GIM使用说明书

驱动使用说明



阅读指南

😊欢迎使用 GIM 关节模组资料库！为帮助您快速获取所需信息，请参考一下阅读指引：

1.产品选型手册

GIM 产品参数详情，请查看 GIM 产品手册 Lite.pdf

2.电机模型文件

请联系我们

3.电机使用

3.1 电机测试

感谢购买 GIM 电机，请参考“GIM GUI 软件使用手册”连接并测试电机

3.2 二次开发

- **内存表定义**

将电机控制集成到自有项目前，请先了解电机内部的内存表定义 1.1 [QD&DD 系列] 内存表&Modbus 地址说明.xlsx，控制电机本质上是对电机内存表的寄存器值进行修改。可通过 485 或 CAN 接口发送数据实现对内存表寄存器值修改。

- **通信协议**

GIM 电机有 485 和 CAN 两种通信接口，请依据具体电机型号，遵循 485 通信协议格式“485 通信控制协议 V2.0”、CAN 通信协议“CAN 通信控制协议 V2.0”格式定义发送和解析交互数据。

- **SDK**

我们提供 485 SDK 接口“485/TTL 通信 SDK”，助力您快速实现对电机的控制。目前 CAN 通信，有“MIT 协议测试 Demo”。

目 录

GIM GUI 软件使用手册 1

485 通信控制协议 V2.0 18

CAN 通信控制协议 V2.0 35

MIT 协议测试 Demo 51

485/TTL 通信 SDK 64

GIM GUI 软件使用手册

1. 软件概述

本软件为 GIM 系列电机调试与参数设置软件，竭力为用户提供可视化友好的操作界面。该软件主要实现对电机的在线搜索、电机控制、参数设置、实时数据监测与曲线绘制、固件升级等功能。软件支持 RS485 和 CAN 两种通信方式，同时支持多个电机在线控制。

2. 设备接线

注意：设备供电前，请确认供电电压范围。

GIM 系列电机提供 RS485 和 CAN 两种通信方式，可通过拨码开关设置 RS485 或 CAN 通信。



图 1 电机通信口设置

电机需要使用 USB 转 RS485 模块或 USB 转 CAN 模块连接到电脑电脑。推荐使用以下 2 种模块。



图 2 USB 转 RS485 模块



图 3 USB 转 CAN 模块

485 与 CAN 通信功能基本一致，下面以 485 通信为例介绍软件功能。485 模块与多个电机接线如下图所示。



图 4 电机接线图

USB 转 485 模块接入电脑后，在设备管理器查看对应 COM 口。

3. 软件启动及界面简介

本软件免安装，双击 GIM_GUI_xxx.exe 文件启动控制软件。软件通过 USB 线与 GIM 电机进行连接。软件启动后界面如下图所示。

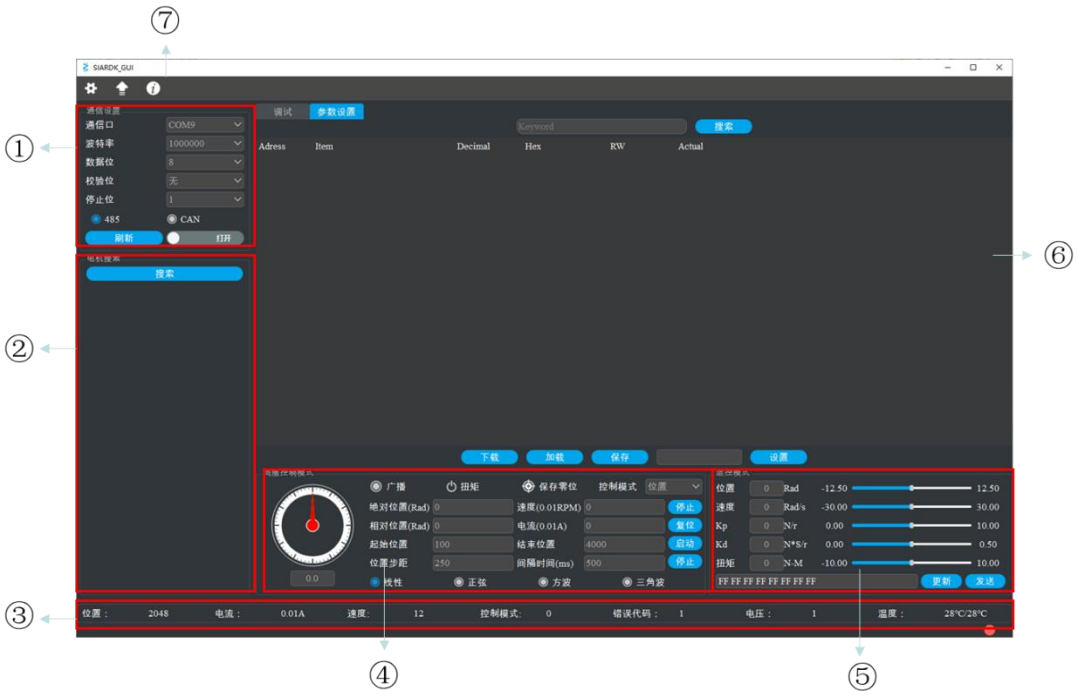


图 5 主界面图

①通信设置区域

②在线设备列表区域

③实时数据显示区域

④伺服控制模式区域

⑤运动模式（MIT）区域

⑥寄存器实时数据表格与实时曲线绘制区域

⑦菜单栏区域

4. 电机调试步骤

4.1 连接设备

首先，将电机通信接口拨码开关拨到 485(AB)侧，使用 USB 转 485 模块连接电机的通信及电源口，485 模块同时插入电脑 USB 口。电脑打开 GIM_GUI 软件，选择正确的 COM 口，选择 485 模式打开串口通信。



图 6 通信设置

成功打开串口通信如下



图 7 成功打开串口通信

给电机供电，点击“搜索”按钮开始搜索所有在线设备并显示在设备列表区域。“搜索”按钮点击后，软件尝试连接 ID 从 1~252 的电机，如在设备列表区域已显示出已连接设备，可点击“停止”按钮结束在线搜索进程。设备列表树一级目录为设备型号，二级目录为设备 ID。同一型号电机在相同一级目录下。

注意：设备 ID 号必须唯一，否则影响设备通信。



图 8 设备搜索结果

鼠标左键点击设备列表中的设备 ID 行，选择当前控制电机，此时软件的右侧参数设置区域及底部的状态监测区将显示选中电机的实时数据，右下角状态栏通信指示灯变成绿色表示通信正常。可再次点击其它设备 ID 行来切换当前受控电机。

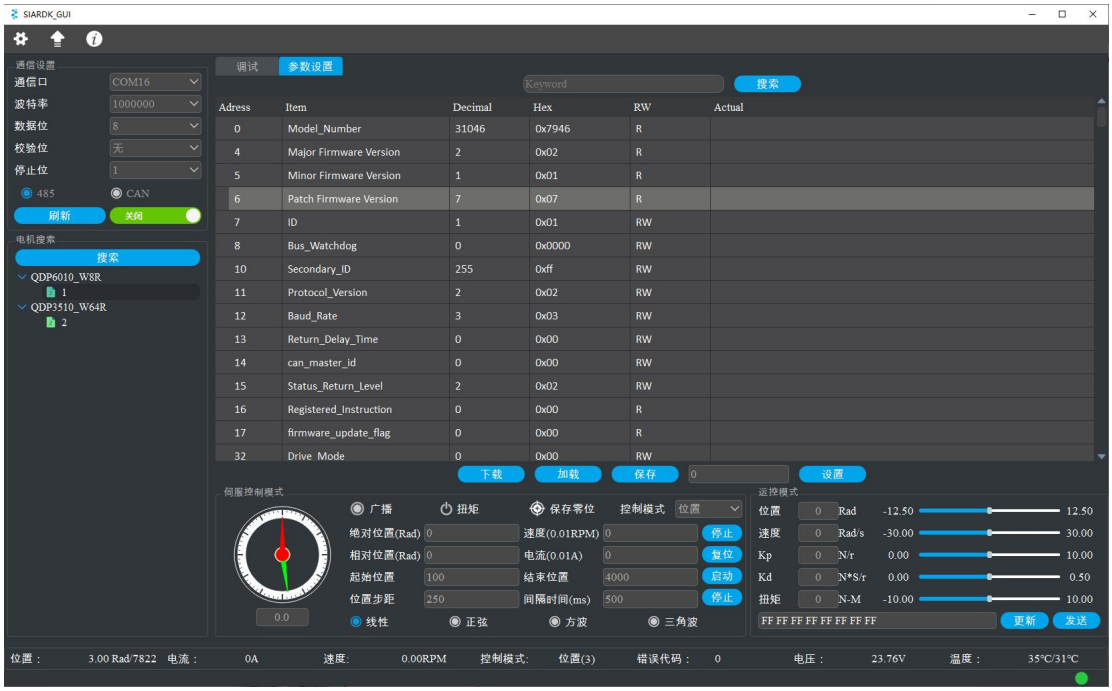


图 9 选择控制设备

4.2 校准电机

注意：电机出货前已完成校准，此步骤可跳过。如需进行电机校准，请按以下步

骤操作。

校准工作主要是修正传感器的安装误差，过程中电机会正向反向转动，请保证电机可以自由转动，最好电机能够空载运行，否则可能造成校准失败。

GIM 电机内部通过寄存器来保存电机运行参数，同时可以通过修改寄存器值来控制电机工作。电机校准寄存器地址为 403。

- 在“参数设置”区域的“keyword”输入框中输入 403，回车。参数表格仅筛选出电机校准寄存器 403 行，寄存器当前值为 48。403 寄存器值含义在后面介绍；
- 鼠标单击选中 403 寄存器行，选中行背景色变成蓝色；
- 在参数设置输入框写入新值，这里写 51（启动内外部编码器校准）。

4. 点击“设置”按钮启动校准。

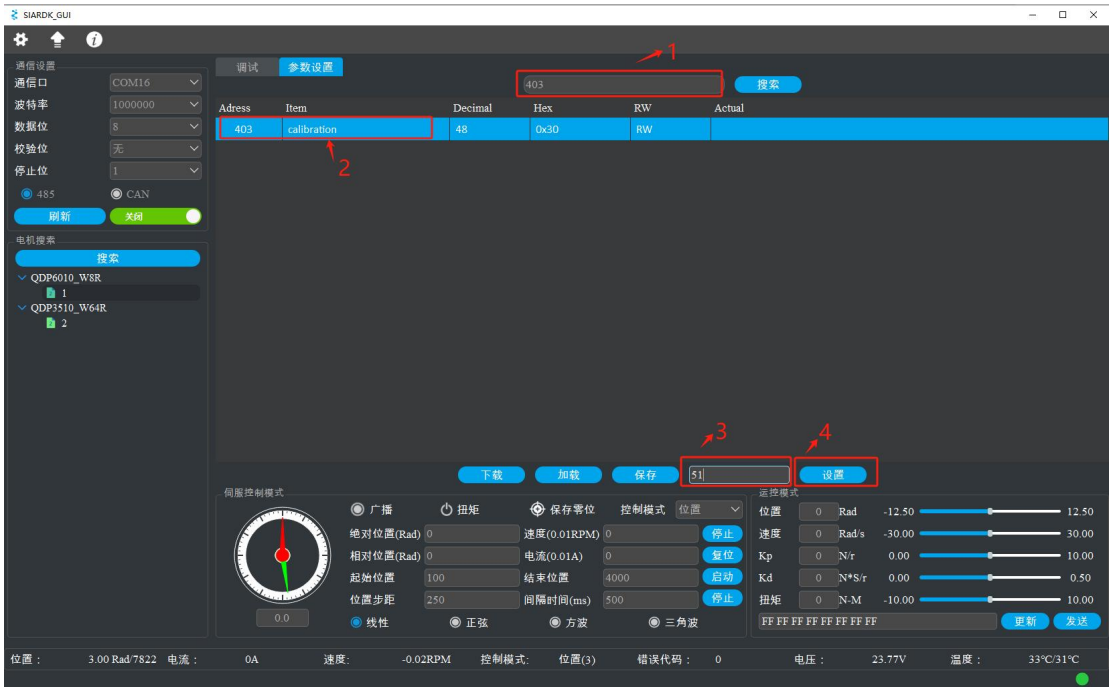


图 10 设置电机校准寄存器

此时可以看到电机开始缓慢转动，表示编码器校准已开始。请耐心等待。当电机停止转动，且 403 寄存器值变为 60，表示内外部编码校准成功。

如果电机有双编码器，可实现断点位置记忆功能。**可断点记忆的位置范围为（-PI，PI）**。

校准寄存器 403，1 个字节。目前已使用 6 个 bit 位。各 bit 位含义如下：

表 1 校准寄存器（403）含义

Bit 7	Bit 6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

保留	保留	外部编码器是否连接（1：连接；0：未连接）	内部编码器是否连接（1：连接；0：未连接）	外部编码器是否校准（1：已校准；0：未校准）	内部编码器是否校准（1：已校准；0：未校准）	是否启动外部编码器校准（1：启动校准；0：不校准）	是否启动内部编码器校准（1：启动校准；0：不校准）

电机上电后根据获取编码器数据情况，自动设定 Bit4 和 Bit5 内部编码器连接情况；Bit3 和 Bit2 由校准程序结果设定；Bit1 和 Bit0 由用户设定是否启动校准，校准结束后恢复为 0。

常见寄存器 403 值含义如下。其它值，请按以上定义分析。

表 2 常见校准寄存器 403 值

48（00110000）	内外部编码器已连接，但未校准
60（00111100）	内外部编码器已连接，且已校准
54（00110110）	内外部编码器已连接，内部编码器已校准，外部编码器未校准，启动外部编码器校准
52（00110100）	内外部编码器已连接，内部编码器已校准，外部编码器未校准

4.3 伺服模式调试



图 11 伺服控制调试 1

以上区域为伺服控制模式区域。

1. “广播”按钮：选中，表示后续操作对所有在线电机有效；否则，仅对当前选择电机有效；

2. “扭矩”按钮：选中，打开电机扭矩开关；取消，关闭电机扭矩开关；

3. “保存零位”按钮：将当前位置设置为零位。该功能需在扭矩开关关闭下生效。成功设置后，可观察到电机实时位置显示为 0；

4. 控制模式：目前支持位置、速度、电流、MIT 四种控制模式。可通过下拉框切换模式。



图 12 伺服控制调试 2

1. 绝对位置：输入目标位置值，回车。电机运行到设定位置；

2. 相对位置：输入相对位置值，回车。电机将在当前位置基础上运行输入的相对位置；

3. 速度：输入目标速度，回车。电机将以设定目标速度运动；

4. “停止”按钮：设置目标速度为 0，电机停止运动；

5. 电流：输入目标电流，回车。电机将以设定电流对应的扭力驱动电机运动；

6. “复位”按钮：设置目标电流为 0，电机停止运动；



图 13 伺服控制调试 3

在“位置”控制模式下，打开扭矩开关，还可以通过鼠标拖动仪表盘指针控制电机运动。仪表盘可控制电机在（0, 2*PI）范围运动，零位在最上方，顺时针递增。

红色指针为目标位置，绿色指针为当前位置。输入框中的数值和仪表盘红色指针同步，表示电机的目标位置，所以也可手动在输入框输入目标位置值控制电机运动到指定位置。



图 14 伺服控制调试 4



图 15 伺服控制调试 5

该区域为模拟曲线控制区。可按照设置的曲线类型及参数连续的发送目标位置命令。目前支持线性、正弦、方波、三角波 4 种模拟曲线。“线性”曲线需设置 4 个参数，其它三种曲线需设置 2 个参数。

1. “线性”

在起始位置与结束位置区间，电机目标位置 = 起始位置+位置步距。软件以设定的时间间隔发送目标位置命令。

2. “正弦”

幅值：正弦波幅值；周期：正弦波周期。软件以 50HZ 固定采样频率计算正弦波取值，并将其设定为位置目标值发送。目标位置发送时间间隔为：1/周期。

一个周期内总的采样数 sample = 周期*0.05;

当前模拟采样索引 sampleIndex;

位置目标值 = 幅值*sin（2 * PI * sampleIndex / sample）;

3. “方波”

幅值：方波幅值；周期：方波波周期。软件以设定幅值为目标值并发送，一个周期内发送两次位置目标值（正幅值、负幅值各一次）。发送时间间隔为：周期/2。

4. “三角波”：

幅值：方波幅值；周期：方波波周期。

4.4 MIT 运控模式调试



图 16 运控模式调试

运控模式即 MIT 模式，需设置位置、目标速度、Kp、Kd、扭矩 5 个目标值。当连接电机后，MIT 的 5 个参数最大、最小值可从电机寄存器获得并更新到界面上。

MIT 相关参数对应的寄存器如下所示，单位 0.01。可通过修改这些寄存器的值来修改范围。

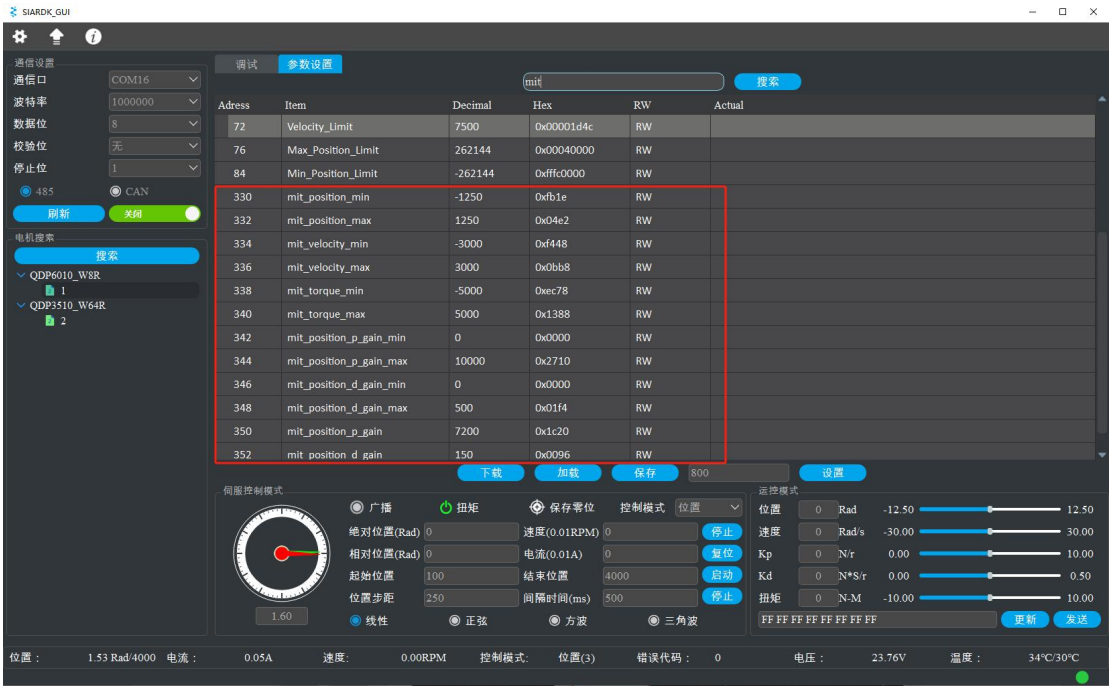


图 17 MIT 控制相关寄存器

首先需要将电机设置为 MIT 模式，再打开扭矩开关。

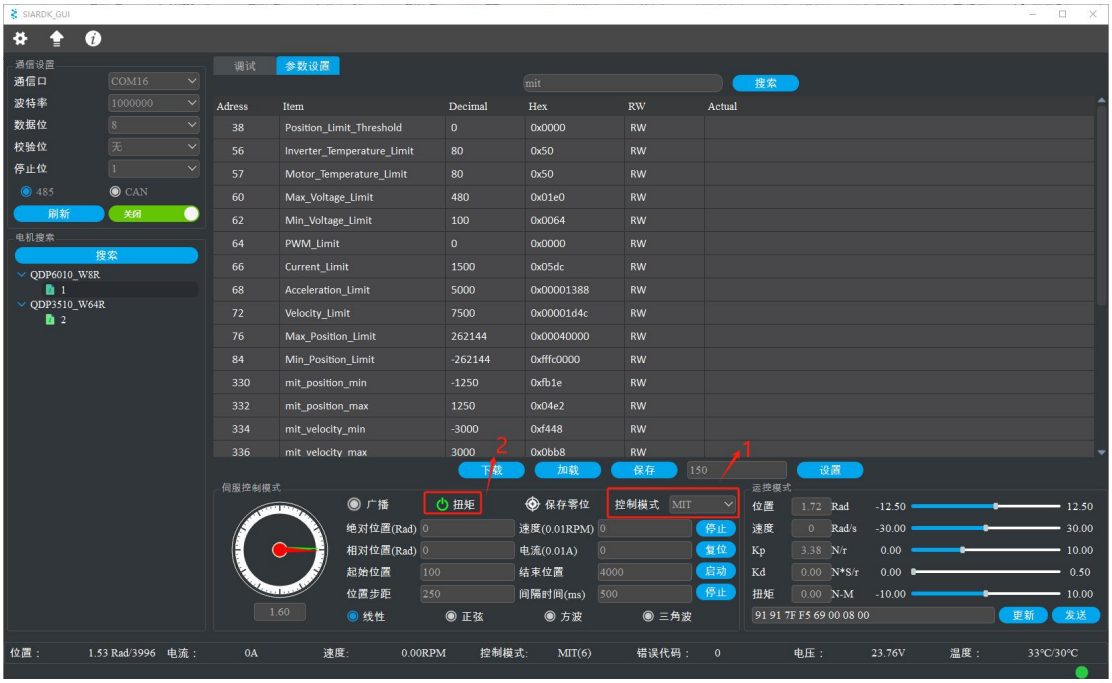


图 18 设置 MIT 模式并打开扭矩开关

拖动滑动条或在输入框里修改 MIT 目标参数值。设定好目标值后，点击“发送”按钮，更新十六进制显示框内容并发送 MIT 命令；点击“更新”按钮仅更新十六进制显示框内容，不发送命令。

下面详细介绍 MIT 模式下的三种控制方法：速度、位置、力矩。为什么 MIT 模式能实现三种控制方法。可从下图 MIT 模式控制框图分析。

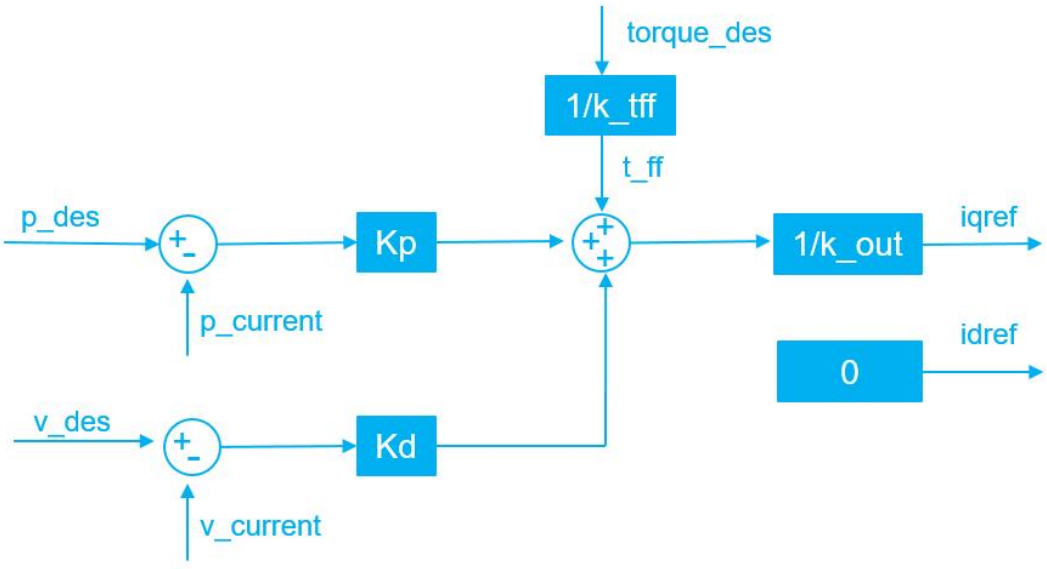


图 19 MIT 控制框图

从 MIT 运行框图可知电机的 q 轴电流 iqref 计算公式如下：

iqref=((p_des-p_current)*Kp+(v_des-v_current)*Kd+(torque_des)*(1/k_tff))*(1/k_out);

其中 k_tff 为扭矩系数，k_out 为电流转换系数，通常为电机常数，不需要修改。

1. 位置模式

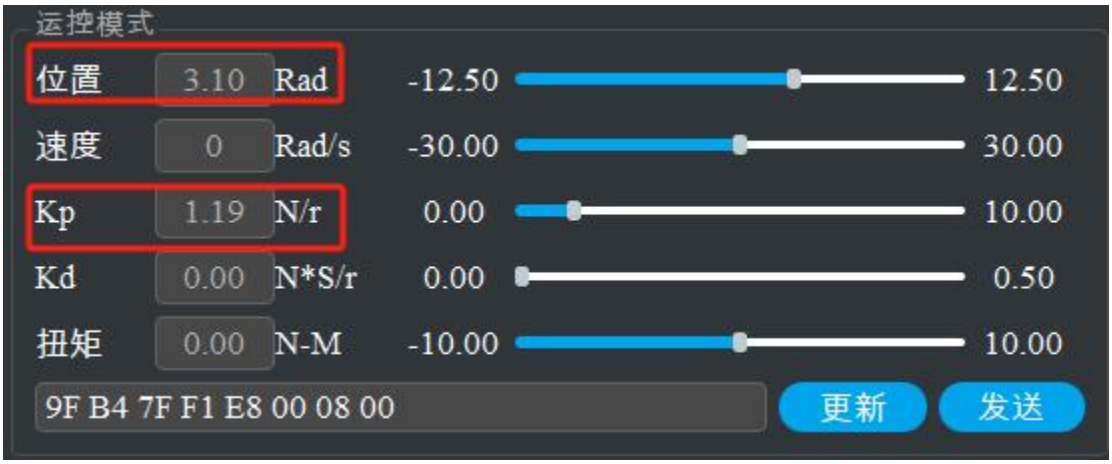


图 20 MIT-位置模式

当给定目标位置及 KP，其它值为 0 时。Iqref 可简化为以下公式。这是一个只有比例环的位置控制，所以可能会有一定的稳态误差。

iqref = (p_des-p_current)*Kp ;

注意：此时应注意电机初始位置，避免目标位置与初始位置差距过大，引起电机冲击。可通过点击“保存零点”将电机当前位置设置为零点，方便“位 置”参数设置。

2. 速度模式



图 21 MIT-速度模式

当给定目标速度及 Kd，其它值为 0 时。Iqref 可简化为以下公式。这是一个只有比例环的速度控制，所以可能会有一定的稳态误差。

iqref =(v_des-v_current)*Kd ;

3. 力矩模式

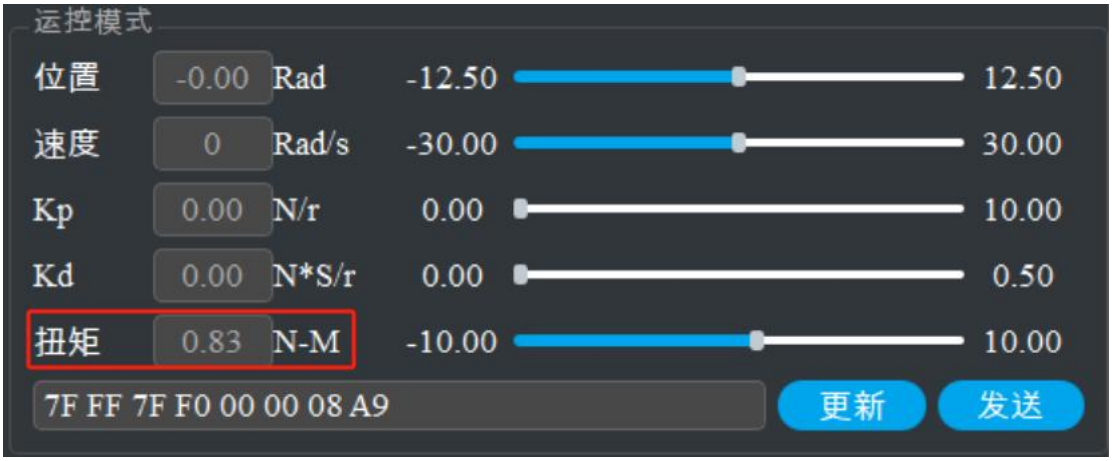


图 22 MIT-力矩模式

当给定目标扭力，其它值为 0 时。

iqref = (torque_des)*(1/k_tff);

注意：空载情况下，即使给定很小的力矩，电机也会加速到最大转速旋转。

5. 其它功能

5.1 设置零

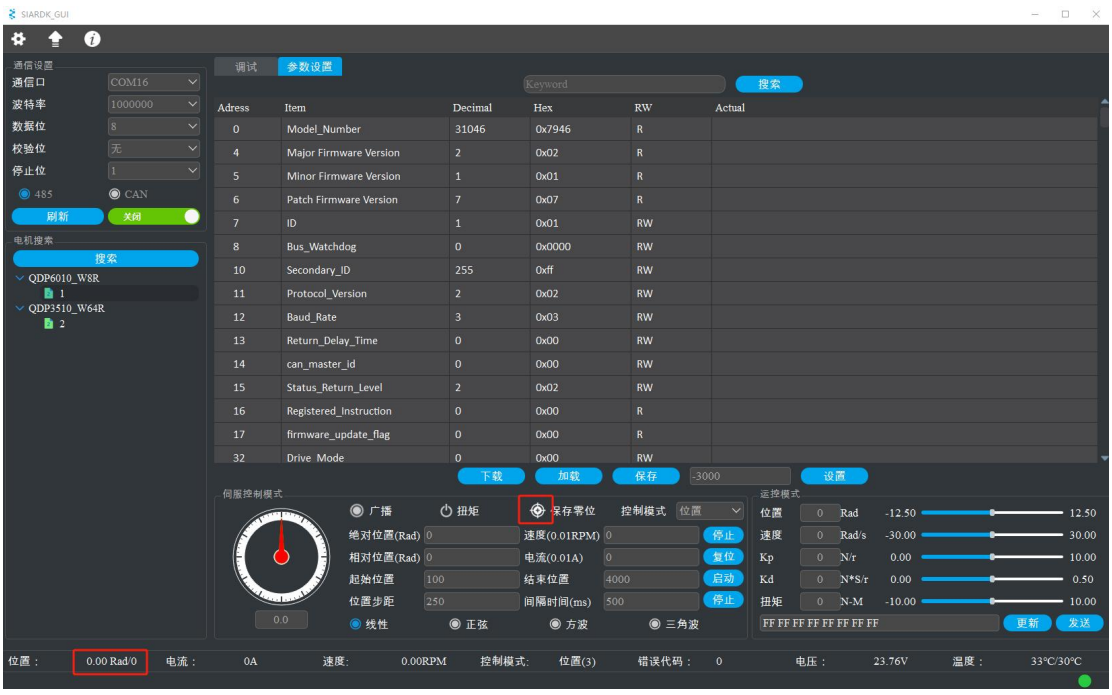


图 23 保存零位

点击“保存零位”按钮，可将当前位置设置为零位。成功设置后，可观察到底部状态栏的位置信息变为 0。

5.2 查看实时曲线

点击“调试”标签页，查看实时曲线。目前仅支持查看实时位置、速度、电流曲线图。点击右上角开关打开需要查看的数据，再点击▶ 按钮，开始曲线绘制。曲线纵轴显示范围可通过右侧的输入框写入最大值，如需要查看正负值，可点击“负值”开关。

曲线绘制区支持鼠标滚轮进行缩放查看；鼠标框选进行区域放大。右键菜单“Reset plot”可恢复曲线原始显示比例。建议点击按钮暂停曲线播放时，再进行框选放大查看，否则，曲线将不断刷新，无法查看数据。



图 24 查看实时曲线

5.3 修改寄存器值

1. 在“keyword”输入搜索关键字，回车，软件将匹配表格的 Adress 和 Item 列内容，过滤出匹配关键字的表格行数据；
2. 选择需要修改的寄存器对应行。选中行背景为蓝色；
3. 在寄存器值输入框内设置新值；
4. 点击“设置”按钮。发送设置命令。

用户在使用电机过程中，可根据具体使用场景，调整电机参数优化电机表现。电机寄存器定义详情请参考《GIM 内存表解析》文档。电机三环控制 PID 参数对电机运行效果有很大影响，建议有一定运动控制原理基础再修改这些参数，同时结合实时曲线情况调整参数。

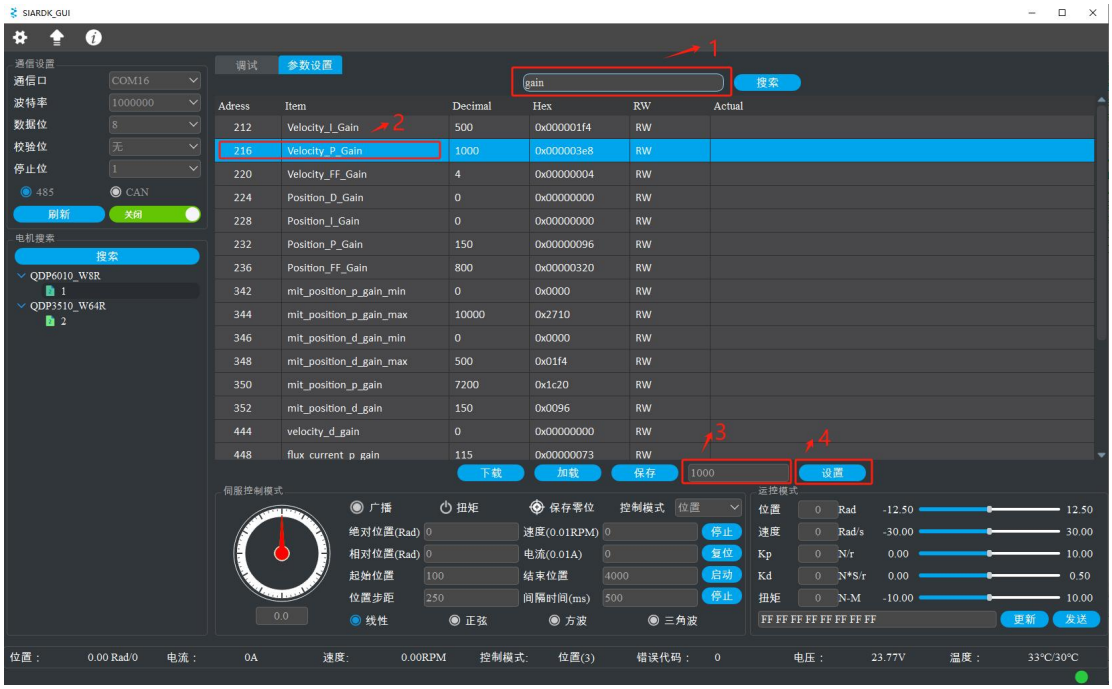


图 25 修改寄存器

5.4 寄存器参数保存、加载及下载

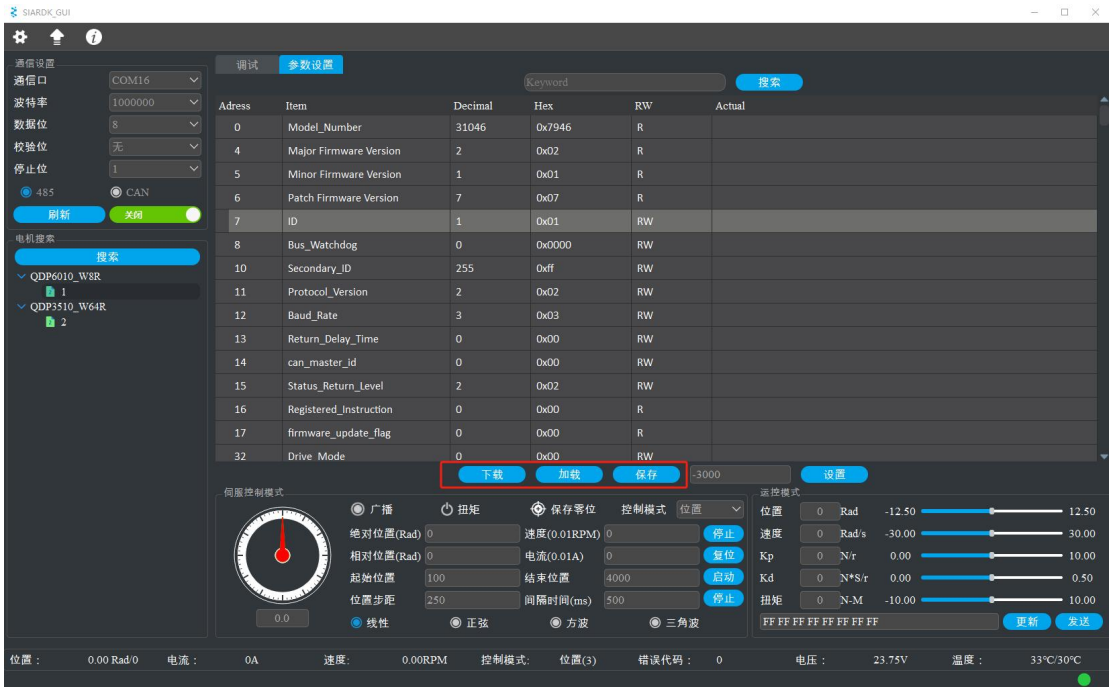


图 26 寄存器参数保存、加载及下载

1. “保存”按钮：可将整个参数表格的内容保存为本地文件，文件格式为.vdat;
2. “加载”按钮：可将选中的本地参数文件（.vdat），加载到参数表格区域;

3. “下载”按钮：可将整个参数表格的内容下载到电机中。该操作仅在 485 通信模式下支持，且有一定风险，请谨慎操作。

5.5 固件升级

点击固件升级菜单，弹出固件升级窗口。选择固件 bin 文件，最后点击“升级”按钮，启动固件升级。

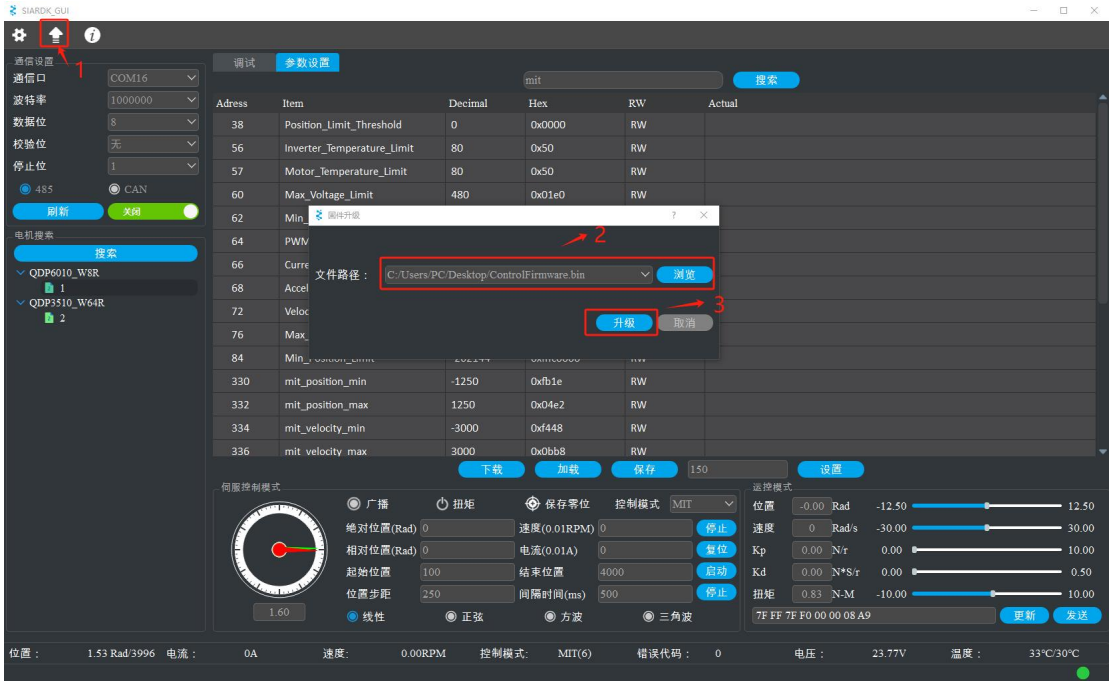


图 27 启动固件升级

固件升级成功提示如下图所示。

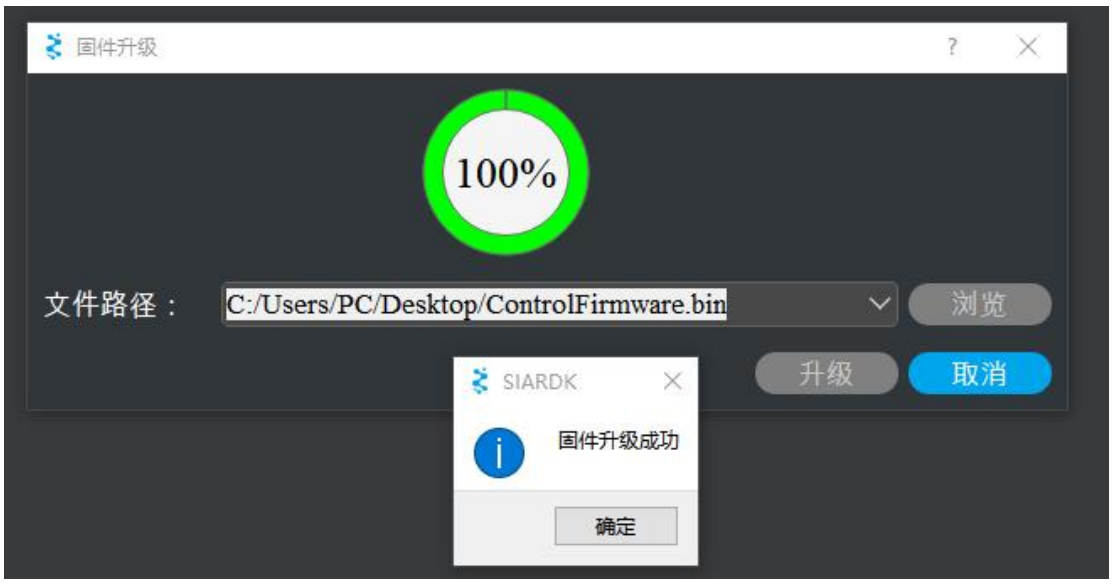


图 28 固件升级成功

5.6 位置单位及语言设置

点击设置菜单，弹窗设置窗口，设置位置单位及软件语言，最后点击“确定”按钮。

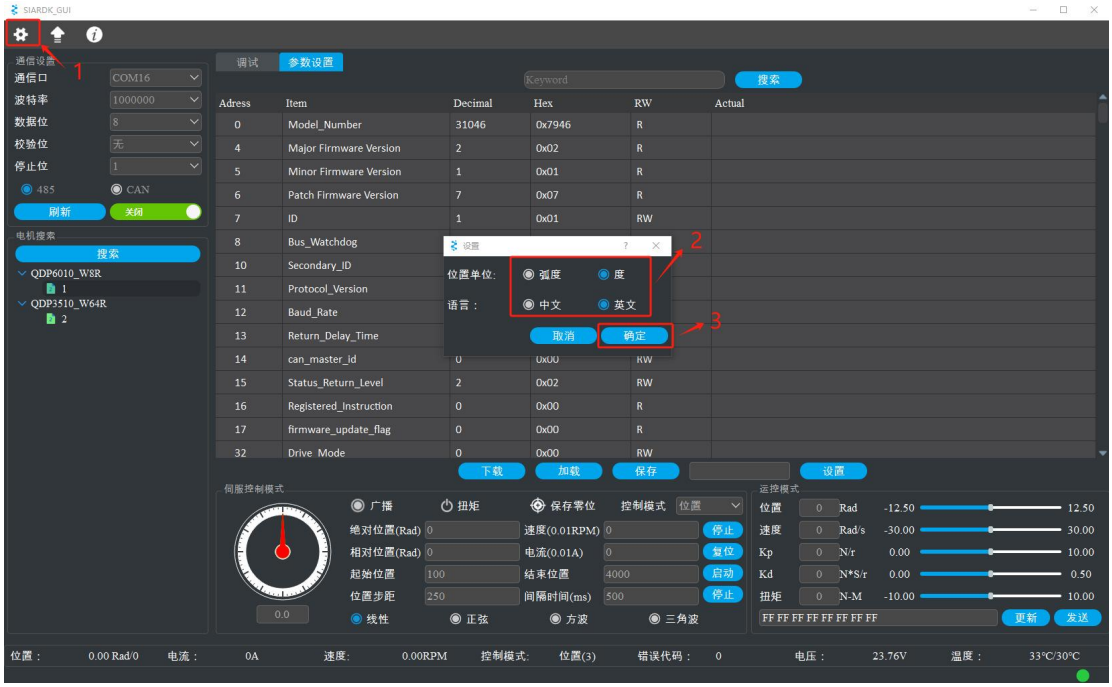


图 29 单位及语言设置

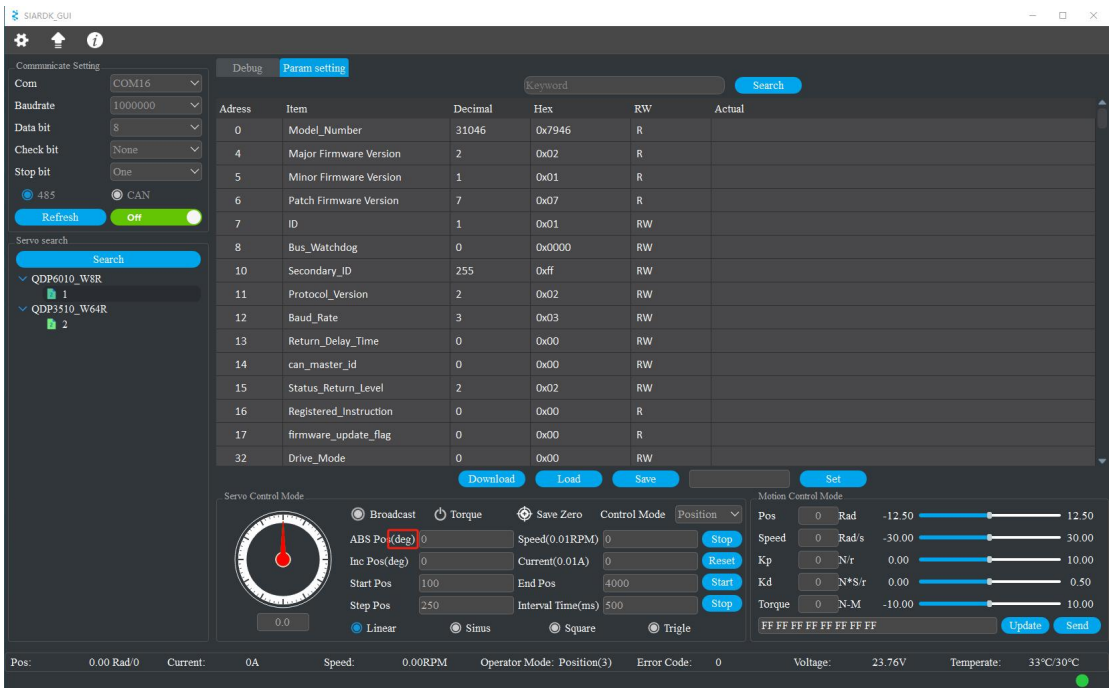


图 30 切换单位及语言结果

485 通信控制协议 V2.0

通信协议中需要指定寄存器起始地址，有关寄存器地址定义，请参考内存表与 modbus 地址定义说明

---

1 简介

本文档定义了 GIM 系列伺服电机 485 通信控制协议格式，协议版本为 V2.0。请严格按本协议格式规范与 GIM 伺服电机进行交互。

2 数据帧格式

GIM 系列伺服电机通信为异步方式，并以字节为单位。通讯数据帧为不确定的串行数据流。数据帧长度合计：（N+10）Byte。

2.1 基本格式

2.1.1 指令包

消息头 H1-H3 (3 个字节)			保留 (RSRV)	数据包 ID	长度 (低字节)	长度 (高字节)	指令	参数	参数	参数	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	ID	Len_L	Len_H	指令	参数 1	…	参数 N	CRC_L	CRC_H

各字节的含义如下：

- **消息头**：3 字节，固定值为 0xFF，0xFF，0xFD；
- **保留**：使用 0X00 （请注意，Reserved 不使用 0XFD）；
- **数据包 ID**：该字段指示应接收指令包并对其进行处理的设备的 ID。

5. ID 范围： 0 ~ 252（0x00 ~ 0xFC），总共有 253 个数字可供使用

2. 广播 ID：254（0xFE），所有连接的设备执行指令数据包

3. **数据长度**：表示数据包长度的字段，在指令包中分为低字节和高字节。数据包长度为 2 字节，等于待发送的参数 N 加上 3，即“N+3”；

4. **指令**：数据包操作功能码，详见 2.2 指令类型；

5. **参数**：作为指令的辅助数据字段，每个 指令的用途都不同；

2. **CRC**：16 位 CRC 字段，用于检查数据包在通信过程中是否损坏。在指令包中分为低字节和高字节。CRC 计算范围：从消息头（FF FF FD）到指令包中 CRC 字段之前的所有参数。

2.1.2 应答包

消息头 H1-H3（3 个字节）			保留（RSRV）	数据包 ID	长度（低字节）	长度（高字节）	指令	ERR	参数	参数	参数	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	ID	Len_L	Len_H	指令	错误	参数 1	…	参数 N	CRC_L	CRC_H

各字节的含义如下：

3. **消息头**：3 字节，固定值为 0xFF，0xFF，0xFD；

4. **保留**：使用 0X00（请注意，Reserved 不使用 0XFD）；

5. **数据包 ID**：该字段指示返回应答包的设备 ID。

6. ID 范围： 0 ~ 252（0x00 ~ 0xFC），总共有 253 个数字可供使用

7. 广播 ID：254（0xFE），即所有连接的设备接收指令数据包；

2. **数据长度**：表示数据包长度的字段字段，在指令包中分为低字节和高字节。数据长度为 2 字节，等于待发送的参数 N 加上 3，即“N+3”；

3. **指令**：数据包操作功能码，详见以下指令类型；

4. **参数**：作为 Instruction 的辅助数据字段，每个 Instruction 的用途都不同；

5. **CRC**：16 位 CRC 字段，用于检查数据包在通信过程中是否损坏。在指令包中分为低字节和高字节。CRC 计算范围：从消息头 （FF FF FD）到指令包中 CRC 字段之前的参数。

2.2 指令类型

不同的指令代表不同的操作，GIM 系列电机支持的指令如下表格所示：

指令	功能	值	参数长度
PING（查询）	查询设备状态	0x01	0
READ（读）	从设备控制表中读取数据	0x02	4
WRITE（写）	往设备控制表里写入数据	0x03	大于等于 2
FACTORY RESET（恢复出厂设置）	将设备控制表重置为其初始出厂默认设置	0x06	大于等于 3
REBOOT（重新启动）	重启设备	0x08	0
SYNC READ（同步读取）	对于多个设备，一次从相同地址读取相同长度的数据	0x82	不小于 4
SYNC WRITE（同步写入）	对于多个设备，一次从相同地址写入相同长度的数据	0x83	大于 4 或大于 5
FAST SYNC READ（快速同步读取）	与 SYNC READ 指令功能类似，但可实现更快的通信	0x8A	大于 4
BULK READ（批量读取）	对于多个设备，一次从不同地址读取不同长度的数据	0x92	大于 5
BULK WRITE（批量写入）	对于多个设备，一次从不同地址写入不同长度的数据	0x93	不小于 5

FAST BULK READ（快速批量读取）	与 Bulk Read 指令功能类似，但可实现更快的通信	0x9A	不小于 5

2.2.1 查询状态 PING

功能 查询设备状态

长度 0X03

指令 0X01

参数 无

当 PING 指令使用广播地址 0XFE 时，所有在线设备返回应答信息。

例： 查询 ID1 设备状态

指令帧：FF FF FD 00 01 03 00 01 19 4E(按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	0x01	0x03	0x00	0x01	0x19	0x4E

返回的数据帧： FF FF FD 00 01 07 00 55 00 06 04 26 65 5D（十六进制显示）

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	P1	P2	P3	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	0x01	0x07	0x00	0x55	0x00	0x06	0x04	0x26	0x65	0x5D

2.2.2 读指令 READ

功能 从设备控制表中读取数据

长度 0X07

指令 0X02

参数 1 起始地址的低序字节

参数 2 起始地址的高序字节

参数 3 数据长度 （X） 的低序字节

参数 4 数据长度 （X） 中的高阶字节

例：读取 ID1 电机当前位置(132, 0x0084, 4[byte])，当前位置值为 166(0x000000A6)

指令帧：FF FF FD 00 01 07 00 02 84 00 04 00 1D 15(按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	P1	P2	P3	P4	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	0x01	0x07	0x00	0x02	0x84	0x00	0x04	0x00	0x1D	0x15

返回的数据帧：FF FF FD 00 01 08 00 55 00 A6 00 00 00 8C C0 （十六进制显示）

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	P1	P2	P3	P4	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	0x01	0x08	0x00	0x55	0x00	0xA6	0x00	0x00	0x00	0x8C	0xC0

2.2.3 写指令 WRITE

功能 往设备控制表里写入数据

长度 0X09

指令 0X03

参数 1 起始地址的低序字节

参数 2 起始地址的高序字节

参数 3 第一个字节

参数 N+1 第 N 个数据

例：设置 ID1 电机目标位置（116, 0x0074, 4[byte]）为 512（0x00000200）

指令帧： FF FF FD 00 01 09 00 03 74 00 00 02 00 00 CA 89 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	P1	P2	P3	P4	P5	P6	CRC 1	CRC 2
0xFF	0xFF	0xF	0x00	0x0	0x09	0x00	0x03	0x74	0x00	0x00	0x02	0x00	0x00	0xCA	0x89

		D		1										

返回的数据帧：FF FF FD 00 01 04 00 55 00 A1 0C（十六进制显示）

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0x01	0x04	0x00	0x55	0x00	0xA1	0x0C

2.2.4 恢复出厂设置 RESET

功能　　将 Control Table 重置为其初始出厂默认设置

长度　0X04

指令　0X06

参数 1

0xFF：　全部重置

0x01：　重置除 ID 之外的所有内容

0x02：重置除 ID 和波特率之外的所有内容

例：ID1 使用参数 0x01 重置除 ID 之外的所有参数

指令帧：FF FF FD 00 01 04 00 06 01 A1 E6 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	P1	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0x01	0x04	0x00	0x06	0x01	0xA1	0xE6

返回的数据帧：FF FF FD 00 01 04 00 55 00 A1 0C（十六进制显示）

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0x01	0x04	0x00	0x55	0x00	0xA1	0x0C

2.2.5 重启设备 REBOOT

功能　　重启设备

长度 0X03

指令 0X08

参数 无

例：ID1 重启设备

指令帧：FF FF FD 00 01 03 00 08 2F 4E (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0x01	0x03	0x00	0x08	0x2F	0x4E

返回的数据帧：FF FF FD 00 01 04 00 55 00 A1 0C （十六进制显示）

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0x01	0x04	0x00	0x55	0x00	0xA1	0x0C

2.2.6 同步读取 SYNC READ

功能 对于多个设备，一次从相同地址读取相同长度的数据

ID 0XFE

长度 0X09

指令 0X82

参数 1 起始地址的低序字节

参数 2 起始地址的高序字节

参数 3 数据长度 （X） 的低序字节

参数 4 数据长度 （X） 的高位字节

…

参数 N+2 第 N 个舵机的 ID 号

…

一条 SYNC READ 指令可一次查询多个电机的控制表内容，同步读指令中指定了需要查询电机的 ID，电机返回应答包顺序按指令包中 ID 顺序返回，使用 SYNC READ 指令时，所有查询的数据长度和数据的首地址必须相同。

例：同时读取 ID1-ID2 的当前位置（132, 0x0084, 4[byte]）

ID1 当前位置为 166(0x000000A6)

ID2 当前位置为 2079(0x0000081F)

同步读指令包：FF FF FD 00 FE 09 00 82 84 00 04 00 01 02 CE FA(按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	P1	P2	P3	P4	P5	P6	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	0xFE	0x09	0x00	0x82	0x84	0x00	0x04	0x00	0x01	0x02	0xCE	0xFA

返回包：

FF FF FD 00 01 08 00 55 00 A6 00 00 00 8C C0

FF FF FD 00 02 08 00 55 00 1F 08 00 00 8A BE

返回包拆解：ID1 返回包：FF FF FD 00 01 08 00 55 00 A6 00 00 00 8C C0 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	P1	P2	P3	P4	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	0x01	0x08	0x00	0x55	0x00	0xA6	0x00	0x00	0x00	0x8C	0xC0

ID2 返回包：FF FF FD 00 02 08 00 55 00 1F 08 00 00 8A BE (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	P1	P2	P3	P4	CRC 1	CRC 2
0xFF	0xFF	0xFD	0x00	0x02	0x08	0x00	0x55	0x00	0x1F	0x08	0x00	0x00	0xBA	0xBE

2.2.7 同步写入 SYNC WRITE

功能 对于多个设备，一次从相同地址写入相同长度的数据

ID 0XFE

长度 0X17

指令 0X83

参数 1 起始地址的低序字节

参数 2 起始地址的高序字节

参数 3 数据长度 （X） 的低序字节

参数 4 数据长度 （X） 的高位字节

参数 5 [第 1 台设备]ID

参数 5+1 [第 1 台设备] 第 1 字节

…

参数 5+X [第 1 台设备] 第 X 字节

参数 6 [第 2 台设备]ID

参数 6+1 [第 2 台设备] 第 1 字节

…

参数 6+X [第 2 台设备] 第 X 字节

例：同时将 ID1 目标位置（116, 0x0074, 4[byte]）设置为 150（0x00000096）

ID2 目标位置设置（116, 0x0074, 4[byte]）为 170（0x000000AA）

指令帧 : FF FF FD 00 FE 11 00 83 74 00 04 00 01 96 00 00 00 02 AA 00 00 00 82 87(按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0xFE	0x11	0x00	0x83	0x74	0x00	0x04	0x00	0x01	0x96	0x00	0x00	0x00	0x02	0xAA	0x00	0x00	0x00	0x82	0x87

因为采用广播 ID 发送指令，所以不会有数据返回。

2.2.8 快速同步读取 Fast Sync Read

该指令实现比同步读取更快速的通信，无论所串联伺服电机数量多少，使用快速同步读取指令时，所有伺服电机仅会生成并返回一个状态数据包，其效果等同于单个伺服电机返回一个状态数据包。本指令的数据包格式与同步读取指令的格式完全一致。

功能 与 SYNC READ 指令功能类似，但可实现更快的通信

长度 0X10

指令 0x8A

参数 1 起始地址的低序字节

参数 2 起始地址的高序字节

参数 3 数据长度 （X） 的低序字节

参数 4 数据长度 （X） 的高位字节

参数 4+1 第 1 个设备 ID

…

参数 4+N 第 N 个设备 ID

返回数据包参数说明

参数 1 [第 1 台设备]ID

参数 2 [第 1 台设备]第 1 个返回字节数据

参数 3 [第 1 台设备]第 2 个返回字节数据

… …

参数 x+1 [第 1 台设备]第 X 个返回字节数据

参数 X+2 [第 1 台设备]CRC 低字节

参数 X+3 [第 1 台设备]CRC 高字节

参数 X+4 [第 2 台设备]error 码

参数 X+5 [第 2 台设备]ID

参数 X+5+1 [第 2 台设备]第 1 个返回字节数据

参数 X+5+2 [第 2 台设备]第 2 个返回字节数据

… …

参数 2X+5 [第 2 台设备]第 X 个返回字节数据

参数 2X+5+1 [第 2 台设备]CRC 低字节

参数 2X+5+ 2 [第 2 台设备]CRC 高字节

… …

参数 nX+5 [第 n 台设备]的第 X 个字节返回数据

注意：

2. 中间设备的 CRC 值用于电机内部运算，以确认各电机设备之间的数据包完整性。在解析数据过程只需对状态数据包末尾 CRC 值进行校验即可。

3. 各设备会分别返回状态数据包中的特定部分，以此响应快速同步读取指令。

例: 同时读取 ID3、ID4、ID7 的当前位置（132, 0x0084, 4[byte]）

ID3 当前位置为 166(0x000000A6)

ID4 当前位置为 1023(0x000003FF)

ID7 当前位置为 2,079(0x0000081F)。

同步读指令帧：FF FF FD 00 FE 0A 00 8A 84 00 04 00 03 07 04 20 F2 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	P1	P2	P3	P4	P5	P6	P7	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0xFE	0x0A	0x00	0x8A	0x84	0x00	0x04	0x00	0x03	0x07	0x04	0x20	0xF2

返回帧：

FF FF FD 00 FE 19 00 55 00 03 A6 00 00 00 84 08

00 07 1F 08 00 00 16 CA

00 04 FF 03 00 00 D1 9E

返回包拆解：ID3 返回包：FF FF FD 00 FE 19 00 55 00 03 A6 00 00 00 84 08 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	ID1	D1	D2	D3	D4	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0xFE	0x19	0x00	0x55	0x00	0x03	0xA6	0x00	0x00	0x00	0x84	0x08

ID7 返回包：00 07 1F 08 00 00 16 CA (按十六进制发送)

ERR	ID2	D1	D2	D3	D4	CRC 1	CRC 2
0x00	0x07	0x1F	0x08	0x00	0x00	0x16	0xCA

ID4 返回包：00 04 FF 03 00 00 D1 9E (按十六进制发送)

ERR	ID3	D1	D2	D3	D4	CRC 1	CRC 2
0x00	0x04	0xFF	0x03	0x00	0x00	0xD1	0x9E

2.2.9 批量读取 BULK READ

功能　　对于多个设备，一次从不同地址读取不同长度的数据

ID　0XFE

长度　0X16

指令　0x92

参数 1　[第 1 台设备]ID

参数 2　[第 1 台设备]起始地址的低序字节

参数 3　[第 1 台设备]起始地址的高序字节

参数 4　[第 1 台设备]数据长度的低序字节

参数 5　[第 1 台设备]数据长度的高阶字节

参数 6　[第 2 台设备]ID

参数 7　[第 2 台设备]起始地址的低序字节

参数 8　[第 2 台设备]起始地址的高序字节

参数 9　[第 2 台设备]数据长度的低序字节

参数 10 [第 2 台设备]数据长度的高阶字节

例: 批量读取 ID1 当前电压（144，0x0090，2[byte]），当前电压值为 119（0x0077）

ID2 当前温度（146，0x0092，1[byte]），当前位置值为 36（0x24）

批量写入指令帧：FF FF FD 00 FE 0D 00 92 01 90 00 02 00 02 92 00 01 00 1A 05 (按十六进制发送)

H	H	H	RS	I	LE	LE	IN	P	P	P	P	P	P	P	P	P	P1	CR	CR

1	2	3	RV	D	N1	N2	ST	1	2	3	4	5	6	7	8	9	0	C1	C2
0xFF	0xFF	0xFD	0x00	0xFE	0x0D	0x00	0x92	0x01	0x90	0x00	0x02	0x00	0x02	0x92	0x00	0x01	0x00	0x1A	0x05

返回帧:

FF FF FD 00 01 06 00 55 00 77 00 C3 69

FF FF FD 00 02 05 00 55 00 24 8B A9

返回包拆解：ID 1 状态数据包：FF FF FD 00 01 06 00 55 00 77 00 C3 69 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	P1	P2	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0x01	0x06	0x00	0x55	0x00	0x77	0x00	0xC3	0x69

返回包拆解：ID2 数据包：FF FF FD 00 02 05 00 55 00 24 8B A9 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	P1	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0x02	0x05	0x00	0x55	0x00	0x24	0x8B	0xA9

2.2.10 批量写入 BULK WRITE

功能 对于多个设备，一次从不同地址写入不同长度的数据

ID 0XFE

长度 0X13

指令 0x93

参数 1 [第 1 台设备]ID

参数 2 [第 1 台设备]起始地址的低序字节

参数 3 [第 1 台设备]起始地址的高序字节

参数 4 [第 1 台设备]数据长度 （X） 的低序字节

参数 5 [第 1 台设备]数据长度 （X） 的高位字节

参数 5+1 [第 1 台设备] 第 1 字节

…

参数 5+X [第 1 台设备]第 X 字节

参数 6+X [第 2 台设备]ID

参数 7+X [第 2 台设备]起始地址的低序字节

参数 8+X [第 2 台设备]起始地址的高序字节

参数 9+X [第 2 台设备]数据长度 （X）的低序字节

参数 10+X [第 2 台设备]数据长度 （X）的高位字节

参数 10+X+1 [第 2 台设备]第 1 字节

参数 10+X+2 [第 2 台设备]第 2 字节

…

参数 10+X+Y [第 2 台设备]第 Y 字节

例：批量写入 ID1 最大电压（32，0x0020， 2[byte]）为 160（0x00A0）

ID2 温度限制（31，0x001F，1[byte]）为 80（0x50）

批量写入指令帧：FF FF FD 00 FE 10 00 93 01 20 00 02 00 A0 00 02 1F 00 01 00 50 B7 68 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P12	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0xFE	0x10	0x00	0x93	0x01	0x20	0x00	0x02	0x00	0XA0	0x00	0x02	0x1F	0x00	0x01	0X00	0x50	0XB7	0X68

2.2.11 快速批量读取 Fast Bulk Read

该指令是相较于批量读取的增强型高速通信指令。无论所串联设备数量多少，使用快速批量读取指令时，所有设备将汇总生成并返回一个状态数据包，效果等同于单台设备返回单个状态数据包。本指令的数据包格式与批量读取指令完全一致。

功能 与 Bulk Read 相比，Enhanced Instruction 可实现更快的通信

ID 0XFE

长度 0X18

指令 0x9A

参数 1 [第 1 台设备]ID

参数 2 [第 1 台设备]起始地址的低序字节

参数 3 [第 1 台设备]起始地址的高序字节

参数 4 [第 1 台设备]来自数据的低序字节（X1）

参数 5 [第 1 台设备]来自数据的高阶字节（X1）

参数 6 [第 2 台设备]ID

参数 7 [第 2 台设备]起始地址的低序字节

参数 8 [第 2 台设备]起始地址的高序字节

参数 9 [第 2 台设备]来自数据的低序字节（X2）

参数 10 [第 2 台设备]来自数据的高阶字节（X2）

返回数据包参数说明

参数 1 [第 1 台设备]ID

参数 2 [第 1 台设备]第 1 个字节数据

参数 3 [第 1 台设备]第 2 个字节数据

… …

参数 x1+1 [第 1 台设备]第 X1 个字节数据

参数 X1+2 [第 1 台设备]CRC 低字节

参数 X1+3 [第 1 台设备]CRC 高字节

参数 X1+4 [第 2 台设备]error 码

参数 X1+5 [第 2 台设备]ID

参数 X1+5+1 [第 2 台设备]第 1 个字节数据

参数 X1+5+2 [第 2 台设备]第 2 个字节数据

… …

参数 X1+5 +X2 [第 2 台设备]第 X2 个字节数据

参数 X1+5 +X2 +1 [第 2 台设备]CRC 低字节

参数 X1+5 +X2 + 2 [第 2 台设备]CRC 高字节

… …

参数 X1+5+X2+… 5+Xn [第 n 台设备]的第 Xn 字节返回数据

例： 同时设置 ID3 当前位置（132，0x0084，4[byte]）为 166（0x000000A6）

ID4 当前温度（146，0x0092，1[byte]）为 31（0x1F）

ID7 当前 PWM（124，0x007C，2[byte]）为 421（0x01A5）

快速批量读取指令帧：FF FF FD 00 FE 12 00 9A 03 84 00 04 00 07 7C 00 02 00 04 92 00 01 00 20 F2 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0xFE	0x12	0x00	0x9A	0x03	0x84	0x00	0x04	0x00	0x07	0x7C	0x00	0x02	0x00	0x04	0x92	0x00	0x01	0x00	0x20	0xF2

返回帧：

FF FF FD 00 FE 14 00 55 00 03 A6 00 00 00 67 A4

00 07 A5 01 24 74

00 04 1F D9 C1

返回包拆解：ID 3 状态数据包返回包：FF FF FD 00 FE 14 00 55 00 03 A6 00 00 00 67 A4 (按十六进制发送)

H1	H2	H3	RSRV	ID	LEN1	LEN2	INST	ERR	ID1	D1	D2	D3	D4	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0xFE	0x14	0x00	0x55	0x00	0x03	0xA6	0x00	0x00	0x00	0x67	0xA4

ID7 返回包：00 07 A5 01 24 74 (按十六进制发送)

ERR	ID2	D1	D2	CRC1	CRC2
0x00	0x07	0xA5	0x01	0x24	0x74

ID4 返回包：00 04 1F D9 C1 00 04 1F (按十六进制发送)

ERR	ID2	D1	CRC1	CRC2
0x00	0x04	0x1F	0xD9	0xC1

CAN 通信控制协议 V2.0

CAN 通信目前支持多种协议：私有协议、MIT 协议。默认通信协议为 MIT 协议。可通过修改寄存器值切换通信协议类型，切换协议类型后，需重启设备生效。

GIM 系列电机 CAN 身份标识由寄存器：ID(7)和 CAN_ID(14)共同定义，其中 ID(7)用于电机接收 CAN 数据报文 ID，CAN_ID（14）用于电机反馈 CAN 数据报文 ID。电机 CAN 通信过程中，接收和发送数据报文 ID 使用不同值减少 CAN 通信过程中数据传输冲突，提高网络通信的可靠性。建议接收数据 ID(7)值与反馈数据 CAN_ID(14)设置不同值，且 ID(7)<CAN_ID(14)，接收数据优先级高于发送数据，可以达到更好的控制效果。

本文档中如无特别说明，所有多字节寄存器内容传输均采用**小端字节序**，即低位在前，高位在后。

1. 私有协议数据帧格式

CAN 私有协议通信，默认波特率 1Mbps，采用扩展帧格式。各字节段定义如下所示。

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	消息类型	数据区 2	目标电机 ID	数据区 1

1.1 设备搜索（0x00）

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x00	0x00	目标电机 ID	无

应答帧：

数据域	**29 位 ID**	**8 字节数据区**

大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x00	Bit16~23: 0x00 Bit15~8:目标电机 ID	目标电机 CAN_ID	Byte0~Byte1：电机型号 Byte2~Byte4：电机固件版本号 Byte5~Byte7: 无

1.2 电机反馈数据（0x01）

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x01	Bite16~23: 电机故障码（495） Bit16：供电电压异常 Bit17：保留 Bit18：保留 Bit19：保留 Bit20：FOC 超时 Bit21：过载 Bit22：电机温度过热 Bit23：温度传感异常 Bit15~Bit8：目标电机 ID	目标电机 CAN_ID	Byte0~ Byte1: 当前位置[0~65535]对应 MIT 位置最大最小值（寄存器 330,332） Byte2~ Byte3: 当前速度[0~65535]对应 MIT 速度最大最小值（寄存器 334,336） Byte4~ Byte5: 当前扭矩[0~65535]对应 MIT 扭矩最大最小值 Byte6：MOS 温度(570) Byte7：电机温度（571） 以上数据低字节在前，高字节在后

1.3 设置电机控制模式（0x02）

数据域	29 位 ID			8 字节数据区
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x02	0x00	目标电机 ID	Byte0：0:电流模式；1:速度模式；3：位置模式；6：MIT 模式。 Byte1~Byte7：无

应答帧：电机反馈数据（0x01）

1.4 电机使能（0x03）

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x03	0x00	目标电机 ID	无

应答帧：电机反馈数据（0x01）

1.5 电机失能（0x04）

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x04	0x00	目标电机 ID	无

应答帧：电机反馈数据（0x01）

1.6 设置当前位置为机械零位（0x05）

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7

描述	0x05	0x00	目标电机 ID	无

应答帧：电机反馈数据（0x01）

1.7 运控模式（MIT）电机指令（0x06）

该命令需在控制模式为：MIT 模式，扭矩开关打开情况下生效。

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7?
描述	0x06	Bit23~Bit8：目标扭矩[0~65535]对应 MIT 扭矩最大最小值	目标电机 ID	Byte0~Byte1:目标位置[0~65535]对应 MIT 位置最大最小值（寄存器 330,332） Byte2~Byte3:目标速度[0~65535]对应 MIT 速度最大最小值（寄存器 334,336） Byte4~Byte5: Kp [0~65535]对应 MIT 位置比例增益最大最小值（寄存器 342,344） Byte6~Byte7: Kd [0~65535]对应 MIT 位置微分增益最大最小值（寄存器 346,348）

应答帧：电机反馈数据（0x01）

注意：MIT 控制中的位置、速度、扭矩、KP、KD 的等比例转换最大最小值保存在 MIT 相关寄存器中，详情请查看第 3 节“CAN 通信 MIT 命令中浮点数与整数转换”中内容。

1.8 位置控制模式电机指令（0x07）

该命令需在控制模式为：位置模式，扭矩开关打开情况下生效。

数据域	**29 位 ID**	**8 字节数据区**

大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x07	0x00	目标电机 ID	Byte0~Byte3:目标位置。 低字节在前，高字节在后

应答帧：电机反馈数据（0x01）

1.9 速度控制模式电机指令（0x08）

该命令需在控制模式为：速度模式，扭矩开关打开情况下生效。

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x08	0x00	目标电机 ID	Byte0~Byte3:目标速度。 低字节在前，高字节在后

应答帧：电机反馈数据（0x01）

1.10 电流控制模式电机指令（0x09）

该命令需在控制模式为：速度模式，扭矩开关打开情况下生效。

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x09	0x00	目标电机 ID	Byte0~Byte1:目标电流。 低字节在前，高字节在后

应答帧：电机反馈数据（0x01）

1.11 写寄存器（0x0A）

该命令用于修改指定地址开始的寄存器数值，一次最多写 8 个字节。

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x0A	Bit23~Bit8：寄存器起始地址	目标电机 ID	Byte0~ Byte7：待写入寄存器数据 **注意**：需要写多少个字节就填充多少个字节数据。最多 8 个字节。

应答帧：电机反馈数据（0x01）

1.12 读取寄存器（0x0B）

该命令读取指定地址开始的寄存器数值，一次最多读取 8 个字节。

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x0B	0x00	目标电机 ID	Byte0~ Byte1: 寄存器起始地址 Byte2： 读取长度

应答帧：

数据域	**29 位 ID**			**8 字节数据区**
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x0B	0x00	目标电机 CAN_ID	Byte0~Byte7：指定寄存器的数据

1.13 读取电机状态 1（0x0C）

数据域	29 位 ID			8 字节数据区
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x0C	0x00	目标电机 ID	无

应答帧：电机反馈数据（0x01）

1.14 读取电机状态 2（0x0D）

数据域	29 位 ID			8 字节数据区
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x0D	0x00	目标电机 ID	无

该命令可一次读取到：Operator_mode、ExencoderBits、Torque、InputVoltage、ErrorCode 多个状态数据。

应答帧

数据域	29 位 ID			8 字节数据区
大小	Bit28~Bit24	Bit23~Bit8	Bit7~Bit0	Byte0~Byte7
描述	0x0D	0x00	目标电机 CAN_ID	Byte0：OperatorMode Byte1: ExencoderBits Byte2：Torque Byte3: INPUT_VOLTAGE(L) Byte4：INPUT_VOLTAGE(H) Byte5: ErrorCode Byte6：T_MOS Byte7: T_Rotor

- OperatorMode：控制模式（速度模式、电流模式、位置模式、MIT 模式等）。详见内存表定义文档
- ExencoderBits：外部编码器位数

- Torque：电机扭矩开关

6. INPUT_VOLTAGE：当前输入电压
3. ErrorCode：错误代码。详见内存表定义文档
4. T_MOS：表示驱动上 MOS 的平均温度，单位℃
5. T_Rotor： 表示电机内部线圈的平均温度，单位℃

2. MIT 协议数据帧格式

CAN MIT 协议通信默认波特率 1Mbps，采用标准帧格式。各字节段定义如下所示。

数据域	**11 位 ID**	**8 字节数据区**
大小	Bit10~Bit0	Byte0~Byte7
描述	目标电机 ID	数据区

MIT 协议中，以 CAN 数据长度来区分消息类型。发送给电机的 CAN 数据报文 ID 使用电机寄存器 ID(7)值填充，电机反馈数据报文 ID 使用电机寄存器 CAN_ID(14)值填充。下面对 CAN 数据区内容进行解释。

2.1 设备搜索（PING）

（1）发送数据

数据长度：1 个字节

数据内容： 0x01

（2）返回数据

数据长度：8 个字节

数据内容：

DATA0	**DATA1**	**DATA2**	**DATA3**
485_ID	CAN_ID	MODEL_NUMBER(L)	MODEL_NUMBER(H)
DATA4	**DATA5**	**DATA6**	**DATA7**

MAJOR_FIRMWARE_VERSION	MINOR_FIRMWARE_VERSION	PATCH_FIRMWARE_VERSION	DRIVE_MODE

6. 485_ID：485 通信 ID
3. CAN_ID：CAN 通信 ID
4. MODEL_NUMBER：电机型号（2 个字节）
5. MAJOR_FIRMWARE_VERSION：固件主版本号
6. MINOR_FIRMWARE_VERSION：固件副版本号
7. PATCH_FIRMWARE_VERSION：固件附加版本号
8. DRIVE_MODE：驱动模式。详见内存表定义文档

2.2 读取设备状态

该命令可一次读取到：Operator_mode、ExencoderBits、Torque、InputVoltage、ErrorCode 多个状态数据。

（1）发送数据

数据长度：1 个字节

数据内容： 0x02

（2）返回数据

数据长度：8 个字节

数据内容：

DATA0	**DATA1**	**DATA2**	**DATA3**
OperatorMode	ExencoderBits	Torque	INPUT_VOLTAGE (L)
DATA4	**DATA5**	**DATA6**	**DATA7**
INPUT_VOLTAGE (H)	ErrorCode	T_MOS	T_Rotor

3. OperatorMode：控制模式（速度模式、电流模式、位置模式、MIT 模式等）。详见内存表定义文档
4. ExencoderBits： 外部编码器位数
5. Torque：电机扭矩开关

6. INPUT_VOLTAGE：当前输入电压

3. ErrorCode：错误代码。详见内存表定义文档

4. T_MOS：表示驱动上 MOS 的平均温度，单位℃

4. T_Rotor： 表示电机内部线圈的平均温度，单位℃

2.3 读取寄存器

该命令读取指定地址开始的寄存器数值，一次最多读取 8 个字节。

（1）发送数据

数据长度：3 个字节

数据内容：

DATA0	DATA1	DATA2
Address (L)	Address (H)	Len

2. Address ： 寄存器开始地址（2 个字节）。详见内存表定义文档

3. Len：读取字节长度（小于等于 8）

（2）返回数据

读取成功，返回寄存器内容；读取失败，无返回。

数据长度：n 个字节

数据内容： 指定读取的寄存器数值

2.4 写寄存器

该命令用于修改指定地址开始的寄存器数值，一次最多写 4 个字节。

（1）发送数据

数据长度：7 个字节

数据内容：

DATA0	DATA1	DATA2	DATA3-DATA6
Address (L)	Address (L)	Len	Value

4. Address ： 寄存器开始地址（2 个字节）。详见内存表定义文档

5. Len：写寄存器字节长度（小于等于 4）

2. Value：待写入寄存器的值。（不够 4 个字节的，需要填充 FF。）

（2）返回数据

数据长度：1 个字节

数据内容： 0x00：失败；0x01：成功

2.5 打开扭矩开关

（1）发送数据

数据长度：8 字节

数据内容：FF FF FF FF FF FF FF FC

（2）返回数据

数据长度：8 字节

DATA0	DATA1	DATA2	DATA3	DATA4	DATA5	DATA6	DATA7
ID|ERR <<4	POS[15 :8]	POS[7: 0]	VEL[11: 4]	VEL[3:0]|T[11:8]	T[7:0]	T_MOS	T_Rotor

3. ID：控制器的 ID，取 CAN_ID 的低 8 位

4. ERR： 表示电机故障

- POS： 表示电机的当前位置
- VEL： 表示电机的当前速度
- T：表示电机的扭矩状态信
- T_MOS：表示驱动上 MOS 的平均温度，单位℃
- T_Rotor： 表示电机内部线圈的平均温度，单位℃

注意：位置、速度和扭矩采用线性映射的关系将浮点型数据转换成有符号的定点数据，其中位置采用 16 位数据，速度和扭矩均使用 12 位。

2.6 关闭扭矩开关

（1）发送数据

数据长度：8 字节

数据内容：FF FF FF FF FF FF FF FD

无返回数据

数据长度：0 字节

2.7 设置当前位置为零位

（1）发送数据

数据长度：8 字节

数据内容：FF FF FF FF FF FF FF FE

无返回数据

数据长度：0 字节

2.8 MIT 控制命令

（1）发送数据

数据长度：8 字节

数据内容：

DATA0	DATA1	DATA2	DATA3	DATA4	DATA5	DATA6	DATA7
p_des [15:8]	p_des [7:0]	v_des [11:4]	v_des[3 :0]| Kp[11:8]	Kp [7:0]	Kd [11:4]	Kd[3:0]| t_ff[11: 8]	t_ff[7:0]

- P_des：位置给定
- V_des：速度给定
- Kp：位置比例系数
- Kd：位置微分系数
- T_ff：转矩给定值

各参数符合前面的映射关系，其中 p_des、v_des、t_ff、Kp、Kd 的范围可通过上位机软件进行设定。标准 CAN 数据一帧只有 8 个字节，MIT 的控制命令格式将 Position、Velocity、Kp、Kd、 Torque 五个参数按位组合在 8 个字节中。其中：Position 占用 2 个字节 16 位、Velocity 占用 12 位、Kp 占用 12 位、Kd 占用 12 位。

（2）返回数据

数据长度：8 字节

DATA0	DATA1	DATA2	DATA3	DATA4	DATA5	DATA6	DATA7

ID|ERR <<4	POS[15 :8]	POS[7: 0]	VEL[11: 4]	VEL[3:0]|T[11:8]	T[7:0]	T_MOS	T_Rotor

- ID：控制器的 ID，取 CAN_ID 的低 8 位
- ERR： 电机故障
- POS： 电机的当前位置
- VEL： 电机的当前速度
- T：电机的扭矩状态信
- T_MOS：驱动上 MOS 的平均温度，单位℃
- T_Rotor： 电机内部线圈的平均温度，单位℃

---

3. CAN 通信 MIT 命令中浮点数与整数转换

MIT 命令采用浮点数据等比例转换成整数发送到驱动器，驱动器再将接收到的整数等比例转换成浮点数据。

这部分转换使用到 uint_to_float、 float_to_uint 两个转换函数进行转换，该部分具体实现如下所示：

```SQL
float uint_to_float(int x_int, float x_min, float x_max, int bits)
{
    /// converts unsigned int to float, given range and number of
bits ///
    float span = x_max - x_min;
    float offset = x_min;
    return ((float)x_int)*span/((float)((1<<bits)-1)) + offset;
 }

 int float_to_uint(float x, float x_min, float x_max, int bits)
 {
    /// Converts a float to an unsigned int, given range and
number of bits ///
    float span = x_max - x_min;
    float offset = x_min;
    return (int) ((x-offset)*((float)((1<<bits)-1))/span);
```

```
}
```

这两个转换函数需要首先确定两个等比例转换的最大最小值。**MIT 控制中的位置、速度、扭矩、KP、KD 的等比例转换最大最小值保存在寄存器中，可通过修改寄存器值来修改最大最小范围，**这些参数可以根据电机的实际参数进行调整。**但发送控制命令时，一定要与设定值保持一致， 否则会控制命令会发生等比例缩放。**

MIT 相关寄存器定义如下。

名称	寄存器地址	字节数	单位	默认值
MIT 最大位置	330	2	0.01 [rad]	1250
MIT 最小位置	332	2	0.01 [rad]	-1250
MIT 最大速度	334	2	0.01 [rad/s]	3000
MIT 最小速度	336	2	0.01 [rad/s]	-3000
MIT 最大扭矩	338	2	0.002 [N*M]	5000
MIT 最小扭矩	340	2	0.002 [N*M]	-5000
MIT 最大 KP	342	2	0.01	10000
MIT 最小 KP	344	2	0.01	0
MIT 最大 KD	346	2	0.01	500
MIT 最小 KD	348	2	0.01	0
MIT 控制 KP	350	2	0.01	7200
MIT 控制 KD	352	2	0.01	150

MIT 控制命令发送函数封装如下。

```
Markdown
```

```
/**************************************************************
*********
* @brief:      mit_ctrl: MIT 模式下的电机控制函数
* @param[in]:   hcan:    指向 CAN_HandleTypeDef 结构的指针，用于指定
CAN 总线
* @param[in]:   motor_id: 机 ID，指定目标电机
* @param[in]:   pos:      位置给定值
* @param[in]:   vel:      速度给定值
* @param[in]:   kp:       位置比例系数
* @param[in]:   kd:       速度微分系数
* @param[in]:   torq:     转矩给定值
* @retval:      void
* @details:    通过 CAN 总线向电机发送 MIT 模式下的控制帧。
**************************************************************
********/
void mit_ctrl(hcan_t* hcan, uint16_t motor_id, float pos, float
vel,float kp, float kd, float torq)
{
    uint8_t data[8];
    uint16_t pos_tmp,vel_tmp,kp_tmp,kd_tmp,tor_tmp;
    uint16_t id = motor_id + MIT_MODE;

    pos_tmp = float_to_uint(pos,  P_MIN,  P_MAX,  16);
    vel_tmp = float_to_uint(vel,  V_MIN,  V_MAX,  12);
    kp_tmp  = float_to_uint(kp,   KP_MIN, KP_MAX, 12);
    kd_tmp  = float_to_uint(kd,   KD_MIN, KD_MAX, 12);
    tor_tmp = float_to_uint(torq, T_MIN,  T_MAX,  12);

    data[0] = (pos_tmp >> 8);
    data[1] = pos_tmp;
    data[2] = (vel_tmp >> 4);
    data[3] = ((vel_tmp&0xF)<<4)|(kp_tmp>>8);
    data[4] = kp_tmp&0xFF;
    data[5] = (kd_tmp >> 4);
    data[6] = ((kd_tmp&0xF)<<4)|(tor_tmp>>8);
    data[7] = tor_tmp&0xFF;

    canx_send_data(hcan, id, data, 8);
}
```

MIT 控制命令返回数据解析函数封装如下

```
Markdown

/**************************************************************************
*********
* @brief:       获取电机反馈数据函数
* @param[in]:   motor:    指向 motor_t 结构的指针，包含电机相关信息和
反馈数据
* @param[in]:   rx_data:  指向包含反馈数据的数组指针
* @param[in]:   data_len: 数据长度
* @retval:     void
* @details:    从接收到的数据中提取电机的反馈信息，包括电机 ID、 状态、
位置、速度、扭矩相关温度参数、寄存器数据等
**************************************************************************
*******/
void can_fbdata(Joint_Motor_t *motor, uint8_t *rx_data,uint32_t
data_len)
{
   if(data_len==FDCAN_DLC_BYTES_8)
   {//返回的数据有 8 个字节
       motor->para.id = (rx_data[0])&0x0F;
       motor->para.state = (rx_data[0])>>4;
       motor->para.p_int=(rx_data[1]<<8)|rx_data[2];
       motor->para.v_int=(rx_data[3]<<4)|(rx_data[4]>>4);
       motor->para.t_int=((rx_data[4]&0xF)<<8)|rx_data[5];
       motor->para.pos = uint_to_float(motor->para.p_int, P_MIN,
P_MAX, 16); // (-12.5,12.5)
       motor->para.vel = uint_to_float(motor->para.v_int, V_MIN,
V_MAX, 12); // (-30.0,30.0)
       motor->para.tor = uint_to_float(motor->para.t_int, T_MIN,
T_MAX, 12);  // (-10.0,10.0)
       motor->para.Tmos = (float)(rx_data[6]);
       motor->para.Tcoil = (float)(rx_data[7]);
    }
}
```

MIT 协议测试 Demo

使用该 Demo 程序控制电机，需要配合标配的 CAN 通信模块（SLCAN 协议），同时需要 Python3.0 以上环境，安装 can 模块。

```
Plain Text
#运行命令安装 can 模块
pip install python-can
```

该 Demo 程序实现功能：实现对多个电机的扫描控制。可根据实际需求修改 COM 口、波特率、目标位置范围、目标位置变化步长，控制频率等变量。

- 修改 COM 口

```
Plain Text
 SLAN_PORT = 'COM29'
```

2.修改电机 ID

```
Plain Text
  MOTOR_CAN_IDS = [  0x01, 0x02,0x3, 0x04, 0x05,0x06]
```

3.修改控制频率

```
Plain Text
 LOOP_FREQUENCY = 1000
```

4.修改目标位置范围和步长

```
Plain Text
MAX_GOAL_POS_RAD = 1.0 * math.pi  # 180 度，可根据需要调整
MIN_GOAL_POS_RAD = -1.0 * math.pi  # -180 度
GOAL_POS_STEP_RAD = 0.001 * math.pi  # 每次更新的步长
```

完整代码如下：

```
Python
import can
import time
import struct
```

```
import math

RAD_TO_SIGNAL = 2607.5946

# 定义电机命令
CMD_PING = 0x01
CMD_READ_STATUS = 0x02  # “02 查询”

# 寄存器地址常量
PRESENT_POSITION = 0x228            # 当前位置寄存器地址 552 = 0x228
GOAL_POSITION = 0x214               # 目标位置寄存器地址 532 = 0x214
TORQUE_ON_OFF_REGISTER = 0x200      # 扭矩开关寄存器地址 512 = 0x200
OPENTING_MODEL_REGISTER = 0x21      # Operating Mode 寄存器地址 33 =
0x21
SET_ZERO_POSITION_REGISTER = 0x34   # 设置零位寄存器 52= 0x34
SET_ZERO_POSITION_VALUE = 0x01      # 设置零位的值

TORQUE_ON_VALUE = 0x01       # 打开扭矩的值
TORQUE_OFF_VALUE = 0x00      # 关闭扭矩的值
# 设置 OPENTING_MODEL 的值 (0：电流模式；1：速度模式；3:位置模式；6：
MIT 模式
OPENTING_MODEL_VALUE = 0x03


# MIT 控制参数的映射范围（示例，请根据实际文档调整）
P_DES_MIN = -12.5
P_DES_MAX = 12.5
V_DES_MIN = -65.0
V_DES_MAX = 65.0
T_FF_MIN = -18.0
T_FF_MAX = 18.0
KP_MIN = 0.0
KP_MAX = 500.0
KD_MIN = 0.0
KD_MAX = 5.0

# 位置/速度/扭矩的实际浮点数范围 (用于 uint_to_float 转换)
# 这些需要根据电机的实际规格来定义
POS_MIN_RANGE = -12.5 * math.pi  # 假设是弧度，根据电机文档调整
POS_MAX_RANGE = 12.5 * math.pi
VEL_MIN_RANGE = -65.0 * math.pi  # 假设是弧度/秒
VEL_MAX_RANGE = 65.0 * math.pi
```

```
TORQUE_MIN_RANGE = -18.0
TORQUE_MAX_RANGE = 18.0


# 将浮点数映射到定点数的函数
def float_to_uint(value, min_val, max_val, num_bits):
    if value < min_val:
        value = min_val
    if value > max_val:
        value = max_val
    scaled_value = (value - min_val) / (max_val - min_val)
    return int(scaled_value * ((1 << num_bits) - 1))


# 将定点数映射回浮点数的函数
def uint_to_float(value, min_val, max_val, num_bits):
    scaled_value = value / ((1 << num_bits) - 1)
    return scaled_value * (max_val - min_val) + min_val


# 对于带符号的定点数，需要做额外的处理
def signed_uint_to_float(value, min_val, max_val, num_bits):
    if value & (1 << (num_bits - 1)):
        value = value - (1 << num_bits)
    range_span = max_val - min_val
    integer_span = (1 << num_bits) - 1  # 注意：对于带符号数，范围可
能不是 (1<<num_bits)-1，而是 (1<<(num_bits-1))-1 到 -(1<<(num_bits-
1))
    return min_val + (value / integer_span) * range_span


class MotorSLCan:
    def __init__(self, port, baudrate=1000000):
        self.port = port
        self.baudrate = baudrate
        self.bus = None

    def connect(self):
        try:
            self.bus = can.interface.Bus(interface='slcan',
                                         channel=self.port,
                                         bitrate=self.baudrate)
            print(f"Connected to SLCan on {self.port} with bitrate
{self.baudrate}")
```

```
            return True
        except Exception as e:
            print(f"Error connecting to SLCan: {e}")
            return False

    def disconnect(self):
        if self.bus:
            self.bus.shutdown()
            print("Disconnected from SLCan.")

    def send_command(self, arbitration_id, data,
is_extended_id=False):
        try:
            message = can.Message(
                arbitration_id=arbitration_id,
                is_extended_id=is_extended_id,
                data=data
            )
            self.bus.send(message)
            return True
        except Exception as e:
            print(f"Error sending CAN message: {e}")
            return False

    def receive_response(self, timeout=1.0):
        try:
            message = self.bus.recv(timeout)
            if message:
                return message
            else:
                return None
        except Exception as e:
            print(f"Error receiving CAN message: {e}")
            return None

    def ping_device(self, can_id):
        self.send_command(can_id, bytearray([CMD_PING]))
        response = self.receive_response(timeout=0.2)  # PING 响应
可能需要稍长一些
        if response and len(response.data) == 8:
            _485_id = response.data[0]
            can_id_response = response.data[1]
            model_number = struct.unpack('<H',
response.data[2:4])[0]
```

```
            major_fw = response.data[4]
            minor_fw = response.data[5]
            patch_fw = response.data[6]
            drive_mode = response.data[7]
            print(f"--- PING Response (ID: {hex(can_id)}) ---")
            print(
                f"  485 ID: {hex(_485_id)}, CAN ID:
{hex(can_id_response)}, Model: {model_number}, FW:
{major_fw}.{minor_fw}.{patch_fw}")
            return {
                "485_ID": _485_id, "CAN_ID": can_id_response,
"MODEL_NUMBER": model_number,
                "MAJOR_FIRMWARE_VERSION": major_fw,
"MINOR_FIRMWARE_VERSION": minor_fw,
                "PATCH_FIRMWARE_VERSION": patch_fw, "DRIVE_MODE":
drive_mode
            }
        else:
            print(f"PING failed or invalid response for CAN ID
{hex(can_id)}.")
            return None

    def read_device_status(self, can_id):
        self.send_command(can_id, bytearray([CMD_READ_STATUS]))
        response = self.receive_response(timeout=0.05)  # 状态查询
响应时间可以短一些
        if response and len(response.data) == 8:
            operator_mode = response.data[0]
            exencoder_bits = response.data[1]
            torque = response.data[2]
            input_voltage = struct.unpack('<H',
response.data[3:5])[0]
            error_code = response.data[5]
            t_mos = response.data[6]
            t_rotor = response.data[7]
            # print(f"--- Device Status (ID: {hex(can_id)}) ---")
            # print(f"  Mode: {operator_mode}, Voltage:
{input_voltage / 100.0:.2f}V, Error: {hex(error_code)}, T_MOS:
{t_mos}, T_Rotor: {t_rotor}")
            return {
                "OperatorMode": operator_mode, "ExencoderBits":
exencoder_bits, "Torque": torque,
                "INPUT_VOLTAGE": input_voltage, "INPUT_VOLTAGE_V":
input_voltage / 100.0,
```

```
                "ErrorCode": error_code, "T_MOS": t_mos,
"T_Rotor": t_rotor
            }
        else:
            # print(f"Read device status failed or invalid
response for CAN ID {hex(can_id)}.")
            return None

    def read_register(self, can_id, address, length):
        if not (1 <= length <= 8):
            print(f"Error: Register read length must be between 1
and 8. (CAN ID: {hex(can_id)})")
            return None

        data = bytearray([
            (address & 0xFF),
            ((address >> 8) & 0xFF),
            length
        ])
        self.send_command(can_id, data)
        response = self.receive_response(timeout=0.05)  # 寄存器读
取响应时间可以短一些
        if response and len(response.data) == length:
            # print(f"--- Read Register (ID: {hex(can_id)}, Addr:
{hex(address)}, Len: {length}) --- Data: {response.data.hex()}")
            return response.data
        else:
            # print(f"Read register failed or invalid response for
CAN ID {hex(can_id)}, Address {hex(address)}.")
            # if response:
            #     print(f"  Received data length:
{len(response.data)}, Expected: {length}")
            #     print(f"  Received data: {response.data.hex()}")
            return None

    def write_register(self, can_id, address, value_bytes):
        if not (1 <= len(value_bytes) <= 4):
            print(
                f"Error: Register write value length must be
between 1 and 4 bytes. Got {len(value_bytes)} bytes. (CAN ID:
{hex(can_id)})")
            return False

        padded_value_bytes = bytearray(value_bytes)
```

```
        while len(padded_value_bytes) < 4:
            padded_value_bytes.append(0xFF)  # 文档说是 FF 填充，但
实际中 00 更常见，请核实

        data = bytearray([
            (address & 0xFF),
            ((address >> 8) & 0xFF),
            len(value_bytes)
        ])
        data.extend(padded_value_bytes)

        self.send_command(can_id, data)
        response = self.receive_response(timeout=0.05)  # 写寄存器
响应时间可以短一些
        if response and len(response.data) >= 1:
            status = response.data[0]
            if status == 0x01:
                # print(f"Write register (ID: {hex(can_id)}, Addr:
{hex(address)}, Val: {value_bytes.hex()}) successful (Status:
{hex(status)}).")
                return True
            else:
                print(
                    f"Write register (ID: {hex(can_id)}, Addr:
{hex(address)}, Val: {value_bytes.hex()}) failed (Status:
{hex(status)}).")
                return False
        else:
            print(f"Write register failed or invalid response for
CAN ID {hex(can_id)}, Address {hex(address)}.")
            if response:
                print(f"  Received response data:
{response.data.hex()}")
            return False

    def open_torque(self, can_id):
        return self.write_register(can_id, TORQUE_ON_OFF_REGISTER,
bytearray([TORQUE_ON_VALUE]))

    def close_torque(self, can_id):
        return self.write_register(can_id, TORQUE_ON_OFF_REGISTER,
bytearray([TORQUE_OFF_VALUE]))

    def set_zero_position(self, can_id):
```

```
        return self.write_register(can_id,
SET_ZERO_POSITION_REGISTER, bytearray([SET_ZERO_POSITION_VALUE]))

    def set_openting_model(self, can_id):
        return self.write_register(can_id,
OPENTING_MODEL_REGISTER, bytearray([OPENTING_MODEL_VALUE]))

    def send_mit_control(self, can_id, p_des, v_des, kp, kd,
t_ff):
        # 暂时不修改这部分，因为它不属于你当前的主要需求，但保留
        p_des_int = float_to_uint(p_des, P_DES_MIN, P_DES_MAX, 16)
        v_des_int = float_to_uint(v_des, V_DES_MIN, V_DES_MAX, 12)
        kp_int = float_to_uint(kp, KP_MIN, KP_MAX, 12)
        kd_int = float_to_uint(kd, KD_MIN, KD_MAX, 12)
        t_ff_int = float_to_uint(t_ff, T_FF_MIN, T_FF_MAX, 12)

        data = bytearray(8)
        data[0] = (p_des_int >> 8) & 0xFF
        data[1] = p_des_int & 0xFF
        data[2] = (v_des_int >> 4) & 0xFF
        data[3] = ((v_des_int & 0x0F) << 4) | ((kp_int >> 8) &
0x0F)
        data[4] = kp_int & 0xFF
        data[5] = (kd_int >> 4) & 0xFF
        data[6] = ((kd_int & 0x0F) << 4) | ((t_ff_int >> 8) &
0x0F)
        data[7] = t_ff_int & 0xFF

        self.send_command(can_id, data)
        response = self.receive_response(timeout=0.05)
        if response and len(response.data) == 8:
            return self._parse_mit_response(response.data)
        else:
            print(f"MIT control failed or invalid response for CAN
ID {hex(can_id)}.")
            return None

    def _parse_mit_response(self, data):
        # 暂时不修改这部分，因为它不属于你当前的主要需求，但保留
        motor_id = data[0] & 0x0F
        err_code = (data[0] >> 4) & 0x0F
        pos_raw = (data[1] << 8) | data[2]
        vel_raw = ((data[3] << 4) | ((data[4] >> 4) & 0x0F))
        torque_raw = (((data[4] & 0x0F) << 8) | data[5])
```

```
        t_mos = data[6]
        t_rotor = data[7]
        # print(f"--- MIT/Torque Response (ID: {hex(motor_id)}) --
- Err: {err_code}, Pos: {pos_raw}, Vel: {vel_raw}, T:
{torque_raw}")
        return {
            "ID": motor_id, "ERR": err_code, "POS_RAW": pos_raw,
            "VEL_RAW": vel_raw, "T_RAW": torque_raw, "T_MOS":
t_mos, "T_Rotor": t_rotor
        }


if __name__ == "__main__":
    SLAN_PORT = 'COM29'
    # 四个驱动器的 CAN ID
    MOTOR_CAN_IDS = [  0x01, 0x02,0x3, 0x04, 0x05,0x06]

    motor_comm = MotorSLCan(SLAN_PORT)

    if motor_comm.connect():
        try:
            # 存储每个电机的最后操作时间，用于调度
            last_status_query_time = {can_id: 0 for can_id in
MOTOR_CAN_IDS}
            last_pos_query_time = {can_id: 0 for can_id in
MOTOR_CAN_IDS}
            last_goal_update_time = {can_id: 0 for can_id in
MOTOR_CAN_IDS}

            # 目标位置生成器的状态
            current_goal_position_rad = {can_id: 0.0 for can_id in
MOTOR_CAN_IDS}
            goal_position_direction = {can_id: 1 for can_id in
MOTOR_CAN_IDS}  # 1 for increasing, -1 for decreasing
            MAX_GOAL_POS_RAD = 1.0 * math.pi  # 180 度，可根据需要
调整
            MIN_GOAL_POS_RAD = -1.0 * math.pi  # -180 度
            GOAL_POS_STEP_RAD = 0.001 * math.pi  # 每次更新的步长

            print("\n--- INITIALIZING MOTORS ---")
            for motor_id in MOTOR_CAN_IDS:
                print(f"\nInitializing Motor {hex(motor_id)}:")

                # 1. PING
```

```
                print(f"  Pinging...")
                motor_info = motor_comm.ping_device(motor_id)
                if not motor_info:
                    print(f"  Failed to PING motor
{hex(motor_id)}. Skipping initialization.")
                    continue  # 如果 PING 失败，跳过这个电机的初始化


                # 2. 设置 OPENTING_MODEL
                print(f"  Setting OPENTING_MODEL...")
                if not motor_comm.set_openting_model(motor_id):
                    print(f"  Failed to set OPENTING_MODEL for
motor {hex(motor_id)}.")
                    continue
                #time.sleep(0.1)
                # 3. 打开扭矩开关
                print(f"  Opening torque...")
                if not motor_comm.open_torque(motor_id):
                    print(f"  Failed to open torque for motor
{hex(motor_id)}.")
                    continue


            print("\n--- STARTING CONTROL LOOP ---")

            # 主循环频率控制
            LOOP_FREQUENCY = 1000  # Hz
            LOOP_PERIOD = 1.0 / LOOP_FREQUENCY  # seconds

            # 指令发送频率
            STATUS_QUERY_PERIOD = 0.100  # 100 ms
            POS_QUERY_PERIOD = 0.010  # 10 ms
            GOAL_UPDATE_PERIOD = 0.010  # 10 ms

            start_time = time.monotonic()
            loop_count = 0

            while True:
                loop_start_time = time.monotonic()

                for motor_id in MOTOR_CAN_IDS:
                    current_time = time.monotonic()

                    # ① 0x02 查询 (100ms)
```

```
                    if current_time -
last_status_query_time[motor_id] >= STATUS_QUERY_PERIOD:
                        status =
motor_comm.read_device_status(motor_id)
                        #if status:
                            # print(
                            #    f"Loop {loop_count}: Motor
{hex(motor_id)} Status: Mode={status['OperatorMode']},
Volt={status['INPUT_VOLTAGE_V']:.2f}V,
Temp={status['T_MOS']}C/{status['T_Rotor']}C")
                        last_status_query_time[motor_id] =
current_time

                    # ② 位置查询 (10ms)
                    if current_time -
last_pos_query_time[motor_id] >= POS_QUERY_PERIOD:
                        # 假设 PRESENT_POSITION 寄存器返回 4 字节
（32 位）表示位置
                        pos_raw_bytes =
motor_comm.read_register(motor_id, PRESENT_POSITION, 4)
                        if pos_raw_bytes and len(pos_raw_bytes) ==
4:
                            pos_raw = struct.unpack('<i',
pos_raw_bytes)[0]  # 假设无符号 16 位小端
                            # 将原始整数位置转换为浮点数 (需要根据你
的电机文档提供正确的映射范围和位数)
                            # 这里我假设 PRESENT_POSITION 也是 16
位，范围与 MIT 的 P_DES 类似
                            #actual_pos_rad =
uint_to_float(pos_raw, POS_MIN_RANGE, POS_MAX_RANGE, 16)
                            #print(
                            #   f"Loop {loop_count}: Motor
{hex(motor_id)} Pos: {pos_raw} ")
                        last_pos_query_time[motor_id] =
current_time

                    # ③ 更新目标位置 (10ms)
                    if current_time -
last_goal_update_time[motor_id] >= GOAL_UPDATE_PERIOD:
                        # 简单的目标位置生成：在 MIN_GOAL_POS_RAD 和
MAX_GOAL_POS_RAD 之间往复运动
                        if goal_position_direction[motor_id] == 1:
                            current_goal_position_rad[motor_id] +=
```

```
GOAL_POS_STEP_RAD
                            if
current_goal_position_rad[motor_id] >= MAX_GOAL_POS_RAD:

current_goal_position_rad[motor_id] = MAX_GOAL_POS_RAD
                                goal_position_direction[motor_id]
= -1
                        else:
                            current_goal_position_rad[motor_id] -=
GOAL_POS_STEP_RAD
                            if current_goal_position_rad[motor_id]
<= MIN_GOAL_POS_RAD:

current_goal_position_rad[motor_id] = MIN_GOAL_POS_RAD
                                goal_position_direction[motor_id]
= 1

                        # 将浮点目标位置转换为电机期望的整数格式（假设
是 16 位，与 P_DES_MIN/MAX 类似）
                        #goal_pos_int =
float_to_uint(current_goal_position_rad[motor_id], POS_MIN_RANGE,
POS_MAX_RANGE, 16)
                        goal_pos_int = int(RAD_TO_SIGNAL *
current_goal_position_rad[motor_id])
                        goal_bytes = struct.pack('<i',
goal_pos_int)  # 假设有符号的 4 字节小端

                        # 写入 GOAL_POSITION 寄存器
                        motor_comm.write_register(motor_id,
GOAL_POSITION, goal_bytes)
                        #print(f"Loop {loop_count}: Motor
{hex(motor_id)} Set Goal:
{current_goal_position_rad[motor_id]:.4f} rad (raw:
{goal_pos_int})")
                        last_goal_update_time[motor_id] =
current_time

                loop_count += 1

                # 确保主循环以 3000Hz 运行
                loop_end_time = time.monotonic()
                time_elapsed = loop_end_time - loop_start_time
                time_to_sleep = LOOP_PERIOD - time_elapsed
                if time_to_sleep > 0:
```

```
                    time.sleep(time_to_sleep)
                else:
                    # 如果循环执行时间超过了周期，打印警告
                    #print(f"Warning: Loop {loop_count} took
{time_elapsed*1000:.2f} ms, exceeding {LOOP_PERIOD*1000:.2f} ms.")
                    pass

        except KeyboardInterrupt:
            print("\nExiting program, closing torque for all
motors.")
            for motor_id in MOTOR_CAN_IDS:
                motor_comm.close_torque(motor_id)
                time.sleep(0.1)  # 间隔一下，确保命令发出
        finally:
            motor_comm.disconnect()
    else:
        print("Failed to connect to SLCan, please check port and
device.")
```

485/TTL 通信 SDK

我们提供跨平台的多语言 C、C++、python、ROS 接口 SDK，助力您快速使用 GIM 系列产品。