

CMC693PR144-M 运动控制芯片 使用手册

宁波中控微电子有限公司

2021 年 01 月 05 日

声 明

- 严禁转载本手册的部分或全部内容。
- 在不经预告和联系的情况下，本手册的内容有可能发生变更，请谅解。
- 本手册所记载的内容，不排除有误记或遗漏的可能性。如对本手册内容有疑问，请与我公司联系。

目 录

1 特性描述.....	1
2 功能简介.....	1
3 专用名词.....	2
4 引脚定义.....	3
5 引脚功能说明.....	4
6 电气特性.....	7
7 控制方法.....	8
7.1 单指令立即执行.....	8
7.2 多指令缓存执行.....	9
7.3 安全连接模式.....	12
8 详细功能介绍.....	13
8.1 单轴运动.....	14
8.1.1 可变速运动.....	14
8.1.2 定长运动.....	16
8.1.3 电气原点回归.....	18
8.1.4 机械原点回归.....	19
8.1.5 应用举例.....	20
8.2 多轴运动.....	20
8.2.1 坐标系选择.....	20
8.2.2 坐标模式选择.....	21
8.2.3 直线插补.....	21
8.2.4 圆弧插补.....	23
8.3 速度模式.....	25
8.3.1 定速.....	25
8.3.2 直线加减速.....	27
8.3.3 S 曲线加减速.....	28
8.4 电子齿轮.....	31
8.5 PWM 输出.....	31
8.6 编码器输入.....	32
8.7 手轮控制.....	33
8.8 IO 控制.....	34
8.8.1 普通 IO.....	34
8.8.2 特殊 IO.....	36
8.9 复位.....	37
8.9.1 硬件复位.....	37
8.9.2 软件复位.....	37
8.10 配置及状态的读写.....	38
8.10.1 芯片配置.....	38
8.10.2 运动状态.....	38
8.10.3 报警及错误状态.....	38
9 封装.....	40
10 推荐电路.....	41

10.1 主控电路最小系统.....	41
10.2 以太网电路.....	41
10.3 与通用 MCU 通讯.....	41
10.4 脉冲控制电路.....	42
10.5 编码器电路.....	42
10.6 手轮电路.....	43
10.7 RS232 通讯电路.....	43
10.8 输入输出电路.....	43
11 时序要求.....	45
12 版本信息.....	46

1 特性描述

CMC693PR144-M 是一颗支持 4 轴 3 联动的运动控制专用芯片，芯片支持 4 轴脉冲+方向信号输出，可用于控制步进电机、伺服电机等常用电机，芯片通过串口或者网口以函数接口的方式进行运动控制指令下发和状态读取。芯片支持单轴运动、多轴插补、S 曲线加减速、梯形加减速、手轮控制、编码器读取等功能，可广泛应用于专机设备、自动化流水线设备等多种场合。

CMC693PR144-M 作为运动控制专用芯片，可免去用户开发运动控制算法的麻烦，运动控制采用 ASIC 方式执行，性能稳定、效率高，同时提供了丰富的通用接口，是普通主控设备的良好补充。

2 功能简介

表 2-1 CMC693PR144-M 功能简介表

参数	描述
独立 4 轴驱动	每个轴均可定速驱动，直线加/减速驱动，S 曲线加/减速驱动等
插补驱动	2 轴/3 轴直线插补，平面圆弧插补
单轴控制	JOG 点动、机械原点回归、电气原点回归、可变速运行、多段速运行、中断减速停止等
运行模式	单条运动控制指令输出、多条指令连续输出
实时监控	状态值，逻辑位置（电气原点）、实际位置（机械原点）、相对位置坐标、驱动速度、加速度、加/减速状态等
脉冲驱动模式	单脉冲模式、双脉冲模式
编程方式	支持 C、C#、C++ 三种编程语言
电子齿轮	支持 4 轴独立电子齿轮功能
回零	支持 2 种回零模式（一次回零、一次回零加回找）
编码器	支持增量式编码器，支持手轮控制

3 专用名词

运动控制 (Motion Control)，是自动化的一个分支，它使用统称为伺服机构的一些设备如液压泵，线性执行机或者是电机来控制机器的速度或位置。

插补 (Interpolation)，即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”。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”。

脉冲当量 (Pulse Equivalent)，是当控制器输出一个定位控制脉冲时，所产生的定位控制移动的位移。对直线运动来说，是指移动的距离；对圆周运动来说，是指其转动的角度。

四轴三联动，运动控制系统总共支持控制 4 个轴，X, Y, Z, U 轴，多轴运动时系统最多只能支持三轴联动，构建三维运动控制系统。

联动，是指系统中能够联动的两个或两个以上的轴，在一个轴运动时，另外的轴也能做匀速或周期的运动，一般用于数控机床。

PPS, (Pulse Per Second)，每秒脉冲数，在以脉冲控制领域，PPS 是常用的单位。

AB 相脉冲，指一个脉冲信号通过相位差形成两个有前后相位的 A 相脉冲和 B 相脉冲，通过两个信号的先后顺序可以起到判断方向的效果。

增量式编码器，将位移转换为周期性的电信号，再把电信号转变成计数脉冲，脉冲个数表示位移的大小，增量式编码器与起止位置没有关系，仅能获取一个相对的位移数据。

手轮，手动操作的摇轮，一般具有 X1, X10, X100 三档倍率，可将手动摇动的位移转换为脉冲电信号输出，也可以对轴进行选择，主要用于数控机床、加工中心等场合。

同步，指从轴与主轴保持线性关系的状态，通常包括速度同步和位置同步，通常在电子齿轮和电子凸轮场景下会涉及到同步的概念。

电子齿轮 (Gearing)，可以完成主轴与从轴间位置的线性传递功能。

4 引脚定义

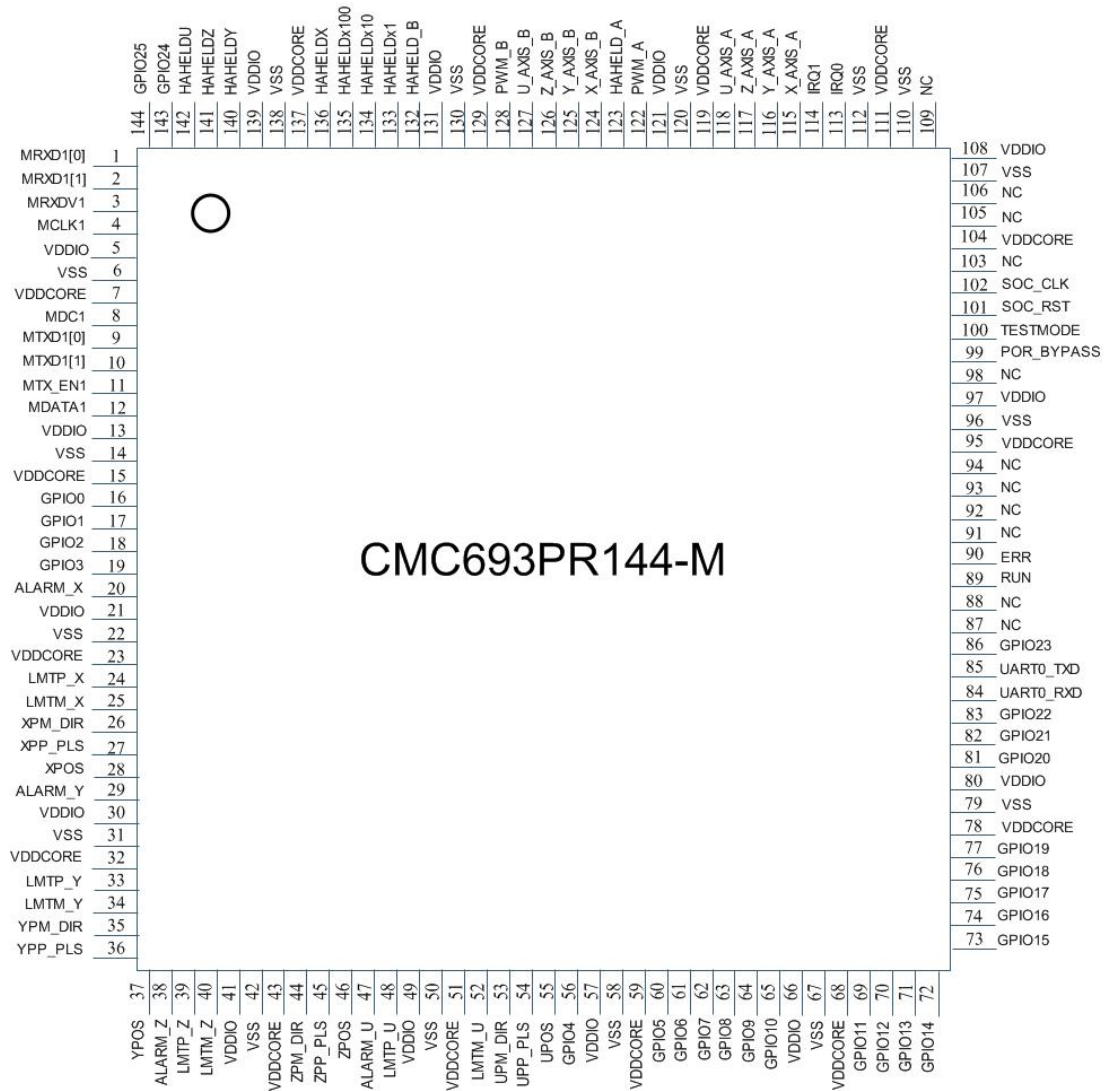


图 4-1 CMC693PR144-M 引脚定义图

5 引脚功能说明

表 5-1 CMC693PR144-M 引脚功能说明表

信号	引脚号	输入/输出	描述
VDDCORE	7, 15, 23, 32, 43, 51, 59, 68, 78, 95, 104, 111, 119, 129, 137	电源	内核电压, 1.2V
VDDIO	5, 13, 21, 30, 41, 49, 57, 66, 80, 97, 108, 121, 131, 139	电源	IO 电压, 3.3V
VSS	6, 14, 22, 31, 42, 50, 58, 67, 79, 96, 107, 110, 112, 120, 130, 138	地线	GND
NC	87, 88, 91, 92, 93, 94, 98, 103, 105, 106, 109		不接
MRXD1[0]	1	输入	MAC 数据接收
MRXD1[1]	2	输入	MAC 数据接收
MRXDV1	3	输入	MAC 数据接收使能
MCLK1	4	输入	MAC 输入时钟
MDC1	8	输入	MAC 数据管理时钟
MTXD1[0]	9	输出	MAC 数据发送
MTXD1[1]	10	输出	MAC 数据发送
MTX_EN1	11	输入	MAC 数据发送使能
MDATA1	12	输入	MAC 数据管理
GPIO0	16	输入/出	普通 IO 口 0
GPIO1	17	输入/出	普通 IO 口 1
GPIO2	18	输入/出	普通 IO 口 2
GPIO3	19	输入/出	普通 IO 口 3
ALARM_X	20	输入	X 轴报警
LMTP_X	24	输入	X 轴机械正限位
LMTM_X	25	输入	X 轴机械负限位
XPM_DIR	26	输出	X 轴方向信号
XPP_PLS	27	输出	X 轴脉冲信号
XPOS	28	DI	X 轴原点信号
ALARM_Y	29	DI	Y 轴报警
LMTP_Y	33	输入	Y 轴机械正限位
LMTM_Y	34	输入	Y 轴机械负限位
YPM_DIR	35	输出	Y 轴方向信号
YPP_PLS	36	输出	Y 轴脉冲信号
YPOS	37	输入	Y 轴原点信号

信号	引脚号	输入/输出	描述
ALARM_Z	38	输入	Z 轴报警
LMT_P_Z	39	输入	Z 轴机械正限位
LMT_N_Z	40	输入	Z 轴机械负限位
ZPM_DIR	44	输出	Z 轴方向信号
ZPP_PLS	45	输出	Z 轴脉冲信号
ZPOS	46	输入	Z 轴原点信号
ALARM_U	47	输入	U 轴报警
LMT_P_U	48	输入	U 轴机械正限位
LMT_N_U	52	输入	U 轴机械负限位
UPM_DIR	53	输出	U 轴方向信号
UPP_PLS	54	输出	U 轴脉冲信号
UPOS	55	输入	U 轴原点信号
GPI04	56	输入/出	普通 I/O 口 4
GPI05	60	输入/出	普通 I/O 口 5
GPI06	61	输入/出	普通 I/O 口 6
GPI07	62	输入/出	普通 I/O 口 7
GPI08	63	输入/出	普通 I/O 口 8
GPI09	64	输入/出	普通 I/O 口 9
GPI010	65	输入/出	普通 I/O 口 10
GPI011	69	输入/出	普通 I/O 口 11
GPI012	70	输入/出	普通 I/O 口 12
GPI013	71	输入/出	普通 I/O 口 13
GPI014	72	输入/出	普通 I/O 口 14
GPI015	73	输入/出	普通 I/O 口 15
GPI016	74	输入/出	普通 I/O 口 16
GPI017	75	输入/出	普通 I/O 口 17
GPI018	76	输入/出	普通 I/O 口 18
GPI019	77	输入/出	普通 I/O 口 19
GPI020	81	输入/出	普通 I/O 口 20
GPI021	82	输入/出	普通 I/O 口 21
GPI022	83	输入/出	普通 I/O 口 22
UART0_RXD	84	输入	串口 0, RX
UART0_TXD	85	输出	串口 0, TX
GPI023	86	输入/出	普通 I/O 口 23
RUN	89	输出	工作指示灯
ERR	90	输出	错误指示灯
POR_BYPASS	99	输入	下拉到地
TESTMODE	100	输入	下拉到地
SOC_RST	101	输入	系统复位
SOC_CLK	102	输入	系统时钟, 一般采用 10MHz 有源晶振
IRQ0	113	输入	急停输入
IRQ1	114	输入	暂停输入
X-AXIS-A	115	输入	编码器输入 A 相, X 轴
Y-AXIS-A	116	输入	编码器输入 A 相, Y 轴
Z-AXIS-A	117	输入	编码器输入 A 相, Z 轴

信号	引脚号	输入/输出	描述
U-AXIS-A	118	输入	编码器输入 A 相, U 轴
PWM-A	122	输出	PWM 输出, A 相
HAHELD-A	123	输入	手轮输入 A 相
X-AXIS-B	124	输入	编码器输入 B 相, X 轴
Y-AXIS-B	125	输入	编码器输入 B 相, Y 轴
Z-AXIS-B	126	输入	编码器输入 B 相, Z 轴
U-AXIS-B	127	输入	编码器输入 B 相, U 轴
PWM-B	128	输出	PWM 输出, B 相
HAHELD-B	132	输入	手轮输入 B 相
HAHELDx1	133	输入	手轮倍率 1 倍选择
HAHELDx10	134	输入	手轮倍率 10 倍选择
HAHELDx100	135	输入	手轮倍率 100 倍选择
HAHELDX	136	输入	手轮 X 轴选择接口
HAHELDY	140	输入	手轮 Y 轴选择接口
HAHELDZ	141	输入	手轮 Z 轴选择接口
HAHELDU	142	输入	手轮 U 轴选择接口
GPI024	143	输入/出	普通 I/O 口 24
GPI025	144	输入/出	普通 I/O 口 25

6 电气特性

表 6-1 电气特性表

参数项	描述	参数值（参考地 VSS）		
		最小值	典型值	最大值
VDDIO	I/O 供电	2.97V	3.3V	3.63V
VDDCORE	内核供电	1.08V	1.2V	1.32V
Vih	输入高电平	2.0V		VDDIO+0.3V
Vil	输入低电平	-0.3V		0.8V
Voh	输出高电平	2.4V		
Vol	输出低电平			0.4V
Ioh	高电平输出电流@Voh=2.4V	9.8mA		35.1mA
IoL	低电平输出电流	8.4mA		16.3mA
IL	输入漏电流			±1uA
Ioz	三态输出漏电流			±1uA

7 控制方法

为了用户可以便捷开发基于运动控制芯片的上层应用，我司为 CMC 运动控制芯片提供了基于 Visual Studio 2019 开发的动态库“motionChip.dll”，用户可以基于该动态库便捷地开发运动控制上层应用。而无需关心底层的通信逻辑。如果用户需要在其他开发环境中使用，我司可提供动态库的源代码，用于只需修改通信接口的实现函数，即可编译使用。

所有的库函数操作提供例程，可联系本公司获得。使用 TCP 进行通信时，一次接口调用在 2ms 内完成（典型值 1ms），使用串口进行通信时（波特率 115200）一次接口调用 6ms 内完成（典型值 5ms）。

CMC693PR144-M 支持三种通讯控制方式，一种是单条指令的一发一收控制，一种是将多条指令连续发送到芯片连续执行，即缓存执行方式。所有的通讯交互都是从上位机发起，包括控制的执行和状态的查询，对应的流程图如下：

7.1 单指令立即执行

单条指令执行的通讯控制流程图如下所示，一般先由上位机与 CMC693PR144-M 芯片建立通讯连接，然后开始按照 API 指令一发一收的方式进行控制通讯，在发送下一条控制指令时，需要先发送状态查询指令，确保上一条指令已经执行完成。

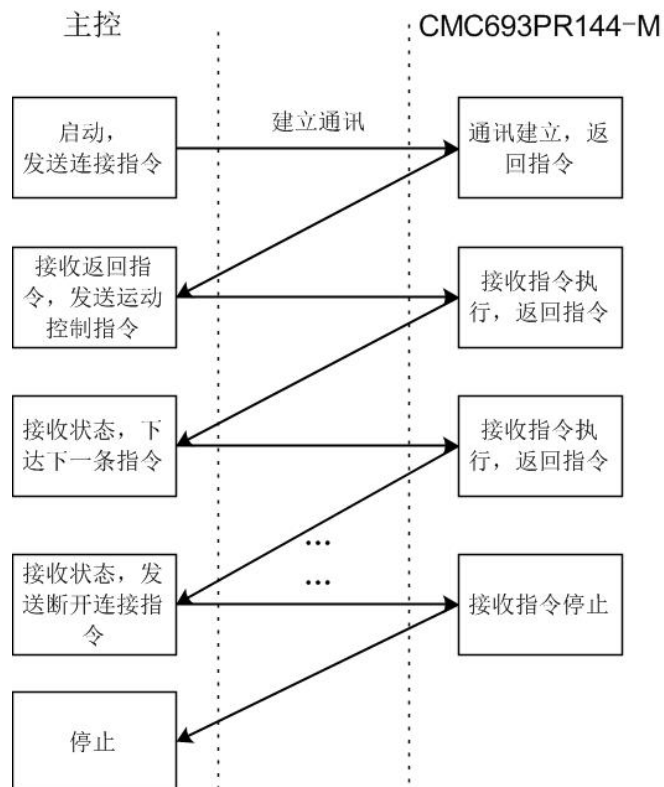


图 7-1 单指令执行示意图

案例程序如下所示：

```

#include <stdint.h>
#include <stdio.h>

#include "cmc_api.h"

int main(void) {
    uint32_t PID;           //产品 ID 号
    uint64_t UID;           //芯片唯一 ID 号
    char     version[256];  //芯片版本信息
    void*     handle = 0;   //定义连接的句柄
    int32_t   result = 0;

    //IP 地址为 192.168.1.7, 端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    //获取版本信息
    result = API_versionInfo(handle, 0, &PID, &UID, version);
    printf("产品的 ID 为: %d, 芯片唯一 ID 号: %lld, 芯片版本信
息: %s", PID, UID, version);
    //快速定位到 10000, 10000, 10000
    result = API_rapidFeed(handle, 0, 10000, 10000, 10000);
    //断开连接
    API_disconnect(handle);
}

```

本例中，采用单指令方式与芯片建立通讯连接，并获取芯片版本信息，同时控制芯片的三个轴快速定位到（10000, 10000, 10000）位置，之后与芯片断开通讯连接，整个控制方式均采用一来一回的方式进行。一般可对每次返回的 result 进行错误码分析，而后再进行相关操作。

7.2 多指令缓存执行

本芯片也支持将一部分指令先压入堆栈，然后发送指令，统一执行的方式，流程图如下图所示。

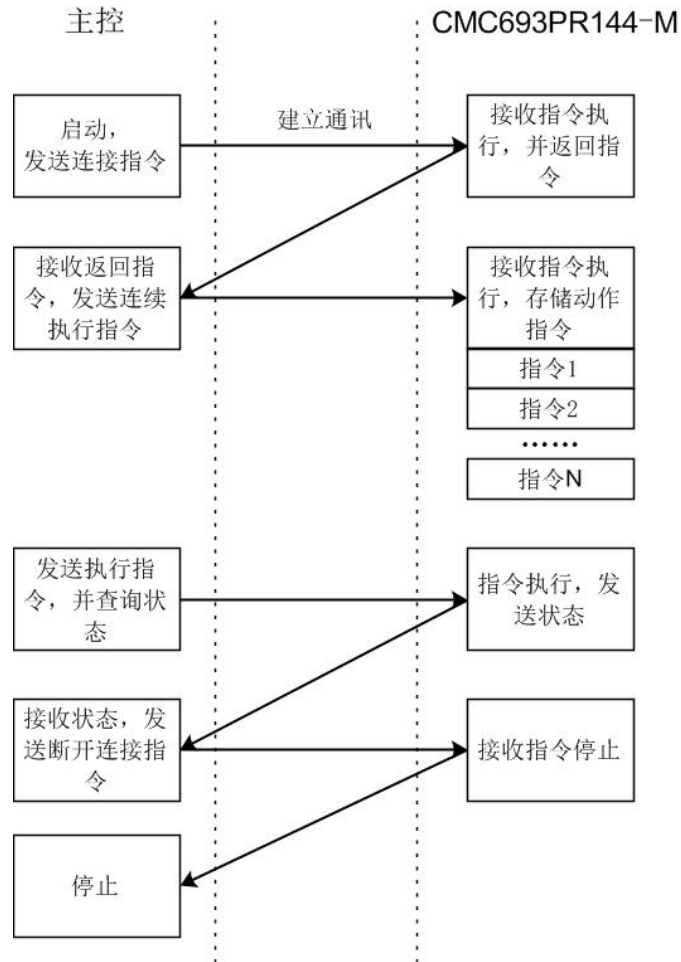


图 7-2 多指令连续执行示意图

案例程序如下所示：

```

#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    void*    handle = 0; //定义连接的句柄
    int32_t  result = 0;
    uint32_t count  = 0; //缓存区计数值

    //IP 地址为 192.168.1.7, 端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }
    //设置缓存区计数器为 0
    result = API_setQueueCounter(handle, 0, 0);

```

```

//缓存执行, 快速定位到 10000, 10000,10000
result = API_rapidFeed(handle, 1, 10000, 10000, 10000);
//缓存执行, 执行直线插补
result = API_linearInterpolation(handle, 1, 20000, 30000, 40000,
2000);
//缓存执行, 执行直线插补
result = API_linearInterpolation(handle, 1, 10000, 10000, 10000,
3000);
//缓存执行, 快速定位到 0,0,0
result = API_rapidFeed(handle, 1, 0, 0, 0);
//设置缓存区计数器为 2, 执行 2 条指令
result = API_setQueueCounter(handle, 0, 2);

do {
    int32_t memoryCost      = 0;
    int32_t remaining       = 0;
    int32_t memoryTotal     = 0;
    int32_t entryTotal      = 0;
    int32_t currCommand     = 0;
    int32_t commandErrorCode = 0;
    //获取缓冲区的相关信息
    result = API_queueInfo(
        handle,
        0,
        &memoryCost,
        &remaining,
        &memoryTotal,
        &entryTotal,
        &currCommand,
        &commandErrorCode);
    printf("内存消耗量%dBByte, 剩余的指令条数%d,内存总量%dBByte, 最多
指令条数%d, 正在执行的指令%d,错误代
码%d\r\n", memoryCost, remaining, memoryTotal, entryTotal, currComma
nd, commandErrorCode);
    Sleep(100); //延时 100 毫秒
    result = API_getQueueCounter(handle, 0, &count);
    printf("缓冲区计数器值为%d\r\n", count);
} while (count != 0);
//缓冲区执行完 2 条后, 停止执行

```

```
API_disconnect(handle); //断开连接
}
```

本例中，在执行缓存运动控制前，先将缓冲区计数器设置为 0，即先不执行写进队列的指令，之后写入了 4 条写入缓冲区的控制指令，并将计数值设置为 2，芯片就会立即开始执行缓冲区中的指令，执行完 2 条后，计数值变为 0，就停止运动。

芯片提供缓冲区操作接口。如下所示，通常在进行缓存控制前，先对缓冲区进行设置，以更好的达到控制效果。

表 7-1 缓冲区操作函数列表

序号	函数名	说明
1	API_queueInfo	获取缓冲区的相关信息
2	API_setQueueCounter	设置缓冲区的指令队列计数器
3	API_getQueueCounter	获取缓冲区的指令队列计数器
4	API_clearQueue	清空缓冲区的所有指令

运动控制芯片内部的指令缓冲区有一个计数器，上电初始值为 0xFFFFFFFF，若设置该计数器的为除 0xFFFFFFFF 外的数值时，每执行一条指令，这个计数器就会减一，当该计数器减到零时，即使缓冲区中有指令，也不会执行。

通过这个计数器可以实现以下逻辑：

- 通过设置该计数器为 0 和 0xFFFFFFFF 来控制缓冲区的指令是否开始执行。
- 通过设置该计数器为“除 0 和 0xFFFFFFFF 以外的值”，控制缓冲区中的指令执行的条数。

例如设置为 5，即使缓冲区中有 10 条指令，执行 5 条指令后也会停止执行。

7.3 安全连接模式

所谓安全连接模式，就是设计了一种更加安全的保证通讯正常的模式，因为在与主控通讯过程中，如果出现通讯异常，被控对象仍然被 CMC 所控制时，容易出现不可预知的风险。

进入安全连接模式后，必须每间隔一段时间和运动控制芯片进行通信，如果在规定时长没有和运动控制芯片进行通信，则认为连接出现异常，立即电机停止运动。实际使用中可以新建一个定时器，每隔一段时间调用一次通信接口，以起到类似喂狗的作用。

退出安全连接模式，只需要将对应间隔时间的值改为 0 即可。

案例程序如下所示：

```
#include <stdint.h>
#include <stdio.h>

#include <windows.h>
```



```

#include "cmc_api.h"



int main(void) {
    void*   handle = 0;
    int32_t result = 0;
    //IP 地址为 192.168.1.7, 端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }
    //进入安全连接模式, 每 500ms 内需与芯片进行通讯
    result = API_safetyConnection(handle, 0, 500);
    //快速定位到 100000, 100000, 100000
    result = API_rapidFeed(handle, 0, 100000, 100000, 100000);
    //延时 1000 毫秒, 由于延时期间没有进行通信, 运动会急停
    Sleep(1000);
    //退出安全连接模式
    result = API_safetyConnection(handle, 0, 0);
    //断开连接
    API_disconnect(handle);
}

```

8 详细功能介绍

CMC 运动控制芯片支持两种脉冲输出模式, 即: (1) 单脉冲模式; (2) 独立 2 脉冲模式。用户可自行配置。独立 2 脉冲方式正方向驱动时, 由 nPP/PLS 输出驱动脉冲, 负方向驱动时, 由 nPM/DIR 输出驱动脉冲。单脉冲模式则由 nPP/PLS 输出驱动脉冲, 由 nPM/DIR 输出方向信号。

模式	脉冲输出方式	驱动方向	输出信号波形	
			nPP/PLS 信号	nPM/DIR 信号
1	独立 2 脉冲方式	+方向驱动输出	正 	0
		-方向驱动输出	0	正 
2	独立 2 脉冲方式	+方向驱动输出	0	正 
		-方向驱动输出	正 	0
3	独立 2 脉冲方式	+方向驱动输出	负 	1
		-方向驱动输出	1	负 
4	独立 2 脉冲方式	+方向驱动输出	1	负 
		-方向驱动输出	负 	1
5	1 脉冲方式	+方向驱动输出	正 	0
		-方向驱动输出	正 	1
6	1 脉冲方式	+方向驱动输出	正 	1
		-方向驱动输出	正 	0
7	1 脉冲方式	+方向驱动输出	负 	1
		-方向驱动输出	负 	0

8	1 脉冲方式	+方向驱动输出	负 	0
		-方向驱动输出	负 	1

脉冲输出模式的设置，可通过 API 函数进行读写，对 API_writeChipConfig 函数的 pulseMode 参数进行设置，即可实现上述几种模式的选择，一般在进行运动控制前，先对该函数进行配置。具体可参考第 8.10 节。

通常在进行运动控制前，需要对运动控制中的共用参数进行设置，比如初速度、加速度、加加速度等，芯片提供了 API_motionParameter 函数进行相关设置。

8.1 单轴运动

8.1.1 可变速运动

对应 X、Y、Z、U 四轴，可变速运动输入信号可分别配置到 4 个外部管脚，在关联完外部引脚后，当外部产生中断或者发送对应指令，会触发该轴以设定的目标速度和方向进行加速运动，运动过程中可以改变目标速度值，CMC 芯片会自动进行加减速控制。直到可变速使能清零，会触发该轴进行减速操作，直到速度降为初速度。如图 8-1 所示。

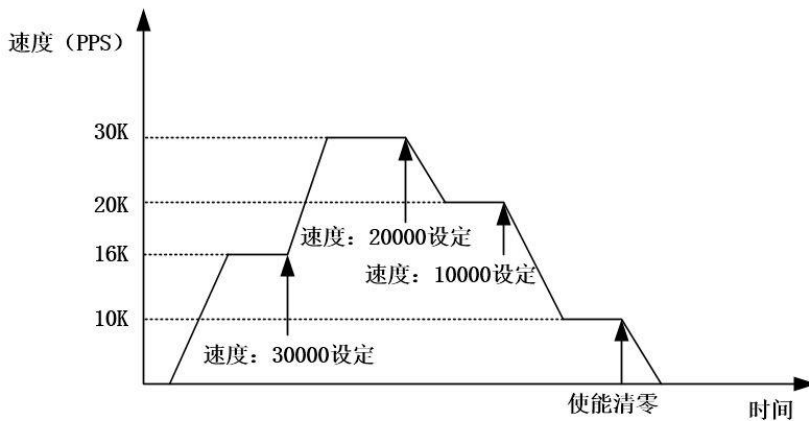


图 8-1 可变速运动

功能使用说明：各轴相互独立，可同时进行；可变速运行使能信号可由外部管脚触发或者通过指令触发调用。

对应 API 函数中的 API_variableSpeed 函数。当配置完轴选、速度、方向等参数后，等待执行命令触发可变速运动。如图 8-1 所示，初次运动 CMC 芯片以 16K 的目标速度驱动选定轴，在驱动过程中，分别修改目标速度，CMC 芯片会自动进行加减速控制。直到该轴的可变速运行使能信号被清 0，CMC 芯片会自动执行减速操作，直到目标速度达到初始速度。

可变速运行根据是否配置有加速度值可以分为有加减速运行和没有加减速运行，没有加减速运行过程中速度是直接跳变的。初速度、加速度、加加速度的配置对应 API 函数中的 API_motionParameter 函数。

案例程序如下所示：

```
#include <stdint.h>
#include <stdio.h>
```

```

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    //定义连接的句柄
    void*    handle = 0;
    int32_t result = 0;
    //IP 地址为 192.168.1.7, 端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }
    //配置运动控制相关参数
    result = API_motionParameter(handle, 0, 1000, 10000, 0);
    //X 轴从 0 加速到 16K 的目标速度
    result = API_variableSpeed(handle, 0, 16000, 1, 1);
    //延时 10 秒
    Sleep(10000);
    //X 轴从 16K 加速到 30K 的目标速度
    result = API_variableSpeed(handle, 0, 30000, 1, 1);
    Sleep(10000);
    //X 轴从 30K 减速到 20K 的目标速度
    result = API_variableSpeed(handle, 0, 20000, 1, 1);
    Sleep(10000);
    //X 轴从 20K 减速到 10K 的目标速度
    result = API_variableSpeed(handle, 0, 10000, 1, 1);
    Sleep(10000);
    //X 轴从 10K 减速到 0 的目标速度
    result = API_variableSpeed(handle, 0, 0, 1, 1);
    Sleep(10000);
    //断开连接
    API_disconnect(handle);
}

```

通常在执行可变速运动前，需要对加速度、最大速度等参数进行设置，然后再调用对应的运动控制函数。在调用不同目标速度时，增加了延时函数，以确保运动到目标速度。

除了延时的方法外，也可通过获取运动状态的方法来判断运动到达目标速度，即采用 API_mcGetStatus 函数，可用来读取当前各个轴的速度。API_mcGetStatus 函数的详细使用方法，请参考《CMC 运动控制芯片库函数编程手册》。

8.1.2 定长运动

对应 X、Y、Z、U 四轴，位置精确控制启动信号由定长运动控制指令控制执行，会触发该轴以设定的目标速度和方向进行加速运动，直到剩余脉冲数少于加速过程消耗脉冲数，会触发该轴进行减速操作，直到速度降为初速度，如下图所示。

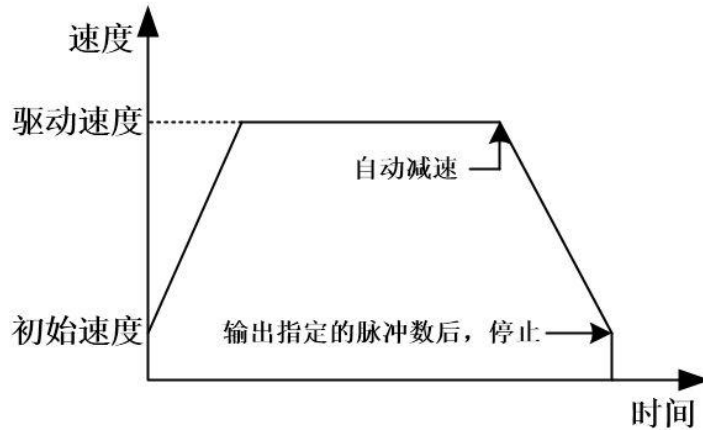


图 8-2 定长运动

功能使用说明：

各轴相互独立，但同一时刻只允许一个轴执行定长运动。

配置完目标速度、加速度、运动方向、脉冲数等参数后，等待执行命令触发单轴位置精确控制运动。当位置精确控制启动信号有效时，CMC 芯片按照设定的目标速度和加速度进行加速运动，当运动过程中剩余脉冲数小于等于加速过程中消耗的脉冲数，自动进行减速操作，直到速度降为初速度，达到目标点，完成位置的精确控制。

案例程序如下所示：

```
#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    void* handle = 0; //定义连接的句柄
    int32_t result = 0;

    //IP 地址为 192.168.1.7，端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }
    //配置运动控制相关参数
    result = API_motionParameter(handle, 0, 1000, 10000, 0);
```

```

//Z 轴执行定长运动，执行长度为 10000 个脉冲，速度为 1000 脉冲/秒。
result = API_fixedLength(handle, 0, 10000, 1000, 4);
//4 轴速度
uint32_t xSpeed = 0;
uint32_t ySpeed = 0;
uint32_t zSpeed = 0;
uint32_t uSpeed = 0;
//4 轴电气坐标
int32_t xEloc = 0;
int32_t yEloc = 0;
int32_t zEloc = 0;
int32_t uEloc = 0;
//4 轴机械坐标
int32_t xMloc = 0;
int32_t yMloc = 0;
int32_t zMloc = 0;
int32_t uMloc = 0;
uint32_t interSpeed = 0; //插补速度
uint32_t statusCode = 0; //状态码
uint32_t alarmFlag = 0; //报警标志
//速度为 0 时，才执行下一步动作
do {
    result = API_mcGetStatus(handle, 0, &xSpeed, &ySpeed, &zSpeed, &uSpeed, &xEloc, &yEloc, &zEloc, &uEloc, &xMloc, &yMloc, &zMloc, &uMloc, &interSpeed, &statusCode, &alarmFlag); //读取运动状态
    printf("Z 轴单轴速度: %d, Z 轴电气坐标: %d, Z 轴机械坐标%d, 插补速度: %d, 状态码: %d, 报警标志: %d\r\n", zSpeed, zEloc, zMloc, interSpeed, statusCode, alarmFlag);
    Sleep(100);
} while (interSpeed != 0);
API_disconnect(handle); //断开连接
}

```

执行完定长运动后，速度会从设置的初速度降到 0，一般情况下，各种运动都需要执行一段时间，建议在执行下一个动作前，通过判断运动状态（如速度、位置等），一般采用 API_mcGetStatus 函数来进行下一步的动作。如上述所示，在断开连接前，需要读取 z 轴的速度是否为 0，来确保定长运动正常执行。

8.1.3 电气原点回归

对应 X、Y、Z、U 四轴，发送对应指令，会触发该轴以设定的目标速度和加速度进行运动，并在当前运动的电气原点处停止。如下图所示。

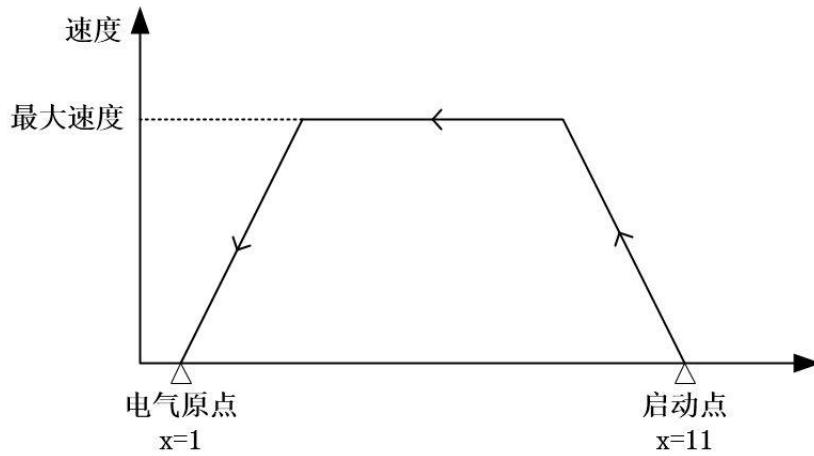


图 8-3 电气原点回归

功能使用说明：

各轴相互独立，可同时进行电气原点回归动作；

配置完当前运动的电气原点、运动方向等参数后，等待执行命令触发电气原点回归运动。如上图所示，假设启动点坐标为 11，电气原点坐标为 1，接收到执行命令后，CMC 芯片会以启动点相对电气原点坐标值的绝对值（ $|1-11|=10$ ）作为电气坐标写入值（电气原点回归输出脉冲总数），以最大速度为目标速度，自动进行加减速控制，直到达到目标点（电气原点）。

另外，可以预先通过参数电气坐标写入值，重新改写当前点对于电气原点的相对坐标，在回归结束后，默认目标点为电气原点，由此达到重新设置电气原点的目的。例如启动点坐标为 11，设置电气坐标写入值为 5，运动方向为负方向，则目标点坐标 6 就可当做下一段单轴运动的电气原点。电气原点回归功能对应 API 函数中的 API_electricalHoming；修改电气坐标功能对应 API 函数中的 API_mcSetPosition。

函数使用举例如下：

```
#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    void*    handle = 0;
    int32_t  result = 0;
    //IP 地址为 192.168.1.7, 端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
```

```

if (handle == 0) { return -1; }
//配置运动控制相关参数
result = API_motionParameter(handle, 0, 1000, 10000, 0);
//设置 X 轴当前电气坐标为 10000
result = API_mcSetPosition(handle, 0, 10000, 0, 0, 0);
//X 轴执行电气回归, 速度为 2000pps
result = API_electricalHoming(handle, 0, 2000, 1);
//延时 1 秒
Sleep(1000);
//断开连接
API_disconnect(handle);
}

```

8.1.4 机械原点回归

对应 X、Y、Z、U 四轴，发送对应指令，会触发该轴以设定的目标速度和方向进行机械原点回归运动，直到检测到相关信号，作相应轴停止操作，即回到了机械原点。

配置完目标速度、运动方向等参数后，等待执行命令触发机械原点回归运动。机械原点回归功能对应 API 函数中的 API_mechanicalHoming。

CMC693PR144-M 芯片支持多种回零控制，回零控制主要是指按照设定方式，每个轴回到指定的原点位置，一般都需要外接接近开关或者光电开关做原点的输入，通常在设备开始工作前，机构的每个轴需要进行一次回零操作，以此作为机械标准位置。目前本芯片支持 2 种回零模式，具体使用方法和原理，见下面说明：

- **回零模式 0（一次回零）：**

电机按选择方向快速运动，感应到原点信号立即停止，如图 8-9 所示。这种回零方式是最直接的回零方式，适用于行程短、速度较慢、安全性要求高的场合，能在检测到原点后立马停止运动。

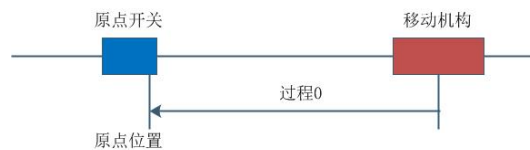


图 8-4 回零模式 0

- **回零模式 1（一次回零加回找）：**

电机按选择方向快速运动，感应到原点信号后过冲减速停止，再正向慢速找原点后停止，如图 8-10 所示。这种回零方式，通常用在对精度有一定要求的场合。



图 8-5 回零模式 1

注：

本芯片默认 X、Y、Z、U 轴机械原点回归的原点信号为 DI0、DI1、DI2、DI3。

8.1.5 应用举例

8.1.5.1 点动功能

在许多场合需要用到这样的功能：在某个按键按下后，电机开始运动，按键松开后，电机停止运动。可以通过可变速运动的功能来实现：

```
void keyDown(void) {
    API_motionParameter(handle, 0, 100, 100000, 0);
    //X 轴以 10000 个脉冲/秒的速度开始运动
    API_variableSpeed(handle, 0, 10000, 1, 1);
}

void keyUp(void) {
    //X 轴减速停止
    API_variableSpeed(handle, 0, 0, 1, 1);
}
```

如需在按键按下后，指定某个轴运动指定长度，每按一次运动一次。则可以通过如下方式来实现：

```
void keyDown(void) {
    API_motionParameter(handle, 0, 100, 100000, 0);
    API_fixedLength(handle, 0, 100000, 1000, 1); //X 轴执行定长运动，执行长度为 100000 个脉冲，速度为 1000 脉冲/秒。
}
```

8.2 多轴运动

8.2.1 坐标系选择

可以选择 X、Y、Z、U 四轴中的任意两轴或者三轴建立一个坐标系，两轴坐标系为一个平面，对应 API 函数中的 API_planeSelect；三轴坐标系为一个空间，对应 API 函数中的 API_spaceSelect。在 G 代码中坐标系的选择为 G17~G26。

以下是单轴、二轴平面、三轴空间的选择说明：

表 8-1 轴、平面和空间选择列表

单轴选择		平面选择		空间选择	
axisX	0b0001	PlaneXY	0b0011	SpaceXYZ	0b0111
axisY	0b0010	PlaneXY	0b0101	SpaceXYU	0b1011
axisZ	0b0100	PlaneXY	0b0110	SpaceXZU	0b1101
axisU	0b1000	PlaneXY	0b1001	SpaceYZU	0b1110
		PlaneXY	0b1010		
		PlaneXY	0b1100		

8.2.2 坐标模式选择

在执行多轴运动时，需要选择当前坐标模式。绝对坐标模式，对应 API 函数中的 `API_absoluteDistance`；相对坐标模式，对应 API 函数中的 `API_incrementalDistance`。绝对坐标模式表示多段运动中，每段运动的原点都是初始设定的电气原点；相对坐标模式表示多段运动中，每段运动的原点，都是上一段运动的终点。函数具体功能，请参看《运动控制芯片库函数编程手册》。

8.2.3 直线插补

对应 X、Y、Z、U 四轴中的三维坐标系，可以通过总线改写运动控制器相应寄存器，从而触发运动控制器以设定的起点，初速度、加速度、目标速度、减速度、目标点等参数进行运动，最终通过拟合得到完整的直线插补运动过程。如下图所示：o 表示运动开始起点，A 表示运动结束终点， V_x, V_y, V_z 分别表示 X, Y, Z 轴运动过程中的速度。

现在直线插补可支持的功能有，从当前位置，设置目标位置和速度，即会运行到对应指定位置。选中三轴直线插补到某一个位置，可以配置插补的速度。相当于 G 代码中的 G01。

对应 API 函数中的 `API_linearInterpolation` 函数，如果需要运行直线插补，除了配置 `API_linearInterpolation` 函数中的目标位置和目标速度，还需要选定坐标系、坐标模式、配置初速度、加速度、加加速度等参数。例如坐标系选择 X、Y、Z 坐标系，坐标模式选择相对坐标模式，初速度为 100 pps，目标速度为 5000 pps，加速度为 1000 pps²，目标位置为 (1000, 1000, 1000)，则表示 CMC 芯片会以 100 的初始速度，1000 的加速度，5000 的目标速度执行直线插补，直到目标位置达到 (1000, 1000, 1000)。

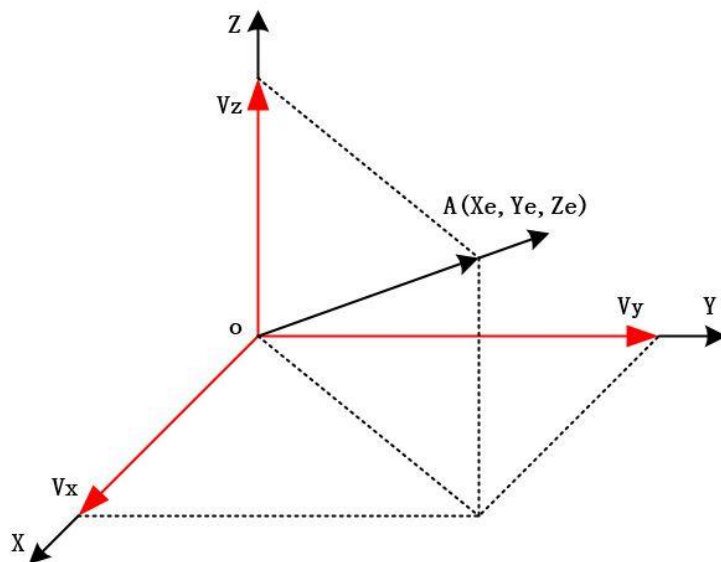


图 8-9 直线插补

对应调用函数如下：

```
#include <stdint.h>
#include <stdio.h>
```

```
#include <windows.h>

#include "cmc_api.h"

int main(void) {

    void*    handle = 0;
    int32_t  result = 0;

    //IP 地址为 192.168.1.7, 端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }
    //配置运动控制相关参数
    result = API_motionParameter(handle, 0, 100, 1000, 0);
    //选择 XYZ 轴执行直线插补
    result = API_spaceSelect(handle, 0, 7);
    //采用相对坐标模式
    result = API_incrementalDistance(handle, 0);
    //按照 5000 的插补速度, 运动到 (1000,1000,1000) 位置
    result = API_linearInterpolation(handle, 0, 1000, 1000, 1000, 5000);

    //4 轴速度
    uint32_t xSpeed = 0;
    uint32_t ySpeed = 0;
    uint32_t zSpeed = 0;
    uint32_t uSpeed = 0;
    //4 轴电气坐标
    int32_t xEloc = 0;
    int32_t yEloc = 0;
    int32_t zEloc = 0;
    int32_t uEloc = 0;
    //4 轴机械坐标
    int32_t xMloc = 0;
    int32_t yMloc = 0;
    int32_t zMloc = 0;
    int32_t uMloc = 0;
    //插补速度
    uint32_t interSpeed = 0;
```

```

//状态码
uint32_t statusCode = 0;
//报警标志
uint32_t alarmFlag = 0;
//速度为0时，才执行下一步动作
do {
    result = API_mcGetStatus(handle, 0, &xSpeed, &ySpeed, &zSpeed,
    &uSpeed, &xEloc, &yEloc, &zEloc, &uEloc, &xMloc, &yMloc, &zMloc,
    &uMloc, &interSpeed, &statusCode, &alarmFlag); //读取运动状态
    Sleep(1000);
} while (interSpeed != 0);

API_disconnect(handle); //断开连接
}

```

8.2.4 圆弧插补

圆弧插补只能支持平面，不支持空间，所以对应 X、Y、Z、U 四轴中的二维坐标系，可以通过总线改写运动控制器相应寄存器，从而触发运动控制器以设定的起点、半径、初速度、加速度、目标速度、减速度、目标点等参数进行运动，最终通过拟合得到完整的圆弧插补运动过程。如下图所示：A 表示圆弧插补开始起点，B 表示圆弧插补结束终点，V 表示切线速度， V_x ， V_y 分别表示 X，Y 轴上速度。（以 o 为圆心，A 为起点，B 为终点的圆有下图所示四分之一圆，也有四分之三圆，圆弧运动方向需要通过寄存器配置）

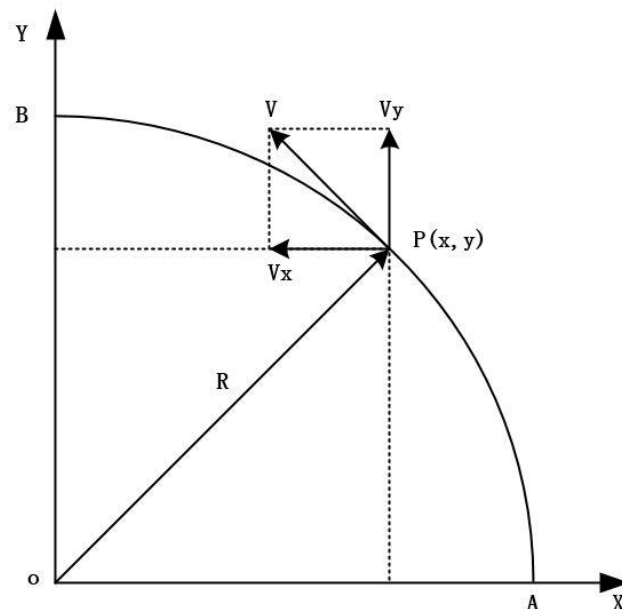


图 8-10 圆弧插补

本芯片可实现的圆弧插补功能为：

选中两轴顺圆插补到某一个位置，可以配置插补的速度。相当于 G 代码中的 G02。对

应 API 函数中的 API_clockwiseCircle 函数。

选中两轴逆圆插补到某一个位置，可以配置插补的速度。相当于 G 代码中的 G03。对应 API 函数中的 API_counterclockwiseCircle 函数。

以顺圆插补为例，和直线插补一样，如果需要运行圆弧插补，除了配置 API_clockwiseCircle 函数中的圆心坐标、目标位置和目标速度之外，必须选定坐标系、坐标模式、配置初速度等参数。同时坐标系的选择必须固定在平面，即对应 G 代码中的 G17~G22。例如选择坐标系为 X、Y 坐标系，坐标模式选择相对坐标模式，初速度为 100 pps，目标速度为 5000 pps，加速度为 1000 pps²，当前坐标为 (0, 0)，目标位置为 (1000, 1000)，圆心坐标为 (1000, 0)，则表示 CMC 芯片会以 (1000, 0) 为圆心，1000 为半径，以 100 的初始速度，1000 的加速度，5000 的目标速度执行圆弧插补，直到目标位置达到 (1000, 1000)。

CMC693PR144-M 支持 4 轴里面任意单轴组成插补加工面实现单轴插补运动，也支持任意 2 轴组成插补加工平面，执行平面插补运动，也支持任意 3 轴组成空间插补加工平面，执行空间插补运动。需注意的点，当前坐标到圆心的距离要和目标坐标到圆心的距离相同。

对应调用函数如下：

```
#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {

    void*    handle = 0; //定义连接的句柄
    int32_t  result = 0;

    //IP 地址为 192.168.1.7，端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }

    //配置运动控制相关参数
    result = API_motionParameter(handle, 0, 100, 1000, 0);
    //选择 XY 轴执行圆弧插补
    result = API_planeSelect(handle, 0, 3);
    //采用相对坐标模式
    result = API_incrementalDistance(handle, 0);
    //按照 5000 的插补速度，运动到 (1000,1000) 位置
    result = API_clockwiseCircle(handle, 0, 1000, 0, 1000, 1000, 5000);

    //4 轴速度
```

```

uint32_t xSpeed = 0;
uint32_t ySpeed = 0;
uint32_t zSpeed = 0;
uint32_t uSpeed = 0;
//4 轴电气坐标
int32_t xEloc = 0;
int32_t yEloc = 0;
int32_t zEloc = 0;
int32_t uEloc = 0;
//4 轴机械坐标
int32_t xMloc = 0;
int32_t yMloc = 0;
int32_t zMloc = 0;
int32_t uMloc = 0;
//插补速度
uint32_t interSpeed = 0;
//状态码
uint32_t statusCode = 0;
//报警标志
uint32_t alarmFlag = 0;

//速度为0时，才执行下一步动作
do {
    result = API_mcGetStatus(handle, 0, &xSpeed, &ySpeed, &zSpeed, &uSpeed, &xEloc, &yEloc, &zEloc, &uEloc, &xMloc, &yMloc, &zMloc, &uMloc, &interSpeed, &statusCode, &alarmFlag); //读取运动状态
    Sleep(1000);
} while (interSpeed != 0);

API_disconnect(handle); //断开连接
}

```

8.3 速度模式

除 8.1.1 中所介绍的可变速运动以外，其他运动控制都支持以下三种速度模式。

8.3.1 定速

定速驱动就是以一成不变的速度输出驱动脉冲。如果设定驱动速度小于初始速度，就没有加/减速驱动，而是定速驱动。

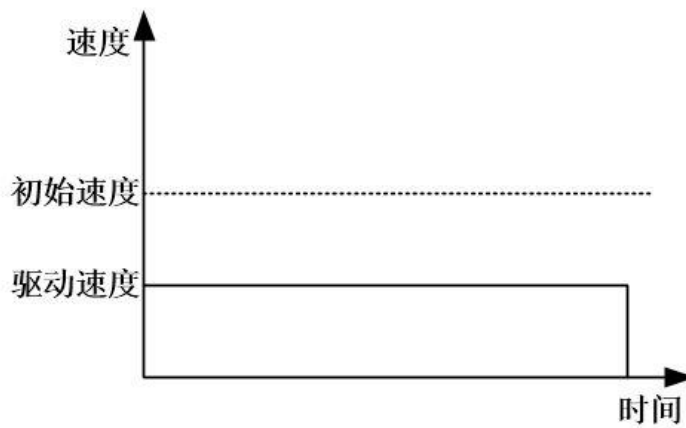


图 8-11 定速驱动

定速运动举例如下：

```
#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    //定义连接的句柄
    void* handle = 0;
    int32_t result = 0;
    //IP 地址为 192.168.1.7，端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }
    //配置初速度为 10000pps
    result = API_motionParameter(handle, 0, 10000, 1000, 0);
    //目标速度为 2000pps
    result = API_fixedLength(handle, 0, 10000, 2000, 1);
    //断开连接
    API_disconnect(handle);
}
```

上述以定长运动进行举例，设置的初速度为 10000pps，而定长的目标速度为 2000pps，小于初始速度，所以指定轴会按照目标速度 2000pps 定速运行 10000 个脉冲后停止。除了定长运动，上面介绍的直线、圆弧插补运动均可支持该方式。

8.3.2 直线加减速

直线加/减速驱动是线性地从驱动开始的初始速度加速到指定的驱动速度。驱动开始后，加速的计数器记录加速所累计的脉冲数，当剩余输出脉冲数少于加速累计脉冲后就开始减速，自动减速时，将用指定的减速度线性地减速至初始速度。加速中命令减速停止或输出脉冲数少于加速至驱动速度所要的脉冲数，就要在加速过程中开始减速。如下图所示。

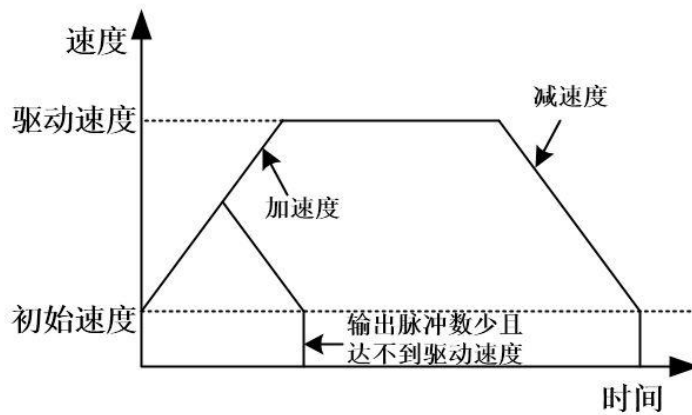


图 8-12 直线加减速图示

直线加减速举例如下：

```
#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    void* handle = 0; //定义连接的句柄
    int32_t result = 0;

    //IP 地址为 192.168.1.7，端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    //配置采用直线加减速
    result = API_motionParameter(handle, 0, 1000, 10000, 0);
    //按照设定参数走定长（直线加减速）
    result = API_fixedLength(handle, 0, 100000, 20000, 1);

    //4 轴速度
    uint32_t xSpeed = 0;
    uint32_t ySpeed = 0;
    uint32_t zSpeed = 0;
```

```

uint32_t uSpeed = 0;
//4 轴电气坐标
int32_t xEloc = 0;
int32_t yEloc = 0;
int32_t zEloc = 0;
int32_t uEloc = 0;
//4 轴机械坐标
int32_t xMloc = 0;
int32_t yMloc = 0;
int32_t zMloc = 0;
int32_t uMloc = 0;
//插补速度
uint32_t interSpeed = 0;
//状态码
uint32_t statusCode = 0;
//报警标志
uint32_t alarmFlag = 0;
//速度为 0 时，才执行下一步动作
do {
    result = API_mcGetStatus(handle, 0, &xSpeed, &ySpeed, &zSpeed,
    &uSpeed, &xEloc, &yEloc, &zEloc, &uEloc, &xMloc, &yMloc, &zMloc,
    &uMloc, &interSpeed, &statusCode, &alarmFlag); //读取运动状态
    Sleep(1000);
} while (interSpeed != 0);

API_disconnect(handle); //断开连接
}

```

上述例子中，在运动参数配置中，加加速度设置为 0，所以指定轴会采用直线加减速方式运行。

8.3.3 S 曲线加减速

如下图所示，整个过程为 7 段式 S 曲线加减速运动，其中 T1 段为加速度增加的加速运动（加加速）；T2 段为匀加速运动（匀加速）；T3 段为加速度减小的加速运动（减加速）；T5 段为减速度增加的减速运动（加减速）；T6 段为匀减速运动（匀减速）；T7 段为减速度减小的减速运动（减减速）；当速度达到驱动速度时，T4 为匀速运动。

整段运动过程，加速度的变化都是以 Jerk（加加速度/减加速度）的斜率匀速变化，其中 T2 段达到加速度的最大值，T6 段达到减速度的最大值。

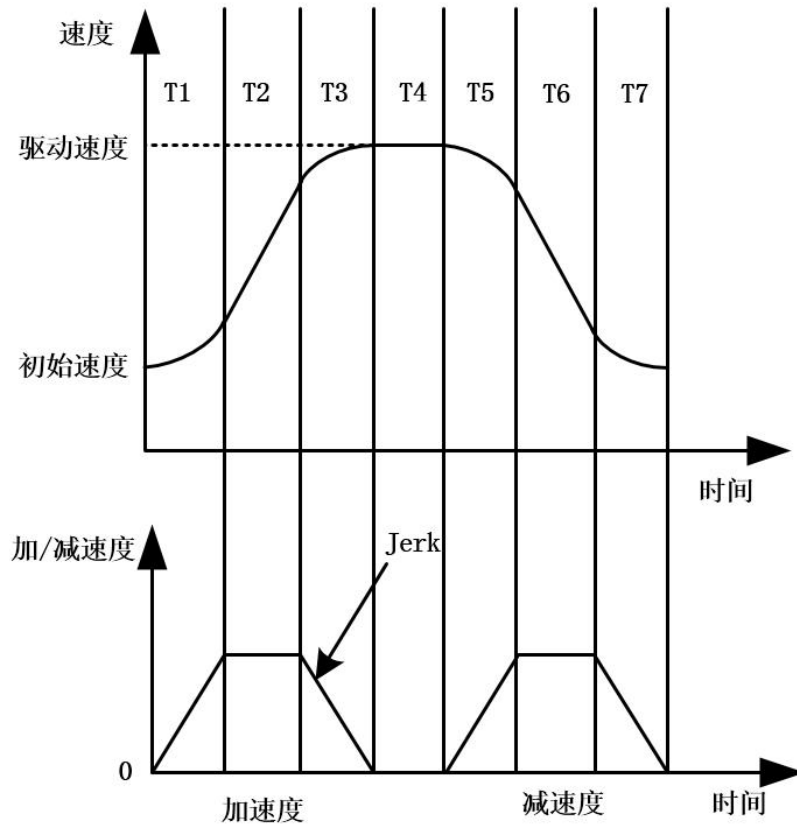


图 8-13 S 曲线加减速图示

S 曲线对应调用函数如下:

```
#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    void* handle = 0; //定义连接的句柄
    int32_t result = 0;

    //IP 地址为 192.168.1.7, 端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }
    //配置运动控制相关参数, 采用 S 曲线
    result = API_motionParameter(handle, 0, 1000, 10000, 100000);
    //按照设定参数走定长 S 曲线
    result = API_fixedLength(handle, 0, 100000, 20000, 1);

    //4 轴速度
```

```

uint32_t xSpeed = 0;
uint32_t ySpeed = 0;
uint32_t zSpeed = 0;
uint32_t uSpeed = 0;
//4 轴电气坐标
int32_t xEloc = 0;
int32_t yEloc = 0;
int32_t zEloc = 0;
int32_t uEloc = 0;
//4 轴机械坐标
int32_t xMloc = 0;
int32_t yMloc = 0;
int32_t zMloc = 0;
int32_t uMloc = 0;
uint32_t interSpeed = 0; //插补速度
uint32_t statusCode = 0; //状态码
uint32_t alarmFlag = 0; //报警标志
//速度为0时，才执行下一步动作
do {
    result = API_mcGetStatus(handle, 0, &xSpeed, &ySpeed, &zSpeed, &uSpeed, &xEloc, &yEloc, &zEloc, &uEloc, &xMloc, &yMloc, &zMloc, &uMloc, &interSpeed, &statusCode, &alarmFlag); //读取运动状态
    printf("Z 轴单轴速度: %d, Z 轴电气坐标: %d, Z 轴机械坐标%d, 插补速度: %d, 状态码: %d, 报警标志: %d\r\n", zSpeed, zEloc, zMloc, interSpeed, statusCode, alarmFlag)
;
    Sleep(100);
} while (interSpeed != 0);

API_disconnect(handle); //断开连接
}

```

在执行定长运动时，S 曲线的参数设置需要满足以下要求，否则无法执行 S 曲线运动。

$$T1 = T3 = T5 = T7 = A / J$$

$$T2 = T6 = (Vt - Vs) / Amax - T1$$

$$T4 = [L - Vs(2T1 + T2)] / Vmax - (2T1 + T2)$$

如图 8-18 所示，其中 A 为加速度设定值，J 为加加速度，Vt 为设定的目标速度，Vs 为初速度，L 为 S 曲线的面积，即长轴走过的距离。公式中要求 T2>0 且 T4>0。

8.4 电子齿轮

在伺服系统中，利用电子齿轮功能可以传递同步运动信息、实现坐标联动、运动形式之间的变换、简化控制、增加传动系统的柔性、减少传动元件数量和传动链长度，还可以实现小数传动比等等，这样就提高了传动精度。如下图即为电子齿轮的同步关系图。

注：该版本没有偏置参数。

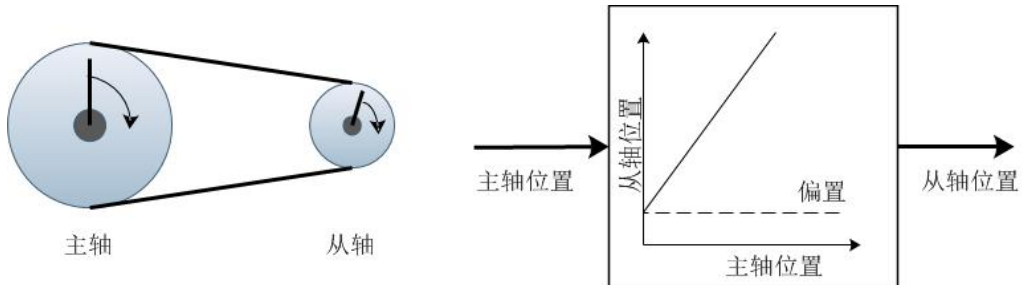


图 8-14 电子齿轮的同步关系图

电子齿轮可以完成主轴与从轴间位置的线性传递功能，在 CMC693PR144-M 中，从轴和主轴的关系如下：

$$S_{\text{slave}} = Gr * S_{\text{master}} + S_b$$

其中， S_{slave} 是从轴的位置， Gr 为电子齿轮比， S_{master} 是主轴的位置， S_b 是偏置距离。所谓的偏置距离，即本芯片支持偏移一段距离后再进行电子齿轮的同步运行。就目前的全数字伺服系统而言，电子齿轮比需满足： $0.01 \leq A/B \leq 100$ 。

启用电子齿轮的方法如下，这里以主轴从轴齿轮比为 1/2 时（暂不考虑脉冲当量）进行说明：

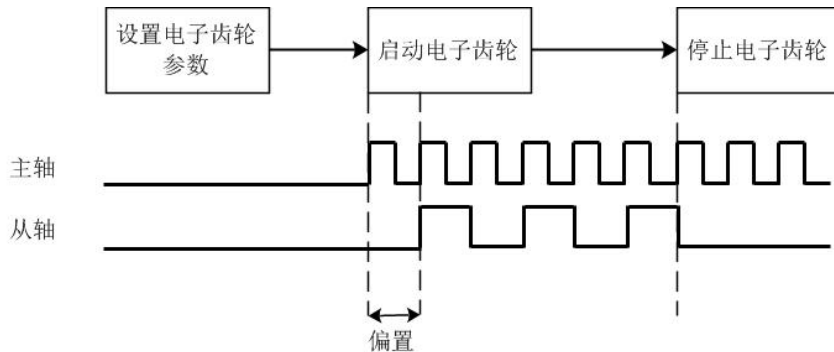


图 8-15 启用电子齿轮方法

电子齿轮的设置，可通过调用 API_writeChipConfig 函数进行配置。

8.5 PWM 输出

运动控制芯片支持输出一路互补的 PWM（脉冲宽度调制），支持动态修改频率、占空比，支持前后死区配置，最大脉冲频率为 1000000Hz。当设置 PWM 的占空比之后，对应的通道 A 和通道 B 就会输出互补的两路信号，如下图所示：

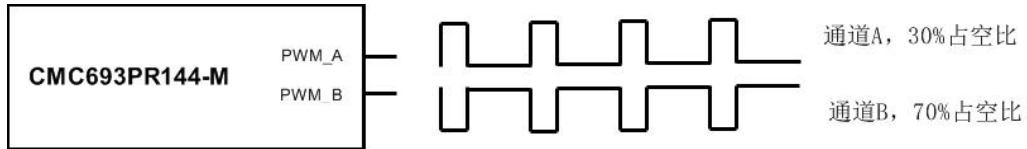


图 8-16 PWM 输出示意图

案例程序如下所示：

```
#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    void* handle = 0; //定义连接的句柄
    int32_t result = 0;
    //IP 地址为 192.168.1.7, 端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }
    //配置运动控制相关参数
    result = API_motionParameter(handle, 0, 1000, 10000, 0);
    //通道 A 和 B 输出 50%占空比 PWM 波
    result = API_pwmControl(handle, 0, 10000, 5000, 5000, 0, 0, 0, 0);
};

//断开连接
API_disconnect(handle);
}
```

8.6 编码器输入

运动控制的每个轴都有一个增量式编码器输入接口，接口支持 A/B 相正交信号，可用于实时监控各轴的运动位置。编码器的正反向通过正交信号的先后相位进行判断，CMC 芯片最高支持 1MHz 的编码器制芯片输入信号频率。

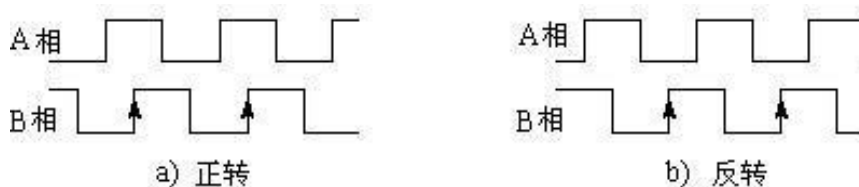


图 8-17 编码器输入信号

芯片提供编码器读取、复位 API 接口，可对编码器数据进行读取和清零操作。主要函数如下：

表 8-2 编码器操作函数列表

序号	函数名称	说明
1	API_resetEncoder	清空编码器计数值
2	API_readEncoder	读取编码器的数值

8.7 手轮控制

运动控制芯片支持一路手轮输入，可支持手轮直接选择轴和倍率，也可通过 API 函数对关联轴进行设置，可支持 X/Y/Z/U 共 4 个轴的选择，倍率选择范围为 x1~x100。同时需要对手轮轴的脉冲当量进行设置。

使能手轮后，关联轴会随着手轮的转动实现位置的跟随，即在 x1 档，手轮转动一格，对应轴就前进 1um（0.1 丝）。不过，这个关系并不是强制的，可根据用户实际的使用方便设置合适的脉冲当量。

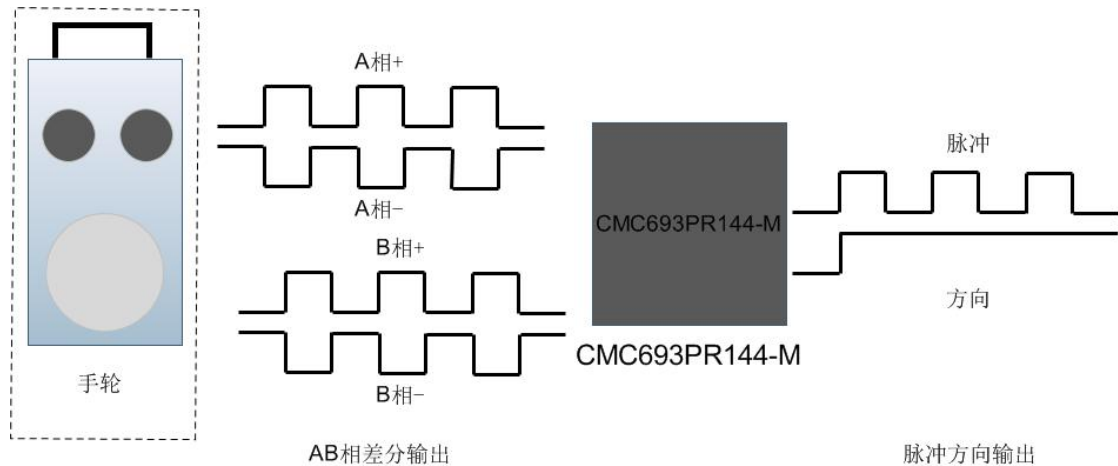


图 8-18 手轮控制示意图

用户可通过调用手轮控制 API 函数，使某个轴根据手轮的操纵做速度跟随，具体使用方法如下：

```
#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    void* handle = 0; //定义连接的句柄
    int32_t result = 0;
    //IP 地址为 192.168.1.7，端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
```

```

if (handle == 0) { return -1; }

//配置运动控制相关参数
result = API_motionParameter(handle, 0, 1000, 10000, 0);
//使能手轮控制，使 Y 轴根据手轮做跟随运动
result = API_handwheel(handle, 0, 1, 1, 1);
}

```

其中，API_handwheel 函数中可对手轮增益和手轮轴的选择进行设置，当设置手轮轴参数为 0 时，芯片会直接按照硬件 HAHELD_{x1}、HAHELD_{x10}、HAHELD_{x100} 接口和对应轴选择引脚的电平情况执行对应的倍率配置和轴选择，如果设置为其他值时，增益设置为 1-100，轴按照 8.2.1 所设定的选择，那么芯片就按照软件设置的数据进行执行。

8.8 IO 控制

运动控制芯片除了运动控制功能、通讯功能外，还提供了总共 26 个 GPIO 口，可进行输入输出的功能配置。可以用于驱动器报警解除、使能控制、开关等功能，能够适用于大部分场合。

8.8.1 普通 IO

所有 GPIO 均可配置为输入和输出，在正常使用 GPIO 前，需要对 GPIO 的输入输出特性进行配置，也包括默认电平情况。本芯片共有 26 个 GPIO 口，对应的编号如下：

表 8-3 GPIO 编号对应表

序号	引脚名称	程序编号
1	GPIO0	0x00
2	GPIO1	0x01
.....
26	GPIO25	0x19

为了方便用户在后续 IO 读写上的可操作性，本芯片同步提供了 DI/DO 重命名的功能，可对配置完输入输出的 IO 口重新命名，以更好的适配后续程序的读写。

上述配置功能通过 IO 配置函数“API_ioModeConfig”实现。具体举例如下：

```

#include <stdint.h>
#include <stdio.h>

#include <windows.h>

#include "cmc_api.h"

int main(void) {

```

```

void*   handle = 0;
int32_t result = 0;
//IP 地址为 192.168.1.7, 端口号为 6666
handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
if (handle == 0) { return -1; }

result = API_ioModeConfig(handle, 0, 0, 1, 0, 0); //配置 GPIO0 为
DI 的模式, 编号为 DI0
result = API_ioModeConfig(handle, 0, 1, 2, 0, 0); //配置 GPIO1 为
DO 的模式, 编号为 D00, 默认输出低电平
result = API_ioModeConfig(handle, 0, 2, 1, 1, 0); //配置 GPIO2 为
DI 的模式, 编号为 DI1
result = API_ioModeConfig(handle, 0, 3, 2, 1, 1); //配置 GPIO3 为
DO 的模式, 编号为 D01, 默认输出高电平
uint32_t diValue = 0;
result          = API_readSingleDi(handle, 0, 0, &diValue); //
读取 DI0, 即 GPIO0 的电平值
result          = API_readSingleDi(handle, 0, 1, &diValue); //
读取 DI1, 即 GPIO2 的电平值

result = API_writeSingleDo(handle, 0, 0, 0); //向 D00, 即
GPIO1, 写入低电平
result = API_writeSingleDo(handle, 0, 1, 1); //向 D01, 即
GPIO3, 写入高电平

uint32_t doValue = 0;
result          = API_readSingleDo(handle, 0, 0, &doValue); //
读取 D00, 即 GPIO1 的输出电平值
result          = API_readSingleDo(handle, 0, 1, &doValue); //
读取 D01, 即 GPIO3 的输出电平值

result = API_readMultiDi(handle, 0, &diValue); //读取 DI 的值, bit0
为 DI0, bit1 为 DI1 以此类推
result = API_writeMultiDo(handle, 0, 0b01); //写入 DO 的值, bit0
为 D00, bit1 为 D01 以此类推
result = API_readMultiDo(handle, 0, &doValue); //读取 DO 的值, bit0
为 D00, bit1 为 D01 以此类推

API_disconnect(handle); //断开连接
}

```

注：

1) 引脚名称必须从 0 开始重命名，且一次完成所有 DI/D0 配置；默认为 18 个 DI，8 个 D0。切记：不要将 GPIO 的设置冲突，即不要将某个 GPIO 既设置成 DI 又设置成 D0，以及将几个 GPIO 设置成相同的名称。

2) 重命名后，未配置的 DI/D0 读取无效，如重命名了 DI0~DI10，则读取 DI11 开始的值无效，D0 相同。

3) 本芯片默认 X、Y、Z、U 轴机械原点回归的原点信号为 DI0、DI1、DI2、DI3，重命名后请注意接线，机械原点回归功能详见章节 8.1.4。

本芯片支持单个通道的输入输出，也支持多个通道同时进行输出控制和输入读取，其主要包括以下几个常用 API：

表 8-4 IO 操作函数列表

序号	函数名称	说明
1	API_readSingleDi	读取单个 DI 口输入
2	API_writeSingleDo	写入单个 D0 口输出
3	API_readSingleDo	读取单个 D0 口输出
4	API_readMultiDi	读取全部 DI 口输入
5	API_writeMultiDo	写入全部 D0 口输出
6	API_readMultiDo	读取全部 D0 口输出

8.8.2 特殊 IO

CMC 芯片有 2 个 IO 口指定用于急停、暂停功能。

当按下急停按钮后，CMC 芯片会立刻停止输出，运动控制状态全部停止。但是对应的位置信息等状态保持当前值，在按钮恢复之后，需要重新发送运动控制指令，才能让芯片重新开始运动。急停功能说明如下所示：

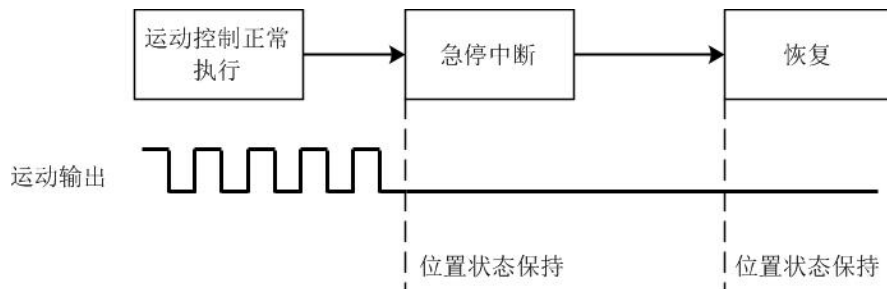


图 8-19 急停功能说明示意图

暂停功能目前仅用于插补运动，当按下暂停按钮后，CMC 芯片也减速暂停，位置状态等信息在停止后保持当前值，在按钮恢复后，CMC 芯片又会重新开始之前的运动，对应的位置信息等状态也同步实时更新。暂停功能说明如下所示：

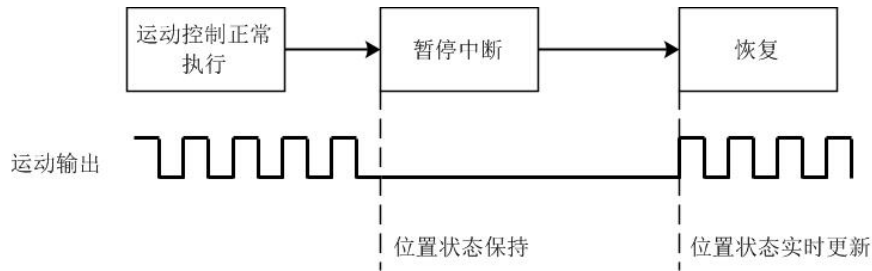


图 8-20 暂停功能说明示意图

需要说明的一点，从暂停状态恢复后，芯片会继续按照之前设定的加速度从 0 开始加速到设定速度，而不是立刻输出最快速度。下面以插补运动为例进行说明，如下图所示：

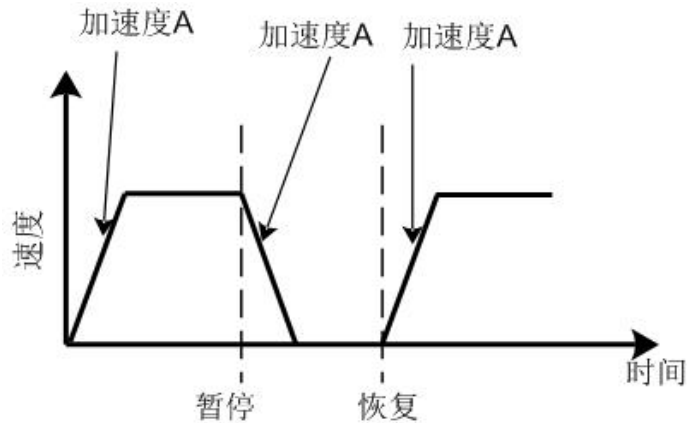


图 8-21 暂停恢复后速度变化示意图

8.9 复位

CMC693PR144-M 支持硬件复位和软件复位两种复位方式。

8.9.1 硬件复位

硬件复位主要是指系统的上电复位和在运行过程中的按键输入复位，输入引脚为 SOC_RST。复位后，芯片恢复到初始状态。硬件复位的时序要求详见第 11 章。

8.9.2 软件复位

软件复位即通过 API 指令对芯片进行复位，复位包括三种：

第一种是复位芯片，相当于断电重启，复位后上位机与 CMC 芯片的通讯会中断，所以上位机上需要注意断开连接并重新建立连接，否则会有内存泄漏的风险。

第二种是复位运动控制单元，该复位仅仅将运动控制相关的功能重置，其余功能不受影响，原先的连接仍然有效。

第三种也是复位运动控制单元，但是不清除当前坐标。

对应的函数应用如下：

```
#include <stdint.h>
#include <stdio.h>
```

```

#include <windows.h>

#include "cmc_api.h"

int main(void) {
    void*    handle = 0;
    int32_t result = 0;
    //IP 地址为 192.168.1.7，端口号为 6666
    handle = API_connectViaEth("192.168.1.7", 6666, 0, 0);
    if (handle == 0) { return -1; }
    //复位运动控制单元，并清除当前坐标
    result = API_chipReset(handle, 0, 1);
    API_disconnect(handle); //断开连接
}

```

8.10 配置及状态的读写

CMC 芯片支持芯片配置的读写、运行状态读取。

8.10.1 芯片配置

芯片配置主要包括芯片的网口 IP 地址、MAC 地址、串口的波特率、限位信号选择、脉冲当量等参数，CMC 芯片提供了芯片配置信息的设置和读取的接口，用户可以能够对芯片基础配置进行读写操作。函数列表如下：

表 8-5 芯片状态函数列表

序号	函数名	说明
1	API_readChipConfig	读取芯片的配置数据
2	API_writeChipConfig	写入芯片配置数据，并保存到芯片的 FLASH 中

需要注意的一点是，配置数据为芯片上电后的默认数据，写入后需要重启后才能生效，由于 FLASH 的写入寿命有限，该接口不可频繁调用。

8.10.2 运动状态

包括各轴速度，各轴电气坐标，各轴机械坐标，当前插补速度，当前的状态码，当前的报警标志，主要接口如下：

表 8-6 运动状态函数列表

序号	函数名	说明
1	API_mcGetStatus	获取运动控制芯片的状态

8.10.3 报警及错误状态

每个 API 调用后都会有返回值，该返回值表示本次调用的结果（缓存执行的返回值只表示当前指令是否成功置于缓冲区，并不表示实际执行结果，执行结果需要后续通过上述

接口查询)

返回值的含义如下:

表 8-7 错误码说明

错误码	说明
0	执行成功。
-1	参数错误, 请检查输入的参数是否正确。
-2	未知错误。
-3	缓冲队列已满。
-4	运动控制单元忙碌, 上一条的运动控制指令还在执行。
-5	运动控制单元异常。
-6	指令不支持。
-7	错误的 CRC 校验码。
-8	通信超时。
-9	连接断开。
-10	无效句柄。

CMC 芯片在以下情况时会产生报警, 报警产生时, 所有运动立即停止, 可以通过 API_mcGetStatus 中的 alarmFlag 的某一位被置位来获取具体哪一种报警被触发:

- 插补运动或者单轴运动时触发限位开关: 越限报警
- 伺服驱动器出现异常: 驱动器报警
- 电子齿轮出错:

如果电子齿轮比 A/B 中 B 的值为零, 则输出报警。如果电子齿轮输出脉冲过程中, 在高电平维持时间内, 接下来触发的脉冲输出也拉高输出信号, 即电子齿轮输出脉冲高低电平维持时间不合理则输出报警, 说明问题出在倍频系数太高或者高电平维持时间过长。

错误指示灯:

芯片的报警指示灯也内置了相关指示作用, 如产生以下问题时, 对应的指示灯会有长短亮灭的变化。

表 8-8 指示灯错误说明表

指示灯状态	含义
电源指示灯不亮	电源异常, 请检查供电
电源指示灯亮, 运行指示灯闪烁	正常运行, 运行指示灯的切换频率表示 CPU 占用率, 1 次/秒表示 CPU 占用率为 0%, 10 次每秒表示 100%
运行指示灯灭, 警报指示灯长亮	不可恢复异常
警报指示灯闪烁 1 次	运动控制内核错误
警报指示灯闪烁 2 次	报警被触发

9 封装

芯片采用 LQFP（Low-profile Quad Flat Package）薄型四方扁平式封装，本体尺寸为 $20 \times 20 \times 1.4\text{mm}$ ，管脚间距为 0.5mm ，其他参数详见下图。

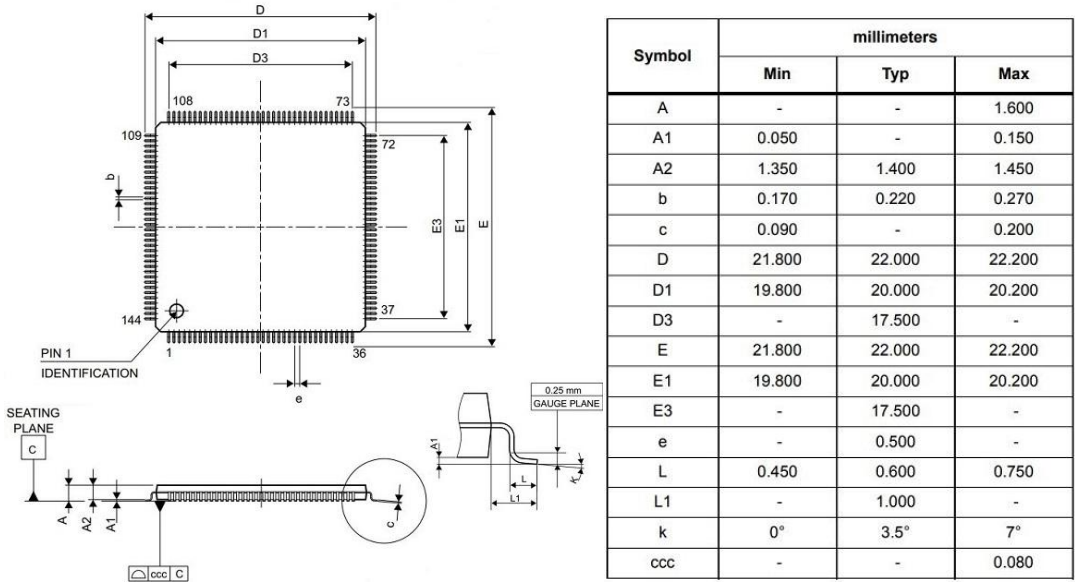


图 9-1 封装尺寸图

通用 MCU 都有 UART 接口，通过 UART 接口发送运动控制指令，即可在一块 PCB 板内实现相关功能。下面以中控微电子的通用 MCU CMC520MD32 为例进行电路说明：

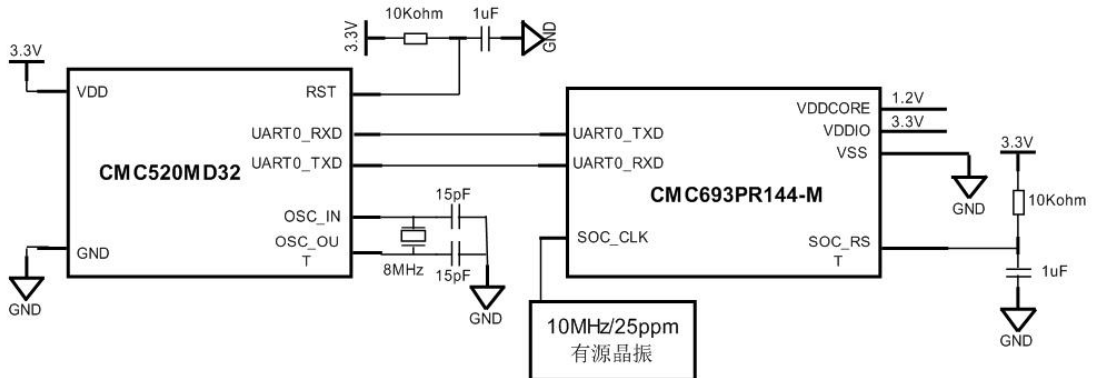


图 10-3 与通用 MCU 通讯

10.4 脉冲控制电路

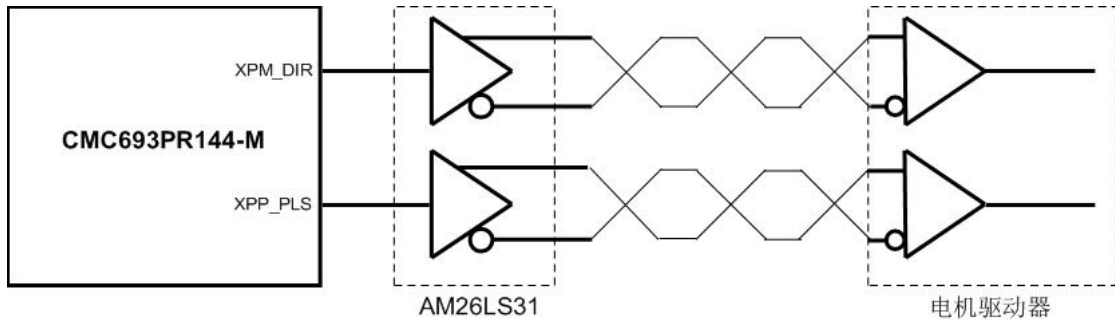


图 10-4 脉冲控制电路

这里仅呈现 X 轴的接线方式，其他轴均可采用相同的接线方式。

10.5 编码器电路

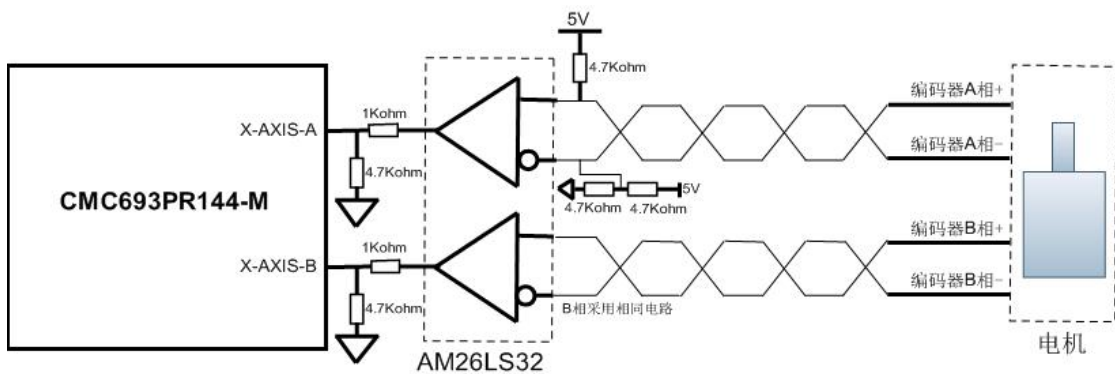


图 10-5 编码器电路

这里仅呈现 X 轴的接线方式，其他轴的编码器电路均可采用相同的接线方式。

10.6 手轮电路

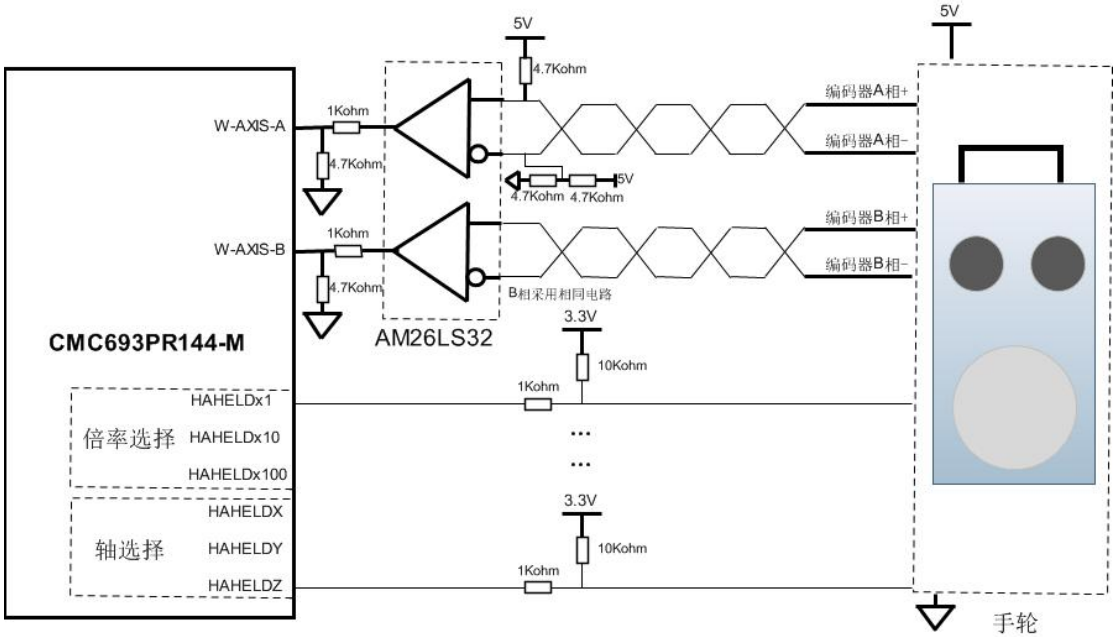


图 10-6 手轮电路示意图

10.7 RS232 通讯电路

支持将串口扩展成 RS232 方式，与外接主控设备进行通讯。

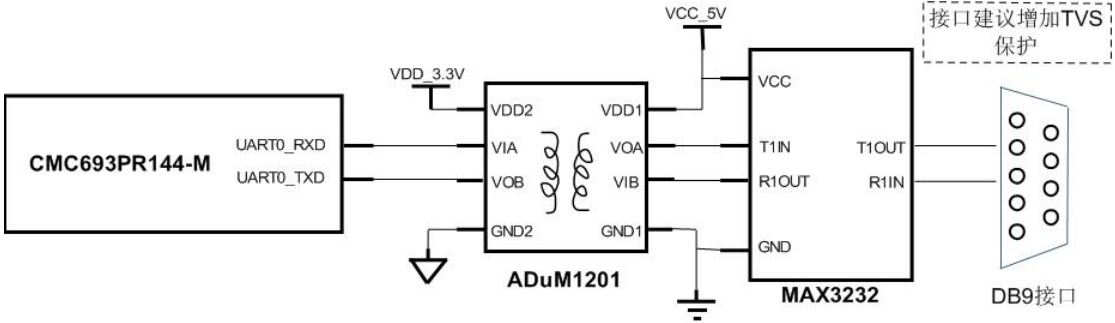


图 10-7 RS232 通讯电路

10.8 输入输出电路

输出 DO 电路：

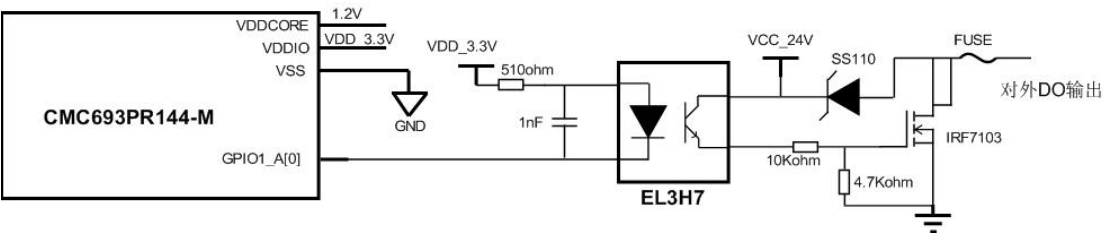


图 10-8 DO 电路

输入 DI 电路：

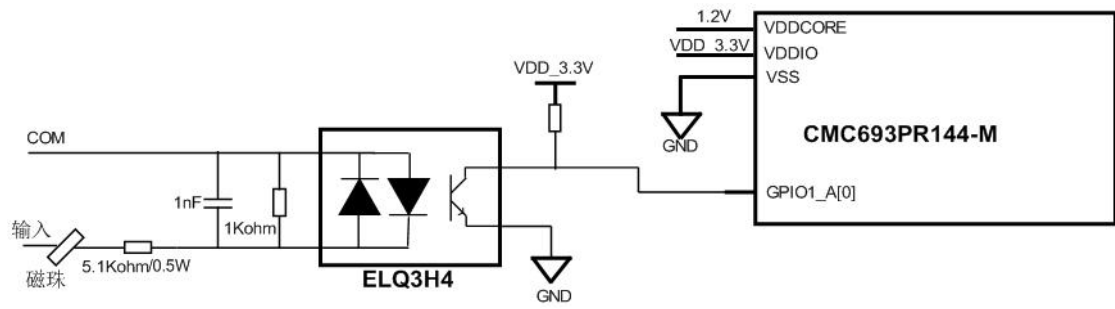
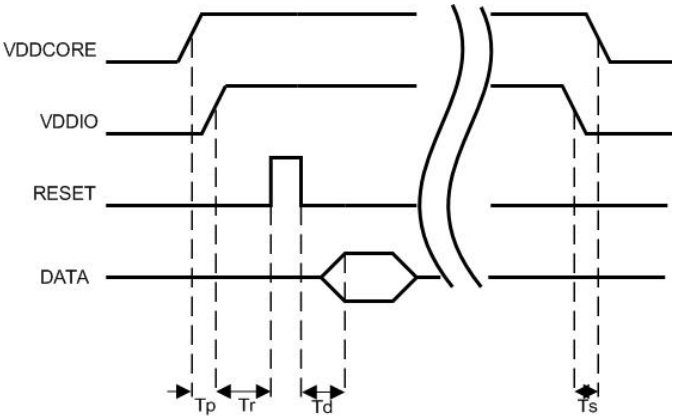


图 10-9 DI 电路

11 时序要求



名称	说明	最小值	最大值	单位
Tp	内核电压与 I/O 电压建立先后时间	0	10	ns
Tr	复位信号与 I/O 电压建立间隔时间	20	60	ms
Td	复位后与数据开始间隔时间	3		s
Ts	I/O 电压与内核电压结束间隔时间	0	10	ns

12 版本信息

资料版本号	起草人	发布日期	更改说明
V1.0	魏彬	2020-01-05	第一版本编写

技术咨询

宁波中控微电子有限公司

浙江省宁波市海曙区丽园北路 1350 号众创空间 2 号楼 501 室

ZIP:315000

TEL:0086-0574-87288895

Email: support@nz-ic.com