MPS-HD1802

USB 二通道 18 位高动态范围

信号采集卡使用说明

Ver 2.0.0

第一章 产品概述

MPS-HD1802 二通道 18 位高动态范围信号采集卡

一、 *产品简介*

MPS-HD1802 是一款基于 USB 总线的双通道 18 位高动态范围信号采集卡，适用于高精度信号测量、便携式数据采集、振动与音频信号分析等领域。

MPS-HD1802 具有两路±5V 量程的同步电压采集通道，采样率在 1kSPS 至 64kSPS 范围内共七档可调，支持连续不间断的信号采集与实时数据传输。MPS-HD1802 具有低噪声的特点，在 64k 采样率时本底噪声可低至 42μVrms(±5V 满量程)，在 1k 采样率时更可低至 8μVrms，有效分辨率可达 19.5 位。

MPS-HD1802 能够提供六种可选的通道输入模式，兼容差分/单端、高输入阻抗/低输入阻抗、直流耦合/交流耦合、IEPE（ICP）恒流供电等多种输入特性，可以适配大部分应用场景，并且支持每个通道定制不同的输入模式，在多类型信号的混合测量中尤其适用。

MPS-HD1802 通过 USB 总线与计算机连接，支持即插即用和热插拔。设备兼容全系列 Windows 操作系统，提供通用的 DLL 驱动函数库，并提供 LabVIEW、VB、C#、Python、Qt、MATLAB 等多种语言下的编程参考示例。

此外，MPS-HD1802 还免费赠送一套多功能采集软件，可直接实现波形显示、信号记录、波形回放、频谱分析等基础功能，以及软件滤波、边沿触发、计算频率、重采样、信号拟合、声音播放等众多高级辅助功能，能够大幅降低使用门槛，帮助用户实现快速测量。

二、 性能指标

2.1、通信总线

· USB2.0 高速总线（兼容 3.0 及更高版本）
· USB 总线供电
· 支持即插即用与热插拔

2.2、输入通道

· 接口类型：BNC 母头
· CH1-CH2 通道：二路信号输入口
· GND/IO 通道的负极：共地线接口
· GND/IO 通道的正极：多功能复用 IO 口

2.3、输入模式

MPS-HD 系列采集卡支持六种信号输入模式，用户可在订购产品时指定每个通道的输入模式。六种输入模式分别为：

S1 模式（直流单端低输入阻抗模式）：
1、 耦合方式：直流耦合
2、 输入类型：单端输入
3、 输入阻抗：输入正极与地线间 50kΩ【1】；输入负极与地线连通
4、 输入量程：±5V【2】
5、 输入耐压：±25V
6、 恒流供电：无

S2 模式（直流单端高输入阻抗模式）：
1、 耦合方式：直流耦合
2、 输入类型：单端输入
3、 输入阻抗：输入正极与地线间 1MΩ【3】；输入负极与地线连通
4、 输入量程：±5V
5、 输入耐压：±25V
6、 恒流供电：无

S3 模式（直流差分模式）：
1、 耦合方式：直流耦合
2、 输入类型：差分输入
3、 输入阻抗：输入正极与地线间 1MΩ；输入负极与地线间 1MΩ
4、 输入量程：±5V
5、 输入耐压：±25V
6、 恒流供电：无

S4 模式（交流单端模式）：
1、 耦合方式：交流耦合
2、 输入类型：单端输入
3、 输入阻抗：输入正极与地线间 10μF||50kΩ；输入负极与地线连通
4、 输入量程：±5V

5、 输入耐压：±25V
6、 恒流供电：无

S5 模式（恒流供电模式）：
1、 耦合方式：直流耦合
2、 输入类型：单端输入
3、 输入阻抗：输入正极与地线间 1MΩ；输入负极与地线连通
4、 输入量程：±5V
5、 输入耐压：±25V
6、 恒流供电：恒定 4mA【4】；开路驱动电压 24V

S6 模式（IEPE 模式）：
1、 耦合方式：交流耦合
2、 输入类型：单端输入
3、 输入阻抗：输入正极与地线间 10μF||50kΩ；输入负极与地线连通
4、 输入量程：±5V
5、 输入耐压：±25V
6、 恒流供电：恒定 4mA；开路驱动电压 24V

【1】*输入内阻的精确额定值为 49.7kΩ，误差 0.1%。*
【2】*量程的精确额定值为 5.3V，误差 0.1%。*
【3】*输入内阻的精确额定值为 1MΩ，误差 0.1%。*
【4】*恒流供电的精确额定值为 4.15mA，误差 1%。*

2.4、采样率

- 64k、32k、16k、8k、4k、2k、1k(SPS)七档可通过软件设置。

2.5、分辨率

- 64k 采样率下，有效分辨率 17.2bit；采样噪声峰峰值 V_{pp}<280μV，有效值 V_{RMS}<42μV；
- 32k 采样率下，有效分辨率 17.6bit；采样噪声峰峰值 V_{pp}<212μV，有效值 V_{RMS}<32μV；
- 16k 采样率下，有效分辨率 18.1bit；采样噪声峰峰值 V_{pp}<146μV，有效值 V_{RMS}<22μV；
- 8k 采样率下，有效分辨率 18.5bit；采样噪声峰峰值 V_{pp}< 113μV，有效值 V_{RMS}<17μV；
- 4k 采样率下，有效分辨率 18.8bit；采样噪声峰峰值 V_{pp}<86μV 伏，有效值 V_{RMS}<13μV；
- 2k 采样率下，有效分辨率 19.2bit；采样噪声峰峰值 V_{pp}<67μV，有效值 V_{RMS}< 10μV；
- 1k 采样率下，有效分辨率 19.5bit；采样噪声峰峰值 V_{pp}< 53μV，有效值 V_{RMS}< 8μV；

2.6、带宽

- 1k - 64k 采样率下，内置 32kHz 抗混叠滤波，信号通带高截止频率为 31.2kHz;
- S4 或 S6 模式下，内置隔直高通滤波器，信号通带低截止频率为 0.3Hz；
- S1、S2、S3 或 S5 模式下，直流耦合，信号通带低截止频率 0Hz；

2.7、板载缓存

- DAQ Buffer：192k Bytes
- USB FIFO ：1k Bytes

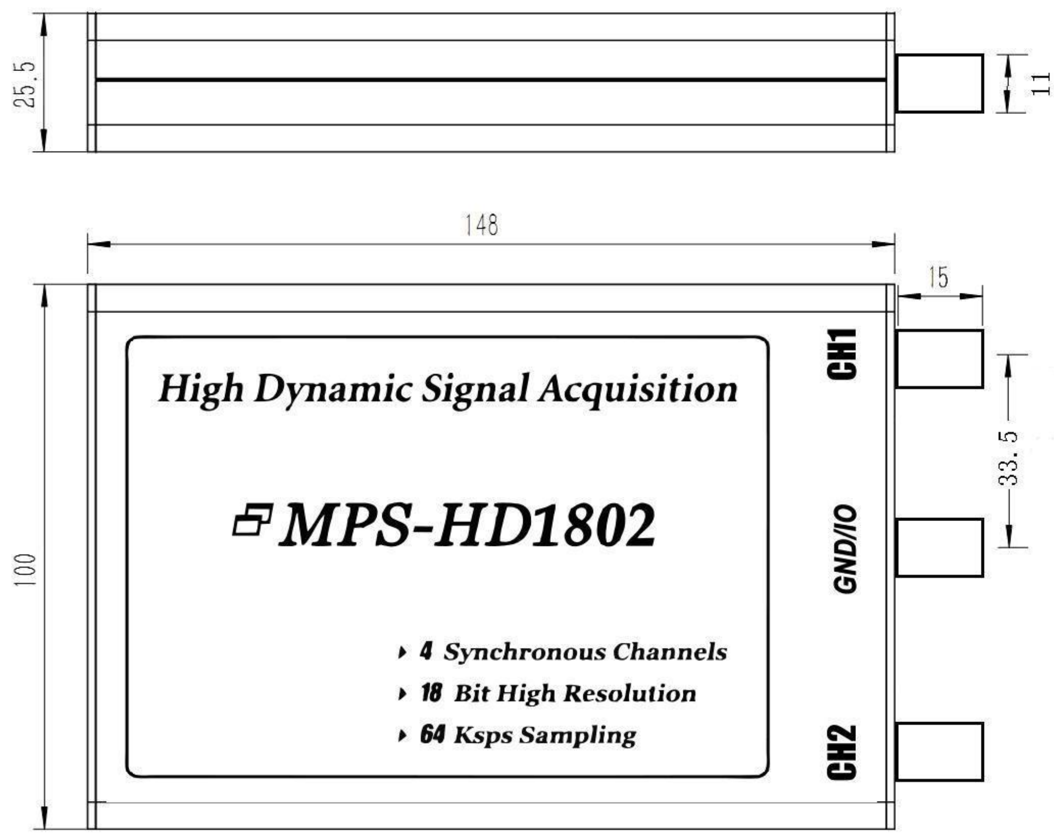
2.8、工作温度

- -40℃～85℃

2.9、工作功率

- 待机功率：电流 < 220mA，功率 < 1.1W;
- 运行功率：电流 < 260mA，功率 < 1.3W;
- IEPE 功率：每接入一只 IEPE 传感器，电流增加 25mA，功率增加 0.125W;
- 最大功率：电流 < 340mA，功率< 1.7W;
- DIO 5V 输出：输出电流 < 50mA;

三、 外观尺寸



外观尺寸：163mm * 100mm * 25.5mm

四、 应用领域

高精度信号采集与记录

工业生产在线监测

便携式信号测量

声音与振动分析

五、 软件资源

提供兼容全系列Windows操作系统的专用驱动程序;提供跨平台的通用DLL动态链接库，用于系统集成与二次开发；提供LabVIEW、VB、C#、Python、Qt、Matlab等多种编程语言的例程；附送一套多功能测试软件。

六、 配件清单

[1] MPS-HD1802信号采集卡一张；

[2] 高屏蔽USB数据传输线一根；

[3] 保修卡一张；

[4] 合格证一张

七、 售后服务

免费保修三年。

第二章 设备安装

一、 *接口说明*

USB： USB 数据总线接口。该接口通过 USB Type-B 数据线与计算机相连。连接成功后，设备会被 windows 系统识别，并在设备管理器中列出。USB 总线同时为设备提供供电，通常情况下，USB 总线自身的供电即可满足设备需求，若 USB 总线供电不足，可考虑使用带外部供电口的 USB 集线器来提供额外供电。

LED1： 设备自检状态指示灯（LED1），亮起时为蓝色，具体状态含义如下：

- 上电后常亮：设备自检通过，运行正常。
- 上电后快速闪烁：设备自检异常。
- 上电后熄灭：设备供电不稳定或存在硬件故障，建议尝试通过更换计算机主机与 USB 数据线的方式来解决。若无效，请及时与售后部门联系。

LED2： 采集与数据传输状态指示灯（LED2），亮起时为白色，具体含义如下：

- 常亮：设备正处于连续采集状态，且数据传输稳定。
- 熄灭：若设备处于待机状态，LED2 熄灭为正常情况；若设备处于采集状态，LED2 亮起后又出现熄灭，可能是软件未及时读取数据，导致硬件数据缓冲区写满后出现数据覆写，LED2 熄灭表明硬件缓冲区处于覆写状态。
- 规律闪烁：通常是由于采集软件采取循环分段采集的模式形成的，即软件每次采集时，LED2 亮起；软件停止采集时，LED2 熄灭；随着上述过程循环往复，LED2 会呈现有规律的亮灭或闪烁。
- 无规律闪烁：若在软件连续采集的情况下，LED2 出现无规律的闪烁，可能为数据传输不稳定的警示。假如软件读取数据的速度过慢或者由于软件运行卡顿，在较长时间内没从硬件中读数，硬件的数据缓冲区会出现覆写，此时 LED2 会随之熄灭；此后若软件恢复数据读取，LED2 则会重新亮起。假如上述过程重复出现，会使 LED 呈现无规律闪烁的现象。当此现象出现时，说明当前计算机软件运行存在问题，已引发数据丢失，导致数据完整性被破坏。此时请尝试优化软件执行效率来提高读取速度，或是通过降低采样率来减少数据总量，以保障读取数据的完整性。

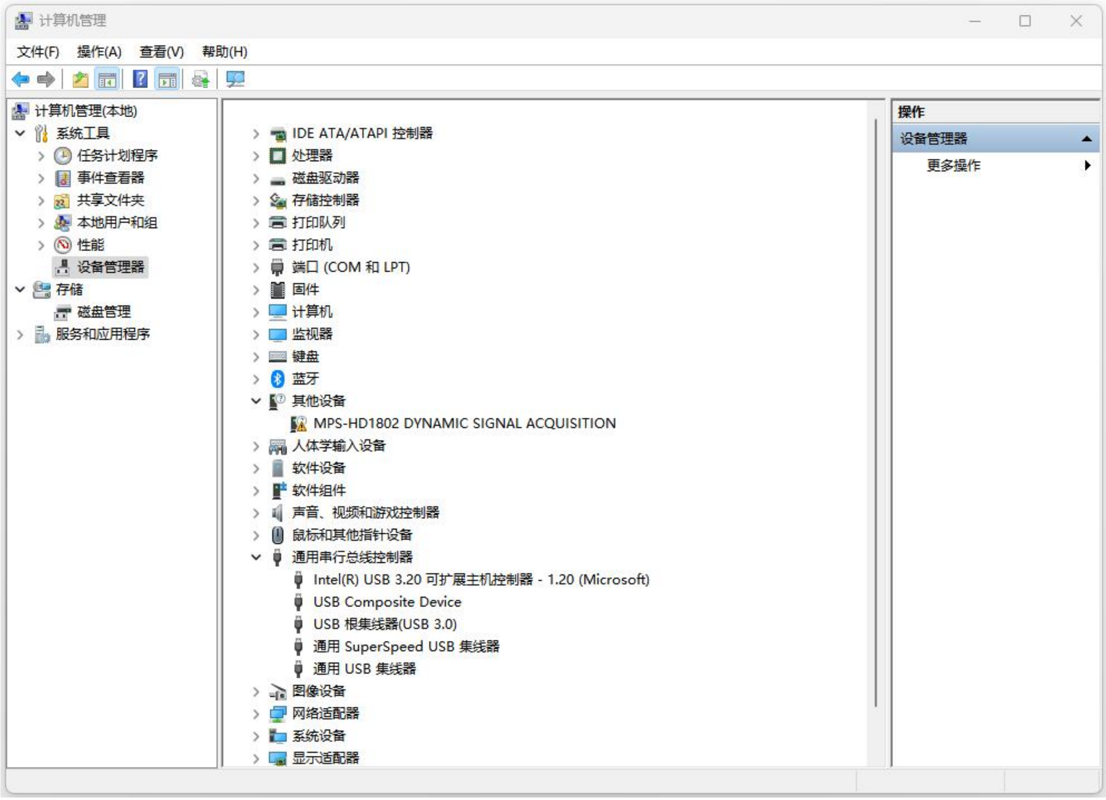
CHx： CH1 与 CH2 分别对应通道 1 和通道 2 的信号输入端。每个通道为 BNC 母座接头，其外圈为负极，内芯为正极。不同输入模式下的接口特性（如阻抗、耦合方式等），请参阅本手册第一章第 2.3 节。

GND/IO：设备信号地与复用 IO 口。该端口为 BNC 母座接头，其外圈为设备信号地，内芯为复用 IO 口，功能如下：

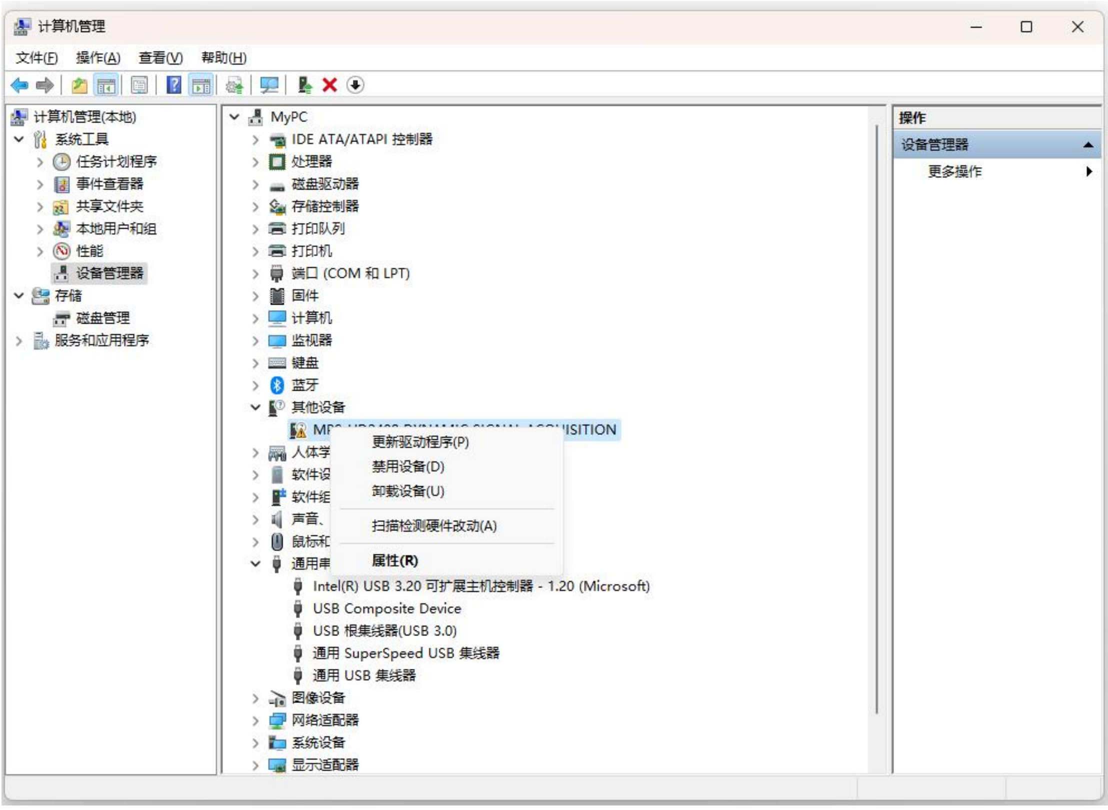
- 信号地：用于连接外部传感器或信号源的地线，建立共地连接点。
- 复用 IO 口：
  - 数字输出：可配置为输出高电平（5V）或低电平。当输出高电平时也可作为 5V 电源使用，最大输出电流为 50mA。
  - 触发输入：可作为外部启动触发信号的输入口。对该口输入一个下降沿信号，可触发设备开始采集。

二、 驱动安装

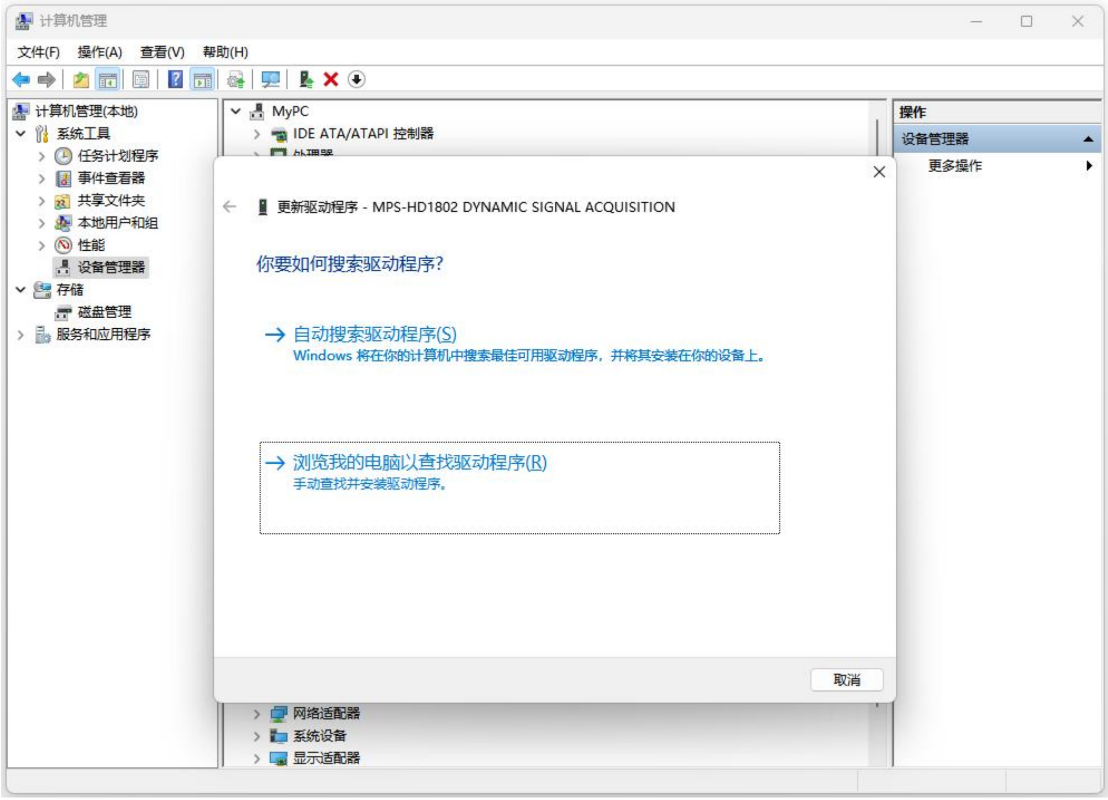
1. 设备接入计算机 USB 后，在“此电脑”上点右键，选择“设备管理器”或者点击“属性”－“管理”－“设备管理器”来打开 WINDOWS 设备管理器，并在“其他设备”下找到本设备。



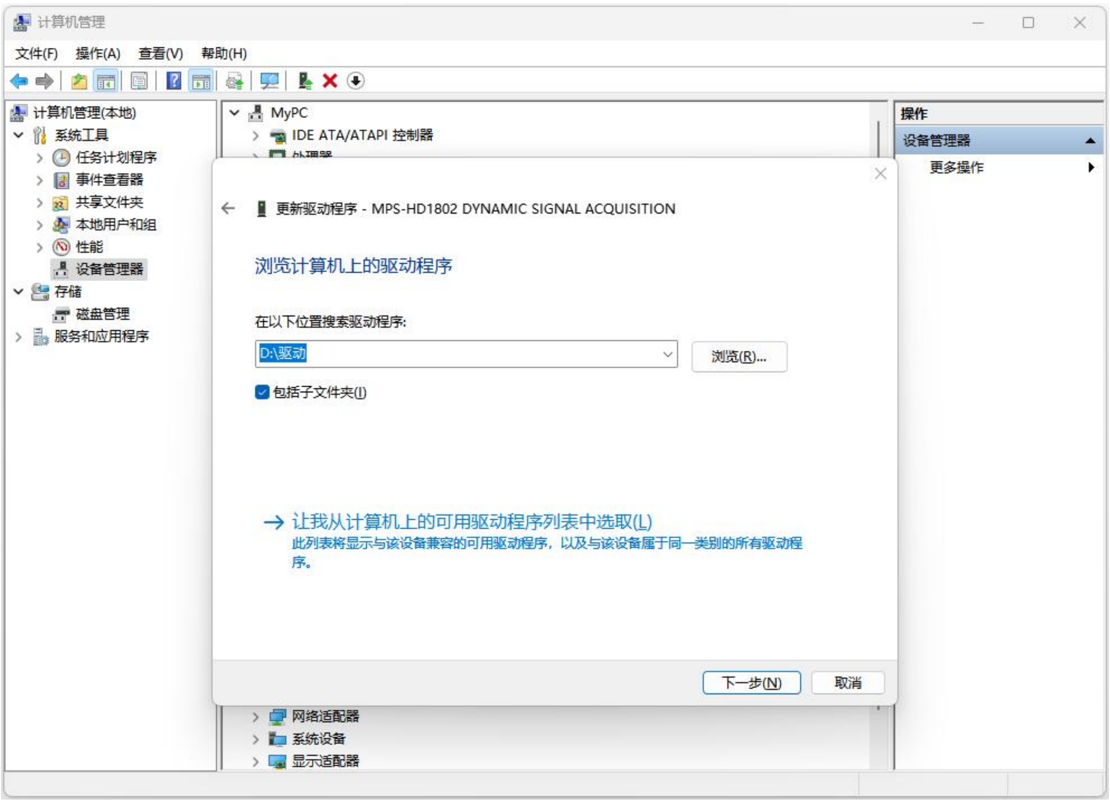
2. 在设备名称上点右键，选择“更新驱动程序”。



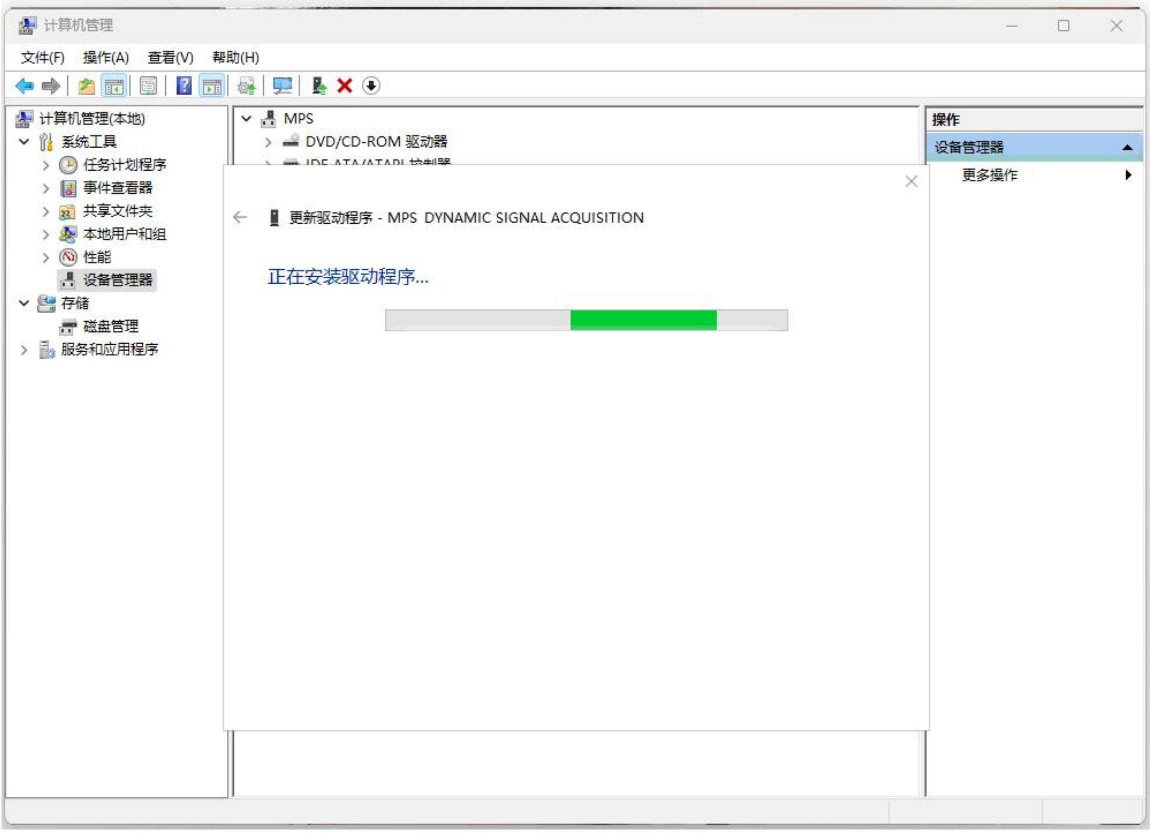
3.　在弹出的驱动安装向导界面中，选择“浏览我的电脑以查找驱动程序”。



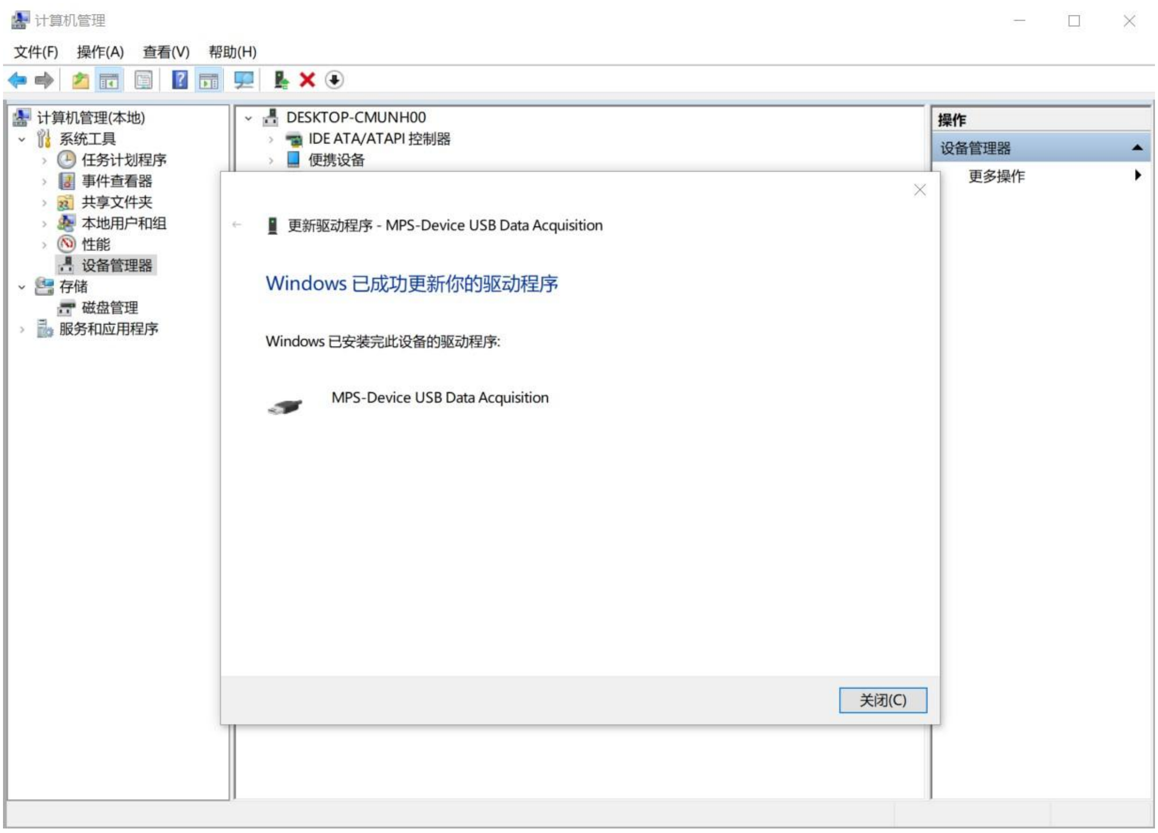
4.　点击“浏览”按钮，选择采集卡驱动所在的文件夹。如果驱动为压缩包，需要先解压缩后再进行操作。



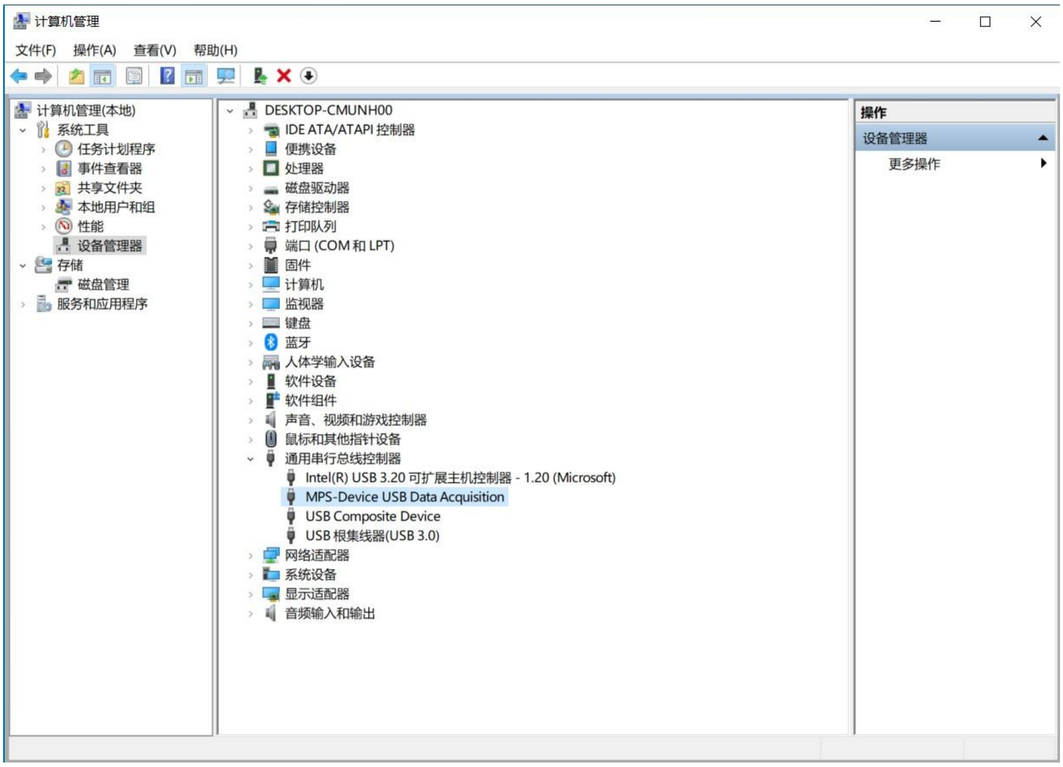
5. 点击下一步，出现如下驱动安装界面。过程中如果出现信任驱动的安全提示，勾选信任选项并点击确认。



6. 驱动安装完成，出现如下提示框。



7. 驱动安装后，可在设备管理器 “通用串行总线控制器”类目下看到设备，名为“MPS-Device USB Data Acquisition”，驱动成功安装。



三、 *信号接入*

MPS-HD1802 有两个模拟信号输入口 CH1 和 CH2，分别对应通道 1 与通道 2。输入信号接头为 BNC 母头，可与使用 BNC 公头的信号线进行对接。

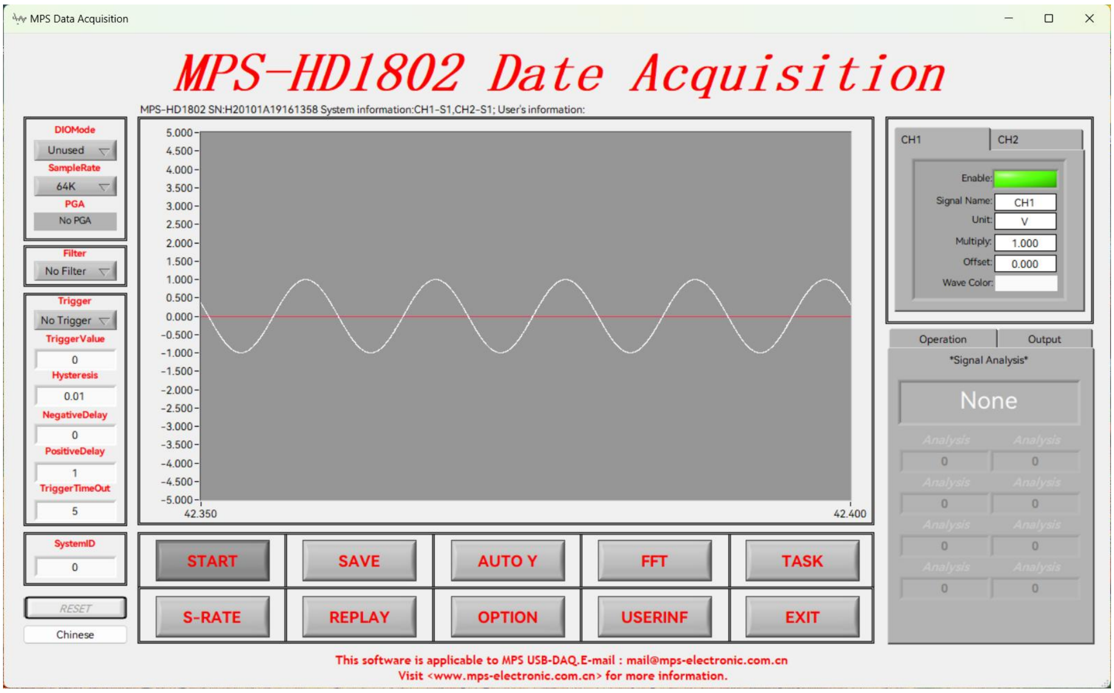
MPS-HD1802 在交付前可对每个通道进行输入模式配置，共六种模式可选（见第一章 2.3 小节），不同的输入模式可匹配不同类型的信号源。交付后如果需要更改模式，可与售后部门联系咨询。

若 MPS-HD1802 的某个通道为 S1、S2 或 S4 模式，该通道为单端输入，输入接头的正极是信号输入，负极是信号共地线。该通道可与外部的单端信号源连接，接头的正极对接单端信号源的信号线，接头的负极对接单端信号源的信号地。

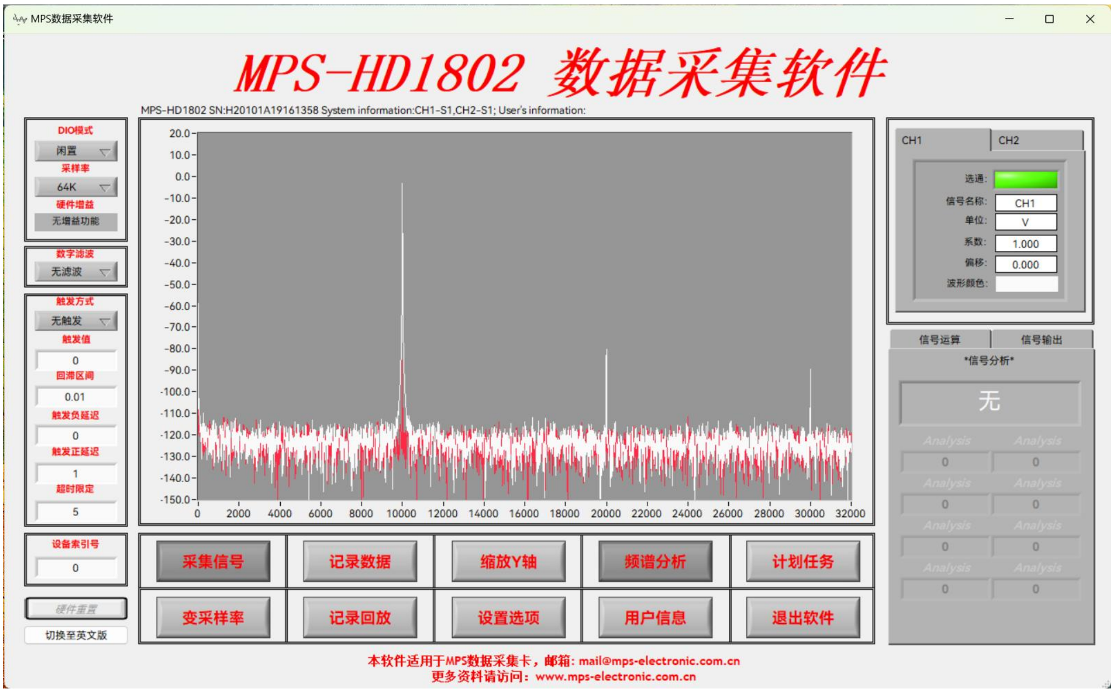
若 MPS-HD1802 的某个通道为 S3 模式，该通道为差分输入，需要与设备的 GND/IO 口负极（信号共地线）联合使用，既可以连接外部的差分信号源，也可以连接单端信号源。连接差分信号源时，差分通道输入接头的正、负极分别对接差分信号源的正、负极，同时 GND/IO 口的负极与差分信号源的独立地线对接进行共地；连接单端信号源时，通道输入正极与单端信号源的信号线连接，通道输入负极与单端信号源的地线连接，同时 GND/IO 口的负极也要与信号源的地线连接以进行共地。

若 MPS-HD1802 的某个通道为 S5 或 S6 模式，该通道为单端输入，输入接头的正极是信号输入，同时也会提供 4mA 的对外恒流输出；接头的负极是信号共地线。该通道可与外部的 IEPE 传感器（S6 模式）、阻值 1250 欧姆以内的电阻型传感器（S5 模式）和某些特定的传感器进行连接。其他类型的传感器一般不能接入，强行接入可能导致传感器无法正常工作，并有损坏传感器的风险。当使用 S6 模式时，通道可以直接对接 IEPE 型传感器，中间不需要额外的信号调理器。

四、 功能测试



MPS 测试软件英文界面波形图



MPS 测试软件中文界面频谱图

1. 下载并解压 MPS Data Acquisition Installer.zip，执行“Setup.exe”安装测试软件。
2. 将 MPS-HD1802 信号采集卡与计算机通过 USB 接口连接，首次接入需要按第 3 小节的步骤来安装硬件驱动。
3. 打开 Windows 的开始菜单，打开名为“MPS Data Acquisition”（ 图标为）的软件，软件界面如上图所示。

4. 左上角“SampleRate”处可设置采样率，从 1k 到 64k 七档可选。
5. 点击“START”，可从波形图中看到所采集的信号波形曲线，未连接信号源时，曲线一般呈现为一条接近于 0 的直线。点击“START”后如果出现错误提示，请检查硬件是否接入，以及驱动是否正确安装。
6. 将传感器或信号源接入采集卡后，可观察到随信号源变化的波形曲线。其中白色曲线对应 CH1，红色曲线对应 CH2。
7. 点击“AUTO Y”可使 Y 轴显示范围自动与当前曲线匹配，直接修改 Y 轴坐标边界值也可以改变显示范围。
8. 软件中数字滤波、软件触发、信号记录、频谱分析、计划任务、变采样率、记录回放、信号分析、运算、输出等附加功能都可供选用。点击左下角“Chinese”可以切换为中文软件界面。
9. 功能测试结束，点击“EXIT”退出软件。
10. 断开信号源，并拔出连接计算机的 USB 插头，测试完成。

第三章 用户编程

一、 *动态链接库（DLL）*

MPS-HD 系列采用 DLL（Dynamic Linkable Library，动态链接库)的方式来进行编程驱动。DLL 与具体的编程语言及编译器无强制关联，只要遵循约定的 DLL 接口规范和调用方式，用各种语言都可以调用 DLL。

以 VC、VB、LabVIEW 等语言下调用 DLL 为例，具体调用方式分别为：

- VC 下调用 DLL

```
typedef void ( * FUNC )(void);                    //定义一个函数指针
FUNC Func;                                        //定义一个函数指针变量
HINSTANCE hDLL=LoadLibrary("DllTest.dll");        //加载 dll
Func=(FUNC)GetProcAddress(hDLL,"FuncInDLL");      //找到 dll 中的函数
Func();                                           //调用 dll 里的函数
```

- VB 下调用 DLL

[Public | Private] Declare Function name Lib " labname " [Alias " aliasname "] [([arglist])] [As type] "

Public（可选）用于声明在所有模块中的所有过程都可以使用的函数；Private（可选）用于声明只能在包含该声明的模块中使用的函数。

Name（必选）任何合法的函数名。动态链接库的入口处（entry points）区分大小写。

Libname（必选）包含所声明的函数动态链接库名或代码资源名。

Alias（可选）表示将被调用的函数在动态链接库 (DLL) 中还有另外的名称。当外部函数名与某个函数重名时，就可以使用这个参数。当动态链接库的函数与同一范围内的公用变量、常数或任何其它过程的名称相同时，也可以使用 Alias。如果该动态链接库函数中的某个字符不符合动态链接库的命名约定时，也可以使用 Alias。

Aliasname（可选）动态链接库。如果首字符不是数字符号 (#)，则 aliasname 是动态链接库中该函数入口处的名称。如果首字符是 (#)，则随后的字符必须指定该函数入口处的顺序号。

Arglist（可选）代表调用该函数时需要传递参数的变量表。

Type（可选）Function 返回值的数据类型；可以是 Byte、Boolean、Integer、Long、Currency、Single、Double、Decimal（目前尚不支持)、Date、String（只支持变长）或 Variant，用户定义类型，或对象类型。

arglist 参数的语法如下：

[Optional] [ByVal | ByRef] [ParaArray] varname [()] [As type]

Optional（可选）表示参数不是必需的。如果使用该选项，则 arglist 中的后续参数都必须是可选的，而且必须都使用 Optional 关键字声明。如果使用了 ParamArray，则任何参数都不能使用 Optional。

ByVal（可选）表示该参数按值传递。

ByRef（可选）表示该参数按地址传递。

- LabVIEW 下调用 DLL

在 LabVIEW 中，调用 DLL 是通过 CLF 节点来完成的。所谓 CLF 节点（Call Library Function，调用函数库节点)，是指可以在 LabVIEW 调用其他语言封装的 DLL，CLF 节点位于 LabVIEW 功能模板中的 Advanced 子模板中，其配置过程如下：

- 在 CLF 节点的右键菜单中选择“Configure”，弹出 CLF 节点配置对话框；
- 点击“Browse”按钮，在随后弹出的选择 DLL 文件对话框中找到你需要用的 DLL 文件，此时，LabVIEW 就会自动装载选定的 DLL 文件，并检测 DLL 文件中所包含函数。但是函数中的参数和参数的数据类型需要用户根据函数的输入、输出参数手动设置。因而在调用 DLL 文件时， 要求用户对 DLL 文件有较为详细的了解。
- 在 FunctionName 下拉列表框中选定动态链接库中所包含的所需要 API 函数；
- 在 Calling Convention 下拉菜单中选择 StdCall(WINAPI)和 C 两个选项， 若用户选定的是 Windows API 函数，则选用 StdCall(WINAPI)选项;若用户选用的 DLL 中的函数是非 Windows API 函数， 则选用 C 选项；
- 设置函数的返回参数。函数参数的类型要与 DLL 中函数本身所定义的函数参数类型相对应， 如果不对应， 函数就会出现数据错误和强制类型转换；
- 根据所选函数的函数原型，设置函数的输入参数及数据类型。点击 Add a Parameter 按钮，即可以添加一个新的输入参数。

➢ ***C#、VB、LabVIEW、Python、Qt、Matlab 等各种语言下的完整代码，请参考官网示例。***

二、 *驱动函数及参数*

MPS-HD 系列采集卡的驱动程序文件为 **MPS Driver.dll**，其中包含以下 11 个函数：

#define HANDLE int
注：若未预先定义 HANDLE 数据类型，可将其定义为 int 类型。

- HANDLE MPS_OpenDevice (int DeviceNumber)

HANDLE MPS_OpenDevice 函数用于打开设备并获取设备的操作句柄。函数执行成功后返回设备的操作句柄，执行失败返回-1。

int DeviceNumber：此参数为待打开设备的序号，有效范围为 0 至 9。

MPS-Device 驱动支持单台计算机最多连接 10 台设备。在多个设备同时连接时，Windows 系统将自动分配设备序号(DeviceNumber)，首个被系统识别的设备分配序号 0，后续设备按识别顺序依次分配序号 1、2、3...最大分配序号为 9。该序号分配基于设备识别顺序，与物理连接端口无关。

在调用 MPS_OpenDevice 函数时，通过指定不同的 DeviceNumber 参数值，可获取对应设备的操作句柄。
对于首个接入的设备，传递参数值 0 可获取其设备句柄：

```
HANDLE hDevice0 = MPS_OpenDevice(0);  // 获取第一个设备的句柄
```

当第二个设备接入时，传递参数值 1 可获取该设备句柄：

```
HANDLE hDevice1 = MPS_OpenDevice(1);  // 获取第二个设备的句柄
```

……
第十个设备接入时，传递参数值 9。

- int MPS_Configure(int SampleRate, int DIOMode, HANDLE DeviceHandle)

int MPS_Configure 函数用于设置设备的采集参数。函数执行成功后，将对设备的采样率和 IO 口模式进行设置。此函数必须在设备处于待机状态时调用。函数执行成功返回 1，失败返回 0。

int SampleRate：此参数用于设置采样率，MPS-HD1802 共有七档采样率可设置，设置规则如下：
该值为 64000 或大于 64000 时，采样率设为每秒 64000 点；
该值为 32000 或小于 64000 且大于 32000 时，采样率设为每秒 32000 点；
该值为 16000 或小于 32000 且大于 16000 时，采样率设为每秒 16000 点；
该值为 8000 或小于 16000 且大于 8000 时，采样率设为每秒 8000 点；
该值为 4000 或小于 8000 且大于 4000 时，采样率设为每秒 4000 点；
该值为 2000 或小于 4000 且大于 2000 时，采样率设为每秒 2000 点；
该值为 1000 或小于 2000 时，采样率设为每秒 1000 点。

int DIOMode：此参数用于设置 IO 口的模式，其取值为 0、1 或 2。

DIOMode = 0:将 IO 口设置为闲置模式,使 IO 口呈现高内阻的低电平状态(约 0.5V)。

DIOMode = 1:将 IO 口设置为高电平输出模式,对外输出 5V 电压(输出电流 ＜50mA),可以作为数字高电平输出使用，也可作为对外的 5V 供电输出来使用。

DIOMode = 2：将 IO 口设置为外部启动触发模式，此模式下 IO 口处于内部带弱上拉的数字高电平输入状态。在外部启动触发模式下，可以通过向 IO 口输入一个电平下降沿来触发设备启动采集。当同时使用多个 MPS-HD 设备时，此模式也可用于实现多个设备的同步启动。

三种模式中，闲置模式和高电平输出模式在设置后，只要不重新设置，模式状态会一直保持；外部启动触发模式在设置后，仅在第一次待机状态中有效，启动采集后 IO 口将自动切换为闲置模式。

关于外部启动触发模式的补充说明：

1、外部启动触发模式下，当 IO 口的输入状态从高电平（2V-5V）转为低电平（0V-1.5V）时（例如将 IO 口与 GND 进行短接），将产生一个下降沿输入，此时若设备处于空闲状态，该下降沿会触发设备进入采集状态。

2、通过下降沿触发启动采集的方式，与上位机调用 MPS_Start 函数启动采集的效果等效，二者均会使设备进入采集状态。

3、在外部启动触发模式下，若通过调用 MPS_Start 函数启动采集，当设备从待机状态切换为采集状态时，IO 口会从外部启动模式（带弱上拉的数字高电平状态）自动变为闲置模式（低电平状态），进而产生一个下降沿。若将该设备的 IO 口与其他（一个或多个）处于外部启动触发模式的设备 IO 口连接，此下降沿会触发其他设备同步启动采集。

4、在程序设计中，若采用外部触发方式启动采集，由于软件无法实时检测外部触发的时间点，必须提前调用 MPS_DataIn 函数开始读取数据，并依赖其内部等待机制判断触发是否成功。

（1）若触发成功，MPS_DataIn 函数将在启动采集后获取数据并返回；

（2）若未触发但设置了 MPS_TimeOut 超时参数，函数将在超时后返回失败值；

（3）若未触发且未设置超时，函数会持续等待，直至调用 MPS_Start 启动采集；

（4）若等待期间出现硬件连接断开或重启等情况，函数将返回失败值。

5、在外部启动触发模式下，若设备接收到外部下降沿或 MPS_Start 函数命令，会启动采集并将 IO 口切换为闲置模式；若设备接收到 MPS_Stop 函数命令，会保持待机状态，但 IO 口仍会从外部启动模式变为闲置模式；若设备接收到 MPS_Configure 命令，会立即将 IO 口设置为新指定模式。无论上述何种情况，IO 口退出外部启动触发模式后均不再响应外部下降沿信号。

HANDLE DeviceHandle：目标设备的操作句柄。

- int MPS_Start (HANDLE DeviceHandle)

int MPS_Start 函数用于启动设备的数据采集。设备上电后，默认进入待机状态，软件可以通过调用 MPS_Start 函数的方式来启动设备采集。函数执行成功返回 1，失败返回 0。

HANDLE DeviceHandle：目标设备的操作句柄。

- int MPS_DataIn(int *DataBuffer, int SampleNumber , HANDLE DeviceHandle)

int MPS_DataIn 函数用于读取设备采集的样点数据。函数执行成功后，将从设备中读取一批样点数据。函数执行成功返回 1，失败返回 0。

设备对输入信号采样时，样点数据会暂存到内部的硬件缓冲（DAQ Buffer）中，等待 MPS_DataIn 函数读取。硬件缓冲遵循先进先出（FIFO）原则，新样点存入缓冲末尾，MPS_DataIn 函数从缓冲开头读取。数据读出后自动回收缓冲空间，后续数据前移，确保同一采集过程中多次调用 MPS_DataIn 函数获得的数据连续无间断。通过循环调用 MPS_DataIn 函数并拼接数据，可构建完整的连续数据流。

MPS_DataIn 函数的执行逻辑如下：

- ◆ 若函数运行时，设备处于采集状态，且缓冲中存在足够样点，则函数立即读取数据并返回成功；
- ◆ 若函数运行时，设备处于采集状态，但缓冲中样点数量不足，则函数进行等待，直到设备采集到足够的样点后返回成功；
- ◆ 若函数运行时，设备处于空闲状态，没有启动采集，则函数将持续等待，不能返回。此时可通过调用 MPS_Start 函数来启动采集，启动采集后 MPS_DataIn 函数即可正常获取数据并返回成功；
- ◆ 若函数运行前，通过 MPS_TimeOut 函数设了超时参数，则 MPS_DataIn 函数在等待过程中会自动计时，超时后被强制结束并返回失败值；
- ◆ 若函数被调用时，设备与计算机未连接，则函数执行返回失败值；
- ◆ 若函数运行过程中，设备与计算机之间的连接中断（如断开 USB 连接、设备断电、重启等），则函数执行结束并返回失败值。

在外部启动触发模式中：

- ◆ 需要在设备处于空闲时调用 MPS_DataIn 函数，此时函数将进入等待外部触发信号的状态；
- ◆ 当收到外部触发信号后，MPS_DataIn 函数将正常读取样点数据并返回成功；
- ◆ 若长时间未收到触发信号，可通过调用 MPS_Start 函数手动启动采集，使 MPS_DataIn 函数终止等待；
- ◆ 也可通过设置 MPS_TimeOut 超时参数，使 MPS_DataIn 函数在达到预设超时时间后自动退出等待状态。

注意：由于板载硬件缓冲容量有限，其可存储的样点数量存在上限。当新采集的样点持续写入而旧样点未能及时读取时，一旦缓冲区写满，将按照 FIFO 原则丢弃最早存入的数据，导致数据丢失。此时通过 MPS_DataIn 函数读取的数据将无法与先前数据保持连续性。因此，在采集软件设计过程中：

- ◆ 必须确保 MPS_DataIn 函数的调用频率不低于数据采集速率；
- ◆ 避免在两次 MPS_DataIn 调用之间添加人为的延迟或等待操作；
- ◆ 在两次 MPS_DataIn 调用之间尽量避免复杂运算、实时绘图或硬盘读写等操作；
- ◆ 建议将采集软件的进程设置为最高优先级，防止因系统资源竞争导致的运行卡

顿。

int *DataBuffer：此参数用来传递指向用户定义的软件缓冲区的指针，用于存储从硬件读出的数据。

软件在通过 MPS_DataIn 函数读取数据时，需要将数据存入一个程序能够直接访问的数据存储区，因此在调用 MPS_DataIn 函数之前，程序需要预先建立一个数据缓冲。该缓冲应定义为长度大于或等于 SampleNumber 的 32 位有符号整型数（int32_t）的一维数组，并取指向该数组首地址的指针，赋值给 DataBuffer 参数。此缓冲与设备内部的硬件缓冲区（DAQ Buffer）无关，是一个独立的软件缓冲区。

写入 DataBuffer 的数据为 32 位有符号整数，如果是在 64 位环境下编译，请务必注意将其定义为 **32 位整型（int32_t）** 的数据类型。DataBuffer 的每个元素对应一个采样点数据，采样点数据按对应的通道和时间次序循环分布，分布方式为：

若共有 N 个通道，则 DataBuffer[(m-1)×N + (n - 1)]对应第 n 通道（CHn）的第 m 个采样点数据（其中 *n ≤ N；(m-1)×N +(n - 1) ≤ SampleNumber - 1*）。即：

DataBuffer[0]对应 CH1 的第一个采样点数据，DataBuffer[1]对应 CH2 的第一个采样点数据，……DataBuffer[n-1]对应 CHn 的第一个采样点数据；

DataBuffer[N]对应 CH1 的第二个采样点数据，DataBuffer[N+1]对应 CH2 的第二个采样点数据……DataBuffer[N+(n-1)]对应 CHn 的第二个采样点数据；

DataBuffer[2×N]对应 CH1 的第三个采样点数据，DataBuffer[2×N+1]对应 CH2 的第三个采样点数据……DataBuffer[2×N+(n-1)]对应 CHn 的第三个采样点数据；

……

DataBuffer[(m-1)×N]对应 CH1 的第 m 个采样点数据，DataBuffer[(m-1)×N+1]对应 CH2 的第 m 个采样点数据……DataBuffer[(m-1)×N+(n-1)]对应 CHn 的第 m 个采样点数据。

以四个通道（N=4）的情况为例：

DataBuffer[0]对应 CH1 的第一个采样点数据，DataBuffer[1]对应 CH2 的第一个采样点数据，DataBuffer[2]对应 CH3 的第一个采样点数据，DataBuffer[3]对应 CH4 的第一个采样点数据；DataBuffer[4]对应 CH1 的第二个采样点数据，DataBuffer[5]对应 CH2 的第二个采样点数据，DataBuffer[6]对应 CH3 的第二个采样点数据，DataBuffer[7]对应 CH4 的第二个采样点数据……以此类推，DataBuffer[(m-1)×4+(n-1)]对应 CHn 的第 m 个采样点数据。

每个采样点数据都是一个带符号的 32 位整型数据，其值的大小与被测信号的电压值成正比，对应关系为：*Voltage[i] = ((double)DataBuffer[i]/8388608) * 5.3*，即：电压值等于整型数除以 8388608（其中 8388608 为 24 位有符号整数的最大范围，即 2^23），再乘以 5.3V（MPS-HD1802 的量程参考值）。在计算中，建议先将整型数转换为 32 位浮点数（float）或 64 位浮点数（double）再进行除法运算，以防在整型除法运算中损失精度。

MPS_DataIn 函数执行成功后，读取到的数据将会被写入到 DataBuffer 所指向的缓冲区，所写入的数据个数由 SampleNumber 参数决定，因此，DataBuffer 缓冲区的元素个数必须大于或等于 SampleNumber。

MPS_DataIn 函数执行失败时，DataBuffer 中的数据无效。

int SampleNumber：此参数用来设置 MPS_DataIn 函数执行一次所读取的样点个数。

在 MPS-HD 系列的数据读取中，SampleNumber 是所有通道的样点数总和，每个通道对应的样点数是 SampleNumber 除以通道个数。SampleNumber 的最小取值为 128，最大取值为 1024000，且取值必须为 128 的整倍数。若取值不是 128 的整倍数，函数会自动配置为小于该数的最大的 128 整倍数。

SampleNumber 必须小于或等于 DataBuffer 缓冲区的大小，如果超出 DataBuffer 缓冲区范围，可能引发内存越界，导致程序错误。

通过“采集时间 = SampleNumber / (通道数 × 采样率)”可以计算这些样点对应的采集时间，例如 SampleNumber = 12800，通道数 = 2，采样率 = 64000 点/秒，通过“12800 样点 / (2 通道 × 64000 样点/秒) = 0.1 秒”可计算出需要 0.1 秒的采集时间。因此，在连续调用 MPS_DataIn 函数来采集的程序设计中，若设置每次读取 12800 个样点，那么程序调用 MPS_DataIn 函数的平均时间间隔不得超过 0.1 秒，否则会使每次读数都有部分数据无法及时读出，积累在硬件缓冲区内，若硬件缓冲区写满，会导致新数据覆盖旧数据，出现数据丢失，影响数据的连续性。

在需要连续采集并实时刷新显示界面的程序中，SampleNumber 对应的采集时间会对软件显示的刷新帧率产生影响。若采集时间较长，MPS_DataIn 单次执行耗时增加，导致循环频率降低，可能使程序响应变慢，给软件使用者带来卡顿的操作体验。但另一方面，若采集时间过短，循环执行过于频繁，会导致界面刷新过快，降低程序效率并增加 CPU 占用。建议通过设置合理的 SampleNumber 值，使循环执行频率控制在每秒 30-50 次，匹配人机交互的流畅帧率，同时避免 CPU 资源浪费。

在针对分段数据进行分析的程序中（例如 FFT 频谱分析），也可以考虑直接设置较大的 SampleNumber 值，一次性读出足够的数据进行分析，从而简化程序，降低编程难度。

HANDLE DeviceHandle：目标设备的操作句柄。

- int MPS_Stop (HANDLE DeviceHandle)

int MPS_Stop 函数用于停止设备的采集。在设备处于采集状态时，调用此函数可使设备停止采集，进入待机状态，并清空硬件缓冲区内的所有未读数据；若设备已处于待机状态，调用此函数无效。函数执行成功返回 1，失败返回 0。

当一次连续的采集过程结束时，软件应调用 MPS_Stop 函数，使设备进入待机状态，为下一次启动采集做好准备。若结束时未调用 MPS_Stop 函数，设备将持续采集并循环覆盖硬件缓冲区，导致下次采集时读取到缓冲区内的过期数据，而非实时数据。

此外，设置参数和获取信息时，也要求设备处于待机状态。为避免设备状态不确定，建议在软件的初始化阶段，预先调用一次 MPS_Stop 函数，强制设备进入待机状态，确保后续设置安全。

HANDLE DeviceHandle：目标设备的操作句柄。

- int MPS_CloseDevice (HANDLE DeviceHandle)

int CloseDevice 函数用于安全关闭设备。在软件退出时，或者释放设备控制权时，需要调用本函数。函数执行成功返回 1，失败返回 0。

若在未关闭设备的情况下强行中断软件，系统可能无法完全释放设备驱动资源，导

致其他程序无法正常打开该设备。此时可以关闭所有与设备驱动相关联的程序和线程，通过重新拔插 USB 连接或发送复位指令等方式对硬件进行复位，必要时重启计算机。

HANDLE DeviceHandle：目标设备的操作句柄。

- int MPS_TimeOut (int TimeOut_ms, HANDLE DeviceHandle)

int MPS_TimeOut 函数用于设置超时时限，以限制 MPS_DataIn 函数的执行时间。MPS_TimeOut 函数需要在 MPS_DataIn 函数之前调用，可通过指定一个时间阈值(TimeOut_ms)，防止出现 MPS_DataIn 函数因无限等待而导致程序阻塞的情况。若 MPS_DataIn 函数在指定时间内未完成，则触发超时处置，强制其终止并返回失败值。若未启用超时限制，则 MPS_DataIn 函数将持续等待，直至成功或因其他原因失败。

MPS_TimeOut 函数主要用于：

1、在外部启动触发模式下，需要提前调用 MPS_DataIn 函数来等待触发。通过设置 MPS_TimeOut 函数，可以为等待时间设定上限，若在指定时间内未收到触发，则自动终止等待。

2、在调用 MPS_Start 函数启动采集之前，通过 MPS_TimeOut 函数预设一个超时时限，可以在出现硬件故障或软件逻辑错误（例如在调用了 MPS_Stop 函数后又调用了 MPS_DataIn 函数）的情况下，强行终止 MPS_DataIn 函数，避免程序阻塞。

通过 MPS_TimeOut 函数启用了超时限制后:

1、在 MPS_DataIn 函数正常返回，没有触发超时处置的情况下，无论调用多少次 MPS_DataIn 函数，已设置的超时时限始终有效，直到出现超时或清除设置。

2、当触发一次超时处置后，所设置的超时限制将失效，系统重置为未启用超时限制的状态。

3、当调用 MPS_Stop 函数时，在停止采集的同时，会清除已设置的超时限制。

MPS_TimeOut 函数执行后，将返回实际生效的 TimeOut_ms 值（具体见下面 TimeOut_ms 参数说明）。

int TimeOut_ms：此参数用于设置超时时间，单位为毫秒。有效的超时时间范围为 100（毫秒）至 1000000（毫秒）。

1、当 TimeOut_ms 设定为 0 或小于 100 的值时，不启用超时限制功能。此时 MPS_TimeOut 函数执行后返回 0。

2、当 TimeOut_ms 设定为 100 至 1000000 之间的值时，超时时间按 TimeOut_ms 的值进行设置，单位为毫秒。此时 MPS_TimeOut 函数执行后返回 TimeOut_ms 的值。

3、当 TimeOut_ms 设定为大于 1000000 的值时，超时时间将被强制设置为 1000000 毫秒。此时 MPS_TimeOut 函数执行后返回 1000000。

HANDLE DeviceHandle：目标设备的操作句柄。

- int MPS_GetDeviceDescriptor (char *DescriptorBuffer, int BufferSize, HANDLE DeviceHandle) int MPS_GetDeviceDescriptor 函数用于获取设备型号描述信息。函数执行成功后，可获得一组字符，例如："MPS-HD2402"、"MPS-HD2404"等,最后以'\0'（空字符）结尾。该组字符可用于识别当前设备的硬件型号。函数执行成功返回 1，失败返回 0。

char *DescriptorBuffer：此参数为指向软件定义的字符缓冲区的指针，该缓冲区用来保存设备型号信息。

字符缓冲区应为一个 char[]类型的数组或是一个字符串，数组或字符串需要提前设定长度，其大小必须大于等于 BufferSize 参数的值，建议分配 64 字节。MPS_GetDeviceDescriptor 函数执行成功后，会向该缓冲区内写入最多 BufferSize 个字节的字符（包含'\0'）。

int BufferSize：此参数指定读取字符的字节数。

MPS_GetDeviceDescriptor 函数执行成功后，DescriptorBuffer 缓冲区中的前 BufferSize 个字符将被更新。BufferSize 的取值必须小于等于 DescriptorBuffer 缓冲区的实际大小，否则可能引发内存越界错误，且最大取值不超过 64，大于 64 时将自动设置为 64。推荐将 BufferSize 的值设置为 64。

HANDLE DeviceHandle：目标设备的操作句柄。

- int MPS_GetDeviceInformation (char *InformationBuffer, int BufferSize, HANDLE DeviceHandle)

int MPS_GetDeviceInformation 函数用于获取设备的硬件信息。函数执行成功后，可获得当前设备的完整硬件信息，包括设备型号、序列号、硬件参数和用户自定义信息。信息以字符形式返回，最后以'\0'（空字符）结尾。函数执行成功返回 1，失败返回 0。

MPS_GetDeviceInformation 函数执行耗时较长，建议放在软件初始化阶段调用。

char *InformationBuffer：此参数为指向软件定义的字符缓冲区的指针，该缓冲区用来保存设备硬件信息。

字符缓冲区应为一个 char[]类型的数组或是一个字符串，数组或字符串需要提前设定长度，其大小必须大于等于 BufferSize 参数的值，建议分配 2048 字节。MPS_GetDeviceInformation 函数执行成功后，会向该缓冲区内写入最多 BufferSize 个字节的字符（包含'\0'）。

int BufferSize：此参数指定读取字符的字节数。

MPS_GetDeviceInformation 函数执行成功后，InformationBuffer 缓冲区中的前 BufferSize 个字符将被更新。BufferSize 的取值必须小于等于 InformationBuffer 缓冲区的实际大小，否则可能引发内存越界错误，且最大取值不超过 2048，大于 2048 时将自动设置为 2048。推荐将 BufferSize 的值设置为 2048。

HANDLE DeviceHandle：目标设备的操作句柄。

● int MPS_SetUserInformation(char *UserInformationBuffer, int BufferSize, HANDLE DeviceHandle)

int MPS_SetUserInformation 函数用于写入用户自定义信息。函数执行成功后，可向设备硬件内写入一段用户自定义的信息，信息写入后断电不丢失。信息以字符形式写入，建议最后以'\0'（空字符）结尾。函数执行成功返回 1，失败返回 0。

MPS_SetUserInformation 函数执行耗时较长，需在设备空闲状态下调用。自定义信息支持约 10 万次重复写入，可用于对类似软件校验码、软件自定义校准系数、传感器的灵敏度等固定信息的离线保存。

char *UserInformationBuffer：此参数为指向软件定义的字符缓冲区的指针，该缓冲区用来保存待写入的自定义信息。

字符缓冲区应为一个 char[]类型的数组或是一个字符串，数组或字符串需要提前设定长度，其大小必须大于等于 BufferSize 参数的值，建议分配 500 字节。在 MPS_SetUserInformation 函数执行前，需要预先向字符缓冲区内写入用户自定义的字符信息，建议最后以'\0'（空字符）结尾。MPS_SetUserInformation 函数执行成功后，会将该缓冲区内前 BufferSize 个字节的字符写入硬件。

int BufferSize：此参数指定写入字符的字节数。

MPS_SetUserInformation 函数执行成功后，UserInformationBuffer 缓冲区中的前 BufferSize 个字符将被写入硬件。BufferSize 的取值必须小于等于 UserInformationBuffer 缓冲区的实际大小，否则可能引发内存越界错误，且最大取值不超过 500，大于 500 时将自动设置为 500。推荐将 BufferSize 的值设置为有效字符数（包含结尾的'\0'）。

HANDLE DeviceHandle：目标设备的操作句柄。

● int MPS_ResetDevice (HANDLE DeviceHandle)

int MPS_ResetDevice 函数用于复位硬件。函数执行成功后，设备硬件会被复位为初始状态。USB 连接将自动断开并重连，复位前的设备句柄将失效，程序需重新调用 MPS_OpenDevice 函数来获取新的句柄，用于后续操作。函数执行成功返回 1，失败返回 0。

当设备状态异常，导致程序出现阻塞时，可以尝试使用复位函数来复位硬件。若因函数阻塞，无法获得当前正在使用的设备句柄，也可以在其他线程中通过再次调用 MPS_OpenDevice 来获取一个新的设备句柄，用来执行 MPS_ResetDevice 函数。

HANDLE DeviceHandle：目标设备的操作句柄。

三、 编程范例

```
/*采集数据的功能子函数*/
/*基本流程：打开设备—>设置参数—>开始采集—>循环获取数据—>停止采集—>关闭设备*/

/*预定义SampleRate参数值*/
#define SampleRate64K 64000                    //采样率为64K
#define SampleRate32K 32000                    //采样率为32K
#define SampleRate16K 16000                    //采样率为16K
#define SampleRate8K 8000                      //采样率为8K
#define SampleRate4K 4000                      //采样率为4K
#define SampleRate2K 2000                      //采样率为2K
#define SampleRate1K 1000                      //采样率为1K

/*预定义DIOMode参数值*/
#define DIOIdle 0                              //DIO口闲置
#define DIOOutput5V 1                          //DIO口输出5V高电平
#define ExternalTrigger 2                      //DIO设置为外部触发启动模式


/*预定义变量*/
#define VoltageRange 5.3                       //MPS-HD1802的电压量程值为5.3V
#define MaxNumberPerCH 7680000                 //每通道电压样点个数的最大值，此处预定义为约768万点，可支持64K采样率下最多连续采集120s
double Voltage[2][MaxNumberPerCH] = {0};       //保存电压的二维缓存数组，每通道最多可保存MaxNumberPerCH点

/*采集数据子函数*/
int GetVoltageData(int SampleNumberPerCH = 64000, int SampleRate = SampleRate64K, int DIOMode = DIOIdle)        //采集数据，参数分别为：采样点数、采样率、DIO的模式
{
    /*函数声明*/
    #define Handle int

    HINSTANCE hDll;                            //打开DLL
    hDll=LoadLibrary("MPS Driver.dll");
    if(NULL==hDll)
    {
        AfxMessageBox("Cannot find DLL");
        return 0;
    }

    typedef Handle(*lpMPS_OpenDevice)(int DeviceNumber);     //打开设备函数的声明
```

```
lpMPS_OpenDevice MPS_OpenDevice=(lpMPS_OpenDevice)GetProcAddress(hDll,"MPS_OpenDevice");
if(NULL==MPS_OpenDevice)
{
    AfxMessageBox("Cannot find <MPS_OpenDevice> function");
}

typedef int(*lpMPS_CloseDevice)(Handle DeviceHandle);   //关闭设备函数的声明
lpMPS_CloseDevice
MPS_CloseDevice=(lpMPS_CloseDevice)GetProcAddress(hDll,"MPS_CloseDevice");
if(NULL==MPS_CloseDevice)
{
    AfxMessageBox("Cannot find <MPS_CloseDevice> function");
}

typedef int(*lpMPS_Configure)(int SampleRate, int DIOMode, Handle DeviceHandle);    //
设置设备参数函数的声明
lpMPS_Configure MPS_Configure=(lpMPS_Configure)GetProcAddress(hDll,"MPS_Configure");
if(NULL==MPS_Configure)
{
    AfxMessageBox("Cannot find <MPS_Configure> function");
}

typedef int(*lpMPS_Start)(Handle DeviceHandle);    //启动设备采集函数的声明
lpMPS_Start MPS_Start=(lpMPS_Start)GetProcAddress(hDll,"MPS_Start");
if(NULL==MPS_Start)
{
    AfxMessageBox("Cannot find <MPS_Start> function");
}

typedef int(*lpMPS_Stop)(Handle DeviceHandle); //停止设备采集函数的声明
lpMPS_Stop MPS_Stop=(lpMPS_Stop)GetProcAddress(hDll,"MPS_Stop");
if(NULL==MPS_Stop)
{
    AfxMessageBox("Cannot find <MPS_Stop> function");
}

typedef int(*lpMPS_DataIn)(int *DataArray,int SampleNumber, Handle DeviceHandle);    //
读取数据函数的声明
lpMPS_DataIn MPS_DataIn=(lpMPS_DataIn)GetProcAddress(hDll,"MPS_DataIn");
if(NULL==MPS_DataIn)
{
    AfxMessageBox("Cannot find <MPS_DataIn> function");
}
```

```
typedef int(*lpMPS_TimeOut)( int TimeOut_ms,Handle DeviceHandle);//设置读数超时函数的声明
lpMPS_TimeOut MPS_TimeOut=(lpMPS_TimeOut)GetProcAddress(hDll,"MPS_TimeOut");
if(NULL==MPS_TimeOut)
{
    AfxMessageBox("Cannot find <MPS_TimeOut> function");
}

/*函数声明结束*/


/*采集数据*/
//定义变量
int Flag = 1;                          //函数执行成功标志
int DeviceNumber = 0;                  //操作系统分配的设备硬件序号
Handle DeviceHandle;                   //设备句柄

DeviceHandle = MPS_OpenDevice(DeviceNumber);  //先打开设备
if(DeviceHandle == -1)                 //若打开失败，报错并返回
{
    AfxMessageBox("OpenDeviceError");
    return 0;
}

Flag = MPS_Configure(SampleRate, DIOMode, DeviceHandle);    //设置采样率和复用IO口为闲置模式

MPS_TimeOut(5000, DeviceHandle);              //设置等待超时为5s，如果用于外启动超时，可根据情况修改设置值

if(DIOMode != ExternalTrigger)Flag = MPS_Start(DeviceHandle);//启动设备采集；外部启动模式时不通过调用MPS_Start来启动


//以下为循环调用采集函数的方式来读取数据
int DataBuffer[2*1024] = {0};          //用于保存读出的原始整形数据的缓存数组，请务必注意此变量一定定义为32位整形数
                                       //用于保存经过计算的电压的数据数组Voltage已定义为全局变量
if(SampleNumberPerCH > MaxNumberPerCH)
        SampleNumberPerCH = MaxNumberPerCH;  //每通道获取到的样点个数取值应小于MaxNumberPerCH

int Counter = 0;                       //数据点数计数变量
```

```
while(1)
{
    Flag = MPS_DataIn(DataBuffer,1024*2,DeviceHandle); //读取数据，每次每通道读取1024点，两个通道共1024*2

    if(Flag != 0)                          //如果采集成功
        {
            for(int i = 0; i < 1024; i++)    //数据处理，计算电压值；两个通道的数据在缓存中循环排列；电压值 = (采集值/(65536*128))*量程
            {
                if(Counter >= SampleNumberPerCH) break;
                                               //如果样点达到SampleNumberPerCH，则跳出for循环
                Voltage[0][Counter] = ((double)DataBuffer[i*2] / 8388608) * VoltageRange;                                         //通道1的电压
                Voltage[1][Counter] = ((double)DataBuffer[i*2 + 1] / 8388608) * VoltageRange;                                         //通道2的电压
                Counter++;
            }
        }
        else
        {
            AfxMessageBox("MPS_DataInError");//如果采集失败报错并跳出while循环
            break;
        }

    if(Counter >= SampleNumberPerCH)       //如果读出全部数据，跳出while循环。此处也可添加其他跳出循环的条件
    {
        break;
    }
}

//至此采集过程完成，后面要停止硬件采集并关闭设备
Flag = MPS_Stop(DeviceHandle);             //采集任务结束后停止采集
Flag = MPS_CloseDevice(DeviceHandle);      //最后关闭设备

if (Counter == SampleNumberPerCH)          //判断GetVoltageData函数是否已经获取到了指定的SampleNumberPerCH个样点数
{
    AfxMessageBox("MPS_DataInSuccess");
    return 1;
}
```

```
    else return 0;
}




/*获取设备硬件信息的功能子函数*/
int GetInformation()                                    //获取设备硬件信息
{
    /*函数声明*/
    #define Handle int

    HINSTANCE hDll;                                     //打开DLL
    hDll=LoadLibrary("MPS Driver.dll");
    if(NULL==hDll)
    {
        AfxMessageBox("Cannot find DLL");
        return 0;
    }

    typedef Handle(*lpMPS_OpenDevice)(int DeviceNumber);//打开设备函数的声明
    lpMPS_OpenDevice MPS_OpenDevice=(lpMPS_OpenDevice)GetProcAddress(hDll,"MPS_OpenDevice");
    if(NULL==MPS_OpenDevice)
    {
        AfxMessageBox("Cannot find <MPS_OpenDevice> function");
    }

    typedef int(*lpMPS_CloseDevice)(Handle DeviceHandle);//关闭设备函数的声明
    lpMPS_CloseDevice
MPS_CloseDevice=(lpMPS_CloseDevice)GetProcAddress(hDll,"MPS_CloseDevice");
    if(NULL==MPS_CloseDevice)
    {
        AfxMessageBox("Cannot find <MPS_CloseDevice> function");
    }

    typedef int(*lpMPS_GetDeviceDescriptor)(char *DescriptorBuffer,int BufferSize,Handle
DeviceHandle);                                          //获取设备型号信息函数的声明
    lpMPS_GetDeviceDescriptor
MPS_GetDeviceDescriptor=(lpMPS_GetDeviceDescriptor)GetProcAddress(hDll,"MPS_GetDeviceDescrip
tor");
    if(NULL==MPS_GetDeviceDescriptor)
    {
        AfxMessageBox("Cannot find <MPS_GetDeviceDescriptor> function");
    }
```

```
typedef int(*lpMPS_GetDeviceInformation)(char *DeviceInformationBuffer,int
BufferSize,Handle DeviceHandle);                    //获取设备完整硬件信息函数的声明
    lpMPS_GetDeviceInformation
MPS_GetDeviceInformation=(lpMPS_GetDeviceInformation)GetProcAddress(hDll,"MPS_GetDeviceInfor
mation");
    if(NULL==MPS_GetDeviceInformation)
    {
        AfxMessageBox("Cannot find <MPS_GetDeviceInformation> function");
    }

    /*函数声明结束*/


    //定义变量
    int Flag = 1;                           //函数执行成功标志
    int DeviceNumber = 0;                   //操作系统分配的设备硬件序号
    Handle DeviceHandle;                    //设备句柄

    //读取设备型号信息示例
    char DeviceDescriptor[64] = {0};        //保存设备型号信息的缓存数组

    DeviceHandle = MPS_OpenDevice(DeviceNumber);   //先打开设备
    if(DeviceHandle == -1)                  //若打开失败，报错并返回
    {
        AfxMessageBox("OpenDeviceError");
        return 0;
    }

    Flag = MPS_GetDeviceDescriptor(DeviceDescriptor,64,DeviceHandle);//获取设备型号

    Flag = MPS_CloseDevice(DeviceHandle);          //最后关闭设备

    AfxMessageBox(DeviceDescriptor);               //显示设备型号

    //读取设备硬件信息示例
    char DeviceInformation[2048] = {0};     //保存设备型号信息的缓存数组

    DeviceHandle = MPS_OpenDevice(DeviceNumber);   //先打开设备
    if(DeviceHandle == -1)                  //若打开失败，报错并返回
    {
        AfxMessageBox("OpenDeviceError");
        return 0;
```

```
    }

    Flag = MPS_GetDeviceInformation(DeviceInformation,2048,DeviceHandle);//获取设备信息

    Flag = MPS_CloseDevice(DeviceHandle);           //最后关闭设备

    AfxMessageBox(DeviceInformation);               //显示设备信息

    return 1;

}




/*写入用户自定义信息的功能子函数*/
int SetUserInformation()                            //写入用户自定义信息
{
    /*函数声明*/
    #define Handle int

    HINSTANCE hDll;                                 //打开DLL
    hDll=LoadLibrary("MPS Driver.dll");
    if(NULL==hDll)
    {
        AfxMessageBox("Cannot find DLL");
        return 0;
    }

    typedef Handle(*lpMPS_OpenDevice)(int DeviceNumber);//打开设备函数的声明
    lpMPS_OpenDevice MPS_OpenDevice=(lpMPS_OpenDevice)GetProcAddress(hDll,"MPS_OpenDevice");
    if(NULL==MPS_OpenDevice)
    {
        AfxMessageBox("Cannot find <MPS_OpenDevice> function");
    }

    typedef int(*lpMPS_CloseDevice)(Handle DeviceHandle);//关闭设备函数的声明
    lpMPS_CloseDevice
MPS_CloseDevice=(lpMPS_CloseDevice)GetProcAddress(hDll,"MPS_CloseDevice");
    if(NULL==MPS_CloseDevice)
    {
        AfxMessageBox("Cannot find <MPS_CloseDevice> function");
    }
```

```
    typedef int(*lpMPS_SetUserInformation)(char *UserInformationBuffer,int BufferSize,Handle
DeviceHandle);                                  //写入用户自定义信息函数的声明
    lpMPS_SetUserInformation
MPS_SetUserInformation=(lpMPS_SetUserInformation)GetProcAddress(hDll,"MPS_SetUserInformation
");
    if(NULL==MPS_SetUserInformation)
    {
        AfxMessageBox("Cannot find <MPS_SetUserInformation> function");
    }

    /*函数声明结束*/

    //定义变量
    int Flag = 1;                           //函数执行成功标志
    int DeviceNumber = 0;                   //操作系统分配的设备硬件序号
    Handle DeviceHandle;                    //设备句柄

    //写入用户自定义信息示例
    char UserInformation[500] = "This is a test."; //待写入的用户信息的缓存数组

    DeviceHandle = MPS_OpenDevice(DeviceNumber);  //先打开设备
    if(DeviceHandle == -1)                  //若打开失败，报错并返回
    {
        AfxMessageBox("OpenDeviceError");
        return 0;
    }

    Flag = MPS_SetUserInformation(UserInformation,500,DeviceHandle);//写入用户信息

    Flag = MPS_CloseDevice(DeviceHandle);   //最后关闭设备

    AfxMessageBox(UserInformation);         //显示写入的内容

    return 1;

}
```

```
/*重置硬件示例*/
int ResetDevice()                           //重置硬件函数
{
    /*函数声明*/
```

```
#define Handle int

HINSTANCE hDll;                               //打开DLL
hDll=LoadLibrary("MPS Driver.dll");
if(NULL==hDll)
{
    AfxMessageBox("Cannot find DLL");
    return 0;
}

typedef Handle(*lpMPS_OpenDevice)(int DeviceNumber);//打开设备函数的声明
lpMPS_OpenDevice MPS_OpenDevice=(lpMPS_OpenDevice)GetProcAddress(hDll,"MPS_OpenDevice");
if(NULL==MPS_OpenDevice)
{
    AfxMessageBox("Cannot find <MPS_OpenDevice> function");
}

typedef int (*lpMPS_ResetDevice)(Handle DeviceHandle);//重置硬件函数的声明
lpMPS_ResetDevice MPS_ResetDevice
=(lpMPS_ResetDevice)GetProcAddress(hDll,"MPS_ResetDevice");
if(NULL==MPS_ResetDevice)
{
    AfxMessageBox("Cannot find <MPS_ResetDevice> function");
}

/*函数声明结束*/

//定义变量
int Flag = 1;                                 //函数执行成功标志
int DeviceNumber = 0;                         //操作系统分配的设备硬件序号
Handle DeviceHandle;                          //设备句柄

DeviceHandle = MPS_OpenDevice(DeviceNumber);  //先打开设备
if(DeviceHandle == -1)                        //若打开失败，报错并返回
{
    AfxMessageBox("OpenDeviceError");
    return 0;
}

Flag = MPS_ResetDevice(DeviceHandle);         //重置硬件函数

if (Flag != 0)
    {
        AfxMessageBox("MPS_ResetDeviceSuccess");
```

```
        }
    else
        {
            AfxMessageBox("MPS_ResetDeviceError");
        }
    return Flag;
}



/*功能测试*/
void Test()
{
    int Flag = 0;
//  Flag = GetInformation();                    //读取设备硬件信息测试，通常需要时在软件程序初始化时读一次即可，不必经常读取
//  Flag = SetUserInformation();                //写入用户自定义信息测试，如不需要写入信息的话不必执行此函数
    Flag = GetVoltageData(64000,SampleRate64K, DIOIdle);//采集数据测试，测试64K采样率采集1s
//  Flag = ResetDevice();                       //重置硬件测试
}
```

第四章 注意事项

- 插拔设备接线端口请用力适度，以免损坏接口。
- 设备与计算机 USB 断开后，请间隔 1s 左右再重新接入。
- 设备通常使用计算机 USB 供电即可正常工作，如出现供电不足，可使用带外部电源的 USB HUB 等方式来提高 USB 总线的供电能力。
- 设备属于精密电子仪器，使用中请注意防尘防潮与防静电。若存在人体静电放电风险，请预先做好防静电措施，并在使用中尽量避免触碰接头、外壳、信号线及传感器中的金属部分等。设备长期不用时，请做好密封保存。
- 禁止用户自行拆卸设备的下层外壳，一经拆卸将不再享有保修服务，且因此导致的设备故障将由用户自行承担。
- MPS-HD 系列产品自出厂之日起三年内，凡用户遵守贮存、运输和使用要求，由于产品质量导致的故障，凭保修卡或订单信息免费维修。因违反操作规定和使用要求造成人为损坏的，需交纳维修费用进行维修。
- MPS 系列信号采集卡由 Morpheus Electronic 提供，更多产品和相关信息请浏览：www.mps-electronic.com.cn,咨询邮箱 mail@mps-electronic.com.cn。