Instruction for N-ScanHub

1 Introduction

Newland’s SDK supports Windows and linux platforms and offers the C/C++ interface to interact with Newland devices. With the SDK, users can carry out secondary development, obtain devices, send instructions, upgrade firmware, etc.

Directory Structure

Items	Descriptions
Platform	Windows and Linux platforms
Programming Language	C/C++
Functions	Obtaining device, sending commands, upgrading firmware, read and write, opening and closing the device , collecting pictures, plugging and unplugging and data acquisition notification, etc.
SDK	N-ScanHubForLinux and N-ScanHubForWindows
API	N-ScanHubForLinux and N-ScanHubForWindows with the same interface name

2 Introduction to N-ScanHubForWindows

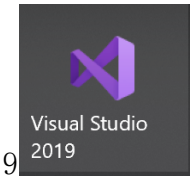
2.1 Directory Structure

N-ScanHubForWindows offers the API under the Windows platform, and its directory is shown as below.

Contents	Descriptions
include	Header file: N-ScanHub.h (all interfaces descriptions included)
lib/x64	64-bit N-ScanHub.dll and N-ScanHub.lib
lib/x86	32-bit N-ScanHub.dll and N-ScanHub.lib
demo	Library file and demo written by Visual Studio2019

help	Help document: N-ScanHub.pdf

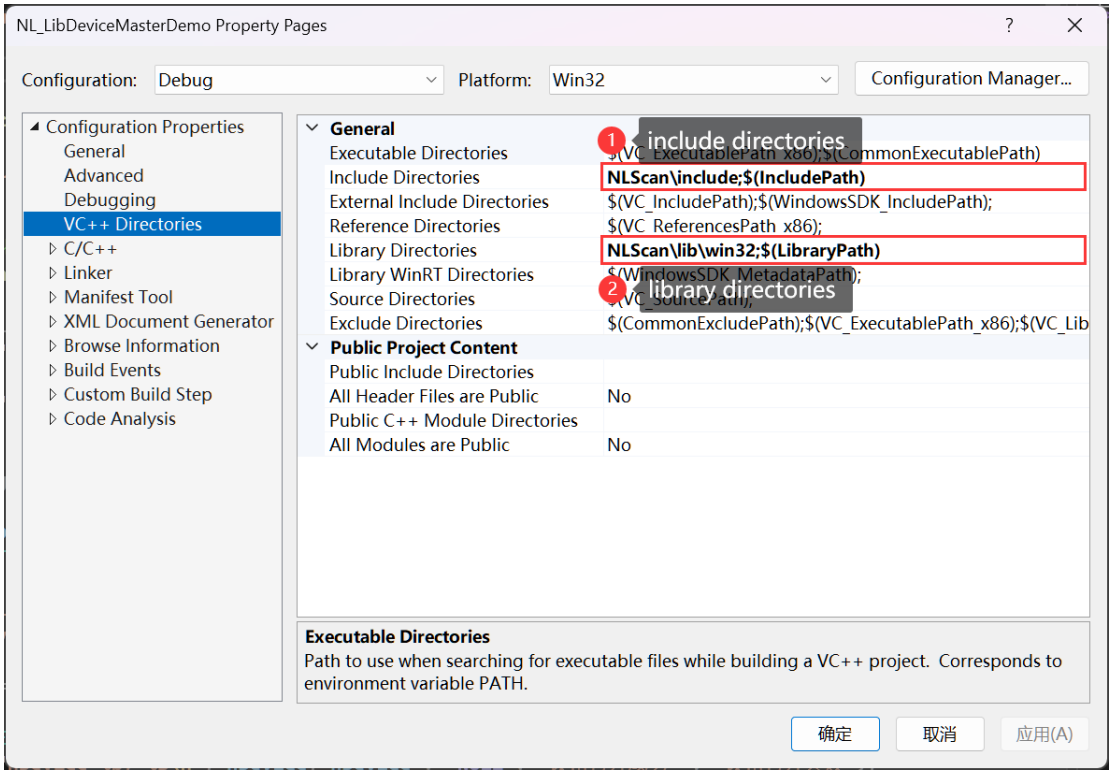
2.2 NLSInfoStreamForWindows Operating Instructions

Device: FM430  Tool: VS2019 

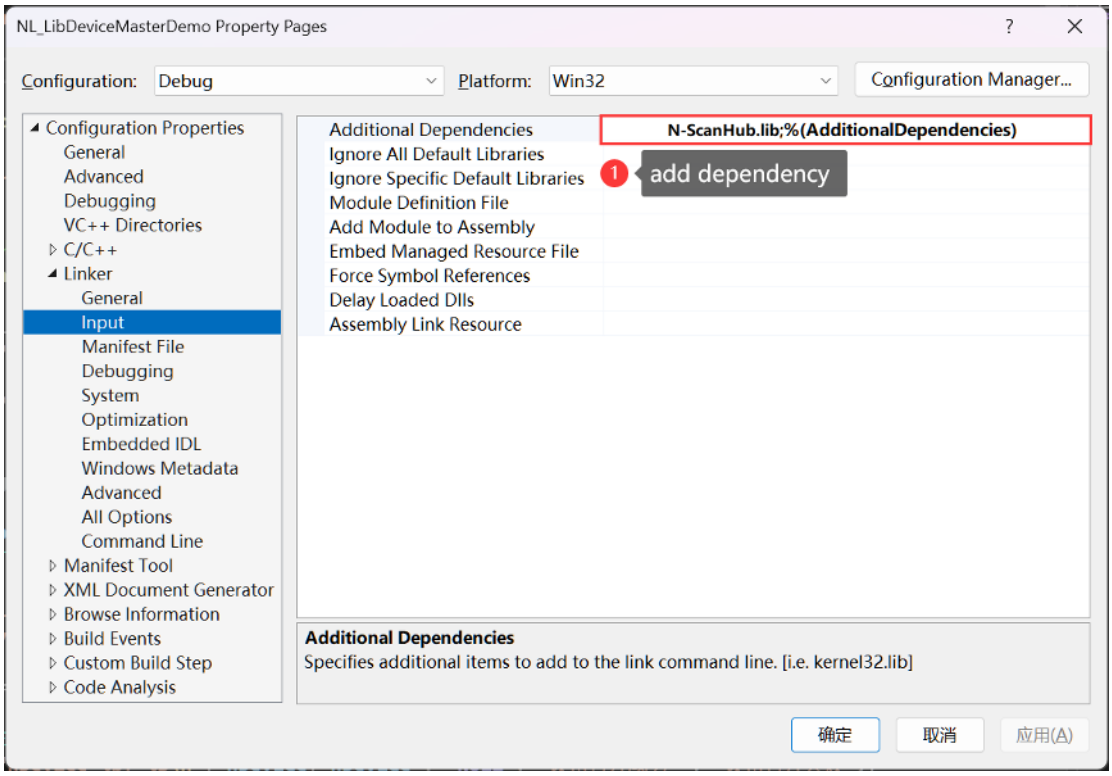
System: Windows 10

N-ScanHubForWindows demo Operating Steps

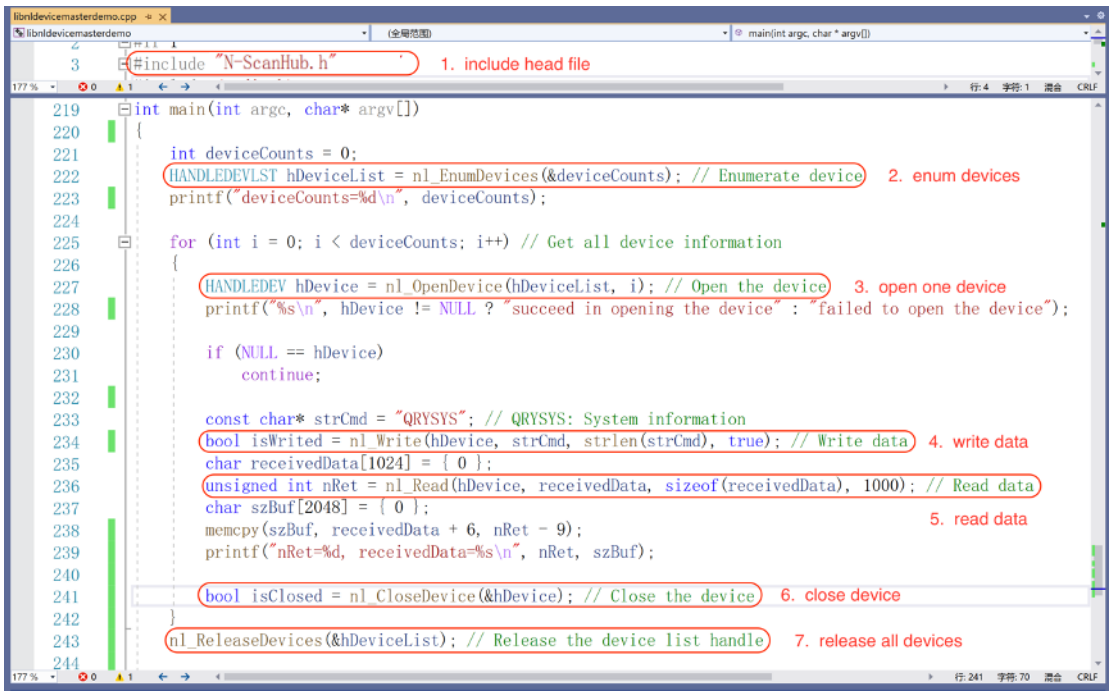
1. The project contains paths of N-ScanHub.h and N-ScanHub.lib.



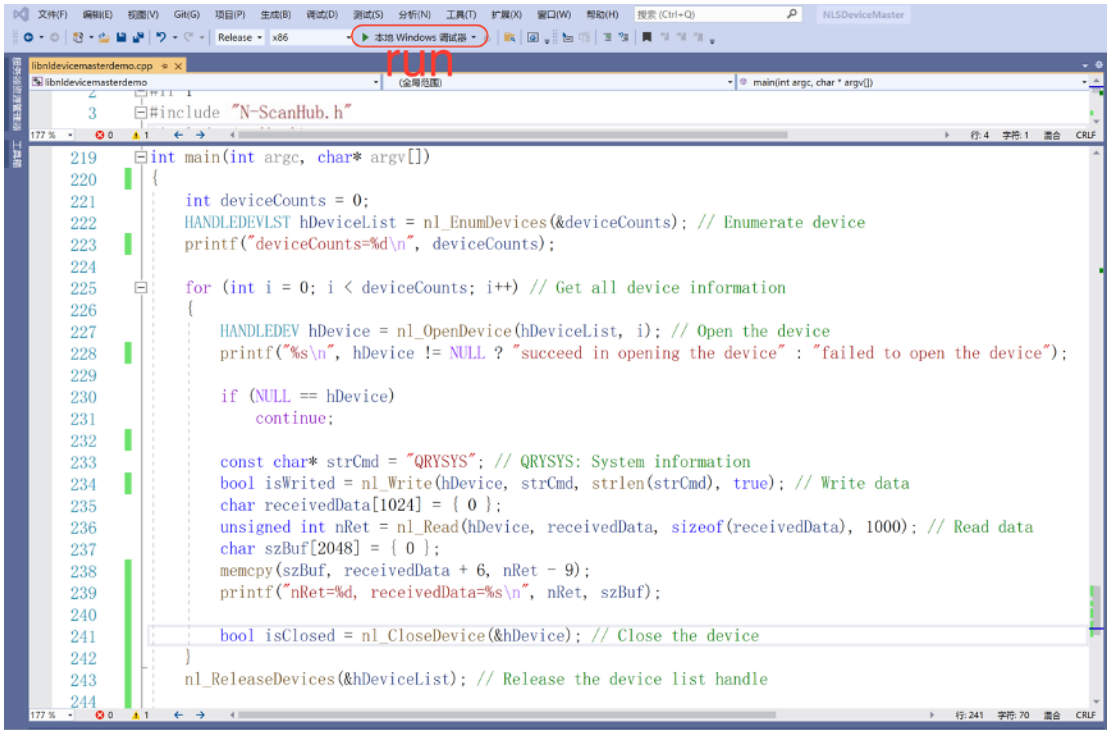
2. Add the dependency libnldevicemaster.lib.



3. Call the function in the SDK.



4. Start running the program.



The result is as follows.

2.3 Example of demo

```
#include "N-ScanHub.h"

#include <stdio.h>

#include <windows.h>

#include <chrono>

#include <vector>
```

```
#include <string>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <thread>
using namespace std;

#define NET_TEST 1

// Read device data
void __stdcall ReadCallback(const HANDLEDEV hDevice, const char* buf, int len)
{
	printf("-----------ReadCallback len=%d, buf=%s\n", len, buf);
}

// Monitoring device status change
void __stdcall DevStatChangeCallback(const HANDLEDEV hDevice, bool isDevExisted)
{
	if (isDevExisted)
		printf("hDevice=%p, device is pushed in\n", hDevice);
	else
		printf("hDevice=%p, device is pushed out\n", hDevice);
}
#if NET_TEST
```

```
void __stdcall TcpServiceBack(int clientSocket, char* clientIp) {
	printf("clientIp=%s\n", clientIp);
	char buf[4096] = { 0 };
	int len = 4096;
	while (1) {
		if (nl_readFromSocket(clientSocket, 2000, buf, &len) == 0) {
			printf("TcpServiceBack buf=%s\n", buf);
			memset(buf, 0, sizeof(buf));
		}
		Sleep(500);
	}
}

int piccount = 0;
void __stdcall readNetImage(unsigned char* data, int data_len){
	char filename[256] = { 0 };
	sprintf(filename, "./pic/%d.jpg", piccount++);
	//nl_saveNetImgData(0, data, data_len, filename);
}

void __stdcall readTcpIpData(unsigned char* data, int data_len) {
	printf("data=%s\n", data);
}
```

```
void ServerThread() {
	int ret = nl_CreateTcpService(10000, TcpServiceBack);
	printf("ServerThread ret=%d\n", ret);
}

void NetImageThread(char* ip, int* port1, int* port2) {
	int socket36520 = -1;

	char sendbuf[1024] = { 0 };
	char recvbuf[1024] = { 0 };
	int socket30000 = -1;

	strcpy(sendbuf, "\x01\x54\x04");

	nl_connectToService(ip, *port1, &socket30000);
	nl_connectToService(ip, *port2, &socket36520);
	const int RECV_BUFFER_SIZE = 1920 * 1080 * 4;
	char* recvBuffer = (char*)malloc(RECV_BUFFER_SIZE);
	int realLen = 0, nRet = -1;
	IMG_TYPE imgtype;
	int w, h, f, q;
	f = 2;
	q = 2;
	char filename[128] = { 0 };
```

```
for (int i = 0; i < 1000; i++) {
        memset(recvBuffer, 0, RECV_BUFFER_SIZE);
        memset(recvbuf, 0, 1024);
        // rigger recognition
        if (nl_sendDataToSocket(socket30000, sendbuf, 3) != 0) {
            printf("nl_sendDataToSocket error\n");
            continue;
        }
        //get code
        if (nl_readFromSocket(socket30000, 2, recvbuf, &realLen) != 0) {
            printf("nl_readFromSocket error\n");
            nl_CloseClientSocket(socket30000);
            Sleep(500);
            nl_connectToService(ip, *port1, &socket30000);
            if (socket30000 == -1) {
                printf("reconncet error\n");
                continue;
            }
        }
        printf("code length=%d,data=%s\n", realLen, recvbuf);
```

// In retrieving images, the two most commonly used parameters are 'f,' which represents the data type of the image, and 'q,' which is the compression quality level for obtaining JPEG image data when 'f' is set to 2

```
nRet = nl_getNetImgData(socket36520, 0, 0, f, q, recvBuffer, &realLen, &imgtype, &w,
&h);

printf("-----------------------------------ip=%s [%d][%d]\n", ip, nRet, realLen);

if (nRet != 0)

    continue;

// When 'f' is set to 0, it represents raw data, and you need to call the image-saving
interface to package the raw data in order to generate an image file

// If you prefer not to use the saving interface, you can also implement image data
packaging on your own

if (f == 0) {

    if (imgtype == TYPE_COLOR) {

        sprintf(filename, "./pic/f0-%s-%04d.bmp", ip, i);



        nl_SavePicDataToFile(filename, (unsigned char*)recvBuffer, w, h, 24); // Save
image



        sprintf(filename, "./pic/f0-%s-%04d.jpg", ip, i);



        nl_SavePicDataToFile(filename, (unsigned char*)recvBuffer, w, h, 23); // Save
image

        DbgPrintf("nl_SavePicDataToFile jpg end");

    }

    else {

        sprintf(filename, "./pic/f0-%s-%04d.bmp", ip, i);

        nl_SavePicDataToFile(filename, (unsigned char*)recvBuffer, w, h, 8); // Save
image
```

```
                sprintf(filename, "./pic/f0-%s-%04d.jpg", ip, i);

                nl_SavePicDataToFile(filename, (unsigned char*)recvBuffer, w, h, 13); // Save
image

            }

        }

        // When 'f' takes any of the following values, you will obtain image format data that can
be directly saved as a binary file

        else if (f == 1) {

            sprintf(filename, "./pic/f1-%s-%04d.bmp", ip, i);

            FILE* fp = fopen(filename, "wb");

            fwrite(recvBuffer, 1, realLen, fp);

            fclose(fp);

        }

        else if (f == 2) {

            sprintf(filename, "./pic/f2-%s-%04d.jpg", ip, i);

            FILE* fp = fopen(filename, "wb");

            fwrite(recvBuffer, 1, realLen, fp);

            fclose(fp);

        }

        else if (f == 3) {

            sprintf(filename, "./pic/f3-%s-%04d.bmp", ip, i);

            FILE* fp = fopen(filename, "wb");

            fwrite(recvBuffer, 1, realLen, fp);

            fclose(fp);

        }
```

```
            else if (f == 4) {
                sprintf(filename, "./pic/f4-%s-%04d.bmp", ip, i);
                FILE* fp = fopen(filename, "wb");
                fwrite(recvBuffer, 1, realLen, fp);
                fclose(fp);
            }

            Sleep(10);
        }
        nl_CloseClientSocket(socket36520);
        nl_CloseClientSocket(socket30000);
        free(recvBuffer);
        //delete[]recvbuf;
        recvBuffer = NULL;
        printf("-------------------------------------ip=%s close\n", ip);
}
#endif

void SplitString(const string& s, vector<string>& v, const string& c)
{
        string::size_type pos1, pos2;
        pos2 = s.find(c);
        pos1 = 0;
        while (string::npos != pos2)
```

```
    {
        v.push_back(s.substr(pos1, pos2 - pos1));

        pos1 = pos2 + c.size();
        pos2 = s.find(c, pos1);
    }
    if (pos1 != s.length())
        v.push_back(s.substr(pos1));
}

int main(int argc, char* argv[])
{
    int deviceCounts = 0;
    HANDLEDEVLST hDeviceList = NULL;

    DbgPrintf("enum nl_EnumDevices begin\n");
    hDeviceList = nl_EnumDevices(&deviceCounts, ENUM_ALL); // Enumerate device
    DbgPrintf("enum nl_EnumDevices end\n");
    printf("deviceCounts=%d\n", deviceCounts);

    for (int i = 0; i < deviceCounts; i++) // Get all device information
    {
        HANDLEDEV hDevice;
        hDevice = nl_OpenDevice(hDeviceList, i); // Open the device
```

```
        printf("hDevice=%p, %s\n", hDevice, hDevice != NULL ? "succeed in opening the device" : "failed to open the device");


        if (NULL == hDevice)
            continue;


        T_DeviceStatus status = nl_GetDevStatus(hDevice); // Get device status
        printf("status=%d\n", status);
        if (argc < 2) {
            // Write character string data
            // trigger recognition
            const char* strCmd = "\x01\x54\x04"; // QRYSYS: System information
            char receivedData[65565] = { 0 };
            int recvlen = 0;
            nl_Write(hDevice, strCmd, 3, false);
            int ret = nl_Read(hDevice, receivedData, 1000, 1000);
            printf("system info: \n%s\n", receivedData);
        }


        if (argc >= 2 && strcmp(argv[1], "--WriteAsHex") == 0) // Write data to the device in HEX character string
        {
            // Write hex character string data
            const char* strCmdhEX = "7e 01 30 30 30 30 40 51 52 59 53 59 53 3b 03"; // System information
```

```
            char receivedData[1024] = { 0 };

            int nRet = 0;

            bool isWrited = nl_WriteAsHex(hDevice, strCmdhEX, true); // Write data

            nRet = nl_Read(hDevice, receivedData, sizeof(receivedData), 0); // Read data

            printf("nRet=%d, receivedData=%s\n", nRet, receivedData);

        }

        else if (argc >= 2 && strcmp(argv[1], "--GetDeviceInfo") == 0) // Write data to the device in HEX character string

        {

            // Retrieve device information

            STDeviceInfo info;

            memset(&info, 0, sizeof(STDeviceInfo));

            nl_GetDeviceInfo(hDeviceList, i, &info);

            printf("GetDeviceInfo -------- info\n %s\ntype=%d\n", info.devInfo, info.devType);

        }

        else if (argc >= 2 && strcmp(argv[1], "--SendCommand") == 0) // Send control commands to the device and obtain the returned information

        {

            // Send a command and check if the command was successful.

            char strCmd[2048] = { 0 };

            strcpy(strCmd, "QRYSYS;");

            int result = nl_SendCommand(hDevice, strCmd, strlen(strCmd)); // Send commands

            printf("result=%d\n", result);

        }
```

```
        else if (argc >= 2 && strcmp(argv[1], "--SendCommandAsHex") == 0) // Send control commands to the device in the form of HEX character string and get the returned information.

        {

            const char* strCmd = "51 52 59 53 59 53"; // QRYSYS: System information

            T_CommunicationResult result = nl_SendCommandAsHex(hDevice, strCmd, strlen(strCmd)); // Send commands

            printf("result=%d\n", result);

        }

        else if (argc >= 2 && strcmp(argv[1], "--GetCommandResponse") == 0) {

            // Send a command, receive the command returned by the device, with a maximum supported length of 20,000.

                char strCmd[20480] = { 0 };

                strcpy(strCmd, "QRYSYS;");

                char recvData[20480] = { 0 };

                int recvLen = 0;

                bool result = nl_GetCommandResponse(hDevice, strCmd, strlen(strCmd), recvData, &recvLen, 500, true, false);

                printf("result=%d\n", result);

                printf("recvData=%s\n", recvData);

        }

        else if (argc >= 2 && strcmp(argv[1], "--GetPicture") == 0) // Get device image

        {

            // Basic interface for obtaining images, with all parameters set to default.

            unsigned int imgWidth = 0, imgHeight = 0;

            char filename[1024] = { 0 };
```

```
            sprintf(filename, "1%d.bmp", i);

            bool isGetPicSizeOK = nl_GetPicSize(hDevice, &imgWidth, &imgHeight); // Get the image width and height

            printf("nl_GetPicSize isGetPicSizeOK=%d\n", isGetPicSizeOK);

            if (isGetPicSizeOK && imgWidth > 0 && imgHeight > 0)

            {

                printf("imgWidth=%d,imgHeight=%d\n", imgWidth, imgHeight);

                const int RECV_BUFFER_SIZE = imgWidth * imgHeight;

                unsigned char* recvBuffer = (unsigned char*)malloc(RECV_BUFFER_SIZE);

                bool isOK = nl_GetPicData(hDevice, recvBuffer, RECV_BUFFER_SIZE); // Get the image raw data

                if(isOK)

                    nl_SavePicDataToFile(filename, recvBuffer, imgWidth, imgHeight, 8);

            }

        }

        else if (argc >= 2 && strcmp(argv[1], "--GetPictureByConfig") == 0) // Get device image

        {

            // Obtain image data with configurable parameters.

            unsigned int imgWidth = 0, imgHeight = 0;

            bool isGetPicSizeOK = nl_GetPicSize(hDevice, &imgWidth, &imgHeight); // Get the image width and height

            printf("nl_GetPicSize isGetPicSizeOK=%d\n", isGetPicSizeOK);

            if (isGetPicSizeOK && imgWidth > 0 && imgHeight > 0)

            {

                printf("imgWidth=%d,imgHeight=%d\n", imgWidth, imgHeight);
```

```
const int RECV_BUFFER_SIZE = imgWidth * imgHeight * 4;

// Allocate a sufficiently large space to store image data

unsigned char* recvBuffer = (unsigned char*)malloc(RECV_BUFFER_SIZE);

STImgParam imgParam;

memset(&imgParam, 0, sizeof(STImgParam));

imgParam.f = 0;

imgParam.q = 3;

STImgResolution imgR[4];

memset(imgR, 0, sizeof(STImgResolution) * 4);

unsigned int nRealLen = 0;

bool isOK = nl_GetPicDataByConfig(hDevice, imgParam, recvBuffer,
&nRealLen, imgR); // Get the image data

printf("isOK=%d\n", isOK);

char filename[1024] = { 0 };

if (isOK) {

    if (imgParam.t == 2) {

        for (int i = 0; i < 4; i++) {

            printf("imgR[%d] width=%d height=%d\n", i, imgR->width,
imgR->height);

        }

    }

    if (imgParam.f == 1) {

        sprintf(filename, "test3%d.bmp", i);

        FILE* fp = fopen(filename, "wb");

        fwrite(recvBuffer, 1, nRealLen, fp);
```

```
        fclose(fp);
    }
    else if (imgParam.f == 2) {
        sprintf(filename, "test4%d.jpg", i);
        FILE* fp = fopen(filename, "wb");
        fwrite(recvBuffer, 1, nRealLen, fp);
        fclose(fp);
    }
    else if (imgParam.f == 3) {
        sprintf(filename, "test5%d.tiff", i);
        FILE* fp = fopen(filename, "wb");
        fwrite(recvBuffer, 1, nRealLen, fp);
        fclose(fp);
    }
    else if (imgParam.f == 4) {
        sprintf(filename, "test6%d.bmp", i);
        FILE* fp = fopen(filename, "wb");
        fwrite(recvBuffer, 1, nRealLen, fp);
        fclose(fp);
    }
    else if (imgParam.f == 0) {
        long outLen = 0;
        STImgResolution imgResIn, imgResOut;
        if (imgParam.r == 1) {
```

```
                imgWidth = imgWidth / 2;
                imgHeight = imgHeight / 2;
            }
            else if (imgParam.r == 2) {
                imgWidth = imgWidth / 4;
                imgHeight = imgHeight / 4;

            }

            imgResIn.width = imgWidth;

            imgResIn.height = imgHeight;

            unsigned int imgLen = 0;

            IMG_TYPE type = nl_GetDeviceImageColorType(hDevice,
&imgResOut, &imgLen);

            if (type == TYPE_COLOR) {

                unsigned char* outBuf = (unsigned char*)malloc(imgLen);

                printf("imgLen=%d\n", imgLen);

                bool res = nl_ConvertImageColorSpace(hDevice, recvBuffer,
RECV_BUFFER_SIZE, imgResIn, outBuf);

                int oWidth, oHeight;

                // If you are capturing a partial image, use the resolution of
the captured portion

                if (strlen(imgParam.b) != 0) {

                    oWidth = stoi(string(imgParam.b).substr(8, 4));

                    oHeight = stoi(string(imgParam.b).substr(12, 4));

                }

                else {

                    oWidth = imgResOut.width;

                    oHeight = imgResOut.height;
```

```
                }
                sprintf(filename, "./pic/test2%d%d.bmp", i, j);
                nl_SavePicDataToFile(filename, outBuf, oWidth, oHeight, 24);
// Save image
                sprintf(filename, "./pic/test2%d%d.jpg", i, j);
                nl_SavePicDataToFile(filename, outBuf, oWidth, oHeight, 23);
// Save image
            }
            else {
                int oWidth, oHeight;
                if (strlen(imgParam.b) != 0) {
                    oWidth = stoi(string(imgParam.b).substr(8, 4));
                    oHeight = stoi(string(imgParam.b).substr(12, 4));
                }
                else {
                    oWidth = imgResOut.width;
                    oHeight = imgResOut.height;
                }
                sprintf(filename, "./pic/test1%d%d.bmp", i, j);
                nl_SavePicDataToFile(filename, recvBuffer, oWidth, oHeight, 8);
// Save image
                sprintf(filename, "./pic/test1%d%d.jpg", i, j);
                nl_SavePicDataToFile(filename, recvBuffer, oWidth, oHeight,
13); // Save image
            }
```

```
                }

            }

            free(recvBuffer);

            recvBuffer = NULL;

        }

    }

    else if (argc >= 2 && strcmp(argv[1], "--SetListener") == 0) // Asynchronous reading of device data

    {

        // Barcode scanning listener, when activated, the device will pass the scanned code to the callback function ReadCallback

        nl_SetListener(hDevice, ReadCallback);

        Sleep(1000000);

        nl_StopListener(hDevice);

    }

    else if (argc >= 2 && strcmp(argv[1], "--ReadDevCfgToXml") == 0) // Read the configuration from the device and save it to the xml file.

    {

        char filename[128] = { 0 };

        strcpy(filename, "./xml/2.xml");

        nl_ReadDevCfgToXml(hDevice, filename);

    }

    else if (argc >= 2 && strcmp(argv[1], "--WriteCfgToDev") == 0) // Write the configuration file information to the device.

    {
```

```
            bool ret = nl_WriteCfgToDev(hDevice, "./xml/2.xml");

            if (!ret)

                printf("WriteCfgToDev %s\n", nl_GetLastError());//If there is an error during XML import, calling nl_GetLastError can provide information about which commands were imported incorrectly

        }

        else if (argc >= 2 && strcmp(argv[1], "--SetCbDevStatusChanged") == 0) // Set the callback function when the device status changes

        {

            // Monitor device plug and unplug states and take actions to open or close, only monitor local serial and USB devices, this interface is not effective for network devices

            nl_SetCbDevStatusChanged(hDevice, DevStatChangeCallback);

            Sleep(100000);

            printf("SetCbDevStatusChanged finish\n");

        }

        else if (argc >= 2 && strcmp(argv[1], "--UpdateFirmware") == 0) // Update device

        {

            unsigned updateError = -1;

            bool isUpdated = nl_UpdateKernelDevice(hDevice, "Y:/Newland/scan/firmware/soldier160/SOLDIER160_V1.04.003.4.bin2", 0, &updateError); // Firmware update

            printf("updateError=%d,%s\n", updateError, isUpdated ? "succeed in updating the firmware" : "failed to update the firmware");

            switch (updateError)

            {

            case Success:

                printf("The firmware update is normal.\n");
```

```
                break;
            case FileNameExtError:
                printf("file name error\n");
                break;
            }
        }
        else if (argc >= 2 && strcmp(argv[1], "--SetNetDeviceConfig") == 0) {
            char configData[2048] = { 0 };
            strcpy(configData, "Serial Number=A6516268F31B66FC;MAC
Address=00:51:62:68:F3:1B;Device Use DHCP=1;Device IP Address=192.168.76.250;Device
SubNetmask=255.255.255.0;Device Gateway Address=192.168.76.1;");
            char outData[2048] = { 0 };
            int nRet = nl_SetNetDeviceConfig(configData, strlen(configData), 5000, outData);
            if (nRet != 0)
            {
                printf("nl_setNetDeviceConfig error\n");
            }
            printf("\n nl_setNetDeviceConfig outData=%s\n", outData);
        }

        bool isClosed = nl_CloseDevice(&hDevice); // Close the device
        printf("hDevice=%p,%s\n", hDevice, isClosed ? "succeed in closing the device" : "failed
to close the device");
        T_DeviceStatus t = nl_GetDevStatus(hDevice);
        printf("T_DeviceStatus t=%d\n", t);
```

```
        }
        nl_ReleaseDevices(&hDeviceList); // Release the device list handle
        Sleep(1000);



#if NET_TEST
        //----------net test------------
        if (argc >= 2 && strcmp(argv[1], "--NetGetImg") == 0) {
                char ip1[20] = { 0 };
                char ip2[20] = { 0 };
                char ip3[20] = { 0 };
                int td1 = 1, td2 = 2;
                int port2 = 36520;
                int port1 = 30000;
                strcpy(ip1, "192.168.3.193");
                thread t1(NetImageThread, ip1, &port1, &port2);
                strcpy(ip2, "192.168.3.219");
                thread t2(NetImageThread, ip2, &port1, &port2);
                strcpy(ip3, "192.168.3.197");
                thread t3(NetImageThread, ip3, &port1, &port2);
                t1.join();
                t2.join();
                t3.join();
                return 0;
```

```
    }
    else if (argc >= 2 && strcmp(argv[1], "--ServerMode") == 0) {
        // The network server only provides a simple mode for reference, and it is
recommended to implement the server mode according to specific requirements
        thread tt(ServerThread);
        tt.detach();
        Sleep(30000);
        nl_ExitTcpService();
        printf("exit\n");
        return 0;
    }
    //----------net test over-------
#endif


    // Network devices can be asynchronously refreshed in the background
    if (argc >= 2 && strcmp(argv[1], "--EnumNetDevAsyn") == 0) {
        printf("begin nl_BeginEnumNetDevice\n");
        nl_BeginEnumNetDevice();
        for (int i = 0; i < 16; i++) {
            hDeviceList = nl_EnumDevices(&deviceCounts);
            printf("------------asyn enum deviceCounts------------=[%d]\n", deviceCounts);
            Sleep(1000);
        }
        nl_StopEnumNetDevice();
```

```
        return 0;
    }



    printf("all over\n");
    system("pause");


    return 0;
}
```

3 Interface description

The SDK under Windows and Linux uses an API with the same name. The specific functions are as follows:

Function list	
Function	description
HANDLEDEVLST nl_EnumDevices(int* deviceCount, EnumType = ENUM_ALL);	brief:enumerate device. param[in] enumType Enumerate all types of devices by default param[out] deviceCount Number of device return:Device list handle Non-null: device list exists. Null: device list doesn't exist.
void nl_ReleaseDevices(HANDLEDEVLST* hDeviceList);	brief:Release the device list handle. param[in] hDeviceList Device list handle
HANDLEDEV nl_OpenDevice(const HANDLEDEVLST hDeviceList, unsigned int index, T_Porotocol porotocol = Nlscan);	brief:Specify the indexed device on the device list. param[in] hDeviceList Device list handle param[in] index device index param[in] porotocol Protocol of the manufacturer return:Device handle Non-null: succeed

	in opening. Null: failed to open.
bool nl_Write(const HANDLEDEV hDevice, const char* data, unsigned int len, bool isPacked = true);	brief:Write data to the device. param[in] hDevice Device handle param[in] data Written data param[in] len Data length param[in] isPacked Whether data is packed return:Whether data is written. true: succeed in writing data. false: failed to write data.
bool nl_WriteAsHex(const HANDLEDEV hDevice, const char* data, bool isPacked = false);	brief:Write data to the device in the form of HEX character string. param[in] hDevice Device handle param[in] data Written data param[in] isPacked Whether data is packed return:Whether data is written. true: succeed in writing data. false: failed to write data.
T_CommunicationResult nl_SendCommand(const HANDLEDEV hDevice, const char* command, unsigned int commandLen);	brief:Send control commands to the device (Commands will be packed according to different protocols inside the interface). param[in] hDevice Device handle param[in] command commands sent param[in] commandLen Command length return:Communication result
T_CommunicationResult nl_SendCommandAsHex(const HANDLEDEV hDevice, const char* command, unsigned int commandLen);	brief:Send control commands to the device in the form of HEX character string (Commands will be packed according to different protocols inside the interface). param[in] hDevice Device handle param[in] command Commands sent param[in] commandLen Command length return:Communication result
unsigned int nl_Read(const HANDLEDEV hDevice, char* buf, unsigned int len, unsigned int timeout);	brief:Read device data. param[in] hDevice Device handle param[out] buf data returned from the device

	param[in] len Received data length param[in] timeout Data reading timeout When it is set as 0, it continues reading until there is no returned data. return:Data length returned from the device
void nl_SetListener(const HANDLEDEV hDevice, readCallback callback);	brief:Set monitor. param[in] hDevice Device handle param[in] callback callback function
bool nl_StopListener(const HANDLEDEV hDevice);	brief:Stop monitoring device data. param[in] hDevice Device handle return:Whether monitoring device data is stopped. true: succeed in stopping monitoring. false: failed to stop monitoring.
bool nl_GetPicSize(const HANDLEDEV hDevice, unsigned int* width, unsigned int* height);	brief:Get the size of device image. param[in] hDevice Device handle param[out] width Image width param[out] height Image height return:Whether device image size is obtained. true: succeed in getting device image. false: failed to get device image.
bool nl_GetPicData(const HANDLEDEV hDevice, unsigned char* imgBuf, int imgBufLen);	brief:Get device image. param[in] hDevice Device handle param[out] imgBuf Image data param[in] imgBufLen Image data length return:Whether device image is obtained. true: succeed in getting device image. false: failed to get device image.
bool nl_UpdateKernelDevice(const HANDLEDEV hDevice, const char* strFileName, unsigned int reserved = 0, unsigned int* error = 0);	brief:Update device. param[in] hDevice Device handle param[in] strFileName path of firmware file param[in] reserved Reserved field param[out] error Error number returned after the update failed. return:Whether updating is successful. true: succeed in updating. false: failed to update.
bool nl_CloseDevice(HANDLEDEV* hDe	brief:Close the device.

vice);	param[in] hDevice Device handle return:Whether the device is closed.true: succeed in closing the device. false: failed to close the device.
bool nl_SavePicDataToFile(const char* bmpName, unsigned char* imgBuf, int width, int height, int flag);	brief:Encapsulate the collected image data into BMP format and save it as a file. param[in] bmpName bmp file name param[in] imgBuf Image buffer data param[in] width Image width param[in] height Image height param[in] flag Image bit depth or image quality level When saving a file as a BMP bitmap, the image bit depth is specified, with possible values of 8 or 24. When saving a file as a JPG, it represents the image quality level. gray image: (10-Low, 11-Middle, 12-High, 13-Highest) color image: (20-Low, 21-Middle, 22-High, 23-Highest) return:Whether it is saved.true: saved. false: failed to save.
T_DeviceStatus nl_GetDevStatus(const HANDLEDEV hDevice);	brief:Get device status. param[in] hDevice Device handle return:Device status
bool nl_ReadDevCfgToXml(const HANDLEDEV hDevice, const char* cfgFilePath);	brief:Read the configuration from the device and save it to the xml file. param[in] hDevice Device handle param[in] cfgFilePath Path of configuration file return:Whether it is saved.true: saved. false: failed to save.
bool nl_WriteCfgToDev(const HANDLEDEV hDevice, const char* cfgFilePath);	brief:Write the configuration file to the device. param[in] hDevice Device handle param[in] cfgFilePath Path of configuration file return:Whether it is written.true:

	written. false: failed to write.
void nl_SetCbDevStatusChanged(const HANDLEDEV hDevice, DevStatChgCallback callback);	brief:Set the callback function when device status changes. param[in] hDevice Device handle param[in] callback Callback function
bool nl_GetCommandResponse(const HANDLEDEV hDevice, const char* command, unsigned int commandLen, char* response, int *responseLen, unsigned int timeout, bool isPacked, bool isHex);	brief Send commands and receive return commands. param[in] hDevice Device handle param[in] command command sent param[in] commandLen command length param[out]response command response param[out]responseLen command response length param[in]timeout time out param[in]isPacked Whether data is packed param[in]isHex Whether data is Hex return true: successful. false: failed
bool nl_GetPicDataByConfig(const HANDLEDEV hDevice, STImgParam imgParam, unsigned char* imgBuf, unsigned int *imgBufLen, STImgResolution* imgR);	brief Retrieve image data based on the parameters param[in] hDevice Device handle param[in] imgParam image param set T, type: 0T - Real-time image (the latest captured image), 1T - Decoded successful image. F, Image format: 0F - Raw data, 1F - BMP, 2F - JPEG Q, JPEG quality level: 0Q - Low, 1Q - Middle, 2Q - High, 3Q - Highest Other parameters are temporarily reserved, initialized as 0x00 param[out]imgBuf The returned image data requires a sufficiently large space for reception param[out]imgBufLen returned image data real len param[out]imgR Keep the parameters, temporarily unused. The coordinates of the four endpoints of the barcode area, if available, require applying for an STImgResolution[4] array in advance. return true: successful. false: failed

IMG_TYPE nl_GetDeviceImageColorType(const HANDLEDEV hDevice, STImgResolution* imgResOut, unsigned int * imgLen);	brief Obtaining the image type of the device's raw image param[in] hDevice Device handle param[out] imgResOut The real resolution of the raw image, If it's a color image, it's the converted resolution. param[out] imgLen image data real length return image type
bool nl_ConvertImageColorSpace(const HANDLEDEV hDevice, unsigned char* imgBufIn, long imgBufInLen, STImgResolution imgResIn, unsigned char* imgBufOut);	brief Color space conversion of the raw image nv12->bgr param[in] hDevice Device handle param[in] imgBufIn Raw image data param[in] imgBufInLen Raw image data length param[in] imgResIn The real resolution of the raw image param[out] imgBufOut Image data after color space conversion return true: successful. false: failed
bool nl_GetDeviceInfo(const HANDLEDEVLST hDeviceList, unsigned int index, STDeviceInfo* stNetDevInfo);	brief Retrieve device information param[in] hDeviceList Device handle list param[in] index device index param[out] stNetDevInfo device information return true: successful. false: failed
bool nl_DeviceIsOpenByHandle(const HANDLEDEV hDevice);	brief Is the device open param[in] hDevice Device handle return true: open. false: close
bool nl_DeviceIsOpenByList(const HANDLEDEVLST hDeviceList, unsigned int index);	brief Is the device open param[in] hDeviceList Device handle list param[in] index device index return true: open. false: close
char *nl_GetLastError();	brief Retrieve the error message from the last operation return error message
void nl_BeginEnumNetDevice();	brief Start searching for network devices in the background return

void nl_StopEnumNetDevice();	brief Stop searching for network devices in the background return
int nl_SetNetDeviceConfig(char* inData,int inDataLen,int recTimeout,char* outdata);	brief Set network device configuration information param[in] inData configuration information param[in] inDataLen configuration information length param[in] recTimeout time out param[in] outdata Retrieve data return 0 successful other fail

以下为网络独立接口

int nl_CreateTcpService(int port, tcpServiceBack callback);	brief Create a network server. param[in] port network port param[in] callback Callback function return Less than 0 fail.
int nl_CloseClientSocket(int socket);	brief Close the client socket param[in] socket network socket return 0 successful other fail
int nl_ExitTcpService();	brief exit tcp service return
int nl_connectToService(char* serviceIp, int port, int* socket);	brief connect to tcp service param[in] serviceIp service ip param[in] port service port param[out] socket network socket return 0 successful other fail
int nl_sendDataToSocket(int socket, char* buf, int buf_len);	brief Send data by socket param[in]socket network socket param[in]buf send data param[in]buf_len send data length return 0 successful other fail
int nl_readFromSocket(int socket, int nTimeout, char* outbuf, int *buflen);	brief Receive network data param[in] socket network socket param[in] nTimeout time out param[in] outbuf Receive data param[in] buflen Receive data length return 0 successful other fail
int nl_getNetImgData(int socket, int T,	brief 通过网络获取图像数据

int R, int F, int Q, char *imgData, int *realLen, IMG_TYPE* imgtype, int *width, int *heigh);	param[in] socket 网络套接字 param[in] T type: 0T - Real-time image (the latest captured image), 1T - Decoded successful image. param[in] RImage ratio, Keep the parameters, temporarily unused, initialized as 0x00 param[in] F Image format: 0F - Raw data, 1F - BMP, 2F - JPEG param[in] Q JPEG quality level: 0Q - Low, 1Q - Middle, 2Q - High, 3Q - Highest param[out] imgData image data param[out] realLen image data length param[out] imgtype image type param[out] width image width param[out] heigh image heigh return 0 successful other fail

Enum Description

```
brief:Abnormal type.
enum T_ErrorType
{
    Success                 = 0,  ///< Normal.
    UnknownError            = 1,  ///< Unknown Error.
    NotExistError           = 2,  ///< The device doesn't exit.
    NotOpenError            = 3,  ///< The device is not opened.
    AlreadyOpenError        = 4,  ///< The device is opened.
    AccessDeniedError       = 5,  ///< Access to the device is denied.
    NotInitializedError       = 6,  ///< The Device is not initialized.
    InvalidParamsError        = 8,  ///< Invalid parameters.
    InvalidFileFormatError    = 9,  ///< Invalid file format.
    FileNameExtError        = 10,///< File name error.
    CommunicationError      = 11,///< Communication error.
    MallocError             = 12,///< Memory allocation error.
    UpdateFailedError       = 13,///< Failed to update.
    NoUpdateObjectError     = 14,///< No updating object.
    FileNotExistError       = 15,///< the file doesn't exist.
    BufferOverflowError     = 16,///< Buffer overflows.
```

```
    FileNotSuitableError     = 17,///< The file is not suitable.
    DeviceNotUniqueError     = 18,///< The device is not unique.
};
```

```
brief:Device status.
enum T_DeviceStatus
{
    Opened = 0,           ///< Opened.
    NotOpened,          ///< Not opened.
    Closed,               ///< Closed.
    NotClosed,          ///< Not closed.
    Updating,             ///< Updating...
    Updated,              ///< Updating is finished.
    Writing,              ///< Writing data...
    Written,              ///< Data writing is finished.
    Reading,              ///< Reading data...
    ReadOK,               ///< Data reading is finished.
    GettingPicData,       ///< Getting image data...
    GetPicDataOK,         ///< Image data has been ontained.
    UnknownStatus     ///< Unknown status.
};
```

```
brief:Commands sending result.
enum T_CommunicationResult
{
    SendError = 0,        ///< Sending error.
    Support,            ///< Commands supported.
    Unsupport,     ///< Commands not supported.
    OutOfRange,         ///< Data value is not within the range.
    UnknownResult,    ///< Unknown error.
};
```

```
brief:Protocol.
enum T_Porotocol
{
    Nlscan = 0, // Newland.
};
```