USB 扩展模块用户手册

2023

Version 1.0

目录

版权声明

本手册中的信息资料仅供参考。由于改进设计和功能等原因，高川自动化保留对本资料的最终解释权，内容如有更改，不另行通知。

调试、运动中的机器有危险！用户有责任在机器中设计有效的出错处理和安全保护机制，高川自动化没有义务或责任对由此造成的附带的或相应产生的损失负责。

联系我们

深圳市高川自动化技术有限公司
电话：0755-23502680
邮箱：sales@gcauto.com.cn
网址：www.gcauto.com.cn

Shenzhen Gaochuan Industrial Automation Co., Ltd.
Tel: 0755-23502680
Email: sales@gcauto.com.cn
Website: www.gcauto.com.cn

文档版本

版本号	修订日期	内容
V1.0	2023 年 1 月 29 日	-

前言

为了给用户提供更快捷，更方便的服务，提高用户的工作效率，本手册主要针对 USB 扩展模块硬件使用上的讲解，包括控制器的产品概述，配线指导，模块测试和软件编程，方便用户更好的使用我们的产品。

1 产品概述

USBIO64M01，USBIO32M01 和 USBIO24M01 模块提供一种低成本、简单易用、功能可靠的 IO 扩展方式，通过 RS485 或 USB2.0 接口与外部连接，可搭配我司的 GCN 系列控制器或者单独使用，无需安装驱动。

1.1 IO64M01(数字量)

1.1.1 尺寸图

如图 1.1.1 为 IO64M01 模块的外观及尺寸：

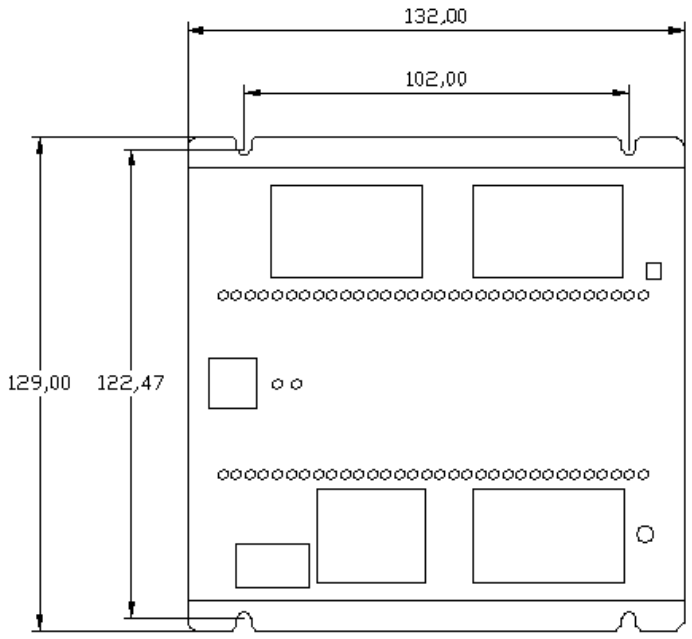

图 1.1.1 IO64M01 外观及尺寸

1.1.2 电气规格

隔离数字量输入	
通道	32
输入类型	干节点/湿节点
输入阻抗	5KΩ
隔离保护	2500 VDC
过压保护	70 VDC

ESD	2000 VDC
输入电流	3.5mA@0 VDC
输入电压	Logic 0: 4Vmax. Logic 1: 5V min(50V max)
隔离数字量输出	
通道	32
输出类型	Sink（NPN）,短路保护
输出电压	Logic 0: 0.5Vmax. Logic 1: 开路(50V max)
隔离保护	2500 VDC
Sink 电流	100mA
通讯	
通讯接口	USB、RS485
常规规格	
尺寸	130.5X125mm
系统供电	24VDC +/- 20%
湿度	5 ~ 95% RH, non-condensing (IEC 68-2-3)
工作温度	0 ~ 60° C (32 ~ 140° F)
存储温度	-20 ~ 85° C (-4 ~ 185° F)
软件支持	
操作系统	Windows XP/7/8/10/11 Linux WinCE
软件兼容性	VB/VC++/BCB/C#
演示软件	GCS.EXE

1.2 IO32M01(数字量)

1.2.1 尺寸图

如图 1.2.1 为 IO32M01 模块的外观及尺寸：

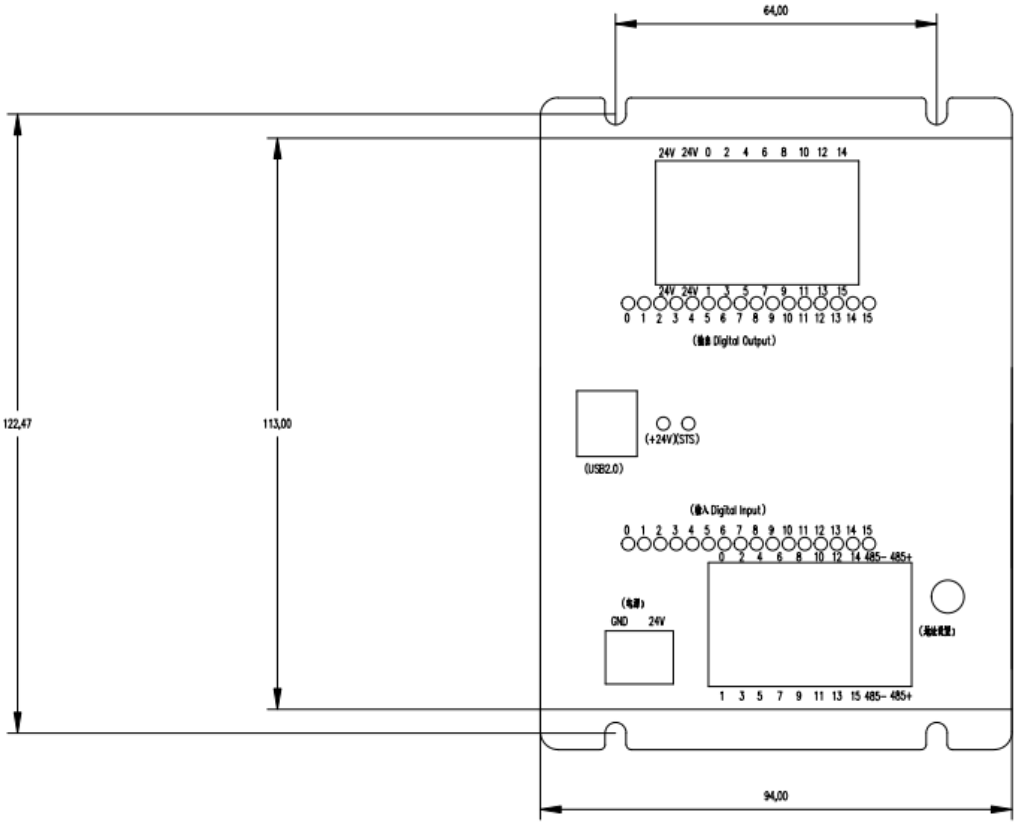

图 1.2.1 IO32M01 外观及尺寸

1.2.2 电气规格

隔离数字量输入	
通道	16
输入类型	干节点/湿节点
输入阻抗	5KΩ
隔离保护	2500 VDC
过压保护	70 VDC
ESD	2000 VDC
输入电流	3.5mA@0 VDC
输入电压	Logic 0: 4Vmax. Logic 1: 5V min(50V max)
隔离数字量输出	
通道	16
输出类型	Sink（NPN）,短路保护

输出电压	Logic 0: 0.5Vmax. Logic 1: 开路(50V max)
隔离保护	2500 VDC
Sink 电流	100mA
通讯	
通讯接口	USB、RS485
常规规格	
尺寸	94X123mm
系统供电	24VDC +/- 20%
湿度	5 ~ 95% RH, non-condensing (IEC 68-2-3)
工作温度	0 ~ 60° C (32 ~ 140° F)
存储温度	-20 ~ 85° C (-4 ~ 185° F)
软件支持	
操作系统	Windows XP/7/8/10/11 Linux WinCE
软件兼容性	VB/VC++/BCB/C#
演示软件	GCS.EXE

1.3 IO24M01(模拟量)

1.3.1 尺寸图

如图 1.3.1 为 IO24M01 模块的外观及尺寸：

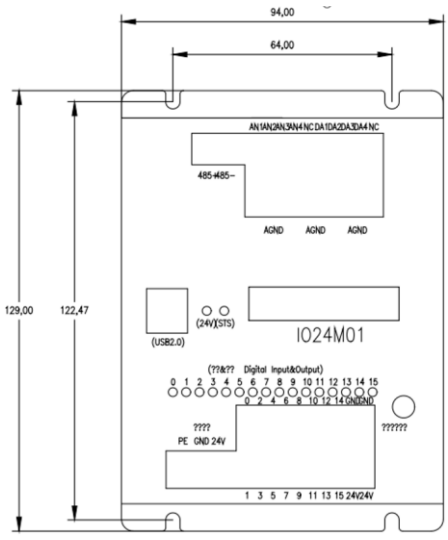

图 1.3.1 IO24M01 外观及尺寸

1.3.2 电气规格

隔离数字量输入	
通道	16 路(输入&输出)共用
输入类型	干节点/湿节点
输入阻抗	5KΩ
隔离保护	2500 VDC
过压保护	70 VDC
ESD	2000 VDC
输入电流	3.5mA@0 VDC
输入电压	Logic 0: 4Vmax. Logic 1: 5V min(50V max)
隔离数字量输出	
通道	16 路(输入&输出)共用
输出类型	Sink（NPN）,短路保护
输出电压	Logic 0: 0.5Vmax. Logic 1: 开路(50V max)
隔离保护	2500 VDC
Sink 电流	100mA
隔离模拟量输入	
通道	4
输入阻抗	5KΩ
隔离保护	2500 VDC
过压保护	70 VDC
ESD	2000 VDC
输入电流	3.5mA@0 VDC
输入电压范围	-10V ~ +10V
精度	12 位精度
隔离模拟量输出	
通道	4

输出电压范围	0 - 10V
隔离保护	2500 VDC
精度	12 位精度
通讯	
通讯接口	USB、RS485
常规规格	
尺寸	130.5X125mm
系统供电	24VDC +/- 20%
湿度	5 ~ 95% RH, non-condensing (IEC 68-2-3)
工作温度	0 ~ 60° C (32 ~ 140° F)
存储温度	-20 ~ 85° C (-4 ~ 185° F)
软件支持	
操作系统	Windows XP/7/8/10/11 Linux WinCE
软件兼容性	VB/VC++/BCB/C#
演示软件	GCS.EXE

2 配线指导

IO64M01，IO32M01和IO24M01在配线和测试使用中是相同的，下面对模块作统一说明。

2.1 GCN控制器配线

IO 模块搭配控制器接线示意图如图 2.1.1，黄色连接线为 RS485 通讯线：

图 2.1.1. IO 模块接线示意图

IO24M01 模块的上下两排的可拔插式接线端子分别为：输入、输出、模拟量输入、模拟量输出电源和 485 通讯,输入输出部分请参考图 2.1.2 接线;

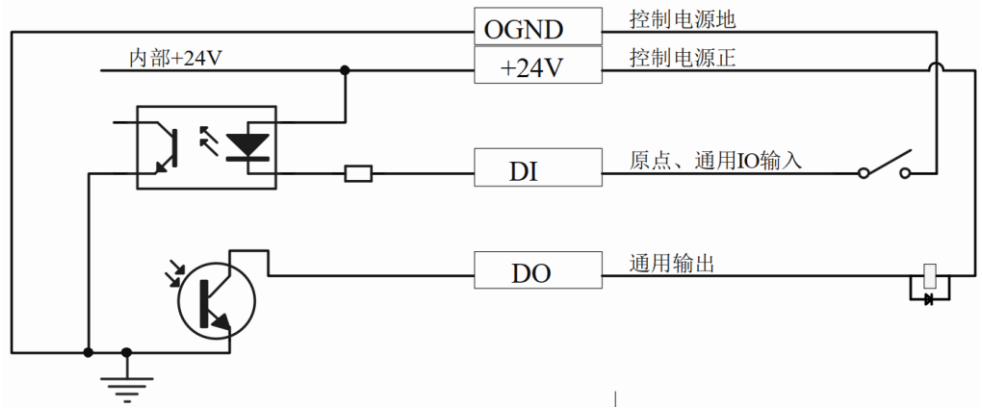

图 2.1.2 数字输入输出配线示意图

⚠	（1）当扩展模块与控制器配合使用时，模块上的拨码地址从 2 开始，一个控制器最多挂接 8 个扩展模块； （2）当模块单独使用时，拨码地址无效，一根 USB 线与模块单独通信，模块之间用 RS485 连接无效；

2.2 单独使用配线

IO 模块搭配控制器接线示意图如图 2.2.1，黄色连接线为 USB 通讯线：

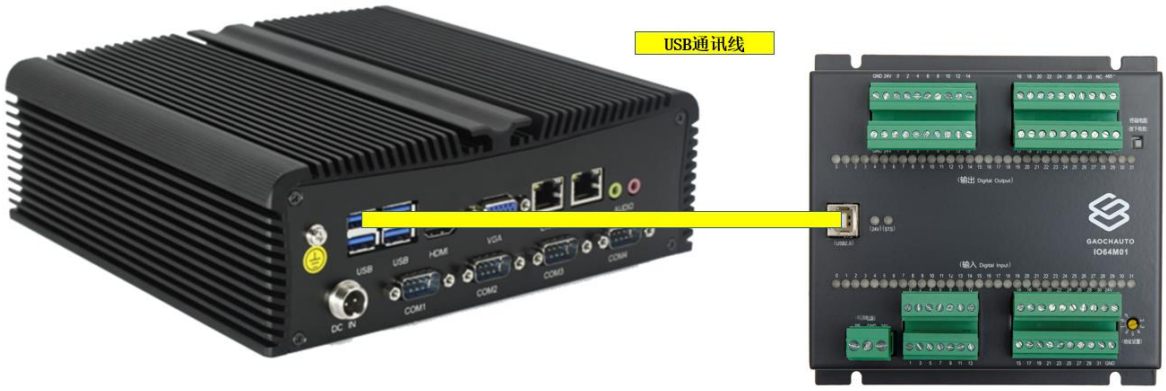

图 2.2.1 IO 模块单独使用连接图

3 模块测试

3.1 搭配GCN控制器

（1）硬件正确连接后上电；

（2）启动 GCS 工具，点击菜单“功能”->“IO 相关”->IO 扩展模块(通用 IO 扩展模块)，启动扩展模块测试模块，如图 3.1.1 数字量扩展模块和图 3.1.2 模拟量扩展模块测试界面；

（3）IO24M02 输入输出测试：选择模块类型为“IO32_DA”，点击“使能”使通讯状态变为绿色，表示扩展模块通讯正常，通道 1 为数字量，通道 2-5 为模拟量；(如果打开“通用 IO 扩展模块”时，模块类型为 IO64,请选择 IO32_DA 后，点击使能，重新打开 GCS 工具)

（4）IO 地址到连接模块的地址，通讯状态变为绿色，表示扩展模块通讯正常，开始输入输出测试；

图 3.1.1 数字量扩展模块测试界面

图 3.1.2 模拟量扩展模块测试界面

3.2 单独使用

电脑与 IO 扩展模块通过 USB 正常连接后，打开 GCS 工具，在菜单栏选择“工具”->“USB-IO”,即可以打开单独使用测试界面如图 3.2.1；

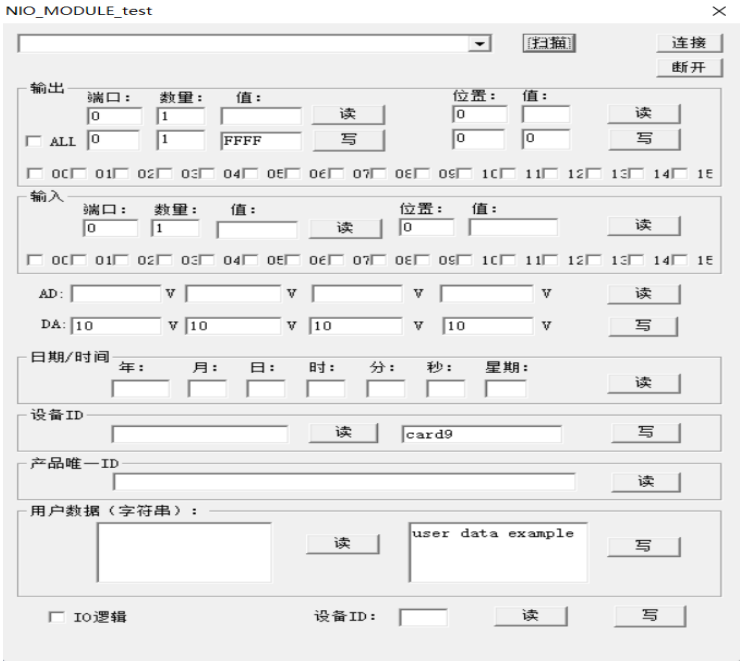

图 3.2.1 IO 单独使用测试界面

	如果在 GCS 中打开“USB-IO”工具失败，请联系我司技术人员；

4 软件编程

4.1 挂接控制器

函数原形	函数说明
NMC_SetDOGroup	设置扩展 IO 输出(按通道,支持超过 32 位),带默认 group
NMC_GetDOGroup	读取扩展 IO 输出(按通道,支持超过 32 位) ,带默认 group
NMC_SetDOBit	按位设置扩展 IO 输出
NMC_GetDOBit	按位读取扩展 IO 输出
NMC_GetDIGroup	读扩展 IO 输入(按通道,支持超过 32 位) ,带默认 group
NMC_GetDIBit	按位读取扩展 IO 输入
NMC_SetDIBitRevs	按位设置扩展 IO 输入信号取反（不会改变灯的状态）
NMC_GetDIBitRevs	按位读取扩展 IO 输入信号取反的设置值
NMC_SetDOBitRevs	按位设置扩展 IO 输出信号取反（会改变灯的状态）
NMC_GetDOBitRevs	按位读取扩展 IO 输出信号取反的设置值
NMC_GetIOModuleSts	读取扩展 IO 模块的状态
NMC_IOModuleSetEn	设置扩展 IO 模块有效(带模块类型)
NMC_IOModuleGetType	读取扩展 IO 模块类型

（1）设置扩展 IO 输出(按通道,支持超过 32 位)，与 NMC_SetDO 功能相同

```
NMC_SetDOGroup( HAND devHandle, long value, short groupID);
```

参数	输入/输出	描述
devHandle	输入	控制器句柄
groupID	输入	DO 组，取值范围[0,n],0：本地 DO0~DO31， 1：本地 DO32~DO63，其他指扩展 IO 模块，根据扩展地址选择
value	输入	按位设置电平输出 默认情况下 1： 表示高电平，0： 表示低电平

（2）按位设置扩展 IO 输出

```
NMC_SetDOBit( HAND devHandle, short bitIndex, unsigned char value );
```

参数	输入/输出	描述
devHandle	输入	控制器句柄

bitIndex	输入	取值范围[0,n]位序号,前 64 位为本地的 IO 输出，大于 64 为扩展 IO 输出
value	输入	按位设置电平输出 默认情况下 1 表示高电平，0 表示低电平

（3）按位读取扩展 IO 输出

NMC_GetDOBit(HAND devHandle,short bitIndex, short *bitValue);

名称	输入/输出	描述
devHandle	输入	控制器句柄
bitIndex	输入	取值范围[0,n]位序号,前 64 位为本地的 IO 输出，大于 64 为扩展 IO 输出
bitValue	输出	按位返回输出电平的状态 默认情况下 1：表示高电平，0：表示低电平

（4）读取扩展 IO 输出(按通道,支持超过 32 位)，此函数与 NMC_GetDO 功能相同

NMC_GetDOGroup(HAND devHandle, long *pDoValue, short groupID);

参数	输入/输出	描述
devHandle	输入	控制器句柄
groupID	输入	DO 组，取值范围[0,n],0：本地 DO0~DO31， 1：本地 DO32~DO63，其他指扩展 IO 模块，根据扩展地址选择
value	输出	按组返回输出电平的状态 默认情况下 1 表示高电平，0 表示低电平

（5）读取扩展 IO 输入(按通道,支持超过 32 位)，此函数与 NMC_GetDI 功能相同

NMC_GetDIGroup(HAND devHandle, long *pInValue, short groupID);

参数	输入/输出	描述
devHandle	输入	控制器句柄
groupID	输入	DI 组，取值范围[0,n],0：本地 DI0~DI31， 1：本地 DI32~DI63，其他指扩展 IO 模块，根据扩展地址选择
pInValue	输出	按位返回输出电平的状态 1：表示高电平/开路 0：表示低电平/闭合

（6）按位读取扩展 IO 输入

NMC_GetDIBit(HAND devHandle, short bitIndex, short *bitValue);

参数	输入/输出	描述

devHandle	输入	控制器句柄
bitIndex	输入	取值范围[0,n]位序号,前 64 位为本地的 IO 输入，大于 64 为扩展 IO 输入
bitValue	输出	按位返回电平输入的状态 1：表示高电平/开路　　0：表示低电平/闭合

（7）按位设置扩展 IO 输入信号取反（不会改变灯的状态）

NMC_SetDIBitRevs (HAND devHandle,short bitIndex,short revs);

参数	输入/输出	描述
devHandle	输入	控制器句柄
bitIndex	输入	取值范围[0,n]位序号,前 64 位为本地的 IO 输入，大于 64 为扩展 IO 输入
revs	输入	是否取反 1：取反，　0：不取反

（8）按位读取扩展 IO 输入信号取反的设置值

NMC_GetDIBitRevs (HAND devHandle,short bitIndex,short *pRevs);

参数	输入/输出	描述
devHandle	输入	控制器句柄
bitIndex	输入	取值范围[0,n]位序号,前 64 位为本地的 IO 输入，大于 64 为扩展 IO 输入
pRevs	输出	是否取反　1：取反，0：不取反

（9）按位设置扩展 IO 输出信号取反（会改变灯的状态）

NMC_SetDOBitRevs(HAND devHandle,short bitIndex,short revs);

参数	输入/输出	描述
devHandle	输入	控制器句柄
bitIndex	输入	取值范围[0,n]位序号,前 64 位为本地的 IO 输出，大于 64 为扩展 IO 输出
revs	输入	是否取反　1：取反,0：不取反

（10）按位读取扩展 IO 输出信号取反的设置值

NMC_GetDOBitRevs(HAND devHandle,short bitIndex,short *pRevs);

参数	输入/输出	描述
devHandle	输入	控制器句柄

bitIndex	输入	取值范围[0,n]位序号,前 64 位为本地的 IO 输出，大于 64 为扩展 IO 输出
pRevs	输出	是否取反　1：取反,0：不取反

（11）读取扩展 IO 模块的状态

```
NMC_GetIOModuleSts(HAND devHandle, unsigned long *sts);
```

参数	输入/输出	描述
devHandle	输入	控制器句柄
sts	输出	在线：比特位为 1；离线：比特位为 0(最大支持 32 位比特位)

（12）设置扩展 IO 模块有效(带模块类型)

```
NMC_IOModuleSetEn( HAND devHandle, unsigned char chDevId,short chDevType);
```

参数	输入/输出	描述
devHandle	输入	控制器句柄
chDevId	输入	设备 ID
chDevType	输入	模块类型，见宏定义 // 扩展模块类型定义 #define IOMODULE_TYPE_IO64　1// 32DI32DO 模块（包括 16DI16DO 模块） #define IOMODULE_TYPE_IO32_DA 2// 4AD4DA 模块

（13）读取扩展 IO 模块类型

```
NMC_IOModuleGetType( HAND devHandle, unsigned char chDevId,short *pChDevType);
```

参数	输入/输出	描述
devHandle	输入	控制器句柄
chDevId	输入	设备 ID
pchDevType	输出	返回的模块类型，见宏定义 // 扩展模块类型定义 #define IOMODULE_TYPE_IO64　1// 32DI32DO 模块（包括 16DI16DO 模块） #define IOMODULE_TYPE_IO32_DA　2　　// 4AD4DA 模块

4.2 单独编程

当需要单独使用扩展模块时，请联系我司技术人员提供对应的库文件和例程，下面罗列出基

本使用函数。

函数原形	函数说明
NIO_Search	模块搜寻
NIO_OpenByID	模块搜寻
NIO_Close	模块关闭
NIO_GetDO	获取设备点位输出状态
NIO_SetDO	设置设备点位输出状态
NIO_GetDOBit	获取设备单个点位输入状态
NIO_SetDOBit	设置设备单个点位输出状态
NIO_GetDI	获取设备点位输入状态
NIO_GetDIBit	获取设备单个点位输入状态
NIO_GetADC	获取设备模拟量输入
NIO_SetDAC	设置设备模拟量输出
NIO_WriteID	修改 SIO 设备在系统中名称
NIO_ReadID	读取 SIO 设备名称
NIO_WriteDevID	修改 SIO 设备 ID 序号（用于 Modbus 通讯的设备 ID）
NIO_ReadDevID	读取 SIO 设备 ID 序号（用于 Modbus 通讯的设备 ID）
NIO_GetIOLogic	获取设备 IO 口逻辑
NIO_SetIOLogic	设置设备 IO 口逻辑
NIO_GetError	SC 设备操作过程中，最后出现的错误，以字符串的形式提供

（1）模块搜寻

NIO_Search(unsigned short *pDevNo, TDevInfo *pInfoList);

参数	输入/输出	描述
pDevNo	输出	搜索到的模块数量
pInfoList	输出	模块信息，结构体请参考头文件

（2）模块打开

NIO_OpenByID(const char *idStr, PHAND pDevHandle);

参数	输入/输出	描述

idStr	输入	模块 ID 字符串，通过搜索模块函数获得的结构体 TDevInfo 中获得
pDevHandle	输出	模块句柄

（3）模块关闭

NIO_Close(HAND devHandle);

参数	输入/输出	描述
devHandle	输入	模块句柄

（4）获取设备点位输出状态

NIO_GetDO(HAND devHandle, unsigned short portStart, unsigned short portCount, unsigned short *value);

参数	输入/输出	描述
devHandle	输入	模块句柄
portStart	输入	起始端口号，取值范围(0-1)
portCount	输入	要获取的端口数量，取值范围(1-2)
value	输出	用于存储端口状态值缓冲(每个端口 16 个点位)，每个点位:1 代表触发; 0 代表未触发

（5）设置设备点位输出状态

NIO_SetDO(HAND devHandle, unsigned short portStart, unsigned short portCount, unsigned short *value);

参数	输入/输出	描述
devHandle	输入	模块句柄
portStart	输入	起始端口号，取值范围(0-1)
portCount	输入	要设置的端口数量，取值范围(1-2)
value	输出	要设置的端口状态值缓冲(每个端口 16 个点位)，每个点位:1 代表触发; 0 代表不触发

（6）获取设备单个点位输入状态

NIO_GetDOBit(HAND devHandle, unsigned short channel, unsigned short *value);

参数	输入/输出	描述
devHandle	输入	模块句柄
channel	输入	点位位置，取值范围: 0-31

value	输出	用于存储获取的点位状态, 1 代表触发; 0 代表未触发

（7） 设置设备单个点位输出状态

NIO_SetDOBit(HAND devHandle, unsigned short channel, unsigned short value);

参数	输入/输出	描述
devHandle	输入	模块句柄
channel	输入	点位位置，取值范围: 0-31
value	输出	用于存储获取的点位状态, 1 代表触发; 0 代表未触发

（8） 获取设备点位输入状态

NIO_GetDI(HAND devHandle, unsigned short portStart, unsigned short portCount, unsigned short *value);

参数	输入/输出	描述
devHandle	输入	模块句柄
portStart	输入	起始端口号，取值范围(0-1)
portCount	输入	要获取的端口数量，取值范围(1-2)
value	输出	用于存储端口状态值缓冲(每个端口 16 个点位)，每个点位:1 代表触发; 0 代表未触发

（9） 获取设备单个点位输入状态

NIO_GetDIBit(HAND devHandle, unsigned short channel, unsigned short *value);

参数	输入/输出	描述
devHandle	输入	模块句柄
channel	输入	点位位置，取值范围: 0-31
value	输出	用于存储获取的点位状态, 1 代表触发; 0 代表未触发

（10） 获取设备模拟量输入

NIO_GetADC(HAND devHandle, unsigned short adcStart, unsigned short adcCount, unsigned short *value);

参数	输入/输出	描述
devHandle	输入	模块句柄
adcStart	输入	起始通道号，取值范围(0-3)
adcCount	输出	要获取的通道数量，取值范围(1-4)
value	输入	用于存储模拟量输入值缓冲

（11） 设置设备模拟量输出

NIO_SetDAC(HAND devHandle, unsigned short dacStart, unsigned short dacCount, const unsigned short *value);

参数	输入/输出	描述
devHandle	输入	模块句柄
dacStart	输入	起始通道号，取值范围(0-3)
dacCount	输出	要获取的通道数量，取值范围(1-4)
value	输入	要设置的端口模拟量输出值缓冲

（12） 修改 SIO 设备在系统中名称

NIO_WriteID(HAND devHandle, const char *idStr);

参数	输入/输出	描述
devHandle	输入	模块句柄
idStr	输入	新的设备名称，最长 16 字节包括末尾'\0'字符

（13） 读取 SIO 设备名称

NIO_ReadID(HAND devHandle, char *productId);

参数	输入/输出	描述
devHandle	输入	模块句柄
productId	输入	用于存放设备名称缓冲区，最长 16 字节

（14） 修改 SIO 设备 ID 序号（用于 Modbus 通讯的设备 ID）

NIO_WriteDevID(HAND devHandle, const unsigned char devId);

参数	输入/输出	描述
devHandle	输入	模块句柄
devId	输入	新的设备序号

（15） 读取 SIO 设备 ID 序号（用于 Modbus 通讯的设备 ID）

NIO_ReadDevID(HAND devHandle, unsigned char *devId);

参数	输入/输出	描述
devHandle	输入	模块句柄
devId	输出	用于存放设备 ID 序号

（16） 获取设备 IO 口逻辑

NIO_GetIOLogic(HAND devHandle, unsigned short *value);

参数	输入/输出	描述
devHandle	输入	模块句柄
value	输出	用于存储 IO 逻辑状态

（17） 设置设备 IO 口逻辑

NIO_SetIOLogic(HAND devHandle, unsigned short value);

参数	输入/输出	描述
devHandle	输入	模块句柄
value	输入	要设置的 IO 逻辑状态

（18） SC 设备操作过程中，最后出现的错误，以字符串的形式提供

NIO_GetError(HAND devHandle, char *str, unsigned short len);

参数	输入/输出	描述
devHandle	输入	模块句柄
str	输入	用于存储错误信息的缓冲
len	输出	缓冲区大小

4.3 应用案例

（1）挂接控制器操作

```
/*********此处省略控制器初始化部分**********/

    short rtn = 0;

    HAND devHandle = 0;

    short rtn = 0;

    long divaule = 0;

    long divalueex = 0;

    long dovalue = 0;

    long dovalueex = 0;

    short di0 = 0;

    short di64 = 0;
```

```
long axis1 = 0;
bool IsExmoNeed = false;
//如果需要读取扩展模块
if (IsExmoNeed==true){
//设备 ID 需要从 2 开始，多个扩展模块，依次增加，拨码开关也要拨码成对应 ID
    rtn = NMC_IOModuleSetEn(devHandle ,2,1);
}
//读取一组本地 IO 数字量输入
rtn = NMC_GetDIGroup(devHandle ,&divaule,0);
//按位读取（单个读取）,这里示范读取第 0 个输入
rtn = NMC_GetDIBit(devHandle ,0,&di0);
//读取一组扩展模块 IO 的数字量输入，这里示范读取地址为 2 的扩展模块 DI
rtn = NMC_GetDIGroup(devHandle ,&divalueex,2);
//按位读取（单个读取），这里示范读取第 0 个输入（地址为 2 的扩展模块 DI）
rtn = NMC_GetDIBit(devHandle ,64,&di64);
//读取本地一组 IO 的数字量输出
rtn = NMC_GetDOGroup(devHandle ,&dovalue,0);
//读取一组扩展模块 IO 的数字量输出（地址为 2 的扩展模块 DO）
rtn = NMC_GetDOGroup(devHandle ,&dovalueex,2);
//设置扩展 IO 数字量输出
//全部输出
rtn = NMC_SetDOGroup(devHandle ,0,0);
//第 1 个输出
rtn = NMC_SetDOGroup(devHandle ,0xFFFD,0);
//第 0 个输出,按位输出
rtn = NMC_SetDOBit(devHandle ,0,0);
return rtn;
```

（2）单独使用操作(C#)

```
/*********using 引用部分自行引入，rtn 返回值自行判断**********/
short rtn = 0;
ushort devNum = 0;
private IntPtr devHandle;
TDevInfo devInfo=new TDevInfo();
rtn = NIO_Search(ref devNum, ref devInfo); // 搜索扩展模块
rtn = NIO_OpenByID("CARD1", ref devHandle);// 打开扩展模块，默认设备名称：CARD1
ushort input = 0;
rtn = NIO_GetDI(devHandle, 0, 1, ref input); // 获取输入状态,0 代表有输入
ushort diVal = 0;
rtn = NIO_GetDIBit(devHandle1, 2, diVal); //按位获取输入
ushort output = 0;
rtn = NIO_GetDO(devHandle, 0, 1, ref output); //获取输出状态，0代表有输出
ushort output = 65534; // 按位设置输出  关闭其他输出
rtn = NIO_SetDO(devHandle, 0, 1, ref output);
rtn = NIO_SetDOBit(devHandle, 1, 0); // 按位设置输出
ushort pv = 0;
rtn = NIO_GetIOLogic(devHandle, ref pv); // 按位获取输出
```