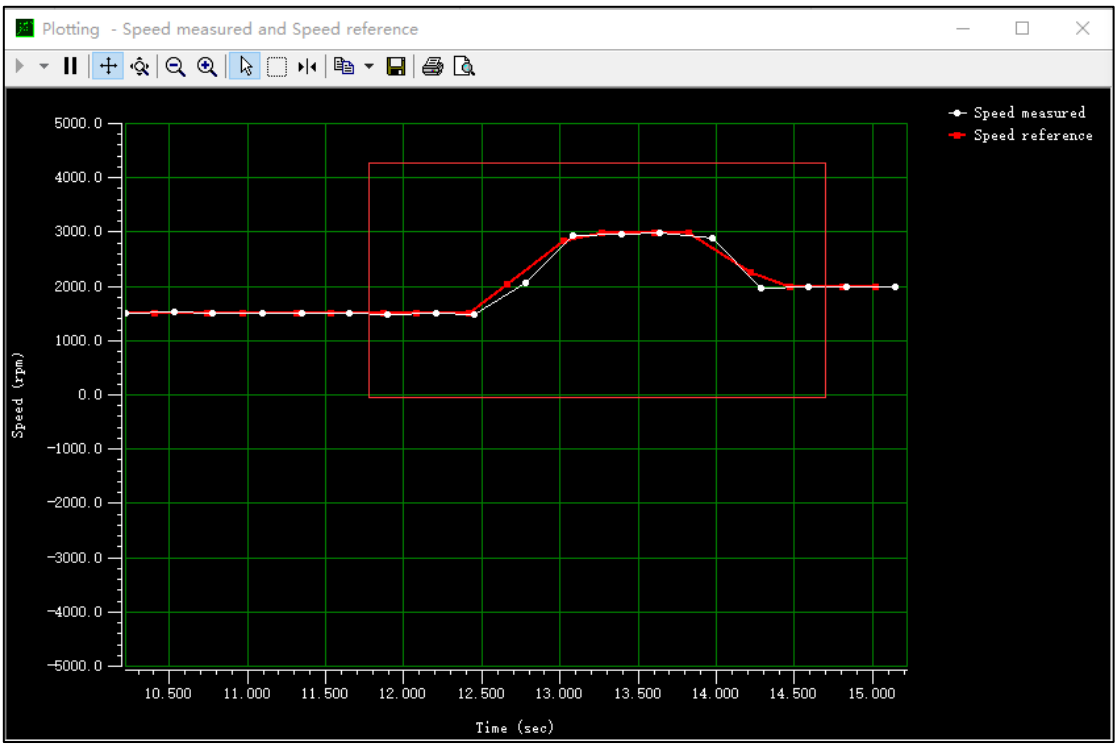


图 27.5.8 速度监控折线图

如果需要进一步调节 PID，可以点击 Advanced 按钮切换到参数调整，如下。

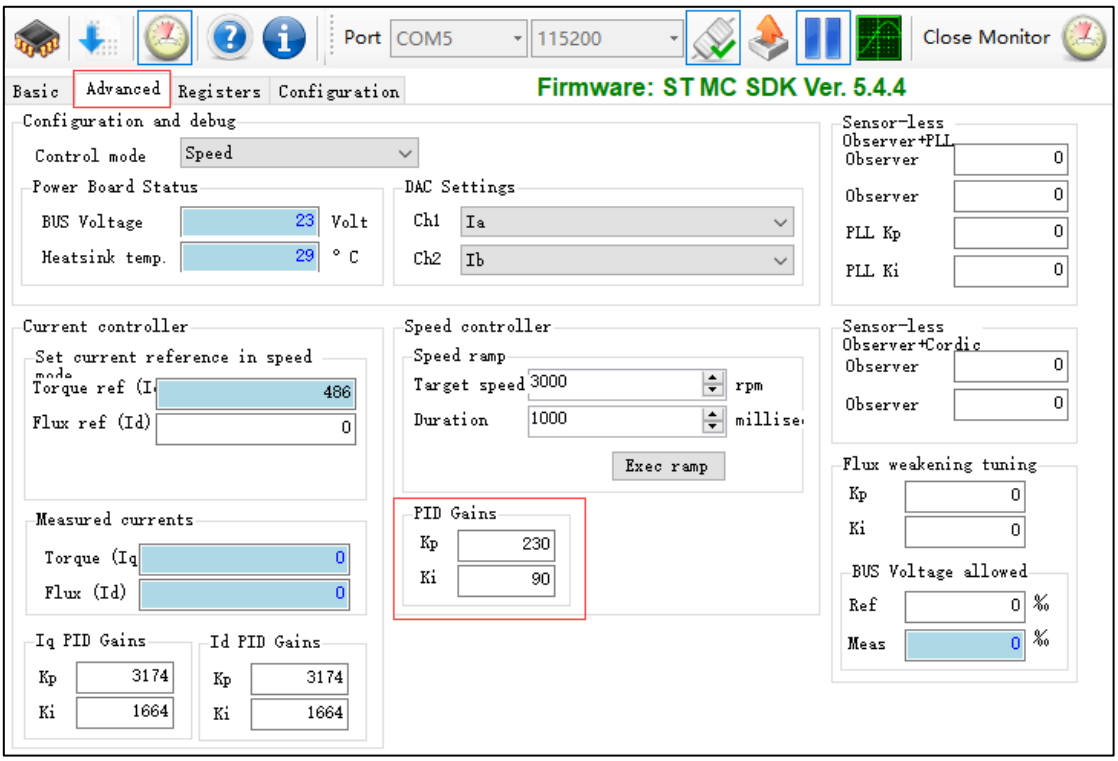


图 27.5.9 参数调整界面

修改好 PID 之后可以点击 按钮把参数上传到开发板，观测运行效果。如果使能了 DAC 输出，还可以选择 DAC 设置，默认是输出 a 和 b 电流。

27.6 ST MCSDK 库源码介绍

以 5.4 版本的 MCSDK 库为例，源码目录如下。

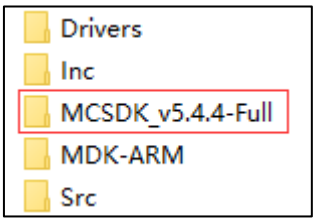


图 27.6.1 电机库总目录

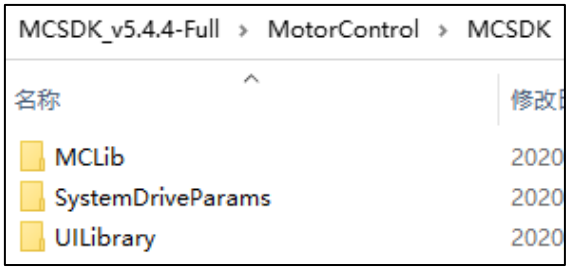


图 27.6.2 SDK 源码目录

其中 MCLib 文件夹存放的是核心程序，其目录下有两个目录，如下。

图 27.6.3 MCLib 目录

其中 Any 文件夹存放的是公共核心程序，F4xx 文件夹存放的是 F4 系列的单片机和硬件相关的驱动源文件，如果是以 F1 单片机则会生成 F1xx 的文件夹，Any 文件夹中包含 Src 和 Inc 文件夹，分别存放核心算法的源文件和头文件，源文件内容如下。

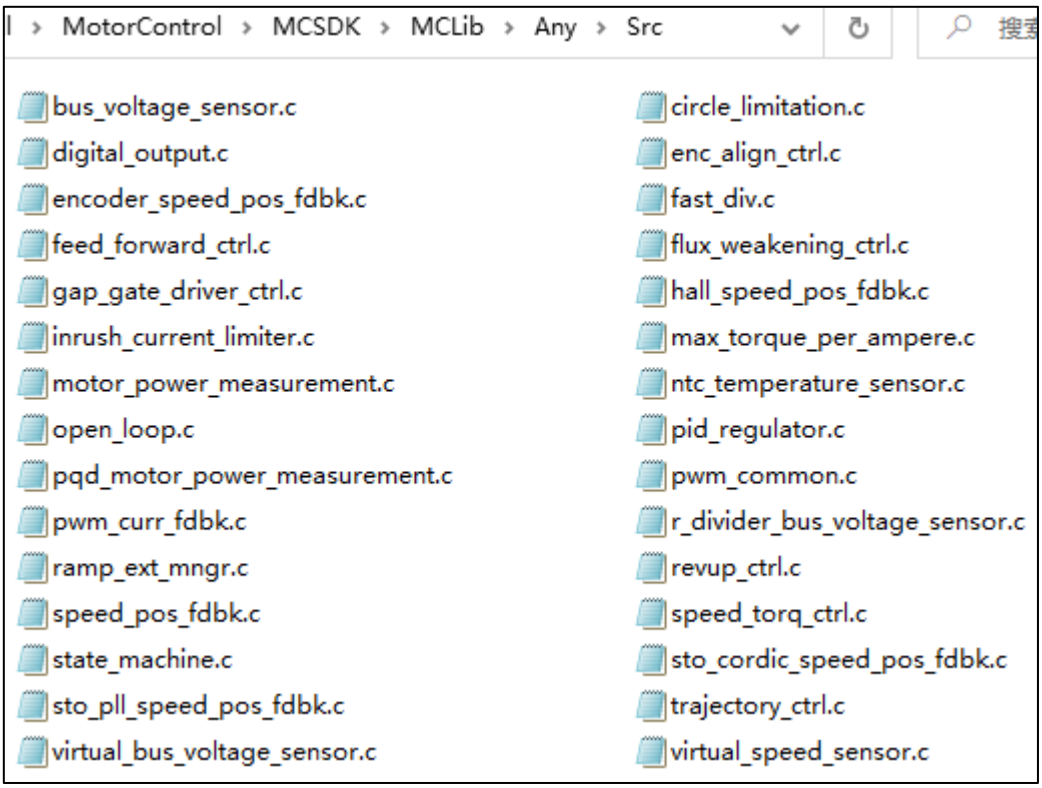


图 27.6.4 MCSDK 核心程序源文件

头文件如下。

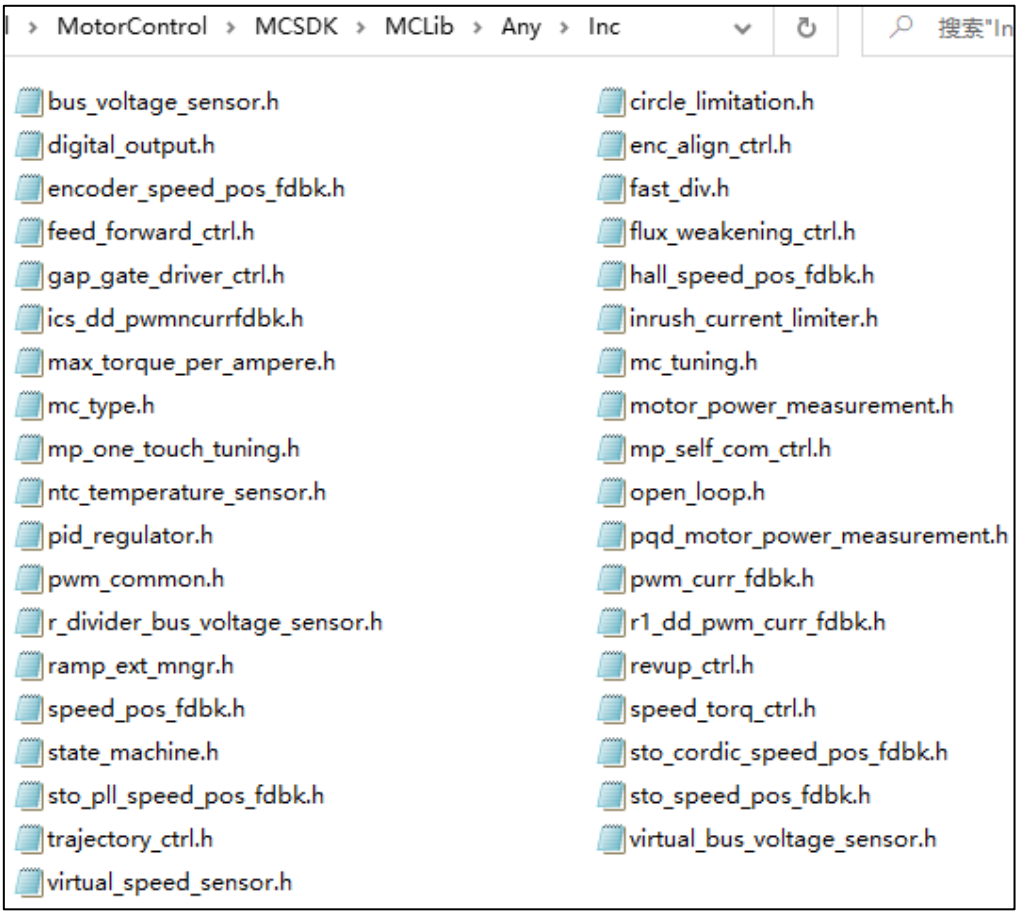


图 27.6.5 MCSDK 核心程序头文件

F4xx 目录同样分为源文件和头文件目录，内容如下。

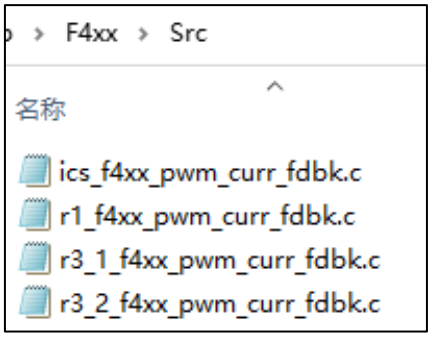


图 27.6.6 MCSDK 核心程序 F4 相关源文件

SystemDriveParams 文件夹用于存放驱动参数，没有相应改动的话一般为空。UILibrary 文件夹同样分为源文件、头文件目录，源文件如下。源文件主要功能是方便上位机或者用户和 SDK 通信，早起版本是支持屏幕交互的，也可能是此原因保留了文件夹命名为 UILibrary。

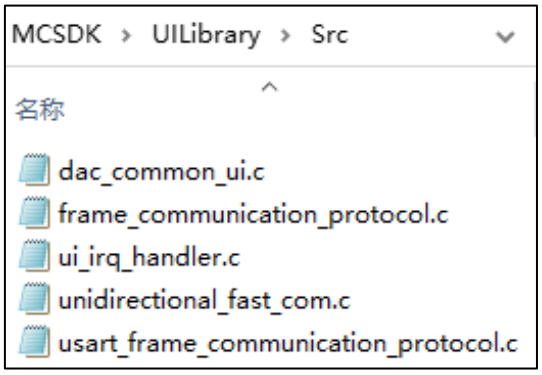


图 27.6.7 MCSDK 通信相关源文件

学习 SDK 最好的方式便是阅读帮助文档，通过 ST MC Workbench 打开帮助文档的方法如下。

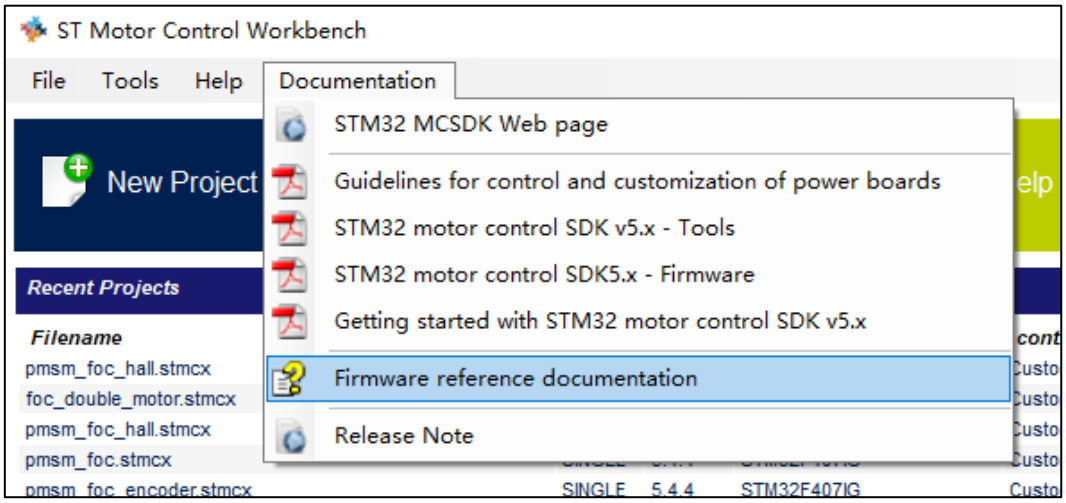


图 27.6.8 MC Workbench 帮助文档

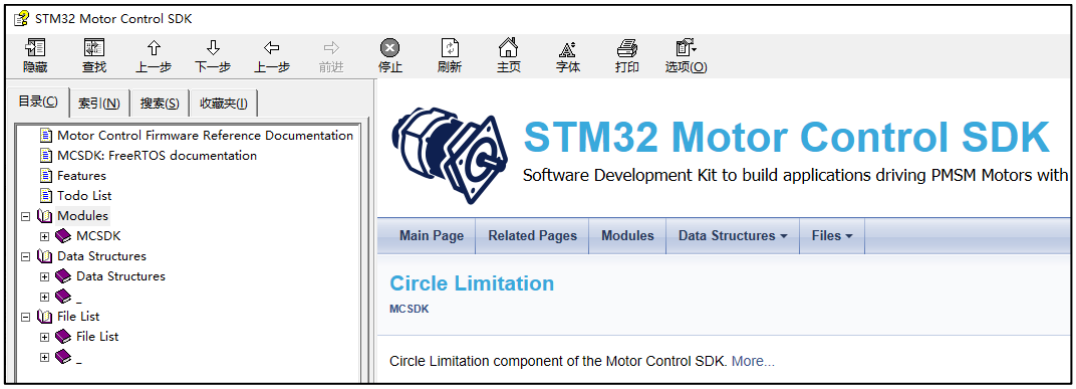


图 27.6.9 SDK 文档

文档详细描述了 SDK 的文件构成、API 介绍等内容，对于学习 SDK 有帮助。

27.7 FOC 例程分析

以 foc_pmsm_m1_encoder_os 为例，即电机接口 1 使用编码器反馈速度、位置在 FreeRTOS 下实现 FOC。

FOC 实现在系统下一共创建了两个任务，startMediumFrequencyTask 和 StartSafetyTask。

```
  /* Create the thread(s) */
  /* definition and creation of mediumFrequency */
  osThreadDef(mediumFrequency, startMediumFrequencyTask, osPriorityNormal, 0,
              128);
  mediumFrequencyHandle = osThreadCreate(osThread(mediumFrequency), NULL);

  /* definition and creation of safety */
  osThreadDef(safety, StartSafetyTask, osPriorityAboveNormal, 0, 128);
  safetyHandle = osThreadCreate(osThread(safety), NULL);
```

可能第一步跳转的是 main 函数的函数实现，这也是由于编译器的查找优先顺序决定的，不过 main 函数中的函数实现是弱定义的，真正的实现在 mc_task.c 中。

```
/* startMediumFrequencyTask function */
void startMediumFrequencyTask(void const * argument)
{
  /* USER CODE BEGIN MF task 1 */
  /* Infinite loop */
  for(;;)
  {
    /* delay of 500us */
    vTaskDelay(1);
    MC_RunMotorControlTasks();
  }
  /* USER CODE END MF task 1 */
}
```

中频任务 startMediumFrequencyTask 的内容为系统演示调用 MC_RunMotorControlTasks 函数。而 MC_RunMotorControlTasks 函数负责循环调度一些任务，主要是电机控制调度和用户接口调度。

```
__weak void MC_RunMotorControlTasks(void)
{
  if ( bMCBootCompleted ) {
    /* ** Medium Frequency Tasks ** */
    MC_Scheduler();

    /* ** User Interface Task ** */
    UI_Scheduler();
  }
}
```

其中 MC_Scheduler 主要由一个变量计数器来实现真正的任务调度，真正负责电机控制调度的实现是 TSK_MediumFrequencyTaskM1 函数。

```
__weak void TSK_MediumFrequencyTaskM1(void)
```

```
{
  /* USER CODE BEGIN MediumFrequencyTask M1 0 */

  /* USER CODE END MediumFrequencyTask M1 0 */

  State_t StateM1;
  int16_t wAux = 0;

  (void) ENC_CalcAvrgMecSpeedUnit( &ENCODER_M1, &wAux );
  PQD_CalcElMotorPower( pMPM[M1] );

  StateM1 = STM_GetState( &STM[M1] );

  switch ( StateM1 )
  {

  case IDLE:
    if ( EAC_GetRestartState( &EncAlignCtrlM1 ) )
    {
      /* The Encoder Restart State is true: the IDLE state has been entered
       * after Encoder alignment was performed. The motor can now be started.
*/
      EAC_SetRestartState( &EncAlignCtrlM1,false );

      /* USER CODE BEGIN MediumFrequencyTask M1 Encoder Restart */

      /* USER CODE END MediumFrequencyTask M1 Encoder Restart */

      STM_NextState( &STM[M1], IDLE_START );
    }
    break;

  case IDLE_START:
     …… ……
```

内容过多这里只截取了部分函数。中频任务函数主要由状态机实现，通过不同的状态来切换控制内容，例如硬件刚刚初始化的时候就是 ICLWAIT 状态，硬件初始化 OK 可以运行的时候就是 IDLE 状态，等等。各状态的定义和注释如下。

```
typedef enum
{
  ICLWAIT = 12,         /*!< Persistent state, the system is waiting for ICL
                         deactivation. Is not possible to run the motor if
                         ICL is active. Until the ICL is active the state is
                         forced to ICLWAIT, when ICL become inactive the state
                         is moved to IDLE */
  IDLE = 0,            /*!< Persistent state, following state can be IDLE_START
                         if a start motor command has been given or
                         IDLE_ALIGNMENT if a start alignment command has been
                         given */
  IDLE_ALIGNMENT = 1,   /*!< "Pass-through" state containg the code to be
                          executed
                         only once after encoder alignment command.
                         Next states can be ALIGN_CHARGE_BOOT_CAP or
                         ALIGN_OFFSET_CALIB according the configuration. It
                         can also be ANY_STOP if a stop motor command has been
                         given. */
  ALIGN_CHARGE_BOOT_CAP = 13,/*!< Persistent state where the gate driver boot
                         capacitors will be charged. Next states will be
                         ALIGN_OFFSET_CALIB. It can also be ANY_STOP if a stop
                         motor command has been given. */
  ALIGN_OFFSET_CALIB = 14,/*!< Persistent state where the offset of motor
                          currents
                         measurements will be calibrated. Next state will be
                         ALIGN_CLEAR. It can also be ANY_STOP if a stop motor
                         command has been given. */
  ALIGN_CLEAR = 15,     /*!< "Pass-through" state in which object is cleared
```

```
                        and
                       set for the startup.
                       Next state will be ALIGNMENT. It can also be ANY_STOP
                       if a stop motor command has been given. */
  … …
} State_t;
```

状态切换由 STM_NextState 函数实现。StartSafetyTask 函数主要做电机控制过程的保护作用。

硬件初始化程序在 main 函数中，如下。

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ADC1_Init();
  MX_ADC2_Init();
  MX_DAC_Init();
  MX_TIM1_Init();
  MX_TIM3_Init();
  MX_USART1_UART_Init();
  MX_MotorControl_Init();

  /* Initialize interrupts */
  MX_NVIC_Init();
```

以电机控制使用的定时器 1 为例，初始化硬件程序如下。

```
void HAL_TIM_MspPostInit(TIM_HandleTypeDef* htim)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(htim->Instance==TIM1)
  {
  /* USER CODE BEGIN TIM1_MspPostInit 0 */

  /* USER CODE END TIM1_MspPostInit 0 */

    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    /**TIM1 GPIO Configuration
    PA10     ------> TIM1_CH3
    PA9      ------> TIM1_CH2
    PA8      ------> TIM1_CH1
    PB13     ------> TIM1_CH1N
    PB14     ------> TIM1_CH2N
    PB15     ------> TIM1_CH3N
    */
    GPIO_InitStruct.Pin = M1_PWM_WH_Pin|M1_PWM_VH_Pin|M1_PWM_UH_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF1_TIM1;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = M1_PWM_UL_Pin|M1_PWM_VL_Pin|M1_PWM_WL_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF1_TIM1;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /* USER CODE BEGIN TIM1_MspPostInit 1 */

  /* USER CODE END TIM1_MspPostInit 1 */
  }

}
```

管脚定义在 main.h 头文件中，通过宏的方式可以较为方便的改动硬件。

```
#define Start_Stop_Pin GPIO_PIN_4
#define Start_Stop_GPIO_Port GPIOE
#define Start_Stop_EXTI_IRQn EXTI4_IRQn
```

```
#define UART_RX_Pin GPIO_PIN_7
#define UART_RX_GPIO_Port GPIOB
#define UART_TX_Pin GPIO_PIN_6
#define UART_TX_GPIO_Port GPIOB
#define M1_PWM_WH_Pin GPIO_PIN_10
#define M1_PWM_WH_GPIO_Port GPIOA
#define M1_PWM_VH_Pin GPIO_PIN_9
#define M1_PWM_VH_GPIO_Port GPIOA
#define M1_PWM_UH_Pin GPIO_PIN_8
#define M1_PWM_UH_GPIO_Port GPIOA
#define M1_ENCODER_B_Pin GPIO_PIN_7
#define M1_ENCODER_B_GPIO_Port GPIOC
#define SHUTDOWN2_Pin GPIO_PIN_2
#define SHUTDOWN2_GPIO_Port GPIOF
#define M1_ENCODER_A_Pin GPIO_PIN_6
#define M1_ENCODER_A_GPIO_Port GPIOC
#define SHUTDOWN_Pin GPIO_PIN_10
#define SHUTDOWN_GPIO_Port GPIOF
#define M1_TEMPERATURE_Pin GPIO_PIN_0
#define M1_TEMPERATURE_GPIO_Port GPIOA
#define DBG_DAC_CH1_Pin GPIO_PIN_4
#define DBG_DAC_CH1_GPIO_Port GPIOA
#define M1_CURR_AMPL_V_Pin GPIO_PIN_6
#define M1_CURR_AMPL_V_GPIO_Port GPIOA
#define DBG_DAC_CH2_Pin GPIO_PIN_5
#define DBG_DAC_CH2_GPIO_Port GPIOA
#define M1_OCP_Pin GPIO_PIN_12
#define M1_OCP_GPIO_Port GPIOB
#define M1_PWM_UL_Pin GPIO_PIN_13
#define M1_PWM_UL_GPIO_Port GPIOB
#define M1_CURR_AMPL_W_Pin GPIO_PIN_3
#define M1_CURR_AMPL_W_GPIO_Port GPIOA
#define M1_BUS_VOLTAGE_Pin GPIO_PIN_1
#define M1_BUS_VOLTAGE_GPIO_Port GPIOB
#define M1_CURR_AMPL_U_Pin GPIO_PIN_0
#define M1_CURR_AMPL_U_GPIO_Port GPIOB
#define M1_PWM_VL_Pin GPIO_PIN_14
#define M1_PWM_VL_GPIO_Port GPIOB
#define M1_PWM_WL_Pin GPIO_PIN_15
#define M1_PWM_WL_GPIO_Port GPIOB
```

为了便于平台生成程序，电机控制所相关的中断全部通过 stm32f4xx_mc_it.c 实现，函数声明如下。

```
/* Public prototypes of IRQ handlers called from assembly code ----------*/
void ADC_IRQHandler(void);
void TIMx_UP_M1_IRQHandler(void);
void TIMx_BRK_M1_IRQHandler(void);
void SPD_TIM_M1_IRQHandler(void);
void USART_IRQHandler(void);
void HardFault_Handler(void);
void SysTick_Handler(void);
void EXTI4_IRQHandler (void);
```

以前版本的 FOC 程序支持显示屏，之后和现在的版本均不支持显示屏控制了，不过还支持按键控制启停的功能。以开发板的按键 2(PE4)作为启停开关，启停通过外部中断的方式实现，程序如下。

```
/**
  * @brief  This function handles Button IRQ on PIN PE4.
  */
void EXTI4_IRQHandler (void)
{
    /* USER CODE BEGIN START_STOP_BTN */
  if ( LL_EXTI_ReadFlag_0_31(LL_EXTI_LINE_4) )
  {
    LL_EXTI_ClearFlag_0_31 (LL_EXTI_LINE_4);
    UI_HandleStartStopButton_cb ();
  }
```

```
}
```

中断程序首先清除中断标志位，然后调用函数 UI_HandleStartStopButton_cb，即用户接口启停按钮回调，程序通过判断电机状态决定是启动电机还是停止电机，函数实现如下。

```
__weak void UI_HandleStartStopButton_cb (void)
{
/* USER CODE BEGIN START_STOP_BTN */
  if (MC_GetSTMStateMotor1() == IDLE)
  {
    /* Ramp parameters should be tuned for the actual motor */
    MC_StartMotor1();
  }
  else
  {
    MC_StopMotor1();
  }
```

同 UI_HandleStartStopButton_cb 一样，很多函数很多情况下都是弱定义的也就是可以由用户覆写以实现更灵活的功能。

串口中断函数主要用于与上位机的通信，在接收到数据的时候通过 UFCP_RX_IRQ_Handler 函数来将有效数据放置到缓存中。接收到完整的数据帧后通过回调函数 MCP_ReceivedFrame 来解析数据帧并实现交互。接收数据帧解析赋值函数通过 MCP_Init 函数来初始化，如下。

```
__weak void MCP_Init( MCP_Handle_t *pHandle,
              FCP_Handle_t * pFCP,
              FCP_SendFct_t fFcpSend,
              FCP_ReceiveFct_t fFcpReceive,
              FCP_AbortReceiveFct_t fFcpAbortReceive,
              DAC_UI_Handle_t * pDAC,
              const char* s_fwVer )
{
  pHandle->pFCP = pFCP;
  pHandle->pDAC = pDAC;
  pHandle->s_fwVer = s_fwVer;
  FCP_SetClient( pFCP, pHandle,
                (FCP_SentFrameCallback_t) & MCP_SentFrame,
                (FCP_ReceivedFrameCallback_t) & MCP_ReceivedFrame,
                (FCP_RxTimeoutCallback_t) & MCP_OnTimeOut );
  pHandle->fFcpSend = fFcpSend;
  pHandle->fFcpReceive = fFcpReceive;
  pHandle->fFcpAbortReceive = fFcpAbortReceive;

  MCP_WaitNextFrame(pHandle);
}
```

由于例程使用了编码器用于反馈速度和位置，所以有 SPD_TIM_M1_IRQHandler 函数，也就是 TIM3_IRQHandler，用于更新编码器计数溢出次数。

```
/**********  AUXILIARY ENCODER TIMER MOTOR 1 *************/
#define M1_PULSE_NBR  ( (4 * (M1_ENCODER_PPR)) - 1 )
#define M1_ENC_IC_FILTER  9
#define SPD_TIM_M1_IRQHandler TIM3_IRQHandler
```

TIMx_BRK_M1_IRQHandler 中断函数用于刹车，也就是 TIM1_BRK_TIM9_IRQHandler，刹车信号有效时调用 R3_2_BRK_IRQHandler 函数，关断输出信号并设置过流标志。

```
/**
 * @brief  It contains the Break event interrupt
 * @param  pHandle: handler of the current instance of the PWM component
 * @retval none
 */
__weak void *R3_2_BRK_IRQHandler(PWMC_R3_2_Handle_t *pHandle)
{

  if ((pHandle->pParams_str->LowSideOutputs)== ES_GPIO)
  {
    LL_GPIO_ResetOutputPin(pHandle->pParams_str->pwm_en_u_port,
```

```
                              pHandle->pParams_str->pwm_en_u_pin);
    LL_GPIO_ResetOutputPin(pHandle->pParams_str->pwm_en_v_port,
                              pHandle->pParams_str->pwm_en_v_pin);
    LL_GPIO_ResetOutputPin(pHandle->pParams_str->pwm_en_w_port,
                              pHandle->pParams_str->pwm_en_w_pin);
  }
  pHandle->OverCurrentFlag = true;

  return &(pHandle->_Super.Motor);
}
```

TIMx_UP_M1_IRQHandler 函数用于在定时器更新中断时使用 ADC 注入通道获取采样电流值用于 FOC 计算的电流反馈。也就是 TIM1_UP_TIM10_IRQHandler。

```
/*************************  IRQ Handler Mapping  ********************/
#define TIMx_UP_M1_IRQHandler TIM1_UP_TIM10_IRQHandler
#define DMAx_R1_M1_IRQHandler DMA2_Stream4_IRQHandler
#define DMAx_R1_M1_Stream     DMA2_Stream4
#define TIMx_BRK_M1_IRQHandler TIM1_BRK_TIM9_IRQHandler
```

TIMx_UP_M1_IRQHandler 函数首先清除更新中断，然后运行 R3_2_TIMx_UP_IRQHandler 回调函数，函数实现如下。

```
/**
  * @brief  It contains the TIMx Update event interrupt
  * @param  pHandle: handler of the current instance of the PWM component
  * @retval none
  */
__weak void * R3_2_TIMx_UP_IRQHandler( PWMC_R3_2_Handle_t * pHandle)
{
  TIM_TypeDef* TIMx = pHandle->pParams_str->TIMx;
  ADC_TypeDef * ADCx_1 = pHandle->pParams_str->ADCx_1;
  ADC_TypeDef * ADCx_2 = pHandle->pParams_str->ADCx_2;
  uint32_t ADCInjFlags;

  /* dual drive check */
  ADCInjFlags = ADCx_1->SR & (LL_ADC_FLAG_JSTRT|LL_ADC_FLAG_JEOS);
  if ( ADCInjFlags == LL_ADC_FLAG_JSTRT )
  {
    /* ADC conversion is on going on the second motor */
    do
    {
      /* wait for end of conversion */
      ADCInjFlags = ADCx_1->SR & (LL_ADC_FLAG_JSTRT|LL_ADC_FLAG_JEOS);
    }
    while ( ADCInjFlags != (LL_ADC_FLAG_JSTRT|LL_ADC_FLAG_JEOS) );
  }
  else if ( ADCInjFlags == 0 )
  {
    /* ADC conversion on the second motor is not yet started */
    while ( ( TIMx->CNT ) < ( pHandle->pParams_str->Tw ) )
    {
      /* wait for a maximum delay */
    }

    ADCInjFlags = ADCx_1->SR & (LL_ADC_FLAG_JSTRT|LL_ADC_FLAG_JEOS);
    if ( ADCInjFlags == LL_ADC_FLAG_JSTRT )
    {
      /* ADC conversion is on going on the second motor */
      do
      {
        /* wait for end of conversion */
        ADCInjFlags = ADCx_1->SR & (LL_ADC_FLAG_JSTRT|LL_ADC_FLAG_JEOS);
      }
      while ( ADCInjFlags != (LL_ADC_FLAG_JSTRT|LL_ADC_FLAG_JEOS) );
    }
  }
  else
  {
```

```
    /* ADC conversion on the second motor is done */
  }

  /* Disabling trigger to avoid unwanted conversion */
  LL_ADC_INJ_StopConversionExtTrig(ADCx_1);
  LL_ADC_INJ_StopConversionExtTrig(ADCx_2);

  /* Set next current channel according to sector  */
  ADCx_1->JSQR = pHandle->pParams_str->ADCConfig1[pHandle->_Super.Sector];
  ADCx_2->JSQR = pHandle->pParams_str->ADCConfig2[pHandle->_Super.Sector];

  LL_ADC_INJ_SetTriggerSource(ADCx_1,pHandle->ADC_ExternalTriggerInjected);
  LL_ADC_INJ_SetTriggerSource(ADCx_2,pHandle->ADC_ExternalTriggerInjected);

  /* enable ADC trigger source */
  LL_TIM_CC_EnableChannel(TIMx, LL_TIM_CHANNEL_CH4);
  LL_ADC_INJ_StartConversionExtTrig(ADCx_1, pHandle->ADCTriggerEdge);
  LL_ADC_INJ_StartConversionExtTrig(ADCx_2, pHandle->ADCTriggerEdge);

  /* reset default edge detection trigger */
  pHandle->ADCTriggerEdge = LL_ADC_INJ_TRIG_EXT_RISING;

  return &( pHandle->_Super.Motor );
}
```

ADC_IRQHandler 函数用于输出调试变量，通过 UI_DACUpdate 实现，通过判断 TSK_HighFrequencyTask 函数返回值判断执行 FOC 循环的电机 id 来决定输出的模拟值以适配不同电机的不同变量。因此中断时也执行一次 TSK_HighFrequencyTask，即 FOC 高频任务，也就是 FOC 循环的主体。函数实现如下。

```
__weak uint8_t TSK_HighFrequencyTask(void)
{
  /* USER CODE BEGIN HighFrequencyTask 0 */

  /* USER CODE END HighFrequencyTask 0 */

  uint8_t bMotorNbr = 0;
  uint16_t hFOCreturn;

  ENC_CalcAngle(&ENCODER_M1);   /* if not sensorless then 2nd parameter is
MC_NULL*/

  /* USER CODE BEGIN HighFrequencyTask SINGLEDRIVE_1 */

  /* USER CODE END HighFrequencyTask SINGLEDRIVE_1 */
  hFOCreturn = FOC_CurrControllerM1();
  /* USER CODE BEGIN HighFrequencyTask SINGLEDRIVE_2 */

  /* USER CODE END HighFrequencyTask SINGLEDRIVE_2 */
  if(hFOCreturn == MC_FOC_DURATION)
  {
    STM_FaultProcessing(&STM[M1], MC_FOC_DURATION, 0);
  }
  else
  {
    /* USER CODE BEGIN HighFrequencyTask SINGLEDRIVE_3 */

    /* USER CODE END HighFrequencyTask SINGLEDRIVE_3 */
  }
  /* USER CODE BEGIN HighFrequencyTask 1 */

  /* USER CODE END HighFrequencyTask 1 */

  return bMotorNbr;
}
```

FOC 循环的过程就是和之前介绍的流程一致，通过 ENC_CalcAngle 获取转子电角度，通过 PWMC_GetPhaseCurrents 获取 a 相以及 b 相电流，然后进行 Clarke 变换，实现函数：MCM_Clarke；接着进行 Park 变换，实现函数：MCM_Park；接着就是对 d 轴以及 q 轴进行 PI 运算，运算后的结果在进行反 Park 变换，实现函数：MCM_Rev_Park；接着就是进行 SVPWM 的运算，实现函数：PWMC_SetPhaseVoltage，整个过程都在 FOC_CurrControllerM1 函数中实现，它的功能主要用于控制电流环，函数实现如下：

```
inline uint16_t FOC_CurrControllerM1(void)
{
  qd_t Iqd, Vqd;
  ab_t Iab;
  alphabeta_t Ialphabeta, Valphabeta;

  int16_t hElAngle;
  uint16_t hCodeError;
  SpeednPosFdbk_Handle_t *speedHandle;

  speedHandle = STC_GetSpeedSensor(pSTC[M1]);
  hElAngle = SPD_GetElAngle(speedHandle);
  PWMC_GetPhaseCurrents(pwmcHandle[M1], &Iab);
  Ialphabeta = MCM_Clarke(Iab);
  Iqd = MCM_Park(Ialphabeta, hElAngle);
  Vqd.q = PI_Controller(pPIDIq[M1],
          (int32_t)(FOCVars[M1].Iqdref.q) - Iqd.q);

  Vqd.d = PI_Controller(pPIDId[M1],
          (int32_t)(FOCVars[M1].Iqdref.d) - Iqd.d);

  Vqd = Circle_Limitation(pCLM[M1], Vqd);
  hElAngle +=
  SPD_GetInstElSpeedDpp(speedHandle)*REV_PARK_ANGLE_COMPENSATION_FACTOR;
  Valphabeta = MCM_Rev_Park(Vqd, hElAngle);
  hCodeError = PWMC_SetPhaseVoltage(pwmcHandle[M1], Valphabeta);
  FOCVars[M1].Vqd = Vqd;
  FOCVars[M1].Iab = Iab;
  FOCVars[M1].Ialphabeta = Ialphabeta;
  FOCVars[M1].Iqd = Iqd;
  FOCVars[M1].Valphabeta = Valphabeta;
  FOCVars[M1].hElAngle = hElAngle;
  return(hCodeError);
}
```

函数 PWMC_SetPhaseVoltage 计算过后的结果，通过函数 pFctSetADCSampPointSectX 设置定时器各通道的比较值大小，进而控制电机。然后又通过 R3_2_WriteTIMRegisters 函数通过检查 TIMx CH4 是否启用。如果标志置位则发生更新事件因此 FOC 速率太高，可能过流。如果过流则执行过流处理，即 STM_FaultProcessing 函数。

```
__weak uint16_t PWMC_SetPhaseVoltage( PWMC_Handle_t * pHandle, alphabeta_t
Valfa_beta )
{
  int32_t wX, wY, wZ, wUAlpha, wUBeta, wTimePhA, wTimePhB, wTimePhC;

  wUAlpha = Valfa_beta.alpha * ( int32_t )pHandle->hT_Sqrt3;
  wUBeta = -( Valfa_beta.beta * ( int32_t )( pHandle->PWMperiod ) ) * 2;

  wX = wUBeta;
  wY = ( wUBeta + wUAlpha ) / 2;
  wZ = ( wUBeta - wUAlpha ) / 2;

  /* Sector calculation from wX, wY, wZ */
  if ( wY < 0 )
  {
    if ( wZ < 0 )
    {
      …… ……
    if ( pHandle->Ic > 0 )
```

```
    {
      pHandle->CntPhC += pHandle->DTCompCnt;
    }
    else
    {
      pHandle->CntPhC -= pHandle->DTCompCnt;
    }
  }

  return ( pHandle->pFctSetADCSampPointSectX( pHandle ) );
}
```

其中 pFctSetADCSampPointSectX 赋值如下：

```
PWMC_R3_2_Handle_t PWM_Handle_M1=
{
  {
    .pFctGetPhaseCurrents            = &R3_2_GetPhaseCurrents,
    .pFctSwitchOffPwm               = &R3_2_SwitchOffPWM,
    .pFctSwitchOnPwm                = &R3_2_SwitchOnPWM,
    .pFctCurrReadingCalib            = &R3_2_CurrentReadingCalibration,
    .pFctTurnOnLowSides              = &R3_2_TurnOnLowSides,
    .pFctSetADCSampPointSectX         = &R3_2_SetADCSampPointSectX,
    .pFctIsOverCurrentOccurred        = &R3_2_IsOverCurrentOccurred,
    .pFctOCPSetReferenceVoltage       = MC_NULL,
    .pFctRLDetectionModeEnable        = &R3_2_RLDetectionModeEnable,
    .pFctRLDetectionModeDisable       = &R3_2_RLDetectionModeDisable,
    .pFctRLDetectionModeSetDuty       = &R3_2_RLDetectionModeSetDuty,
    .hT_Sqrt3 = (PWM_PERIOD_CYCLES*SQRT3FACTOR)/16384u,
    …… ……
};
```

R3_2_SetADCSampPointSectX 函数定义如下：

```
__weak uint16_t R3_2_SetADCSampPointSectX( PWMC_Handle_t * pHdl )
{
  PWMC_R3_2_Handle_t * pHandle = ( PWMC_R3_2_Handle_t * )pHdl;

  uint16_t hCntSmp;
  uint16_t hDeltaDuty;
  register uint16_t lowDuty = pHdl->lowDuty;
  register uint16_t midDuty = pHdl->midDuty;

  /* Check if sampling AB in the middle of PWM is possible */
  if ( ( uint16_t )( pHandle->Half_PWMPeriod - lowDuty ) >
pHandle->pParams_str->hTafter )
  {
    pHandle->_Super.Sector = SECTOR_4;

    /* set sampling  point trigger in the middle of PWM period */
    hCntSmp = ( uint32_t )( pHandle->Half_PWMPeriod ) - 1u;
  }
  else
  {
    … …
  }

  return R3_2_WriteTIMRegisters( &pHandle->_Super, hCntSmp );
}
```

接着看下返回值 R3_2_WriteTIMRegisters 的函数定义：

```
__STATIC_INLINE uint16_t R3_2_WriteTIMRegisters( PWMC_Handle_t * pHdl,
uint16_t hCCR4Reg )
{
  PWMC_R3_2_Handle_t * pHandle = (PWMC_R3_2_Handle_t *) pHdl;
  TIM_TypeDef * TIMx = pHandle->pParams_str->TIMx;
  uint16_t hAux;

  LL_TIM_OC_SetCompareCH1( TIMx, pHandle->_Super.CntPhA );
  LL_TIM_OC_SetCompareCH2( TIMx, pHandle->_Super.CntPhB );
```

```
  LL_TIM_OC_SetCompareCH3( TIMx, pHandle->_Super.CntPhC );
  LL_TIM_OC_SetCompareCH4( TIMx, hCCR4Reg );

  /* Limit for update event */
  /* Check the if TIMx CH4 is enabled. If it is set, an update event has
occurred
  and thus the FOC rate is too high */
  if (LL_TIM_CC_IsEnabledChannel(TIMx, LL_TIM_CHANNEL_CH4))
  {
    hAux = MC_FOC_DURATION;
  }
  else
  {
    hAux = MC_NO_ERROR;
  }
  if ( pHandle->_Super.SWerror == 1u )
  {
    hAux = MC_FOC_DURATION;
    pHandle->_Super.SWerror = 0u;
  }
  return hAux;
}
```

MC WorkBench 上位机目前仅支持速度控制和扭矩控制模式，主要是通过调节电流的方式实现，主要逻辑就是在 FOC_CurrControllerM1 函数中实现。

```
inline uint16_t FOC_CurrControllerM1(void)
{
  qd_t Iqd, Vqd;
  ab_t Iab;
  alphabeta_t Ialphabeta, Valphabeta;

  int16_t hElAngle;
  uint16_t hCodeError;
  SpeednPosFdbk_Handle_t *speedHandle;

  speedHandle = STC_GetSpeedSensor(pSTC[M1]);
  hElAngle = SPD_GetElAngle(speedHandle);
  PWMC_GetPhaseCurrents(pwmcHandle[M1], &Iab);
  Ialphabeta = MCM_Clarke(Iab);
  Iqd = MCM_Park(Ialphabeta, hElAngle);
  Vqd.q = PI_Controller(pPIDIq[M1],
          (int32_t)(FOCVars[M1].Iqdref.q) - Iqd.q);

  Vqd.d = PI_Controller(pPIDId[M1],
          (int32_t)(FOCVars[M1].Iqdref.d) - Iqd.d);

  Vqd = Circle_Limitation(pCLM[M1], Vqd);
  hElAngle +=
SPD_GetInstElSpeedDpp(speedHandle)*REV_PARK_ANGLE_COMPENSATION_FACTOR;
  Valphabeta = MCM_Rev_Park(Vqd, hElAngle);
  hCodeError = PWMC_SetPhaseVoltage(pwmcHandle[M1], Valphabeta);
  FOCVars[M1].Vqd = Vqd;
  FOCVars[M1].Iab = Iab;
  FOCVars[M1].Ialphabeta = Ialphabeta;
  FOCVars[M1].Iqd = Iqd;
  FOCVars[M1].Valphabeta = Valphabeta;
  FOCVars[M1].hElAngle = hElAngle;
  return(hCodeError);
}
```

首先通过 STC_GetSpeedSensor 函数获取速度传感器句柄，然后通过 SPD_GetElAngle 函数和速度传感器句柄获取到转子角度 hElAngle，然后通过函数 PWMC_GetPhaseCurrents 获取定子电流到 Iab，实际上是通过电流获取函数指针指向的 R3_2_GetPhaseCurrents 函数实现的，获取电流之后通过 MCM_Clarke 函数进行 Clarke 变换到直角坐标系，然后再经过 MCM_Park 函数进行 Park 变换得到 DQ 轴电流 Iqd，例程使用速度控制模式，不过速度环经过 PID 的输出还是要附加到 DQ 轴电流上，用于最终达到设定速度。PI 控制使用

PI_Controller 函数实现，速度环 PI 控制器输出 DQ 轴电流限制使用 Circle_Limitation 函数实现。然后经过 MCM_Rev_Park 进行反 Park 变换得到直角坐标系电流值，最终通过 PWMC_SetPhaseVoltage 函数把电流通过 SVPWM 输出到电机。

输出限制函数 Circle_Limitation 实现如下。

```
__weak qd_t Circle_Limitation( CircleLimitation_Handle_t * pHandle, qd_t Vqd )
{
  uint16_t table_element;
  uint32_t uw_temp;
  int32_t  sw_temp;
  qd_t local_vqd = Vqd;

  sw_temp = ( int32_t )( Vqd.q ) * Vqd.q +
            ( int32_t )( Vqd.d ) * Vqd.d;

  uw_temp = ( uint32_t ) sw_temp;

  /* uw_temp min value 0, max value 32767*32767 */
  if ( uw_temp > ( uint32_t )( pHandle->MaxModule ) * pHandle->MaxModule )
  {

    uw_temp /= ( uint32_t )( 16777216 );

    /* wtemp min value pHandle->Start_index, max value 127 */
    uw_temp -= pHandle->Start_index;

    /* uw_temp min value 0, max value 127 - pHandle->Start_index */
    table_element = pHandle->Circle_limit_table[( uint8_t )uw_temp];

    sw_temp = Vqd.q * ( int32_t )table_element;
    local_vqd.q = ( int16_t )( sw_temp / 32768 );

    sw_temp = Vqd.d * ( int32_t )( table_element );
    local_vqd.d = ( int16_t )( sw_temp / 32768 );
  }

  return ( local_vqd );
}
```

程序通过计算 DQ 轴电流的平方和和最大限制输出占比的平方做比较，判断是否需要进行限制。最大限制输出占比初始赋值如下。

```
CircleLimitation_Handle_t CircleLimitationM1 =
{
  .MaxModule         = MAX_MODULE,
  .MaxVd             = (uint16_t)(MAX_MODULE * 950 / 1000),
  .Circle_limit_table = MMITABLE,
  .Start_index       = START_INDEX,
};
```

其最大输出占比宏定义如下。

```
/* MMI Table Motor 1 MAX_MODULATION_89_PER_CENT */
#define START_INDEX 50
#define MAX_MODULE 29162
```

程序限制在 89%输出，即 32767*0.89=29162.63≈29162。计算结果除以 65536 除以 256 即 16777216 将数值限制在 0~255，然后减去 50 定位到限制数组中，结果乘以原始 DQ 轴电流再除以 32768 以得到最终限制输出电流值。

PWMC_SetPhaseVoltage 用以将直角坐标系电流值赋值到定时器以输出 SVPWM，将 Valfa_beta 乘以 hT_Sqrt3 转换成三相坐标系 wX, wY, wZ。hT_Sqrt3 定义为 0xDDB4 即 56756，即$\sqrt{3}/2$*65536≈56,755.840862。

接着看下开方快速计算因子赋值：

```
PWMC_R3_2_Handle_t PWM_Handle_M1=
{
  {
    .pFctGetPhaseCurrents            = &R3_2_GetPhaseCurrents,
    .pFctSwitchOffPwm                = &R3_2_SwitchOffPWM,
```

```
    .pFctSwitchOnPwm                = &R3_2_SwitchOnPWM,
    .pFctCurrReadingCalib            = &R3_2_CurrentReadingCalibration,
    .pFctTurnOnLowSides              = &R3_2_TurnOnLowSides,
    .pFctSetADCSampPointSectX         = &R3_2_SetADCSampPointSectX,
    .pFctIsOverCurrentOccurred        = &R3_2_IsOverCurrentOccurred,
    .pFctOCPSetReferenceVoltage       = MC_NULL,
    .pFctRLDetectionModeEnable        = &R3_2_RLDetectionModeEnable,
    .pFctRLDetectionModeDisable       = &R3_2_RLDetectionModeDisable,
    .pFctRLDetectionModeSetDuty       = &R3_2_RLDetectionModeSetDuty,
    .hT_Sqrt3 = (PWM_PERIOD_CYCLES*SQRT3FACTOR)/16384u,
    … …
};
```

其中计算因子宏定义为：

```
#define SECTOR_1  0u
#define SECTOR_2  1u
#define SECTOR_3  2u
#define SECTOR_4  3u
#define SECTOR_5  4u
#define SECTOR_6  5u
#define SQRT3FACTOR (uint16_t) 0xDDB4 /* = (16384 * 1.732051 * 2)*/
```

通过比较其大小判断所处扇区。然后独立计算三相定时器的比较值大小，设置定时器各通道的比较值，这样就可以实现，控制三相逆变电路的开关了，从而实现 FOC 控制。最后再附加死区控制完成完整的 SVPWM。

至此，整个 DMF407 电机开发板的 FOC 实验内容就讲完了，其中关于 FOC 的内容理解也参考了不少网上的资料，对此表示衷心的感谢，同时也希望我们这些代码，可以让大家有所受益，能开发出更强更好的产品。

总的来说我们的教程里介绍的电机种类是非常多的，所以关于电机的相关代码也是比较多的，希望大家，慢慢理解，各个攻破，最后祝大家身体健康、学习进步！

正点原子
2022-5-14
于广州