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-ScanHubForLinux

2.1 Directory Structure

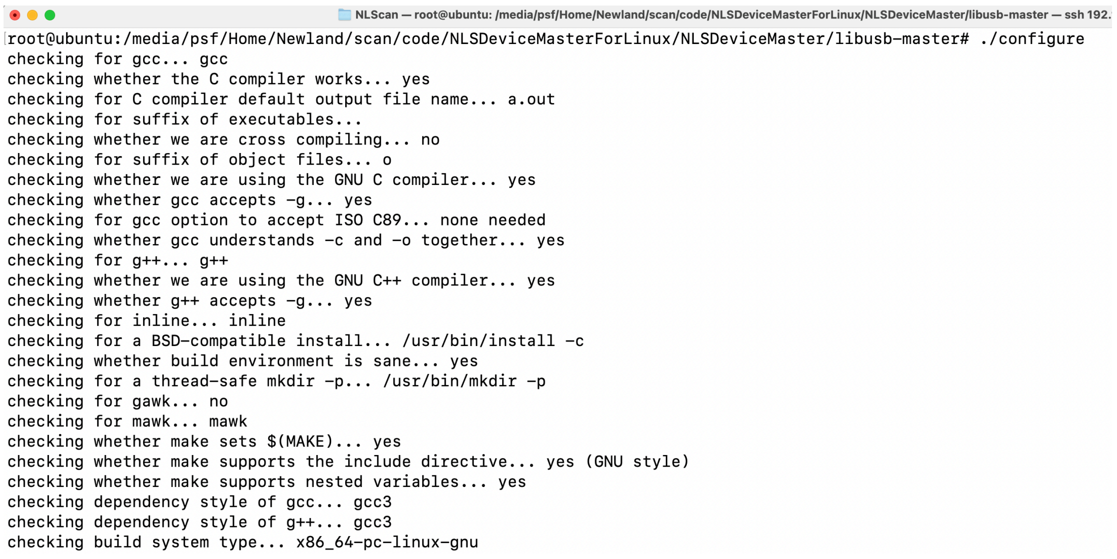
N-ScanHubForLinux offers the API under the linux platform, and its directory is shown as below.

Contents	Descriptions
libusb	The dynamic library depends on the source of third-party USB library
lib	64-bit N-ScanHub.a and N-ScanHub.so
include	Header file: N-ScanHub.h (all interfaces descriptions included)
demo	The source code of demo and executable file
help	Help document: N-ScanHub.pdf

2.2 Compile

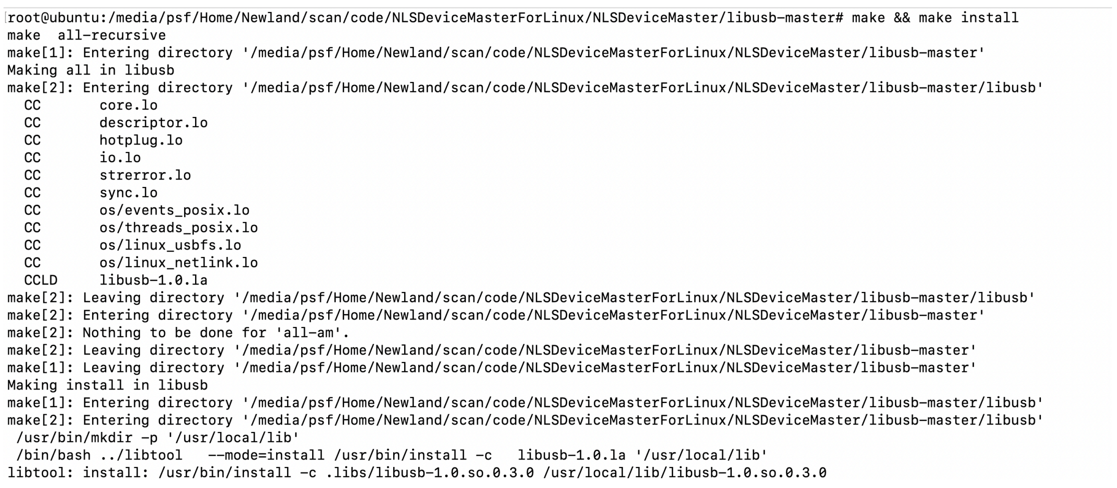
2.2.1 Compile the third-party library

./configure


NLScan — root@ubuntu: /media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master — ssh 192.

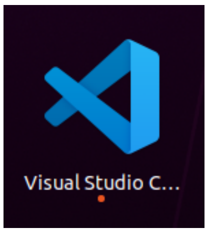
```
root@ubuntu:/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master# ./configure
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking whether gcc understands -c and -o together... yes
checking for g++... g++
checking whether we are using the GNU C++ compiler... yes
checking whether g++ accepts -g... yes
checking for inline... inline
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /usr/bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking whether make supports the include directive... yes (GNU style)
checking whether make supports nested variables... yes
checking dependency style of gcc... gcc3
checking dependency style of g++... gcc3
checking build system type... x86_64-pc-linux-gnu
```

make && make install



```
root@ubuntu:/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master# make && make install
make  all-recursive
make[1]: Entering directory '/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master'
Making all in libusb
make[2]: Entering directory '/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master/libusb'
  CC       core.lo
  CC       descriptor.lo
  CC       hotplug.lo
  CC       io.lo
  CC       strerror.lo
  CC       sync.lo
  CC       os/events_posix.lo
  CC       os/threads_posix.lo
  CC       os/linux_usbfs.lo
  CC       os/linux_netlink.lo
  CCLD     libusb-1.0.la
make[2]: Leaving directory '/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master/libusb'
make[2]: Entering directory '/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master'
make[2]: Nothing to be done for 'all-am'.
make[2]: Leaving directory '/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master'
make[1]: Leaving directory '/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master'
Making install in libusb
make[1]: Entering directory '/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master/libusb'
make[2]: Entering directory '/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/libusb-master/libusb'
 /usr/bin/mkdir -p '/usr/local/lib'
 /bin/bash ../libtool   --mode=install /usr/bin/install -c   libusb-1.0.la '/usr/local/lib'
libtool: install: /usr/bin/install -c .libs/libusb-1.0.so.0.3.0 /usr/local/lib/libusb-1.0.so.0.3.0
```

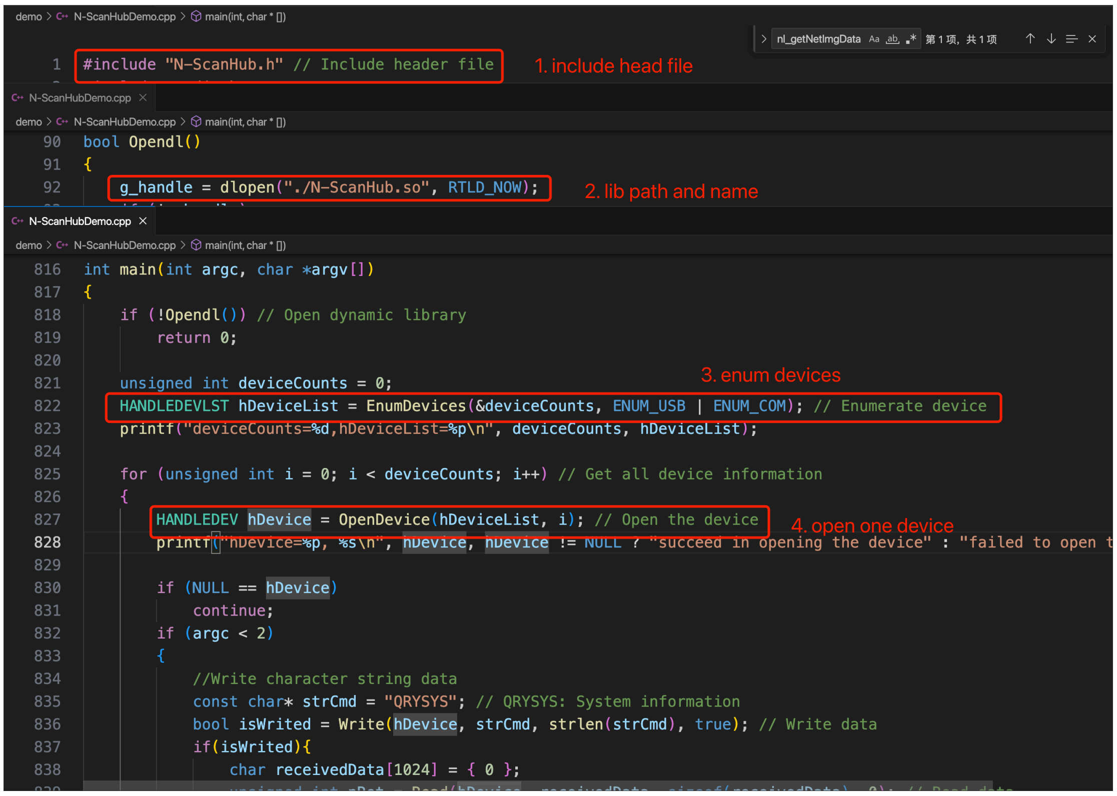
2.3 N-ScanHubForLinux Operating Instructions

Device: FM430 Tool: VSCode  System: Ubuntu 20.04.3 LTS

N-ScanHubForLinux demo Operating Steps

1. Copy the dynamic library N-ScanHub.so to the demo directory for future use

2. Start calling the functions in the SDK, shown as below.



```
#include "N-ScanHub.h" // Include header file
```

```
bool Opendl()
{
    g_handle = dlopen("./N-ScanHub.so", RTLD_NOW);
```

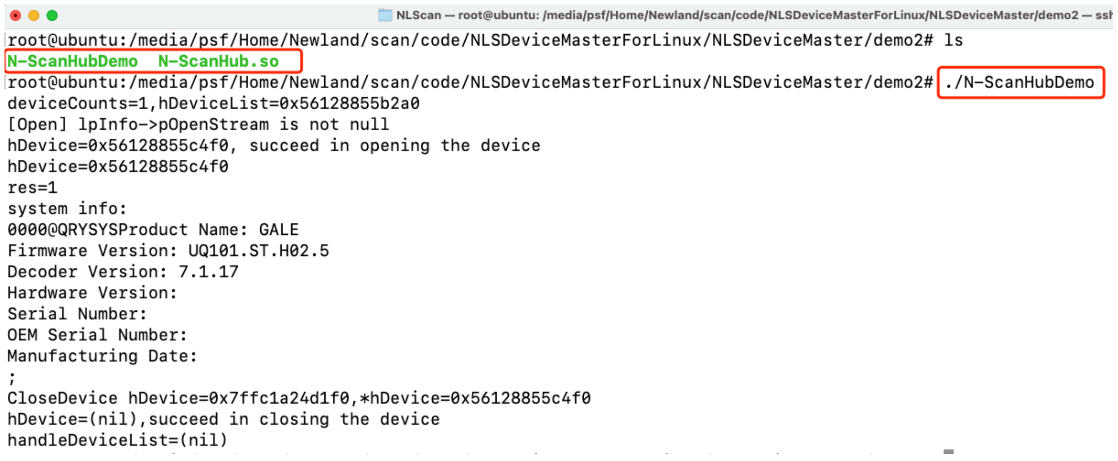
```
int main(int argc, char *argv[])
{
    if (!Opendl()) // Open dynamic library
        return 0;

    unsigned int deviceCounts = 0;
    HANDLEDEVLST hDeviceList = EnumDevices(&deviceCounts, ENUM_USB | ENUM_COM); // Enumerate device
    printf("deviceCounts=%d,hDeviceList=%p\n", deviceCounts, hDeviceList);

    for (unsigned int i = 0; i < deviceCounts; i++) // Get all device information
    {
        HANDLEDEV hDevice = OpenDevice(hDeviceList, i); // Open the device
        printf("hDevice=%p, %s\n", hDevice, hDevice != NULL ? "succeed in opening the device" : "failed to open t

        if (NULL == hDevice)
            continue;
        if (argc < 2)
        {
            //Write character string data
            const char* strCmd = "QRYSYS"; // QRYSYS: System information
            bool isWrited = Write(hDevice, strCmd, strlen(strCmd), true); // Write data
            if(isWrited){
                char receivedData[1024] = { 0 };
```

3. Start running the program: enter the make command at the terminal and then sudo ./ N-ScanHubDemo, shown as below.


NLScan — root@ubuntu: /media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/demo2 — ssh

```
root@ubuntu:/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/demo2# ls
N-ScanHubDemo  N-ScanHub.so
root@ubuntu:/media/psf/Home/Newland/scan/code/NLSDeviceMasterForLinux/NLSDeviceMaster/demo2# ./N-ScanHubDemo
deviceCounts=1,hDeviceList=0x56128855b2a0
[Open] lpInfo->pOpenStream is not null
hDevice=0x56128855c4f0, succeed in opening the device
hDevice=0x56128855c4f0
res=1
system info:
0000@QRYSYSProduct Name: GALE
Firmware Version: UQ101.ST.H02.5
Decoder Version: 7.1.17
Hardware Version:
Serial Number:
OEM Serial Number:
Manufacturing Date:
;
CloseDevice hDevice=0x7ffc1a24d1f0,*hDevice=0x56128855c4f0
hDevice=(nil),succeed in closing the device
handleDeviceList=(nil)
```

2.4 Example of demo

#include "N-ScanHub.h" // Include header file

#include <stdio.h>

#include <dlfcn.h>

#include <cstring>

#include <stdlib.h>

#include <fstream>

#include <unistd.h>

#include <string.h>

#include <ctime>

#include <thread>

using namespace std;

static void *g_handle = NULL; // Dynamic library handle

int ReadFromSocket(int socket, int nTimeout, char *outbuf, int *buflen);

```
int CreateTcpService(int port, tcpServiceBack callback)

{

        if (g_handle == NULL)

                return false;



        char *error = NULL;

        typedef int (*pf_t)(int, tcpServiceBack); // Declare function pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_CreateTcpService");



        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return false;

        }



        int isSended = pf(port, callback);

        return isSended;

}



int ExitTcpService()

{

        if (g_handle == NULL)

                return false;
```

```
    char *error = NULL;

    typedef int (*pf_t)(); // Declare function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_ExitTcpService");



    if ((error = dlerror()) != NULL)

    {

        fprintf(stderr, "%s\n", error);

        return false;

    }



    int isSended = pf();

    return isSended;

}



void TcpServiceBack(int clientSocket, char *clientIp)

{

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

    char buf[4096] = {0};

    int len = 4096;

    while (1)

    {

        if (ReadFromSocket(clientSocket, 2000, buf, &len) == 0)

        {
```

```
                        printf("TcpServiceBack buf=%s\n", buf);

                        memset(buf, 0, sizeof(buf));

                }

                usleep(500);

        }

}


// Read device data

void ReadCallback(const HANDLEDEV hDevice, const char *buf, int len)

{

        printf("Callback hDevice=%p,receivedDataLen=%d\nreceivedData=%s\n", hDevice, len, buf);

}


// Monitoring device state change

void 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);

}


/**

  * @brief Open the dynamic library.
  * @return Return the result of opening the dynamic library. true: succeed in    opening the
dynamic library; false: failed to open the dynamic library.

  */

bool Opendl()

{

        g_handle = dlopen("./N-ScanHub.so", RTLD_NOW);

        if (!g_handle)

        {

                fprintf(stderr, "%s\n", dlerror());

                return false;

        }



        return true;

}



/**

  * @brief Close dynamic library

  */

void Closedl()

{

        if (g_handle != NULL)

                dlclose(g_handle); // Close dynamic library calling handle

}
```

```
bool GetPicData(const HANDLEDEV hDevice, unsigned char *imgBuf, int imgBufLen)

{

        if (g_handle == NULL)

                return false;



        char *error = NULL;

        typedef bool (*pf_t)(const HANDLEDEV, unsigned char *, int); // Declare function pointer
type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_GetPicData");



        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return false;

        }



        bool ret = pf(hDevice, imgBuf, imgBufLen);

        return ret;

}



bool GetPicDataByConfig(const HANDLEDEV hDevice, STImgParam imgParam, unsigned char
*imgBuf, unsigned int *imgBufLen, STImgResolution *imgR)

{

        if (g_handle == NULL)

                return false;
```

```
    char *error = NULL;

    typedef bool (*pf_t)(const HANDLEDEV, STImgParam, unsigned char *, unsigned int *,
STImgResolution *); // Declare function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_GetPicDataByConfig");



    if ((error = dlerror()) != NULL)

    {

        fprintf(stderr, "%s\n", error);

        return false;

    }



    bool ret = pf(hDevice, imgParam, imgBuf, imgBufLen, imgR);

    return ret;

}



IMG_TYPE GetDeviceImageColorType(const HANDLEDEV hDevice, STImgResolution *imgResOut,
unsigned int *imgLen)

{

    if (g_handle == NULL)

        return TYPE_UNKNOW;



    char *error = NULL;

    typedef IMG_TYPE (*pf_t)(const HANDLEDEV, STImgResolution *, unsigned int *); // Declare
function pointer type
```

```
    pf_t pf = (pf_t)dlsym(g_handle, "nl_GetDeviceImageColorType");


    if ((error = dlerror()) != NULL)

    {

        fprintf(stderr, "%s\n", error);

        return TYPE_UNKNOW;

    }


    IMG_TYPE ret = pf(hDevice, imgResOut, imgLen);

    return ret;

}


bool ConvertImageColorSpace(const HANDLEDEV hDevice, unsigned char *imgBufIn, long
imgBufInLen, STImgResolution imgResIn, unsigned char *imgBufOut)

{

    if (g_handle == NULL)

        return false;


    char *error = NULL;

    typedef bool (*pf_t)(const HANDLEDEV, unsigned char *, long, STImgResolution, unsigned
char *); // Declare function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_ConvertImageColorSpace");


    if ((error = dlerror()) != NULL)

    {
        fprintf(stderr, "%s\n", error);

        return false;

    }



    bool ret = pf(hDevice, imgBufIn, imgBufInLen, imgResIn, imgBufOut);

    return ret;

}



bool GetPicSize(const HANDLEDEV hDevice, unsigned int *width, unsigned int *height)

{

    if (g_handle == NULL)

        return false;



    char *error = NULL;

    typedef unsigned (*pf_t)(const HANDLEDEV, unsigned int *, unsigned int *); // Declare
function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_GetPicSize");



    if ((error = dlerror()) != NULL)

    {

        fprintf(stderr, "%s\n", error);

        return false;

    }
```

```
        bool ret = pf(hDevice, width, height);

        return ret;

}



unsigned int Read(const HANDLEDEV hDevice, char *buf, unsigned int len, unsigned int timeout)

{

        if (g_handle == NULL)

                return false;



        char *error = NULL;

        typedef unsigned int (*pf_t)(const HANDLEDEV, char *, unsigned int, unsigned int); //
Declare function pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_Read");



        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return false;

        }



        unsigned int ret = pf(hDevice, buf, len, timeout);

        return ret;

}
```

```
bool GetCommandResponse(const HANDLEDEV hDevice, const char *command, unsigned int
commandLen, char *response, int *responseLen, unsigned int timeout, bool isPacked, bool isHex)

{

        if (g_handle == NULL)

                return false;



        char *error = NULL;

        typedef bool (*pf_t)(const HANDLEDEV, const char *, unsigned int, char *, int *, unsigned int,
bool, bool); // Declare function pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_GetCommandResponse");



        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return false;

        }

        printf("hDevice=%p\n", hDevice);



        bool isSended = pf(hDevice, command, commandLen, response, responseLen, timeout,
isPacked, isHex);

        return isSended;

}



bool Write(const HANDLEDEV hDevice, const char *data, unsigned int len, bool isPacked = true)

{
        if (g_handle == NULL)

                return false;



        char *error = NULL;

        typedef bool (*pf_t)(const HANDLEDEV, const char *, unsigned int, bool); // Declare function
pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_Write");



        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return false;

        }

        printf("hDevice=%p\n", hDevice);



        bool isSended = pf(hDevice, data, len, isPacked);

        return isSended;

}



bool WriteAsHex(const HANDLEDEV hDevice, const char *data, bool isPacked = false)

{

        if (g_handle == NULL)

                return false;
        char *error = NULL;

        typedef bool (*pf_t)(const HANDLEDEV, const char *, bool); // Declare function pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_WriteAsHex");


        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return false;

        }

        printf("hDevice=%p\n", hDevice);


        bool isSended = pf(hDevice, data, isPacked);

        return isSended;

}


T_CommunicationResult SendCommand(const HANDLEDEV hDevice, const char *command,
unsigned int commandLen)

{

        if (g_handle == NULL)

                return T_CommunicationResult::SendError;


        char *error = NULL;

        typedef T_CommunicationResult (*pf_t)(const HANDLEDEV, const char *, unsigned int); //
Declare function pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_SendCommand");
        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return T_CommunicationResult::SendError;

        }

        printf("hDevice=%p\n", hDevice);



        T_CommunicationResult result = pf(hDevice, command, commandLen);

        return result;

}



T_CommunicationResult SendCommandAsHex(const HANDLEDEV hDevice, const char *command,
unsigned int commandLen)

{

        if (g_handle == NULL)

                return T_CommunicationResult::SendError;



        char *error = NULL;

        typedef T_CommunicationResult (*pf_t)(const HANDLEDEV, const char *, unsigned int); //
Declare function pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_SendCommandAsHex");



        if ((error = dlerror()) != NULL)

        {
```

```
        fprintf(stderr, "%s\n", error);

        return T_CommunicationResult::SendError;

    }

    printf("hDevice=%p\n", hDevice);



    T_CommunicationResult result = pf(hDevice, command, commandLen);

    return result;

}



void SetListener(const HANDLEDEV hDevice, readCallback callback)

{

    if (g_handle == NULL)

        return;



    char *error = NULL;

    typedef bool (*pf_t)(const HANDLEDEV, readCallback); // Declare function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_SetListener");



    if ((error = dlerror()) != NULL)

    {

        fprintf(stderr, "%s\n", error);

        return;

    }

    printf("hDevice=%p\n", hDevice);
    pf(hDevice, callback);
}

void StopListener(const HANDLEDEV hDevice)
{
    if (g_handle == NULL)
        return;

    char *error = NULL;
    typedef bool (*pf_t)(const HANDLEDEV); // Declare function pointer type
    pf_t pf = (pf_t)dlsym(g_handle, "nl_StopListener");

    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);
        return;
    }
    printf("hDevice=%p\n", hDevice);

    pf(hDevice);
}

bool ReadDevCfgToXml(const HANDLEDEV hDevice, const char *cfgFilePath)
```

```
{
    if (g_handle == NULL)
        return false;

    char *error = NULL;
    typedef bool (*pf_t)(const HANDLEDEV, const char *); // Declare function pointer type
    pf_t pf = (pf_t)dlsym(g_handle, "nl_ReadDevCfgToXml");

    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);
        return false;
    }
    printf("hDevice=%p\n", hDevice);

    bool isok = pf(hDevice, cfgFilePath);
    return isok;
}

bool WriteCfgToDev(const HANDLEDEV hDevice, const char *cfgFilePath)
{
    if (g_handle == NULL)
        return false;
    char *error = NULL;

    typedef bool (*pf_t)(const HANDLEDEV, const char *); // Declare function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_WriteCfgToDev");


    if ((error = dlerror()) != NULL)

    {

        fprintf(stderr, "%s\n", error);

        return false;

    }

    printf("hDevice=%p\n", hDevice);


    bool isok = pf(hDevice, cfgFilePath);

    return isok;

}


void SetCbDevStatusChanged(const HANDLEDEV hDevice, DevStatChgCallback callback)

{

    if (g_handle == NULL)

        return;


    char *error = NULL;

    typedef bool (*pf_t)(const HANDLEDEV, DevStatChgCallback); // Declare function pointer
type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_SetCbDevStatusChanged");
```

```
        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return;

        }

        printf("hDevice=%p\n", hDevice);



        pf(hDevice, callback);

}



bool UpdateKernelDevice(const HANDLEDEV hDevice, const char *strFileName, unsigned int
reserved = 0, unsigned int *errorUpdate = 0)

{

        if (g_handle == NULL)

                return false;



        char *error = NULL;

        typedef unsigned (*pf_t)(const HANDLEDEV, const char *, unsigned int, unsigned int *); //
Declare function pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_UpdateKernelDevice");



        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);
        return false;

    }



    bool isUpdated = pf(hDevice, strFileName, 0, errorUpdate);

    return isUpdated;

}



bool GetDeviceInfo(const HANDLEDEVLST hDeviceList, unsigned int index, STDeviceInfo
*stNetDevInfo)

{

    if (g_handle == NULL)

        return false;



    char *error = NULL;

    typedef bool (*pf_t)(const HANDLEDEVLST, unsigned int index, STDeviceInfo *); // Declare
function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_GetDeviceInfo");



    if ((error = dlerror()) != NULL)

    {

        fprintf(stderr, "%s\n", error);

        return false;

    }



    bool isUpdated = pf(hDeviceList, index, stNetDevInfo);
    return isUpdated;
}


bool CloseDevice(HANDLEDEV *hDevice)
{
    if (g_handle == NULL)
        return false;

    char *error = NULL;
    typedef unsigned (*pf_t)(HANDLEDEV *); // Declare function pointer type
    pf_t pf = (pf_t)dlsym(g_handle, "nl_CloseDevice");
    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);
        return false;
    }

    printf("CloseDevice hDevice=%p,*hDevice=%p\n", hDevice, *hDevice);
    bool isClosed = pf(hDevice);
    return isClosed;
}


HANDLEDEV OpenDevice(const HANDLEDEVLST hDeviceList, unsigned int index, T_Porotocol
porotocol = Nlscan)
```

```
{
        if (g_handle == NULL)
                return NULL;


        char *error = NULL;
        typedef HANDLEDEV (*pf_t)(const HANDLEDEVLST, unsigned, T_Porotocol); // Declare
function pointer type
        pf_t pf = (pf_t)dlsym(g_handle, "nl_OpenDevice");
        if ((error = dlerror()) != NULL)
        {
                fprintf(stderr, "%s\n", error);
                return NULL;
        }


        HANDLEDEV isOpened = pf(hDeviceList, index, porotocol);
        return isOpened;
}


void ReleaseDevices(HANDLEDEVLST *deviceList)
{
        if (g_handle == NULL)
                return;


        char *error = NULL;
```

```
    typedef void (*pf_t)(HANDLEDEVLST *); // Declare function pointer type
    pf_t pf = (pf_t)dlsym(g_handle, "nl_ReleaseDevices");

    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);
        return;
    }

    return pf(deviceList);
}

HANDLEDEVLST EnumDevices(unsigned int *deviceCounts, EnumType enumType)
{
    if (g_handle == NULL)
        return 0;

    char *error = NULL;
    typedef HANDLEDEVLST (*pf_t)(unsigned int *, EnumType); // Declare function pointer type
    pf_t pf = (pf_t)dlsym(g_handle, "nl_EnumDevices");

    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);
        return 0;
    }

    return pf(deviceCounts, enumType);
}

void BeginEnumNetDevice()
{
    if (g_handle == NULL)
        return;

    char *error = NULL;
    typedef void (*pf_t)(); // Declare function pointer type
    pf_t pf = (pf_t)dlsym(g_handle, "nl_BeginEnumNetDevice");

    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);
        return;
    }

    return pf();
}
```

```
void StopEnumNetDevice()

{

        if (g_handle == NULL)

                return;



        char *error = NULL;

        typedef void (*pf_t)(); // Declare function pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_StopEnumNetDevice");



        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return;

        }



        return pf();

}



int SetNetDeviceConfig(char *inData, int inDataLen, int recTimeout, char *outdata)

{

        if (g_handle == NULL)

                return -1;



        char *error = NULL;
```

```
    typedef int (*pf_t)(char *, int, int, char *); // Declare function pointer type
    pf_t pf = (pf_t)dlsym(g_handle, "nl_SetNetDeviceConfig");

    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);
        return -1;
    }

    return pf(inData, inDataLen, recTimeout, outdata);
}

bool SavePicDataToFile(const char *bmpName, unsigned char *imgBuf, int width, int height, int
biBitCount = 8)
{
    if (g_handle == NULL)
        return false;

    char *error = NULL;
    typedef bool (*pf_t)(const char *, unsigned char *, int, int, int); // Declare function pointer
type
    pf_t pf = (pf_t)dlsym(g_handle, "nl_SavePicDataToFile");

    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);

        return false;

    }



    bool isSaved = pf(bmpName, imgBuf, width, height, biBitCount);

    return isSaved;

}



int ConnectToService(char *serviceIp, int port, int *retSocket)

{

    if (g_handle == NULL)

        return -1;



    char *error = NULL;

    typedef int (*pf_t)(char *, int, int *); // Declare function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_connectToService");



    if ((error = dlerror()) != NULL)

    {

        fprintf(stderr, "%s\n", error);

        return -1;

    }



    return pf(serviceIp, port, retSocket);
```

```
}


int SendDataToSocket(int socket, char *buf, int buf_len)
{
    if (g_handle == NULL)
        return -1;

    char *error = NULL;
    typedef int (*pf_t)(int, char *, int); // Declare function pointer type
    pf_t pf = (pf_t)dlsym(g_handle, "nl_sendDataToSocket");

    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);
        return -1;
    }

    return pf(socket, buf, buf_len);
}

int ReadFromSocket(int socket, int nTimeout, char *outbuf, int *buflen)
{
    if (g_handle == NULL)
        return -1;
    char *error = NULL;

    typedef int (*pf_t)(int, int, char *, int *); // Declare function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_readFromSocket");


    if ((error = dlerror()) != NULL)

    {

        fprintf(stderr, "%s\n", error);

        return -1;

    }


    return pf(socket, nTimeout, outbuf, buflen);

}


int GetNetImgData(int socket, int T, int R, int F, int Q, char *imgData, int *realLen, IMG_TYPE
*imgtype, int *width, int *heigh)

{

    if (g_handle == NULL)

        return -1;


    char *error = NULL;

    typedef int (*pf_t)(int, int, int, int, int, char *, int *, IMG_TYPE *, int *, int *); // Declare
function pointer type

    pf_t pf = (pf_t)dlsym(g_handle, "nl_getNetImgData");
```

```
        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return -1;

        }



        return pf(socket, T, R, F, Q, imgData, realLen, imgtype, width, heigh);

}



int CloseClientSocket(int socket)

{

        if (g_handle == NULL)

                return -1;



        char *error = NULL;

        typedef int (*pf_t)(int); // Declare function pointer type

        pf_t pf = (pf_t)dlsym(g_handle, "nl_CloseClientSocket");



        if ((error = dlerror()) != NULL)

        {

                fprintf(stderr, "%s\n", error);

                return -1;

        }
        return pf(socket);
}


void NetImageThread(char *ip, int *port1, int *port2)
{
        int socket36520 = -1;
        int socket30000 = -1;
        char sendbuf[1024] = {0};
        char recvbuf[1024] = {0};
        int realLen = 0, nRet = -1;

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

        nRet = ConnectToService(ip, *port1, &socket30000);
        if (nRet != 0)
        {
                printf("connect 30000 error\n");
                return;
        }
        nRet = ConnectToService(ip, *port2, &socket36520);
        if (nRet != 0)
        {
                printf("connect 36520 error\n");
                return;
```

```
}
const int RECV_BUFFER_SIZE = 1920 * 1080 * 4;
char *recvBuffer = (char *)malloc(RECV_BUFFER_SIZE);

IMG_TYPE imgtype;
int w, h, f, q;
f = 2;
q = 2;
char filename[128] = {0};

for (int i = 0; i < 50; i++)
{
        memset(recvBuffer, 0, RECV_BUFFER_SIZE);
        memset(recvbuf, 0, 1024);

        if (SendDataToSocket(socket30000, sendbuf, 3) != 0)
        {
                printf("nl_sendDataToSocket error\n");
                continue;
        }

        if (ReadFromSocket(socket30000, 2000, recvbuf, &realLen) != 0)
        {
                printf("nl_readFromSocket error\n");
        continue;
    }
    printf("code length=%d,code=%s\n", realLen, recvbuf);


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


    printf("-------------------------------------ip=%s [%d][%d]\n", ip, nRet, realLen);
    if (nRet != 0)
        continue;
    if (f == 0)
    {
        if (imgtype == TYPE_COLOR)
        {
            sprintf(filename, "./pic/f0-%s-%04d.bmp", ip, i);


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


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


            SavePicDataToFile(filename, (unsigned char *)recvBuffer, w, h, 23); // Save
image
            printf("nl_SavePicDataToFile jpg end");
        }
        else
```

```
                {
                        sprintf(filename, "./pic/f0-%s-%04d.bmp", ip, i);
                        SavePicDataToFile(filename, (unsigned char *)recvBuffer, w, h, 8); // Save
image
                        sprintf(filename, "./pic/f0-%s-%04d.jpg", ip, i);
                        SavePicDataToFile(filename, (unsigned char *)recvBuffer, w, h, 13); // Save
image
                }
        }
        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);
                printf("write %s %d success\n", filename, realLen);
        }
        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(3);

    }

    CloseClientSocket(socket36520);

    CloseClientSocket(socket30000);

    free(recvBuffer);

    recvBuffer = NULL;

    printf("-----------------------------------ip=%s close\n", ip);

}


void ServerThread()
```

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


int main(int argc, char *argv[])
{
        if (!Opendl()) // Open dynamic library
                return 0;


        unsigned int deviceCounts = 0;
        HANDLEDEVLST hDeviceList = EnumDevices(&deviceCounts, ENUM_USB | ENUM_COM); //
Enumerate device
        printf("deviceCounts=%d,hDeviceList=%p\n", deviceCounts, hDeviceList);


        for (unsigned int i = 0; i < deviceCounts; i++) // Get all device information
        {
                HANDLEDEV hDevice = 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;
                if (argc < 2)
                {
```

```
            //Write character string data

            const char* strCmd = "QRYSYS"; // QRYSYS: System information

            bool isWrited = Write(hDevice, strCmd, strlen(strCmd), true); // Write data

            if(isWrited){

                char receivedData[1024] = { 0 };

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

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

            bool isWrited = WriteAsHex(hDevice, strCmdhEX, false);                        //
Write data

            if (isWrited)

            {

                char receivedData[1024] = {0};

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

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

            }
        }

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

        {

            const char *strCmd = "QRYSYS"; // QRYSYS: System information

            char receivedData[1024] = {0};

            int recvlen = 0;

            bool res = GetCommandResponse(hDevice, strCmd, strlen(strCmd), receivedData,
&recvlen, 0, true, false);

            printf("0-----res=%d-------system info: \n%s\n", res, receivedData);

        }

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

        {

            const char *strCmd = "QRYSYS";
// QRYSYS: System information

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

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

    }

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

    {

        unsigned int imgWidth = 0, imgHeight = 0;

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

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

        {

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

            const int RECV_BUFFER_SIZE = imgWidth * imgHeight * 4;

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



            STImgParam imgParam;

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

            imgParam.f = 2;

            imgParam.q = 3;

            STImgResolution imgR[4];

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

            unsigned int nRealLen = 0;

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

            printf("isOK=%d, recvBuffer1=%02x recvBuffer1=%02x\n", isOK,
recvBuffer[RECV_BUFFER_SIZE - 2], recvBuffer[RECV_BUFFER_SIZE - 1]);
```

```
                    char filename[128] = {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)
                {
                    STImgResolution imgResIn, imgResOut;
                    imgResIn.width = imgWidth;
                    imgResIn.height = imgHeight;
                    unsigned int imgLen = 0;
                    IMG_TYPE type = GetDeviceImageColorType(hDevice, &imgResOut,
&imgLen);
                    printf("ConvertImageColorSpace IMG_TYPE=%d\n", type);
```

```
        if (type == TYPE_COLOR)
        {
            unsigned char *outBuf = (unsigned char *)malloc(imgLen);
            bool res = ConvertImageColorSpace(hDevice, recvBuffer,
RECV_BUFFER_SIZE, imgResIn, outBuf);
            printf("ConvertImageColorSpace res=%d\n", res);
            if (res == false)
                return 0;
            int oWidth, oHeight;
            if (strlen(imgParam.b) != 0)
            { // If you are capturing a partial image, use the resolution of
the captured portion
                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, "test2%d.bmp", i);
            SavePicDataToFile(filename, outBuf, oWidth, oHeight, 24); //
Save image
            sprintf(filename, "test2%d.jpg", i);
```

```
                SavePicDataToFile(filename, outBuf, oWidth, oHeight, 23); //
Save image
            }
            else
            {
                int oWidth, oHeight;
                if (strlen(imgParam.b) != 0)
                { // If you are capturing a partial image, use the resolution of
the captured portion
                    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, "test1%d.bmp", i);
                SavePicDataToFile(filename, recvBuffer, oWidth, oHeight, 8); //
Save image
                sprintf(filename, "test1%d.jpg", i);
                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
        {
            SetListener(hDevice, ReadCallback);
            sleep(50);
            StopListener(hDevice);
        }
        else if (argc >= 3 && strcmp(argv[1], "--ReadDevCfgToXml") == 0) // Read the
configuration from the device and save it to the xml file.
        {
            bool isok = ReadDevCfgToXml(hDevice, argv[2]);
            printf(isok ? "ReadDevCfgToXml succeeded\n" : "ReadDevCfgToXml failed\n");
        }
        else if (argc >= 3 && strcmp(argv[1], "--WriteCfgToDev") == 0) // Read the configuration
from the device and save it to the xml file.
        {
            bool isok = WriteCfgToDev(hDevice, argv[2]);
            printf(isok ? "WriteCfgToDev succeeded\n" : "WriteCfgToDev failed\n");
        }
        else if (argc >= 2 && strcmp(argv[1], "--SetCbDevStatusChanged") == 0) // Set the
callback function when the device status changes.

        {

            SetCbDevStatusChanged(hDevice, DevStatChangeCallback);

            sleep(50);

            printf("SetCbDevStatusChanged finish\n");

        }

        else if (argc >= 3 && strcmp(argv[1], "--UpdateFirmware") == 0) // Update device

        {

            unsigned updateError = -1;

            bool isUpdated = UpdateKernelDevice(hDevice, argv[2], 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], "--GetDeviceInfo") == 0) // Write data to the device
in HEX character string
        {
            STDeviceInfo info;
            memset(&info, 0, sizeof(STDeviceInfo));
            GetDeviceInfo(hDeviceList, i, &info);
            printf("GetDeviceInfo -------- info\n %s\ntype=%d\n", info.devInfo, info.devType);
        }
        else if (argc >= 2 && strcmp(argv[1], "--SetNetDeviceConfig") == 0)
        {
            char configData[2048] = {0};
            strcpy(configData, "Serial Number=N5BC00202NOM;MAC
Address=E0:5A:9F:8E:D1:33;Device Use DHCP=1;Device IP Address=192.168.3.193;Device
SubNetmask=255.255.255.0;Device Gateway Address=192.168.3.1;");
            char outData[2048] = {0};
            int nRet = SetNetDeviceConfig(configData, strlen(configData), 5000, outData);
            if (nRet != 0)
            {
                printf("nl_setNetDeviceConfig error\n");
            }
            printf("\n nl_setNetDeviceConfig outData=%s\n", outData);
        }

        bool isClosed = CloseDevice(&hDevice); // Close the device
        printf("hDevice=%p,%s\n", hDevice, isClosed ? "succeed in closing the device" : "failed to close the device");
    }

    ReleaseDevices(&hDeviceList); // Release the device list handle
    printf("handleDeviceList=%p\n", hDeviceList);

    if (argc >= 2 && strcmp(argv[1], "--NetGetImg") == 0)
    {
        char ip1[20] = {0};
        char ip2[20] = {0};
        char ip3[20] = {0};
        int port2 = 36520;
        int port1 = 30000;

        strcpy(ip1, "192.168.3.205");
        thread t1(NetImageThread, ip1, &port1, &port2);

        strcpy(ip2, "192.168.3.199");
        thread t2(NetImageThread, ip2, &port1, &port2);

        strcpy(ip3, "192.168.3.197");
        thread t3(NetImageThread, ip3, &port1, &port2);
        t1.join();

        t2.join();

        t3.join();

}

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

{

        thread tt(ServerThread);

        tt.detach();

        sleep(30);

        ExitTcpService();

        printf("exit\n");

}

// Network devices can be asynchronously refreshed in the background

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

{

        BeginEnumNetDevice();

        for (int i = 0; i < 15; i++)

        {

                hDeviceList = EnumDevices(&deviceCounts, ENUM_ALL);

                printf("asyn enum deviceCounts=%d\n", deviceCounts);

                for (unsigned int j = 0; j < deviceCounts; j++)

                {

                        STDeviceInfo info;

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

```
                    GetDeviceInfo(hDeviceList, j, &info);

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

               }

               sleep(1);

          }

          StopEnumNetDevice();

          return 0;

     }

     Closedl();

     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*	brief:Release the device list handle. param[in] hDeviceList Device list handle

hDeviceList);	
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 vice);	brief:Close the device. 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 hD evice);	brief:Get device status. param[in] hDevice Device handle return:Device status
bool nl_ReadDevCfgToXml(const HANDL EDEV 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[in/out] responseLen [in]The length of response allocation space [out] 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[in/out] imgBufLen [in]The length of imgBuf allocation space [out] image data length 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
int nl_SetNetDeviceConfig(NET_SETTING_TYPE type, char* inData,int inDataLen,int recTimeout,char* outdata);	brief Set network device configuration information param[in] type setting type 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, int R, int F, int Q, char *imgData, int *realLen, IMG_TYPE* imgtype, int *width, int *heigh);	brief 通过网络获取图像数据 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[in/out] realLen [in]The length of imgData allocation space [out] 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.
};
```

```
brief:Image color types
enum IMG_TYPE {
    TYPE_UNKNOW = 0, ///< unknow type
    TYPE_GRAY = 1,     ///< gray
    TYPE_COLOR = 2   ///< color
};
```

```
brief device types
enum NL_DEVICE_TYPE {
    DEV_TYPE_UNKNOW = 0,  ///< unknow
    DEV_TYPE_USB = 1,        ///< usb
    DEV_TYPE_COM = 2,       ///< serial
    DEV_TYPE_NET = 3,        ///< network
};
```

```
brief The type of network parameter setting.
enum NET_SETTING_TYPE {
    DEV_SETTING = 0,      ///< Device network parameters
    GROUP_SETTING = 1,  ///< Network group parameters
};
```