

MPS-150102
MiniDAQ 精巧型数据采集器
使用说明

Ver. 1.1

第一章 产品概述



一、 产品简介

MPS-150102 (MiniDAQ) 是一款基于 USB 总线的精巧型信号采集器。MPS-150102 外形与 U 盘相似，但其内部却集成了高速 AD 和复合功能的输入输出双向数字接口。MPS-150102 能够采集 $\pm 5V$ 之间的单端电压信号，采样率高达 100Ksps，并且内置了放大器，可以由外部的增益电阻来设置放大倍数。采集到的信号数据可以连续不断的传送到计算机，通过使用配套应用软件或另行编程，可以实现电压表、虚拟示波器、信号记录仪、频谱分析仪等多种高级功能。此外，MPS-150102 还具有一个复合型的数字输入输出端口，既可以作为外部数字电平信号的采集端口，也可以作为数字电平的输出使用，此外 5V 高电平输出时可以作为电源输出来驱动继电器或为外部器件供电。MPS-150102 便于携带，能够极大提高工程师的工作效率，测量测试行业的必备工具。

MPS-150102 采用 USB2.0 全速总线接口，无需外部电源，连接到计算机即可使用。MPS-150102 内部集成了高达 100Ksps 采样率的 10bitADC，能以每秒 1000 点到每秒 100000 点的速率采集信号，采集到的数据可以实时连续的上传到计算机。同一台计算机上可以连接多个 MPS-150102，多个采集器既可以独立工作，也可受软件控制协调工作。

MPS-150102 采用跨平台的动态链接库作为驱动函数接口，可工作在 Win9X/Me、Win2000/XP、VISTA、WIN7、WIN8、WIN10、WIN11 等常用操作系统下，支持 64 位系统，支持 VB, VC, C++Builder, C#, Dephi, LabVIEW, Matlab 等绝大多数编程语言，驱动函数简洁清晰，编程简单高效。随卡还附送了数款功能强大的应用软件，常用的功能可以直接实现，无需编程更加方便。

二、 性能指标

2.1、USB 总线

- USB2.0 全速总线传输
- 支持热插拔和即插即用

2.2、输入通道

- 单通道单端输入

2.3、电压量程

- -4.86V 到+4.86V（典型值）

2.4、输入阻抗

- 2M 欧姆

2.5、采样率

- 1K 到 100K 任意可调，步进为 1K
- 支持连续采样和实时上传

2.6、分辨率

- 10bit

2.7、复合功能 DIO

- 可设置为输出低电平【0V】、输出高电平【5V】和输入模式三种状态
- 低电平吸收电流 < 10mA
- 高电平输出电流 < 100mA（USB 供电充足），可驱动继电器或为外部器件供电
- 输入模式下为高输入阻抗状态，引脚悬空时弱上拉到 3.3V
- 即使在输出状态下，计算机也可读到端口的电平状态

2.8、工作温度

- -40℃ - 70℃

三、应用领域

传感器信号采集

便携式虚拟示波器

信号记录

频谱分析

四、软件资源

Windows95/98/NT/2000/XP/VISTA/Window 7/Window 8/ Window 10/ Window 11 下的驱动程序；通用 DLL 动态链接库；LabVIEW、VB、VC 环境下的编程参考代码；配套应用软件等，其他资源参见网站信息。

五、配件清单

- [1] MPS-150102 信号采集器一套；
- [2] 保修卡一张；

六、售后服务

一年保修。

第二章 设备安装

一、 MPS-150102 信号采集卡硬件接口

USB:	接至计算机的 USB 总线插口
DIO:	复合型数字输入输出端口
DC:	正负 5V 直流电压输入端口
AC:	正负 5V 交流电压输入端口
GND:	采集器地线
GR:	增益电阻配置口

二、 接口说明

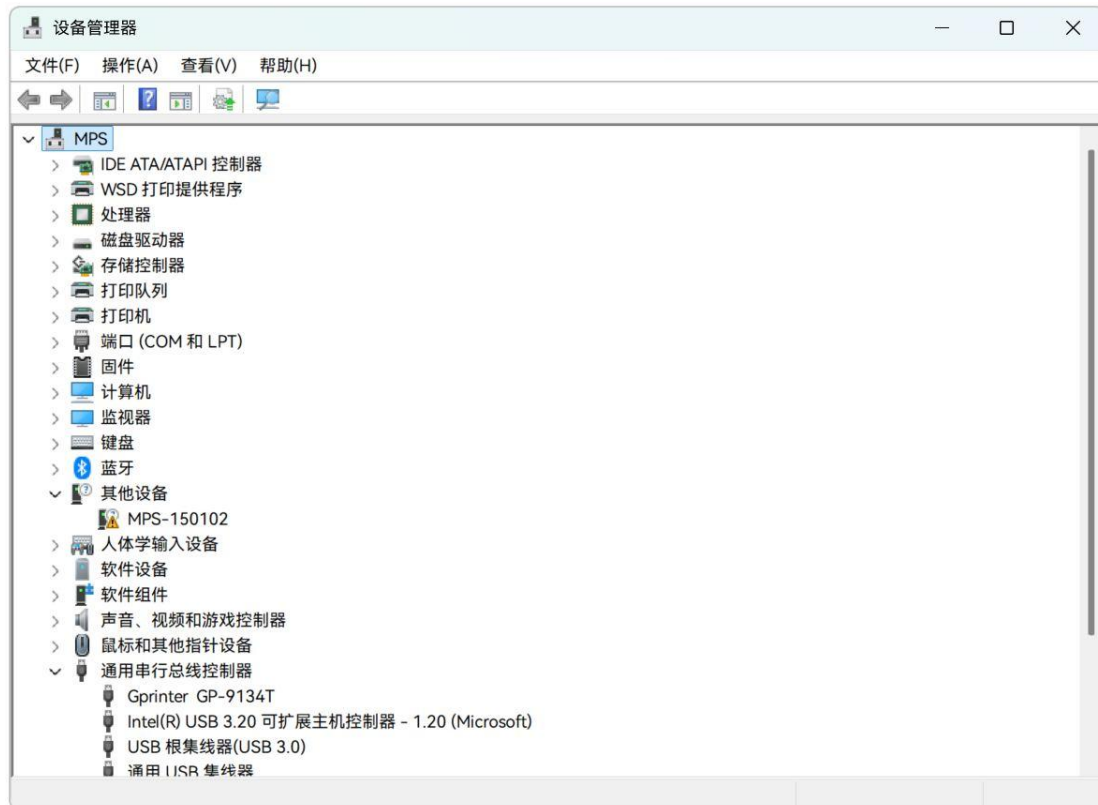
- 使用设备时请通过 USB 数据线将设备与计算机连接，设备正常连接后，可在 WINDOWS 的设备管理器中查询到设备信息。
- 设备地线与单端信号源的地线接通，设备的信号输入端与信号源的信号线接通。建议先将设备连接到计算机，再接入外部信号（或给信号源上电）。
- 复合型数字端口配置为输入模式时，被内部弱上拉拉至 3.3V，端口输入电压在 2.5V 以上时识别为高电平，1V 以下时识别为低电平。
- 复合型数字端口配置为 5V 输出时，内部与 USB 总线 5V 电源连通，可最大输出 100mA 的电流。
- DC 端口接入电压的范围为正负 5V，不能长期接入超出该范围的电压。
- AC 端口内置 10 微法隔直电容，只允许通过交流信号。
- DC 或 AC 中的某个作为输入时，另一个应悬空，以免产生输入耦合。
- GR 端口悬空时，内部放大器不起作用，增益为 1；GR 端口与 GND 直接接入电阻 R，内部放大器对信号进行放大，放大倍数为 $(1 + 10K / R)$ ，例如接入 5K 电阻，增益为 $(1 + 10 / 5) = 3$ 倍。
- GR 连接增益电阻后，DC、AC 端口的量程也随之改变，改变后的量程为 $\pm 5V / (1 + 10K / R)$ 。

三、 指示灯说明

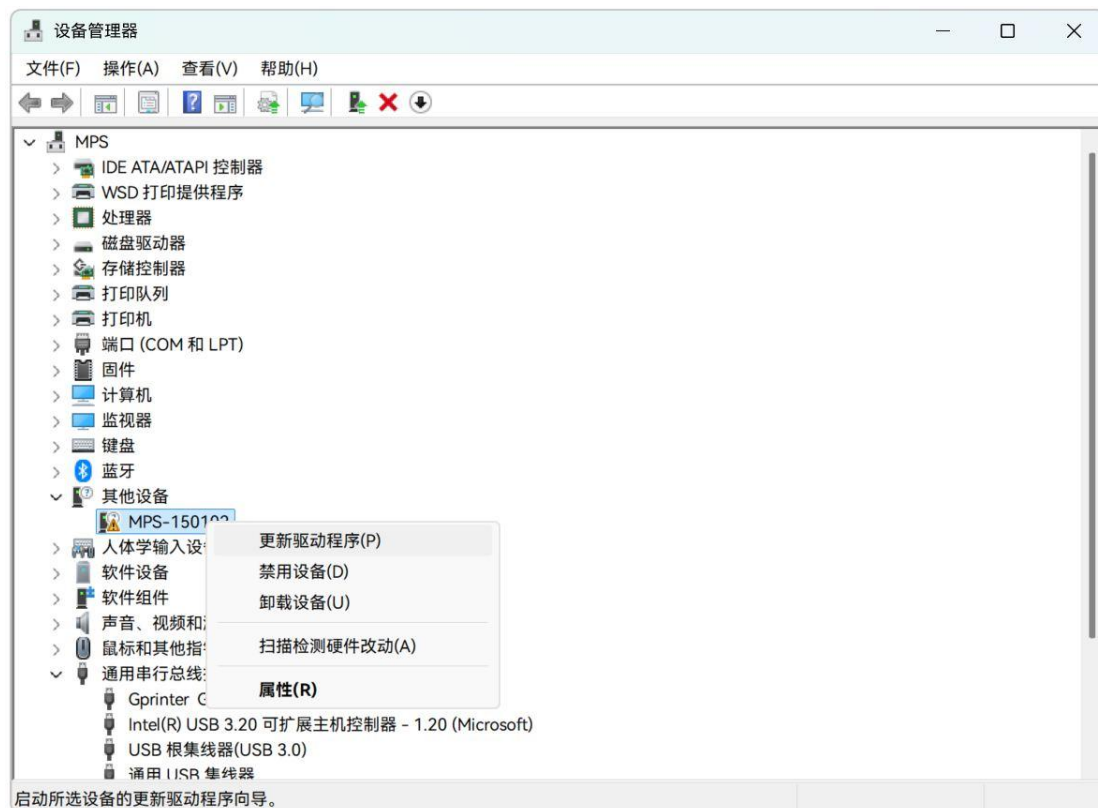
- MPS-150102 内置蓝色和绿色两只用于指示采集器的工作状态的指示灯。
- 上电时，蓝色指示灯闪烁一次，说明采集器已自检成功。当采集器进行数据采集时，蓝色指示灯点亮，当采集停止进入空闲状态时，蓝色指示灯熄灭。
- 绿色指示灯用于指示 DIO 端口的当前电平状态，当 DIO 端口处于高电平状态时，绿色指示灯亮起，当 DIO 端口处于低电平状态时，绿色指示灯熄灭。

四、 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。



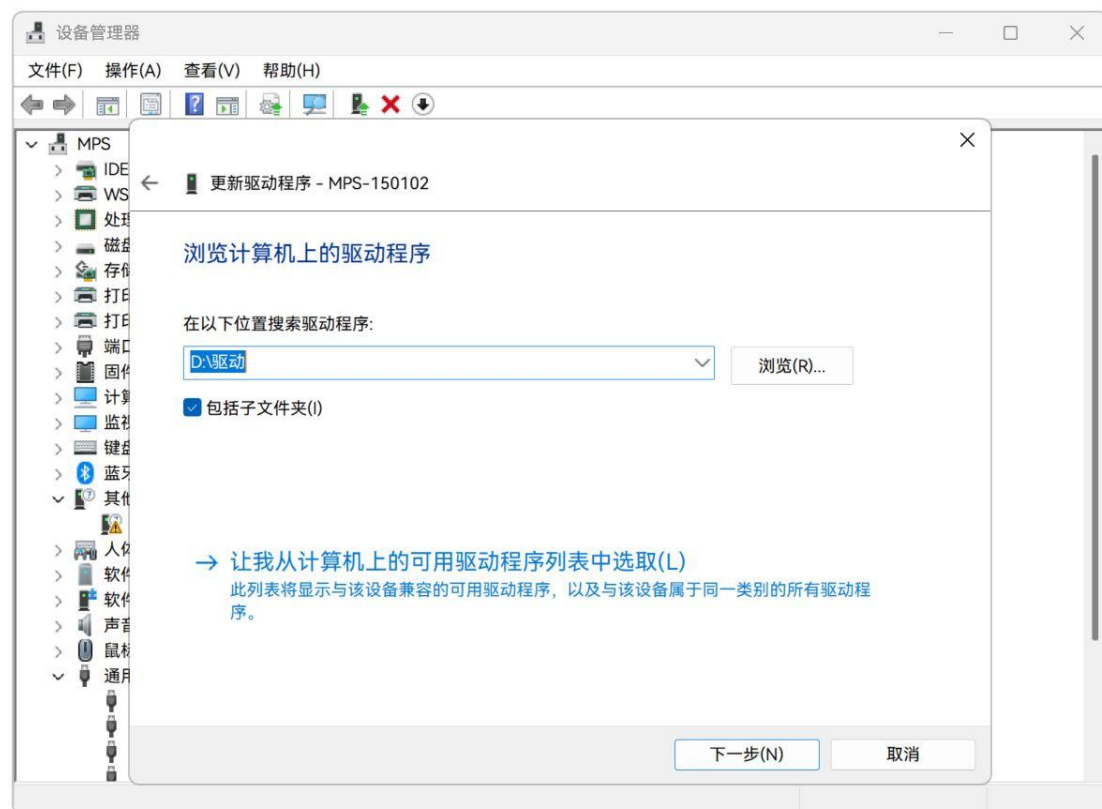
2. 在设备名称上点右键，选择“更新驱动程序”。



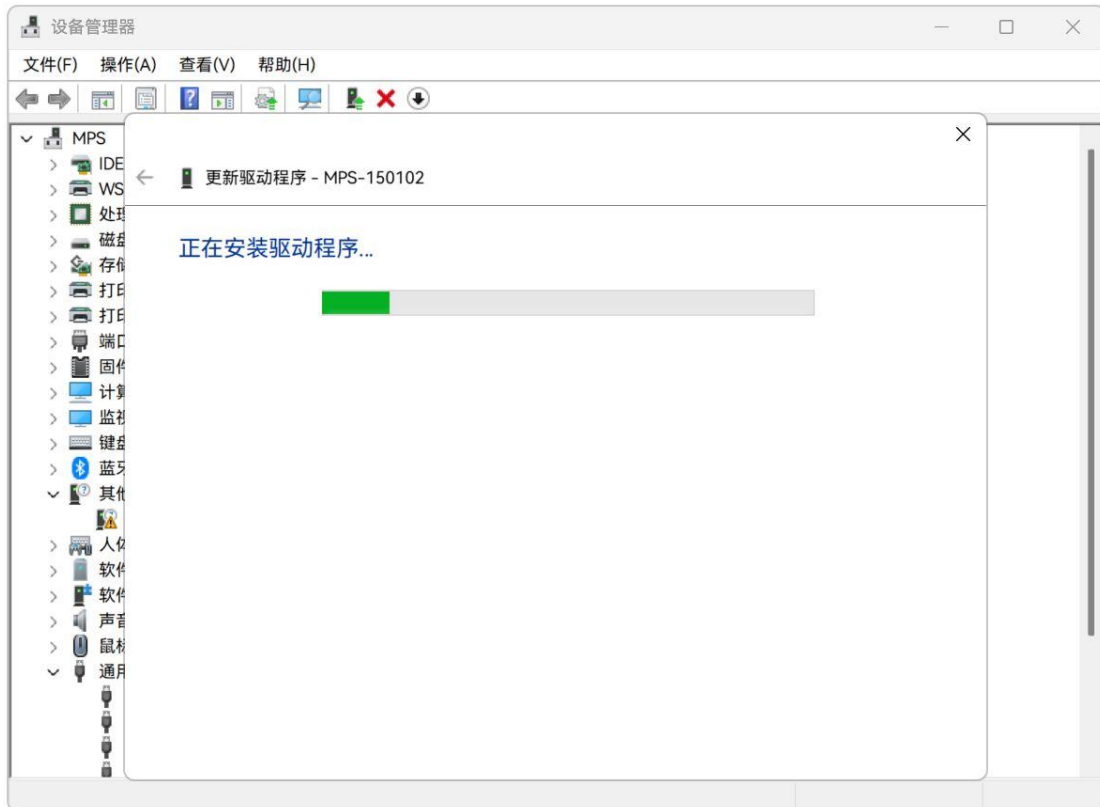
3. 在弹出的驱动安装向导界面中，选择“浏览我的电脑以查找驱动程序”。



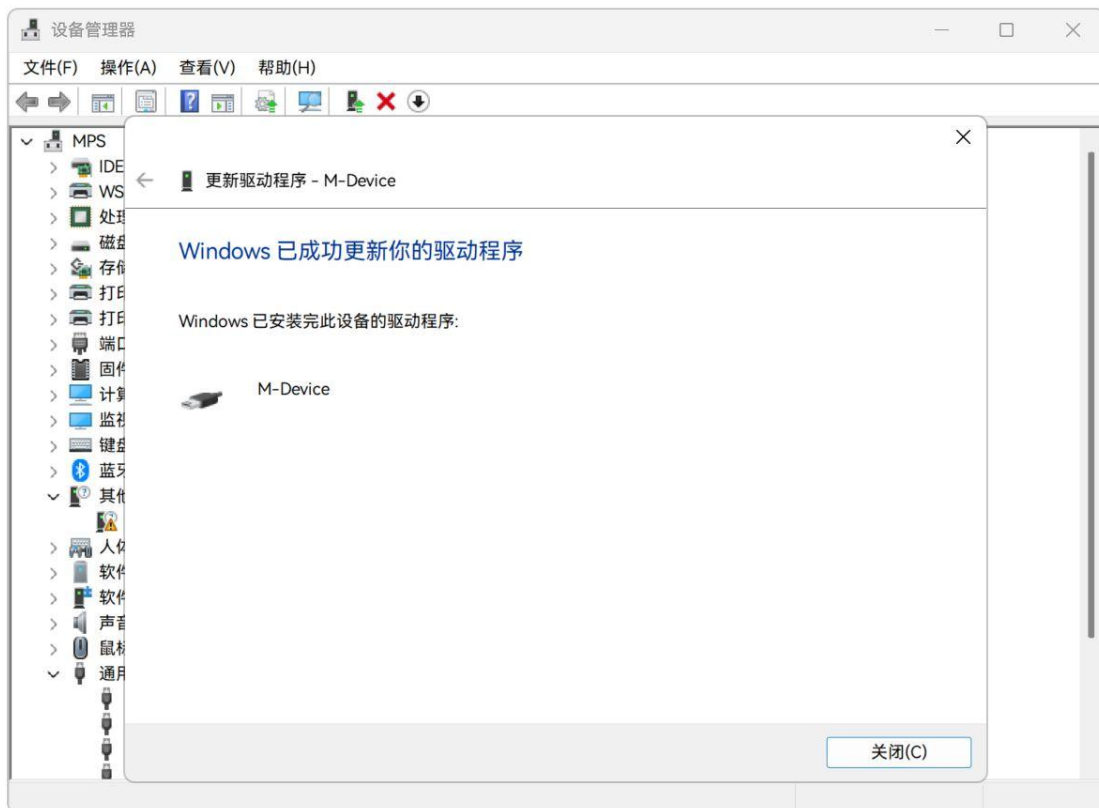
4. 点击“浏览”按钮，选择采集卡驱动所在的文件夹。如果驱动为压缩包，需要先解压缩后再进行操作。



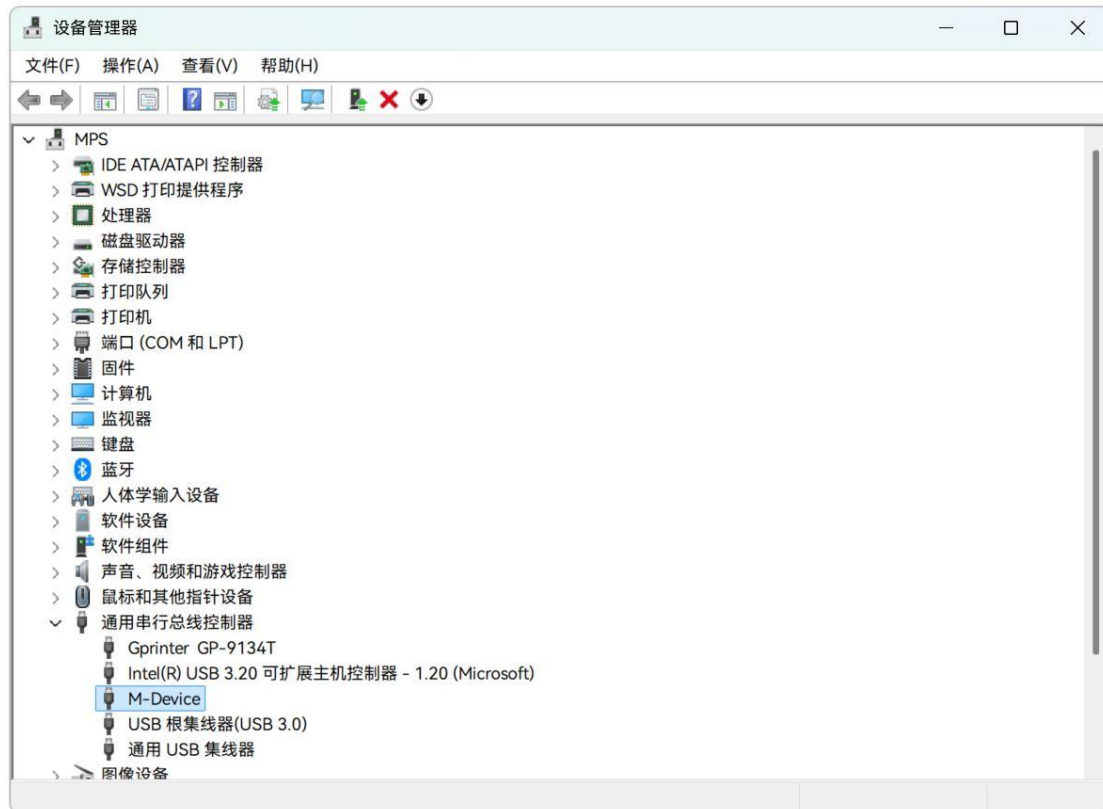
5. 点击下一步，出现如下驱动安装界面。过程中如果出现信任驱动的安全提示，勾选信任选项并点击确认。



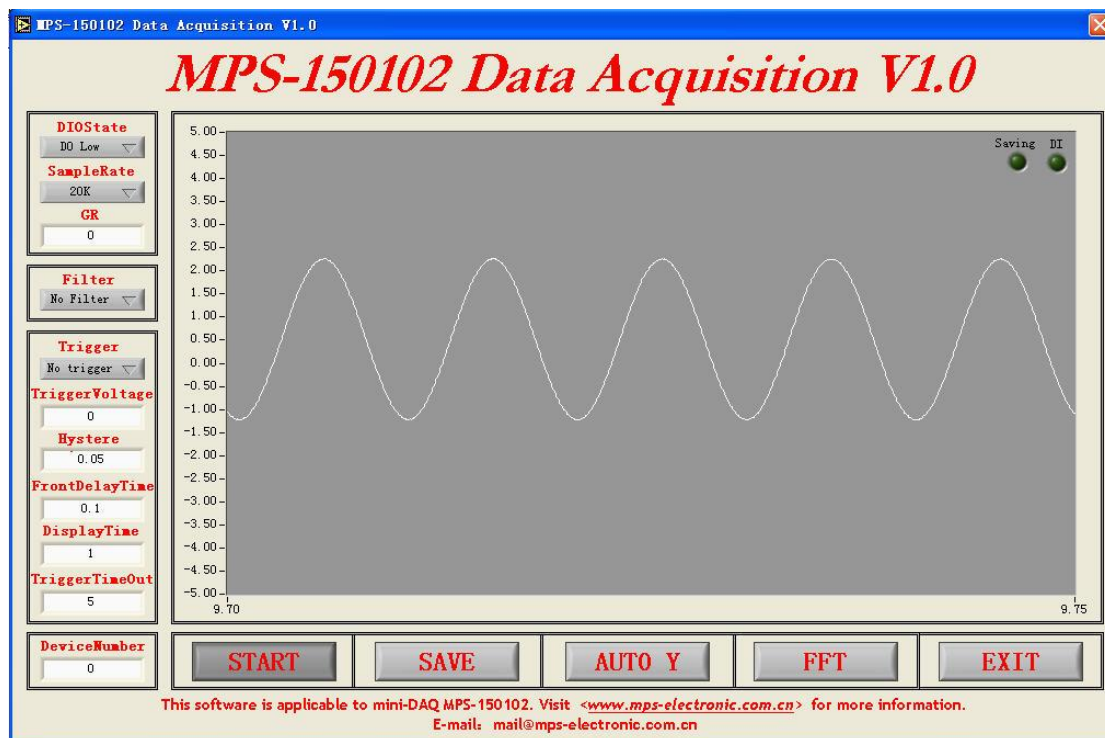
6. 驱动安装完成，出现如下提示框。



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。



五、 功能测试



MPS-150102 数据采集软件截图

1. 将 MPS-150102 数据采集卡与计算机 USB 接口连接。
2. 按第三小节所描述的过程安装驱动程序。
3. 下载并解压安装 MPS 应用软件运行库 “MPSSoftwareRuntimeLib.rar”。
4. 打开测试程序 “MPS-150102 Data Acquisition V1.0.exe”，出现如上图所示的界面。
5. 将待测信号连接到 MPS-150102 输入端，测试图中使用 100Hz 正弦波作为输入。
6. 点击 “START” 按钮，即可在软件中看到采集到的信号波形。
7. 修改左上角的 “DIOState” 设置可以改变 DIO 端口的状态；修改 “SampleRate” 可以设置采样率；当在采集器 GR 端口与 GND 之间接入增益电阻时，将电阻值输入 GR 可自动进行信号还原运算。
8. 点击 “Filter” 可启用软件内置的滤波器，选择滤波器类型后，会弹出对话框进行参数配置。
9. 点击 “Trigger” 可以启用软件内置的触发功能，其下各项为触发参数的配置项。
10. 若计算机上同时连接了多块 MPS-150102，按接入计算机的先后顺序，对应的设备序号依次对应 0、1、2……。在 “DeviceNumber” 中输入对应的设备序号，软件就针对特点的采集器进行操作。当只有一个 MPS-150102 时，设备序号为 0。
11. 点击 SAVE 可以将采集到的数据记录到 D:\MPS-150102 DataFiles 文件夹中。记录的文件可通过 “MPS Signal Replay.exe” 进行回放和文本导出。
12. 点击 “AUTO Y” 按钮，可自动将 Y 轴坐标与显示的波形范围相匹配。
13. 点击 “FFT” 按钮可显示信号的频谱信息。
14. 在波形图上点击右键，可在弹出的菜单中启用高级显示功能，包括 X 轴最大范围显示、拖动曲线、曲线区间放大、绘图方式设定、清除曲线和将曲线导出为图片等。
15. 波形图右上角的 “DI” 指示灯指示 DIO 端口的电平状态。
16. 点击 “EXIT” 按钮或窗口右上角的 “×” 可退出程序。

17. 如计算机未连接 MPS-150102、驱动未安装或连接出错，软件会出现异常提示，此时应检查硬件是否正确连接。

第三章 用户编程

一、 动态链接库 (DLL)

MPS-150102 采用 DLL (Dynamic Linkable Library, 动态链接库) 的方式来进行编程驱动。DLL 的编制与具体的编程语言及编译器无关, 只要遵循约定的 DLL 接口规范和调用方式, 用各种语言编写的 DLL 都可以相互调用。

DLL 可以方便的在 VC、VB、LabVIEW 等语言下被调用, 具体方式分别为:

- 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 " libname " [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] [ParamArray] varname [()] [As type]
```

Optional (可选) 表示参数不是必需的。如果使用该选项, 则 **arglist** 中的后续参数都必需是可选的, 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连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。

二、编程函数及参数

MPS-150102 提供的驱动 DLL 文件名为 MPS-150102.dll，软件程序员只需调用该 DLL 内的驱动函数即可实现对板卡的控制，编程与打包发布时该 DLL 必须被包含在工程内。

MPS-150102 的驱动函数包含两类，一类是以 MPS 开头的标准驱动函数，用于对板卡基本的驱动功能进行编程。另一类是以 Express 开头的快速实现函数，这类函数事实上是基于 MPS 标准驱动函数，但内部按照特定的功能对进行了处理，可直接实现一些常用的功能，能够简化编程。但快速实现函数往往存在功能上的限制，适用范围有限，在高级编程中仍然需要使用标准驱动函数。

MPS-150102.dll 内部的驱动函数如下：

- `Int __stdcall MPS_OpenDevice (int DeviceNumber, HANDLE *DeviceHandle)`

`int MPS_OpenDevice` 函数执行打开设备的功能。如设备打开成功将返回 0；打开失败则返回-1；未检测到驱动文件返回-2。

`int DeviceNumber` 当前打开设备的序号。当有多套 MPS-150102 设备同时连接到计算机时，将按照设备连接到计算机的先后顺序从 0 到 9 依次分配序号，打开特定序号的设备时将返回设备的句柄，该句柄将用后续操作。最多支持同时连接 10 个设备。只有一块卡连接时，默认设备号为 0。

`HANDLE *DeviceHandle` 指向用来存放设备句柄的变量的指针。函数成功执行后，指针指向的变量被更新为设备的句柄。

- `int __stdcall MPS_FlushBuffer (HANDLE DeviceNumber)`

`int MPS_FlushBuffer` 函数执行清除历史缓冲数据的功能。主要用于间断读取数据时，确保调用 `DataIn` 函数获得的是最新数据。一般是在启动采集之前，或是两次调用 `DataIn` 函数间隔较长时，调用一次该函数来清除不需要的历史数据。在连续不断采集的过程中无需调用该函数。如成功清除缓冲区的历史数据，函数返回 0；失败则返回-1。

`HANDLE DeviceHandle`：操作所针对的设备句柄。

- `int __stdcall MPS_Configure (int SampleRate, int DIOState, HANDLE DeviceHandle)`

`int MPS_Configure` 函数执行设定设备参数的功能。若函数执行成功，返回 0；执行失败返回-1。

`int SampleRate`: 信号采集采样率。即每秒内每个信号输入通道所采集的样点个数。取值范围为 1000 到 100000 之间，且应为 1000 的整倍数。当设定为 1000 至 100000 范围内的其他值时，函数会将设定值自动校正为小于并且最接近该值的有效值，当设定值小于 1000 时被自动校正为 1000，当设定值大于 100000 时被自动校正为 100000。

`int DIOWState`: 复合型数字输入\输出端口工作模式。DIO 端口可以被设置为三种工作模式，当 DIOWState 为 0 时，端口被连接到地，输出低电平，吸收电流小于 10mA；当 DIOWState 为 1 时，端口被连接到 5V 电源，输出高电平，输出电流小于 100mA；当 DIOWState 为 2 时，端口处于高阻输入态，并被弱上拉到 3.3V。当 DIOWState 小于 0 时被自动校正为 0，当 DIOWState 大于 2 时被自动校正为 2。

`HANDLE DeviceHandle`: 操作所针对的设备句柄。

- `int __stdcall MPS_Start (HANDLE DeviceHandle)`

`int MPS_Start` 函数执行启动采集的功能。若函数执行成功，返回 0；执行失败返回-1。板卡在待机状态时不能调用 `MPS_DataIn` 函数，只有在调用 `MPS_Start` 后，设备才开始采集和传输数据，才能通过调用 `MPS_DataIn` 函数获取数据，否则会导致 `MPS_DataIn` 执行报错行。

`HANDLE DeviceHandle`: 操作所针对的设备句柄。

- `int __stdcall MPS_DataIn(float * DataBuffer, int * DI, int SampleNumber, float GR, HANDLE DeviceHandle)`

`int MPS_DataIn` 函数执行获取数据的功能。若函数执行成功，返回 0；执行失败返回-1。当板卡处于待机状态，不能调用该函数，否则函数会报错。

`float * DataBuffer`: 保存所采集到数据的缓存区的首地址。若函数执行成功，该数组内数据被自动更新为最新采集到的数据（更新的元素个数由 `SampleNumber` 决定）；若函数执行失败，该数组内数据无效。`DataBuffer` 为一个一维数组，其每个元素对应一个采样点的数据，每个采样点数据是一个 `float` 浮点型数据，依次代表采集到的一个样点电压值。当函数成功执行后，`DataBuffer` 的前 `SampleNumber` 个数据将被更新。

`int * DI`: 从复合型数字输入输出端口 (DIO) 读取的当前电平状态。`DI` 为一个 32 位带符号整形数(或定义为一个只包含 1 个元素的 `int` 型数组)，执行函数时将其地址指针(数组首地址指针)作为参数传递给 `MPS_DataIn` 函数，当函数成功执行后，指针所指向的地址内的数据将被更新，即 `DI` 被更新。更新后的数据为最新获得的一个 DIO 电平状态点。`DI` 的更新值为 0 或者 1, 0 代表 DIO 端口当前处于低电平状态(小于 1V)，1 代表 DIO 端口当前处于高电平状态(高于 2.5V)。无论 DIO 处于何种工作模式，从 `DI` 中均可获得当前端口的状态，当端口被设置成输出低电平时读到的为 0，被设置为输出高电平时读到的是 1。

`int SampleNumber`: 执行一次 `MPS_DataIn` 函数所获取的样点个数。该参数决定函数执行一次数据数组中所更新的数据个数，函数从所设定的输入通道中读到 `SampleNumber` 个数据点后将成功返回。参数的最小值为 500，且设定值应为 500 的整倍数，否则会自动校正为低于该值并与之最接近的 500 的整倍数。当参数小于 500 时，函

数会直接返回，无法正常采集到数据。

float GR: 外接的增益电阻阻值。当 GR 端口和 GND 端口之间接入增益电阻时，MPS-150102 内部放大器被启用，信号进行增益放大后再被采集，适用于对幅值低于标准量程的小信号进行采集。当 GR 端悬空时，MPS_DataIn 函数的 GR 参数应为 0，此时函数返回与标准量程对应的采样值。当 GR 与 GND 之间接入阻值为 R 的电阻时，信号被放大 $(1 + 10K / R)$ 倍，此时将电阻值 R 作为参数传递给 MPS_DataIn 函数，则函数在获取数据的时候自动对采集到的数据进行运算，返回的数据和接入的实际电压值相符。若此时将 MPS_DataIn 函数的 GR 参数设置为 0，则获取到的是放大后的电压值。

HANDLE DeviceHandle: 操作所针对的设备句柄。

- **int __stdcall MPS_Stop (HANDLE DeviceHandle)**

int MPS_Stop 函数执行停止板卡工作的功能。若函数执行成功，返回 0；执行失败返回-1。该函数执行后，在下次执行 MPS_Start 函数之前，板卡处于待机状态，待机状态下不能执行 MPS_DataIn，否则无法响应。

HANDLE DeviceHandle: 操作所针对的设备句柄。

- **int __stdcall MPS_CloseDevice (HANDLE DeviceHandle)**

int MPS_Close 函数执行关闭设备的功能。若函数执行成功，返回 0；执行失败返回-1。设备打开并执行完所有操作后，必须对设备进行关闭。若没有经过关闭而意外退出，必须关闭所有与设备操作函数相关的程序线程，再重新连接设备硬件。

HANDLE DeviceHandle: 操作所针对的设备句柄。

- **int __stdcall Express_GetSingleSample(float *Data, int *DI, float GR, int DeviceNumber)**

int Express_GetSingleSample 函数执行单点采集的快速功能。若函数执行成功，返回 0；执行失败返回-1。函数每次调用 Express_GetSingleSample 函数获得一个单一的数据样点，该样点事实上由 5K 采样率下采集到的 500 个样点平均而成，实际有效的分辨率比物理分辨率高。但 Express_GetSingleSample 函数耗时约为 100ms，只适用于低速采样的应用场合。Express_GetSingleSample 是独立的功能函数，只调用该函数即可实现单点采集的功能。但该函数不能与 MPS 标准驱动函数混用，不能在调用 OpenDevice 而未调用 CloseDevice 关闭之前调用该函数。

float * Data: 保存所采集到数据的缓存区的地址。Data 为一个 float 型的数(或定义为一个只包含 1 个元素的 float 型数组)，执行函数时将其地址指针(数组首地址指针)作为参数传递给 Express_GetSingleSample 函数，若函数执行成功，该数据被自动更新为最新采集到的数据；若函数执行失败，该数据无效。

int * DI: 与“MPS_DataIn”函数中的“DI”参数定义相同。

float GR: 与“MPS_DataIn”函数中的“GR”参数定义相同。

int DeviceNumber: 操作所针对的设备序号，与“MPS_OpenDevice”函数的“DeviceNumber”参数定义相同。注意此处是 DeviceNumber 而不是 DeviceHandle。

- **int __stdcall Express_GetMultiSamples(float * Data, int * DI, int SampleRate,**

```
int SampleNumber, float GR, int DeviceNumber)
```

`int Express_GetMultiSamples` 函数执行多样点采集的快速功能。若函数执行成功，返回 0；执行失败返回-1。`Express_GetMultiSamples` 是独立的功能函数，只调用该函数即可实现多样点采集的功能。但该函数不能与 MPS 标准驱动函数混用，不能在调用 `OpenDevice` 而未调用 `CloseDevice` 关闭之前调用该函数。前后连续两次调用该函数时获取的数据不一定连续，因此该函数只适用于分段短时间采集的设计，不适合需要长时间连续采集的场合。

`float * Data`: 与“MPS_DataIn”函数中的“DataBuffer”参数定义相同。

`int * DI`: 与“MPS_DataIn”函数中的“DI”参数定义相同。

`int SampleRate`: 与“MPS_Configure”函数中的“SampleRate”参数定义类似。

`int SampleNumber`: 与“MPS_DataIn”函数中的“SampleNumber”参数定义类似，但该函数的 `SampleNumber` 可以为任意值，不受 500 整倍数的限制。

`float GR`: 与“MPS_DataIn”函数中的“GR”参数定义相同。

`int DeviceNumber`: : 操作所针对的设备序号。

● `int __stdcall Express_SetDIO (int DIOState, int DeviceNumber)`

`int Express_SetDIO` 函数执行设置 DIO 状态的快速功能。若函数执行成功，返回 0；执行失败返回-1。`Express_GetMultiSamples` 是独立的功能函数，只调用该函数即可实现设置 DIO 状态的功能。但该函数不能与 MPS 标准驱动函数混用，不能在调用 `OpenDevice` 而未调用 `CloseDevice` 关闭之前调用该函数。

`int DIOState`: 与“MPS_Configure”函数中的“DIOState”参数定义相同。

`int DeviceNumber`: 操作所针对的设备序号。

三、 编程范例

一个标准驱动采集过程的编程流程:

OpenDevice → FlushBuffer → Configure → Start → 循环 DataIn → Stop → CloseDevice

```
int MPS_TEST()
{
    //Declare function in DLL.
    HINSTANCE hDll; //open DLL.
    hDll = LoadLibrary("MPS-150102.dll");
    if(NULL == hDll)
    {
        AfxMessageBox("Can't find DLL");
        return 0;
    }

    typedef int(__stdcall *lpOpenDevice)(int DeviceNumber, HANDLE* Handle); //function
    "OpenDevice".
    lpOpenDevice OpenDevice = (lpOpenDevice)GetProcAddress(hDll, "MPS_OpenDevice");
    if(NULL == OpenDevice)
```

```
{
    AfxMessageBox("Can't find <MPS_OpenDevice> function");
}

typedef int(_stdcall *lpCloseDevice) (HANDLE DeviceHandle); //function "CloseDevice".
lpCloseDevice CloseDevice = (lpCloseDevice)GetProcAddress(hDll, "MPS_CloseDevice");
if(NULL == CloseDevice)
{
    AfxMessageBox("Can't find <MPS_CloseDevice> function");
}

typedef int(_stdcall *lpStart) (HANDLE DeviceHandle); //function "Start".
lpStart Start = (lpStart)GetProcAddress(hDll, "MPS_Start");
if(NULL == Start)
{
    AfxMessageBox("Can't find <MPS_Start> function");
}

typedef int(_stdcall *lpStop) (HANDLE DeviceHandle); //function "Stop".
lpStop Stop = (lpStop)GetProcAddress(hDll, "MPS_Stop");
if(NULL == Stop)
{
    AfxMessageBox("Can't find <MPS_Stop> function");
}

typedef int(_stdcall *lpFlushBuffer) (HANDLE DeviceHandle); //function "FlushBuffer".
lpFlushBuffer FlushBuffer = (lpFlushBuffer)GetProcAddress(hDll, "MPS_FlushBuffer");
if(NULL == FlushBuffer)
{
    AfxMessageBox("Can't find <MPS_FlushBuffer> function");
}

typedef int(_stdcall *lpConfigure) (int SampleRate, int DI0State, HANDLE DeviceHandle);
//function "Configure".
lpConfigure Configure = (lpConfigure)GetProcAddress(hDll, "MPS_Configure");
if(NULL == Configure)
{
    AfxMessageBox("Can't find <MPS_Configure> function");
}

typedef int(_stdcall *lpDataIn) (float *dataArray, int *DI, int SampleNumber, float GR, HANDLE
DeviceHandle);
//function "DataIn".
lpDataIn DataIn = (lpDataIn)GetProcAddress(hDll, "MPS_DataIn");
if(NULL == DataIn)
```

```
{
    AfxMessageBox("Can't find <MPS_DataIn> function");
}

//main program.
HANDLE DeviceHandle; // Hhandle of device.
float DataInBuffer[1000] = {0}; // Temporary array to hold data returned
                                // from "DataIn" function.
int DI[1] = {0}; // One element DI array to hold DIO state
                 // returned from "DataIn" function.
float GR = 0; // External gain amplifier resistance.
int FLAG = 0; // Error flag.

FLAG = OpenDevice(0, &DeviceHandle); // Open device .
if(FLAG != 0) // If open faild, send a message and return.
{
    AfxMessageBox("OpenDeviceError");
    return -1;
}

FlushBuffer(DeviceHandle); // Flush buffer.
Configure(50000, 0, DeviceHandle); // Configure the device as 50K samplerate,
                                   // low level D0 output.
Start(DeviceHandle); // Start input and output.

for(int i = 0 ; i < 10; i++) // Input several times.
{
    // FlushBuffer(DeviceHandle); // If time in data processing is too long,
                                // flush the buffer.

    FLAG = DataIn(DataInBuffer, DI, 1000, GR, DeviceHandle) // Input.
    if(FLAG != 0) AfxMessageBox("Error!"); // If input faild, send a message.
    else // If successful.
    {
        // Add processing code.
    }
}

Stop(DeviceHandle); // Stop DAQ.
CloseDevice(DeviceHandle); // Close device to end an operation cycle.
return FLAG; // Return the flag.
}
```

```
int EXPRESS_TEST()
{
    //Declare function in DLL.
    HINSTANCE hDll; //open DLL.
    hDll=LoadLibrary("MPS-150102.dll");
    if(NULL==hDll)
    {
        AfxMessageBox("Can't find DLL");
        return 0;
    }

    typedef int(_stdcall *lpGetSingleSample)( float *Data, int *DI, float GR, int DeviceNumber);
    //function "GetSingleSample".
    lpGetSingleSample GetSingleSample = (lpGetSingleSample)GetProcAddress(hDll,
"Express_GetSingleSample");
    if(NULL == GetSingleSample)
    {
        AfxMessageBox("Can't find <Express_GetSingleSample> function");
    }

    typedef int(_stdcall *lpGetMultiSamples)( float *Data, int *DI, int SampleRate, int
SampleNumber, float GR, int DeviceNumber); //function "GetMultiSamples".
    lpGetMultiSamples GetMultiSamples = (lpGetMultiSamples)GetProcAddress(hDll,
"Express_GetMultiSamples");
    if(NULL == GetMultiSamples)
    {
        AfxMessageBox("Can't find <Express_GetMultiSamples> function");
    }

    typedef int(_stdcall *lpSetDIO)(int DIOState, int DeviceNumber); //function "SetDIO".
    lpSetDIO SetDIO = (lpSetDIO)GetProcAddress(hDll, "Express_SetDIO");
    if(NULL == SetDIO)
    {
        AfxMessageBox("Can't find <Express_SetDIO> function");
    }

    //main program.
    int DeviceNumber = 0; // Device number.
    float SingleDataInBuffer[1] = {0}; // Temporary array to hold data returned from
"GetSingleSample" function.
    float MultiDataInBuffer[1000] = {0}; // Temporary array to hold data returned from
"GetMultiSamples" function.
    int DI[1] = {0}; // One element array to hold DIO state returned.
```

```
float GR = 0; // External gain amplifier resistance.
int SampleRate = 10000; // Sample rate.
int SampleNumber = 1000; // Number of samples to read from
"GetMultiSamples" function. It can be any number less than the length of "MultiDataInBuffer".
int FLAG =0; // Error flag.

// SetDIO as High.
FLAG = SetDIO(1,DeviceNumber);
if(FLAG != 0) // If error occurred, send a message and return.
{
    AfxMessageBox("Error");
    return 0;
}

// Get a sample.
FLAG = GetSingleSample(SingleDataInBuffer,DI,GR, DeviceNumber);
if(FLAG != 0) // If error occurred, send a message and return.
{
    AfxMessageBox("Error");
    return -1;
}

// Get multi samples.
FLAG = GetMultiSamples(MultiDataInBuffer,DI, SampleRate, SampleNumber, GR, DeviceNumber);
if(FLAG != 0) // If error occurred, send a message and return.
{
    AfxMessageBox("Error");
    return -1;
}

else
{
    // Add processing code.
}

return 0;
}
```

第四章 注意事项

- 拔插设备接线端口请用力适度，以免损害接口。
- 信号输入端允许接入的电压范围为-5V 至+5V，超出量程之外的电压可能损害设备。
- 当 DIO 设置为高电平输出时，务必防止 DIO 与 GND 短路，否则可能损害计算机 USB 接口。
- 用户须注意电源的开关顺序，使用时要求先将设备上电或连至计算机，而后开启信号源电源；关闭时先关信号源电源，后将设备断电。
- 将采集卡拔离计算机后，请间隔一定时间再重新插入。拔插过快有可能造成计算机系统配置驱动异常，此时请重新拔插采集卡。
- 本设备属于精密电子仪器，请注意妥善保管，注意防尘防潮与防静电。
- 禁止用户自行打开设备外壳，一经打开即失去保修资格，出现的故障与其他后果由用户自行承担。
- MPS-150102 自出厂之日起一年内，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，凭保修卡免费维修。因违反操作规定和使用要求而造成损坏的，需交纳器件维修费。
- 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。