USB 4-Channel 16-bit High Dynamic Range Signal Acquisition Card User Manual

Ver 2.0.0

Chapter 1: Product Overview

USB 4-Channel 16-bit High Dynamic Range Signal Acquisition Card

1. Product Overview

MPS-HD1604 is a 4-channel 16-bit high dynamic range data acquisition card based on the USB bus, suitable for high-precision signal measurement, portable data acquisition, vibration and audio signal analysis, and other fields.

MPS-HD1604 features 4 synchronous voltage acquisition channels with a ±5V range. The sampling rate is adjustable in 7 levels within the range of 1kSPS to 64kSPS, supporting continuous uninterrupted signal acquisition and real-time data transmission. MPS-HD1604 is characterized by low noise: the background noise can be as low as 50μVrms (at ±5V full scale) at 64kSPS, and even as low as 10μVrms at 1kSPS, with an effective resolution of up to 19.0 bits.

MPS-HD1604 offers six optional channel input modes, compatible with differential/single-ended, high input impedance/low input impedance, DC coupling/AC coupling, IEPE (ICP) constant current power supply, and other input characteristics. It can adapt to most application scenarios and supports customizing different input modes for each channel, making it particularly suitable for mixed measurement of multiple types of signals.

MPS-HD1604 connects to a computer via the USB bus, supporting plug-and-play and hot-swapping. The device is compatible with the full range of Windows operating systems,

providing a universal DLL driver library, as well as programming reference examples in various languages including LabVIEW, VB, C#, Python, Qt, and MATLAB.

In addition, MPS-HD1604 comes with a free multi-functional acquisition software, which can directly implement basic functions such as waveform display, signal recording, waveform playback, and spectrum analysis, as well as numerous advanced auxiliary functions including software filtering, edge triggering, frequency calculation, resampling, signal fitting, and sound playback. This significantly lowers the threshold for use and helps users achieve rapid measurements.

2. Performance Specifications

2.1 Communication Bus

- USB2.0 high-speed bus (compatible with 3.0 and higher versions)
- USB bus powered
- Supports plug-and-play and hot-swapping

2.2 Input Channels

- Interface type: BNC female connector
- CH1-CH4 channels: Four signal input ports
- GND/IO channel negative: Common ground interface
- GND/IO channel positive: Multi-function multiplexed IO port
- GND/CLK channel negative: Common ground interface
- GND/CLK channel positive: Clock signal input/output port

2.3 Input Modes

The MPS-HD series acquisition cards support six signal input modes. Users can specify the input mode for each channel when ordering the product. The six input modes are as follows:

S1 Mode (DC Single-ended Low Input Impedance Mode):

Coupling method: DC coupling
Input type: Single-ended input
Input impedance: 50kΩ between input positive and ground [1]; input negative connected to ground
Input range: ±5V [2]
Input withstand voltage: ±25V
Constant current supply: None

S2 Mode (DC Single-ended High Input Impedance Mode):

Coupling method: DC coupling
Input type: Single-ended input
Input impedance: 1MΩ between input positive and ground [3]; input negative connected to ground

Input range: ±5V
Input withstand voltage: ±25V
Constant current supply: None

S3 Mode (DC Differential Mode):
Coupling method: DC coupling
Input type: Differential input
Input impedance: 1MΩ between input positive and ground; 1MΩ between input negative and ground
Input range: ±5V
Input withstand voltage: ±25V
Constant current supply: None

S4 Mode (AC Single-ended Mode):
Coupling method: AC coupling (DC-blocking)
Input type: Single-ended input
Input impedance: 10μF||50kΩ between input positive and ground; input negative connected to ground
Input range: ±5V
Input withstand voltage: ±25V
Constant current supply: None

S5 Mode (Constant Current Supply Mode):
Coupling method: DC coupling
Input type: Single-ended input
Input impedance: 1MΩ between input positive and ground; input negative connected to ground
Input range: ±5V
Input withstand voltage: ±25V
Constant current supply: 4mA [4]; open-circuit drive voltage 24V

S6 Mode (IEPE Mode):
Coupling method: AC coupling (DC-blocking)
Input type: Single-ended input
Input impedance: 10μF||50kΩ between input positive and ground; input negative connected to ground
Input range: ±5V
Input withstand voltage: ±25V
Constant current supply: 4mA; open-circuit drive voltage 24V

[1] *Exact rated value of input internal resistance: 49.7kΩ, error 0.1%.*
[2] *Exact rated value of measurement range: 5.2V, error 0.1%*
[3] *Exact rated value of input internal resistance: 1MΩ, error 0.1%.*
[4] *Exact rated value of constant current supply: 4.15mA, error 1%*

2.4 Sampling Rate

▪ Seven adjustable levels via software: 64k, 32k, 16k, 8k, 4k, 2k, 1k (SPS)

2.5 Resolution

▪ 64k SPS: Effective resolution 16.0bit; Sampling noise Vpp < 350μV, VRMS < 50μV
▪ 32k SPS: Effective resolution 16.7bit; Sampling noise Vpp < 240μV, VRMS < 35μV
▪ 16k SPS: Effective resolution 17.3bit; Sampling noise Vpp < 180μV, VRMS < 26μV
▪ 8k SPS: Effective resolution 18.1bit; Sampling noise Vpp < 125μV, VRMS < 18μV
▪ 4k SPS: Effective resolution 18.4bit; Sampling noise Vpp < 100μV, VRMS < 15μV
▪ 2k SPS: Effective resolution 18.7bit; Sampling noise Vpp < 80μV, VRMS < 12μV
▪ 1k SPS: Effective resolution 19.0bit; Sampling noise Vpp < 68μV, VRMS < 10μV

2.6 Bandwidth

▪ 1k - 64k SPS: Built-in 32kHz anti-aliasing filter, signal passband high cut-off frequency 31.2kHz
▪ S4 or S6 mode: Built-in DC-blocking high-pass filter, signal passband low cut-off frequency 0.3Hz
▪ S1, S2, S3 or S5 mode: DC coupling, signal passband low cut-off frequency 0Hz

2.7 On-board Buffer

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

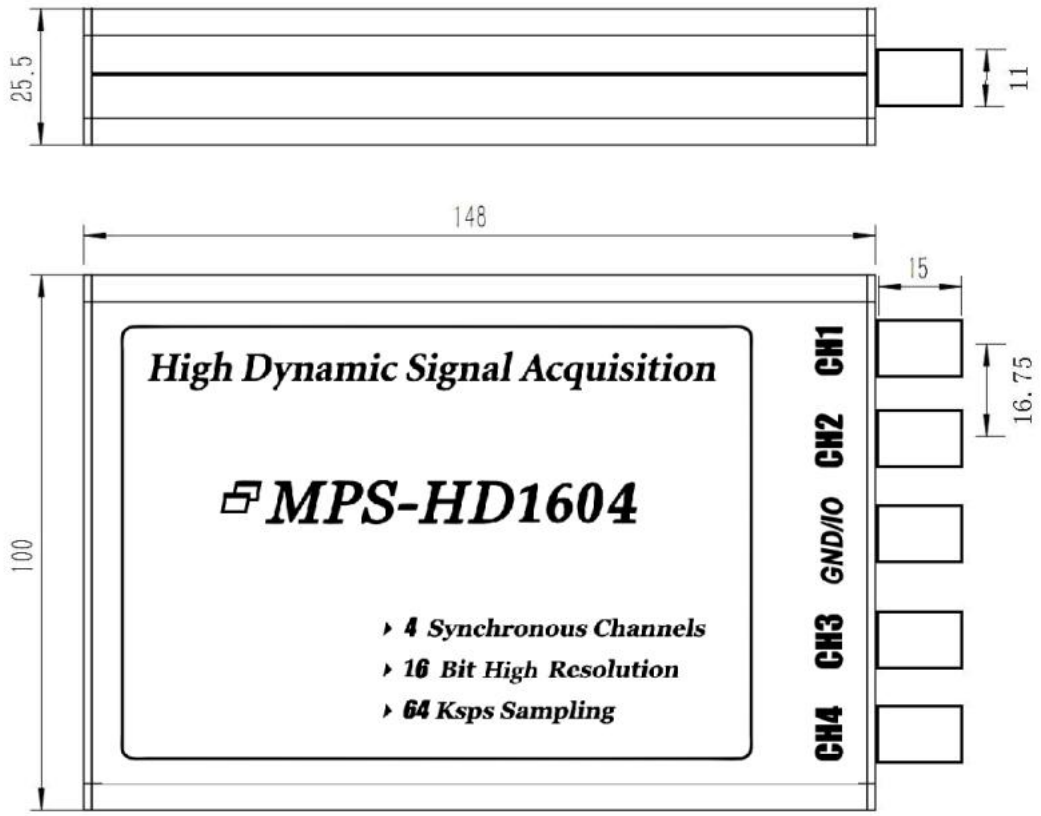
2.8 Operating Temperature

▪ -40℃ ～ 85℃

2.9 Operating Power

▪ Standby power: Current < 240mA, Power < 1.2W
▪ Operating power: Current < 280mA, Power < 1.4W
▪ IEPE power: Current increases by 25mA, power increases by 0.125W for each IEPE sensor connected
▪ Maximum power: Current < 400mA, Power < 2W
▪ DIO 5V output: Output current < 50mA

3. Physical Dimensions



Dimensions: 163 mm × 100 mm × 25.5 mm

4. Application Fields

- High-precision signal acquisition and recording
- Industrial production online monitoring
- Portable signal measurement
- Sound and vibration analysis

5. Software Resources

- Special driver program compatible with all Windows operating systems
- Cross-platform general DLL dynamic link library for system integration and secondary development
- Example programs for multiple programming languages: LabVIEW, VB, C#, Python, Qt, Matlab
- Free multifunctional test software included

6. Accessory List

[1] One MPS-HD1604 signal acquisition card
[2] One high-shielding USB data transmission cable
[3] One warranty card
[4] One certificate of conformity

7. After-sales Service

Three-year free warranty

Chapter 2: Device Installation

1. Interface Description

USB: USB data bus interface. This interface is connected to the computer via a USB Type-B data cable. After successful connection, the device will be recognized by the Windows system and listed in Device Manager. The USB bus also supplies power to the device. Normally, the power supply from the USB bus itself can meet the device requirements. If the USB bus power supply is insufficient, you can use a USB hub with an external power port to provide additional power.

LED1: Device self-test status indicator (LED1), blue when on. Specific status meanings are as follows:

- Steady on after power-on: Device self-test passed, operating normally.
- Fast flashing after power-on: Device self-test abnormal.
- Off after power-on: Unstable device power supply or hardware failure. It is recommended to try replacing the computer host and USB data cable. If ineffective, please contact the after-sales department in a timely manner.

LED2: Acquisition and data transmission status indicator (LED2), white when on. Specific meanings are as follows:

- Steady on: The device is in continuous acquisition state with stable data transmission.
- Off: If the device is in standby state, LED2 off is normal; if the device is in acquisition state and LED2 turns off after being on, the software may not read data in time, causing data overwrite after the hardware data buffer is full. LED2 off indicates that the hardware buffer is in overwriting state.
- Regular flashing: Usually formed when the acquisition software adopts cyclic segmented acquisition mode, i.e., LED2 is on when the software acquires each time; LED2 is off when the software stops acquisition; LED2 will flash regularly with the above cycle.
- Irregular flashing: If LED2 flashes irregularly during continuous software acquisition, it may be a warning of unstable data transmission. If the software reads data too slowly or does not read data from the hardware for a long time due to software operation lag, the hardware data buffer will be overwritten and LED2 will turn off; if the software resumes data reading later, LED2 will turn on again. If the above process repeats, the LED will flash irregularly. When this phenomenon occurs, it indicates that there is a problem with the current computer software operation, which has caused data loss and damaged data integrity. At this time, please try to optimize the software execution efficiency to improve the reading speed, or reduce the total amount of data by lowering the

sampling rate to ensure the integrity of the read data.

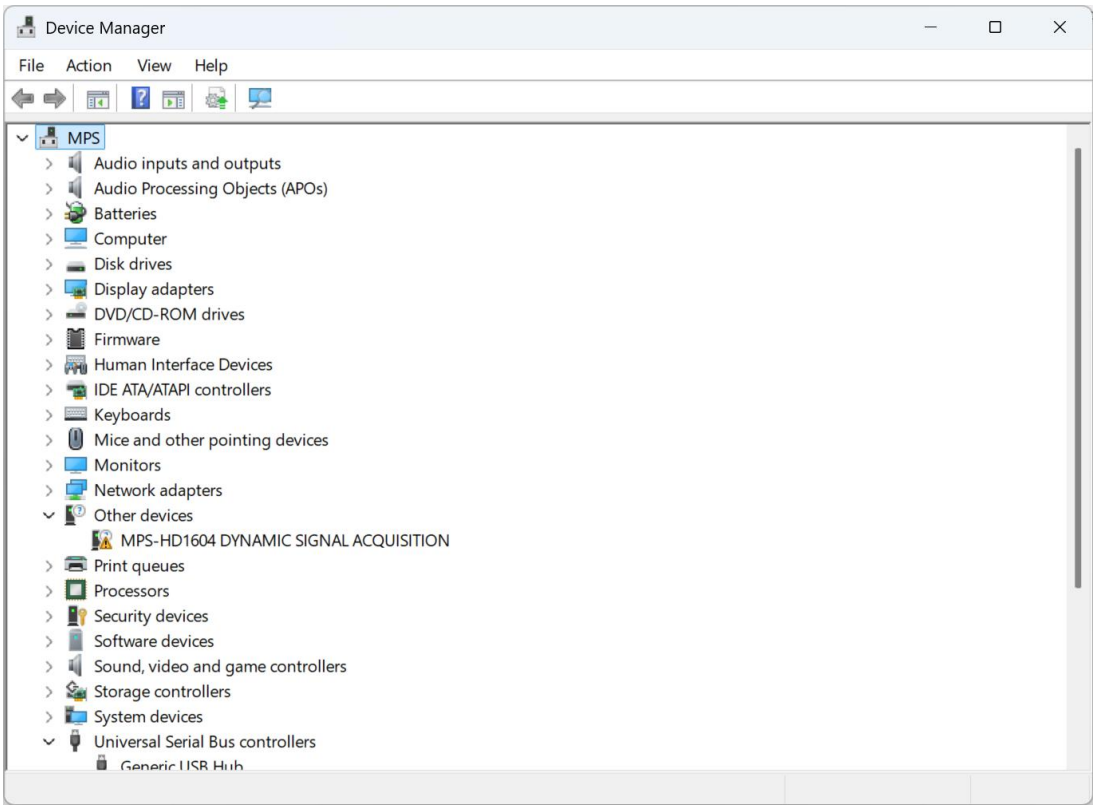
CHx: CH1-CH4 correspond to the signal input terminals of Channel 1 to Channel 4. Each channel is a BNC female connector, with the outer ring as the negative pole and the inner core as the positive pole. For the interface characteristics (such as impedance, coupling method, etc.) in different input modes, please refer to Section 2.3 of Chapter 1 of this manual.

GND/IO: Device signal ground and multiplexed IO port. This port is a BNC female connector, with the outer ring as the device signal ground and the inner core as the multiplexed IO port. The functions are as follows:

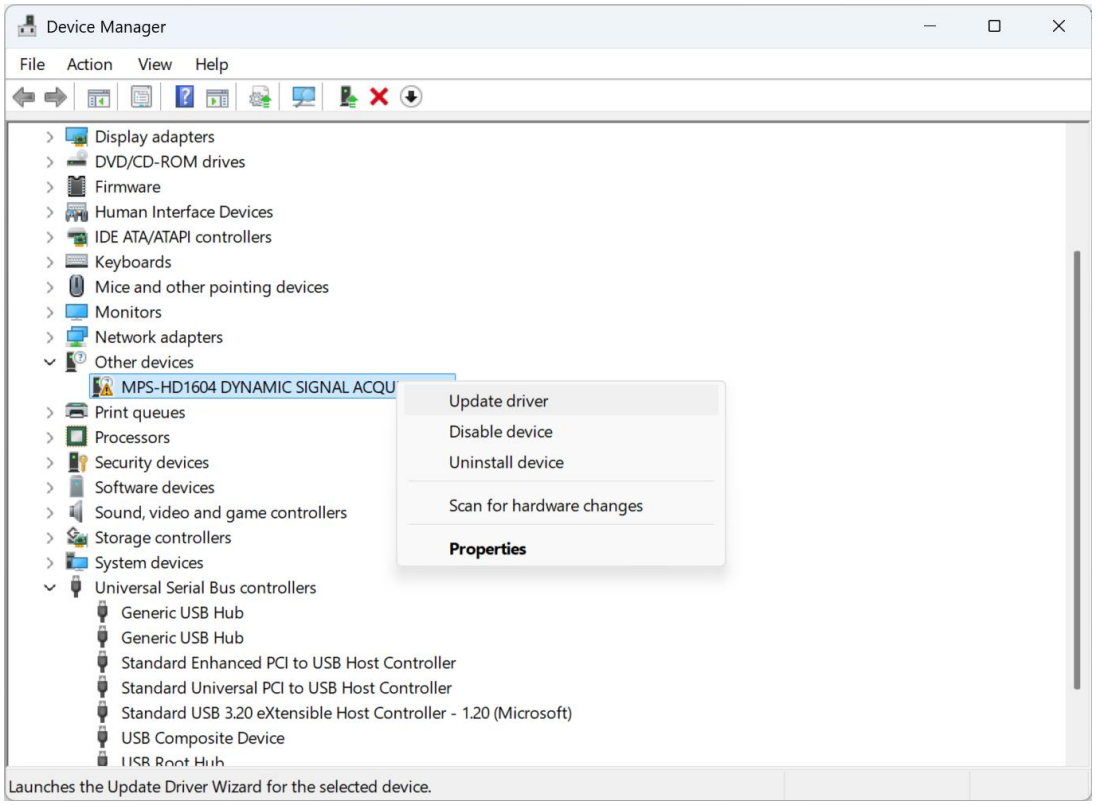
- Signal ground: Used to connect the ground wire of external sensors or signal sources to establish a common ground connection point.
- Multiplexed IO port:
  - Digital output: Configurable to output high level (5V) or low level. It can also be used as a 5V power supply when outputting high level, with a maximum output current of 50mA.
  - Trigger input: Can be used as the input port for external start trigger signal. Inputting a falling edge signal to this port can trigger the device to start acquisition.

2. Driver Installation

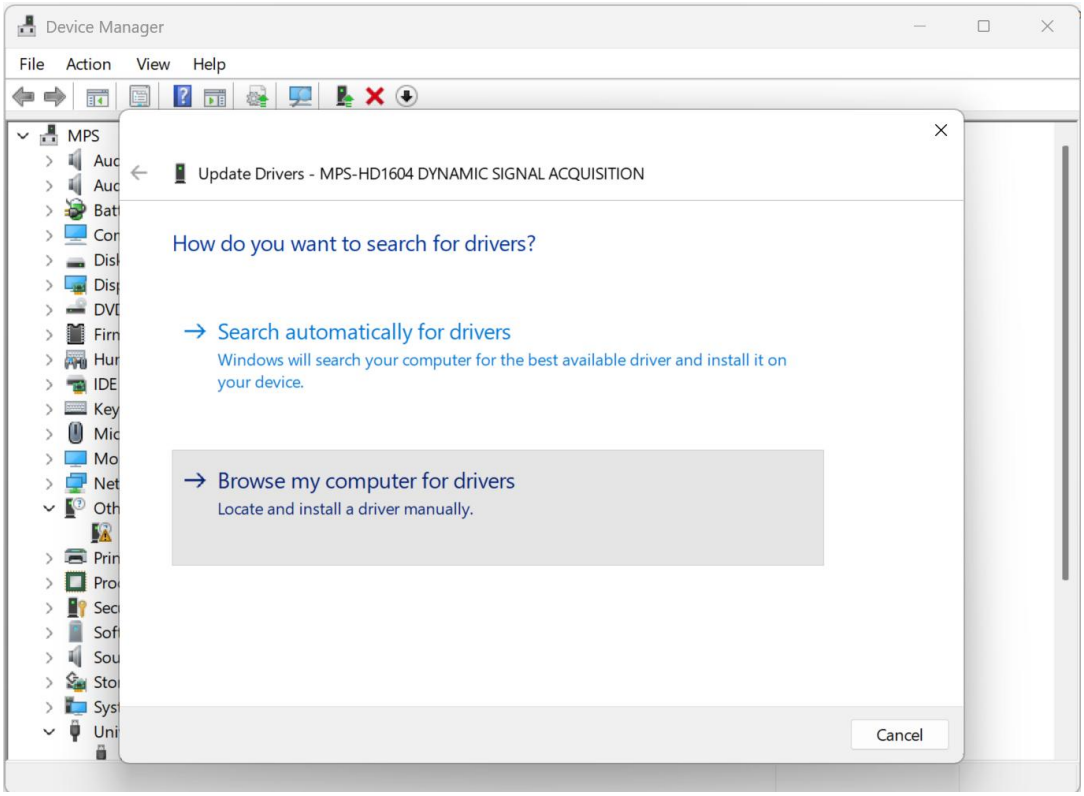
① After connecting the device to the computer USB, right-click on "This PC", select "Device Manager" or click "Properties" - "Management" - "Device Manager" to open the WINDOWS Device Manager, and find the device under "Other devices".



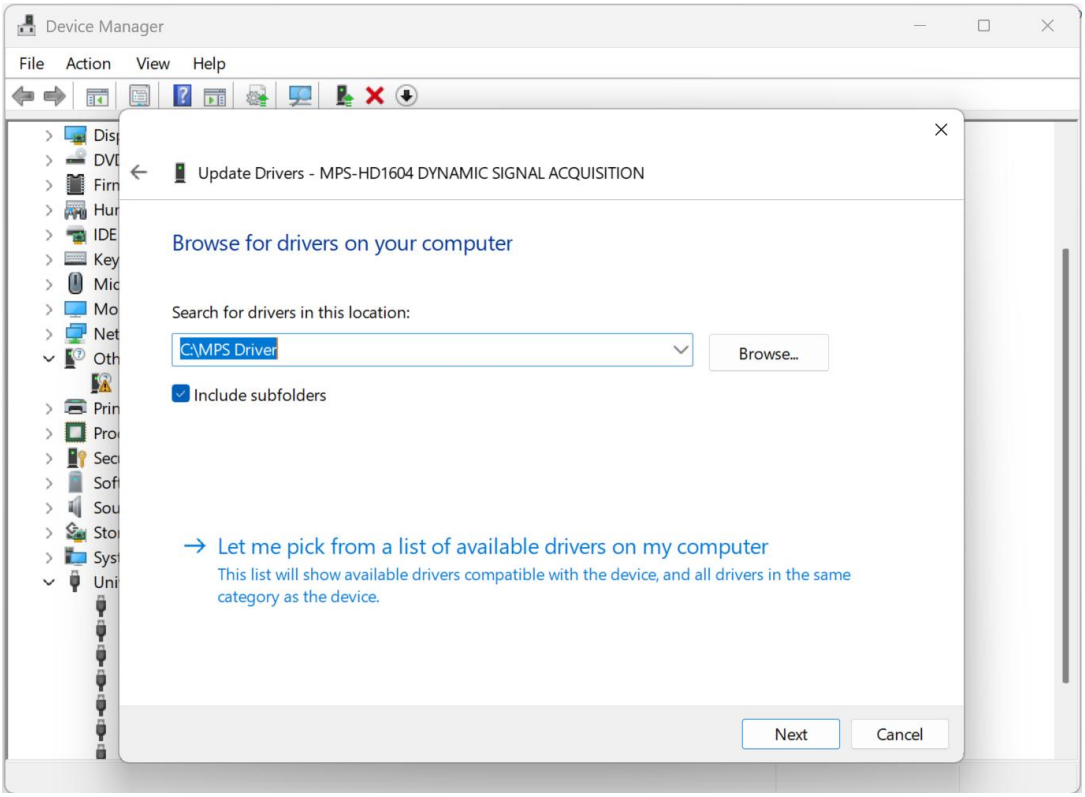
② Right-click on the device name and select "Update driver".



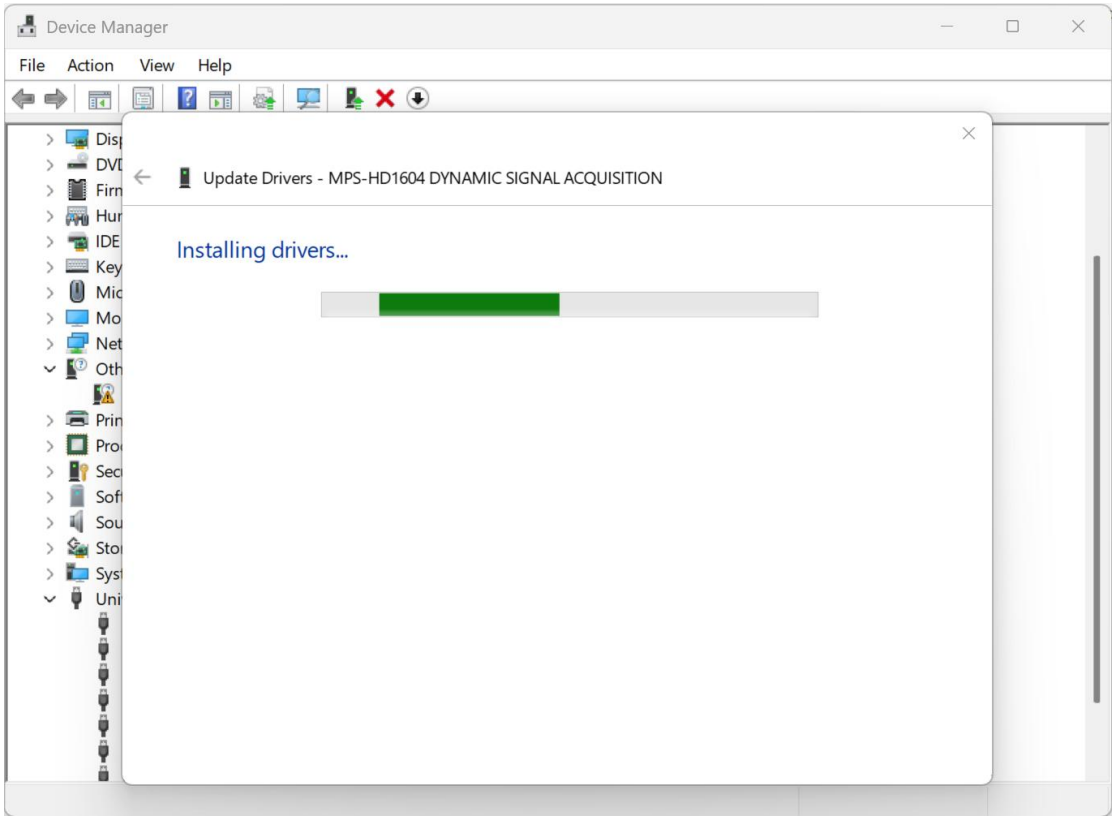
③ In the pop-up driver installation wizard interface, select "Browse my computer for drivers".



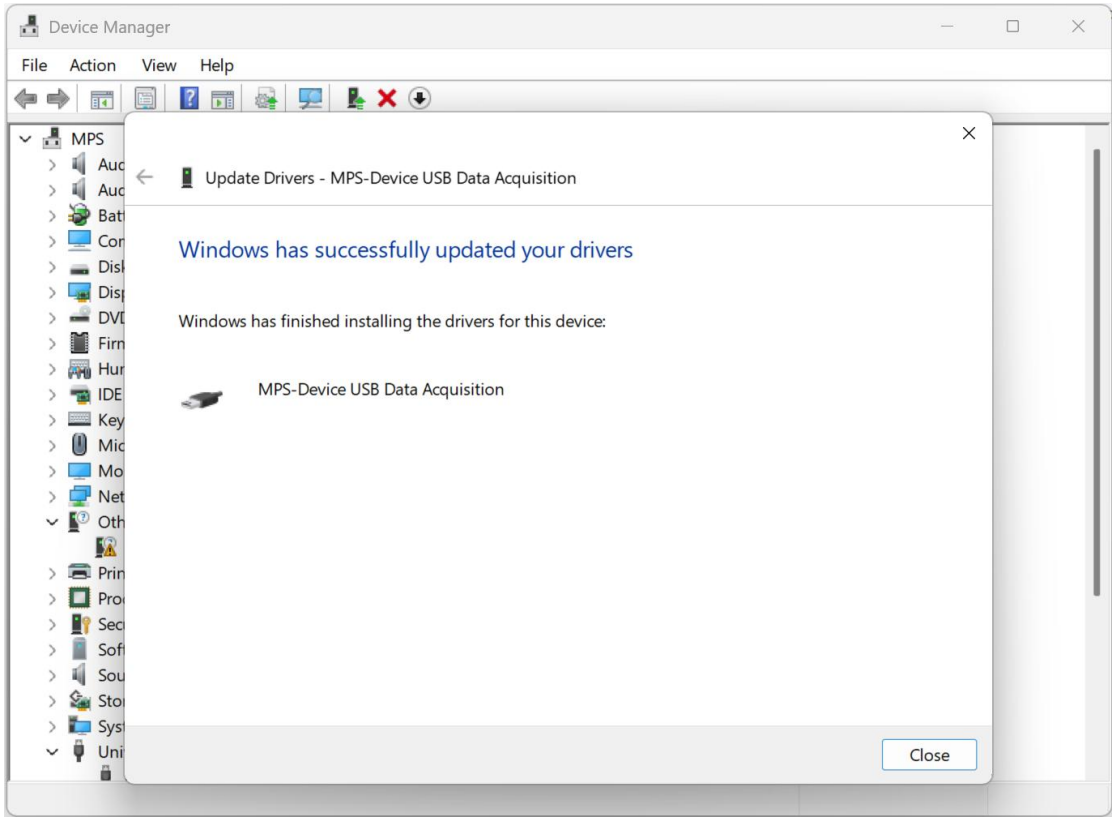
④ Click the "Browse" button and select the folder where the acquisition card driver is located. If the driver is a compressed package, it needs to be decompressed first before operation.



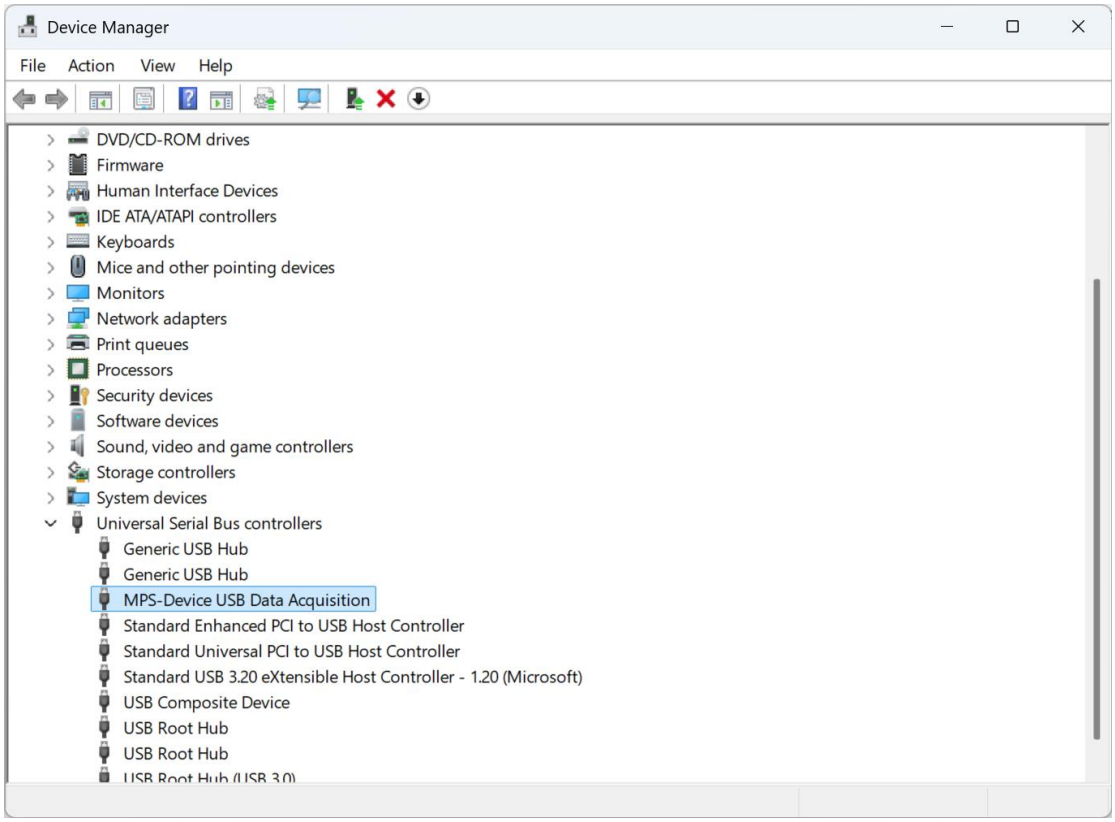
⑤ Click Next to appear the following driver installation interface. If a security prompt to trust the driver appears during the process, check the trust option and click Confirm.



⑥ Driver installation is completed, the following prompt box appears.



⑦ After the driver is installed, the device can be seen in the "Universal Serial Bus controllers" category of Device Manager, named "MPS-Device USB Data Acquisition", indicating that the driver is installed successfully.



3. Signal Connection

MPS-HD1604 has four analog signal input ports, CH1 to CH4, corresponding to Channel 1 to Channel 4 respectively. The input signal connector is a BNC female connector, which can be connected with a signal cable using a BNC male connector.

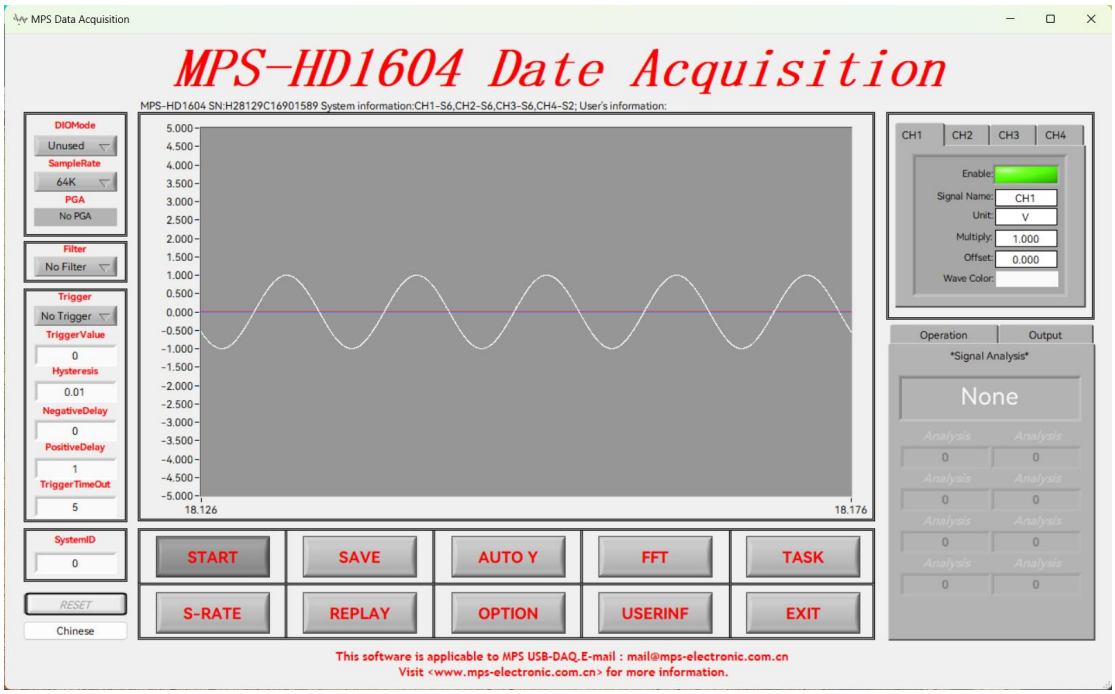
The input mode of each channel of MPS-HD1604 can be configured before delivery, with six modes optional (see Section 2.3 of Chapter 1). Different input modes can match different types of signal sources. If you need to change the mode after delivery, you can contact the after-sales department for consultation.

If a channel of MPS-HD1604 is in S1, S2 or S4 mode, the channel is single-ended input, the positive pole of the input connector is the signal input, and the negative pole is the signal common ground. This channel can be connected to an external single-ended signal source, with the positive pole of the connector connected to the signal line of the single-ended signal source, and the negative pole of the connector connected to the signal ground of the single-ended signal source.

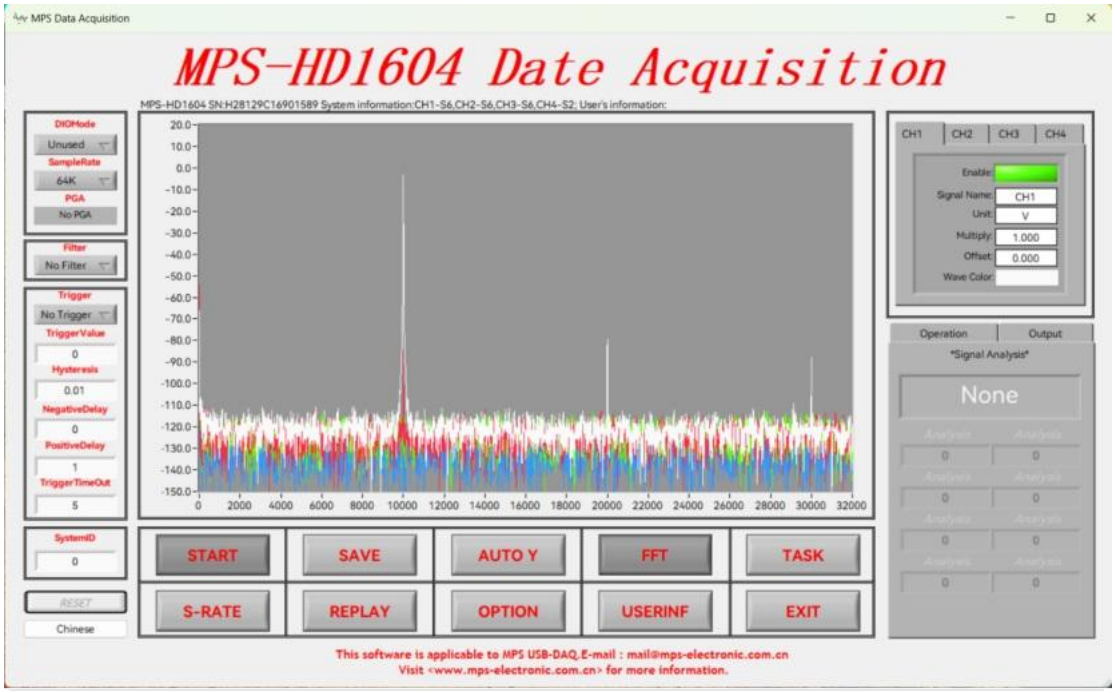
If a channel of MPS-HD1604 is in S3 mode, the channel is differential input and needs to be used in conjunction with the negative pole (signal common ground) of the device's GND/IO port, which can be connected to both external differential signal sources and single-ended signal sources. When connecting a differential signal source, the positive and negative poles of the differential channel input connector are connected to the positive and negative poles of the differential signal source respectively, and the negative pole of the GND/IO port is connected to the independent ground of the differential signal source for common ground; when connecting a single-ended signal source, the channel input positive pole is connected to the signal line of the single-ended signal source, the channel input negative pole is connected to the ground of the single-ended signal source, and the negative pole of the GND/IO port is also connected to the ground of the signal source for common ground.

If a channel of MPS-HD1604 is in S5 or S6 mode, the channel is single-ended input, the positive pole of the input connector is the signal input and also provides 4mA external constant current output; the negative pole of the connector is the signal common ground. This channel can be connected to external IEPE sensors (S6 mode), resistance sensors with resistance within 1250 ohms (S5 mode) and some specific sensors. Other types of sensors generally cannot be connected, and forced connection may cause the sensor to fail to work normally and risk damaging the sensor. When using S6 mode, the channel can be directly connected to the IEPE sensor without an additional signal conditioner in between.

4. Function Test



Waveform Graph of MPS Test Software (English Interface)



Spectrum of MPS Test Software (Chinese Interface)

1. Download and decompress MPS Data Acquisition Installer.zip, execute "Setup.exe" to install the test software.

2. Connect the MPS-HD1604 signal acquisition card to the computer through the USB interface. For the first connection, you need to install the hardware driver according to the steps in Section 2.3.

3. Open the Windows Start menu and open the software named "MPS Data Acquisition".
4. The sampling rate can be set at "SampleRate" in the upper left corner, with seven options from 1k to 64k.
5. Click "START", and the collected signal waveform curve can be seen from the waveform graph. When no signal source is connected, the curve generally appears as a straight line close to 0. If an error prompt appears after clicking "START", please check whether the hardware is connected and whether the driver is installed correctly.
6. After connecting the sensor or signal source to the acquisition card, the waveform curve changing with the signal source can be observed. The white curve corresponds to CH1, the red curve to CH2, the green curve to CH3, and the blue curve to CH4.
7. Click "AUTO Y" to make the Y-axis display range automatically match the current curve. You can also change the display range by directly modifying the Y-axis coordinate boundary values.
8. Additional functions such as digital filtering, software triggering, signal recording, spectrum analysis, scheduled tasks, variable sampling rate, recording playback, signal analysis, operation and output are available in the software. Click "Chinese" in the lower left corner to switch to the Chinese software interface.
9. After the function test is completed, click "EXIT" to exit the software.
10. Disconnect the signal source and pull out the USB plug connected to the computer, the test is completed.

Chapter 3: User Programming

1. Dynamic Link Library (DLL)

The MPS-HD series uses DLL (Dynamic Link Library) for programming driving. DLL has no mandatory association with specific programming languages and compilers. As long as the agreed DLL interface specifications and calling methods are followed, the DLL can be called in various languages.

Taking calling DLL in VC, VB, LabVIEW and other languages as examples, the specific calling methods are as follows:

- Calling DLL in VC:

```
typedef void ( * FUNC )(void);                    //Define a function pointer
FUNC Func;                                        //Define a function pointer variable
HINSTANCE hDLL=LoadLibrary("DllTest.dll"); //Load dll
Func=(FUNC)GetProcAddress(hDLL,"FuncInDLL"); //Get function address
Func();                                           //Call the function in dll
FreeLibrary(hDLL);                                //Release DLL
```

- Calling DLL in VB:

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

Public (optional) is used to declare functions that can be used in all procedures in all modules; Private (optional) is used to declare functions that can only be used in the module containing the declaration.

Name (required) Any valid function name. The entry points of the dynamic link library are case-sensitive.

Libname (required) The name of the dynamic link library or code resource containing the declared function.

Alias (optional) indicates that the function to be called has another name in the dynamic link library (DLL). This parameter can be used when the external function name is the same as a function name. It can also be used when the function of the dynamic link library has the same name as a public variable, constant or any other procedure in the same scope. Alias can also be used if a character in the dynamic link library function does not conform to the naming convention of the dynamic link library.

Aliasname (optional) Dynamic link library. If the first character is not a number sign (#), then aliasname is the name of the function entry point in the dynamic link library. If the first character is (#), the subsequent characters must specify the

sequence number of the function entry point.
Arglist (optional) represents the variable list of parameters that need to be passed when calling the function.

Type (optional) The data type of the Function return value; can be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (only variable length supported) or Variant, user-defined type, or object type.

Arglist (optional) represents the variable list of parameters that need to be passed when calling the function.

Type (optional) The data type of the Function return value; can be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (only variable length supported) or Variant, user-defined type, or object type.

The syntax of the arglist parameter is as follows:
[Optional] [ByVal | ByRef] [ParaArray] varname [()] [As type]

Optional (optional) indicates that the parameter is not required. If this option is used, all subsequent parameters in arglist must be optional and must all be declared with the Optional keyword. If ParamArray is used, no parameter can use Optional.
ByVal (optional) indicates that the parameter is passed by value.
ByRef (optional) indicates that the parameter is passed by address.

- Calling DLL in LabVIEW:

In LabVIEW, calling DLL is done through the CLF node. The so-called CLF node (Call Library Function node) refers to the ability to call DLL encapsulated in other languages in LabVIEW. The CLF node is located in the Advanced sub-template in the LabVIEW function template. Its configuration process is as follows:

- Select "Configure" in the right-click menu of the CLF node to pop up the CLF node configuration dialog box;
- Click the "Browse" button, find the DLL file you need to use in the subsequent pop-up select DLL file dialog box. At this time, LabVIEW will automatically load the selected DLL file and detect the functions contained in the DLL file. However, the parameters in the function and the data types of the parameters need to be manually set by the user according to the input and output parameters of the function. Therefore, when calling a DLL file, the user is required to have a detailed understanding of the DLL file.
- Select the required API function contained in the dynamic link library in the

FunctionName drop-down list box;

- Select the StdCall(WINAPI) and C options in the Calling Convention drop-down menu. If the user selects a Windows API function, select the StdCall(WINAPI) option; if the user selects a non-Windows API function in the DLL, select the C option;
- Set the return parameters of the function. The type of function parameters must correspond to the function parameter types defined by the function itself in the DLL. If they do not correspond, the function will have data errors and forced type conversion;
- Set the input parameters and data types according to the function prototype of the selected function. Click the Add a Parameter button to add a new input parameter.

❖ ***For complete code in C#, VB, LabVIEW, Python, Qt, Matlab and other languages, please refer to the official website examples.***

2. Driver Functions and Parameters

The driver program file of the MPS-HD series acquisition card is **"MPS Driver.dll"**, which contains the following 11 functions:

#define HANDLE int
Note: If the HANDLE data type is not predefined, it can be defined as int type.

- HANDLE MPS_OpenDevice (int DeviceNumber)

  HANDLE The MPS_OpenDevice function is used to open the device and obtain the operation handle of the device. The function returns the operation handle of the device after successful execution, and returns -1 if the execution fails.

  int DeviceNumber:This parameter is the serial number of the device to be opened, with an effective range of 0 to 9.

  The MPS-Device driver supports a maximum of 10 devices connected to a single computer. When multiple devices are connected at the same time, the Windows system will automatically assign device serial numbers (DeviceNumber). The first device recognized by the system is assigned serial number 0, and subsequent devices are assigned serial numbers 1, 2, 3... in the order of recognition. The maximum assigned serial number is 9. The serial number assignment is based on the device recognition order and has nothing to do with the physical connection port.

  When calling the MPS_OpenDevice function, the operation handle of the corresponding device can be obtained by specifying different DeviceNumber parameter values.
  For the first connected device, pass parameter value 0 to obtain its device handle:
  ```
  HANDLE hDevice0 = MPS_OpenDevice(0);  // Get the handle of the first device
  ```
  When the second device is connected, pass parameter value 1 to obtain the device handle:
  ```
  HANDLE hDevice1 = MPS_OpenDevice(1);  // Get the handle of the second device
  ```
  .....
  For the tenth device, pass parameter value 9.

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

The MPS_Configure function is used to set the acquisition parameters of the device. After the function is executed successfully, it will set the sampling rate and IO port mode of the device. This function must be called when the device is in standby state. The function returns 1 if executed successfully, and 0 if failed.

int SampleRate: This parameter is used to set the sampling rate. MPS-HD1604 has a total of seven sampling rates that can be set. The setting rules are as follows:

• When the value is 64000 or greater than 64000, the sampling rate is set to 64000 points per second.
• When the value is 32000 or less than 64000 and greater than 32000, the sampling rate is set to 32000 points per second.
• When the value is 16000 or less than 32000 and greater than 16000, the sampling rate is set to 16000 points per second.
• When the value is 8000 or less than 16000 and greater than 8000, the sampling rate is set to 8000 points per second.
• When the value is 4000 or less than 8000 and greater than 4000, the sampling rate is set to 4000 points per second.
• When the value is 2000 or less than 4000 and greater than 2000, the sampling rate is set to 2000 points per second.
• When the value is 1000 or less than 2000, the sampling rate is set to 1000 points per second.

int DIOMode:This parameter is used to set the mode of the IO port, and its value is 0, 1, or 2.

- DIOMode = 0: Set the IO port to idle mode, making the IO port present a high internal resistance low level state (about 0.5V).
- DIOMode = 1: Set the IO port to high level output mode, outputting 5V voltage externally (output current < 50mA), which can be used as digital high level output or as an external 5V power supply output;
- DIOMode = 2: Set the IO port to external start trigger mode. In this mode, the IO port is in a digital high level input state with internal weak pull-up. In external start trigger mode, the device can be triggered to start acquisition by inputting a level falling edge to the IO port. When multiple MPS-HD devices are used at the same time, this mode can also be used to achieve synchronous start of multiple devices.

Among the above three modes, the idle mode and high level output mode will remain as long as they are not reset after setting; the external start trigger mode is only valid in the first standby state after setting, and the IO port will automatically switch to idle mode after starting acquisition.

Supplementary explanation about external start trigger mode:

1. In external start trigger mode, when the input state of the IO port changes from high level (2V-5V) to low level (0V-1.5V) (for example, shorting the IO port to GND), a falling edge input will be generated. If the device is in an idle state at this time, the falling edge will trigger the device to enter the acquisition state.
2. Starting acquisition by falling edge trigger is equivalent to starting acquisition by the upper computer calling the MPS_Start function, both of which will make the device enter the acquisition state.
3. In external start trigger mode, if acquisition is started by calling the MPS_Start function, when the device switches from standby state to acquisition state, the IO port will automatically change from external start mode (digital high level state with weak pull-up) to idle mode (low level state), thus generating a falling edge. If the IO port of this device is connected to the IO ports of other (one or more) devices in external start trigger mode, this falling edge will trigger other devices to start acquisition synchronously.
4. In program design, if the external trigger mode is used to start acquisition, since the software cannot detect the time point of the external trigger in real time, the MPS_DataIn function must be called in advance to start reading data, and its internal waiting mechanism is relied on to judge whether the trigger is successful.
    (1) If the trigger is successful, the MPS_DataIn function will acquire data and return after starting acquisition;
    (2) If no trigger is received but the MPS_TimeOut timeout parameter is set, the function will return a failure value after timeout;
    (3) If no trigger is received and no timeout is set, the function will wait continuously until MPS_Start is called to start acquisition;
    (4) If hardware connection disconnection or restart occurs during the waiting period, the function will return a failure value.
5. In external start trigger mode, if the device receives an external falling edge or MPS_Start function command, it will start acquisition and switch the IO port to idle mode; if the device receives an MPS_Stop function command, it will remain in standby state, but the IO port will still change from external start mode to idle mode; if the device receives an MPS_Configure command, it will immediately set the IO port to the newly specified mode. In any of the above cases, the IO port will no longer respond to external falling edge signals after exiting the external start trigger mode.

HANDLE DeviceHandle: Operation handle of the target device.

- int MPS_Start (HANDLE DeviceHandle)

The MPS_Start function is used to start data acquisition of the device. After the device is powered on, it enters the standby state by default. The software can start device acquisition by calling the MPS_Start function. The function returns 1 on success and 0 on failure.

HANDLE DeviceHandle: Operation handle of the target device.

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

The MPS_DataIn function is used to read the sample data collected by the device. After the function is executed successfully, a batch of sample data will be read from the device. The function returns 1 on success and 0 on failure.

When the device samples the input signal, the sample data will be temporarily stored in the internal hardware buffer (DAQ Buffer) and wait to be read by the MPS_DataIn function. The hardware buffer follows the First-In-First-Out (FIFO) principle. New samples are stored at the end of the buffer, and the MPS_DataIn function reads from the beginning of the buffer. The buffer space is automatically reclaimed after data reading, and subsequent data is moved forward to ensure that the data obtained by multiple calls to the MPS_DataIn function during the same acquisition process is continuous and uninterrupted. A complete continuous data stream can be constructed by cyclically calling the MPS_DataIn function and splicing the data.

The execution logic of the MPS_DataIn function is as follows:

- If the device is in the acquisition state and there are enough samples in the buffer when the function is running, the function reads the data immediately and returns success.
- If the device is in the acquisition state but the number of samples in the buffer is insufficient when the function is running, the function waits until the device collects enough samples and then returns success.
- If the device is in the idle state and acquisition is not started when the function is running, the function will wait continuously and cannot return. At this time, the MPS_Start function can be called to start acquisition, and the MPS_DataIn function can obtain data normally and return success after starting acquisition.
- If the timeout parameter is set through the MPS_TimeOut function before the function runs, the MPS_DataIn function will automatically time during the waiting process, and will be forced to end and return a failure value after timeout.
- If the device is not connected to the computer when the function is called, the function execution returns a failure value.
- If the connection between the device and the computer is interrupted during the function operation (such as USB disconnection, device power off, restart, etc.), the function execution ends and returns a failure value.

In the external start trigger mode:

- The MPS_DataIn function needs to be called when the device is idle, and the function will enter the state of waiting for an external trigger signal.
- After receiving the external trigger signal, the MPS_DataIn function will read the sample data normally and return success.
- If the trigger signal is not received for a long time, the MPS_Start function can be called to start acquisition manually to make the MPS_DataIn function terminate the waiting.
- The MPS_TimeOut timeout parameter can also be set to make the MPS_DataIn function automatically exit the waiting state after reaching the preset timeout time.

Note: Due to the limited capacity of the on-board hardware buffer, there is an upper limit on the number of samples it can store. When newly collected samples are continuously written and old samples cannot be read in time, once the buffer is full, the earliest stored data will be discarded according to the FIFO principle, resulting in data loss. At this time, the data read by the MPS_DataIn function will not be continuous with the previous data. Therefore, in the design of acquisition software:

- The calling frequency of the MPS_DataIn function must not be lower than the data acquisition rate.
- Avoid adding artificial delay or waiting operations between two MPS_DataIn calls.
- Try to avoid complex operations, real-time drawing or hard disk reading and writing between two MPS_DataIn calls.
- It is recommended to set the process of the acquisition software to the highest priority to prevent operation freezes caused by system resource competition.

int *DataBuffer: This parameter is used to pass a pointer to the user-defined software buffer for storing the data read from the hardware.

When the software reads data through the MPS_DataIn function, it needs to store the data in a data storage area that the program can directly access. Therefore, before calling the MPS_DataIn function, the program needs to pre-establish a data buffer. The buffer should be defined as a one-dimensional array of 32-bit signed integers (int32_t) with a length greater than or equal to SampleNumber, and take the pointer to the first address of the array and assign it to the DataBuffer parameter. This buffer is independent of the device's internal hardware buffer (DAQ Buffer) and is a separate software buffer.

The data written to DataBuffer is a 32-bit signed integer. If compiling in a 64-bit environment, be sure to define it as a 32-bit integer (int32_t) data type. Each element of DataBuffer corresponds to a sampling point data, and the sampling point data is cyclically distributed according to the corresponding channel and

time order. The distribution method is:

If there are a total of N channels, then DataBuffer[(m-1)×N + (n–1)] corresponds to the m-th sampling point data of the n-th channel (CHn) (where n ≤ N; (m-1)×N +(n–1) ≤ SampleNumber - 1). That is:

DataBuffer[0] corresponds to the first sampling point data of CH1, DataBuffer[1] corresponds to the first sampling point data of CH2, ... DataBuffer[n-1] corresponds to the first sampling point data of CHn;
DataBuffer[N] corresponds to the second sampling point data of CH1, DataBuffer[N+1] corresponds to the second sampling point data of CH2...DataBuffer[N+(n-1)] corresponds to the second sampling point data of CHn;
DataBuffer[2×N] corresponds to the third sampling point data of CH1, DataBuffer[2×N+1] corresponds to the third sampling point data of CH2...DataBuffer[2×N+(n-1)] corresponds to the third sampling point data of CHn;
...
DataBuffer[(m-1)×N] corresponds to the m-th sampling point data of CH1, DataBuffer[(m-1)×N+1] corresponds to the m-th sampling point data of CH2...DataBuffer[(m-1)×N+(n-1)] corresponds to the m-th sampling point data of CHn.

Take the case of four channels (N=4) as an example:

DataBuffer[0] corresponds to the first sampling point data of CH1, DataBuffer[1] corresponds to the first sampling point data of CH2, DataBuffer[2] corresponds to the first sampling point data of CH3, DataBuffer[3] corresponds to the first sampling point data of CH4; DataBuffer[4] corresponds to the second sampling point data of CH1, DataBuffer[5] corresponds to the second sampling point data of CH2, DataBuffer[6] corresponds to the second sampling point data of CH3, DataBuffer[7] corresponds to the second sampling point data of CH4... and so on, DataBuffer[(m-1)×4+(n-1)] corresponds to the m-th sampling point data of CHn.

Each sampling point data is a signed 32-bit integer data, and its value is proportional to the voltage value of the measured signal. The corresponding relationship is: ***Voltage[i] = ((double)DataBuffer[i]/8388608) * 5.2***, that is: the voltage value is equal to the integer divided by 8388608 (where 8388608 is the maximum range of 24-bit signed integers, i.e., 2^23), and then multiplied by 5.2V (the range reference value of MPS-HD1604). In the calculation, it is recommended to convert the integer to a 32-bit floating point number (float) or 64-bit floating point number (double) before performing the division operation to prevent precision loss in the integer division operation.

After the MPS_DataIn function is executed successfully, the read data will be written to the buffer pointed to by DataBuffer. The number of written data is determined by the SampleNumber parameter. Therefore, the number of elements in the DataBuffer buffer must be greater than or equal to SampleNumber.

If the MPS_DataIn function fails to execute, the data in DataBuffer is invalid.

int SampleNumber: This parameter is used to set the number of samples read by the MPS_DataIn function in one execution.

In the data reading of the MPS-HD series, SampleNumber is the total number of samples of all channels, and the number of samples corresponding to each channel is SampleNumber divided by the number of channels. The minimum value of SampleNumber is 128, the maximum value is 1024000, and the value must be an integer multiple of 128. If the value is not an integer multiple of 128, the function will automatically configure it to the largest integer multiple of 128 less than the number.

SampleNumber must be less than or equal to the size of the DataBuffer buffer. If it exceeds the range of the DataBuffer buffer, it may cause memory out of bounds and lead to program errors.

The acquisition time corresponding to these samples can be calculated by "Acquisition Time = SampleNumber / (Number of Channels × Sampling Rate)". For example, SampleNumber = 25600, Number of Channels = 4, Sampling Rate = 64000 samples/second, the acquisition time of 0.1 second can be calculated by "25600samples / (4 channels × 64000 samples/second) = 0.1 second". Therefore, in the program design of continuously calling the MPS_DataIn function for acquisition, if 25600 samples are read each time, the average time interval for the program to call the MPS_DataIn function must not exceed 0.1 second. Otherwise, some data cannot be read in time for each reading and accumulate in the hardware buffer. If the hardware buffer is full, new data will overwrite old data, resulting in data loss and affecting data continuity.

In programs that require continuous acquisition and real-time refresh of the display interface, the acquisition time corresponding to SampleNumber will affect the refresh frame rate of the software display. If the acquisition time is long, the single execution time of MPS_DataIn increases, resulting in a lower cycle frequency, which may slow down the program response and bring a stuck operation experience to software users. On the other hand, if the acquisition time is too short, the loop execution is too frequent, which will cause the interface to refresh too fast, reduce program efficiency and increase CPU usage. It is recommended to set a reasonable SampleNumber value to control the loop

execution frequency at 30-50 times per second, match the smooth frame rate of human-computer interaction, and avoid wasting CPU resources at the same time.

In programs for analyzing segmented data (such as FFT spectrum analysis), you can also consider directly setting a larger SampleNumber value to read enough data at one time for analysis, thereby simplifying the program and reducing programming difficulty.

HANDLE DeviceHandle: Operation handle of the target device.

- int MPS_Stop (HANDLE DeviceHandle)

The MPS_Stop function is used to stop the acquisition of the device. When the device is in the acquisition state, calling this function can make the device stop acquisition, enter the standby state, and clear all unread data in the hardware buffer; if the device is already in the standby state, calling this function is invalid. The function returns 1 on success and 0 on failure.

When a continuous acquisition process ends, the software should call the MPS_Stop function to make the device enter the standby state and prepare for the next start of acquisition. If the MPS_Stop function is not called at the end, the device will continue to collect and cyclically overwrite the hardware buffer, resulting in reading expired data in the buffer instead of real-time data during the next acquisition.

In addition, setting parameters and obtaining information also require the device to be in the standby state. To avoid uncertain device status, it is recommended to call the MPS_Stop function once in the initialization phase of the software to force the device to enter the standby state and ensure the safety of subsequent settings.

HANDLE DeviceHandle: Operation handle of the target device.

- int MPS_CloseDevice (HANDLE DeviceHandle)

The MPS_CloseDevice function is used to close the device safely. This function needs to be called when the software exits or releases the device control right. The function returns 1 on success and 0 on failure.

If the software is forcibly interrupted without closing the device, the system may not be able to completely release the device driver resources, resulting in other programs being unable to open the device normally. At this time, you can close all programs and threads associated with the device driver, reset the hardware

by re-plugging the USB connection or sending a reset command, and restart the computer if necessary.

HANDLE DeviceHandle: Operation handle of the target device.

- int MPS_TimeOut (int TimeOut_ms, HANDLE DeviceHandle)

The MPS_TimeOut function is used to set the timeout period to limit the execution time of the MPS_DataIn function. The MPS_TimeOut function needs to be called before the MPS_DataIn function. A time threshold (TimeOut_ms) can be specified to prevent the MPS_DataIn function from causing program blocking due to infinite waiting. If the MPS_DataIn function is not completed within the specified time, a timeout handling is triggered to force it to terminate and return a failure value. If the timeout limit is not enabled, the MPS_DataIn function will wait continuously until it succeeds or fails for other reasons.

The MPS_TimeOut function is mainly used for:

- In the external start trigger mode, the MPS_DataIn function needs to be called in advance to wait for the trigger. By setting the MPS_TimeOut function, an upper limit can be set for the waiting time. If the trigger is not received within the specified time, the waiting will be automatically terminated.
- Before calling the MPS_Start function to start acquisition, a timeout period can be preset through the MPS_TimeOut function. In case of hardware failure or software logic error (such as calling the MPS_DataIn function after calling the MPS_Stop function), the MPS_DataIn function can be forcibly terminated to avoid program blocking.

After the timeout limit is enabled through the MPS_TimeOut function:

- If the MPS_DataIn function returns normally without triggering timeout handling, the set timeout period is always valid no matter how many times the MPS_DataIn function is called until a timeout occurs or the setting is cleared.
- When a timeout handling is triggered once, the set timeout limit will become invalid, and the system will reset to the state where the timeout limit is not enabled.
- When the MPS_Stop function is called, the set timeout limit will be cleared while stopping the acquisition.

After the MPS_TimeOut function is executed, it will return the actual effective TimeOut_ms value (see the TimeOut_ms parameter description below for details).

int TimeOut_ms: This parameter is used to set the timeout time in milliseconds.

The valid timeout time range is 100 (milliseconds) to 1000000 (milliseconds).

- When TimeOut_ms is set to 0 or a value less than 100, the timeout limit function is not enabled. At this time, the MPS_TimeOut function returns 0 after execution.
- When TimeOut_ms is set to a value between 100 and 1000000, the timeout time is set according to the value of TimeOut_ms in milliseconds. At this time, the MPS_TimeOut function returns the value of TimeOut_ms after execution.
- When TimeOut_ms is set to a value greater than 1000000, the timeout time will be forcibly set to 1000000 milliseconds. At this time, the MPS_TimeOut function returns 1000000 after execution.

HANDLE DeviceHandle: Operation handle of the target device.

- int MPS_GetDeviceDescriptor (char *DescriptorBuffer, int BufferSize, HANDLE DeviceHandle)

The MPS_GetDeviceDescriptor function is used to obtain the device model description information. After the function is executed successfully, a set of characters can be obtained, such as: "MPS-HD1604", "MPS-HD2404", etc., ending with '\0' (null character) at last. This set of characters can be used to identify the hardware model of the current device. The function returns 1 on success and 0 on failure.

char *DescriptorBuffer: This parameter is a pointer to a user-defined character buffer, which is used to save the device model information.

The character buffer should be a char[] type array or a string. The array or string needs to set the length in advance, and its size must be greater than or equal to the value of the BufferSize parameter. It is recommended to allocate 64 bytes. After the MPS_GetDeviceDescriptor function is executed successfully, it will write up to BufferSize bytes of characters (including '\0') into the buffer.

int BufferSize: This parameter specifies the number of bytes of characters to read.

After the MPS_GetDeviceDescriptor function is executed successfully, the first BufferSize characters in the DescriptorBuffer buffer will be updated. The value of BufferSize must be less than or equal to the actual size of the DescriptorBuffer buffer, otherwise it may cause memory out of bounds error, and the maximum value is not more than 64. If it is greater than 64, it will be automatically set to 64. It is recommended to set the value of BufferSize to 64.

HANDLE DeviceHandle: Operation handle of the target device.

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

  The MPS_GetDeviceInformation function is used to obtain the hardware information of the device. After the function is executed successfully, the complete hardware information of the current device can be obtained, including the device model, serial number, hardware parameters and user-defined information. The information is returned in character form, ending with '\0' (null character) at last. The function returns 1 on success and 0 on failure.

  The MPS_GetDeviceInformation function takes a long time to execute, so it is recommended to call it in the software initialization phase.

  char *InformationBuffer: This parameter is a pointer to a user-defined character buffer, which is used to save the device hardware information.

  The character buffer should be a char[] type array or a string. The array or string needs to set the length in advance, and its size must be greater than or equal to the value of the BufferSize parameter. It is recommended to allocate 2048 bytes. After the MPS_GetDeviceInformation function is executed successfully, it will write up to BufferSize bytes of characters (including '\0') into the buffer.

  int BufferSize: This parameter specifies the number of bytes of characters to read.

  After the MPS_GetDeviceInformation function is executed successfully, the first BufferSize characters in the InformationBuffer buffer will be updated. The value of BufferSize must be less than or equal to the actual size of the InformationBuffer buffer, otherwise it may cause memory out of bounds error, and the maximum value is not more than 2048. If it is greater than 2048, it will be automatically set to 2048. It is recommended to set the value of BufferSize to 2048.

  HANDLE DeviceHandle: Operation handle of the target device.

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

  The MPS_SetUserInformation function is used to write user-defined information.

After the function is executed successfully, a section of user-defined information can be written into the device hardware, and the information will not be lost after power off. The information is written in character form, and it is recommended to end with '\0' (null character) at last. The function returns 1 on success and 0 on failure.

The MPS_SetUserInformation function takes a long time to execute and needs to be called when the device is in the idle state. The custom information supports about 100,000 repeated writes, and can be used for offline storage of fixed information such as software check codes, software custom calibration coefficients, and sensor sensitivity.

char *UserInformationBuffer: This parameter is a pointer to a user-defined character buffer, which is used to save the custom information to be written.

The character buffer should be a char[] type array or a string. The array or string needs to set the length in advance, and its size must be greater than or equal to the value of the BufferSize parameter. It is recommended to allocate 500 bytes. Before the MPS_SetUserInformation function is executed, the user-defined character information needs to be written into the character buffer in advance, and it is recommended to end with '\0' (null character) at last. After the MPS_SetUserInformation function is executed successfully, it will write the first BufferSize bytes of characters in the buffer into the hardware.

int BufferSize: This parameter specifies the number of bytes of characters to write.

After the MPS_SetUserInformation function is executed successfully, the first BufferSize characters in the UserInformationBuffer buffer will be written into the hardware. The value of BufferSize must be less than or equal to the actual size of the UserInformationBuffer buffer, otherwise it may cause memory out of bounds error, and the maximum value is not more than 500. If it is greater than 500, it will be automatically set to 500. It is recommended to set the value of BufferSize to the number of valid characters (including the ending '\0').

HANDLE DeviceHandle: Operation handle of the target device.

- int MPS_ResetDevice (HANDLE DeviceHandle)

The MPS_ResetDevice function is used to reset the hardware. After the function is executed successfully, the device hardware will be reset to the initial state. The USB connection will be automatically disconnected and reconnected, the device handle before reset will become invalid, and the program needs to call the MPS_OpenDevice function again to obtain a new handle for subsequent

operations. The function returns 1 on success and 0 on failure.

When the device status is abnormal and causes program blocking, you can try to use the reset function to reset the hardware. If the current device handle in use cannot be obtained due to function blocking, a new device handle can also be obtained by calling MPS_OpenDevice again in another thread to execute the MPS_ResetDevice function.

HANDLE DeviceHandle: Operation handle of the target device.

3. Code Examples

```
/*Functional subfunction for data acquisition*/
/*Basic process: Open Device -> Set Parameters -> Start Acquisition -> Loop to Get Data -> Stop Acquisition -> Close Device*/

/*Predefined SampleRate parameter values*/
#define SampleRate64K 64000                    //Sampling rate: 64K
#define SampleRate32K 32000                    //Sampling rate: 32K
#define SampleRate16K 16000                    //Sampling rate: 16K
#define SampleRate8K 8000                      //Sampling rate: 8K
#define SampleRate4K 4000                      //Sampling rate: 4K
#define SampleRate2K 2000                      //Sampling rate: 2K
#define SampleRate1K 1000                      //Sampling rate: 1K

/*Predefined DIOMode parameter values*/
#define DIOIdle 0                              //DIO port idle
#define DIOOutput5V 1                          //DIO port outputs 5V high level
#define ExternalTrigger 2                      //DIO set to external trigger start mode


/*Predefined variables*/
#define VoltageRange 5.2                       //Voltage range of MPS-HD1604: 5.2V
#define MaxNumberPerCH  7680000                //Maximum number of voltage samples per channel, predefined as about 7.68 million points here, supporting continuous acquisition for up to 120s at 64K sampling rate
double Voltage[4][MaxNumberPerCH] = {0};       //2D cache array for storing voltage, each channel can store up to MaxNumberPerCH points

/*Data acquisition subfunction*/
int GetVoltageData(int SampleNumberPerCH = 64000, int SampleRate = SampleRate64K, int DIOMode = DIOIdle)   //Acquire data, parameters: number of samples, sampling rate, DIO mode
{
    /*Function declarations*/
    #define Handle int

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

```
typedef Handle(*lpMPS_OpenDevice)(int DeviceNumber);      //Declaration of Open Device function
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);      //Declaration of Close Device function
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);      //Declaration of
Configure Device Parameters function
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);                //Declaration of Start Device Acquisition
function
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);                //Declaration of Stop Device Acquisition
function
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);      //Declaration of
Read Data function
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);    //Declaration of Set Read Timeout function
    lpMPS_TimeOut MPS_TimeOut=(lpMPS_TimeOut)GetProcAddress(hDll,"MPS_TimeOut");
    if(NULL==MPS_TimeOut)
    {
        AfxMessageBox("Cannot find <MPS_TimeOut> function");
    }

    /*End of function declarations*/


    /*Acquire data*/
    //Define variables
    int Flag = 1;                                   //Function execution success flag
    int DeviceNumber = 0;                           //Hardware serial number of the device assigned by the OS
    Handle DeviceHandle;                            //Device handle

    DeviceHandle = MPS_OpenDevice(DeviceNumber);    //Open device first
    if(DeviceHandle == -1)                          //If open failed, report error and return
    {
        AfxMessageBox("OpenDeviceError");
        return 0;
    }

    Flag = MPS_Configure(SampleRate, DIOMode, DeviceHandle);//Set sampling rate and set multiplexed IO port to idle mode

    MPS_TimeOut(5000, DeviceHandle);                //Set wait timeout to 5s, modify the value if used for external trigger timeout

    if(DIOMode != ExternalTrigger)Flag = MPS_Start(DeviceHandle);//Start device acquisition; do not start via MPS_Start in external trigger mode


    //Read data by calling acquisition function in loop below
    int DataBuffer[4*1024] = {0};                   //Cache array for storing raw integer data read out, note that this variable must be defined as 32-bit integer
                                                    //Voltage array for storing calculated voltage values has been defined as global variable
```

```
    if(SampleNumberPerCH > MaxNumberPerCH)
            SampleNumberPerCH = MaxNumberPerCH;        //Number of samples acquired per channel
should be less than MaxNumberPerCH

    int Counter = 0;                                   //Data point count variable

    while(1)
    {
        Flag = MPS_DataIn(DataBuffer,1024*4,DeviceHandle);  //Read data, 1024*4 points in total for four
channels each time

        if(Flag != 0)                                  //If acquisition succeeded
            {
                for(int i = 0; i < 1024; i++)          //Data processing, calculate voltage values;
data of four channels are arranged cyclically in cache; Voltage = (Acquired Value/(65536*128)) * Range
                {
                    if(Counter >= SampleNumberPerCH) break;
                                                       //Break for loop if sample points reach
SampleNumberPerCH
                    Voltage[0][Counter] = ((double)DataBuffer[i*4] / 8388608) * VoltageRange;
                                                       //Voltage of Channel 1
                    Voltage[1][Counter] = ((double)DataBuffer[i*4 + 1] / 8388608) * VoltageRange;
                                                       //Voltage of Channel 2
                    Voltage[2][Counter] = ((double)DataBuffer[i*4 + 2] / 8388608) * VoltageRange;
                                                       //Voltage of Channel 3
                    Voltage[3][Counter] = ((double)DataBuffer[i*4 + 3] / 8388608) * VoltageRange;
                                                       //Voltage of Channel 4
                    Counter++;
                }
            }
            else
            {
                AfxMessageBox("MPS_DataInError");      //Report error and break while loop if
acquisition failed
                break;
            }

        if(Counter >= SampleNumberPerCH)               //Break while loop if all data are read out.
Other loop exit conditions can be added here
        {
            break;
        }
    }
```

```
	//Acquisition process completed here, stop hardware acquisition and close device below
	Flag = MPS_Stop(DeviceHandle);			//Stop acquisition after acquisition task is
completed
	Flag = MPS_CloseDevice(DeviceHandle);		//Close device finally

	if (Counter == SampleNumberPerCH)		//Check if GetVoltageData function has acquired
the specified SampleNumberPerCH samples
	{
		AfxMessageBox("MPS_DataInSuccess");
		return 1;
	}
	else return 0;
}



/*Functional subfunction for getting device hardware information*/
int GetInformation()					//Get device hardware information
{
	/*Function declarations*/
	#define Handle int

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

	typedef Handle(*lpMPS_OpenDevice)(int DeviceNumber);//Declaration of Open Device function
	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);//Declaration of Close Device function
	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);
                                                    //Declaration of Get Device Model Information
function
    lpMPS_GetDeviceDescriptor
MPS_GetDeviceDescriptor=(lpMPS_GetDeviceDescriptor)GetProcAddress(hDll,"MPS_GetDeviceDescriptor");
    if(NULL==MPS_GetDeviceDescriptor)
    {
        AfxMessageBox("Cannot find <MPS_GetDeviceDescriptor> function");
    }


    typedef int(*lpMPS_GetDeviceInformation)(char *DeviceInformationBuffer,int BufferSize,Handle
DeviceHandle);                                      //Declaration of Get Complete Device Hardware
Information function
    lpMPS_GetDeviceInformation
MPS_GetDeviceInformation=(lpMPS_GetDeviceInformation)GetProcAddress(hDll,"MPS_GetDeviceInformation
");
    if(NULL==MPS_GetDeviceInformation)
    {
        AfxMessageBox("Cannot find <MPS_GetDeviceInformation> function");
    }

    /*End of function declarations*/


    //Define variables
    int Flag = 1;                                   //Function execution success flag
    int DeviceNumber = 0;                           //Hardware serial number of the device assigned
by the OS
    Handle DeviceHandle;                            //Device handle

    //Example of reading device model information
    char DeviceDescriptor[64] = {0};                //Cache array for storing device model information

    DeviceHandle = MPS_OpenDevice(DeviceNumber);    //Open device first
    if(DeviceHandle == -1)                          //If open failed, report error and return
    {
        AfxMessageBox("OpenDeviceError");
        return 0;
    }

    Flag = MPS_GetDeviceDescriptor(DeviceDescriptor,64,DeviceHandle);//Get device model
```

```
    Flag = MPS_CloseDevice(DeviceHandle);                    //Close device finally

    AfxMessageBox(DeviceDescriptor);                         //Display device model

    //Example of reading device hardware information
    char DeviceInformation[2048] = {0};                      //Cache array for storing device model information

    DeviceHandle = MPS_OpenDevice(DeviceNumber);             //Open device first
    if(DeviceHandle == -1)                                   //If open failed, report error and return
    {
        AfxMessageBox("OpenDeviceError");
        return 0;
    }

    Flag = MPS_GetDeviceInformation(DeviceInformation,2048,DeviceHandle);//Get device information

    Flag = MPS_CloseDevice(DeviceHandle);                    //Close device finally

    AfxMessageBox(DeviceInformation);                        //Display device information

    return 1;

}



/*Functional subfunction for writing user-defined information*/
int SetUserInformation()                                     //Write user-defined information
{
    /*Function declarations*/
    #define Handle int

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

    typedef Handle(*lpMPS_OpenDevice)(int DeviceNumber);//Declaration of Open Device function
    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);//Declaration of Close Device function
    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);                                  //Declaration of Write User-Defined Information
function
    lpMPS_SetUserInformation
MPS_SetUserInformation=(lpMPS_SetUserInformation)GetProcAddress(hDll,"MPS_SetUserInformation");
    if(NULL==MPS_SetUserInformation)
    {
        AfxMessageBox("Cannot find <MPS_SetUserInformation> function");
    }

    /*End of function declarations*/

    //Define variables
    int Flag = 1;                               //Function execution success flag
    int DeviceNumber = 0;                       //Hardware serial number of the device assigned
by the OS
    Handle DeviceHandle;                        //Device handle

    //Example of writing user-defined information
    char UserInformation[500] = "This is a test.";  //Cache array for user information to be written

    DeviceHandle = MPS_OpenDevice(DeviceNumber);    //Open device first
    if(DeviceHandle == -1)                      //If open failed, report error and return
    {
        AfxMessageBox("OpenDeviceError");
        return 0;
    }

    Flag = MPS_SetUserInformation(UserInformation,500,DeviceHandle);//Write user information

    Flag = MPS_CloseDevice(DeviceHandle);       //Close device finally
```

```
    AfxMessageBox(UserInformation);                    //Display written content

    return 1;

}



/*Hardware reset example*/
int ResetDevice()                                      //Hardware reset function
{
    /*Function declarations*/
    #define Handle int

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

    typedef Handle(*lpMPS_OpenDevice)(int DeviceNumber);//Declaration of Open Device function
    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);//Declaration of Hardware Reset function
    lpMPS_ResetDevice MPS_ResetDevice
=(lpMPS_ResetDevice)GetProcAddress(hDll,"MPS_ResetDevice");
    if(NULL==MPS_ResetDevice)
    {
        AfxMessageBox("Cannot find <MPS_ResetDevice> function");
    }

    /*End of function declarations*/

    //Define variables
    int Flag = 1;                                      //Function execution success flag
    int DeviceNumber = 0;                              //Hardware serial number of the device assigned
```

```
by the OS
        Handle DeviceHandle;                                    //Device handle

        DeviceHandle = MPS_OpenDevice(DeviceNumber);            //Open device first
        if(DeviceHandle == -1)                                  //If open failed, report error and return
        {
                AfxMessageBox("OpenDeviceError");
                return 0;
        }

        Flag = MPS_ResetDevice(DeviceHandle);                   //Hardware reset function

        if (Flag != 0)
                {
                        AfxMessageBox("MPS_ResetDeviceSuccess");
                }
        else
                {
                        AfxMessageBox("MPS_ResetDeviceError");
                }
        return Flag;
}



/*Function test*/
void Test()
{
        int Flag = 0;
//      Flag = GetInformation();                                //Device hardware information read test, usually
read once during software initialization when needed, no need to read frequently
//      Flag = SetUserInformation();                            //User-defined information write test, do not
execute this function if no information needs to be written
        Flag = GetVoltageData(64000,SampleRate64K, DIOIdle);//Data acquisition test, acquire 1s data at 64K
sampling rate
//      Flag = ResetDevice();                                   //Hardware reset test
}
```

Chapter 4: Precautions

- Please use moderate force when plugging and unplugging the device wiring ports to avoid damaging the interface.
- Please wait about 1 second before reconnecting after the device is disconnected from the computer USB.
- The device usually works normally with computer USB power supply. If there is insufficient power supply, you can use a USB HUB with external power supply to improve the power supply capacity of the USB bus.
- The device is a precision electronic instrument, please pay attention to dustproof, moistureproof and anti-static during use. If there is a risk of human electrostatic discharge, please take anti-static measures in advance, and try to avoid touching metal parts in connectors, shells, signal lines and sensors during use. Please keep the device sealed when not in use for a long time.
- Users are prohibited from disassembling the lower shell of the device by themselves. Once disassembled, the warranty service will no longer be available, and the user will be responsible for any device failure caused thereby.
- The MPS-HD series products are covered by a three-year warranty from the date of shipment.Any failure caused by product quality defects shall be repaired free of charge upon presentation of the warranty card or order information, provided that the user complies with the storage, transportation and usage requirements.Man-made damage resulting from violation of operating regulations and usage requirements shall be subject to a repair charge.
- MPS series signal acquisition cards are provided by Morpheus Electronic. For more products and information, please visit: www.mps-electronic.com.cn, consultation email: mail@mps-electronic.com.cn.