China Daheng Group, Inc. Beijing Image Vision Technology Branch

C SDK
Programmer's Guide

Version：V2.0.4
Date：2019-02-21

DAHENG IMAGING 大恒图像

Notice

Web: http://www.daheng-imaging.com

Sales Email: isales@daheng-imaging.com

Sales Tel: +86 10 8282 8878

Support Email: isupport@daheng-imaging.com

Contents

1. Camera's Work Flow

1.1. Overall working flow

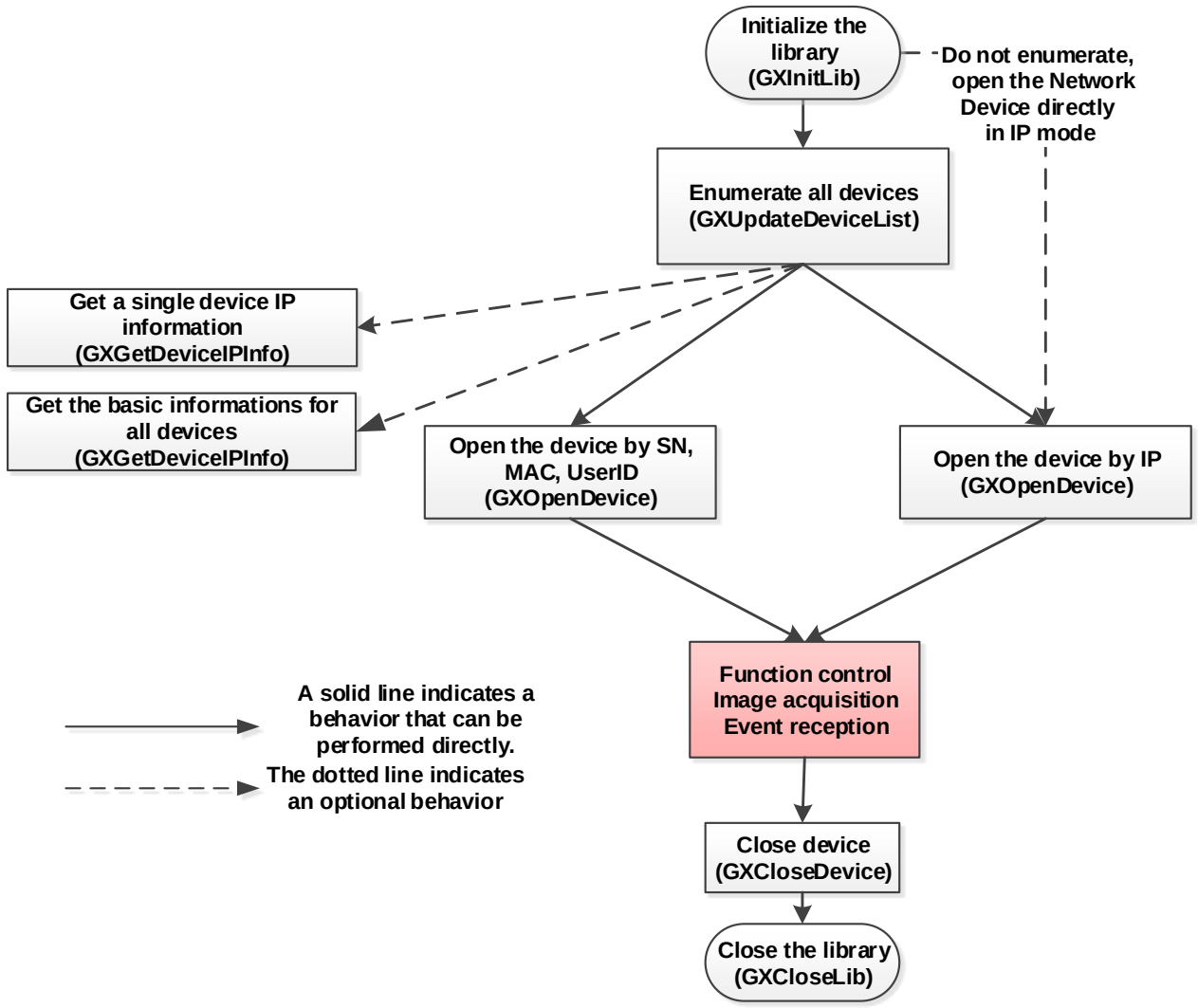

Figure 1: Overall working flow

1.2. Camera control flow

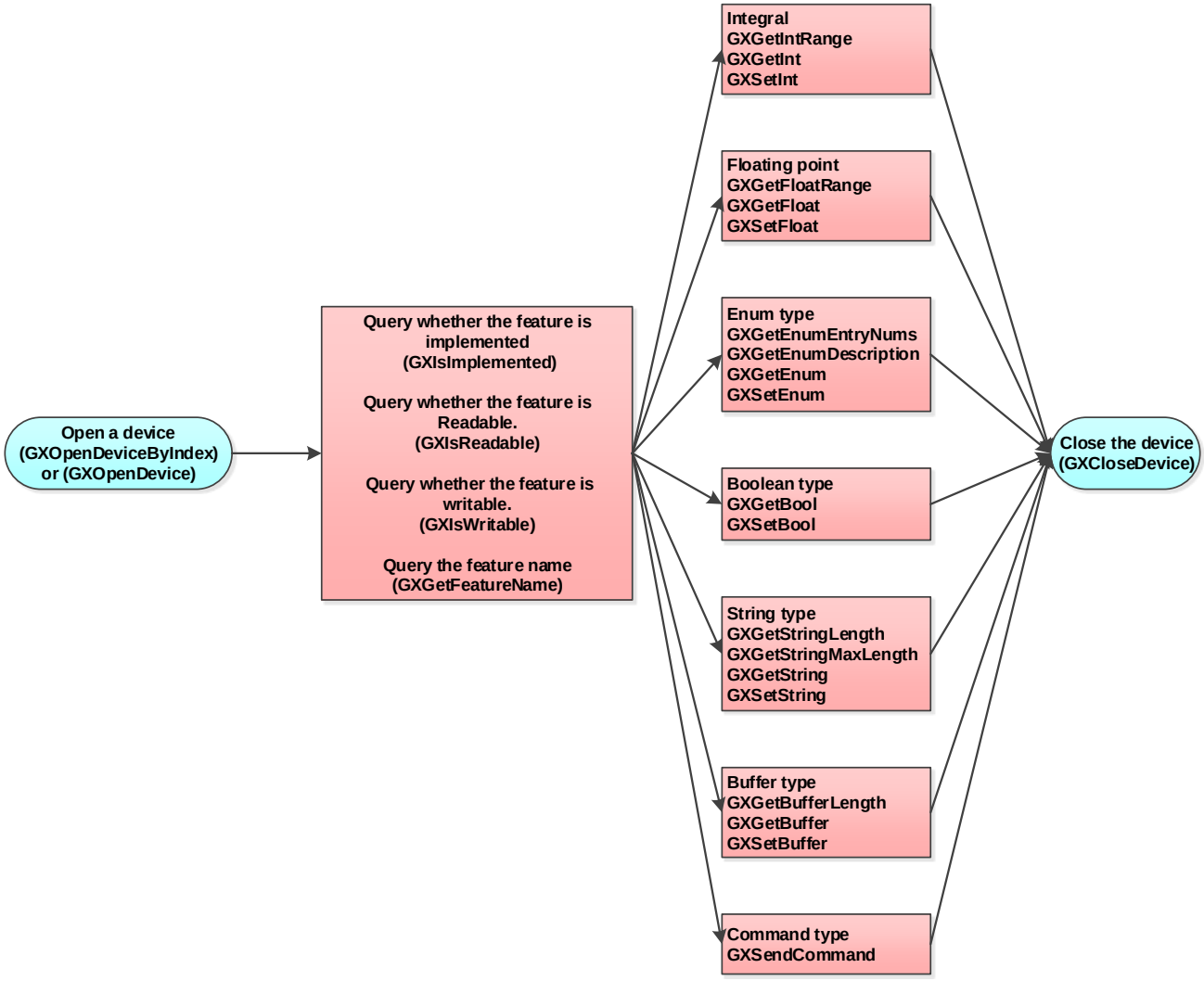

Figure 2: Camera control flow

1.3. DQBuf acquisition flow (zero copy, Linux only)

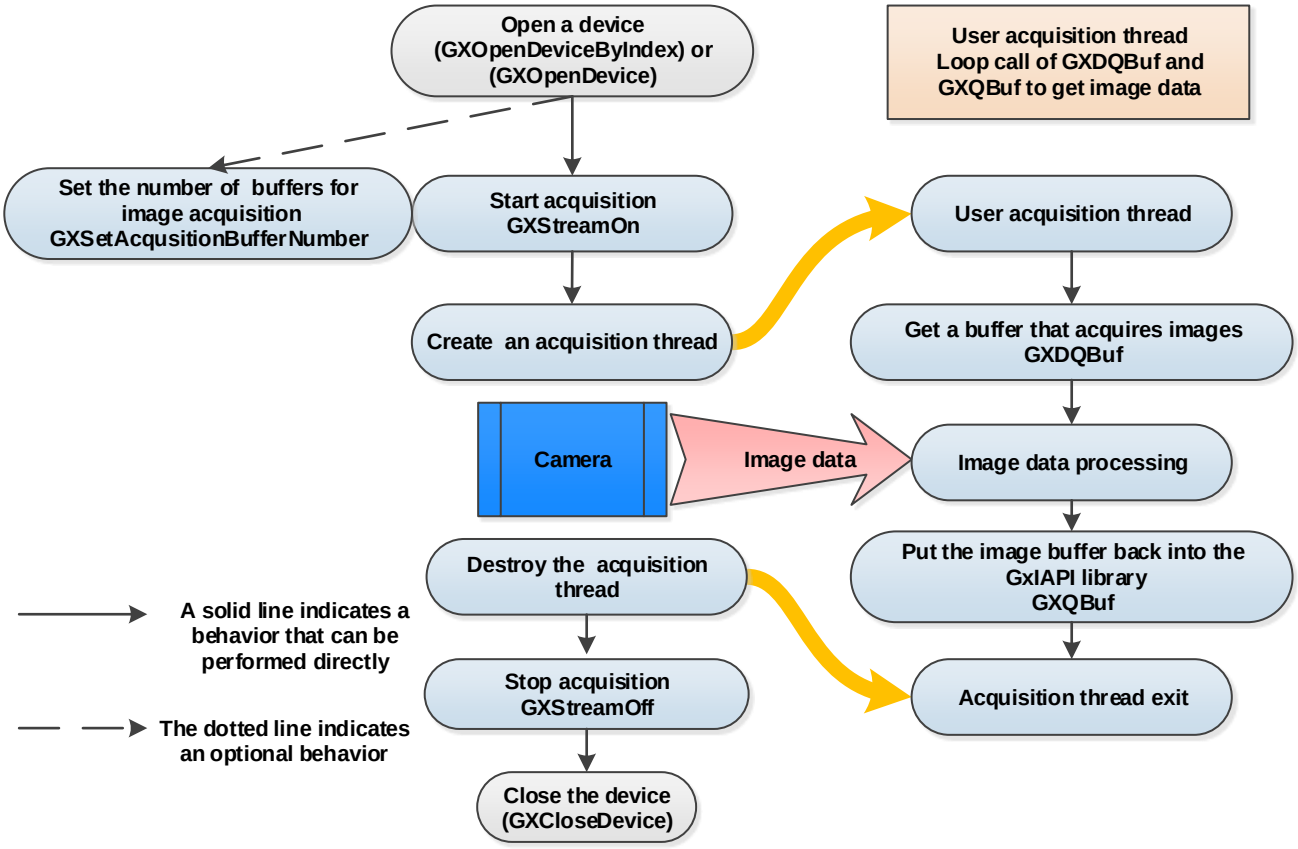

Figure 3: DQBuf acquisition flow

Precautions:

1) The user acquisition thread must be started after the start of the acquisition and be destroyed before the acquisition is stopped;
2) After calling the GXQBuf interface and putting the image buffer back into the GxIAPI library, you can no longer access the image buffer pointer;
3) The GXStreamOff interface will put all image buffers obtained by GXDQBuf back into the GxIAPI library, and then users can no longer access these image buffers;
4) The above flowchart demonstrates the way to start the thread for image acquisition, or you can call GXDQBuf directly to acquire images without creating a thread.

1.4. DQAllBufs acquisition flow (zero copy, Linux only)

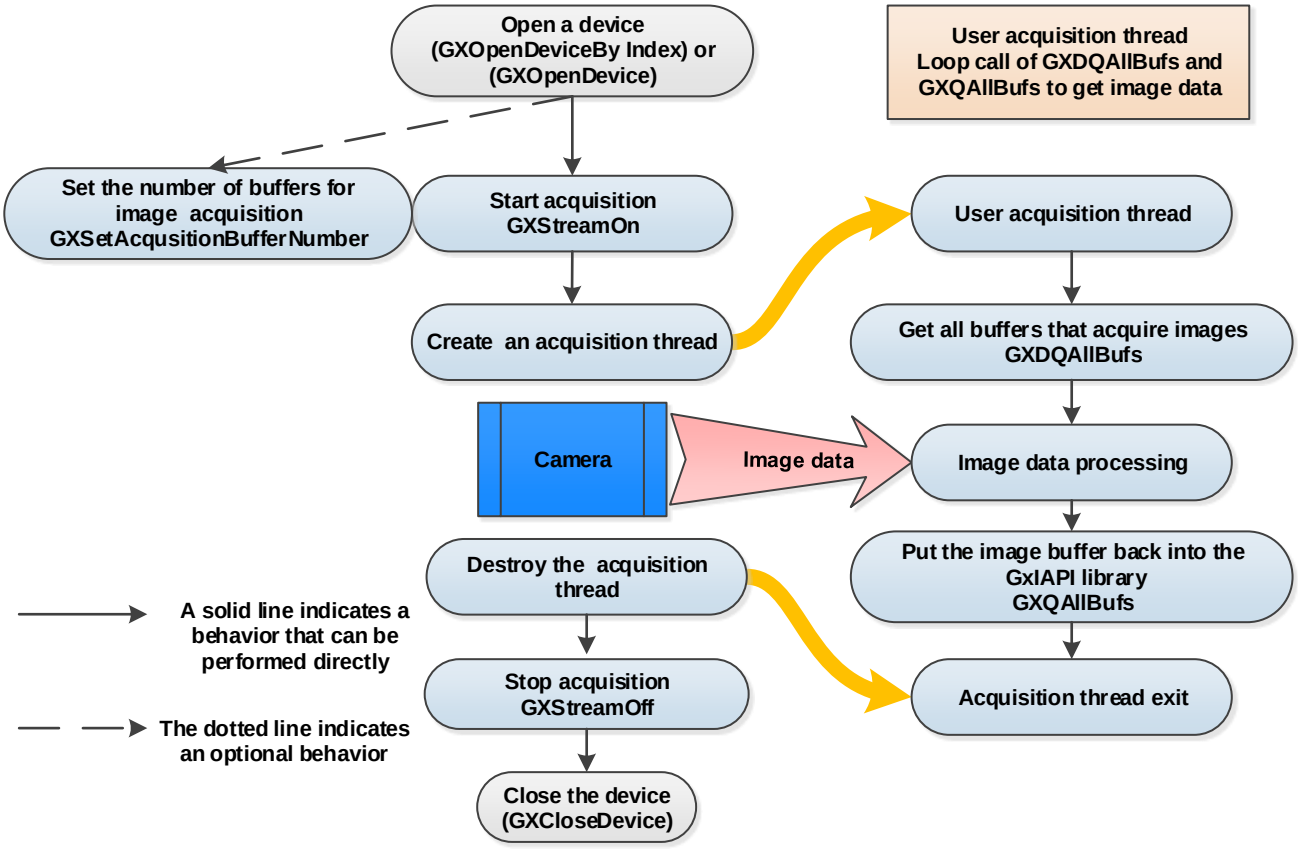

Figure 4: DQAllBufs acquisition flow

Precautions:

1) The user acquisition thread must be started after the start of the acquisition and be destroyed before the acquisition is stopped;
2) The GXStreamOff interface will put all image buffers obtained by GXQAllBufs back into the GxIAPI library, and then users can no longer access these image buffers;
3) The above process demonstrates the way to start the thread for image acquisition, or you can call GXDQAllBufs to acquire images without creating a thread.

Recommended scenarios :

1) In the case of having delay with image processing or display, the delay can be alleviated by only processing or displaying the latest image in the array.

1.5. Capture callback flow

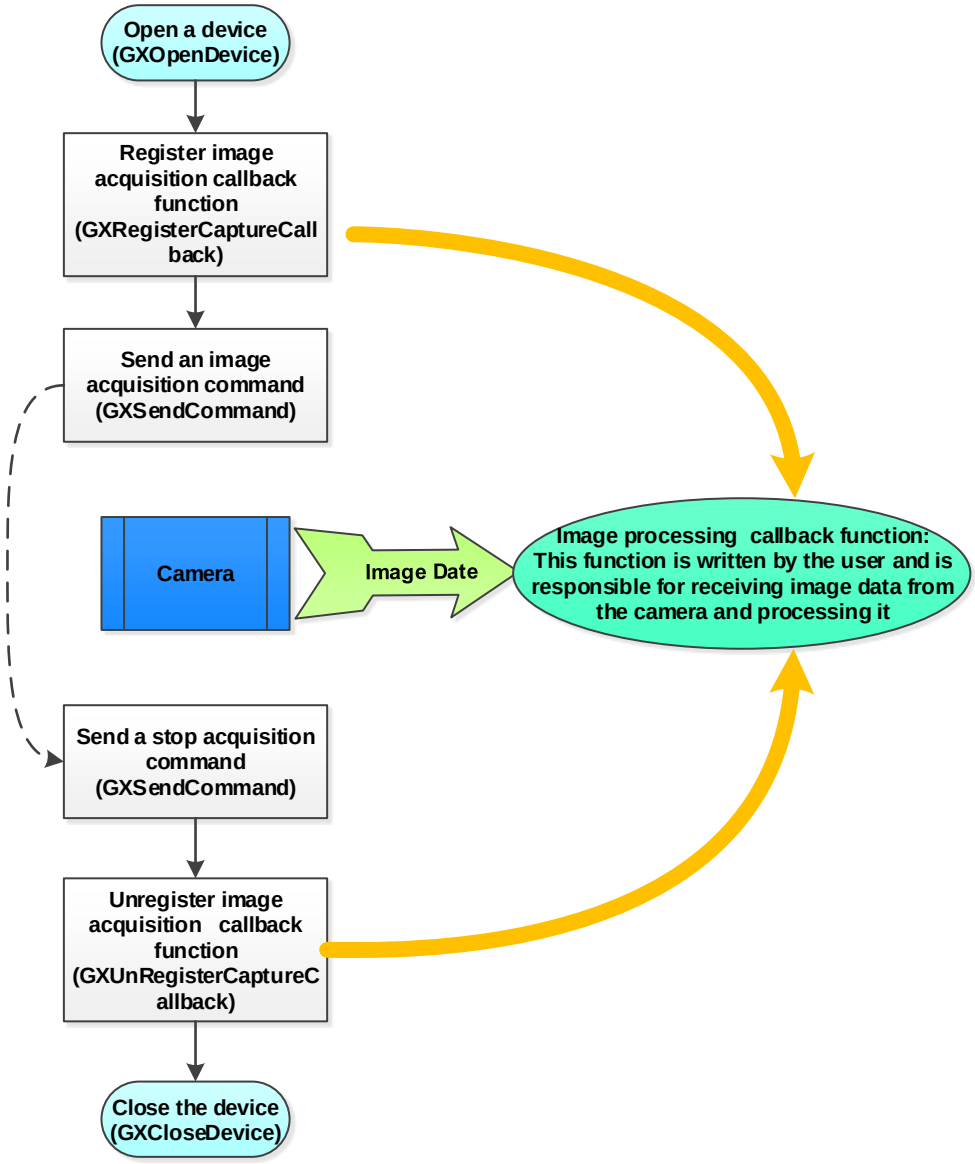

Figure 5: Capture callback flow

1.6. Offline events get flow

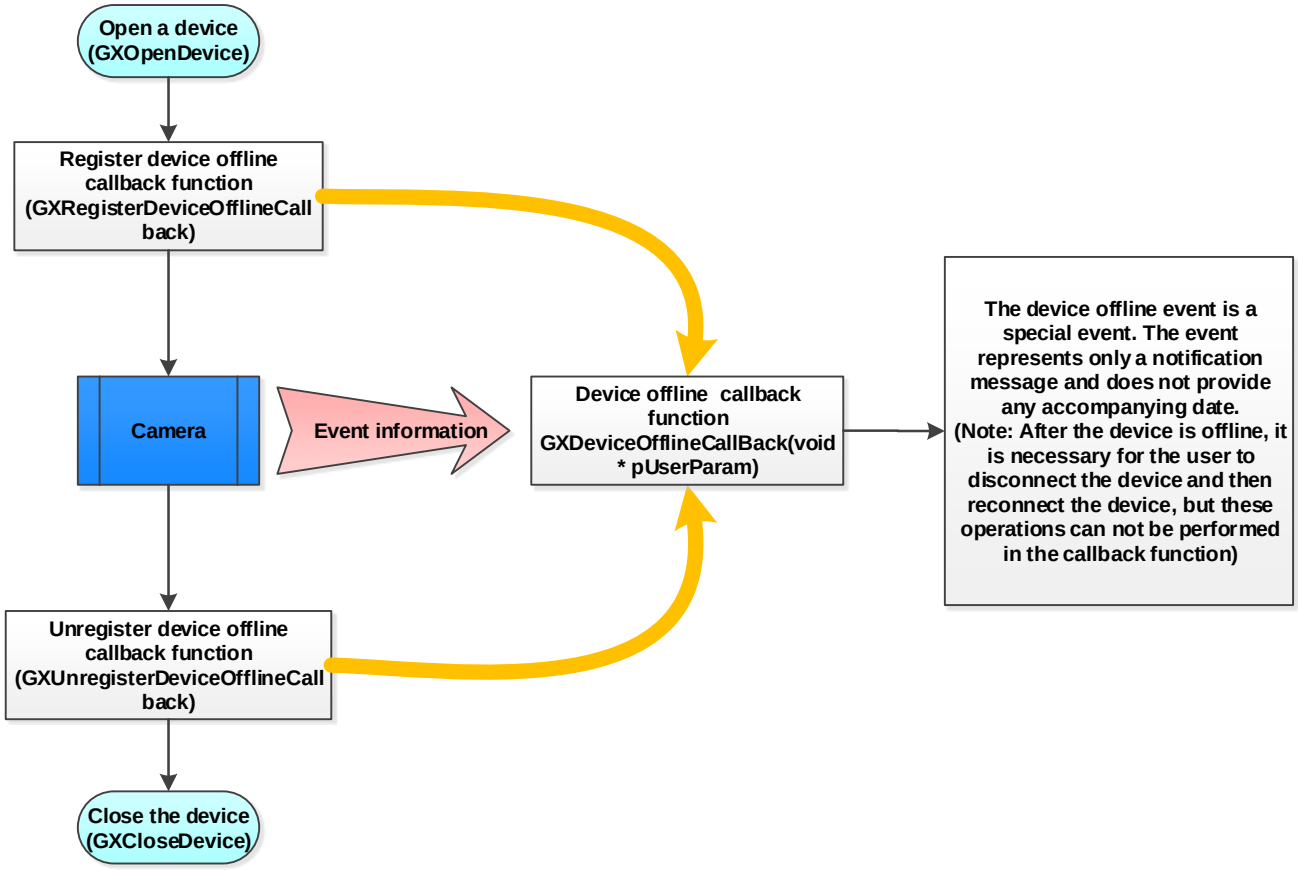

Figure 6: Offline events get flow

1.7. Remote device events get flow

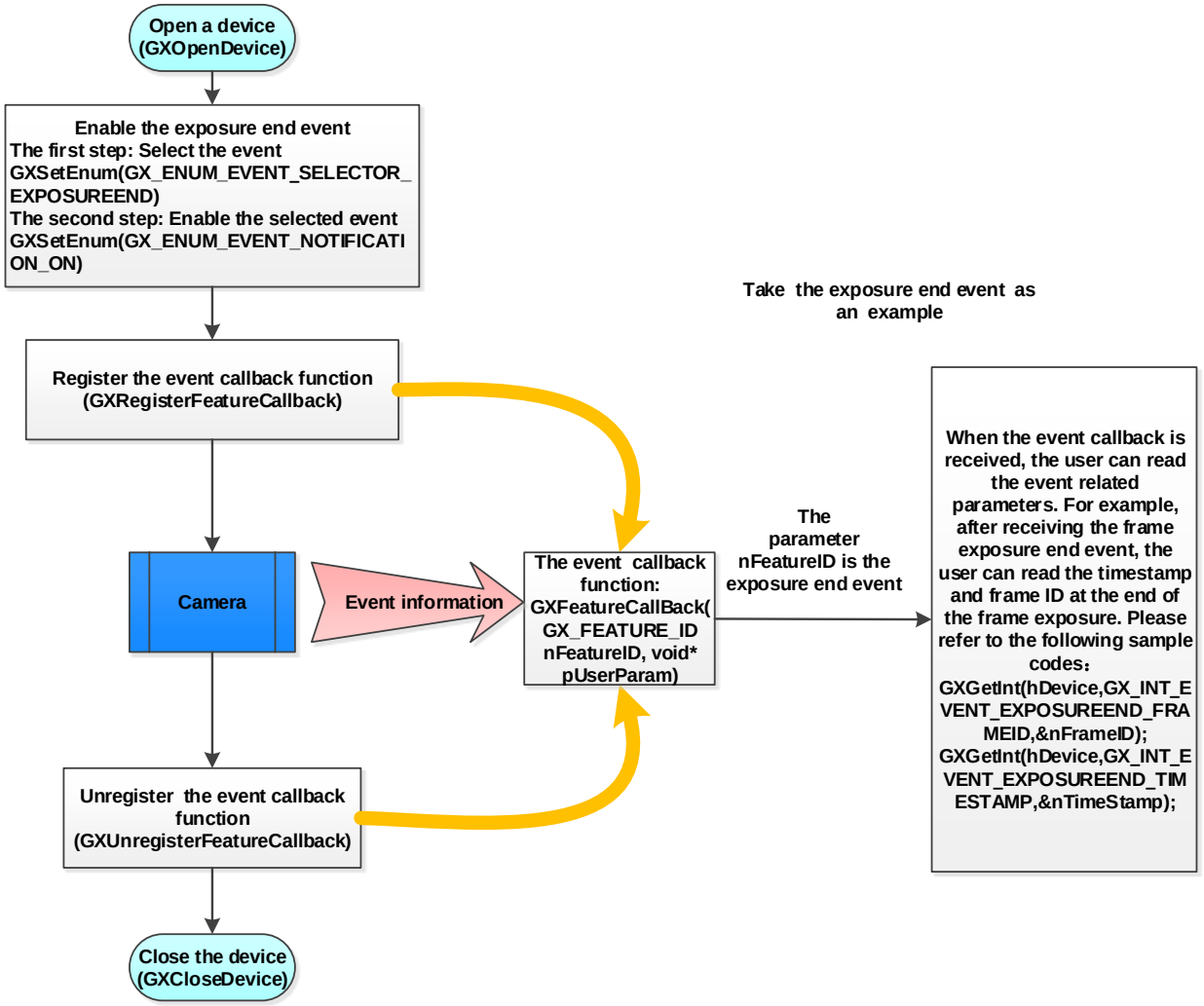

Figure 7: Remote device events get flow

2. Programming Guide

2.1. Build Programming Environment

2.1.1. Windows

2.1.1.1. C/C++

The GxIAPI and DxImageProc are the general C++ programming interface and image processing algorithm interface produced by the software department of Daheng Imaging. The interfaces are suit for all the Daheng's cameras that support the GenICam protocol. Under the installation directory of the installation packet, the SDK folder contains GxIAPI.h, DxImageProc.h and GxIAPI.lib, DxImageProc.lib files and some simple sample programs.

- GxIAPI.dll, DxImageProc.dll Dynamic link library files
- GxIAPI.lib, DxImageProc.lib Static link library files
- GxIAPI.h, DxImageProc.h Interfaces and macro declaration header files

When installing the install packet, the installation paths of the .dll files are added to the system environment variable "path", so your applications will automatically search the .dll file from the environment variable and load it.

When creating the project with the GxIAPI or DxImageProc, the .h header file and .lib static link library are needed, and then you can call and compile it successfully.

2.1.1.1.1. Configuration the VC6 programming environment

Click **Project->Settings** in the menu, pop-up the **project settings** window, click on the **C/C++** tab, set the **Category** as **Preprocessor**, fill in the .h files directory path address (based on the actual installation directory) in the **Additional include directories**, see Figure 8:

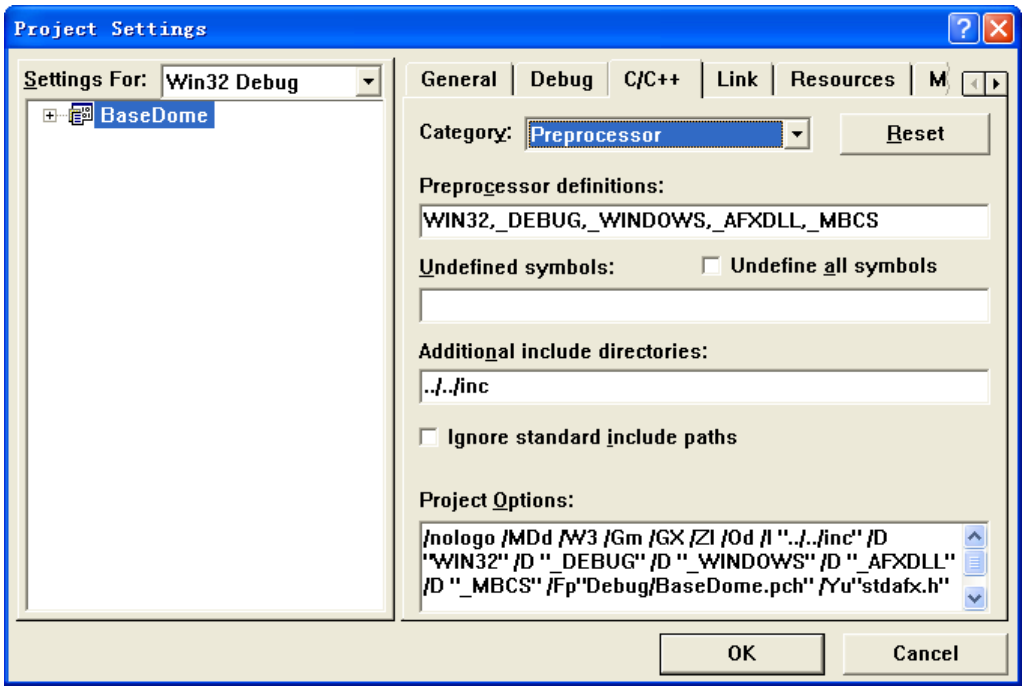

Figure 8

Select **Link** tab, set **Category** as **General**, fill **GxIAPI.lib** and **DxImageProc.lib** in the **Object/library modules**, and see Figure 9:

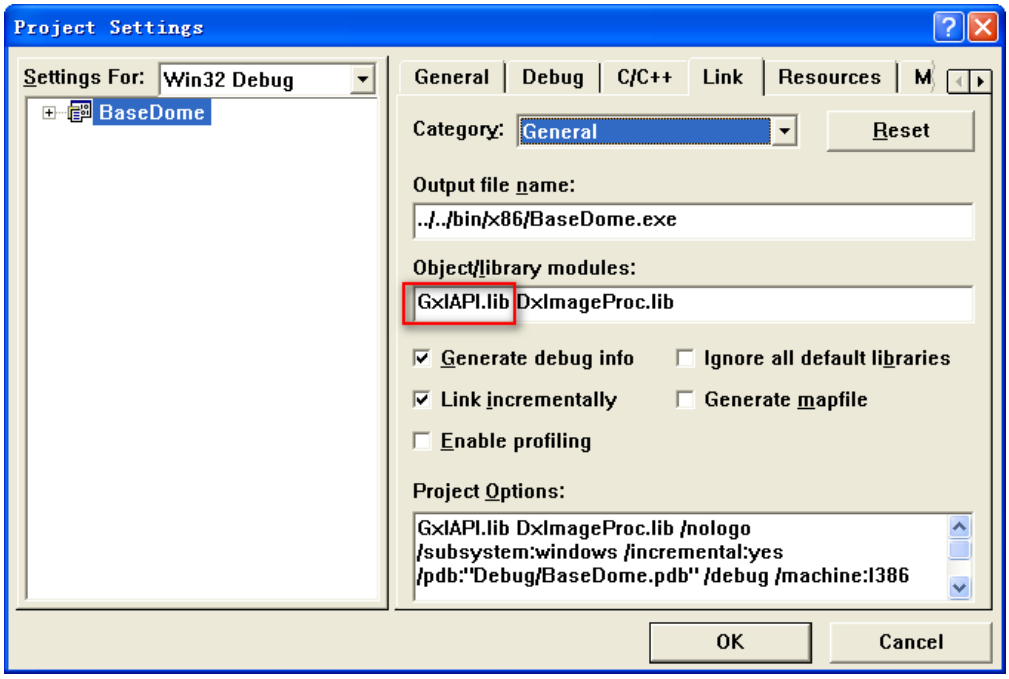

Figure 9

Still stay in **Link** tab, set **Category** as **Input**, and fill in the directory path address (based on the actual installation directory) of the **GxIAPI.lib** and **DxImageProc.lib** in the **Additional library path**, and see Figure 10:

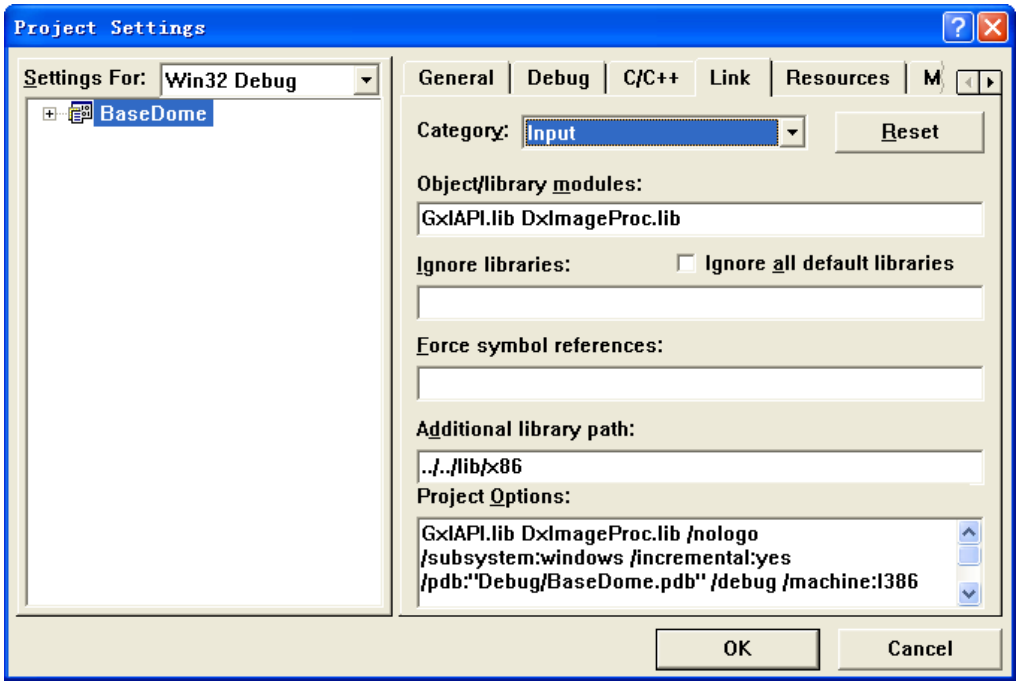

Figure 10

2.1.1.1.2. Configuration the VS2005 programming environment

Select the project which created by yourself in the solution resource management window, then click **project**->**properties** in the menu, pop-up the **Property page** window. Select **Configuration Properties->C/C++->General**, fill in the directory path address (based on the actual installation directory) of the **GxIAPI.h** and **DxImageProc.h** in the **Additional Include Directories**, and see Figure 11:

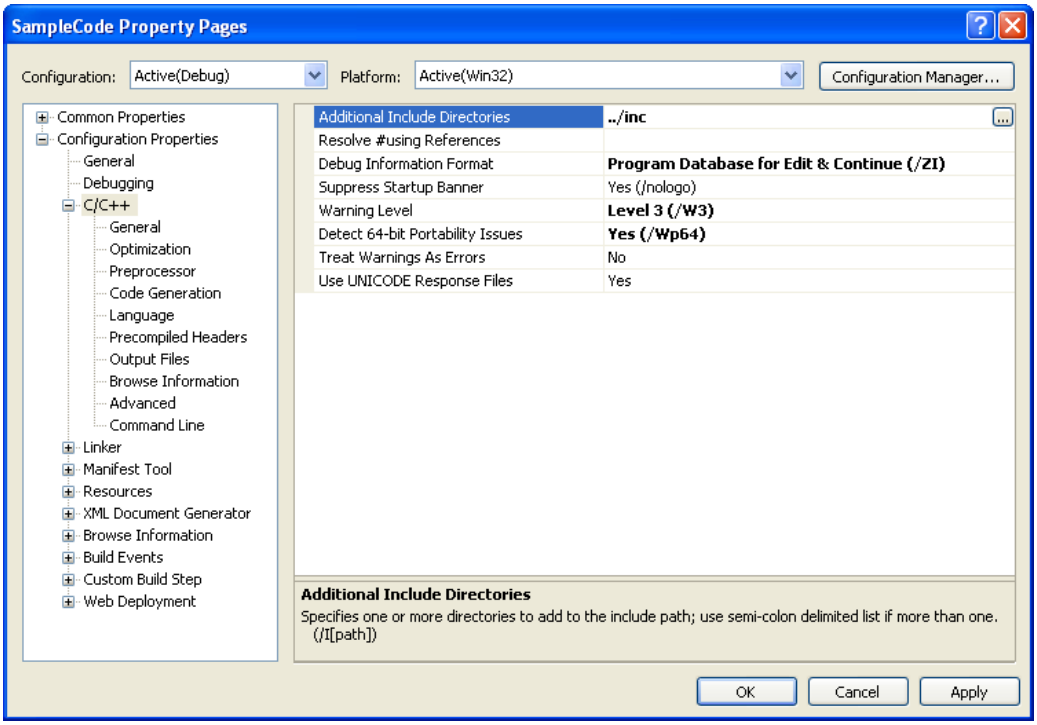

Figure 11

Select **Configuration Properties->Linker->General**, fill in the directory path address (based on the actual installation directory) of the **GxIAPI.lib** and **DxImageProc.lib** in the **Additional Library Directories**, and see Figure 12:

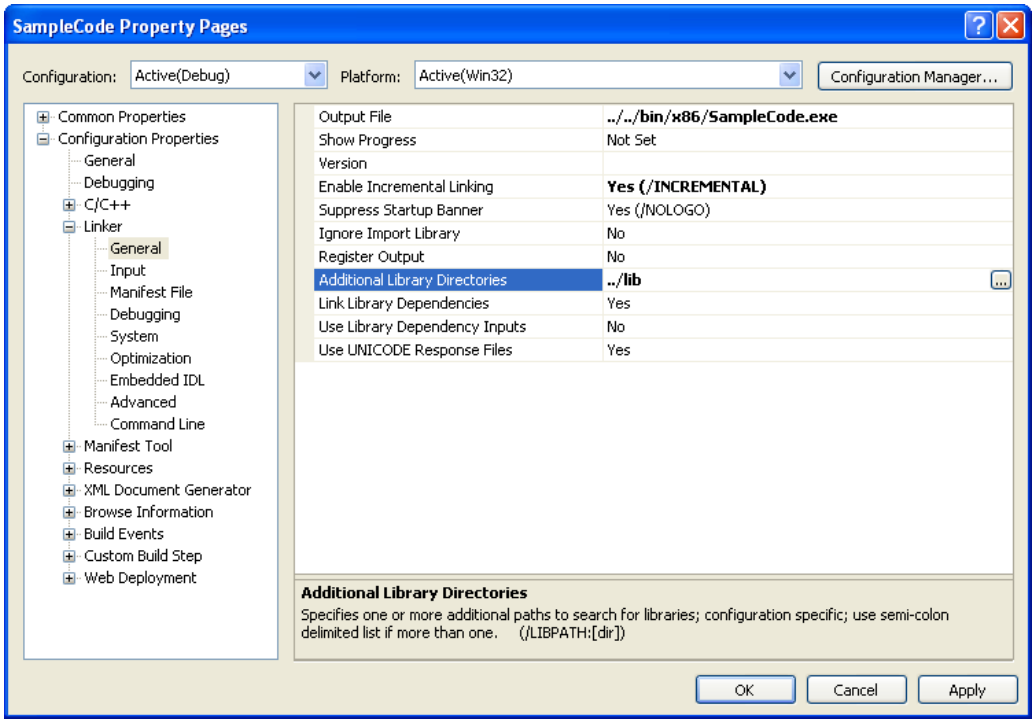

Figure 12

Select **Configuration Properties->Linker->Input**, fill in the **GxIAPI.lib** and **DxImageProc.lib** in the **Additional Dependencies**, and see Figure 13:

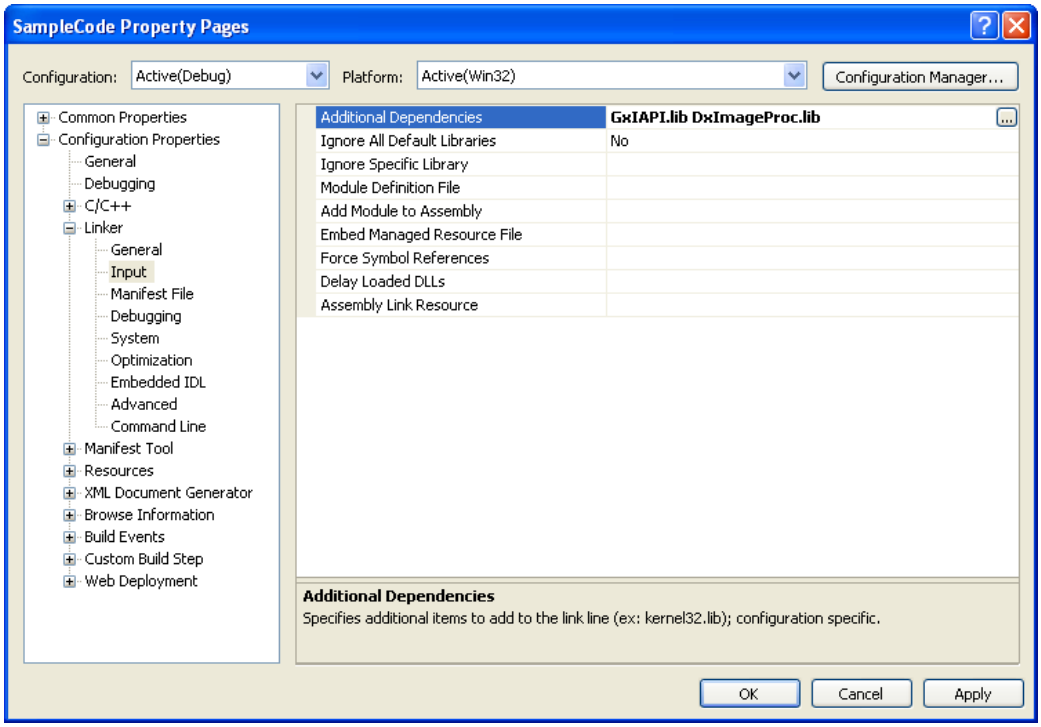

Figure 13

2.1.2. Linux

2.1.2.1. Console program

➢ **Makefile**

1) Before compiling the program, you need to copy the GxIAPI. h and DxImageProc. h header files from the include directory of the SDK library to the project directory, or add the include directory of the SDK library to the search path by the "- I" parameter;
2) When linking the application, you must link to the libgxiapi.so (-lgxiapi) ;

There are a series of sample programs in the Linux package, the following contents are the generic Makefile contents of the sample programs.

The red mark: samplename is the name of the compile and output program.

```
# Makefile for sample program
.PHONY      : all clean

# the program to build
NAME        := samplename

# Build tools and flags
CXX         := g++
LD          := g++
SRCS        := $(wildcard *.cpp)
OBJS        := $(patsubst %cpp, %o, $(SRCS))
CPPFLAGS    := -w -I./

LDFLAGS     := -lgxiapi -lpthread

all         : $(NAME)
```

```
$(NAME)     : $(OBJS)
$(LD) -o $@ $^ $(CPPFLAGS) $(LDFLAGS)

%.o     : %.cpp
$(CXX) $(CPPFLAGS) -c -o $@ $<

clean       :
$(RM) *.o $(NAME)
```

2.1.2.2. Qt-based sample program

The following describes the installation method and configuration steps of the qtcreator integrated development environment.

2.1.2.2.1. Install qtcreator

a) Installation
   sudo apt-get install qtcreator
b) Run
   Run qtcreator at the terminal

2.1.2.2.2. Configuring the environment under qtcreator (based on Qt Creator 3.5.1)

Step 1: After running the program, the display interface is as shown in Figure 14. Select **Open Project** and select the project file of the project to be opened in the pop-up dialog box.

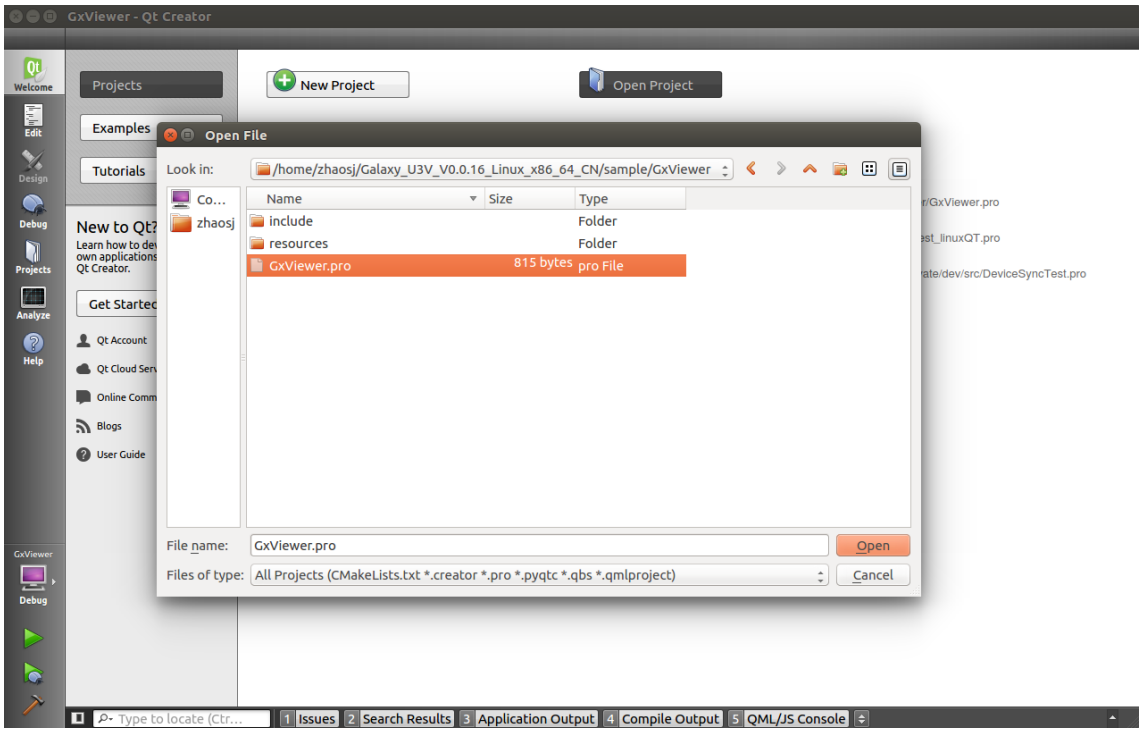

Figure 14

Step 2: When opening the project, if the project cannot be opened and the Kit related contents are prompted, you need to check the configuration of the qtcreator. Select **Tool -> Options -> Build & Run -> Qt Versions** to see if qmake is detected. If it is not detected automatically, you need to load it manually, as shown in Figure 15. This step can be ignored if the project can be opened normally.

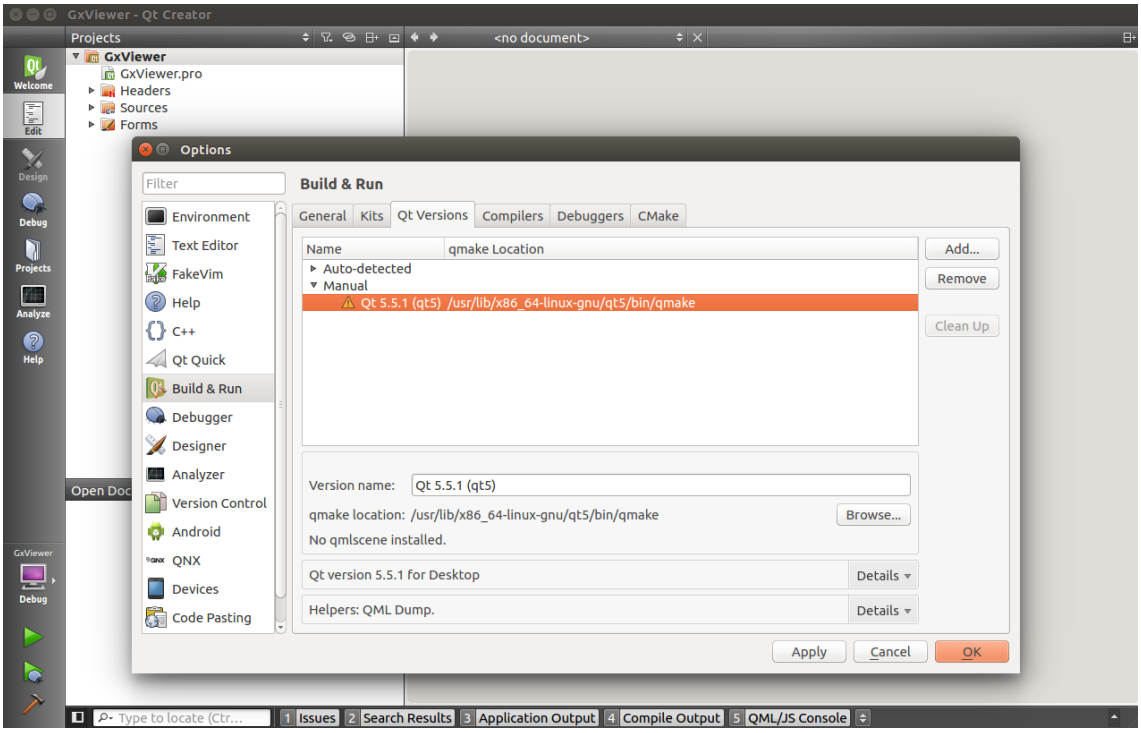

Figure 15

Step 3: After qmake loads successfully, add qmake to the Qt Version of the build options you want to use in the **Tool -> Options -> Build & Run -> Kits** tab, as shown in Figure 16. This step can be ignored if the project can be opened normally.

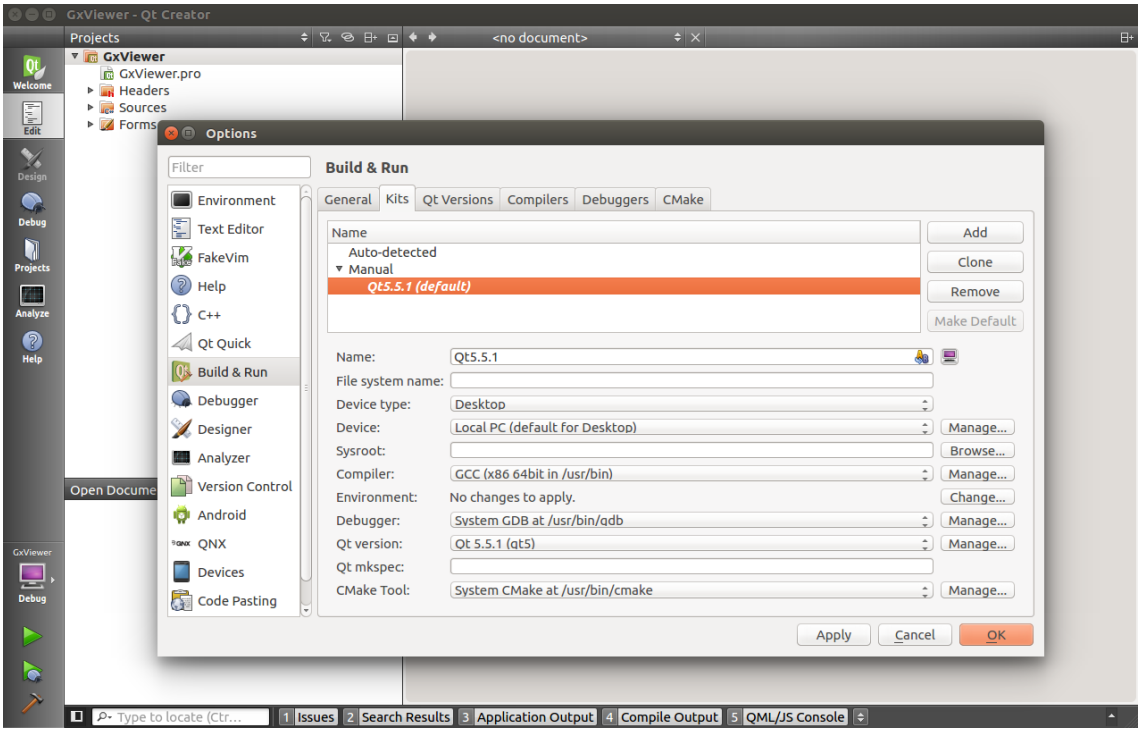

Figure 16

Step 4: After the project is successfully opened, compile and run.

2.2. Quick Guide

- Initialization and uninitialization GxIAPI runtime library
- Enumeration cameras and get the information
- Configurate the camera IP address
- Open and close the camera
- Camera control function
- DQBuf image acquisition (Linux only)
- Callback image acquisition

2.2.1. Initialization and uninitialization GxIAPI runtime library

Before using the GxIAPI interface (except *GXCloseLib*/ *GXGetLastError*), you must call the *GXInitLib* to initialize the library. Before exiting the application, you also need to call *GXCloseLib* to release resources in order to corresponding with *GXInitLib*. All the sample codes in this manual have been initialized and uninitialized GxIAPI library.

Code Sample:

```
#include "GxIAPI.h"

int main(int argc, char* argv[])
{
    GX_STATUS status = GX_STATUS_SUCCESS;

    // Calls GXInitLib() at the start location to initialize resources.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    //Uses GxIAPI.
    //...

    //Calls GXCLoseLib() at the end of the program to release resources.
    status = GXCloseLib();

    return 0;
}
```

2.2.2. Enumerate cameras and get information

You can call *GXUpdateDeviceList* interface to enumerate the current available device, return the number of the device, and then use *GxGetAllDeviceBaseInfo* to get the device's basic information without opening the camera.

Code Sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
uint32_t nDeviceNum = 0;
status = GXUpdateDeviceList(&nDeviceNum, 1000);
if (status == GX_STATUS_SUCCESS && nDeviceNum > 0)
{
   GX_DEVICE_BASE_INFO *pBaseinfo = new GX_DEVICE_BASE_INFO[nDeviceNum];
   uint32_t nSize = nDeviceNum * sizeof(GX_DEVICE_BASE_INFO);

   // Gets the basic information of all devices.
   status = GXGetAllDeviceBaseInfo(pBaseinfo, &nSize);
   delete []pBaseinfo;
}
```

If the device is a GigE camera, you can also use *GxGetDeviceIPInfo* to get the network information of the device.

Code Sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
uint32_t nDeviceNum = 0;
status = GXUpdateDeviceList(&nDeviceNum, 1000);
if (status == GX_STATUS_SUCCESS && nDeviceNum > 0)
{
   GX_DEVICE_IP_INFO stIPInfo;

   // Gets the network information for the first device.
   status = GXGetDeviceIPInfo(1, &stIPInfo);
}
```

2.2.3. Configurate the camera IP address

1) For GigE cameras, you can call *GXGigEIpConfiguration* to configurate the IP address for the camera, and the IP address set in this way is still valid after the camera has been rebooted. Calling this function requires the MAC address of the destination camera, which can be got by calling *GxGetDeviceIPInfo*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;

//This example is described in persistent IP configuration mode, and
//other IP configuration modes are similar.
//Sets to persistent IP configuration mode.
status = GXGigEIpConfiguration(szMAC, emIpConfigureMode,
                               szIpAddress, szSubnetMask,
                               szDefaultGateway, szUserID);
```

2) You can also call *GXGigEForceIp* to configurate the camera IP address, but the IP address set in this way may effective only for this use, when the camera rebooted, the camera may use the previous IP address. Calling this function requires the MAC address of the destination camera, which can be got by calling *GxGetDeviceIPInfo*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;

//Force Ip
status = GXGigEForceIp(szMAC, szIpAddress, szSubnetMask,
                       szDefaultGateway);
```

2.2.4. Open and close the camera

After calling the *GXUpdateDeviceList* interface to enumerate the device, if the return value of the device number is greater than 0, it means that there have devices can be used currently, you can call the *GxOpenDevice* interface to open them.

Code Sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
size_t nDeviceNum = 0;
GX_OPEN_PARAM stOpenParam;

status = GXUpdateDeviceList(&nDeviceNum, 1000);
if (status == GX_STATUS_SUCCESS && nDeviceNum > 0)
{
   GX_DEV_HANDLE hDevice = NULL;

   // Open the first device in the enumeration list.
   // Assuming that the user enumerates three available devices, the user can
   // set the pszContent field of the stOpenParam parameter to 1, 2, 3.
   stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
   stOpenParam.openMode   = GX_OPEN_INDEX;
   stOpenParam.pszContent = "1";

   // Opens the device via a serial number.
   //stOpenParam.openMode = GX_OPEN_SN;
   //stOpenParam.pszContent = "EA00010002";

   // Opens the device via an IP address.
   //stOpenParam.openMode = GX_OPEN_IP;
   //stOpenParam.pszContent = "192.168.40.35";

   // Opens the device via a MAC address.
   //stOpenParam.openMode = GX_OPEN_MAC;
   //stOpenParam.pszContent = "54-04-A6-C2-7C-2F";

   status = GXOpenDevice(&stOpenParam, &hDevice);


   // Operates the device: control and acquisition.
   //..

   // Closes the Device.
   status = GXCloseDevice(hDevice);

}
```

The *GxOpenDevice* can open the camera by specifying access mode (exclusive, control, etc.), opening mode (Via IP, SN, MAC, Index, etc.). See the explanation for the *GxOpenDevice* interface later.

2.2.5. Camera control function

Int:

Related Interfaces:

GXGetIntRange：Get the range, Minimum, Maximum, Steps.

GXGetInt：Get the value.

GXSetInt：Set the value.

Code Sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;

// Gets the integer range.
GX_INT_RANGE stIntRange;
status = GXGetIntRange(hDevice, GX_INT_WIDTH, &stIntRange);

// Gets the current value.
int64_t nValue = 0;
status = GXGetInt(hDevice, GX_INT_WIDTH, &nValue);

// Sets the current value to the minimum value.
nValue = stIntRange.nMin;
status = GXSetInt(hDevice, GX_INT_WIDTH, nValue);
```

Floating-point type related interfaces:

GXGetFloatRange：Get the query value's range, Minimum, Maximum, Steps, and Unit.

GXGetFloat：Get the value.

GXSetFloat：Set the value.

Enumerate type related interfaces:

GXGetEnumEntryNums:Get the number of enumerated items.

GXGetEnumDescription：Get enumerate description, the value and the description of enumerated items.

GXGetEnum：Get the value.

GXSetEnum：Set the value.

Boolean type related interfaces:

GXGetBool：Get the value.

GXSetBool：Set the value.

Character string related interfaces:

GXGetStringLength：Get the length of the current string, unit: byte.

GXGetStringMaxLength：Get the maximum length of the string, unit: byte.

GXGetString：Get the value.

GXSetString：Set the value.

Chunk data related interfaces:

GXGetBufferLength：Get the length of chunk data, use this length to apply memory, and then call the *GXGetBuffer* to get the data.

GXGetBuffer：Get the buffer.

GXSetBuffer：Set the buffer.

Command related interfaces:

GXSendCommand：Send command.

2.2.6. DQBuf image acquisition (Linux only)

After the *GXOpenDevice* interface is called to open the device, the *GXStreamOn* interface can be called in sequence to enable the stream acquisition and remote device acquisition. In this case, *GXDQBuf* and *GXQBuf* interfaces (*GXDQAllBufs*, *GXQAllBufs* are similarly used) can be used to continuously acquire images. After calling the *GXDQBuf* interface, you can get an image buffer and do your image processing, and calling the *GXQBuf* interface, put the buffer back into the capture queue.

Code Sample:

```
//------------------------------
// The GXDQBuf interface acquires one frame of image at a time. This sample
// code demonstrates how to use these interfaces to get a frame of image.
//------------------------------
#include "GxIAPI.h"

int main(int argc, char* argv[])
{
    GX_STATUS     status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    uint32_t      nDeviceNum = 0;

    // Initializes the library.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    // Updates the enumeration list for the devices.
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }

    // Opens the device.
    status = GXOpenDeviceByIndex(1, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Define the incoming parameters of GXDQBuf.
        PGX_FRAME_BUFFER pFrameBuffer;

        // Stream On.
        status = GXStreamOn(hDevice);
        if (status == GX_STATUS_SUCCESS)
        {
            // Calls GXDQBuf to get a frame of image.
            status = GXDQBuf(hDevice, &pFrameBuffer, 1000);
            if (status == GX_STATUS_SUCCESS)
            {
                if (pFrameBuffer->nStatus == GX_FRAME_STATUS_SUCCESS)
                {
                    // Image acquisition succeeded.
                    // Image processing...
```

```
                }

                // Calls GXQBuf to put the image buffer back into the library
                //and continue acquiring.
                status = GXQBuf(hDevice, pFrameBuffer);
            }
        }

        // Sends a stop acquisition command.
        status = GXStreamOff(hDevice);
    }
    status = GXCloseDevice(hDevice);
    status = GXCloseLib();

    return 0;
}
```

2.2.7. CallBack image acquisition

Terms：

- Image processing callback function: the user-defined image processing function, the GxIAPI specifies the return value of the function, formal parameter and etc. (see the GxIAPI library specification->type->*Callback Function Type*->GXCaptureCallBack).
- Registered callback function: The user calls the *GxRegisterCaptureCallback* interface to pass the pointer of the user-defined image processing callback function to GxIAPI library (see *GxRegisterCaptureCallback* interface for details).
- Unregistered callback function: The user calls the *GxUnregisterCaptureCallback* interface to notify the GxIAPI library to release the callback function pointer which registered by the user.

Code Sample:

```
#include "GxIAPI.h"

// Image processing callback function.
static void GX_STDC OnFrameCallbackFun(GX_FRAME_CALLBACK_PARAM* pFrame)
{
    if (pFrame->status == GX_FRAME_STATUS_SUCCESS)
    {
        // Do some image processing operations.
    }
    return;
}

int main(int argc, char* argv[])
{
    GX_STATUS     status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    uint32_t      nDeviceNum = 0;

    // Initializes the library.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    // Updates the enumeration list for the devices.
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }

    // Opens the device.
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode   = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
```

```
        // Registers image processing callback function.

        status = GXRegisterCaptureCallback(hDevice, NULL,
                                           OnFrameCallbackFun);

        // Stream On.
        status = GXStreamOn(hDevice);

        //----------------------
        //
        // In this interval, the image will be returned to the user via the
        // OnFrameCallbackFun interface.
        //
        //----------------------

        // Sends a stop acquisition command.
        status = GXStreamOff(hDevice);

        // Unregisters image processing callback function.
        status = GXUnregisterCaptureCallback(hDevice);
    }
    status = GXCloseDevice(hDevice);
    status = GXCloseLib();

    return 0;
}
```

You can also call the *GxGetImage* interface to get images, but this method cannot be used with the callback mode simultaneously (see the *GxGetImage* interface for details).

2.3. Use the Gigabit Camera to Debug the Programs

Windows users are likely to encounter a situation that the device offline caused by a heartbeat timeout when using the Visual Studio development platform to debug a gigabit camera in debug mode. The application must be at a fixed time interval to send the heartbeat package to the device. If the device does not receive the heartbeat packages, you can think that the current connection has been disconnected, thus no longer accepting any commands sent from the application.

When the user runs the application program successfully, the underlying library will send heartbeat package normally, to maintain the connection state with the device, but if the user sets breakpoints in the application program, when the program runs to the breakpoint, the debugger will suspend all threads, including the thread which sends heartbeat packages, so when you debugging the program code in the Debug mode, the thread will not send heartbeat packages to the device.

You can increase the heartbeat timeout time to resolve this problem. Under debugging mode, the gigabit transfer layer software will set the timeout time as 5 minutes automatically when the device is opened. If the environment variable **GIGE_HEARTBEAT_TIMEOUT** is added in the system, and the value of it is greater than 0, then the heartbeat timeout value of the device will automatically be set to the value of the environment variable.

You can modify the heartbeat timeout by two ways. The first way is to add the following codes to the

program when the device is opened.

```
//The hDevice is the device handle. The device has been opened via the
//GXOpenDevice interface.

//Sets the heartbeat time to 5 minutes.
GXSetInt(hDevice, GX_INT_GEV_HEARTBEAT_TIMEOUT, 300000);
```

The second way is to add environment variables **GIGE_HEARTBEAT_TIMEOUT** to the system and set the value greater than 0, then use the application program to open the device and the heartbeat timeout value will automatically become the value of environment variable. Note that you only need to add this environment variable to the developed system, both the debug and the release application will work.

Note: If you set the heartbeat timeout very long, when the program is ended, it still not calls the GXCloseDevice interface to close the device, or not call the GXCloseLib interface to release resources, this will cause the device cannot be reset during a heartbeat time, which leads to fail when you try to open the device again. The solution to this problem is to wait for the device to reset after a heartbeat timeout or to reactivate the device.

2.4. Precautions When Running the Programs

When you use the SDK to develop a new program and generate the new executable program, you must run it with the administrator authority to open the device or you will fail to open the device.

Here are two ways to run the administrator authority, only for reference:

1) Right-click the executable program, and select run as an administrator;
2) Take VS2008 platform for example, set the default permissions of the program as administrator when at the compiling stage, as shown in Figure 17: set **UAC Execution Level** to **requireAdministrator**.

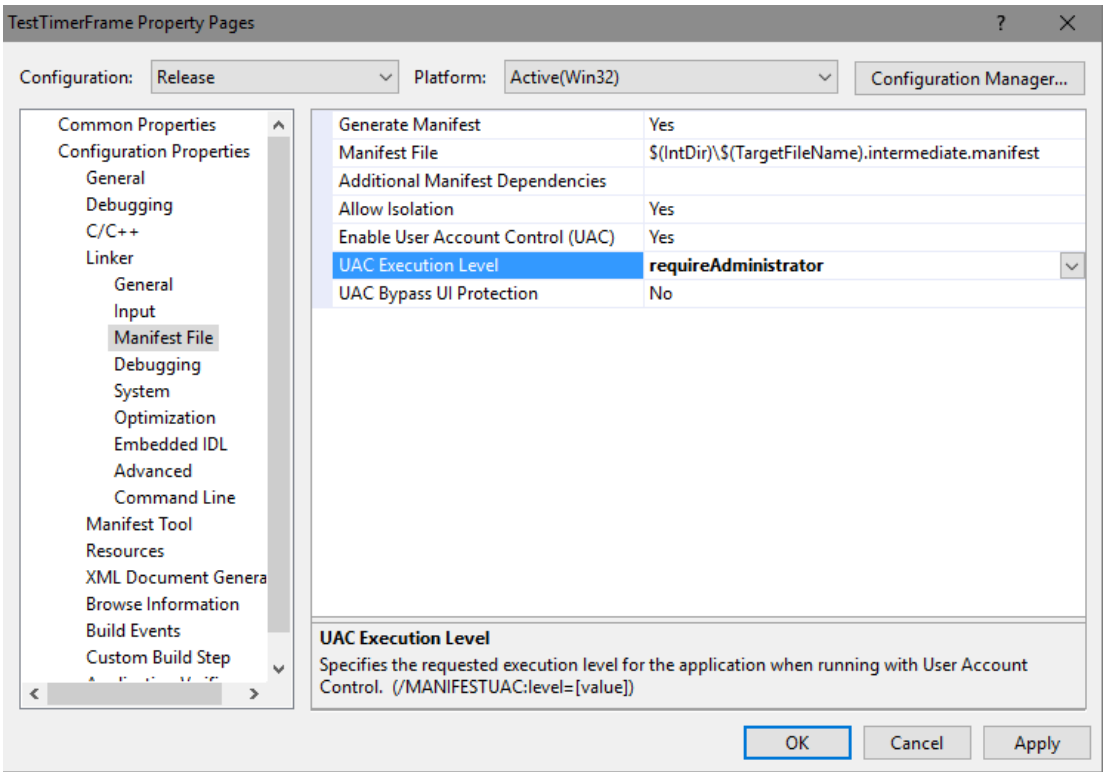

Figure 17

3. Camera Function Attribute Specification

3.1. Device Control

3.1.1. Device Information

3.1.1.1. Read-only Information

When you open the device, you can use the device handle to get the device information.

- Related Parameters

`GX_STRING_DEVICE_VENDOR_NAME`: The name of the device's vendor.

`GX_STRING_DEVICE_MODEL_NAME`: The model name of the device.

`GX_STRING_DEVICE_FIRMWARE_VERSION`: The version of the device's firmware.

`GX_STRING_DEVICE_VERSION`: The version of the device.

`GX_STRING_DEVICE_SERIAL_NUMBER`: The series number of the device.

`GX_STRING_FACTORY_SETTING_VERSION`: The version of device's factory setting.

- Code Sample

```
// To get the name of the device's vendor, for example, the first step is to get
// the length of the string, the second step is to allocate the buffer according
// to the length got, and the third step is to get the string content.

GX_STATUS status = GX_STATUS_SUCCESS;
size_t nSize = 0;

// Gets the maximum length allowed by the string (this length contains the
// terminator '\0').
status = GXGetStringMaxLength(hDevice, GX_STRING_DEVICE_VENDOR_NAME,
                              &nSize);

// Applies for memory based on the length got.
char *pszText = new char[nSize];
status = GXGetString(hDevice, GX_STRING_DEVICE_VENDOR_NAME, pszText,
                     &nSize);
```

3.1.1.2. Readable and Writable Information

- Related Parameters

  `GX_STRING_DEVICE_USERID`：The user-defined name of the device.

- Code Sample

```
// Takes the get of the user ID as an example. The first step is to get the length
// of the string. The second step is to allocate the buffer according to the length
// got. The third step is to get/set the string content.

GX_STATUS status = GX_STATUS_SUCCESS;
size_t nSize = 0;

// Gets the maximum length allowed by the string (this length contains the
// terminator '\0').
status = GXGetStringMaxLength(hDevice, GX_STRING_DEVICE_USERID, &nSize);

// Applies for memory based on the length got.
char *pszText = new char[nSize];
status = GXGetString(hDevice, GX_STRING_DEVICE_USERID, pszText, &nSize);

// Sets the user-defined name of the device.
status = GXSetString(hDevice, GX_STRING_DEVICE_USERID, "TestUserID");
```

3.1.2. Device Control

- Terms

Device reset: Restores the device to the initial state and the device is powered on again. After the call is completed, the host will lose its connection to the current device. And because the interface can be set only when the device is open, after the call is completed, the developer needs to actively call the GXCloseDevice interface to close the device to release the corresponding memory resources.

Timestamp tick frequency: Read-only information, which represents the frequency of timestamp counter, and its value is 125000000Hz.

Timestamp latch: Latchs the current timestamp value, that is, the time value taken from the start of the device power-on to the call of the timestamp latch command. The time value needs to be read through the "timestamp latch value".

Reset timestamp: Resets the timestamp counter and recount from 0.

Reset timestamp latch: First latchs the current timestamp value, then resets the timestamp counter and recounts from 0.

- Related Parameters

`GX_COMMAND_DEVICE_RESET:`	Device reset
`GX_INT_TIMESTAMP_TICK_FREQUENCY:`	Timestamp tick frequency
`GX_COMMAND_TIMESTAMP_LATCH:`	Timestamp latch
`GX_COMMAND_TIMESTAMP_RESET:`	Reset timestamp
`GX_COMMAND_TIMESTAMP_LATCH_RESET:`	Reset timestamp latch
`GX_INT_TIMESTAMP_LATCH_VALUE:`	Timestamp latch value

- Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;

//Send reset timestamp command
emStatus = GXSendCommand(hDevice, GX_COMMAND_TIMESTAMP_RESET);

//Send device reset command
emStatus = GXSendCommand(hDevice, GX_COMMAND_DEVICE_RESET);
```

3.2. Image Format

3.2.1. ROI

- Terms

ROI：Region of interest, a configurable rectangle selected area related to the sensor of the camera, the camera just output the data of selected area, and the data beyond the area will be ignored.

- Related Parameters

`GX_INT_SENSOR_WIDTH`：The width of the sensor, unit: pixel.

`GX_INT_SENSOR_HEIGHT`：The height of the sensor, unit: pixel.

`GX_INT_WIDTH_MAX`：The maximum width of the Image, unit: pixel.

`GX_INT_HEIGHT_MAX`：The maximum height of the Image, unit: pixel.

`GX_INT_WIDTH`：The width of ROI, unit: pixel.

`GX_INT_HEIGHT`：The height of ROI, unit: pixel.

`GX_INT_OFFSET_X`：Relative to the x direction offset in the upper left corner of the sensor, unit: pixel.

`GX_INT_OFFSET_Y`：Relative to the y direction offset in the upper left corner of the sensor, unit: pixel.

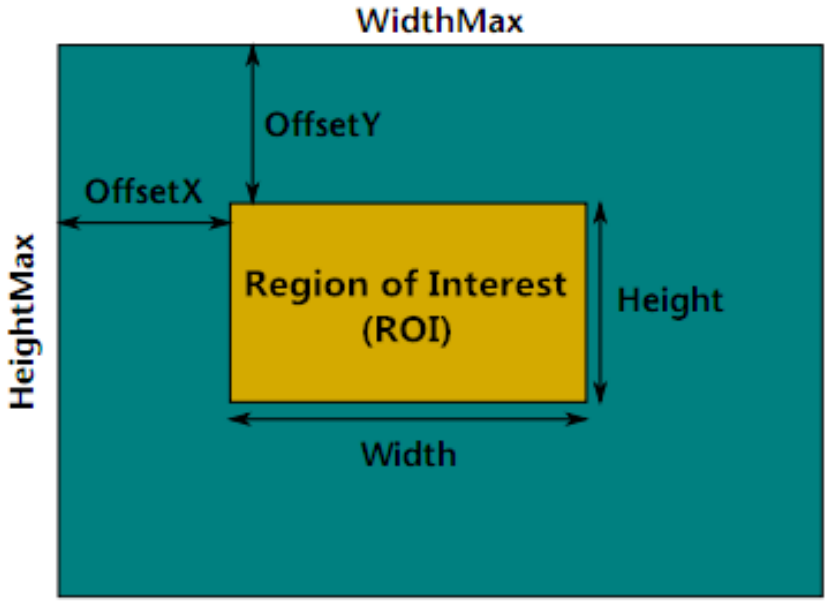

Figure 18: The related parameters of ROI

- Effect images

Figure 19: The original image

Figure 20: The image after ROI

- Code Sample

```
// Sets an area with an offset of (0,0) and a size of 600x400.
GX_STATUS status = GX_STATUS_SUCCESS;
int64_t nWidth   = 600;
int64_t nHeight  = 400;
int64_t nOffsetX = 0;
int64_t nOffsetY = 0;

status = GXSetInt(hDevice, GX_INT_WIDTH, nWidth);
status = GXSetInt(hDevice, GX_INT_HEIGHT, nHeight);
status = GXSetInt(hDevice, GX_INT_OFFSET_X, nOffsetX);
status = GXSetInt(hDevice, GX_INT_OFFSET_Y, nOffsetY);
```

- Precautions

1) There are two pairs of attributes that affect the maximum ROI size:

GX_INT_SENSOR_WIDTH
GX_INT_SENSOR_HEIGHT

The width and height of the sensor determine the effective resolution of the image sensor, and also determine the total of the pixels available, and the values are fixed. In the default mode (no binning, no decimation, no ROI), the image size is equal to the sensor_width * sensor_height.

GX_INT_WIDTH_MAX

GX_INT_HEIGHT_MAX

The maximum width and the maximum height determine the maximum size of the current ROI available, and the maximum ROI width can be affected by binning or decimation. In the default mode (no binning, no decimation), width_max * height_max = sensor_ width * sensor_ height.

2) In order to ensure that the ROI is valid, the four attributes of ROI need to follow the relational formula, as follows:

OffsetX + Width <= WidthMax (the maximum width of the current image)
OffsetY + Height <= HeightMax (the maximum height of the current image)

Two formulas above illustrate the maximum of four attributes are dynamic changes, for example, if you adjust the value of the width, then the adjustable maximum value of the OffsetX will be affected, the correlation has been achieved in the GxIAPI.

3.2.2. Image Resolution

- Terms

Binning and Decimation directly affect the camera sensor, before generating the image, the sub-sample processing operation has completed. These two functions can improve the frame rate by modifying the resolution of the horizontal or the vertical direction. Compared to ROI, the ROI function is to cut the field of view, but the Binning and Decimation are handled with the whole image, do not affect the field of view.

1) Binning: It is an image output mode. Two types of Binning are available: horizontal Binning and vertical Binning. In this mode, the charges of adjacent pixels are added according to the Sum mode selected by the user, or the average of adjacent pixel charges is taken. The value is output in one-pixel mode. The advantage of using Binning is that several pixels can be combined for use as one pixel, which increases the camera's response to light, output speed, and reduce resolution. When Binning is used for both rows and columns, the aspect ratio of the image does not change. When using 2:2 Binning, the resolution of the image will be reduced by 75%.
2) Decimation：The pixel skip output, picking the N^{th} (horizontal or vertical) pixel for output, and the other pixels are ignored.
3) Horizontal pixel Binning mode: This mode has two modes: Sum and Average. In Sum mode, adjacent charges are added together and then output in one-pixel mode. When the pixel value is greater than the maximum value, the maximum value is taken. In Average mode: the adjacent charges are added together and averaged.
4) Vertical pixel Binning mode: Similar to horizontal pixel Binning mode.

- Related Parameters

`GX_INT_BINNING_HORIZONTAL`：The parameter of horizontal Binning.

`GX_INT_BINNING_VERTICAL`：The parameter of vertical Binning.

`GX_INT_DECIMATION_HORIZONTAL`：The parameter of horizontal decimation.

`GX_INT_DECIMATION_VERTICAL`：The parameter of vertical decimation.

`GX_ENUM_BINNING_HORIZONTAL_MODE`：Horizontal pixel Binning mode, the enumeration value ref: `GX_BINNING_HORIZONTAL_MODE_ENTRY`.

`GX_ENUM_BINNING_VERTICAL_MODE`：Vertical pixel Binning mode, the enumeration value ref: `GX_BINNING_VERTICAL_MODE_ENTRY`.

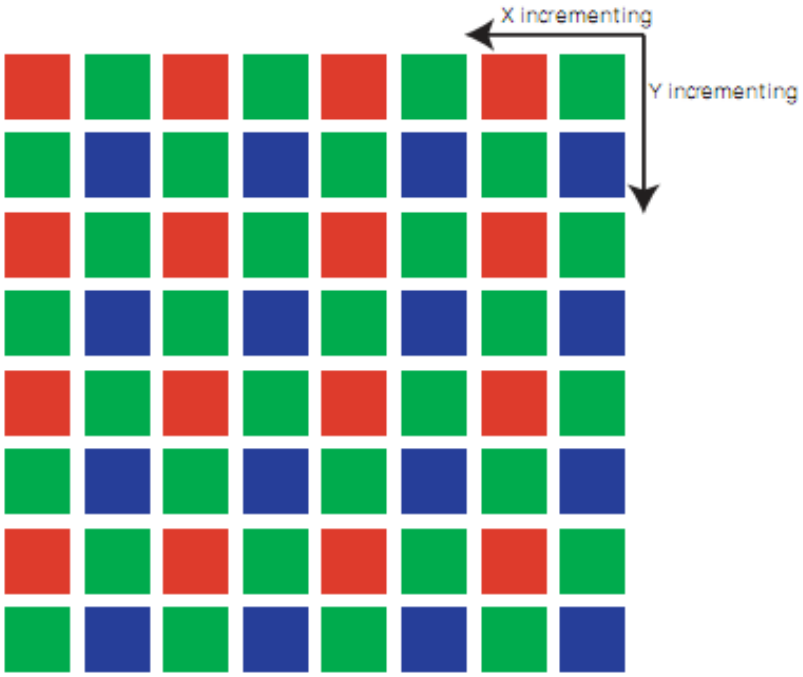

Figure 21:Original image

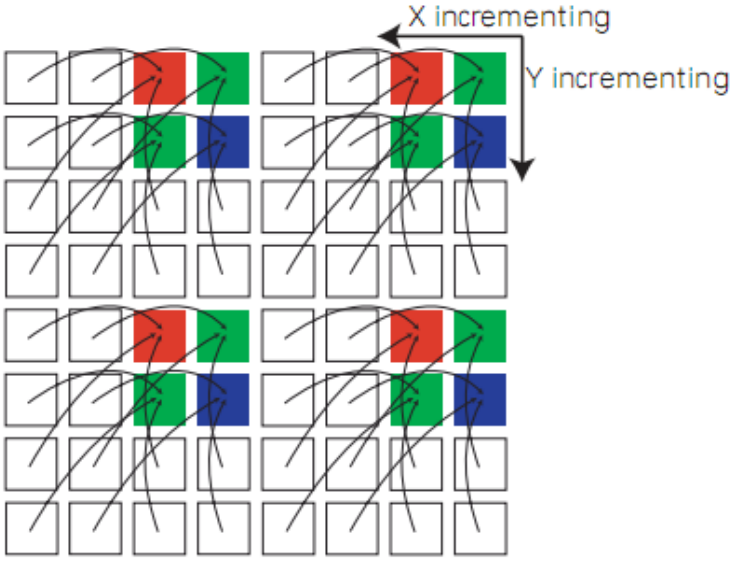

Figure 22: Binning (2x2) processing

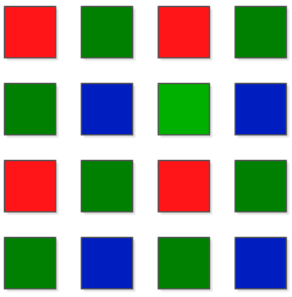

Figure 23:The Binning processing result

Figure 24: Decimation (2x2) processing

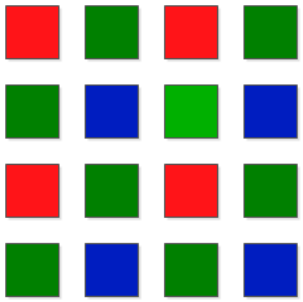

Figure 25: The decimation processing result

- Code Sample

```
//Configures a 2x2 Binning and 2x2 Decimation.
GX_STATUS status = GX_STATUS_SUCCESS;
int64_t nBinningH = 2;
int64_t nBinningV = 2;
int64_t nDecimationH= 2;
int64_t nDecimationV= 2;

//Set horizontal and vertical Binning mode to Sum mode.
status=GXSetEnum(hDevice,GX_ENUM_BINNING_HORIZONTAL_MODE,
                 GX_BINNING_HORIZONTAL_MODE_SUM);
status= GXSetEnum(hDevice,GX_ENUM_BINNING_VERTICAL_MODE,
                 GX_BINNING_VERTICAL_MODE_SUM);
status = GXSetInt(hDevice, GX_INT_BINNING_HORIZONTAL, nBinningH);
status = GXSetInt(hDevice, GX_INT_BINNING_VERTICAL, nBinningV);
status = GXSetInt(hDevice, GX_INT_DECIMATION_HORIZONTAL, nDecimationH);
status = GXSetInt(hDevice, GX_INT_DECIMATION_VERTICAL, nDecimationV);
```

- Precautions

3.2.3. Data Format

- Terms

1) Data bit depth: The data bit depth represents that the data bits of each pixel gray value occupied. For example, 8 bits of data represent a gray value range of 0 to 255. The RAW data format converts the captured optical signal into a digital signal by a CMOS or CCD image sensor without any compression. RAW8 indicates that the output image data bit is 8 bits, and RAW12 indicates that the output image data bit is 12 bits.
2) Bayer color conversion: The Bayer type is the formatting of RAW image data, see Figure 26, when the image data is processed or displayed, it needs to convert it to a 24 bits RGB real color image data. A simple interpolation algorithm is as follows:

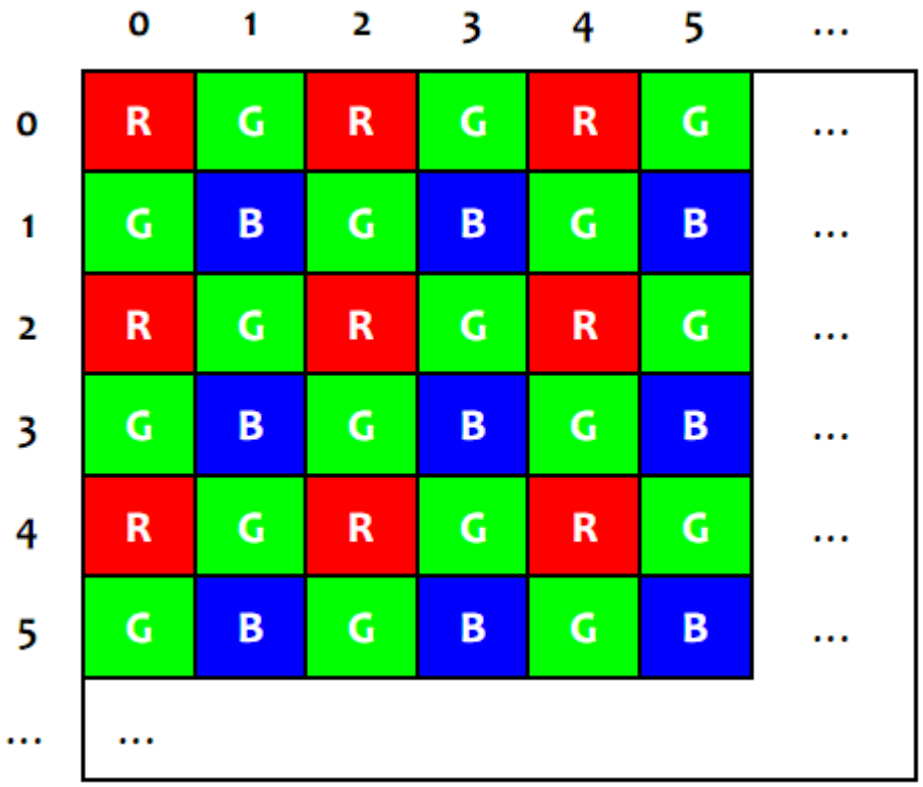

Figure 26: Bayer format

One simple transformation is as follows:

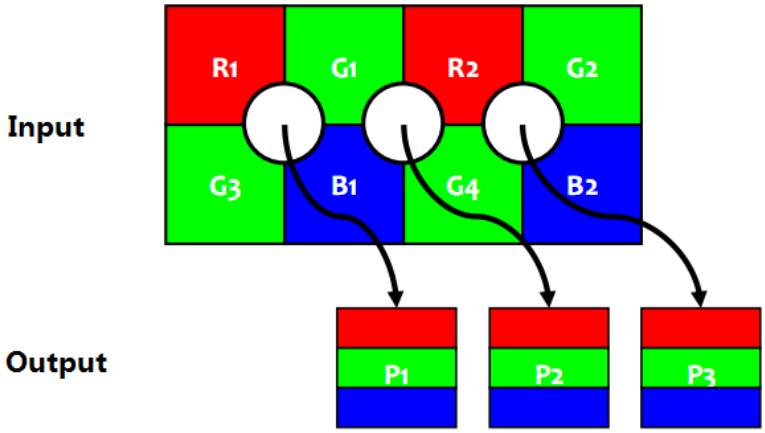

Figure 27: Bayer conversion

Pixel 1 (P1)

$$P1_{Red} = R1$$

$$P1_{Green} = \frac{G1 + G3}{2}$$

$$P1_{Blue} = B1$$

Pixel 2 (P2)

$$P2_{Red} = R2$$

$$P2_{Green} = \frac{G1 + G4}{2}$$

$$P2_{Blue} = B1$$

Pixel 3 (P3)

$$P3_{Red} = R2$$

$$P3_{Green} = \frac{G2 + G4}{2}$$

$$P3_{Blue} = B2$$

Figure 28

- Related Parameters

  GX_ENUM_PIXEL_SIZE: Data bit depth, the enumeration value ref: GX_PIXEL_SIZE_ENTRY.

  GX_ENUM_PIXEL_COLOR_FILTER: Bayer format, the enumeration value ref: GX_PIXEL_COLOR_FILTER_ENTRY.

  GX_ENUM_PIXEL_FORMAT：Data format, the enumeration value ref: GX_PIXEL_FORMAT_ENTRY.

Data Format: The PixelFormat is 8Byte, 32bit. It gathers various of information together to form a single amount of information. The highest two bytes represents the color type (color or monochrome), the followed two bytes represent the data bit depth (8bit, 10bit, etc.), and the lowest four bytes represent the coding sequence number. (Ref the GX_PIXEL_FORMAT_ENTRY).

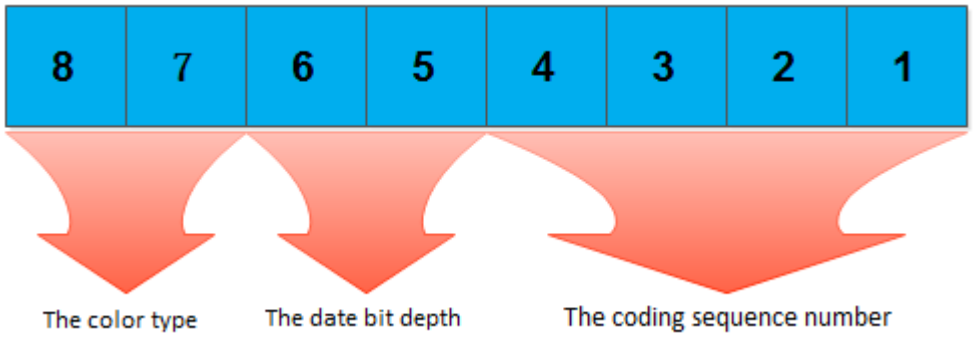

Figure 29: Data format

- Code Sample

```
// Uses the GXGetEnumEntryNums and GXGetEnumDescription interfaces to
// query the GX_ENUM_PIXEL_FORMAT types supported by the current camera.
// Please refer to the interface descriptions, which are omitted here.
    GX_STATUS status = GX_STATUS_SUCCESS;

        // Reads the current pixelformat.
        int64_t nPixelFormat = 0;
        status = GXGetEnum(hDevice, GX_ENUM_PIXEL_FORMAT, &nPixelFormat);

        // Sets the pixelformat to the bayer format of the BG type.
        nPixelFormat = GX_PIXEL_FORMAT_BAYER_BG10;
        status = GXSetEnum(hDevice, GX_ENUM_PIXEL_FORMAT, nPixelFormat);

        // Reads the current pixelsize.
        int64_t nPixelSize = 0;
        status = GXGetEnum(hDevice, GX_ENUM_PIXEL_SIZE, &nPixelSize);


        // Reads the current colorfilter.
        int64_t nColorFilter = 0;
        status = GXGetEnum(hDevice, GX_ENUM_PIXEL_COLOR_FILTER,
                           &nColorFilter);
```

3.2.4. Test Images

- Terms

Test Images: The Gige Vision series cameras support three test images: gray gradient test image, moving vertical stripe test image, and moving diagonal gray gradient test image; The USB3 Vision series cameras support three test images: gray gradient test image, moving diagonal gray gradient test image, and static diagonal gray gradient test image; The Pallas series cameras support three test images: gray gradient test image, moving diagonal gray gradient test image, and static diagonal gray gradient test image.

- Related Parameters

`GX_ENUM_TEST_PATTERN_GENERATOR_SELECTOR`：The test image source selection. The source used to select the test image. Currently only one source is supported, and no setting is required by default.

`GX_ENUM_TEST_PATTERN`： Test Images.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

// Queries what type of test images the current camera supports. See
// the usage of the GXGetEnumDescription interface.
// This example assumes that the current camera supports all types of test
// images.

// Sets to scroll vertical stripes test image. See GX_TEST_PATTERN_ENTRY for
// selecting test image types.
status = GXSetEnum(hDevice, GX_ENUM_TEST_PATTERN,
                   GX_ENUM_TEST_PATTERN_VERTICAL_LINE_MOVING);

// Closes the test image function.
status = GXSetEnum(hDevice, GX_ENUM_TEST_PATTERN, GX_ENUM_TEST_PATTERN_OFF);
```

3.2.5. Frame Information Control

- Terms

When the frame information is activated, the frame information is appended to the end of the image data in the following format, which identifies the information of the current image, such as the frame number.

Before the frame information is activated:

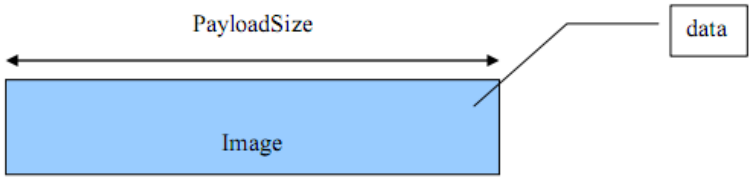

Figure 30: The image data before activating the frame information

After the frame information is activated, take MER-U3x camera as an example:

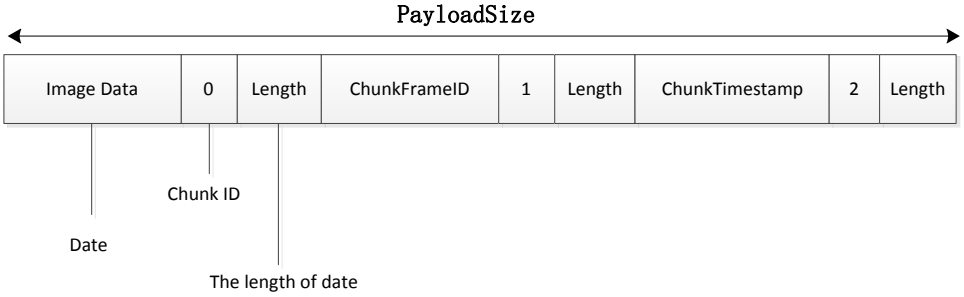

Figure 31: The image data after activating the frame information

- Related Parameters

`GX_BOOL_CHUNKMODE_ACTIVE`：Frame information enable.

`GX_ENUM_CHUNK_SELECTOR`：Frame information selection.

`GX_BOOL_CHUNK_ENABLE`：Single frame information enable.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

// Sets the frame information mode to the enablestate.
status = GXSetBool(hDevice, GX_BOOL_CHUNKMODE_ACTIVE, true);

// Queries which frame information types are supported by the current camera,
// see the usage of the GXGetEnumDescription interface.
// This example assumes that the current camera supports all types of frame
// information.

// Selects the frame number.
status = GXSetEnum(hDevice, GX_ENUM_CHUNK_SELECTOR, GX_CHUNK_SELECTOR_CHUNK_FRAME_ID);

// Sets the frame number to enable state.
status = GXSetBool(hDevice, GX_BOOL_CHUNK_ENABLE, true);
```

- Precautions

The C software interface opens the control interface of the frame information, but does not open an interface for acquiring frame information. If you need to analysis the frame information, you should analysis the data from the lowest bit (tail). For the specific analysis method and the order of the frame information of each camera, please contact technical support.

3.3. Acquisition Control

3.3.1. Acquisition

- Terms

1) Acquisition mode: To set the image generation mode of the camera. It mainly defines the number of the images in an acquisition period of the camera. An acquisition period means that from send start acquisition command to send stop acquisition command.
2) Acquisition frame count: When the acquisition mode is multi-frame, the parameter determines the number of the images in an acquisition period of the camera.
3) Acquisition speed level: To control the frame rate of the camera. The larger the acquisition speed level, the greater the frame rate; the smaller the acquisition speed level, the smaller the frame rate.
4) Acquisition burst frame count: Acquire multiple frames of images by a single trigger. For example, if the “Acquisition burst frame count” parameter is set to three, the camera automatically outputs three images after the trigger.
5) Acquisition status selection: The acquisition status is used to determine if the camera is waiting for a trigger signal. There are two modes: FrameTriggerWait and AcquisitionTriggerWait. FrameTriggerWait: After selecting this status, you can determine whether the camera is waiting for the frame start trigger signal by querying the acquisition function status. AcquisitionTriggerWait: After selecting this status, you can determine whether the camera is waiting for the trigger signal in the multi-frame acquisition status by querying the acquisition status function. The function is only used in the trigger mode and has no effect on the continuous acquisition mode.
6) Acquisition status: This function code is used together with the acquisition status selection function. For details, please refer to the acquisition status selection. During non-acquisition period (before the acquisition is started, after the acquisition is stopped), the value of the query has no meaning in the non-trigger mode.

- Related Parameters

`GX_ENUM_ACQUISITION_MODE:` Acquisition mode, the enumeration value ref: `GX_ACQUISITION_MODE_ENTRY.`

`GX_COMMAND_ACQUISITION_START:` Acquisition start command.

`GX_COMMAND_ACQUISITION_STOP:` Acquisition stop command.

`GX_INT_ACQUISITION_FRAME_COUNT:` Acquisition frame count.

`GX_INT_ACQUISITION_SPEED_LEVEL:` Acquisition speed level.

`GX_INT_ACQUISITION_BURST_FRAME_COUNT:` Number of frames to acquire by a single trigger in trigger mode.

`GX_ENUM_ACQUISITION_STATUS_SELECTOR:` Acquisition status selection, refer to: `GX_ACQUISITION_STATUS_SELECTOR_ENTRY.`

`GX_BOOL_ACQUISITION_STATUS:` Acquisition status.

- Control Workflow

  (About the concept of trigger mode, please ref the next section--*Trigger*)

Non-trigger + Continous acquisition mode:

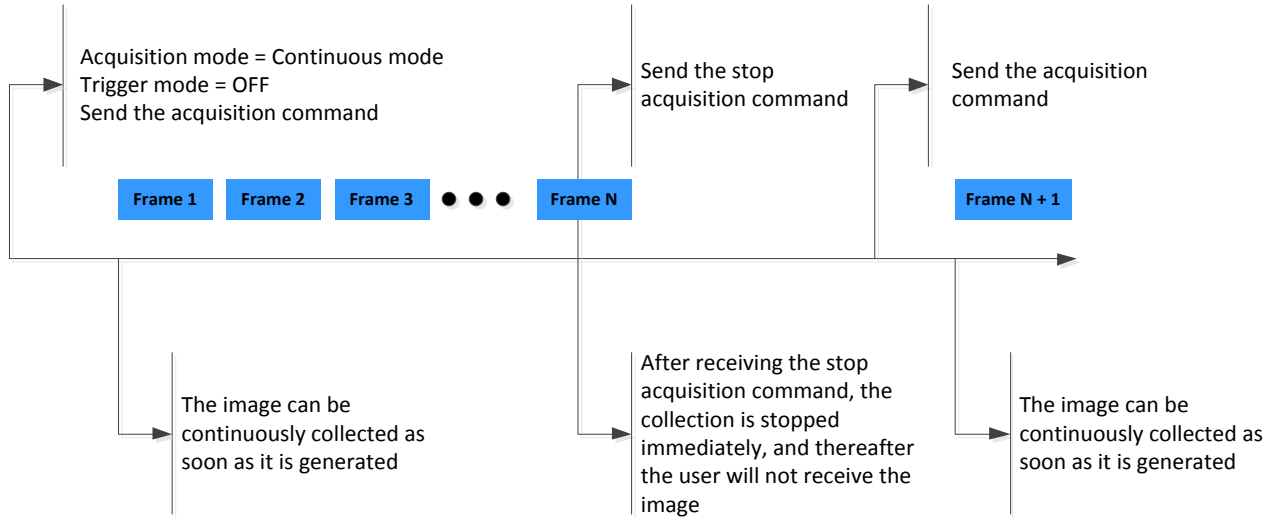

Figure 32: Non-trigger + Continous acquisition mode

Trigger + Continous acquisition mode:

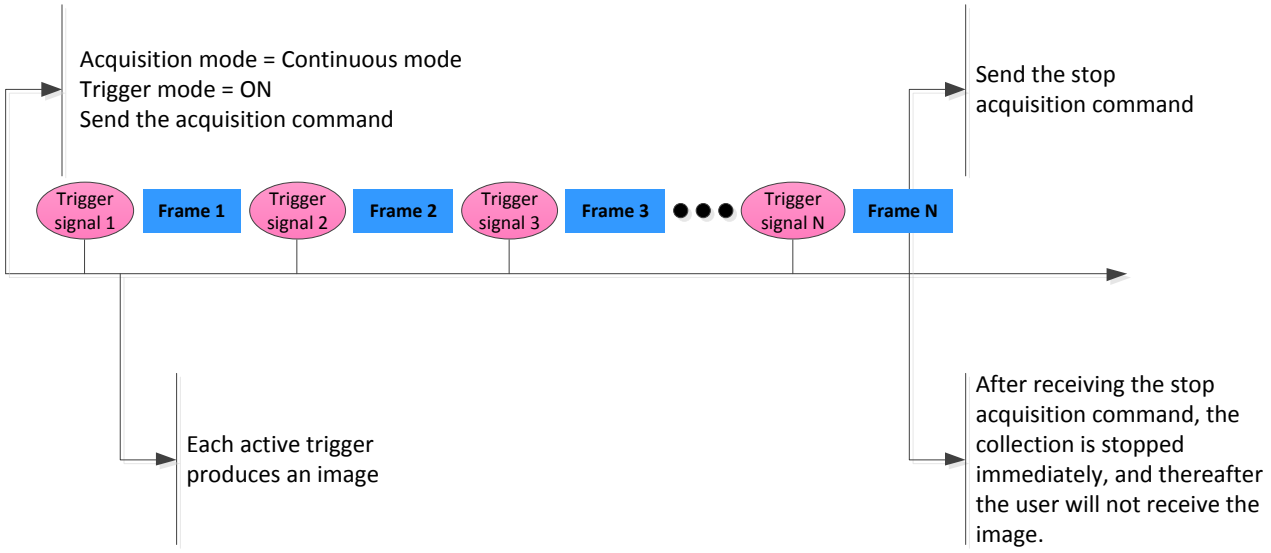

Figure 33: Trigger + continous acquisition mode

Non-trigger + Multi-frame acquisition mode:

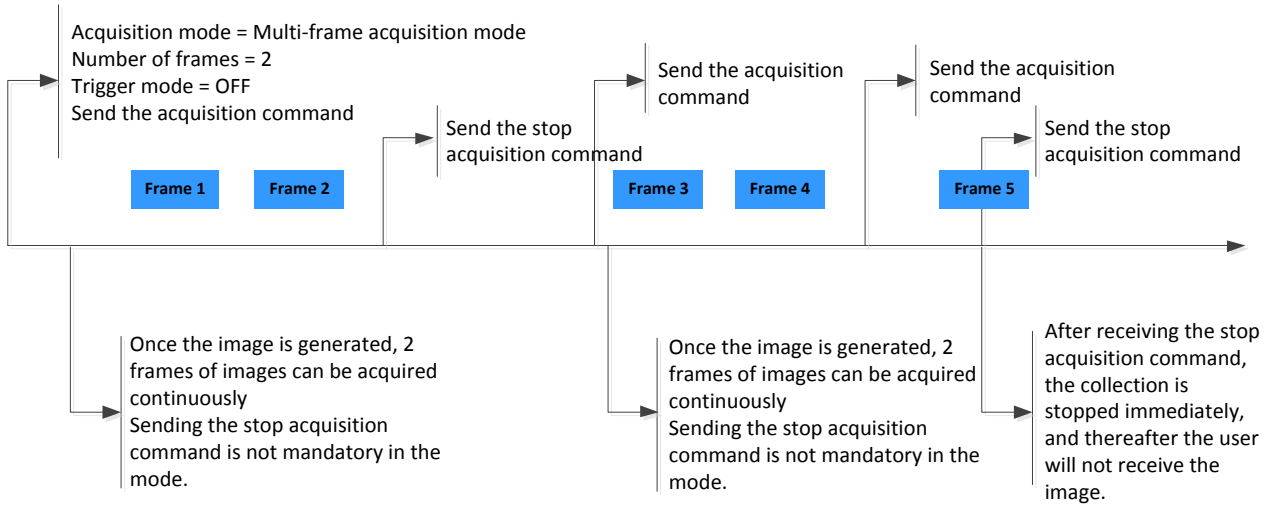

Figure 34: Non-trigger + multi-frame acquisition mode

Trigger + Multi-frame acquisition mode:

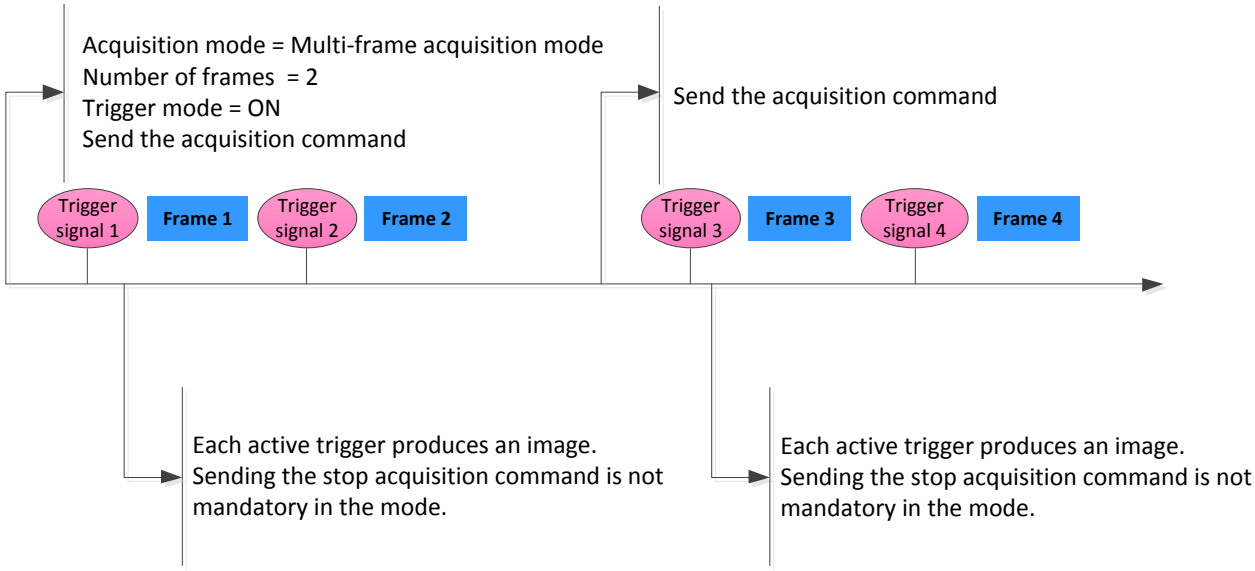

Figure 35: Trigger + multi-frame acquisition mode

Non-trigger +Single frame acquisition mode:

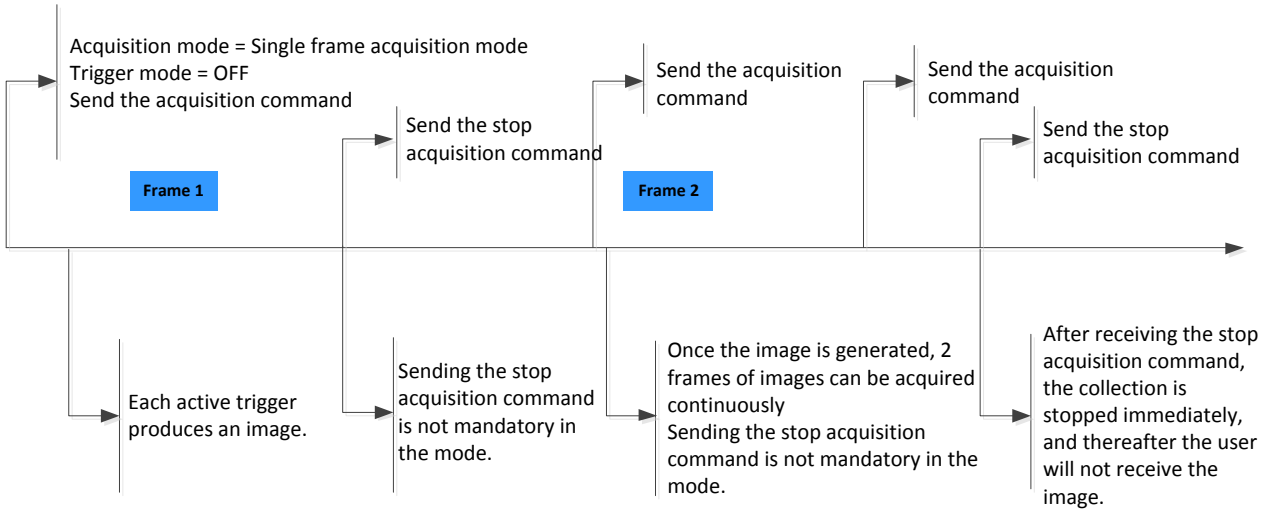

Figure 36: Non-trigger + single frame acquisition mode

Trigger +Single frame acquisition mode:

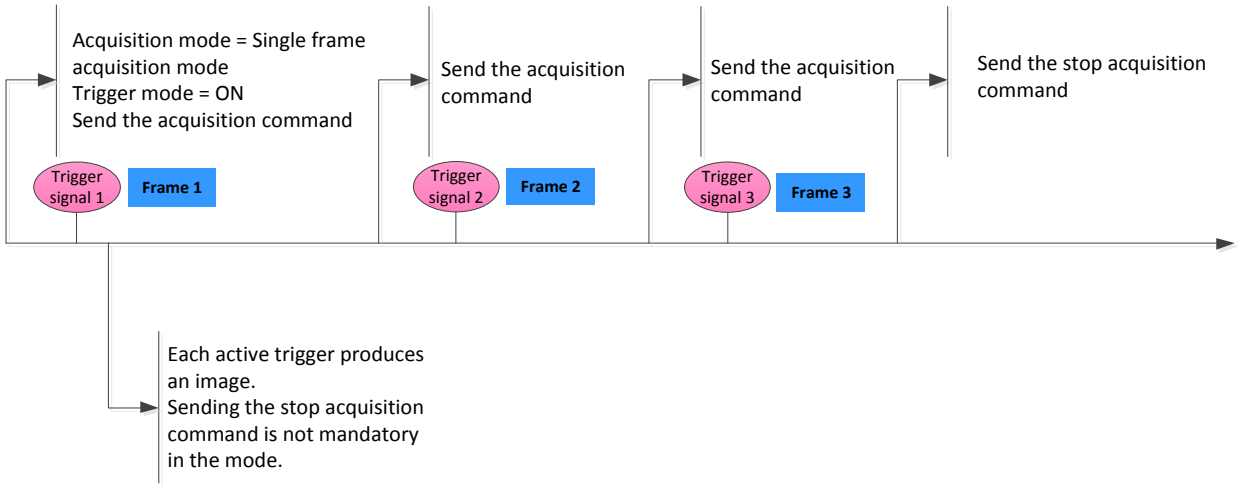

Figure 37: Trigger + single frame acquisition mode

- Code Sample

```
#include "GxIAPI.h"

// Image processing callback function.
static void GX_STDC OnFrameCallbackFun(GX_FRAME_CALLBACK_PARAM* pFrame)
{
    if (pFrame->status == 0)
    {
        // Performs some image processing operations.
    }
    return;
}
int main(int argc, char* argv[])
{
    GX_STATUS     status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    uint32_t      nDeviceNum = 0;
```

```
    // Initializes the library.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    // Updates the enumeration list for the devices.
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }

    // Opens the device.
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode   = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Sets acquisition mode. Default acquisition mode for General camera
        // is continuous mode.
        // int64_t nAcqMode = GX_ACQ_MODE_CONTINUOUS;
        // status = GXSetEnum(hDevice, GX_ENUM_ACQUISITION_MODE, nAcqMode);

        // Registers image processing callback function.
        status = GXRegisterCaptureCallback(hDevice, NULL,
                                           OnFrameCallbackFun);

        // Stream On.
        status = GXStreamOn(hDevice);

        //----------------------
        //
        // In this interval, the image will be returned to the user via the
        // OnFrameCallbackFun interface.
        //
        //----------------------

        // Sends a stop acquisition command.
        status = GXStreamOff(hDevice);

        // Unregisters image processing callback function.
        status = GXUnregisterCaptureCallback(hDevice);
    }
    status = GXCloseDevice(hDevice);
    status = GXCloseLib();
    return 0;
}
```

- Precautions

3.3.2. Trigger

3.3.2.1. General Function

- Terms

1) Trigger Source: Select the source of the trigger signal. Software trigger or hardware trigger.
2) Trigger Mode: To determine whether the trigger signal is valid or not. On -valid; OFF -invalid.
3) Trigger Polarity: The activate mode of trigger. Rising edge valid or falling edge valid.
4) Software Command: Simulation the trigger signal with software.

- Related Parameters

`GX_ENUM_TRIGGER_MODE`: Trigger Mode, the enumeration value ref: `GX_TRIGGER_MODE_ENTRY`.
`GX_COMMAND_TRIGGER_SOFTWARE`: Software Trigger Command.
`GX_ENUM_TRIGGER_ACTIVATION`: Trigger Polarity, the enumeration value ref: `GX_TRIGGER_ACTIVATION_ENTRY`.
`GX_ENUM_TRIGGER_SWITCH`: Hardware Trigger Switch, ref: `GX_TRIGGER_SWITCH_ENTRY`.
`GX_ENUM_TRIGGER_SOURCE`: Trigger Source, the enumeration value ref: `GX_TRIGGER_SOURCE_ENTRY`.
`GX_ENUM_TRIGGER_SELECTOR`: Trigger Selector, ref: `GX_TRIGGER_SELECTOR_ENTRY`.
`GX_FLOAT_TRIGGER_DELAY`: Trigger Delay.

- Code Sample

```
#include "GxIAPI.h"

// Image processing callback function.
static void GX_STDC OnFrameCallbackFun(GX_FRAME_CALLBACK_PARAM* pFrame)
{
    if (pFrame->status == 0)
    {
        // Performs some image processing operations.
    }
    return;
}

int main(int argc, char* argv[])
{
    GX_STATUS     status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    uint32_t      nDeviceNum = 0;

    // Initializes the library.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    // Updates the enumeration list for the devices.
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
```

```
        }

        // Opens the device.
        stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
        stOpenParam.openMode   = GX_OPEN_INDEX;
        stOpenParam.pszContent = "1";
        status = GXOpenDevice(&stOpenParam, &hDevice);
        if (status == GX_STATUS_SUCCESS)
        {
            // Sets the trigger mode to ON.
            status = GXSetEnum(hDevice, GX_ENUM_TRIGGER_MODE,
                               GX_TRIGGER_MODE_ON);

            // Sets the trigger activation mode to the rising edge.
            status = GXSetEnum(hDevice,GX_ENUM_TRIGGER_ACTIVATION,
                               GX_TRIGGER_ACTIVATION_RISINGEDGE);

            // Registers image processing callback function.
            status = GXRegisterCaptureCallback(hDevice, NULL,
                                               OnFrameCallbackFun);

            // Stream On.
            status = GXStreamOn(hDevice);

            //----------------------
            //
            // In this interval, the image will be returned to the user via the
            // OnFrameCallbackFun interface.
            //----------------------

            // Sends a stop acquisition command.
            status = GXStreamOff(hDevice);

            // Unregisters image processing callback function.
            status = GXUnregisterCaptureCallback(hDevice);
        }
        status = GXCloseDevice(hDevice);
        status = GXCloseLib();
        return 0;
    }
```

- Precautions

3.3.2.2. Advanced Function

- Terms

1) Trigger filtering value at rising edge: If the pulse width at the rising edge is smaller than the trigger filtering value, then the camera will not handle the trigger signal. When the pulse width at the rising edge is larger or equal to the trigger filtering value, it will generate the valid trigger signal.
2) Trigger filtering value at falling edge: If the pulse width at the falling edge is smaller than the trigger filtering value, then the camera will not handle the trigger signal. When the pulse width at the falling edge is larger or equal to the trigger filtering value, it will generate the valid trigger signal.

- Related Parameters

`GX_FLOAT_TRIGGER_FILTER_RAISING:` Trigger filtering value at rising edge.
`GX_FLOAT_TRIGGER_FILTER_FALLING:` Trigger filtering value at falling edge.

- Trigger waveform figure

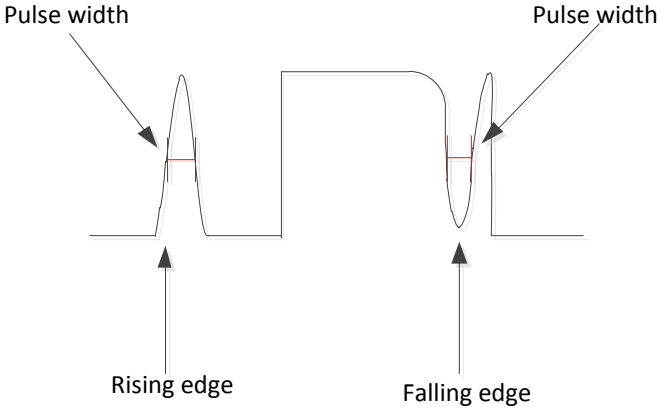

Figure 38: Trigger waveform

- Code Sample

```
// Gets the settings range of rising edge filter.
GX_FLOAT_RANGE raisingRange;
status = GXGetFloatRange(hDevice,GX_FLOAT_TRIGGER_FILTER_RAISING,&raisingRange);

// Sets the rising edge filter to the minimum value.
status = GXSetFloat(hDevice, GX_FLOAT_TRIGGER_FILTER_RAISING, raisingRange.dMin);
// Sets the rising edge filter to the maximum value.
status = GXSetFloat(hDevice, GX_FLOAT_TRIGGER_FILTER_RAISING, raisingRange.dMax);

// Gets the current rising edge filter value.
double dRaisingValue = 0;
status = GXGetFloat(hDevice, GX_FLOAT_TRIGGER_FILTER_RAISING, &dRaisingValue);
// Gets the settings range of falling edge filter.
GX_FLOAT_RANGE fallingRange;
status = GXGetFloatRange(hDevice, GX_FLOAT_TRIGGER_FILTER_FALLING, &fallingRange);

// Sets the falling edge filter to the minimum value.
status = GXSetFloat(hDevice, GX_FLOAT_TRIGGER_FILTER_FALLING, fallingRange.dMin);
// Sets the falling edge filter to the maximum value.
status = GXSetFloat(hDevice, GX_FLOAT_TRIGGER_FILTER_FALLING, fallingRange.dMax);
// Gets the current falling edge filter value.
double dFallingValue = 0;
status = GXGetFloat(hDevice, GX_FLOAT_TRIGGER_FILTER_FALLING, &dFallingValue);
```

- Precautions

3.3.3. Exposure

- Terms

1) Exposure Mode: Different modes of exposure, in Timed mode, you can use GX_FLOAT_EXPOSURE_TIME/ GX_ENUM_EXPOSURE_AUTO.
2) Exposure Time: That is shutter time, refers to the time interval of the sensor shutter from open to close, in the interval the object can be captured in the sensor. If you reduce the exposure time, the image will darker, increase the exposure time, the image will brighter.
3) Exposure Delay: The exposure delay function can effectively solve the flash delay problem. Most flashlights have a delay of at least tens of microseconds from trigger to light. When the camera is working in a small exposure mode, the fill light effect of the flash will be affected. The exposure delay is achieved by the flash signal and the delay of the actual exposure starting.

- Related Parameters

`GX_ENUM_EXPOSURE_MODE`: Exposure mode, the enumeration value ref: `GX_EXPOSURE_MODE_ENTRY`.
`GX_FLOAT_EXPOSURE_TIME`: Exposure time.
`GX_ENUM_EXPOSURE_AUTO`: Auto exposure enable, the enumeration value ref: `GX_EXPOSURE_AUTO_ENTRY`.
`GX_FLOAT_EXPOSURE_DELAY`: Exposure delay.

- Effect images

Figure 39: Original image

Figure 40: Reduce exposure time

Figure 41: Increase exposure time

- Code Sample

```
// Gets the adjustment range of exposure time.
GX_FLOAT_RANGE shutterRange;
double dExposureValue = 2.0;
status = GXGetFloatRange(hDevice, GX_FLOAT_EXPOSURE_TIME, &shutterRange);

// Sets the exposure time to the minimum.
status = GXSetFloat(hDevice, GX_FLOAT_EXPOSURE_TIME, shutterRange.dMin);
// Sets the exposure time to the maximum.
status = GXSetFloat(hDevice, GX_FLOAT_EXPOSURE_TIME, shutterRange.dMax);

//Sets the exposure mode to continuous automatic exposure
status = GXSetEnum(hDevice, GX_ENUM_EXPOSURE_AUTO,
                   GX_EXPOSURE_AUTO_CONTINUOUS);
//Set the exposure delay to 2us
status = GXSetFloat(hDevice, GX_FLOAT_EXPOSURE_DELAY, dExposureValue);
```

- Precautions

When the external light source is sunlight or direct current (DC) light source, no special requirements for the exposure time, but when the external light source is alternating current (AC) light source, the exposure time must synchronize with the external light source (under 50Hz light source, the exposure time must be a multiple of 1/100s. Under 60Hz light source, the exposure time must be a multiple of 1/120s).

3.3.4. Transfer Control

- Terms

When multiple cameras are connected to the host by switches, if trigger these cameras to acquire images at the same time, some data may lost because the large instantaneous bandwidth of the switch and the storage capacity is limited. So, you need to use frame transfer delay to avoid this problem.

In trigger mode, by setting the transmission control mode as "User Control", when the camera receives software trigger commands or hardware trigger signal and completes the image acquisition, the images will be stored in the camera inside the frame memory, waiting for the host to send "start transfer" command, and then the camera will transfer the images to the host. The transfer delay time is determined by the host. When multi-cameras are triggered simultaneously, you can set different transmission delays for each camera, to avoid the large instantaneous bandwidth of the switch.

- Related Parameters

`GX_ENUM_TRANSFER_CONTROL_MODE`: Selects the control mode for the transfer.
`GX_ENUM_TRANSFER_OPERATION_MODE`: Selects the operation mode for the transfer.
`GX_COMMAND_TRANSFER_START`: Starts the transfer.
`GX_INT_TRANSFER_BLOCK_COUNT`: The number of transferred frames.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;
// The premise must be to ensure that the trigger mode is open.
emStatus = GXSetEnum(hDevice, GX_ENUM_TRIGGER_MODE, GX_TRIGGER_MODE_ON);

// Sets the transfer control mode to user control mode.
emStatus = GXSetEnum(hDevice, GX_ENUM_TRANSFER_CONTROL_MODE,
                     GX_ENUM_TRANSFER_CONTROL_MODE_USERCONTROLED);

// Sets the transfer operation mode to the specified transfer frame mode.
emStatus = GXSetEnum(hDevice, GX_ENUM_TRANSFER_OPERATION_MODE,
                     GX_ENUM_TRANSFER_OPERATION_MODE_MULTIBLOCK);

// Sets the number of output frames per command.
emStatus = GXSetInt(hDevice, GX_INT_TRANSFER_BLOCK_COUNT, 1);

// Sends a software trigger signal (or external trigger) after the start of
// acquisition.
emStatus = GXStreamOn(hDevice);
emStatus = GXSendCommand(hDevice, GX_COMMAND_TRIGGER_SOFTWARE);

// Sents a transfer command after the image is triggered.
emStatus = GXSendCommand(hDevice, GX_COMMAND_TRANSFER_START);
```

- Precautions

The transfer control function can only work in trigger mode.

3.3.5. Frame Store Mechanism

- Terms

Frame Store Coverage: When the average bandwidth of the data being written to the internal frame is greater than the average bandwidth of the data read from it, then the frame store will full. If the frame store is full, the image data in it will be overwritten.

- Related Parameters

`GX_BOOL_FRAMESTORE_COVER_ACTIVE`：Enable the frame store coverage function.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

// Enables the frame store coverage.
status = GXSetBool(m_hDevice, GX_BOOL_FRAMESTORE_COVER_ACTIVE, true);
// Disables the frame store coverage.
status = GXSetBool(m_hDevice, GX_BOOL_FRAMESTORE_COVER_ACTIVE, false);
```

3.3.6. Frame Rate Control

- Terms

1) Acquisition frame rate mode: The function is to control whether to open the frame rate control mode. "on" is to start the frame rate control and "off" is to disable the frame rate control. For more details, please ref GxIAPI.h: GX_ACQUISITION_FRAME_RATE_MODE_ENTRY.
2) Acquisition frame rate: Desired acquisition frame rate.
3) Current acquisition frame rate: Acquisition frame mode in actual operation.

- Related Parameters

`GX_ENUM_ACQUISITION_FRAME_RATE_MODE`: This enumeration sets the acquisition frame rate mode.

`GX_FLOAT_ACQUISITION_FRAME_RATE`: Acquisition frame rate.

`GX_FLOAT_CURRENT_ACQUISITION_FRAME_RATE`: The current acquisition frame rate.

- Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;

//Enables the frame rate adjustment mode.
status = GXSetEnum(m_hDevice, GX_ENUM_ACQUISITION_FRAME_RATE_MODE,
                   GX_ACQUISITION_FRAME_RATE_MODE_ON);
//Sets the acquisition frame rate. Assuming the setting is 10.0. The
//user can set this value according to actual needs.
status = GXSetFloat(m_hDevice, GX_FLOAT_ACQUISITION_FRAME_RATE, 10.0);
```

- Precautions

If the value of "GX_FLOAT_ACQUISITION_FRAME_RATE" was set too large, beyond the actual operation ability of the camera, the camera will run at the maximum frame rate it can achieve. That is the value of the current acquisition frame rate (GX_FLOAT_CURRENT_ACQUISITION_FRAME_RATE). (The frame rate is affected by the exposure time, ROI and etc.)

3.4. Digital IO

3.4.1. Pin Control

- Related Parameters

  GX_ENUM_LINE_SELECTOR：Line（Pin) selector.

  GX_ENUM_LINE_MODE：Line （Pin) mode.

  GX_BOOL_LINE_INVERTER：Line (Pin) level inversion.

  GX_ENUM_LINE_SOURCE：Line (Pin) output source.

  GX_BOOL_LINE_STATUS：Line（Pin) status.

  GX_INT_LINE_STATUS_ALL：Status of all lines (pins).

  GX_ENUM_USER_OUTPUT_SELECTOR: This enumeration selects the user settable output signal to configure.

  GX_BOOL_USER_OUTPUT_VALUE: This boolean value sets the state of the selected user settable output signal.

  GX_FLOAT_PULSE_WIDTH：User-defined pulse width.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

//For example, the pin is selected as Line2.
emStatus = GXSetEnum(hDevice, GX_ENUM_LINE_SELECTOR,
                     GX_ENUM_LINE_SELECTOR_LINE2);

//Sets the pin direction to output mode.
emStatus = GXSetEnum(hDevice, GX_ENUM_LINE_MODE, GX_ENUM_LINE_MODE_OUTPUT);

//Optional operation: pin level inversion
//emStatus = GXSetBool(hDevice, GX_BOOL_LINE_INVERTER, true);

//You can set the output source to the strobe. The code is as follows
emStatus = GXSetEnum(hDevice, GX_ENUM_LINE_SOURCE,
                     GX_ENUM_LINE_SOURCE_STROBE);

//You can set the output source to the user-defined output. The code is as follows
emStatus = GXSetEnum(hDevice, GX_ENUM_LINE_SOURCE,
                     GX_ENUM_LINE_SOURCE_USEROUTPUT0);

// You can set the output source to a user-defined output value. The code is
// as follows
emStatus=GXSetEnum(hDevice,GX_ENUM_USER_OUTPUT_SELECTOR,GX_USER_OUTPUT_SE
                   LECTOR_OUTPUT0);
emStatus = GXSetBool(hDevice, GX_BOOL_USER_OUTPUT_VALUE,true);

//Gets the status of the line2 pin (The current pin is selected as line2).
bool bLineStatus = true;
emStatus = GXGetBool(hDevice, GX_BOOL_LINE_STATUS, &bLineStatus);

// Gets the status of all pins.
int64_t nAllLineStatus = 0;
emStatus = GXGetInt(hDevice, GX_INT_LINE_STATUS_ALL, &nAllLineStatus);
```

3.4.2. The I/O Control of the USB2.0 Camera

The IO control of USB2.0 is special, and the "pins control" operation is not the same operation process as shown in the previous section. The IO control of USB2.0 involves the following three functions:

- Terms

1) User IO output mode: There are two modes: strobe mode and user-defined mode. In the strobe mode, the camera sends trigger signals to activate the strobe (the trigger signal has rising edge and falling edge). In user-defined mode, you can set the camera's output level to meet the special demand, such as control the LED lamp. (the output level is high or low)
2) Output signal polarity: When the output signal mode is strobe, the polarity means rising edge or falling edge, and when the output signal mode is user-defined, the polarity means high level or low level.
3) Strobe switch: The switch just works on the strobe mode. When the switch is set to open, it will output strobe signal, if the switch is set to close, it will not output strobe signal.

- Related Parameters

`GX_ENUM_USER_OUTPUT_MODE`: The output mode of the user IO, the enumeration value ref: `GX_USER_OUTPUT_MODE_ENTRY` (USB2.0 camera only).

`GX_BOOL_USER_OUTPUT_VALUE`: The user-defined output value (If use the USB2.0 camera, the value means the polarity of the output signal, when the output signal mode is strobe, the polarity means rising edge or falling edge, and when the output signal mode is user-defined, the polarity means high level or low level).

`GX_ENUM_STROBE_SWITCH`: Strobe switch, the enumeration value ref: `GX_STROBE_SWITCH_ENTRY` (USB2.0 camera only).

- Control Flow Chart

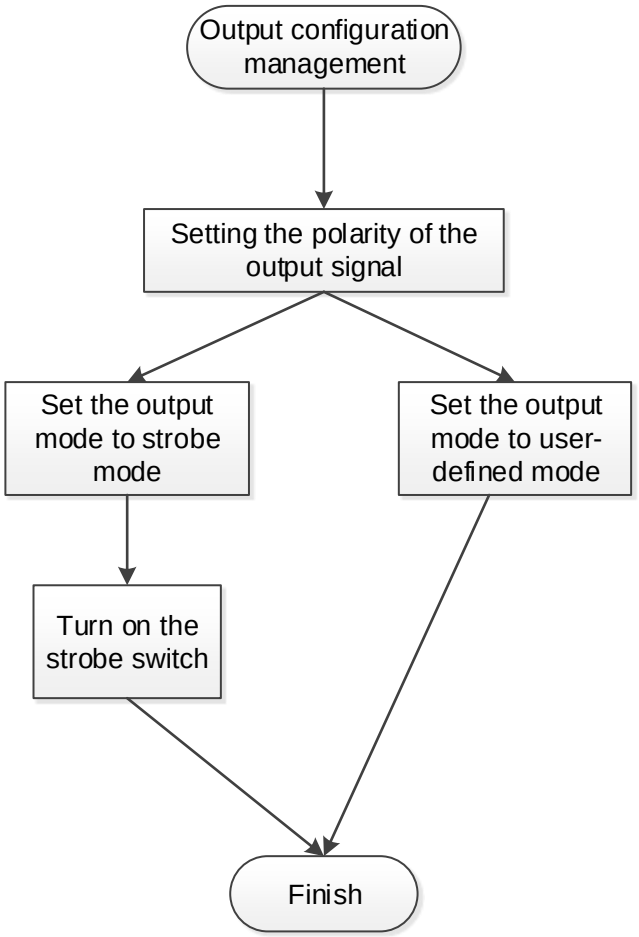

Figure 42: IO control flowchart

- Code sample:

```
#include"GxIAPI.h"

int main(intargc, char* argv[])
{
   GX_STATUS status = GX_STATUS_SUCCESS;
   GX_DEV_HANDLE hDevice = NULL;
   GX_OPEN_PARAM stOpenParam;
   uint32_t nDeviceNum = 0;

   // Initializes the library.
   status = GXInitLib();
   if (status!= GX_STATUS_SUCCESS)
   {
      return 0;
   }

   //Updates the enumeration list for the devices.
   status = GXUpdateDeviceList(&nDeviceNum, 1000);
   if ((status != GX_STATUS_SUCCESS) || (nDeviceNum<= 0))
   {
      return 0;
   }

   //Opens the device
   stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
```

```
    stOpenParam.openMode = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Sets the user-defined output polarity to high
        status = GXSetBool(hDevice,GX_BOOL_USER_OUTPUT_VALUE,true);

        //Sets the output mode to the strobe mode
        status = GXSetEnum(hDevice,
                           GX_ENUM_USER_OUTPUT_MODE,
                           GX_USER_OUTPUT_MODE_STROBE);

//Turn on the strobe switch
        status = GXSetEnum(hDevice,
                           GX_ENUM_STROBE_SWITCH,
                           GX_STROBE_SWITCH_ON);
    }
    status = GXCloseDevice(hDevice);
    GXCloseLib();
    return 0;
}
```

- Precautions

3.5. Analog Control

3.5.1. Gain

The gain is a multiplication factor for improving the value of the pixels, and the effect is to increase the brightness of the image. In certain conditions (external environments, etc.) the sensor does not have the desired saturation, so the gain is adjusted. Of course, improving gain also increases noise, so improving the gain does not improve the dynamic range of actual pixels. So, when you need to improve the image brightness, you should first consider to adjust the exposure time, only when adjusting the exposure time can not meet the requirements, then you can adjust the gain.

- Terms

1) Gain channel selection: You should select the channel before adjusting the gain, there are four channels: ALL, Red, Green, and Blue.
2) Gain control mode: Manual adjustment and automatical adjustment.

- Related Parameters

`GX_ENUM_GAIN_SELECTOR`： Gain channel selector，the enumeration value ref: `GX_GAIN_SELECTOR_ENTRY`.

`GX_FLOAT_GAIN`： The gain value.

`GX_ENUM_GAIN_AUTO`： Auto gain enable control，the enumeration value ref: `GX_GAIN_AUTO_ENTRY`.

- Effect images

Figure 43: Original image

Figure 44: Reduce gain

Figure 45: Increase gain

- Code Sample:

```
// Manual gain control------------------------
// Use the GXGetEnumEntryNums and GXGetEnumDescription interfaces to
// query the GX_ENUM_GAIN_SELECTOR types supported by the current camera.
// Please refer to the interface descriptions, which are omitted here.

// Assumes that the current camera supports GX_GAIN_SELECTOR_ALL.

// Selects the gain channel type.
status = GXSetEnum(hDevice, GX_ENUM_GAIN_SELECTOR, GX_GAIN_SELECTOR_ALL);
//status = GXSetEnum(hDevice, GX_ENUM_GAIN_SELECTOR, GX_GAIN_SELECTOR_RED);
//status = GXSetEnum(hDevice, GX_ENUM_GAIN_SELECTOR, GX_GAIN_SELECTOR_GREEN);
//status = GXSetEnum(hDevice, GX_ENUM_GAIN_SELECTOR, GX_GAIN_SELECTOR_BLUE);

// Gets the adjustment range of the gain.
GX_FLOAT_RANGE gainRange;
status = GXGetFloatRange(hDevice, GX_FLOAT_GAIN, &gainRange);

// Sets the gain to the minimum.
status = GXSetFloat(hDevice, GX_FLOAT_GAIN, gainRange.dMin);
// Sets the gain to the maximum.
status = GXSetFloat(hDevice, GX_FLOAT_GAIN, gainRange.dMax);

// Sets the auto gain adjustment to continuous mode.
```

```
status = GXSetEnum(hDevice, GX_ENUM_GAIN_AUTO, GX_GAIN_AUTO_CONTINUOUS);
```

- Precautions

For USB2.0 camera, the unit of the gain is not "dB", it is the value of the register, and it has an adjustment range, the larger the value, the larger the magnification of the gain.

3.5.2. Black Level

- Terms

1) Black Level: The black level is to define the reference level of the image data. Adjust the black level will not affect the amplification of the signal. It only shifts the signal up and down. Up the black level, the image is brighter, down the black level, the image is darker.
2) Control Mode of Black Level: Manual adjustment and automatic adjustment.

- Related Parameters

`GX_ENUM_BLACKLEVEL_SELECTOR`: Black level channel selector, the enumeration value ref: `GX_BLACKLEVEL_SELECTOR_ENTRY`.

`GX_FLOAT_BLACKLEVEL`: The black level value.

`GX_ENUM_BLACKLEVEL_AUTO`: Black level auto function enable，the enumeration value ref: `GX_BLACKLEVEL_AUTO_ENTRY`.

- Effect Images

Figure 46: Original image

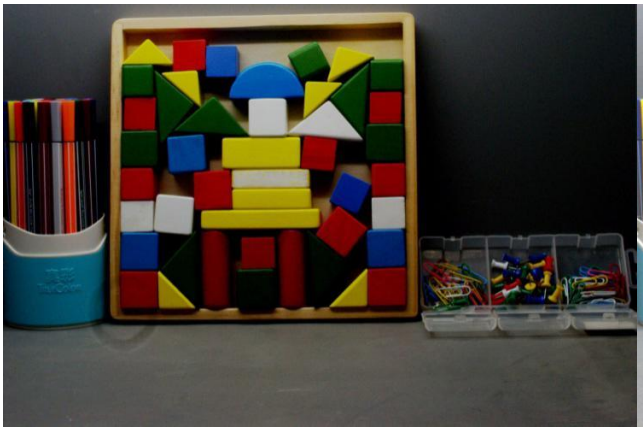

Figure 47: Reduce black level

Figure 48: Increase black level

- Code Sample

```
//Manual black level adjustment ------------------------
//Use the GXGetEnumEntryNums and GXGetEnumDescription interfaces to
//query the GX_ENUM_BLACKLEVEL_SELECTOR types supported by the current camera
//Please refer to the interface descriptions, which are omitted here.
```

```
//Assumes that the current camera supports GX_BLACKLEVEL_SELECTOR_ALL.

//Selects the black level channel type.
status = GXSetEnum(hDevice, GX_ENUM_BLACKLEVEL_SELECTOR, GX_BLACKLEVEL_SELECTOR_ALL);
//status=GXSetEnum(hDevice, GX_ENUM_BLACKLEVEL_SELECTOR, GX_BLACKLEVEL_SELECTOR_RED);
//status=GXSetEnum(hDevice, GX_ENUM_BLACKLEVEL_SELECTOR, GX_BLACKLEVEL_SELECTOR_GREEN);
//status=GXSetEnum(hDevice, GX_ENUM_BLACKLEVEL_SELECTOR,
//                 GX_BLACKLEVEL_SELECTOR_BLUE);

// Gets the range of the black level.
GX_FLOAT_RANGE blackLevelRange;
status = GXGetFloatRange(hDevice, GX_FLOAT_BLACKLEVEL, &blackLevelRange);

// Sets the black level to the minimum.
status = GXSetFloat(hDevice, GX_Float_BLACKLEVEL, blackLevelRange.dMin);
// Sets the black level to the maximum.
status = GXSetFloat(hDevice, GX_Float_BLACKLEVEL, blackLevelRange.dMax);

//Automatic black level adjustment--------------------------
//Sets to continuous automatic black level mode.
status = GXSetEnum(hDevice, GX_ENUM_ BLACKLEVEL _AUTO, GX_BLACKLEVEL_AUTO_CONTINUOUS);
```

- Precautions

3.5.3. White Balance

- Terms

1) White Balance: Under different color temperatures, the object's color may change. Especially the white objects. Indoors, the white objects look with orange tonal under the tungsten light which is of a low color temperature, under that light condition, the image will yellow shift; if in the blue sky which is of high color temperature, the image will be bluer. In order to minimize the external light's impact and to restore the real color of the object, color correction is required, to achieve the correct color balance, known as the white balance adjustment.
2) White Balance Control Mode: Manual adjustment and automatical adjustment.
3) Auto White Balance Light Environment: You should set this value by the light environment of the camera, it is better to the white balance effect.
4) Auto White Balance ROI: Auto White Balance function use the image data of the "white dot" area (ROI) to calculate the white balance coefficient, and then use the coefficient to handle the components of the image, in order to make the R/G/B component the same in the ROI. The auto white balance function is only available on color sensors.

- Related Parameters

`GX_ENUM_BALANCE_RATIO_SELECTOR`：White balance channel selector, the enumeration value ref：`GX_BALANCE_RATIO_SELECTOR_ENTRY`.

`GX_FLOAT_BALANCE_RATIO`： The white balance ratio.

`GX_ENUM_BALANCE_WHITE_AUTO`： Automatic white balance enables, the enumeration value ref：`GX_BALANCE_WHITE_AUTO_ENTRY`.

`GX_ENUM_AWB_LAMP_HOUSE`： Automatic white balance illumination environment, the enumeration value ref：`GX_AWB_LAMP_HOUSE_ENTRY`.

`GX_INT_AWBROI_OFFSETX`: This value sets the X offset (left offset) for the ROI in pixels for Auto White Balance.

`GX_INT_AWBROI_OFFSETY`: This value sets the Y offset (top offset) for the ROI for Auto White Balance.

`GX_INT_AWBROI_WIDTH`: This value sets the width of the ROI in pixels for Auto White Balance.

`GX_INT_AWBROI_HEIGHT`: This value sets the height of the ROI in pixels for Auto White Balance.

- Effect Images

Figure 49: Image color shift under D65 environment

Figure 50: After white balance adjustmen

- Code Sample

```
// The manule white balance adjustment-----------------------
// Uses the GXGetEnumEntryNums and GXGetEnumDescription interfaces to
// query the GX_ENUM_BALANCE_RATIO_SELECTOR types supported by the current
// camera.
// Please refer to the interface description, which is omitted here.

// Assumes that the current camera supports GX_BALANCE_RATIO_SELECTOR_RED.

// Selects the white balance channel.
status=GXSetEnum(hDevice,GX_ENUM_BALANCE_RATIO_SELECTOR,GX_BALANCE_RATIO
                 _SELECTOR_RED);
// status=GXSetEnum(hDevice,GX_ENUM_BALANCE_RATIO_SELECTOR,GX_BALANCE_RAT
//                  IO_SELECTOR_GREEN);
// status=GXSetEnum(hDevice,GX_ENUM_BALANCE_RATIO_SELECTOR,GX_BALANCE_RAT
//                  IO_SELECTOR_BLUE);

// Gets the range of the white balance adjustment.
GX_FLOAT_RANGE ratioRange;
status = GXGetFloatRange(hDevice, GX_FLOAT_BALANCE_RATIO, &ratioRange);

// Sets the white balance ratio to the minimum.
status = GXSetFloat(hDevice, GX_FLOAT_BALANCE_RATIO, ratioRange.dMin);
// Sets the white balance ratio to the maximum.
status = GXSetFloat(hDevice, GX_FLOAT_BALANCE_RATIO, ratioRange.dMax);

// Sets the automatic white balance ROI (the user can modify the parameters
// according to their own need).
status = GXSetInt(hDevice, GX_INT_AWBROI_WIDTH, 100);
status = GXSetInt(hDevice, GX_INT_AWBROI_HEIGHT, 100);
status = GXSetInt(hDevice, GX_INT_AWBROI_OFFSETX, 0);
status = GXSetInt(hDevice, GX_INT_AWBROI_OFFSETY, 0);

// Auto white balance setting--------------------------
// Sets the lighting environment for the automatic white balance, such as
// the current camera in the fluorescent environment.
status=GXSetEnum(hDevice,GX_ENUM_AWB_LAMP_HOUSE,
                 GX_AWB_LAMP_HOUSE_FLUORESCENCE);
// Sets to continuous automatic white balance mode.
status=GXSetEnum(hDevice,GX_ENUM_BALANCE_WHITE_AUTO,
                 GX_BALANCE_WHITE_AUT O_CONTINUOUS);
```

- Precautions

If the automatic white balance function fails, it may be because the ROI of the automatic white balance is too small to find the white spots.

3.6. Transport Layer Control

3.6.1. PayloadSize

- Terms

PayloadSize: This parameter indicates the size (unit: bytes) of each image data, which outputting from the camera. According to this value, you can allocate the buffer size to capture images. When the frame information is closed, the output image does not attach the frame information, and the size of the payloadSize represents the image size. When the frame information is opened, the output image comes with the frame information, and then the size of the payloadSize is equal the image size and the frame information size.

- Related Parameters

`GX_INT_PAYLOAD_SIZE`: The size of the payload.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;
int64_t nValue = 0;
status = GXGetInt(hDevice, GX_INT_PAYLOAD_SIZE, &nValue);
```

- Precautions

When the user allocates the buffer for image data, it is strongly recommended that the user use the value of payloadSize.

3.6.2. IP Configuration

- Related Parameters

`GX_BOOL_GEV_CURRENT_IPCONFIGURATION_LLA`: Configure IP with LLA mode.
`GX_BOOL_GEV_CURRENT_IPCONFIGURATION_DHCP`: Configure IP with DHCP mode.
`GX_BOOL_GEV_CURRENT_IPCONFIGURATION_PERSISTENTIP`: Configure IP with persistent mode.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;
// This example is described in persistent IP configuration mode, and other
// IP configuration modes are similar.
// Sets the IP configuration mode to persistent IP mode.
status = GXSetBool(hDevice, GX_BOOL_GEV_CURRENT_IPCONFIGURATION_PERSISTENTIP, true);
```

3.6.3. Estimate the Bandwidth

- Terms

Estimate the bandwidth: The network bandwidth required to transfer image data under the current condition of the camera.

- Related Parameters

`GX_INT_ESTIMATED_BANDWIDTH`: The bandwidth estimated.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

// Reads the current estimated bandwidth.
int64_t nValue = 0;
status = GXGetInt(hDevice, GX_INT_ESTIMATED_BANDWIDTH, &nValue);
```

3.6.4. The Heartbeat Timeout Time of the Device

- Terms

The heartbeat timeout time of the device: If the device does not receive the GVCP command packet from the host in this time, it will disconnect with the host.

- Related Parameters

`GX_INT_GEV_HEARTBEAT_TIMEOUT`: The heartbeat timeout time of the device, unit: ms.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

// Reads the current heartbeat timeout time of the device.
int64_t nValue = 0;
status = GXGetInt(hDevice, GX_INT_GEV_HEARTBEAT_TIMEOUT, &nValue);

// Reads the settable range of the device's heartbeat timeout time. Please
// refer to the GXGetIntRange interface for use.
// Sets the heartbeat timeout time of the device.
status = GXSetInt(hDevice, GX_INT_GEV_HEARTBEAT_TIMEOUT, nValue);
```

- Precautions

It is strongly recommended that the user use the default value unless special situation allow the user to modify the value based on the application environment.

3.6.5. Packet Size

- Terms

Packet Size: It means that the GigE Vision camera transfers the network packet size of the stream channel data to the host, unit: bytes.

- Related Parameters

`GX_INT_GEV_PACKETSIZE`: The packet size.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

// Reads the packet size.
int64_t nPacketSize = 0;
status = GXGetInt(hDevice, GX_INT_GEV_PACKETSIZE, &nPacketSize);
```

3.6.6. Packet Delay

- Terms

Packet delay: To control the transfer image data bandwidth of the GigE Vision camera. The packet delay is the number of idle clocks that are inserted between adjacent network packets. Increase the packet delay will reduce the camera's bandwidth usage, and may also reduce the frame rate of the camera.

- Related Parameters

`GX_INT_GEV_PACKETDELAY`: The packet delay.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;
// Reads the packet delay.
int64_t nPacketDelay = 0;
status = GXGetInt(hDevice, GX_INT_GEV_PACKETDELAY, &nPacketDelay);
```

3.6.7. Link Speed

- Terms

Link Speed: The current network environment is gigabit network or 100M Ethernet.

- Related Parameters

`GX_INT_GEV_LINK_SPEED`: Link Speed.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

// Reads the link speed of the current network environment.
int64_t nLinkSpeed = 0;
status = GXGetInt(hDevice, GX_INT_GEV_LINK_SPEED, &nLinkSpeed);
```

3.7. Bandwidth Control

- Terms

The USB3 Vison series camera supports bandwidth control function, to control the bandwidth upper limit of a single device. The bandwidth of the current device will not change when the link bandwidth limit of the device is greater than the bandwidth of the current device. When the link bandwidth limit of the device is less than the current device, the bandwidth of the current device will be reduced to the limit of the link bandwidth of the device. The current device acquisition bandwidth can be read from the camera.

For example: The current acquisition bandwidth of the device is 35000000Bps, the device link bandwidth limit is 40000000Bps, then the current acquisition device bandwidth is still 35000000Bps; if the current acquisition device bandwidth is 70000000Bps, the device link bandwidth limit is 40000000Bps, then the current acquisition device bandwidth is still 40000000Bps.

- Related Parameters

`GX_INT_DEVICE_LINK_SELECTOR`: Selects which Link of the device to control.
`GX_ENUM_DEVICE_LINK_THROUGHPUT_LIMIT_MODE`: Device bandwidth limit mode.
`GX_INT_DEVICE_LINK_THROUGHPUT_LIMIT`: Device bandwidth limit.
`GX_INT_DEVICE_LINK_CURRENT_THROUGHPUT`: Current device acquisition bandwidth.

- Code Sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

// Enables the bandwidth limit mode.
GX_DEVICE_LINK_THROUGHPUT_LIMIT_MODE_ENTRY nValue;
nValue= GX_DEVICE_LINK_THROUGHPUT_LIMIT_MODE_ON;
status = GXSetEnum(hDevice, GX_ENUM_DEVICE_LINK_THROUGHPUT_LIMIT_MODE, nValue);

// Reads the current bandwidth value.
int64_t nLinkThroughputVal;
status = GXGetInt(hDevice, GX_INT_DEVICE_LINK_CURRENT_THROUGHPUT,
                  &nLinkThroughputVal);

// Sets the limiting value of the bandwidth.
int64_t nLinkThroughputLimitVal = 40000000;
status =GXSetInt(hDevice,GX_INT_DEVICE_LINK_THROUGHPUT_LIMIT,
                 nLinkThroughputLimitVal);
```

- Precautions

When the device bandwidth limit mode is opened or the bandwidth of the device is changed before, you should stop the acquisition.

3.8. Event Control

- Terms

When event notification is set to "on", the camera can generate an "event" and transfer a related event message to the computer whenever a specific situation has occurred. For MER-Gx series camera, the camera can generate and transfer events for the following types of situations:

- The camera has ended exposure
- An image block is discarded
- The trigger signal overflow
- The image frame buffer is not empty
- The event array is overrun

Every event has a corresponding enable status, and in default all the events' enable status are disable.

- Related Parameters

`GX_ENUM_EVENT_SELECTOR`: Event source selector, ref `GX_EVENT_SELECTOR_ENTRY`.
`GX_ENUM_EVENT_NOTIFICATION`: Event enable, ref: `GX_EVENT_NOTIFICATION_ENTRY`.
`GX_INT_EVENT_EXPOSUREEND`: Exposure end event ID.
`GX_INT_EVENT_EXPOSUREEND_TIMESTAMP`: The timestamp for exposure end event.
`GX_INT_EVENT_EXPOSUREEND_FRAMEID`: The frame ID for exposure end event.
`GX_INT_EVENT_BLOCK_DISCARD`: This enumeration value indicates the BlockDiscard event ID.
`GX_INT_EVENT_BLOCK_DISCARD_TIMESTAMP`: The timestamp for the BlockDiscard event.
`GX_INT_EVENT_OVERRUN`: The EventOverrun event ID.
`GX_INT_EVENT_OVERRUN_TIMESTAMP`: The timestamp of the EventOverrun event.
`GX_INT_EVENT_FRAMESTART_OVERTRIGGER`: The ID of the FrameStartOverTrigger event.
`GX_INT_EVENT_FRAMESTART_OVERTRIGGER_TIMESTAMP`: The timestamp of the FrameStartOvertrigger event.
`GX_INT_EVENT_BLOCK_NOT_EMPTY`: The BlockNotEmpty event ID.
`GX_INT_EVENT_BLOCK_NOT_EMPTY_TIMESTAMP`: The timestamp of the BlockNotEmpty event.
`GX_INT_EVENT_INTERNAL_ERROR`: The event ID of internal error.
`GX_INT_EVENT_INTERNAL_ERROR_TIMESTAMP`: The timestamp of the internal error event.

- Code Sample

See *GXRegisterFeatureCallback* interface description.

3.9. LUT (Look-up Table) Control

- Terms

LUT: The look-up table is a mapping table for pixel value conversions, which corresponds to pixel value one by one, and is used to change the output of each pixel value. It can be used for the conversion of high bit depth images to low bit depth images, and can also be used for simple pixel value conversion of the same bit depth images. The look-up table gives the user maximum flexibility to implement linear and non-linear conversion of pixel values (such as logarithmic and exponential corrections) without consuming valuable computing resources in the system.

- Related Parameters

`GX_ENUM_LUT_SELECTOR`: LUT selector, ref: `GX_LUT_SELECTOR_ENTRY`.
`GX_BUFFER_LUT_VALUEALL`: The value of LUT.
`GX_BOOL_LUT_ENABLE`：LUT enable
`GX_INT_LUT_INDEX`：LUT index

`GX_INT_LUT_VALUE`：LUT value

- Code Sample
  See *GXSetBuffer*, *GXGetBuffer* interfaces description.
- Precautions

3.10. User Configuration

This section describes the control of all parameter configurations for the camera, allowing the user to save or load the vendor parameter group or user parameter group.

- Terms

1) Parameter type: The user selects a type of parameters (default, userset0) to save or load.
2) Load parameters: Load the selected parameters to the camera.
3) Save parameters: Save the camera's current parameters to the user selected parameter group.
4) Set parameters: Set the parameter group used when the device is powered on.

- Related Parameters:

`GX_ENUM_USER_SET_SELECTOR`: Select the parameters to be used (default, userset0), the default parameters are read-only, can not be modified. When you select the default parameter, you can only select load operation and not save the operation, the enumeration value ref: `GX_USER_SET_SELECTOR_ENTRY`.

`GX_COMMAND_USER_SET_LOAD`: This command loads the selected configuration set from the non-volatile memory in the camera to the volatile memory and makes the selected set to the active configuration set. Once the selected set is loaded, the parameters in the selected set will control the camera.

`GX_COMMAND_USER_SET_SAVE`: This command copies the parameters in the current active configuration set into the selected user set in the camera's non-volatile memory.

`GX_ENUM_USER_SET_DEFAULT`: This enumeration sets the configuration set to be used as the default startup set. The configuration set that has been selected as the default startup set will be loaded as the active set whenever the camera is powered on or reset, ref: `GX_USER_SET_DEFAULT_ENTRY`.

- Code Sample

```
// Uses the GXGetEnumEntryNums and GXGetEnumDescription interfaces to
// query the GX_ENUM_USER_SET_SELECTOR type supported by the current camera.
// Please refer to the interface descriptions, which are omitted here.

// Assume that the current camera supports two parameter groups, such as the
// vendor default parameter group, and user parameter group 0.

//Chooses to load the vendor default parameter group.
status = GXSetEnum(hDevice, GX_ENUM_USER_SET_SELECTOR,
                   GX_ENUM_USER_SET_SELECTOR_DEFAULT);
status = GXSendCommand(hDevice, GX_COMMAND_USER_SET_LOAD);

// Chooses to load the user parameter group 0.
status = GXSetEnum(hDevice, GX_ENUM_USER_SET_SELECTOR,
```

```
                        GX_ENUM_USER_SET_SELECTOR_USERSET0);
    status = GXSendCommand(hDevice, GX_COMMAND_USER_SET_LOAD);

    //Sets the startup parameter group to the vendor parameter group.
    emStatus = GXSetEnum(hDevice, GX_ENUM_USER_SET_DEFAULT,
                         GX_ENUM_USER_SET_DEFAULT_DEFAULT);
```

- Precautions

3.11. Set Camera's IP Address

3.11.1. Configuration Static IP Address

- Terms

1) Static IP: Configuration the camera IP in a static IP mode.
2) DHCP: Configure the camera IP with DHCP, which is the IP allocated by the DHCP server.
3) LLA: Configure the camera IP in LLA (Link-Local Address) mode.

- Related Parameters

`GX_IP_CONFIGURE_STATIC_IP`: Configuration the camera IP in a static IP mode.

`GX_IP_CONFIGURE_DHCP`: Configure the camera IP with DHCP, which is the IP allocated by the DHCP server.

`GX_IP_CONFIGURE_LLA`: Configure the camera IP in LLA (Link-Local Address) mode.

`GX_IP_CONFIGURE_DEFAULT`: Configure the camera IP in the default mode. In this mode, the camera will enable all the three configurations available, but still configure the camera in a static IP mode.

- Code sample

```
    GX_STATUS status = GX_STATUS_SUCCESS;

    // This is the MAC sample address. The actual MAC address of the camera
    // can be obtained from the GXGetDeviceIPInfo interface.
    char szMAC[]         = "00-21-49-00-00-00";

    char szIpAddress[]      = "192.168.10.10";
    char szSubnetMask[]     = "255.255.255.0";
    char szDefaultGateway[] = "192.168.10.2";
    char szUserID[]         = "Daheng Imaging";
    GX_IP_CONFIGURE_MODE  emIpConfigureMode  = IP_CONFIGURE_STATIC_IP;

    //This example is described in persistent IP configuration mode, and
    //other IP configuration modes are similar.
    //Sets to static IP configuration mode.
    status = GXGigEIpConfiguration(szMAC, emIpConfigureMode,
                                   szIpAddress, szSubnetMask,
                                   szDefaultGateway, szUserID);
```

- Precautions

➢ Before calling this interface, you must do enumeration operation first, and to perform this operation when the camera is not open.

➢ When you select the GX_IP_CONFIGURE_STATIC_IP and GX_IP_CONFIGURE_DEFAULT parameters to configuration the camera IP, the parameters pointers to the IP address, subnet mask, default gateway etc. are not NULL.

➢ When you select the GX_IP_CONFIGURE_LLA and GX_IP_CONFIGURE_DHCP parameters to

configuration the camera IP, the parameters pointers to the IP address, subnet mask, default gateway etc. can be NULL.

- The maximum length of user-defined name (UserID) is 16 characters. Configuration cameras IP in any way, the user-defined name (UserID) parameter pointers can be NULL.

3.11.2. Force IP

- Terms

Force IP: Use the Force IP mode can temporarily change the camera's IP address, only for the use of the camera, which will restore the original IP when the camera has rebooted.

- Code sample

```
GX_STATUS status = GX_STATUS_SUCCESS;

// This is the MAC sample address. The actual MAC address of the camera
// can be obtained from the GXGetDeviceIPInfo interface.

char szMAC[]         = "00-21-49-00-00-00";

char szIpAddress[]      = "192.168.10.10";
char szSubnetMask[]     = "255.255.255.0";
char szDefaultGateway[] = "192.168.10.2";

// ForceIp
status = GXGigEForceIp(szMAC, szIpAddress, szSubnetMask,
                       szDefaultGateway);
```

- Precautions

Before calling this interface, you must do enumeration operation first, and to perform this operation when the camera is not open.

3.12. Other Functions

3.12.1. Auto Exposure/ Auto Gain Related Function

During the acquisition process, the camera can automatically adjust the gain value and exposure time within a certain range according to the change of the ambient light to maximize the user's desired gray level. This is the automatic gain and automatic exposure function. By default, the camera calculates and adjusts the brightness according to the entire image, making the image brightness in the area of interest the expected value.

The automatic gain and auto exposure functions of the variable area are very flexible, and the user can set the desired value of the image brightness according to different application scenarios. In some applications where the backlight or the local brightness difference of the image is large, the user can delineate the region of interest as needed. The camera will intelligently select the gain and exposure adjustment ratio according to the set value to ensure the best image quality. In addition, the user can also set parameters such as the upper and lower limits of the exposure time adjustment and the upper and lower limits of the gain adjustment.

3.12.1.1. Expectation Gray Value

- Terms

Expectation Gray Value: The expectation of gray is a basic parameter of automatic exposure and automatic gain adjustment, and the final expectation of automatic function adjustment is to achieve the

expectation gray value that the user set.

- Related Parameters
  `GX_INT_GRAY_VALUE`: Expectation gray value.
- Code Sample：

```
// Gets the adjustment range of the expectation gray value.
GX_INT_RANGE grayValueRange;
status = GXGetIntRange(hDevice, GX_INT_GRAY_VALUE, &grayValueRange);
// Sets the gray value to the minimum.
status = GXSetInt(hDevice, GX_INT_GRAY_VALUE, grayValueRange.nMin);
// Sets the gray value to the maximum.
status = GXSetInt(hDevice, GX_INT_GRAY_VALUE, grayValueRange.nMax);
```

- Precautions

3.12.1.2. Light Environment

- Terms

Light Environment: The light environment is the external work environment of the camera, which is divided into daylight, 50Hz AC light source and 60Hz AC light source. The automatic function can adapt better according to the external illumination condition.

- Related Parameters

`GX_ENUM_AA_LIGHT_ENVIRMENT`: Automatic function of light environment, the enumeration value ref: `GX_AA_LIGHT_ENVIRMENT_ENTRY`.

- Code Sample

```
// Uses the GXGetEnumEntryNums and GXGetEnumDescription interfaces to
// query the GX_ENUM_AA_LIGHT_ENVIRMENT types supported by the current
// camera.
// Please refer to the interface descriptions, which are omitted here.
// Selects the type of the gain channel.
status = GXSetEnum(hDevice, GX_ENUM_AA_LIGHT_ENVIRMENT,
                    GX_AA_LIGHT_ENVIRMENT_NATURELIGHT);
//status = GXSetEnum(hDevice, GX_ENUM_AA_LIGHT_ENVIRMENT,
//                   GX_AA_LIGHT_ENVIRMENT_AC60HZ);
//status = GXSetEnum(hDevice, GX_ENUM_AA_LIGHT_ENVIRMENT,
//                   GX_AA_LIGHT_ENVIRMENT_AC60HZ);
```

- Precaution

3.12.1.3. Statistical Area

- Terms

Statistical Area: The camera supports automatic function statistics of variable area, in default, the camera to statistics the whole image to achieve the expectation gray value. The statistical area function allows the user to automatically select a part of the area to execute auto-function automatically, which is more flexible.

- Related Parameters

`GX_INT_AAROI_WIDTH`: Automatically adjust the width of the region of interest.
`GX_INT_AAROI_HEIGHT`: Automatically adjust the height of the region of interest.
`GX_INT_AAROI_OFFSETX`: Automatically adjust the X offset (left offset) of the region of interest.
`GX_INT_AAROI_OFFSETY`: Automatically adjust the Y offset (top offset) of the region of interest.

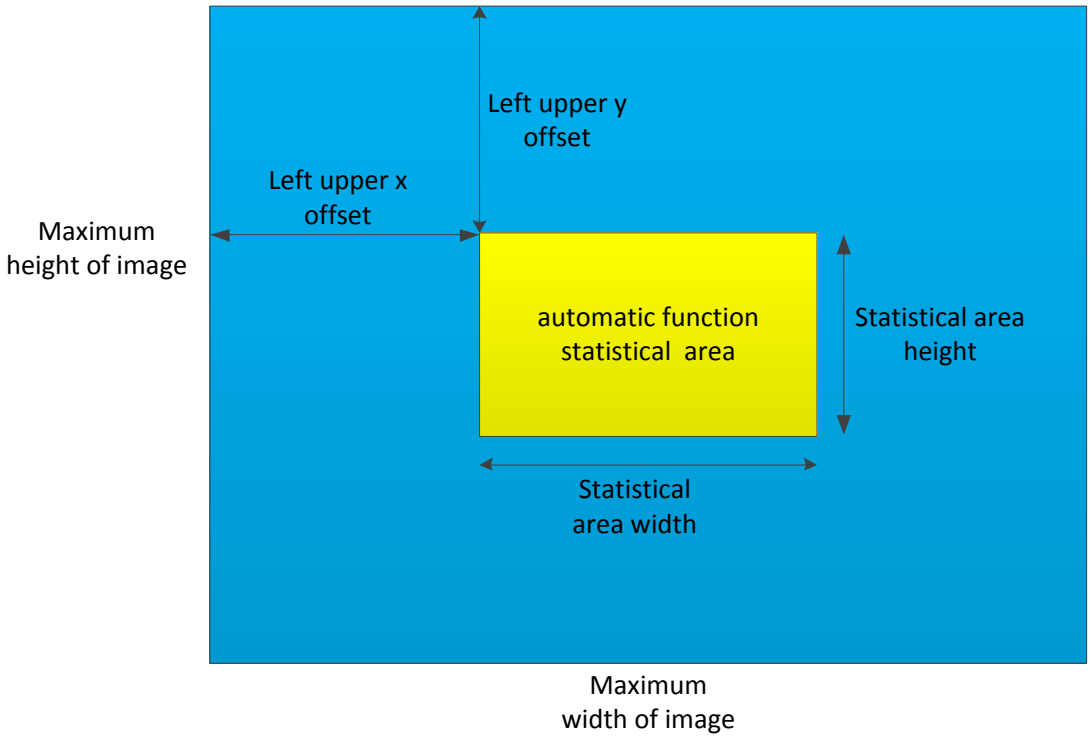

Figure 51: Statistical Area

- Code Sample

```
// Gets the adjustment range.
GX_INT_RANGE stROIWidthRange;
GX_INT_RANGE stROIHeightRange;
GX_INT_RANGE stROIXRange;
GX_INT_RANGE stROIYRange;
status = GXGetIntRange(hDevice, GX_INT_AAROI_WIDTH, &stROIWidthRange);
status = GXGetIntRange(hDevice, GX_INT_AAROI_HEIGHT, &stROIHeightRange);
status = GXGetIntRange(hDevice, GX_INT_AAROI_OFFSETX, &stROIXRange);
status = GXGetIntRange(hDevice, GX_INT_AAROI_OFFSETY, &stROIYRange);

// Sets the statistics area to the whole image.
status = GXSetInt(hDevice, GX_INT_AAROI_WIDTH, stROIWidthRange.nMax);
status = GXSetInt(hDevice, GX_INT_AAROI_HEIGHT, stROIHeightRange.nMax);
status = GXSetInt(hDevice, GX_INT_AAROI_OFFSETX, 0);
status = GXSetInt(hDevice, GX_INT_AAROI_OFFSETY, 0);
```

- Precautions

1) The statistical area only provides a statistical range for automatic exposure and automatic gain calculation, then the camera will calculate to get the adjustment parameters according to the data within the scope.
2) To ensure the effectiveness of the AAROI, the four-attribute range of AAROI can be ref as the following formulas:
   0 <AAROIOffsetX<= Width (Max. Width of the image)-AAROIWidth
   0 <AAROIOffsetY < =Height (Max. Height of the image)-AAROIHeight
   0 <AAROIWidth <= Width (Max. Width of the image)-AAROIOffsetX
   0 <AAROIHeight <= Height (Max. Height of the image)-AAROIOffsetY

3.12.1.4. Auto Adjustment Range

- Terms

1) Auto Exposure Range: When the automatic exposure mode is turned on, the camera can adjust the exposure time dynamically in a range to adapt the current environment, and the user can adjust the range.
2) Auto Gain Range: When the automatic gain mode is turned on, the camera can adjust the gain dynamically in a range to adapt the current environment, and the user can adjust the range.
3) Auto Exposure: A work mode of the camera. In this mode, the camera automatically adjusts the exposure time in a range to achieve the best image quality according to the environment.
4) Auto Gain: A work mode of the camera. In this mode, the camera automatically adjusts the gain in a range to achieve the best image quality according to the environment.

- Related Parameters

`GX_FLOAT_AUTO_GAIN_MIN`：Minimum value of the automatic gain adjustable range.

`GX_FLOAT_AUTO_GAIN_MAX`：Maximum value of the automatic gain adjustable range.

`GX_FLOAT_AUTO_EXPOSURE_TIME_MIN`:Minimum value of the automatic exposure adjustable range.

`GX_FLOAT_AUTO_EXPOSURE_TIME_MAX`：Maximum value of the automatic exposure adjustable range.

- Code Sample:

```
// Gets the adjustment range.
GX_FLOAT_RANGE autoGainMinRange;
GX_FLOAT_RANGE autoGainMaxRange;
GX_FLOAT_RANGE autoExposureMinRange;
GX_FLOAT_RANGE autoExposureMaxRange;
status = GXGetFloatRange(hDevice, GX_FLOAT_AUTO_GAIN_MIN, &autoGainMinRange);
status = GXGetFloatRange(hDevice, GX_FLOAT_AUTO_GAIN_MAX, &autoGainMaxRange);
status = GXGetFloatRange(hDevice,
                         GX_FLOAT_AUTO_EXPOSURE_TIME_MIN,
                         &autoExposureMinRange);
status = GXGetFloatRange(hDevice,
                         GX_FLOAT_AUTO_EXPOSURE_TIME_MAX,
                         &autoExposureMaxRange);
// Sets the boundary value for the adjustment range.
status = GXSetInt(hDevice, GX_FLOAT_AUTO_GAIN_MIN, autoGainMinRange.dMin);
status = GXSetInt(hDevice, GX_FLOAT_AUTO_GAIN_MAX, autoGainMinRange.dMax);
status = GXSetInt(hDevice,
                  GX_FLOAT_AUTO_EXPOSURE_TIME_MIN,
                  autoShutterMinRange.dMin);
status = GXSetInt(hDevice,
                  GX_FLOAT_AUTO_EXPOSURE_TIME_MAX,
                  autoShutterMaxRange.dMax);
```

- Precautions

In order to ensure the correct legality of automatic functional range values, some special processing is done for automatic function tunable:

For GX_FLOAT_AUTO_GAIN_MIN (the minimum value of auto gain), the minimum value itself has the adjusting range, the maximum of the range must not greater than the current value of the maximum auto gain value. Similarly, for GX_FLOAT_AUTO_GAIN_MAX (the maximum value of auto gain), this value also has adjustable range, the minimum of the range must not smaller than the current value of the minimum auto gain value.

3.12.2. Dead Pixel Correction

- Terms

Dead Pixel: There may be very few bad pixels or dead pixels on the sensor surface, so that the actual color cannot be correctly reflected. When for interpolation calculation, these dead pixels will also pollute the surrounding colors. In order to solve this problem, the manufacturer adopts the software method. The image is preprocessed to eliminate dead pixels.

- Related Parameters

`GX_ENUM_DEAD_PIXEL_CORRECT`: Automatic dead pixel correction enable, the enumeration value ref: `GX_DEAD_PIXEL_CORRECT_ENTRY`.

- Code Sample

```
// Enables automatic dead pixel correction algorithm.
    status=GXSetEnum(hDevice, GX_ENUM_DEAD_PIXEL_CORRECT,
                     GX_DEAD_PIXEL_CORRECT_ON);

// Disables automatic dead pixel correction algorithm.
status=GXSetEnum(hDevice, GX_ENUM_DEAD_PIXEL_CORRECT, GX_DEAD_PIXEL_CORRECT_OFF);
```

- Precautions

3.12.3. ADCLevel (DigitalShift)

- Terms

Assuming the current camera is 10 bits or 12 bits output, you need to select 8 bits from the 10 bits or 12 bits to display the image. When the camera output image is not 8 bits, this function can be used to select which 8 bits image to output.

In some devices, the ADCLevel is called digital shift, that is DigitalShift.

If the current camera output 10 bits, there are three conversion level:

Level0: 0~7bit
Level1: 1~8bit
Level2: 2~9bit

If the current camera output 12 bits, there are five conversion level:

Level0: 0~7bit
Level1: 1~8bit
Level2: 2~9bit
Level3: 3~10bit
Level4: 4~11bit

ADCLevel: The lower the level, the brighter the image and the louder the noise; the higher the level, the darker the image and the lower the noise.

DigitalShift: The lower the value, the darker the image and the lower the noise; the larger the value, the brighter the image and the louder the noise.

- Related Parameters

`GX_INT_ADC_LEVEL`：Conversion level.

`GX_INT_DIGITAL_SHIFT`：Digital shift.

- Effect images (If the camera output 10 bits, and the user select 8 bits to output)

Figure 52：Select Level0 (0~7bit)

Figure 53: Select Level1 (1~8bit)

Figure 54: Select Level2 (2~9bit)

- Code Sample

```
// Gets the range of the ADC conversion level (The sample also applies
// to the bit output GX_INT_DIGITAL_SHIFT).
GX_INT_RANGE ADCLevelRange;
status = GXGetIntRange(hDevice, GX_INT_ADC_LEVEL, &ADCLevelRange);

//Sets the ADC conversion level to the minimum.
status = GXSetInt(hDevice, GX_INT_ADC_LEVEL, ADCLevelRange.nMin);
//Sets the ADC conversion level to the maximum.
status = GXSetInt(hDevice, GX_INT_ADC_LEVEL, ADCLevelRange.nMax);
```

- Precautions

If the current camera's output is 10 bits, and the user's camera setting is 10 bits, then there is no need to select 8 bits. At this time, the ADCLevel function is disable. Only when the camera's output is 10 bits, and the user's camera need to output 8 bits, you will need to select which 8 bits.

3.12.4. Blanking Control

- Terms

1) Horizontal blanking (line blanking): In the scan process of converting light signal to electrical signals, scanning always starts from the top left corner of the image and horizontal scan forward, scanning point also move down at a slower rate. When the scanning point reach the right side of the image, the scanning point quickly returned to the left, and to restart the second line scan below the starting line of the first line. The return processes of the lines between is referred to as horizontal blanking (HBlank).
2) Vertical blanking (field blanking): A complete image scan signal consisting of a sequence of line signals separated by horizontal blanking intervals, referred to as a frame. After scanning a frame, the scanning point is returned from the lower right corner of the image to the upper left corner of the image to start a new frame scanning. This time interval is called vertical blanking, also called field blanking (VBlank).

- Related Parameters

`GX_INT_H_BLANKING`：Horizontal blanking (line blanking).

`GX_INT_V_BLANKING`：Vertical blanking (field blanking).

- Code Sample

```
// Gets the range of the horizontal blanking.
GX_INT_RANGE HBRange;
status = GXGetIntRange(hDevice, GX_INT_H_BLANKING, &HBRange);
// Sets the value of the horizontal blanking to the minimum.
status = GXSetInt(hDevice, GX_INT_H_BLANKING, HBRange.nMin);
// Sets the value of the horizontal blanking to the maximum.
status = GXSetInt(hDevice, GX_INT_H_BLANKING, HBRange.nMax);

// Gets the range of the vertical blanking.
GX_INT_RANGE VBRange;
status = GXGetIntRange(hDevice, GX_INT_V_BLANKING, &VBRange);
// Sets the value of the vertical blanking to the minimum.
status = GXSetInt(hDevice, GX_INT_V_BLANKING, VBRange.nMin);
// Sets the value of the vertical blanking to the maximum.
status = GXSetInt(hDevice, GX_INT_V_BLANKING, VBRange.nMax);
```

- Precautions

Increasing the horizontal or vertical blanking will reduce the frame rate, conversely, it increases the frame rate.

3.12.5. The User Data Encryption Area

- Terms

The user data encryption area is designed to protect the user's self-owned intellectual property rights. Users can define it by themselves. When the user data encryption area can be accessed, the ordinary parameters are used, if the user data encryption area is not accessible, the user must use a secret key to open it. The users can be more closely tied to their software through the data encryption area, then to improve the difficulty of decryption and protect their self-owned intellectual property.

The user data encryption area can be accessed when the camera leaves the factory.

- Related Parameters

`GX_STRING_USER_PASSWORD`: The user password of the encryption area.

`GX_STRING_VERIFY_PASSWORD`: The verify code of the data encryption area.

`GX_BUFFER_USER_DATA`: The user data of the data encryption area.

- Code Sample

```
Please contact technical support for specific use (support@daheng-imaging.com).
```

- Precautions

Using the encryption area can improve some difficulty of decryption only, can not ensure that the program don't be cracked, so Daheng company does not bear any legal liability for the loss of the safe.

4. Local Device Control

4.1. Related Parameters

`GX_DEV_INT_COMMAND_TIMEOUT`：The command is timeout.

`GX_DEV_INT_COMMAND_RETRY_COUNT`：Command retry number.

4.2. Code Sample

```
// Sets the number of times the command is retried.
uint64_t nCommandRetryCount = 0;
emStatus = GXSetInt(hDevice, GX_DEV_INT_COMMAND_RETRY_COUNT, nCommandRetryCount);
```

4.3. Precautions

5. Flow Layer Control

5.1. Statistical Parameters

5.1.1. Related Parameters

`GX_DS_INT_ANNOUNCED_BUFFER_COUNT`： Number of announced (known) buffers.

`GX_DS_INT_DELIVERED_FRAME_COUNT`：Total number of delivered frames since last acquisition start (including incomplete frames).

`GX_DS_INT_LOST_FRAME_COUNT`：Number of lost frames due to the buffer insufficient.

`GX_DS_INT_INCOMPLETE_FRAME_COUNT`：Number of incomplete frames due to lost packets.

In addition, the GigE Vision camera has some special statistical parameters, including:

`GX_DS_INT_DELIVERED_PACKET_COUNT`：Number of received packets since last acquisition start (excluding exception packages).

`GX_DS_INT_RESEND_PACKET_COUNT`：Number of resend packets since last acquisition start.

`GX_DS_INT_RESCUED_PACKED_COUNT`：Number of resend successfully packets since last acquisition start.

`GX_DS_INT_RESEND_COMMAND_COUNT`：Number of resend commands since last acquisition start.

`GX_DS_INT_UNEXPECTED_PACKED_COUNT`：Number of unexpected packets.

`GX_DS_INT_MISSING_BLOCKID_COUNT`：Number of missing BlockID.

5.1.2. Precautions

During the acquisition process, the user can read these parameters at any time to observe the current driver's acquisition status.

5.1.3. Code Sample

```
// Reads the number of frames received.
uint64_tnGetFrameCount = 0;
emStatus = GXGetInt(hDevice, GX_DS_INT_DELIVERED_FRAME_COUNT, &nGetFrameCount);
```

5.2. Control Parameters

5.2.1. Related Parameters

- GigE Vision camera related parameters:

`GX_DS_INT_MAX_PACKET_COUNT_IN_ONE_BLOCK`：The maximum resend packet count of a block, that is the maximum resend packet of a frame, if a frame image has send too much resend packets more than the value, then it is not to send the resend command, but directly determine the frame is an incomplete frame. If the value is 0, it means the function is closed.

`GX_DS_INT_MAX_PACKET_COUNT_IN_ONE_COMMAND`：The maximum packet count of one resend process. The packet is judged lost in the receiving process. If the count is too much greater than the value, then it is not to send the resend command, but directly determine the frame is an incomplete frame. If the value is 0, it means the function is closed.

`GX_DS_INT_RESEND_TIMEOUT`：The timeout time of resent packets. After the retransmission command is sent, the time to wait for the resend packet arrives. If the resend packet does not arrive in this time, it may be an incomplete frame.

`GX_DS_INT_MAX_WAIT_PACKET_COUNT`：The maximum count of waiting packets. If a frame has not be transferred completed, the next frame is coming, and the next frame image has received N packets, the previous frame still not transferred completed, then stop the previous frame, and make it as an incomplete frame. If the value is 0, it means the function is closed.

`GX_DS_ENUM_RESEND_MODE`：The mode of resend, it controls the resend switch. You can reference the define of GX_DS_RESEND_MODE_ENTRY in GxIAPI.h.

`GX_DS_INT_BLOCK_TIMEOUT`：The timeout time of data block. The total time of frame receives. If the receive time is longer than the value, you can take the frame as an incomplete one.

`GX_DS_INT_MAX_NUM_QUEUE_BUFFER`：The maximum buffer count of the acquisition queue, it can adjust the number of the buffers which involved in data receiving underlying. When many high-resolution cameras acquisition simultaneously fail, you can reduce the value of this parameter appropriately, so that more cameras can acquie at the same time. If the value is too small in a high-resolution acquisition mode, some frames may be lost. Please contact technical support for specific usage.

- USB3 Vision camera related parameters:

`GX_DS_INT_STREAM_TRANSFER_SIZE`: The size of the transmission data block. For USB3 Vision series camera, it means the size of every transmission data block.

`GX_DS_INT_STREAM_TRANSFER_NUMBER_URB`: The number of transmission data block. This parameter limits the data block number of single camera mapping to the system kernel actually. The default value is 64, if the value is too small, it will affect the transmission efficiency; However, the number of total data block which can be mapping to the system kernel is limited. When multiple cameras are used together, you can reduce the value appropriately, and increase the number of acquisition device at the same time.

5.2.2. Precautions

The above control parameters can only be modified after stopping acquisition, and the control parameters are not allowed to be modified in the acquisition process.

5.2.3. Code Sample

```
// Sets the timeout time for resend.
uint64_tnResendTimeout = 128;
emStatus = GXSetInt(hDevice, GX_DS_INT_RESEND_TIMEOUT, nResendTimeout);
```

6. Functions Affected by Camera Model

Table 1: MER-040-60Ux

Camera Model: MER-040-60Ux	
1	[UC/UM] Adjust the value of GX_INT_WIDTH will update the current value and minimum value of the GX_INT_H_BLANKING automatically. Formulas are as follows: HBlanking_min=MAX(-21,0x236-Width+1); HBlanking_cur=MAX(HBlanking_cur, HBlanking_min);
2	[UM] Both GX_INT_BINNING_HORIZONTAL and GX_INT_BINNING_VERTICAL have interconnected relationships.

Table 2: MER-500-7Ux

Camera Model: MER-500-7Ux	
1	When adjusting the Binning function, the horizontal and vertical value of Decimation must be 0. When adjusting the Decimation function, the horizontal and vertical value of Binning must be 0.
2	The horizontal parameter value of Binning must not be 2, it may be 0, 1, 3.

7. GxIAPI Library Definitions

7.1. Type

7.1.1. Data Type

Name	Description
Int8_t	8-bit signed integer
Int16_t	16-bit signed integer
Int32_t	32-bit signed integer
Int64_t	64-bit signed integer
uint8_t	8-bit unsigned integer
uint16_t	16-bit unsigned integer
uint32_t	32-bit unsigned integer
uint64_t	64-bit unsigned integer

7.1.2. Handle Type

Name	Description
GX_DEV_HANDLE	Device handle. It can be obtained through the GXOpenDevice interface and can be used to achieve control and acquisition
GX_EVENT_CALLBACK_HANDLE	Device callback handle. It can be used to register callback functions for related events, such as a device offline callback function
GX_FEATURE_CALLBACK_HANDLE	Device attributes update callback handle. It can be used to register device attribute and update callback function

7.1.3. Callback Function Type

Name	Description
typedef void (GX_STDC * GXCaptureCallBack) (GX_FRAME_CALLBACK_PARAM *pFrameData)	Capture callback function type
typedef void (GX_STDC *GXDeviceOfflineCallBack) (void *pUserParam)	Device offline callback function type
typedef void (GX_STDC *GXFeatureCallBack) (GX_FEATURE_ID_CMD nFeatureID, void *pUserParam)	Device attribute update callback function type

7.2. Constant

7.2.1. GX_STATUS_LIST

```
typedef enum GX_STATUS_LIST
{
    GX_STATUS_SUCCESS                = 0,
    GX_STATUS_ERROR                  = -1,
    GX_STATUS_NOT_FOUND_TL           = -2,
    GX_STATUS_NOT_FOUND_DEVICE = -3,
    GX_STATUS_OFFLINE                = -4,
    GX_STATUS_INVALID_PARAMETER = -5,
    GX_STATUS_INVALID_HANDLE         = -6,
    GX_STATUS_INVALID_CALL           = -7,
    GX_STATUS_INVALID_ACCESS         = -8,
    GX_STATUS_NEED_MORE_BUFFER= -9,
    GX_STATUS_ERROR_TYPE             = -10,
    GX_STATUS_OUT_OF_RANGE           = -11,
    GX_STATUS_NOT_IMPLEMENTED    = -12,
    GX_STATUS_NOT_INIT_API           = -13,
    GX_STATUS_TIMEOUT                = -14,
}GX_STATUS_LIST;
```

Name	Description
GX_STATUS_SUCCESS	Success
GX_STATUS_ERROR	There is an unspecified internal error that is not expected to occur
GX_STATUS_NOT_FOUND_TL	The TL library cannot be found
GX_STATUS_NOT_FOUND_DEVICE	The device is not found
GX_STATUS_OFFLINE	The current device is in an offline status
GX_STATUS_INVALID_PARAMETER	Invalid parameter. Generally, the pointer is NULL or the input IP and other parameter formats are invalid
GX_STATUS_INVALID_HANDLE	Invalid handle
GX_STATUS_INVALID_CALL	The interface is invalid, which refers to software interface logic error
GX_STATUS_INVALID_ACCESS	The function is currently inaccessible or the device access mode is incorrect
GX_STATUS_NEED_MORE_BUFFER	The user request buffer is insufficient: the user input buffersize during the read operation is less than the actual need
GX_STATUS_ERROR_TYPE	The type of FeatureID used by the user is incorrect, such as an integer interface using a floating-point function code
GX_STATUS_OUT_OF_RANGE	The value written by the user is crossed
GX_STATUS_NOT_IMPLEMENTED	This function is not currently supported
GX_STATUS_NOT_INIT_API	There is no call to initialize the interface
GX_STATUS_TIMEOUT	Timeout error

7.2.2. GX_FRAME_STATUS

```
enum GX_FRAME_STATUS_LIST
{
    GX_FRAME_STATUS_SUCCESS          = 0,
    GX_FRAME_STATUS_INCOMPLETE       = -1,
};
typedef int32_t GX_FRAME_STATUS;
```

Name	Description
GX_FRAME_STATUS_SUCCESS	Normal frame
GX_FRAME_STATUS_INCOMPLETE	Incomplete frame

7.2.3. GX_DEVICE_CLASS

```
enum GX_DEVICE_CLASS_LIST
{
    GX_DEVICE_CLASS_UNKNOWN          = 0,
    GX_DEVICE_CLASS_USB2             = 1,
    GX_DEVICE_CLASS_GEV              = 2,
    GX_DEVICE_CLASS_U3V              = 3,
};
typedef    int32_t    GX_DEVICE_CLASS;
```

Name	Description
GX_DEVICE_CLASS_UNKNOWN	Unknown device type
GX_DEVICE_CLASS_USB2	USB2.0 Vision device
GX_DEVICE_CLASS_GEV	Gige Vision device
GX_DEVICE_CLASS_U3V	USB3 Vision device

7.2.4. GX_FEATURE_TYPE

```
typedef enum GX_FEATURE_TYPE
{
    GX_FEATURE_INT                   =0x10000000,
    GX_FEATURE_FLOAT                 =0X20000000,
    GX_FEATURE_ENUM                  =0x30000000,
    GX_FEATURE_BOOL                  =0x40000000,
    GX_FEATURE_STRING                =0x50000000,
    GX_FEATURE_BUFFER                =0x60000000,
    GX_FEATURE_COMMAND               =0x70000000,
} GX_FEATURE_TYPE;
```

Name	Description
GX_FEATURE_INT	Integer type
GX_FEATURE_FLOAT	Floating point type
GX_FEATURE_ENUM	Enum type

GX_FEATURE_BOOL	Boolean type
GX_FEATURE_STRING	String type
GX_FEATURE_BUFFER	Block data type
GX_FEATURE_COMMAND	Command type

7.2.5. GX_FEATURE_LEVEL

```
typedef enum GX_FEATURE_LEVEL
{
    GX_FEATURE_LEVEL_REMOTE_DEV     =0x00000000,
    GX_FEATURE_LEVEL _TL            =0X01000000,
    GX_FEATURE_LEVEL _IF            =0x02000000,
    GX_FEATURE_LEVEL _DEV           =0x03000000,
    GX_FEATURE_LEVEL _DS            =0x04000000,
} GX_FEATURE_LEVEL;
```

Name	Description
GX_FEATURE_LEVEL_REMOTE_DEV	Remote device layer
GX_FEATURE_LEVEL_TL	System layer
GX_FEATURE_LEVEL_IF	Interface layer
GX_FEATURE_LEVEL_DEV	Device layer
GX_FEATURE_LEVEL_DS	DataStream layer

7.2.6. GX_ACCESS_MODE

```
typedef enum GX_ACCESS_MODE
{
    GX_ACCESS_READONLY      =2,
    GX_ACCESS_CONTROL       =3,
    GX_ACCESS_EXCLUSIVE     =4,
} GX_ACCESS_MODE;
typedef int32_t GX_ACCESS_MODE_CMD;
```

Name	Description
GX_ACCESS_READONLY	Open the device in read-only mode
GX_ACCESS_CONTROL	Open the device in controlled mode
GX_ACCESS_EXCLUSIVE	Open the device in exclusive mode

7.2.7. GX_ACCESS_STATUS

```
typedef enum GX_ACCESS_STATUS
{
    GX_ACCESS_STATUS_UNKNOWN    = 0,
    GX_ACCESS_STATUS_READWRITE  = 1,
```

```
    GX_ACCESS_STATUS_READONLY    = 2,
    GX_ACCESS_STATUS_NOACCESS    = 3,
} GX_ACCESS_STATUS;
typedef int32_t GX_ACCESS_STATUS_CMD;
```

Name	Description
GX_ACCESS_STATUS_UNKNOWN	The device's current status is unknown
GX_ACCESS_STATUS_READWRITE	The device currently supports reading and writing
GX_ACCESS_STATUS_READONLY	The device currently only supports reading
GX_ACCESS_STATUS_NOACCESS	The device currently does neither support reading nor support writing

7.2.8. GX_OPEN_MODE

```
typedef enum GX_OPEN_MODE
{
    GX_OPEN_SN          =0,
    GX_OPEN_IP          =1,
    GX_OPEN_MAC         =2,
    GX_OPEN_INDEX       =3,
    GX_OPEN_USERID      =4,
} GX_OPEN_MODE;
typedef int32_t GX_OPEN_MODE_CMD;
```

Name	Description
GX_OPEN_SN	Opens the device via a serial number
GX_OPEN_IP	Opens the device via an IP address
GX_OPEN_MAC	Opens the device via a MAC address
GX_OPEN_INDEX	Opens the device via a serial number (Start from 1, such as 1, 2, 3, 4…)
GX_OPEN_USERID	Opens the device via user defined ID

7.2.9. GX_IP_CONFIGURE_MODE_LIST

```
typedef enum GX_IP_CONFIGURE_MODE_LIST
{
    GX_IP_CONFIGURE_DHCP ,
    GX_IP_CONFIGURE_LLA ,
    GX_IP_CONFIGURE_STATIC_IP ,
    GX_IP_CONFIGURE_DEFAULT
}GX_IP_CONFIGURE_MODE_LIST;
typedef int32_t GX_IP_CONFIGURE_MODE;
```

Name	Description
GX_IP_CONFIGURE_DHCP	Enable the DHCP mode to allocate the IP address by the

	DHCP server
GX_IP_CONFIGURE_LLA	Enable the LLA mode to allocate the IP addresses
GX_IP_CONFIGURE_STATIC_IP	Enable the static IP mode to configure the IP address
GX_IP_CONFIGURE_DEFAULT	Enable the default mode to configure the IP address

7.3. Structure

7.3.1. GX_DEVICE_BASE_INFO

Related interface：*GXGetAllDeviceBaseInfo*

This structure represents the basic information of the device, whether it is a USB camera or a GigE camera.

```
typedef struct GX_DEVICE_BASE_INFO
{
    char szVendorName[GX_INFO_LENGTH_32_BYTE];
    char szModelName[GX_INFO_LENGTH_32_BYTE];
    char szSN[GX_INFO_LENGTH_32_BYTE];
    char szDisplayName[GX_INFO_LENGTH_128_BYTE + 4];
    char szDeviceID[GX_INFO_LENGTH_64_BYTE + 4];
    char szUserID[GX_INFO_LENGTH_64_BYTE + 4];
    GX_ACCESS_STATUS_CMD   accessStatus;
    GX_DEVICE_CLASS    deviceClass;
    char reserved[300];
} GX_DEVICE_BASE_INFO;
```

Name	Description
szVendorName	32 bytes, vendor name
szModelName	32 bytes, model name
szSN	32 bytes, device serial number
szDisplayName	128+4 bytes, device display name
szUserID	64+4 bytes，user-defined name
szDeviceID	64+4 bytes, the unique identifier of the device
accessStatus	4 bytes, access status that is currently supported by the device. Refer to GX_ACCESS_STATUS
deviceClass	4 bytes，device type, such as USB2.0, GEV
reserved	300 bytes， reserved

7.3.2. GX_DEVICE_IP_INFO

Related interface: *GXGetDeviceIPInfo*

This structure represents some of the GigE cameras property descriptions.

```
typedef struct GX_DEVICE_IP_INFO
{
    char szDeviceID[GX_INFO_LENGTH_64_BYTE + 4];
    char szMAC[GX_INFO_LENGTH_32_BYTE];
    char szIP[GX_INFO_LENGTH_32_BYTE];
    char szSubNetMask[GX_INFO_LENGTH_32_BYTE];
    char szGateWay[GX_INFO_LENGTH_32_BYTE];
    char szNICMAC[GX_INFO_LENGTH_32_BYTE];
    char szNICIP[GX_INFO_LENGTH_32_BYTE];
    char szNICSubNetMask[GX_INFO_LENGTH_32_BYTE];
    char szNICGateWay[GX_INFO_LENGTH_32_BYTE];
    char szNICDescription[GX_INFO_LENGTH_128_BYTE + 4];
    char reserved[512];
} GX_DEVICE_IP_INFO;
```

Name	Description
szDeviceID	64+4 bytes，the unique identifier of the device
szMAC	32 bytes，MAC address
szIP	32 bytes，IP address
szSubNetMask	32 bytes，subnet mask
szGateWay	32 bytes，gateway
szNICMAC	32 bytes，the MAC address of the corresponding NIC (Network Interface Card)
szNICIP	32 bytes，the IP address of the corresponding NIC
szNICSubNetMask	32 bytes，the subnet mask of the corresponding NIC
szNICGateWay	32 bytes，the gateway of the corresponding NIC
szNICDescription	128+4 bytes，the description of the corresponding NIC
reserved	512 bytes，reserved

7.3.3. GX_OPEN_PARAM

Related interface: *GXOpenDevice*

This structure is designed for the open device interface.

```
typedef struct GX_OPEN_PARAM
{
    char                *pszContent;
    GX_OPEN_MODE_CMD    openMode;
    GX_ACCESS_MODE_CMD  accessMode;
} GX_OPEN_PARAM;
```

Name	Description
pszContent	Standard C string that is decided by openMode. It could be an IP address, a camera serial number, and so on
openMode	Device open mode. The device can be open via the SN, IP, MAC, etc. Please refer to GX_OPEN_MODE
accessMode	Device access mode, such as read-only, control, exclusive, etc. Please refer to GX_ACCESS_MODE

7.3.4. GX_FRAME_CALLBACK_PARAM

Related interface：*GXRegisterCaptureCallback*

This is the formal parameter type of the callback function, and the formal parameter type of the callback function that the user writes must be of this type.

```
typedef struct GX_FRAME_CALLBACK_PARAM
{
    void*               pUserParam;
    GX_FRAME_STATUS     status;
    const void*         pImgBuf;
    int32_t             nImgSize;
    int32_t             nWidth;
    int32_t             nHeight;
    int32_t             nPixelFormat;
    uint64_t            nFrameID;
    uint64_t            nTimestamp;
    int32_t             reserved[1];
} GX_FRAME_CALLBACK_PARAM;
```

Name	Description
pUserParam	User's private data pointer
status	The image state returned by the callback function. Please refer to GX_FRAME_STATUS
pImgBuf	The image data address (After the frame information is enabled, the pImgBuf contains image data and frame information data)
nImgSize	Data size, in bytes (After the frame information is enabled, nImgsize is the sum of the size of the image data and the size of the frame information)
nWidth	Image width

nHeight	Image height
nPixelFormat	PixelFormat of image
nFrameID	Frame identification of image
nTimestamp	Timestamp of image
reserved	4 bytes, reserved

7.3.5. GX_FRAME_DATA

Related interface：*GXGetImage*

```
typedef struct GX_FRAME_DATA
{
    GX_FRAME_STATUS     nStatus;
    void*               pImgBuf;
    int32_t             nWidth;
    int32_t             nHeight;
    int32_t             nPixelFormat;
    int32_t             nImgSize;
    uint64_t            nFrameID;
    uint64_t            nTimestamp;
    int32_t             nOffsetX;
    int32_t             nOffsetY;
    int32_t             reserved[1];
}GX_FRAME_DATA;
```

Name	Description
nStatus	The state of the acquired image. Please refer to *GX_FRAME_STATUS*
pImgBuf	The image data address (After the frame information is enabled, the pImgBuf contains image data and frame information data)
nWidth	Image width
nHeight	Image height
nPixelFormat	Pixel format of image
nImgSize	Data size (After the frame information is enabled, nImgsize is the sum of the size of the image data and the size of the frame information)
nFrameID	Frame identification of image
nTimestamp	Timestamp of image
nOffsetX	X-direction offset of the image
nOffsetY	Y-direction offset of the image
reserved	4 bytes，reserved

7.3.6. GX_FRAME_BUFFER

Related interface：*GXDQBuf*、*GXQBuf*、*GXDQAllBufs*

```
typedef struct GX_FRAME_BUFFER
{
    GX_FRAME_STATUS     nStatus;
    void*               pImgBuf;
    int32_t             nWidth;
    int32_t             nHeight;
    int32_t             nPixelFormat;
    int32_t             nImgSize;
    uint64_t            nFrameID;
    uint64_t            nTimestamp;
    uint64_t            nBufID;
    int32_t             nOffsetX;
    int32_t             nOffsetY;
    int32_t             reserved[16];
}GX_FRAME_BUFFER;
typedef GX_FRAME_BUFFER* PGX_FRAME_BUFFER;
```

Name	Description
nStatus	The state of the acquired image. Please refer to *GX_FRAME_STATUS*
pImgBuf	The image data pointer (After the frame information is enabled, the pImgBuf contains image data and frame information data)
nWidth	Image width
nHeight	Image height
nPixelFormat	Pixel format of image
nImgSize	Data size, in bytes (After the frame information is enabled, nImgsize is the sum of the size of the image data and the size of the frame information)
nFrameID	Frame identification of image
nTimestamp	Timestamp of image
nBufID	BufID
nOffsetX	X-direction offset of the image
nOffsetY	Y-direction offset of the image
reserved	64 bytes，reserved

7.3.7. GX_INT_RANGE

Related interface：*GXGetIntRange*

The interface describes the maximum value, minimum value and step length of the integer type.

```
typedef struct GX_INT_RANGE
{
    int64_t nMin;
    int64_t nMax;
    int64_t nInc;
    int32_t reserved[8];
}GX_INT_RANGE;
```

Name	Description
nMin	Minimum value
nMax	Maximum value
nInc	Step size
reserved	32 bytes，reserved

7.3.8. GX_FLOAT_RANGE

Related interface：*GXGetFloatRange*

The interface describes the maximum value, minimum value, step length and unit of the float-point type.

```
typedef struct GX_FLOAT_RANGE
{
    double   dMin;
    double   dMax;
    double   dInc;
    char     szUnit[GX_INFO_LENGTH_8_BYTE];
    bool     bIncIsValid;
    int8_t   reserved[31];
}GX_FLOAT_RANGE;
```

Name	Description
dMin	Minimum value
dMax	Maximum value
dInc	Step size
szUnit	Unit. 8 bytes
bIncIsValid	1 byte，indicates whether the step size is supported
reserved	31 bytes，reserved

7.3.9. GX_ENUM_DESCRIPTION

Related interface：*GXGetEnumDescription*

The interface describes the value and description information of all enumerated items.

```
typedef struct GX_ENUM_DESCRIPTION
{
    int64_t   nValue;
    char      szSymbolic[GX_INFO_LENGTH_64_BYTE];
    int32_t   reserved[8];
}GX_ENUM_DESCRIPTION;
```

Name	Description
nValue	The value of the enumeration item
szSymbolic	64 bytes，the character description information of the enumeration item
reserved	32 bytes，reserved

7.4. Interfaces

7.4.1. GXGetLibVersion (Linux only)

Declaration:

GX_EXTC const char* GX_STDC GXGetLibVersion()

Descriptions:

Gets the library version number.

Formal parameter:

Not have.

Return value:

Library version number of string type.

Code sample:

```
#include "GxIAPI.h"

int main(int argc, char* argv[])
{
// Gets the library version number.
    const char *pLibVersion = GXGetLibVersion();

    return 0;
}
```

7.4.2. GXInitLib

Declaration:

GX_API GXInitLib()

Descriptions:

Initialize the device library for some resource application operations. This interface must be called before using the GxIAPI to interact with the camera, and the *GXCloseLib* must be called to release all the resources when the GxIAPI is stopped for all control of the device.

Formal parameter:

Not have.

Return value:

GX_STATUS_SUCCESS	The operation is successful, no error occurs.
GX_STATUS_NOT_FOUND_TL	Can not found the library.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

Before calling the other interfaces (except *GXCloseLib*/ *GXGetLastError*), you must to call the *GXInitLib* interface for initialization first, otherwise the error GX_STATUS_NOT_INIT_API will return.

Code Sample:

```
#include "GxIAPI.h"

int main(int argc, char* argv[])
{
    GX_STATUS status = GX_STATUS_SUCCESS;

    // Calls GXInitLib () at the start location to initialize and apply for
    // resources.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    // Uses GxIAPI.
    //...

    // Calls GXCLoseLib() at the end of the program to release the resource.
    status = GXCloseLib();

    return 0;
}
```

7.4.3. GXCloseLib

Declaration:

GX_API GXCloseLib()

Descriptions:

Close the device library to release resources. You must to call this interface to release resources when the GxIAPI stopped all the controls of the device. Corresponding to the *GXInitLib*.

Formal parameter:

Not have.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code Sample:

```
#include "GxIAPI.h"

int main(int argc, char* argv[])
{
    GX_STATUS status = GX_STATUS_SUCCESS;

    // Calls GXInitLib() at the start location to initialize and apply for
    // resources.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    // Uses GxIAPI.
    //...

    // Calls GXCLoseLib() at the end of the program to release the resource.
    status = GXCloseLib();

    return 0;
}
```

7.4.4. GXGetLastError

Declaration:

```
GX_API GXGetLastError (GX_STATUS *pErrorCode,
                       char *pszErrText,
                       size_t *pnSize)
```

Descriptions:

To get the latest error descriptions information of the program.

Formal parameter:

[out]*pErrorCode*	Return the last error code. You could set the parameter to NULL if you don't need this value.
[out]*pszErrText*	Return the address of the buffer allocated for error information.
[in,out]*pnSize*	The address size of the buffer allocated for error information. Unit: byte.

If *pszErrText* is NULL:

[out]*pnSize*	Return the actual required buffer size.

If *pszErrText* is not NULL:

[in]*pnSize*	It is the actual allocated buffer size.
[out]*pnSize*	Return the actual allocated buffer size.

Return value:

`GX_STATUS_SUCCESS`	The operation is successful and no error occurs.
`GX_STATUS_INVALID_PARAMETER`	The pointer that the user input is NULL.
`GX_STATUS_NEED_MORE_BUFFER`	The buffer that the user filled is too small.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
GX_STATUS errCode = GX_STATUS_SUCCESS;
char *pszTemp = NULL;
size_t nSize = 0;
// First, the user passed the NULL pointer to get the actual size, followed
// by the request buffer to obtain the description information.
GXGetLastError(&errCode, NULL, &nSize);
 pszTemp = new char[nSize];
status = GXGetLastError(&errCode, pszTemp, &nSize);
delete pszTemp;
```

7.4.5. GXUpdateDeviceList

Declaration:

GX_API GXUpdateDeviceList (uint32_t* punNumDevices,
uint32_t nTimeOut)

Descriptions:

Enumerating currently all available devices in subnet and gets the number of devices.

Formal parameter:

[out]*punNumDevices*	The address pointer used to return the number of devices, and the pointer can not be NULL.
[in]*nTimeOut*	The timeout time of enumeration (unit: ms). If the device is successfully enumerated within the specified timeout time, the value returns immediately. If the device is not enumerated within the specified timeout time, then it waits until the specified timeout time is over and then it returns.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
uint32_t nDeviceNum = 0;

// Enumerates to get the number of devices. The timeout time is decided by
// the user's environment, and can be set by the user, 1000ms is an example.
status = GXUpdateDeviceList(&nDeviceNum, 1000);
```

7.4.6. GXUpdateAllDeviceList

Declaration:

GX_API GXUpdateAllDeviceList (uint32_t* punNumDevices,
uint32_t nTimeOut)

Descriptions:

Enumerating currently all available devices in entire network and gets the number of devices.

Formal parameter:

[out]*punNumDevices*	The address pointer used to return the number of devices, and the pointer can not be NULL.
[in]*nTimeOut*	The timeout time of enumeration (unit: ms). If the device is successfully enumerated within the specified timeout time, the value returns immediately. If the device is not enumerated within the specified timeout time, then it waits until the specified timeout time is over and then it returns.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
uint32_t nDeviceNum = 0;

// The number of enumerated devices. The timeout time is limited by the user's
// use environment, and can be set by the user, not limited to 1000ms.
status = GXUpdateAllDeviceList(&nDeviceNum, 1000);
```

7.4.7. GXGetAllDeviceBaseInfo

Declaration:

GX_API GXGetAllDeviceBaseInfo (GX_DEVICE_BASE_INFO* pDeviceInfo,
size_t* pnBufferSize)

Descriptions:

To get the basic information of all devices.

Formal parameter:

[out]*pDeviceInfo* The structure pointer of the device information.

[in,out]*pnBufferSize* The buffer size of device information structure, unit: byte.

If *pDeviceInfo* is NULL:

[out]*pnBufferSize* Return the actual size of the device information.

If *pDeviceInfo* is not NULL:

[in]*pnBufferSize* The size of the buffer that the user allocated.
[out]*pnBufferSize* Return the actual allocated buffer size.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

You should call the *GxUpdateDeviceList()* interface for an enumeration before calling the function to get the device information. Otherwise, the device information that the user gets is inconsistent with the device that is currently connected.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
uint32_t nDeviceNum = 0;
status = GXUpdateDeviceList(&nDeviceNum, 1000);
if (status == GX_STATUS_SUCCESS && nDeviceNum > 0)
{
   GX_DEVICE_BASE_INFO *pBaseinfo = new GX_DEVICE_BASE_INFO[nDeviceNum];
   uint32_t nSize = nDeviceNum * sizeof(GX_DEVICE_BASE_INFO);

   // Gets the basic information of all devices.
   status = GXGetAllDeviceBaseInfo(pBaseinfo, &nSize);
   delete []pBaseinfo;
}
```

7.4.8. GXGetDeviceIPInfo

Declaration:

GX_API GXGetDeviceIPInfo(uint32_t nIndex, GX_DEVICE_IP_INFO* pstDeviceIPInfo)

Descriptions:

To get the network information of all devices.

Formal parameter:

[in]*nIndex*	The serial number of the device.
[out]*pstDeviceIPInfo*	The structure pointer of the device information.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_PARAMETER	The index that the user input is cross the border.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

You should call the *GxUpdateDeviceList()* interface for an enumeration before calling the function to get the device information. Otherwise, the device information that the user gets is inconsistent with the device that is currently connected.

Code Sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
uint32_t nDeviceNum = 0;
status = GXUpdateDeviceList(&nDeviceNum, 1000);
if (status == GX_STATUS_SUCCESS && nDeviceNum > 0)
{
    GX_DEVICE_IP_INFO stIPInfo;

    // Gets the network information of the first device.
    status = GXGetDeviceIPInfo(1, &stIPInfo);
}
```

7.4.9. GXOpenDeviceByIndex

Declaration:

GX_API GXOpenDeviceByIndex (uint32_t nDeviceIndex, GX_DEV_HANDLE* phDevice)

Descriptions:

Open the device by index, starting from 1.

Formal parameter:

[in]*nDeviceIndex*	The index of the device starts from 1, for example: 1, 2, 3, 4...
[out]*phDevice*	Device handle returned by the interface.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_OUT_OF_RANGE	The index of the user input is bigger than the available devices number.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

The index of the device starts from 1.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;

GX_DEV_HANDLE hDevice = NULL;
// Opens the first device.
status = GXOpenDeviceByIndex(1, &hDevice);
```

7.4.10. GXOpenDevice

Declaration:

GX_API GXOpenDevice (GX_OPEN_PARAM* pOpenParam,
GX_DEV_HANDLE* phDevice)

Descriptions:

Open the device by a specific unique identification, such as: SN, IP, MAC, Index etc.

Formal parameter:

[in]*pOpenParam*	The open device parameter which is configurated by user. Ref:*GX_OPEN_PARAM*.
[out] *phDevice*	The device handle returned by the interface.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_NOT_FOUND_DEVICE	Not found the device that matches the specific information.
GX_STATUS_INVALID_ACCESS	The device can not be opened under the current access mode.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

It is recommended that you call the *GxUpdateDeviceList()* interface to make an enumeration before calling the function. To ensure that device list within the library is consistent with the current device.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
GX_OPEN_PARAM stOpenParam;

// Access mode.
stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
//stOpenParam.accessMode = GX_ACCESS_READONLY;
//stOpenParam.accessMode = GX_ACCESS_CONTROL;

// Access via serial number.
stOpenParam.openMode = GX_OPEN_SN;
stOpenParam.pszContent = "EA00010002";

// Access via IP address.
//stOpenParam.openMode = GX_OPEN_IP;
//stOpenParam.pszContent = "192.168.40.35";

// Access via MAC address.
//stOpenParam.openMode = GX_OPEN_MAC;
//stOpenParam.pszContent = "54-04-A6-C2-7C-2F";

/ Access via enumeration number 1, 2, 3...
//stOpenParam.openMode = GX_OPEN_INDEX;
//stOpenParam.pszContent = "1";

GX_DEV_HANDLE hDevice = NULL;
status = GXOpenDevice(&stOpenParam, &hDevice);
```

7.4.11. GXCloseDevice

Declaration:

GX_API GXCloseDevice (GX_DEV_HANDLE hDevice)

Descriptions:

Specify the device handle to close the device.

Formal parameter:

[in]*hDevice*	The device handle that the user specified to close. The *hDevice* can be get by *GXOpenDevice* interface.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The illegal handle that the user introduces, or reclose the device.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

Close the device handle that has been closed, return the GX_STATUS_INVALID_HANDLE error.

Code sample:

```
    GX_STATUS status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    uint32_t nDeviceNum = 0;

   // Enumerates to get the number of devices. The timeout time is decided by
   // the user's environment, and can be set by the user, 1000ms is an example.
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if (status == GX_STATUS_SUCCESS && nDeviceNum > 0)
    {
        // Sets the parameters of the opening device structures.
        stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
        stOpenParam.openMode   = GX_OPEN_INDEX;
        stOpenParam.pszContent = "1";

        GX_DEV_HANDLE hDevice = NULL;
        status = GXOpenDevice(&stOpenParam, &hDevice);
        if (status == GX_STATUS_SUCCESS)
        {
           // Operates device: control, acquisition.
           //…
           // Closes the device.
            status = GXCloseDevice(hDevice);
        }
    }
```

7.4.12. GXGetDevicePersistentIpAddress

Declaration:

```
GX_API GXGetDevicePersistentIpAddress (GX_DEV_HANDLE hDevice,
                                       char* pszIP,
                                       size_t *pnIPLength,
                                       char* pszSubNetMask,
                                       size_t *pnSubNetMaskLength,
                                       char* pszDefaultGateWay,
                                       size_t *pnDefaultGateWayLength)
```

Descriptions:

Get the persistent IP information of the device.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*pszIP*	The character string address of the device persistent IP.
[in, out]*pnIPLength*	The character string length of the device persistent IP address.
[in]*pnIPLength*	The user buffer size.
[out]*pnIPLength*	The actual filled buffer size.
[in]*pszSubNetMask*	The device persistent subnet mask character string address.
[in, out] *pnSubNetMaskLength*	The character string length of the device persistent subnet mask.
[in]*pnSubNetMaskLength*	The user buffer size.
[out]*pnSubNetMaskLength*	The actual filled buffer size.
[in]*pszDefaultGateWay*	The character string address of the device persistent gateway.
[in, out]*pnDefaultGateWayLength*	The character string length of the device persistent gateway.
[in]*pnDefaultGateWayLength*	The user buffer size.
[out]*pnDefaultGateWayLength*	The actual filled buffer size.

Return value:

`GX_STATUS_SUCCESS`	The operation is successful and no error occurs.
`GX_STATUS_NOT_INIT_API`	The GXInitLib initialization library is not called.
`GX_STATUS_INVALID_PARAMETER`	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code Sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
char szIP[32] = {0};
size_tnIPLen = 32;
char  szSubNetMask[32] = {0};
size_tnSubNetMaskLen = 32;
char szDefaultGateWay[32] = {0};
size_tnDefaultGateWayLen = 32;
status = GXGetDevicePersistentIpAddress(hDevice,
                                        szIP,
                                        &nIPLen,
                                        szSubNetMask,
                                        &nSubNetMaskLen,
                                        szDefaultGateWay,
                                        &nDefaultGateWayLen);
```

7.4.13. GXSetDevicePersistentIpAddress

Declaration:

```
GX_API GXSetDevicePersistentIpAddress (GX_DEV_HANDLE hDevice,
                                       const char* pszIP,
                                       const char* pszSubNetMask,
                                       const char* pszDefaultGateWay)
```

Descriptions:

Set the persistent IP information of the device.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*pszIP*	The persistent IP character string of the device. End with'\0'.
[in]*pszSubNetMask*	The persistent subnet mask character string of the device. End with'\0'.
[in]*pszDefaultGateWay*	The persistent gateway character string of the device. End with'\0'.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code Sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;

//Sets the persistent IP information for the device.
status = GXSetDevicePersistentIpAddress(hDevice,
                                        "192.168.1.2",
                                        "255.255.255.0",
                                        "192.168.1.1");
```

7.4.14. GXGetFeatureName

Declaration:

```
GX_API GXGetFeatureName (GX_DEV_HANDLE hDevice,
                         GX_FEATURE_ID featureID,
                         char* pszName,
                         size_t* pnSize)
```

Descriptions:

Get the string description for the feature code.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pszName*	The character string buffer address that the user inputs. The character string length includes the end terminator '\0'.
[in,out]*pnSize*	The length of the character string buffer address that the user inputs. Unit: byte.

If *pszName* is NULL:

[out]*pnSize*	Return the actual size of the character string.

If *pszName* is not NULL:

[in]*pnSize*	The size of the buffer that the user allocated.
[out]*pnSize*	Return the actual filled buffer size.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_NEED_MORE_BUFFER	The buffer that the user allocated is too small.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;

char *pszName = NULL;
size_t nSize = 0;
// First, the user passed the NULL pointer to get the actual size, followed
// by the request buffer to obtain the description information.
GXGetFeatureName(hDevice, GX_FLOAT_GAIN, NULL, &nSize);
pszName = new char[nSize];
status = GXGetFeatureName(hDevice, GX_FLOAT_GAIN, pszName, &nSize);
delete pszName;
```

7.4.15. GXIsImplemented

Declaration:

GX_API GXIsImplemented (GX_DEV_HANDLE hDevice,
GX_FEATURE_ID featureID,
bool* pbIsImplemented)

Descriptions:

Inquire the current camera whether support a special feature. Usually the camera doesn't support a feature means that:

1) By inquiring the camera register, the current camera really does not support this feature.
2) There is no description of this feature in the current camera description file.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pbIsImplemented*	To return the result whether is support this feature. If support, then returns **true**, if not support, **false** will return.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
bool bIsImplemented = false;

// The parameter hDevice has been obtained via GXOpenDevice, and is no longer
// described later.
status = GXIsImplemented(hDevice, GX_FLOAT_GAIN, &bIsImplemented);
```

7.4.16. GXIsReadable

Declaration:

```
GX_API GXIsReadable(GX_DEV_HANDLE hDevice,
                    GX_FEATURE_ID featureID,
                    bool* pbIsReadable)
```

Descriptions:

Inquire if a feature code is currently readable.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pbIsReadable*	To return the result whether the feature code ID is readable. If readable, then will return **true**, if not readable, **false** will return.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
bool bIsReadable = false;
status = GXIsReadable(hDevice, GX_FLOAT_GAIN, &bIsReadable);
```

7.4.17. GXIsWritable

Declaration:

```
GX_API GXIsWritable(GX_DEV_HANDLE hDevice,
                    GX_FEATURE_ID featureID,
                    bool* pbIsWritable)
```

Descriptions:

Inquire if a feature code is currently writable.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pbIsWritable*	To return the result whether the feature code ID is writable. If writable, then will return **true**, if not writable, **false** will return.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
bool bIsWritable = false;
status = GXIsWritable(hDevice, GX_FLOAT_GAIN, &bIsWritable);
```

7.4.18. GXGetIntRange

Declaration:

```
GX_API GXGetIntRange (GX_DEV_HANDLE hDevice,
                      GX_FEATURE_ID featureID,
                      GX_INT_RANGE* pIntRange)
```

Descriptions:

To get the minimum value, maximum value and steps of the int type.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pIntRange*	The structure of range description. Reference *GX_INT_RANGE.*

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not read the int range.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
GX_INT_RANGE stIntRange;
status = GXGetIntRange(hDevice, GX_INT_WIDTH, &IntRange);
```

7.4.19. GXGetInt

Declaration:

```
GX_API GXGetInt (GX_DEV_HANDLE hDevice,
                 GX_FEATURE_ID featureID,
                 int64_t* pnValue)
```

Descriptions:

Get the current value of the int type.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pnValue*	Point to the pointer of the current value returned.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not read.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
int64_t nValue = 0;
status = GXGetInt(hDevice, GX_INT_WIDTH, &nValue);
```

7.4.20. GXSetInt

Declaration:

```
GX_API GXSetInt (GX_DEV_HANDLE hDevice,
                 GX_FEATURE_ID featureID,
                 int64_t nValue)
```

Descriptions:

Set the value of int type.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[in]*nValue*	The value that the user will set.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_OUT_OF_RANGE	The value that the user introduces is across the border, smaller than the minimum, or larger than the maximum, or is not an integer multiple of the step.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not write.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
int64_t nValue = 200;
status = GXSetInt(hDevice, GX_INT_WIDTH, nValue);
```

7.4.21. GXGetFloatRange

Declaration:

```
GX_API GXGetFloatRange (GX_DEV_HANDLE hDevice,
                        GX_FEATURE_ID featureID,
                        GX_FLOAT_RANGE* pFloatRange)
```

Descriptions:

To get the minimum value, maximum value, steps and unit of the float type.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pFloatRange*	The description structure pointer of float type. Reference the *GX_FLOAT_RANGE.*

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not read the range of the float type.

The errors that are not covered above please reference *GX_STATUS_LIST.*

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
GX_FLOAT_RANGE stFloatRange;
status = GXGetFloatRange(hDevice, GX_FLOAT_EXPOSURE_TIME, &stFloatRange);
```

7.4.22. GXGetFloat

Declaration:

GX_API GXGetFloat (GX_DEV_HANDLE hDevice,
GX_FEATURE_ID featureID,
double* pdValue)

Descriptions:

Get the value of float type.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pdValue*	Point to the pointer of the float value returned.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not read.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
double dValue = 0;
status = GXGetFloat(hDevice, GX_FLOAT_EXPOSURE_TIME, &dValue);
```

7.4.23. GXSetFloat

Declaration:

```
GX_API GXSetFloat (GX_DEV_HANDLE hDevice,
                   GX_FEATURE_ID featureID,
                   double dValue)
```

Descriptions:

Set the value of float type.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[in]*dValue*	The float value that the user will set.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_OUT_OF_RANGE	The value that the user introduces is across the border, smaller than the minimum, or larger than the maximum.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not write.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
double dValue = 3000;
status = GXSetFloat(hDevice, GX_FLOAT_EXPOSURE_TIME, dValue);
```

7.4.24. GXGetEnumEntryNums

Declaration:

GX_API GXGetEnumEntryNums (GX_DEV_HANDLE hDevice,
GX_FEATURE_ID featureID,
uint32_t* pnEntryNums)

Descriptions:

Get the number of the options for the enumeration item.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pnEntryNums*	The pointer that point to the number returned.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
uint32_t nEntryNums = 0;
status = GXGetEnumEntryNums(hDevice, GX_ENUM_GAIN_AUTO, &nEntryNums);
```

7.4.25. GXGetEnumDescription

Declaration:

```
GX_API GXGetEnumDescription (GX_DEV_HANDLE hDevice,
                             GX_FEATURE_ID featureID,
                             GX_ENUM_DESCRIPTION* pEnumDescription,
                             size_t* pnBufferSize)
```

Descriptions:

To get the description information of the enumerated type values: the number of enumerated items and the value and descriptions of each item, please reference *GX_ENUM_DESCRIPTION*.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pEnumDescription*	The array pointer, used for the enumeration description information returned.
[in,out]*pnBufferSize*	The size of the GX_ENUM_DESCRIPTION array that the user introduces, unit: byte.

If *pEnumDescription* is NULL:

[out]*pnBufferSize*	The actual size of the buffer needed.

If *pEnumDescription* is not NULL:

[in]*pnBufferSize*	The size of the buffer that the user allocated.
[out]*pnBufferSize*	Return the actual filled buffer size.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_NEED_MORE_BUFFER	The buffer that the user allocates is too small.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
uint32_t nEntryNums = 0;

// Gets the number of this types supported by the device.
status = GXGetEnumEntryNums(hDevice, GX_ENUM_GAIN_AUTO, &nEntryNums);

// Applies memory according to the number.
GX_ENUM_DESCRIPTION* pEnumDescrption = new GX_ENUM_DESCRIPTION[nEntryNums];

size_t nSize = nEntryNums * sizeof(GX_ENUM_DESCRIPTION);
status = GXGetEnumDescription(hDevice, GX_ENUM_GAIN_AUTO, pEnumDescrption, &nSize);
delete []pEnumDescrption;
```

7.4.26. GXGetEnum

Declaration:

GX_API GXGetEnum (GX_DEV_HANDLE hDevice,
GX_FEATURE_ID featureID,
int64_t* pnValue)

Descriptions:

To get the current enumeration value.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pnValue*	The pointer that point to the return values.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not read.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
int64_t nValue = 0;
status = GXGetEnum(hDevice, GX_ENUM_GAIN_AUTO, &nValue);
```

7.4.27. GXSetEnum

Declaration:

GX_API GXSetEnum (GX_DEV_HANDLE hDevice,
GX_FEATURE_ID featureID,
int64_t nValue)

Descriptions:

Set the enumeration value.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[in]*nValue*	The enumeration values that the user will set. The value range can be got by the *nValue* of the GX_ENUM_DESCRIPTION.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_OUT_OF_RANGE	The value that the user introduces is cross the border.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not write.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
int64_t nValue = GX_GAIN_AUTO_CONTINUOUS;
status = GXSetEnum(hDevice, GX_ENUM_GAIN_AUTO, nValue);
```

7.4.28. GXGetBool

Declaration:

```
GX_API GXGetBool (GX_DEV_HANDLE hDevice,
                  GX_FEATURE_ID featureID,
                  bool* pbValue)
```

Descriptions:

Get the value of bool type.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pbValue*	The pointer that point to the bool value returned.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not read.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
bool bTemp = false;
status = GXGetBool(hDevice, GX_BOOL_REVERSE_X, &bTemp);
```

7.4.29. GXSetBool

Declaration:

```
GX_API GXSetBool (GX_DEV_HANDLE    hDevice,
                  GX_FEATURE_ID    featureID,
                  bool    bValue)
```

Descriptions:

Set the value of bool type.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[in]*bValue*	The bool value that the user will set.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not write.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
bool bTemp = true;
status = GXSetBool(hDevice, GX_BOOL_REVERSE_X, bTemp);
```

7.4.30. GXGetStringLength

Declaration:

```
GX_API GXGetStringLength (GX_DEV_HANDLE hDevice,
                          GX_FEATURE_ID featureID,
                          size_t* pnSize)
```

Descriptions:

Get the current value length of the character string type. Unit: byte. The user can allocate the buffer size according to the length information that is get from the function, and then call the GXGetString to get the character string information.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pnSize*	The pointer that point to the length value returned. The length value is end with '\0', unit: byte.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
size_t nSize = 0;
status = GXGetStringLength(hDevice, GX_STRING_DEVICE_VENDOR_NAME, &nSize);
```

7.4.31. GXGetStringMaxLength

Declaration:

```
GX_API GXGetStringMaxLength (GX_DEV_HANDLE hDevice,
                             GX_FEATURE_ID featureID,
                             size_t* pnSize)
```

Descriptions:

Get the maximum length of the string type value. Unit: byte. The user allocates buffer according to the length information obtained, then call the *GXGetString* to get the string information. This interface can get the maximum possible length of the string (including the terminator '\ 0'), but the actual length of the string might not be that long, if the user wants to allocate buffer according to the actual string length, the user can call the *GXGetStringLength* interface to get the actual string length.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pnSize*	The pointer that point to the length value returned. The length value isend with '\0', unit: byte.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
size_t nSize = 0;
status = GXGetStringMaxLength(hDevice, GX_STRING_DEVICE_VENDOR_NAME, &nSize);
```

7.4.32. GXGetString

Declaration:

```
GX_API GXGetString (GX_DEV_HANDLE hDevice,
                    GX_FEATURE_ID featureID,
                    char* pszContent,
                    size_t* pnSize)
```

Descriptions:

Get the content of the string type value.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pszContent*	Point to the string buffer address that the user allocated.
[in,out]*pnSize*	The length of the string buffer address that the user inputs.

If *pszContent* is NULL:

[out]*pnSize*	Return the actual size of the buffer needed.

If *pszContent* is not NULL:

[in]*pnSize*	The size of the buffer that the user allocated.
[out]*pnSize*	Return the actual filled buffer size.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not read.
GX_STATUS_NEED_MORE_BUFFER	The buffer that the user allocates is too small.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
size_t nSize = 0;

// Gets string length.
status = GXGetStringLength(hDevice, GX_STRING_DEVICE_VENDOR_NAME, &nSize);

// Applies memory based on the length.
char *pszText = new char[nSize];

status = GXGetString(hDevice, GX_STRING_DEVICE_VENDOR_NAME, pszText, &nSize);
```

7.4.33. GXSetString

Declaration:

```
GX_API GXSetString (GX_DEV_HANDLE hDevice,
                    GX_FEATURE_ID featureID,
                    char* pszContent)
```

Descriptions:

Set the content of the string value.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[in]*pszContent*	The string address that the user will set. The string is end with '\0'.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user introduces is NULL.
GX_STATUS_OUT_OF_RANGE	The maximum length that the content the user writes exceeds the string size.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not write.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
char *pszText = "test";
status = GXSetString(hDevice, GX_STRING_USER_PASSWORD, pszText);
```

7.4.34. GXGetBufferLength

Declaration:

```
GX_API GXGetBufferLength (GX_DEV_HANDLE hDevice,
                          GX_FEATURE_ID featureID,
                          size_t* pnSize)
```

Descriptions:

Get the length of the chunk data and the unit is byte, the user can apply the buffer based on the length obtained, and then call the GXGetBuffer to get the chunk data.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[in]*bValue*	The pointer that points to the length value returned. Unit: byte.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
size_t nLength = 0;
status = GXGetBufferLength(hDevice, GX_BUFFER_FRAME_INFORMATION, &nLength);
```

7.4.35. GXGetBuffer

Declaration:

```
GX_API GXGetBuffer (GX_DEV_HANDLE hDevice,
                    GX_FEATURE_ID featureID,
                    uint8_t* pBuffer,
                    size_t* pnSize)
```

Descriptions:

Get the chunk data.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pBuffer*	The pointer that point to the chunk data buffer address that the user applied.
[in,out]*pnSize*	The length of the buffer address that the user inputs.

If *pBuffer* is NULL:

[out]*pnSize*	Return the actual size of the buffer needed.

If *pBuffer* is not NULL:

[in]*pnSize*	The size of the buffer that the user allocated.
[out]*pnSize*	Return the actual filled buffer size.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not read.
GX_STATUS_NEED_MORE_BUFFER	The buffer that the user allocates is too small.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
// Takes the read of the look-up table as an example.
// Reads the look-up table length.
size_t nLutLength = 0;
emStatus = GXGetBufferLength(hDevice, GX_BUFFER_LUT_VALUEALL, &nLutLength);

// Applies buffer.
uint8_t *pGetLutBuffer = new uint8_t[nLutLength];

// Reads the look-up table contents.
emStatus = GXGetBuffer(hDevice, GX_BUFFER_LUT_VALUEALL, pGetLutBuffer,
                       nLutLength);
```

7.4.36. GXSetBuffer

Declaration:

```
GX_API GXSetBuffer (GX_DEV_HANDLE hDevice,
                    GX_FEATURE_ID featureID,
                    uint8_t* pBuffer,
                    size_t nSize)
```

Descriptions:

Set the chunk data.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[out]*pBuffer*	The pointer that point to the chunk data buffer address that the user will set.
[in]*nSize*	The length of the buffer address that the user inputs.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_PARAMETER	The pointer that the user introduces is NULL.
GX_STATUS_OUT_OF_RANGE	The maximum length that the content the user writes exceeds the string size.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not write.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
// To set the look-up table as an example.
// Reads the look-up table length.
size_t nLutLength = 0;
emStatus = GXGetBufferLength(hDevice, GX_BUFFER_LUT_VALUEALL, &nLutLength);

// Applies buffer.
uint8_t *pSetLutBuffer = new uint8_t[nLutLength];
 // Sets the look-up table contents (For example, you can read a look-up table
 // from the local file).
 // There does not explain how to set the contents of the look-up table
 // buffer. Please install the settings you need to find the look-up table
 // contents.
 // Sets the look-up table.
emStatus = GXSetBuffer(hDevice, GX_BUFFER_LUT_VALUEALL, pSetLutBuffer, nLutLength);
```

7.4.37. GXSendCommand

Declaration:

GX_API GXSendCommand (GX_DEV_HANDLE hDevice,
GX_FEATURE_ID featureID)

Descriptions:

Send the command.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_NOT_IMPLEMENTED	The feature that is not support currently.
GX_STATUS_ERROR_TYPE	The featureID type that the user introduces is error.
GX_STATUS_INVALID_ACCESS	Currently inaccessible, can not send command.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
status = GXSendCommand(hDevice, GX_COMMAND_TRIGGER_SOFTWARE);
```

7.4.38. GXSetAcqusitionBufferNumber (Linux only)

Declaration:

GX_API GXSetAcqusitionBufferNumber (GX_DEV_HANDLE hDevice,
uint64_t nBufferNum)

Descriptions:

Set the number of the acquisition buffers.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*nBufferNum*	The number of the acquisition buffers that the user sets.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The input parameter that the user introduces is invalid.
GX_STATUS_INVALID_CALL	After sending the start acquisition command, the user can not set the number of the acquisition buffers.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

1) This interface is an optional interface, not a necessary part of the acquisition process.
2) When the acquisition command is sent, it will be not allowed to set up the number of buffers, otherwise, it will return GX_STATUS_INVALID_CALL.
3) The number of the acquisition buffers that the user set must greater than 0, and if the number of buffers is 0, it will return GX_STATUS_INVALID_PARAMETER.
4) The user should consider the reasonableness when setting the number of acquisition buffers, if the number is set too many, the memory that is occupied in the acquisition process is large; if the number is set too small, the memory is small when in the acquisition process, but some frames will be lost because of the lack of the buffer.
5) Once the user has set the number of the buffers and has captured successfully, then the number of buffers will remain valid until the device is turned off.

Code sample:

```
#include"GxIAPI.h"

// Image processing callback function.
static void GX_STDC OnFrameCallbackFun(GX_FRAME_CALLBACK_PARAM* pFrame)
{
    if (pFrame->status == GX_FRAME_STATUS_SUCCESS)
    {
        // Do some operations on images.
    }
    return;
}

int main(int argc, char* argv[])
{
    GX_STATUS status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
```

```
    GX_OPEN_PARAM stOpenParam;
    uint32_t nDeviceNum = 0;

    // Initializes the library.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    // Updates the enumeration list for the devices.
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS)||(nDeviceNum <= 0))
    {
        return 0;
    }

    // Opens the device.
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Sets the number of acquisition buffers.
        status = GXSetAcqusitionBufferNumber(hDevice,10);

        // Registers image processing callback function.
        status = GXRegisterCaptureCallback(hDevice, NULL,
                                           OnFrameCallbackFun);

        // Stream On.
        status = GXStreamOn(hDevice);

        //----------------------
        //
       // In this interval, the image will be returned to the user via the
        // OnFrameCallbackFun interface.
        //
        //----------------------

        // Sends a stop acquisition command.
        status = GXStreamOff(hDevice);

        // Unregisters image processing callback function.
        status = GXUnregisterCaptureCallback(hDevice);
    }
    status = GXCloseDevice(hDevice);
    status = GXCloseLib();

    return 0;
}
```

7.4.39. GXStreamOn (Linux only)

Declaration:

GX_API GXStreamOn(GX_DEV_HANDLE hDevice)

Descriptions:

Start acquisition, including stream acquisition and device acquisition.

Formal parameter:

[in]*hDevice*	The handle of the device.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_ACCESS	Device access mode error.
GX_STATUS_ERROR	Unspecified internal errors that are not expected to occur

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions：

Start acquisition for all stream channels.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
status = GXStreamOn(hDevice);
```

7.4.40. GXStreamOff (Linux only)

Declaration:

GX_API GXStreamOff(GX_DEV_HANDLE hDevice)

Descriptions:

Stop acquisition, including stop stream acquisition and stop device acquisition

Formal parameter:

[in]*hDevice*	The handle of the device.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_ACCESS	Device access mode error.
GX_STATUS_INVALID_CALL	Invalid interface call.
GX_STATUS_ERROR	Unspecified internal errors that are not expected to occur.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions :

Stop acquisition for all stream channels.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
status = GXStreamOff(hDevice);
```

7.4.41. GXDQBuf (Linux only)

Declaration:

GX_API GXDQBuf(GX_DEV_HANDLE hDevice,
PGX_FRAME_BUFFER* ppFrameBuffer,
uint32_t nTimeOut)

Descriptions:

After starting the acquisition, an image (zero copy) can be acquired through this interface.

Formal parameter:

[in]*hDevice*	The handle of the device.
[out]*ppFrameBuffer*	Address pointer of image data output by the interface.
[in]*nTimeOut*	Take timeout time (unit: ms).

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The pointer that the user introduces is NULL.
GX_STATUS_INVALID_CALL	Acquisition is not started or the callback is registered, this interface is not allowed to be called.
GX_STATUS_TIMEOUT	Acquire image timeout error.
GX_STATUS_ERROR	Unspecified internal errors that are not expected to occur.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions :

1) The GXDQBuf interface is not allowed to be called until the acquisition is started. If called, it will return GX_STATUS_INVALID_CALL error;
2) After registering the capture callback, the GXDQBuf interface is not allowed to be called. If called, it will return GX_STATUS_INVALID_CALL error;
3) The GXDQBuf interface needs to be used with the GXQBuf interface, otherwise the image cannot be continuously acquired.

Code sample:

```
//------------------------------
// The GXDQBuf interface acquires one frame of image at a time. This sample
// code demonstrates how to use this interface to get a frame of image.
//------------------------------
#include "GxIAPI.h"

int main(int argc, char* argv[])
{
    GX_STATUS     status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    uint32_t      nDeviceNum = 0;

    // Initializes the library.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }
```

```
    // Updates the enumeration list for the devices.
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }

    // Opens the device.
    status = GXOpenDeviceByIndex(1, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Defines the incoming parameters of GXDQBuf.
        PGX_FRAME_BUFFER pFrameBuffer;

        // Stream on.
        status = GXStreamOn(hDevice);
        if (status == GX_STATUS_SUCCESS)
        {
            // Calls GXDQBuf to get a frame of image.
            status = GXDQBuf(hDevice, &pFrameBuffer, 1000);
            if (status == GX_STATUS_SUCCESS)
            {
                if (pFrameBuffer->nStatus == GX_FRAME_STATUS_SUCCESS)
                {
                    // Successfully acquired images.
                    // Image processing...
                }

                // Calls GXQBuf to put the image buf back into the library and
                //continue to acquire images.
                status = GXQBuf(hDevice, pFrameBuffer);
            }
        }

        // Sends a stop acquisition command.
        status = GXStreamOff(hDevice);
    }
    status = GXCloseDevice(hDevice);
    status = GXCloseLib();

    return 0;
}
```

7.4.42. GXQBuf (Linux only)

Declaration:

GX_API GXQBuf(GX_DEV_HANDLE hDevice, PGX_FRAME_BUFFER pFrameBuffer)

Descriptions:

After the acquisition is started, the image data buffer can be placed back into the GxIAPI library through this interface and continue to be used for acquisition.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*pFrameBuffer*	Image data buffer pointer to be placed back into the GxIAPI library.

Return value:

`GX_STATUS_SUCCESS`	The operation is successful and no error occurs.
`GX_STATUS_NOT_INIT_API`	The GXInitLib initialization library is not called.
`GX_STATUS_INVALID_HANDLE`	The handle that the user introduces is illegal.
`GX_STATUS_INVALID_PARAMETER`	The user incoming pointer is NULL. / Incoming unreasonable pointer.
`GX_STATUS_INVALID_CALL`	Acquisition is not started or the callback is registered. It is not allowed to call the interface.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

1) It is not allowed to call the GXQBuf interface before starting the acquisition. If it is called, it will return the GX_STATUS_INVALID_CALL error;
2) After registering the capture callback, the GXQBuf interface is not allowed to be called. If it is called, it will return GX_STATUS_INVALID_CALL error;
3) The GXDQBuf interface needs to be used with the GXQBuf interface. Otherwise, the image cannot be continuously acquired;
4) After the GXQBuf interface puts the image buffer back into the GxIAPI library, the image buffer pointer can no longer be accessed.

Code sample:

See the *GXDQBuf* sample program.

7.4.43. GXDQAllBufs (Linux only)

Declaration:

```
GX_API GXDQAllBufs(GX_DEV_HANDLE hDevice,
                   PGX_FRAME_BUFFER *ppFrameBufferArray,
                   uint32_t nFrameBufferArraySize,
                   uint32_t *pnFrameCount,
                   uint32_t nTimeOut)
```

Descriptions:

After starting the acquisition, all the buffers (zero copies) of the acquired images can be obtained through this interface. The order of the stored images in the image data array is from old to new, that is, ppFrameBufferArray[0] stores the oldest image, and ppFrameBufferArray[nFrameCount – 1] stores the latest image.

Formal parameter:

[in]*hDevice*	The handle of the device.
[out]*ppFrameBufferArray*	Array of image data pointers.
[in]*nFrameBufferArraySize*	The number of applications for image arrays.
[out]*pnFrameCount*	Returns the number of actual filled images.
[in]*nTimeOut*	Take timeout time (unit: ms).

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The pointer that the user introduces is NULL.
GX_STATUS_INVALID_CALL	Acquisition is not started or the callback is registered. It is not allowed to call the interface.
GX_STATUS_NEED_MORE_BUFFER	Insufficient buffer requested by the user: When reading, the user input buffersize is smaller than the actual need.
GX_STATUS_TIMEOUT	Acquire image timeout error.
GX_STATUS_ERROR	Unspecified internal errors that are not expected to occur.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

1) It is not allowed to call the GXDQAllBufs interface before starting the acquisition. If it is called, it will return the GX_STATUS_INVALID_CALL error;
2) After registering the capture callback, the GXDQAllBufs interface is not allowed to be called. If it is called, it will return GX_STATUS_INVALID_CALL error;
3) The GXDQAllBufs interface needs to be used with the GXQAllBufs interface. Otherwise, the image cannot be continuously acquired;
4) The array size of the image data pointer should be greater than or equal to the number of image acquire buffers (default is 5). Otherwise, GX_STATUS_NEED_MORE_BUFFER error will be returned.

Code sample:

```
#include "GxIAPI.h"
#include <stdint.h>

int main(int argc, char* argv[])
{
    GX_STATUS     status = GX_STATUS_SUCCESS;
```

```
GX_DEV_HANDLE hDevice = NULL;
uint32_t      nDeviceNum = 0;

// Initializes the library.
status = GXInitLib();
if (status != GX_STATUS_SUCCESS)
{
    return 0;
}

// Updates the enumeration list for the devices.
status = GXUpdateDeviceList(&nDeviceNum, 1000);
if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
{
    return 0;
}

// Opens the device.
status = GXOpenDeviceByIndex(1, &hDevice);
if (status == GX_STATUS_SUCCESS)
{
    // Defines the array of received images.
    // The number of image acquire buffers is 5 by default, and the image
    // array size should be greater than or equal to the number of image
    // acquire buffers.
    PGX_FRAME_BUFFER pFrameBuffer[5];

    // Defines the actual number of filled images.
    uint32_t nFrameCount = 0;

    // Stream On.
    status = GXStreamOn(hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Calls GXDQAllBufs to get all the images in the queue.
        status = GXDQAllBufs(hDevice, pFrameBuffer, 5, &nFrameCount,
                             1000);
        if (status == GX_STATUS_SUCCESS)
        {
            for (int i = 0; i < nFrameCount; i++)
            {
               if (pFrameBuffer[i] != NULL &&
                    pFrameBuffer[i]->nStatus == GX_FRAME_STATUS_SUCCESS)
              {
                  // The image i was successfully acquired.
                  // Image processing...
              }
             }

       // Calls GXQAllBufs to put all the acquired image buffers back
       // into the library and continue to acquire images.
       status = GXQAllBufs(hDevice);
         }
```

```
        }

        //Sends a stop acquisition command.
        status = GXStreamOff(hDevice);
    }
    status = GXCloseDevice(hDevice);
    status = GXCloseLib();

    return 0;
}
```

7.4.44. GXQAllBufs (Linux only)

Declaration:

GX_API GXQAllBufs(GX_DEV_HANDLE hDevice)

Descriptions:

After the acquisition is started, all the acquired image data buffers can be put back into the GxIAPI library through this interface, and continue to be used for acquisition.

Formal parameter:

[in]*hDevice* The handle of the device.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_CALL	Acquisition is not started or the callback is registered. It is not allowed to call the interface.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

1) It is not allowed to call the GXQAllBufs interface before starting the acquisition. If it is called, it will return the GX_STATUS_INVALID_CALL error;
2) After registering the capture callback, the GXQAllBufs interface is not allowed to be called. If it is called, it will return GX_STATUS_INVALID_CALL error;
3) The GXQAllBufs interface needs to be used with the GXDQAllBufs interface. Otherwise, the image cannot be continuously acquired;
4) The GXQAllBufs interface puts all the image buffers obtained by GXDQAllBufs back into the GxIAPI library, and these image buffer pointers can no longer be accessed.

Code sample:

See the sample code of *GXDQAllBufs*.

7.4.45. GXRegisterCaptureCallback

Declaration:

GX_API GXRegisterCaptureCallback (GX_DEV_HANDLE hDevice,
void *pUserParam,
GXCaptureCallBack callBackFun)

Descriptions:

Register the capture callback function, corresponding to *GXUnregisterCaptureCallback*.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*pUserParam*	The private data pointer that the user will use in the callback function.
[in]*callBackFun*	The callback function that the user will register, for the function type, see *GXCaptureCallBack.*

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The pointer that the user introduces is NULL.
GX_STATUS_INVALID_CALL	After sending the start acquisition command, the user can not register the capture callback function.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

After sending the start acquisition command, the user can not register the callback function. Otherwise, it will return GX_STATUS_INVALID_CALL.

Code sample:

```
#include "GxIAPI.h"

// Image processing callback function.
static void GX_STDC OnFrameCallbackFun(GX_FRAME_CALLBACK_PARAM* pFrame)
{
    if (pFrame->status == GX_FRAME_STATUS_SUCCESS)
    {
        // Do some image processing operations.
    }
    return;
}

int main(int argc, char* argv[])
{
    GX_STATUS     status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    uint32_t      nDeviceNum = 0;

    // Initializes the library.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
```

```
    }

    // Updates the enumeration list for the devices.
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }

    // Opens the device.
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode   = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Registers image processing callback function.
        status = GXRegisterCaptureCallback(hDevice, NULL,
                                           OnFrameCallbackFun);

        // Stream On.
        status = GXStreamOn(hDevice);

        //----------------------
        //
        // In this interval, the image will be returned to the user via the
        // OnFrameCallbackFun interface.
        //
        //----------------------

        // Sends a stop acquisition command.
        status = GXStreamOff(hDevice);

        // Unregisters image processing callback function.
        status = GXUnregisterCaptureCallback(hDevice);
    }
    status = GXCloseDevice(hDevice);
    status = GXCloseLib();

    return 0;
}
```

7.4.46. GXUnregisterCaptureCallback

Declaration:

GX_API GXUnregisterCaptureCallback(GX_DEV_HANDLE hDevice)

Descriptions:

Unregister the capture callback function, corresponding to *GXRegisterCaptureCallback*.

Formal parameter:

[in]*hDevice*	The handle of the device.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_CALL	After sending the stop acquisition command, the user can not unregister the capture callback function.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

Before sending the stop acquisition command, the user can not unregister the callback function. Otherwise, it will return GX_STATUS_INVALID_CALL.

Code sample:

```
#include "GxIAPI.h"

// Image processing callback function.
static void GX_STDC OnFrameCallbackFun(GX_FRAME_CALLBACK_PARAM* pFrame)
{
    if (pFrame->status == GX_FRAME_STATUS_SUCCESS)
    {
        // Do some image processing operations.
    }
    return;
}

int main(int argc, char* argv[])
{
    GX_STATUS     status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    uint32_t      nDeviceNum = 0;

    // Initializes the library.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    // Updates the enumeration list for the devices.
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
```

```
        return 0;
    }

    // Opens the device.
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode   = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Registers image processing callback function.
        status = GXRegisterCaptureCallback(hDevice, NULL,
                                           OnFrameCallbackFun);

        // Stream On.
        status = GXSendCommand(hDevice, GX_COMMAND_ACQUISITION_START);

        //----------------------
        //
        // In this interval, the image will be returned to the user via the
        // OnFrameCallbackFun interface.
        //
        //----------------------

        // Sends a stop acquisition command.
        status = GXSendCommand(hDevice, GX_COMMAND_ACQUISITION_STOP);


        // Unregisters image processing callback function.
        status = GXUnregisterCaptureCallback(hDevice);
    }
    status = GXCloseDevice(hDevice);
    status = GXCloseLib();

    return 0;
}
```

7.4.47. GXGetImage

Declaration:

GX_API GXGetImage (GX_DEV_HANDLE hDevice,
GX_FRAME_DATA *pFrameData,
int32_t nTimeout)

Descriptions:

After starting acquisition, you can call this function to get images directly. Noting that the interface can not be mixed with the callback capture mode.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in,out]*pFrameData*	The pointer to the address that the user introduced to receive the image data.
[in]*nTimeout*	The timeout time of capture image (unit: ms).

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_CALL	After registering the capture callback function, the user calls the GXGetImage to get image.
GX_STATUS_INVALID_PARAMETER	User incoming image address pointer is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

The GXGetImage interface is not allowed to be called after the capture callback function is registered, and the call will return GX_STATUS_INVALID_CALL error. When using high resolution cameras for high-speed acquisition, because there is a buffer copy within the GXGetImage interface, it will affect the transport performance. It is recommended that users use the capture callback mode in this case.

Code sample:

```
//------------------------------
// The GXGetImage interface can take one image at a time. The implementation
// process is shown below.
//------------------------------
#include "GxIAPI.h"

int main(int argc, char* argv[])
{
    GX_STATUS     status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    uint32_t      nDeviceNum = 0;

    // Initializes the library.
    status = GXInitLib();
    if (status != GX_STATUS_SUCCESS)
    {
        return 0;
    }

    // Updates the enumeration list for the devices.
```

```
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }

    // Opens the device.
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode   = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        int64_t nPayLoadSize = 0;
        // Gets the image buffer size, and then apply for dynamic memory.
        status = GXGetInt(hDevice, GX_INT_PAYLOAD_SIZE, &nPayLoadSize);

        if (status == GX_STATUS_SUCCESS && nPayLoadSize > 0)
        {
            // Defines the incoming parameters of the GXGetImage interface.
            GX_FRAME_DATA stFrameData;

            // Applies for buffer according to the acquired image buffer size
            // m_nPayLoadSize.
            stFrameData.pImgBuf     = malloc((size_t)nPayLoadSize);

            // Stream On.
            status = GXStreamOn(hDevice);
            if (status == GX_STATUS_SUCCESS)
            {
                // Calls GXGetImage to get an image.
                while(GXGetImage(hDevice, &stFrameData, 100) != GX_STATUS_SUCCESS)
                {
                    Sleep(10);
                }
                 if (stFrameData.nStatus == GX_FRAME_STATUS_SUCCESS)
                {
                    // Acquiring image is successful.
                    // Image processing...
                }
            }
            // Sends a stop acquisition command.
            status = GXStreamOff(hDevice);

            // Frees image buffer.
            free(stFrameData.pImgBuf);
        }
    }
    status = GXCloseDevice(hDevice);
    status = GXCloseLib();

    return 0;
}
```

7.4.48. GXFlushQueue

Declaration:

GX_API GXFlushQueue(GX_DEV_HANDLE hDevice)

Descriptions:

Empty the cache image in the image output queue.

Formal parameter:

[in]*hDevice*	The handle of the device.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

If the user processes the images too slow, the image of last acquisition may be remained in the queue. Especially in the trigger mode, after the user send the trigger signal, and get the old image (last image). If you want to get the current image that corresponding to trigger signal, you should call the GXFlushQueue interface before sending the trigger signal to empty the image output queue.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
status = GXFlushQueue(hDevice);
```

7.4.49. GXRegisterDeviceOfflineCallback

Declaration:

GX_API GXRegisterDeviceOfflineCallback(GX_DEV_HANDLE hDevice,
void* pUserParam,
GXDeviceOfflineCallBack callBackFun,
GX_EVENT_CALLBACK_HANDLE *pHCallBack)

Descriptions:

At present, the Mercury Gigabit camera provides the device offline notification event mechanism, the user can call this interface to register the event handle callback function.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*pUserParam*	User private parameter.
[in]*callBackFun*	The user event handle callback function, for the function type, see *GXDeviceOfflineCallBack*.
[out]*pHCallBack*	The handle of offline callback function, the handle is used for unregistering the callback function.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The unsupported event ID or the callback function is illegal.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample: Get the offline event

```
#include "GxIAPI.h"

// Device offline callback function.
static void GX_STDC OnDeviceOfflineCallbackFun(void* pUserParam)
{
    // After receiving the offline notification, the user needs to notify the
    // main thread actively to stop acquiring or closing the device.
    return;
}

int main(int argc, char* argv[])
{
    GXInitLib();
    GX_STATUS status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode   = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
```

```
    if (status == GX_STATUS_SUCCESS)
    {
        // Defines the handle to the offline callback function.
        GX_EVENT_CALLBACK_HANDLE hCB;

        // Registers device offline callback function.
        GXRegisterDeviceOfflineCallback(hDevice,NULL,
                                        OnDeviceOfflineCallbackFun, &hCB);

        //----------------------
        //
        // The event notification will be returned to the user via the
        // OnDeviceOfflineCallbackFun interface.
        //
        //----------------------

        // Unregisters device offline callback function.
        GXUnregisterDeviceOfflineCallback(hDevice,hCB);
    }
    status = GXCloseDevice(hDevice);
    GXCloseLib();
    return 0;
}
```

7.4.50. GXUnregisterDeviceOfflineCallback

Declaration:

GX_API GXUnregisterDeviceOfflineCallback(GX_DEV_HANDLE hDevice,
GX_EVENT_CALLBACK_HANDLE hCallBack)

Descriptions:

Unregister event handle callback function.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*hCallBack*	The handle of device offline callback function.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample: Get device offline event

```
#include "GxIAPI.h"

// Device offline callback function.
static void GX_STDC OnDeviceOfflineCallbackFun(void* pUserParam)
{
    // After receiving the offline notification , the user needs to notify the
    // main thread actively to stop acquiring or close the device.
    return;
}

int main(int argc, char* argv[])
{
    GXInitLib();
    GX_STATUS status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode   = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Defines the handle to the offline callback function.
        GX_EVENT_CALLBACK_HANDLE hCB;

       // Registers device offline callback function.
      GXRegisterDeviceOfflineCallback(hDevice,NULL,
                                      OnDeviceOfflineCallbackFun, &hCB);
```

```
        //----------------------
        //
        // The event notification will be returned to the user via the
        // OnDeviceOfflineCallbackFun interface.
        //
        //----------------------

        // Unregisters device offline callback function.
        GXUnregisterDeviceOfflineCallback(hDevice,hCB);
    }
    status = GXCloseDevice(hDevice);
    GXCloseLib();
    return 0;
}
```

7.4.51. GXRegisterFeatureCallback

Declaration:

```
GX_API GXRegisterFeatureCallback(GX_DEV_HANDLE hDevice,
                                 void* pUserParam,
                                 GXFeatureCallBack   callBackFun,
                                 GX_FEATURE_ID   featureID,
                                 GX_FEATURE_CALLBACK_HANDLE *pHCallBack)
```

Descriptions:

Register device attribute update callback function. When the current value of the device property has updated, or the accessible property is changed, call this callback function.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*pUserParam*	User private parameter.
[in]*callBackFun*	The user event handle callback function, for function type, see *GXFeatureCallBack.*
[in]*featureID*	The feature code ID.
[out]*pHCallBack*	The handle of property update callback function, to unregister the callback function.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The unsupported event ID or the callback function is illegal.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample: Get the event of remote device

```
#include "GxIAPI.h"

// The event callback processing function of the remote device.
static void GX_STDC OnFeatureCallbackFun(GX_FEATURE_ID featureID, void*
                                         pUserParam)
{
if (featureID == GX_INT_EVENT_EXPOSUREEND)
    {
        // Gets the event data such as timestamp and frame ID for the frame
        // exposure end event.
        int64_t nFrameID=0;
        GXGetInt(hDevice, GX_INT_EVENT_EXPOSUREEND_FRAMEID, &nFrameID);
    }

    return;
}

int main(int argc, char* argv[])
{
    GXInitLib();
    GX_STATUS status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
```

```
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode   = GX_OPEN_INDEX;
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Selects the exposure end event.
        GXSetEnum(hDevice,GX_ENUM_EVENT_SELECTOR,
                  GX_ENUM_EVENT_SELECTOR_EXPOSUREEND);

        // Enables the exposure end event.
        GXSetEnum(hDevice,GX_ENUM_EVENT_NOTIFICATION,
                  GX_ENUM_EVENT_NOTIFICATION_ON);

       // Declares the attribute update callback function handle.
       GX_FEATURE_CALLBACK_HANDLE hCB;

        // Registers the callback function for the exposure end event.
       GXRegisterFeatureCallback(hDevice,
                                 NULL,
                                 OnFeatureCallbackFun,
                                 GX_INT_EVENT_EXPOSUREEND,
                                 &hCB);

        //----------------------
        //
        // The event notification will be returned to the user via the
        // OnFeatureCallbackFun interface.
        //
        //----------------------

        // Unregisters the callback function for the exposure end event.
        GXUnregisterFeatureCallback(hDevice, GX_INT_EVENT_EXPOSUREEND,
                                    hCB);
    }
    status = GXCloseDevice(hDevice);
    GXCloseLib();
    return 0;
}
```

7.4.52. GXUnregisterFeatureCallback

Declaration:

GX_API GXUnregisterFeatureCallback(GX_DEV_HANDLE hDevice,
GX_FEATURE_ID featureID,
GX_FEATURE_CALLBACK_HANDLE hCallBack)

Descriptions:

Unregister device attribute update callback function.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*featureID*	The feature code ID.
[in]*hCallBack*	The attribute update callback function handle.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The unsupported event ID.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample： Get the event of remote device

```
#include "GxIAPI.h"

// The event callback processing function of the remote device.
static void GX_STDC OnFeatureCallbackFun(GX_FEATURE_ID featureID, void*
                                         pUserParam)
{
    if (featureID == GX_INT_EVENT_EXPOSUREEND)
    {
        // Gets the event data such as timestamp and frame ID for the frame
        // exposure end event.
        int64_t nFrameID=0;
        GXGetInt(hDevice, GX_INT_EVENT_EXPOSUREEND_FRAMEID, &nFrameID);
    }

    return;
}

int main(int argc, char* argv[])
{
    GXInitLib();
    GX_STATUS status = GX_STATUS_SUCCESS;
    GX_DEV_HANDLE hDevice = NULL;
    GX_OPEN_PARAM stOpenParam;
    status = GXUpdateDeviceList(&nDeviceNum, 1000);
    if ((status != GX_STATUS_SUCCESS) || (nDeviceNum <= 0))
    {
        return 0;
    }
    stOpenParam.accessMode = GX_ACCESS_EXCLUSIVE;
    stOpenParam.openMode   = GX_OPEN_INDEX;
```

```
    stOpenParam.pszContent = "1";
    status = GXOpenDevice(&stOpenParam, &hDevice);
    if (status == GX_STATUS_SUCCESS)
    {
        // Selects the exposure end event.
        GXSetEnum(hDevice,GX_ENUM_EVENT_SELECTOR,
                  GX_ENUM_EVENT_SELECTOR_EXPOSUREEND);

        // Enables the exposure end event.
        GXSetEnum(hDevice,GX_ENUM_EVENT_NOTIFICATION,
                  GX_ENUM_EVENT_NOTIFICATION_ON);

       // Declares the attribute update callback handle.
       GX_FEATURE_CALLBACK_HANDLE hCB;

      // Registers the callback function for the exposure end event.
        GXRegisterFeatureCallback(hDevice,
                                  NULL,
                                  OnFeatureCallbackFun,
                                  GX_INT_EVENT_EXPOSUREEND,
                                  &hCB);

        //----------------------
        //
        // The event notification will be returned to the user via the
        // OnFeatureCallbackFun interface.
        //
        //----------------------

        // Unregisters the callback function for the exposure end event.
        GXUnregisterFeatureCallback(hDevice, GX_INT_EVENT_EXPOSUREEND,
                                    hCB);
    }
    status = GXCloseDevice(hDevice);
    GXCloseLib();
    return 0;
}
```

7.4.53. GXFlushEvent

Declaration:

GX_API GXFlushEvent (GX_DEV_HANDLE hDevice)

Descriptions:

Empty the device event, such as the frame exposure to end the event data queue.

Formal parameter:

[in]*hDevice*	The handle of the device.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Precautions:

The library internal event data receiving and processing using caching mechanism, if the user receiving, processing event speed is slower than the event generates, then the event data will be accumulated in the library, it will affect the the user to get real-time event data. If you want to get the real-time event data, you need to call the GXFlushEvent interface to clear the event cache data. This interface empties all the event data at once.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
status = GXFlushEvent(hDevice);
```

7.4.54. GXGetEventNumInQueue

Declaration:

GX_API GXGetEventNumInQueue (GX_DEV_HANDLE hDevice, uint32_t *pnEventNum)

Descriptions:

Get the number of the events in the current remote device event queue cache.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*pnEventNum*	The pointer of event number.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.
GX_STATUS_INVALID_PARAMETER	The pointer that the user input is NULL.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
uint32_t  nEventNum = 0;
status = GXGetEventNumInQueue(hDevice, &nEventNum);
```

7.4.55. GXExportConfigFile

Declaration:

GX_API GXExportConfigFile (GX_DEV_HANDLE hDevice, const char * pszFilePath)

Descriptions:

Export the current parameter of the camera to the configuration file.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*pszFilePath*	The path of the configuration file that to be generated.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
status = GXExportConfigFile(hDevice, "D://test.ini");
```

7.4.56. GXImportConfigFile

Declaration:

GX_API GXImportConfigFile (GX_DEV_HANDLE hDevice, const char * pszFilePath)

Descriptions:

Import the configuration file for the camera.

Formal parameter:

[in]*hDevice*	The handle of the device.
[in]*pszFilePath*	The path of the configuration file.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_HANDLE	The handle that the user introduces is illegal.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
status = GXImportConfigFile(hDevice, "D://test.ini");
```

7.4.57. GXGigEIpConfiguration

Declaration:

```
GX_API GXGigEIpConfiguration(const char* pszDeviceMacAddress,
                             GX_IP_CONFIGURE_MODE emIpConfigMode,
                             const char* pszIpAddress,
                             const char* pszSubnetMask,
                             const char* pszDefaultGateway,
                             const char* pszUserID);
```

Descriptions:

Configure the static IP address of the camera.

Formal parameter:

[in]*pszDeviceMacAddress*	The MAC address of the device.
[in]*emIpConfigModel*	IP Configuration.
[in]*pszIpAddress*	The IP address to be set.
[in]*pszSubnetMask*	The subnet mask to be set.
[in]*pszDefaultGateway*	The default gateway to be set.
[in]*pszUserID*	The user-defined name to be set.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.
GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_PARAMETER	The parameter is invalid.
GX_STATUS_NOT_FOUND_DEVICE	Can not found the device.
GX_STATUS_ERROR	The operation is failed.
GX_STATUS_INVALID_ACCESS	Access denied.
GX_STATUS_TIMEOUT	The operation is timed out.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code Sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
status = GXGigEIpConfiguration(szMAC, emIpConfigureMode,
                               szIpAddress,
                               szSubnetMask, szDefaultGateway,
                               szUserID);
```

7.4.58. GXGigEForceIp

Declaration:

```
GX_API GXGigEForceIp(const char* pszDeviceMacAddress,
                     const char* pszIpAddress,
                     const char* pszSubnetMask,
                     const char* pszDefaultGateway);
```

Descriptions:

Execute the Force IP.

Formal parameter:

[in]*pszDeviceMacAddress*	The MAC address of the device.
[in]*pszIpAddress*	The IP address to be set.
[in]*pszSubnetMask*	The subnet mask to be set.
[in]*pszDefaultGateway*	The default gateway to be set.

Return value:

GX_STATUS_SUCCESS	The operation is successful and no error occurs.

GX_STATUS_NOT_INIT_API	The GXInitLib initialization library is not called.
GX_STATUS_INVALID_PARAMETER	The parameter is invalid.
GX_STATUS_NOT_FOUND_DEVICE	Can not found the device.
GX_STATUS_ERROR	The operation is failed.
GX_STATUS_INVALID_ACCESS	Access denied.
GX_STATUS_TIMEOUT	The operation is timed out.

The errors that are not covered above please reference *GX_STATUS_LIST*.

Code Sample:

```
GX_STATUS status = GX_STATUS_SUCCESS;
status = GXGigEForceIp(szMAC, szIpAddress, szSubnetMask,
                       szDefaultGateway);
```

8. Image Processing Interface Description

The image processing interface mainly includes image pixel format conversion interface and color quality enhancement interface. The prototype of the image processing interface function is declared in the DxImageProc.h file.

8.1. Type

8.1.1. Data Type

Name	Description
VxInt8	8-bit signed integer
VxInt16	16-bit signed integer
VxInt32	32-bit signed integer
VxInt64	64-bit signed integer
VxUint8	8-bit unsigned integer
VxUint16	16-bit unsigned integer
VxUint32	32-bit unsigned integer

8.2. Constant

8.2.1. Status code

```
typedef enum    tagDX_STATUS
{
    DX_OK                                = 0,
    DX_PARAMETER_INVALID                 = -101,
    DX_PARAMETER_OUT_OF_BOUND            = -102,
    DX_NOT_ENOUGH_SYSTEM_MEMORY          = -103,
    DX_NOT_FIND_DEVICE                   = -104,
    DX_STATUS_NOT_SUPPORTED              = -105,
    DX_CPU_NOT_SUPPORT_ACCELERATE        = -106
} DX_STATUS;
```

Name	Description
DX_OK	Successful operation
DX_PARAMETER_INVALID	Invalid input parameter
DX_PARAMETER_OUT_OF_BOUND	The parameter is out of bound
DX_NOT_ENOUGH_SYSTEM_MEMORY	The system does not have enough memory
DX_NOT_FIND_DEVICE	No device found
DX_STATUS_NOT_SUPPORTED	The format is not supported.
DX_CPU_NOT_SUPPORT_ACCELERATE	The CPU does not support acceleration.

8.2.2. Pixel Bayer format

```
typedef enum tagDX_PIXEL_COLOR_FILTER
{
    NONE     = 0,
    BAYERRG = 1,
    BAYERGB = 2,
    BAYERGR = 3,
    BAYERBG = 4
} DX_PIXEL_COLOR_FILTER;
```

Name	Description
NONE	Non-Bayer format
BAYERRG	The first line of the Raw image starts with RG
BAYERGB	The first line of the Raw image starts with GB
BAYERGR	The first line of the Raw image starts with GR
BAYERBG	The first line of the Raw image starts with BG

8.2.3. Bayer conversion type

```
typedef enum tagDX_BAYER_CONVERT_TYPE
{
    RAW2RGB_NEIGHBOUR = 0,
    RAW2RGB_ADAPTIVE = 1,
    RAW2RGB_NEIGHBOUR3 = 2
} DX_BAYER_CONVERT_TYPE;
```

Name	Description
RAW2RGB_NEIGHBOUR	Neighborhood average interpolation algorithm
RAW2RGB_ADAPTIVE	Edge adaptive interpolation algorithm
RAW2RGB_NEIGHBOUR3	Neighborhood average interpolation algorithm for larger regions

8.2.4. Valid data bit

```
typedef enum tagDX_VALID_BIT
{
    DX_BIT_0_7      = 0,
    DX_BIT_1_8      = 1,
    DX_BIT_2_9      = 2,
    DX_BIT_3_10         = 3,
    DX_BIT_4_11         = 4
} DX_VALID_BIT;
```

If the original Raw image is 8 bits (0~7), you can only select the DX_BIT_0_7 algorithm; If the original image is 10 bits (0~9), when you conversion it to 8 bits, you can select the DX_BIT_0_7, DX_BIT_1_8 and DX_BIT_2_9 algorithms; If the original image is 12 bits (0~11), when you conversion it to 8 bits, you can select the DX_BIT_0_7, DX_BIT_1_8, DX_BIT_2_9, DX_BIT_3_10 and DX_BIT_4_11 all the five algorithms.

Name	Description
DX_BIT_0_7	Takes the 0~7bit
DX_BIT_1_8	Takes the 1~8bit
DX_BIT_2_9	Takes the 2~9bit
DX_BIT_3_10	Takes the 3~10bit
DX_BIT_4_11	Takes the 4~11bit

8.2.5. The actual image bit depth

```
typedef enum tagDX_ACTUAL_BITS
{
    DX_ACTUAL_BITS_10 = 10,
    DX_ACTUAL_BITS_12 = 12,
    DX_ACTUAL_BITS_14 = 14,
    DX_ACTUAL_BITS_16 = 16
} DX_ACTUAL_BITS;
```

The actual bit depth of the image data, which is the number of bytes occupied by per pixel.

Name	Description
DX_ACTUAL_BITS_10	10 bits
DX_ACTUAL_BITS_12	12 bits
DX_ACTUAL_BITS_14	14 bits
DX_ACTUAL_BITS_16	16 bits

8.2.6. The image mirror and flip type

```
typedef enum DX_IMAGE_MIRROR_MODE
{
    HORIZONTAL_MIRROR = 0,
    VERTICAL_MIRROR     = 1
}DX_IMAGE_MIRROR_MODE;
```

Name	Description
HORIZONTAL_MIRROR	Horizontal mirror
VERTICAL_MIRROR	Vertical mirror

8.3. Structure

8.3.1. Monochrome image process function set structure

```
typedef struct MONO_IMG_PROCESS
{
    bool        bDefectivePixelCorrect;
    bool        bSharpness;
    bool        bAccelerate;
    float fSharpFactor;
    VxUint8     *pProLut;
    VxUint16    nLutLength;
    VxUint8     arrReserved[32];
} MONO_IMG_PROCESS;
```

Input the parameters by the DxMono8ImgProcess interface.

Name	Description
bDefectivePixelCorrect	Dead pixel correction switch
bSharpness	Sharpening switch
bAccelerate	The enabling state of an accelerated function
fSharpFactor	Sharpening intensity factor
pProLut	Look-up table buffer pointer
nLutLength	The length of the look-up table buffer
arrReserved[32]	Reserved 32 bytes

8.3.2. Color image process function set structure

```
typedef struct COLOR_IMG_PROCESS
{
    bool                    bDefectivePixelCorrect;
    bool                    bDenoise;
    bool                    bSharpness;
    bool                    bAccelerate;
    VxInt16                 *parrCC;
    VxUint8                 nCCBufLength;
    Float                   fSharpFactor;
    VxUint8                 *pProLut;
    VxUint16                nLutLength;
    DX_BAYER_CONVERT_TYPEcvType;
    DX_PIXEL_COLOR_FILTER   emLayOut;
    bool                    bFlip;
    VxUint8                 arrReserved[32];
} COLOR_IMG_PROCESS;
```

Input the parameters by the DxRaw8ImgProcess interface.

Name	Description
bDefectivePixelCorrect	Dead pixel correction switch
bDenoise	Noise reduction switch
bSharpness	Sharpening switch
bAccelerate	The enabling state of an accelerated function
*parrCC	Color processing parameter array address
nCCBufLength	The length of parrCC (sizeof (VxInt16) *9)
fSharpFactor	Sharpening intensity factor
pProLut	Look-up table buffer pointer
nLutLength	The length of the look-up table buffer
cvType	Interpolation method
emLayOut	BAYER format
bFlip	Flip sign
arrReserved[32]	Reserved 32 bytes

8.4. Interfaces

8.4.1. DxRaw12PackedToRaw16

Declaration:

VxInt32 DHDECL DxRaw12PackedToRaw16 (void* pInputBuffer, void* pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight)

Descriptions:

This function is used to convert the Raw12 Packed format data to Raw16 format.

Formal parameter:

pInputBuffer	Point to the 12 bits Packed data buffer of the original image.
pOutputBuffer	Point to the data buffer of the target image, it needs to be new, not to use the original data buffer, and the size is image width * image height * 2.
nWidth	Image width.
nHeight	Image height.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Applies for the output buffer.
BYTE *pRaw16Buf = new BYTE[nWidth * nHeight * 2];
if(pRaw16Buf == NULL)
{
    return ;
}
else
{

   // Initializes the buffer.
```

```
    memset(pRaw16Buf,0,nWidth * nHeight * 2 * sizeof(BYTE));
}

VxInt32 DxStatus = DxRaw12PackedToRaw16(pPackedRaw12Buf,pRaw16Buf,nWidth,nHeight);
if (DxStatus != DX_OK)
{
    if (pRaw16Buf != NULL)
    {
        delete []pRaw16Buf;
        pRaw16Buf = NULL;
    }
    return ;
}

// Processing the data of the 16 bits raw image.
// ……

// Processing is complete.
if(pRaw16Buf != NULL)
{
    delete []pRaw16Buf;
    pRaw16Buf = NULL;
}
```

8.4.2. DxRaw10PackedToRaw16

Declaration:

VxInt32 DHDECL DxRaw10PackedToRaw16 (void* pInputBuffer, void* pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight)

Descriptions:

This function is used to convert the Raw10 Packed format data to Raw16 format data.

Formal parameter:

pInputBuffer	Point to the 10 bits Packed data buffer of the original image.
pOutputBuffer	Point to the data buffer of the target image, it needs to be new, not to use the original data buffer, and the size is image width * image height * 2.
nWidth	Image width.
nHeight	Image height.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Applies for the output buffer.
BYTE *pRaw16Buf = new BYTE[nWidth * nHeight * 2];
if(pRaw16Buf == NULL)
{
    return ;
}
else
{
    // Initializes the buffer.
    memset(pRaw16Buf,0,nWidth * nHeight * 2 * sizeof(BYTE));
}

VxInt32 DxStatus = DxRaw10PackedToRaw16(pPackedRaw10Buf,pRaw16Buf,nWidth,nHeight);
if (DxStatus != DX_OK)
{
    if (pRaw16Buf != NULL)
    {
        delete []pRaw16Buf;
        pRaw16Buf = NULL;
    }

    return ;
}

// Processing the data of the 16 bits raw image.
// ..................

// Processing is complete.
if(pRaw16Buf != NULL)
{
    delete []pRaw16Buf;
    pRaw16Buf = NULL;
}
```

8.4.3. DxRaw8toRGB24

Declaration:

VxInt32 DHDECL DxRaw8toRGB24 (void *pInputBuffer, void *pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight, DX_BAYER_CONVERT_TYPE cvtype, DX_PIXEL_COLOR_FILTER nBayerType, bool bFlip)

Descriptions:

This function is used to convert the Bayer image to RGB image.

Formal parameter:

pInputBuffer	Point to the 8 bits data buffer of the original image.
pOutputBuffer	Point to the data buffer (RGB data) of the target image, and the size is image width * image height * 3.
nWidth	Image width.
nHeight	Image height.
cvtype	The type of conversion algorithm.
nBayerType	The type of Bayer image format.
bFlip	If flip, the value is **true**, and the image will vertical flip; otherwise, the value is **false**, not flip.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Applies for the RGB data of the output image.
BYTE *pRGB24Buf = new BYTE[nWidth * nHeight * 3];

if (pRGB24Buf == NULL)
{
    return ;
}
else
{
    // Initializes the buffer.
    memset(pRGB24Buf,0,nWidth * nHeight * 3 * sizeof(BYTE));
}

// Selects the interpolation algorithm.
DX_BAYER_CONVERT_TYPE cvtype    = RAW2RGB_NEIGHBOUR;
DX_PIXEL_COLOR_FILTER nBayerType = BAYERRG; // Selects the image Bayer format.
bool bFlip = true;

VxInt32 DxStatus = DxRaw8toRGB24(pRaw8Buf,pRGB24Buf,nWidth,
                                 nHeight,cvtype,nBayerType,bFlip);
if (DxStatus != DX_OK)
{
    if (pRGB24Buf != NULL)
    {
        delete []pRGB24Buf;
        pRGB24Buf = NULL;
    }
```

```
    return ;
}

// Processing the data of the 24 bits RGB image.
.................

if (pRGB24Buf != NULL)
{
    delete []pRGB24Buf;
    pRGB24Buf = NULL;
}
```

8.4.4. DxRotate90CW8B

Declaration:

VxInt32 DHDECL DxRotate90CW8B (void* pInputBuffer, void* pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight)

Descriptions:

The function will rotate the 8 bits image 90 degrees clockwise. For Bayer image, the Bayer format will be changed after the rotation. For example, rotate the raw image of BAYER_RG type clockwise, it becomes a BAYER_GR type. After rotation, the width and height of the image are equal to those of the original one.

Formal parameter:

pInputBuffer	Point to the data buffer of the original image.
pOutputBuffer	Point to the data buffer of the target image.
nWidth	The width of original image.
nHeight	The height of original image.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Rotates the Raw image in BAYER_RG format with a width of 1600 and a height
// of 1234.
BYTE* pTemp = new BYTE[nWidth * nHeight];
if(pTemp == NULL)
{
    return;
}
else
{
    memset(pTemp,0,nWidth * nHeight * sizeof(BYTE));
}

// Rotates 90 degrees clockwise.
VxInt32 DxStatus = DxRotate90CW8B(pRaw8Buf,pTemp,nWidth,nHeight);
if (DxStatus != DX_OK)
{
    if(pTemp != NULL)
    {
        delete []pTemp;
        pTemp = NULL;
```

```
        }

        return ;
    }
    else
    {
       // Copies to the Raw image buffer.
        memcpy(pRaw8Buf,pTemp,nWidth * nHeight * sizeof(BYTE));
    }

    if(pTemp != NULL)
    {
        delete []pTemp;
        pTemp = NULL;
    }
    // At this point, the bayer type in pRaw8Buf is BAYER_GR. The width of the
    // image is 1234 and the height is 1600.
```

8.4.5. DxRotate90CCW8B

Declaration:

VxInt32 DHDECL DxRotate90CCW8B (void* pInputBuffer, void* pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight)

Descriptions:

The function will rotate the 8 bits gray image 90 degrees counter clockwise. For Bayer image, the Bayer format will be changed after the rotation. For example, rotate the raw image of BAYER_RG type clockwise, it becomes a BAYER_GB type. After rotation, the width and height of the image are equal to those of the original one.

Formal parameter:

pInputBuffer	Point to the data buffer of the original image.
pOutputBuffer	Point to the data buffer of the target image.
nWidth	The width of original image.
nHeight	The height of original image.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
    // Rotates the Raw image in BAYER_RG format with a width of 1600 and a height
    // of 1234.
    BYTE* pTemp = new BYTE[nWidth * nHeight];
    if(pTemp == NULL)
    {
        return;
    }
    else
    {
        memset(pTemp,0,nWidth * nHeight * sizeof(BYTE));
    }

  // Rotates 90 degrees counter clockwise.
    VxInt32 DxStatus = DxRotate90CCW8B(pRaw8Buf,pTemp,nWidth,nHeight);
    if (DxStatus != DX_OK)
```

```
    {
        if(pTemp != NULL)
        {
            delete []pTemp;
            pTemp = NULL;
        }
        return ;
    }
    else
    {
    // Copies to the Raw image buffer.
    memcpy(pRaw8Buf,pTemp,nWidth * nHeight * sizeof(BYTE));
    }
    if(pTemp != NULL)
    {
        delete []pTemp;
        pTemp = NULL;
    }
    // At this point, the bayer type in pRaw8Buf is BAYER_GR. The width of the
    // image is 1234 and the height is 1600.
```

8.4.6. DxBrightness

Declaration:

VxInt32 DHDECL DxBrightness (void* pInputBuffer, void* pOutputBuffer, VxUint32 nImagesize, VxInt32 nFactor)

Descriptions:

The function will adjust the brightness of the input images, and the input images are 24 bits RGB or 8 bits gray image.

Formal parameter:

pInputBuffer	Point to the data buffer of the original image.
POutputBuffer	Point to the data buffer of the target image.
nImagesize	The buffer length of input images, unit: byte. (For RGB images, the size equal the image width * image height* 3).
nFactor	The factor of brightness adjustment, range of value: -150~150. 0: The brightness has not changed. > 0: Increase the brightness. < 0: Reduces the brightness.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample

```
    // Adjusts the brightness of the 24bitRGB image.
    VxInt32 DxStatus = DxBrightness((BYTE*)pRGBBuf,pRGBBuf,nWidth * nHeight * 3,50);
    if (DxStatus != DX_OK)
    {
        return ;
    }
    // Adjusts the brightness of the 8 bits gray image.
    VxInt32 DxStatus = DxBrightness((BYTE*)pYbuf,pYbuf,nWidth * nHeight,50);
    if (DxStatus != DX_OK)
    {
```

```
        return ;
    }
```

Effect images：

Figure 55 shows an image with no brightness adjustment. Figure 56 shows the image after brightness adjustment.

Figure 55: Before brightness adjustment

Figure 56: After brightness adjustment

8.4.7. DxContrast

Declaration:

VxInt32 DHDECL DxContrast (void* pInputBuffer, void* pOutputBuffer, VxUint32 nImagesize, VxInt32 nFactor)

Descriptions:

The function will adjust the contrast of the input images, and the input images are 24 bits RGB or 8 bits gray image.

Formal parameter:

pInputBuffer	Point to the data buffer of the original image.
pOutputBuffer	Point to the data buffer of the target image.
nImagesize	The buffer length of input images, unit: byte. (For RGB images, the size equal the image width * image height * 3).
nFactor	The factor of contrast adjustment, range of value: -50~100. 0: The contrast has not changed. > 0: Increase the contrast. < 0: Reduces the contrast.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Adjusts the contrast of the 24bitRGB image.
VxInt32 DxStatus = HVContrast((BYTE*)pRGBBuf,pRGBBuf,nWidth * nHeight * 3,50);
if (DxStatus != DX_OK)
{
    return ;
}

// Adjusts the contrast of the 8 bits gray image.
VxInt32 DxStatus = HVContrast((BYTE*)pYbuf,pYbuf,nWidth * nHeight,50);
if (DxStatus != DX_OK)
{
    return ;
}
```

Effect images：

Figure 57 shows an image with no contrast adjustment. Figure 58 shows the image after contrast adjustment.

Figure 57: Before contrast adjustment

Figure 58: After contrast adjustment

8.4.8. DxSharpen24B

Declaration:

VxInt32 DHDECL DxSharpen24B (void* pInputBuffer, void* pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight, float fFactor)

Descriptions:

The function will sharpen the input images, and the input images are 24bitRGB images.

Formal parameter:

pInputBuffer	Point to the data buffer of the original image.
pOutputBuffer	The buffer of the RGB image that after being sharpened.
nWidth	The width of the image.
nHeight	The height of the image.
fFactor	The factor of sharpness adjustment, range: 0.1~5.0.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Sharpens the 24bitRGB image.
VxInt32 DxStatus = HVSharpen24B((BYTE*)pRGBBuf,
                                (BYTE*)pRGBBuf,nWidth,nHeight,2.0);
if (DxStatus != DX_OK)
{
    return ;
}
```

Effect images：

Figure 59 shows an image with no sharpen adjustment. Figure 60 shows the image after sharpen adjustment.

Figure 59: Original image

Figure 60: After sharpen adjustment

8.4.9. DxSaturation

Declaration:

VxInt32 DHDECL DxSaturation (void* pInputBuffer, void* pOutputBuffer, VxUint32 nImagesize, VxInt32 nFactor)

Descriptions:

The function is used to adjust the saturation of the input image, and the input image is 24bitRGB image.

Formal parameter:

pInputBuffer	Point to the data buffer of the original image.
pOutputBuffer	The buffer of the image that after saturation adjustment.
nImagesize	The buffer length of the image, unit: byte. (For RGB image, the buffer length is equal image width * image height)
nFactor	The factor of saturation adjustment, range: 0~128. 64: There's no change in saturation. > 64: Increase the saturation. < 64: Reduce the saturation. 128: The saturation is twice as the current value. 0: It is a monochrome image.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Adjusts the saturation of the 24bitRGB image.
VxInt32 DxStatus = DxSaturation((BYTE*)pRGBBuf,pRGBBuf,nWidth * nHeight,90);
if (DxStatus != DX_OK)
{
    return ;
}
```

Effect images：

Figure 61 shows an image with no saturation adjustment. Figure 62 shows the image after saturation adjustment.

Figure 61: Before saturation adjustment

Figure 62：After saturation adjustment

8.4.10. DxGetWhiteBalanceRatio

Declaration:

VxInt32 DHDECL DxGetWhiteBalanceRatio (void *pInputBuffer, VxUint32 nWidth, VxUint32 nHeight, double* dRatioR, double* dRatioG, double* dRatioB)

Descriptions:

This function is used to get the white balance ratio. The input is 24 bits RGB image and output is white balance ratio. In order to calculate accurately, the input image should be objective "white" area.

Formal parameter:

pInputBuffer	"White" image data buffer.
nWidth	The image width.
nHeight	The image height.
dRatioR	The white balance ratio of the red component.
dRatioG	The white balance ratio of the green component.
dRatioB	The white balance ratio of the blue component.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
double dRatioR = 1.0;
double dRatioG = 1.0;
double dRatioB = 1.0;

// Calculates the white balance coefficient.
VxInt32 DxStatus = DxGetWhiteBalanceRatio((BYTE*)pRGBBuf,nWidth,nHeight,
                                          &dRatioR,&dRatioG,&dRatioB);
if (DxStatus != DX_OK)
{
   return ;
}
```

Effect images：

Figure 63 shows an RGB image with no white balance adjustment:

Figure 63: An RGB image with no white balance adjustment

1) If use this function to get the white balance ratio, to be accurate, you should give an objective "white" region sub-image to the function in the RGB image. As shown in the frame area in Figure 64. (The actual object in the area is the white building block). The image after white balance adjustment is shown in Figure 65.
2) The use method: (pRGBBuf is the buffer of the sub-image in the frame area in Figure 64, nWidth is the sub-image width, nHeight is the sub-image height), it needs to create a new buffer for the sub-image and then pass the new buffer to DxGetWhiteBalanceRatio.

Figure 64: The "white" region selected

After calculating the white balance ratio of the three components of R/G/B, the white balance operation method is as follows:

```
// Adds macro definitions to the header file.

//<Determines the range of bit data.
#define CLIP8(a)        (((a) & 0xFFFFFF00) ? (((a) < 0) ? 0 : 255 ) : (a))

int i    = 0;
int j    = 0;
int nPos = 0;

// Whites balance look-up table.
BYTE  arrRLut[256];
BYTE  arrGlut[256];
BYTE  arrBLut[256];

// Defines R, G, and B pixel.
int  R = 0;
int  G = 0;
int  B = 0;

// Calculates the look-up table for white balance coefficients.
for(i = 0; i < 256; i++)
{
   // Calculates the pixel values after white balance processing.
   R = i * dRatioR;
   G = i * dRatioG;
   B = i * dRatioB;

   arrRLut[i] = CLIP8(R);
   arrGlut[i] = CLIP8(G);
```

```
        arrBLut[i] = CLIP8(B);
    }

    // White balance processing.
     for (i = 0; i < nHeight; i++)
    {
       for(j = 0; j < nWidth; j++)
      {
          nPos = 3 * i * nWidth + 3 * j;
          pRGBBuf[nPos + 0] = arrRLut[pRGBBuf[nPos + 0]];
          pRGBBuf[nPos + 1] = arrGLut[pRGBBuf[nPos + 1]];
          pRGBBuf[nPos + 2] = arrBLut[pRGBBuf[nPos + 2]];
       }
    }
```

Figure 65: An image after white balance adjustment

8.4.11. DxAutoRawDefectivePixelCorrect

Declaration:

VxInt32 DHDECL DxAutoRawDefectivePixelCorrect (void* pRawImgBuf, VxUint32 nWidth, VxUint32 nHeight, VxInt32 nBitNum)

Descriptions:

The function automatically detects and corrects the bad pixel of the raw image in real time. The input image can be 8 bits raw image or 16bits raw image. This function supports raw images in Bayer format. The nBitNum is the actual bit depth of the data, if the image is 8 bits raw, the value is 8, and if the image is 16bits raw, then the value is the actual bit depth of the data. For example, a raw image of a 12 bits data, it takes two bytes (16bits), but the actual bit depth is 12, then the value is 12. This function does not support the raw image of Packet format. For the Packet format, you should use the DxRaw10PackedToRaw16 function or DxRaw10PackedToRaw16 function to convert it into raw16 format first. Due to the real-time detection and correction, every image must be checked and corrected when this function is turned on.

Formal parameter:

pRawImgBuf	The input buffer of Raw image (8 bits or 16bitRaw image).
nWidth	The width of the image.
nHeight	The height of the image.
nBitNum	The actual bit depth of data.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Determines whether to open the automatic dead pixel correction. pPaw16Buf
// is the 16 bits raw image.
if(bAutoCorrect)
{
  VxInt32 DxStatus  = DxAutoRawDefectivePixelCorrect(pPaw16Buf,nWidth,nHeight,12);
   if (DxStatus != DX_OK)
   {
        if (pRaw16Buf != NULL)
         {
            delete []pRaw16Buf;
            pRaw16Buf = NULL;
          }

          return ;
   }
}

// Processes Raw images.
// ..................
```

8.4.12. DxRaw16toRaw8

Declaration:

VxInt32 DHDECL DxRaw16toRaw8 (void *pInputBuffer, void *pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight, DX_VALID_BIT nValidBits);

Descriptions:

The function converts raw16 images (the actual bit depth is 16, effective bit depth is 10 or 12) to raw8 images (the actual bit depth and valid bit depth are 8 bits).

Formal parameter:

pInputBuffer	The data buffer of original image.
pOutputBuffer	The data buffer of target image.
nWidth	The width of the image.
nHeight	The height of the image.
nValidBits	The valid bit of data.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
BYTE *pRaw8Buf = new BYTE[nWidth * nHeight]; // Applies for the output buffer.
if(pRaw8Buf == NULL)
{
    return;
}
else
{
    //Initializes the buffer.
    memset(pRaw8Buf,0,nWidth * nHeight * sizeof(BYTE));
}

VxInt32 DxStatus = DxRaw16toRaw8(pRaw16Buf,pRaw8Buf,nWidth,nHeight, DX_BIT_4_11);
if (DxStatus != DX_OK)
{
    if (pRaw8Buf != NULL)
    {
        delete []pRaw8Buf;
        pRaw8Buf = NULL;
    }

    return;
}

// Processing the data of the 8 bits raw image.
// ..................

// Processing is complete.
if(pRaw8Buf != NULL)
{
    delete []pRaw8Buf;
    pRaw8Buf = NULL;
}
```

8.4.13. DxRGB48toRGB24

Declaration:

VxInt32 DHDECL DxRGB48toRGB24 (void *pInputBuffer, void *pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight, DX_VALID_BIT nValidBits);

Descriptions:

The function converts RGB 48 bits color image data to RGB 24 bits image data, because the three channels process the image at the same time, each channel will convert the 16 bits (the valid bit depth may be 12 bits, 10 bits, etc.) data to 8 bits data.

Formal parameter:

pInputBuffer	The data buffer of original image (the buffer size is nWidth * nHeight * 3 * 2 BYTE).
pOutputBuffer	The data buffer of target image (the buffer size is nWidth * nHeight * 3 * BYTE).
nWidth	The width of the image.
nHeight	The height of the image.
nValidBits	The valid bit of data.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Applies for the output buffer.
BYTE *pRGB24Buf = new BYTE[nWidth * nHeight * 3];
if(pRGB24Buf == NULL)
{
    return ;
}
else
{
    // Initializes the buffer.
    memset(pRGB24Buf,0,nWidth * nHeight * 3 * sizeof(BYTE));
}

VxInt32 DxStatus = DxRGB48toRGB24(pRGB48Buf,pRGB24Buf,nWidth,nHeight,DX_BIT_4_11);
if (DxStatus != DX_OK)
{
    if (pRGB24Buf != NULL)
    {
        delete []pRGB24Buf;
        pRGB24Buf = NULL;
    }

    return ;
}

// Processing the data of the 8 bits raw image.
// ...................

// Processing is complete.
if(pRGB24Buf != NULL)
{
    delete []pRGB24Buf;
    pRGB24Buf = NULL;
}
```

8.4.14. DxRaw16toRGB48

Declaration:

VxInt32 DHDECL DxRaw16toRGB48 (void *pInputBuffer, void *pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight, DX_ACTUAL_BITS nActualBits, DX_BAYER_CONVERT_TYPE cvtype, DX_PIXEL_COLOR_FILTER nBayerType, bool bFlip)

Descriptions:

The function converts raw 16bits image (each pixel is 16 bits) to RGB 48 bits image data (each RGB component is 16 bits), and you can also use the function DxRGB48toRGB24 to convert RGB 48 bits image to RGB 24 bits image for display.

Formal parameter:

pInputBuffer	The data buffer of original image.
pOutputBuffer	The data buffer of target image.
nWidth	The width of the image.
nHeight	The height of the image.
nActualBits	The valid bit of data.
cvtype	The conversion algorithm type.
nBayerType	The type of Bayer image format.
bFlip	If flip, the value is **true**, and the image will vertical flip; otherwise, the value is **false**, not flip.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Outputs RGB image data.
BYTE *pRGB48Buf = new BYTE[nWidth * nHeight * 3 * 2];
if (pRGB48Buf == NULL)
{
    return ;
}
else
{
    // Initializes the buffer.
    memset(pRGB48Buf,0,nWidth * nHeight * 3 * 2 * sizeof(BYTE));
}

// Selects the interpolation algorithm.
DX_BAYER_CONVERT_TYPE cvtype     = RAW2RGB_NEIGHBOUR;
// Selects the image Bayer format.
DX_PIXEL_COLOR_FILTER nBayerType   = BAYERRG;
DX_ACTUAL_BITS    nActualBits  = DX_ACTUAL_BITS_16; // The actual bit width.
bool bFlip = true;

VxInt32 DxStatus = DxRaw16toRGB48(pRaw16Buf,pRGB48Buf,nWidth,nHeight,
                                  ActualBits,cvtype,nBayerType,bFlip);
if (DxStatus != DX_OK)
{
    if (pRGB48Buf != NULL)
    {
        delete []pRGB48Buf;
```

```
            pRGB48Buf = NULL;
        }

        return ;
    }

    // Processing RGB48 data.
    // ..................

    if (pRGB48Buf != NULL)
    {
        delete []pRGB48Buf;
        pRGB48Buf = NULL;
    }
```

8.4.15. DxGetContrastLut

Declaration:

VxInt32 DHDECL DxGetContrastLut (int nContrastParam, void *pContrastLut, int *pLutLength);

Descriptions:

This function is used to calculate the contrast look-up table for the input of image quality promotion function, and only 24 bits RGB images are supported.

Formal parameter:

nContrastParam	The parameter of contrast adjustment. You can get it by the GX_INT_CONTRAST_PARAM code of the GxIAPI library.
pContrastLut	Contrast look-up table.
pLutLength	The length of contrast look-up table, unit: byte;

If the length value inputted by the user less than the current actual length value, it will return errors, and the *pLutLength* returns the actual length value.

If the *pContrastLut* inputted by the user is NULL, it returns success, the *pLutLength* returns the actual length value.

If the operation succeeds, the *pLutLength* returns the actual length value.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Determines whether the current camera supports contrast acquisition.
GX_STATUSGxStatus = GXIsImplemented(hDevice, GX_INT_CONTRAST_PARAM,
                                    &bIsImplemented);
if (GxStatus!= GX_STATUS_SUCCESS)
{
    return;
}

if (bIsImplemented)
{
     //Gets the contrast adjustment parameter.
     GxStatus = GXGetInt (hDevice, GX_INT_CONTRAST_PARAM, &nContrastParam);
     if (GxStatus!= GX_STATUS_SUCCESS)
     {
       return;
     }
}
else
{
     nContrastParam = 0;
}

// Gets the length of the contrast look-up table.
VxInt32 DxStatus= DxGetContrastLut(nContrastParam, NULL, &nLutLength);
if(DxStatus != DX_OK)
{
    return;
}

// Applies memory for the contrast look-up table.
pContrastLut = new BYTE[nLutLength];
if (pContrastLut == NULL)
{
    return;
}
```

```
    //Calculates the contrast look-up table.
    DxStatus = DxGetContrastLut(nContrastParam, pContrastLut, &nLutLength);
     if (DxStatus != DX_OK)
    {
        if (pContrastLut!= NULL)
        {
            delete []pContrastLut;
            pContrastLut = NULL;
         }
        return;
    }

    // Image processing.
    // ……

    if (pContrastLut!= NULL)
    {
        delete []pContrastLut;
        pContrastLut= NULL;
    }
```

8.4.16. DxGetGammatLut

Declaration:

VxInt32 DHDECL DxGetGammatLut (double dGammaParam, void *pGammaLut, int *pLutLength);

Descriptions:

This function is used to calculate the Gamma look-up table for the input of image quality promotion function, and only 24 bits RGB images are supported.

Formal parameter:

dGammaParam	The parameter of Gamma adjustment. You can get it by the GX_FLOAT_GAMMA_PARAM code of the GxIAPI library.
pGammaLut	Gamma look-up table.
pLutLength	The length of Gamma look-up table, unit: byte;

If the length value inputted by the user less than the current actual length value, it will return errors, and the *pLutLength* returns the actual length value.

If the *pGammaLut* inputted by the user is NULL, it returns success, the *pLutLength* returns the actual length value.

If the operation succeeds, the *pLutLength* returns the actual length value.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Determines whether the current camera supports Gamma capture.
GX_STATUSGxStatus = GXIsImplemented(hDevice, GX_FLOAT_GAMMA_PARAM,
                                    &bIsImplemented);
if (GxStatus!= GX_STATUS_SUCCESS)
{
    return;
}

if (bIsImplemented)
{
    //Gets the Gamma adjustment parameter.
    GxStatus = GXGetFloat (hDevice, GX_FLOAT_GAMMA_PARAM, &dGammaParam);
    if (GxStatus!= GX_STATUS_SUCCESS)
    {
        return;
    }
}
else
{
     dGammaParam = 1;
}

// Gets the length of the Gamma look-up table.
VxInt32 DxStatus= DxGetGammaLut(dGammaParam, NULL, &nLutLength);
if (DxStatus != DX_OK)
{
    return;
}
// Applies memory for the Gamma look-up table.
pGammaLut= new BYTE[nLutLength];
if (pGammaLut== NULL)
{
    return;
}

//Calculates the Gamma look-up table.
```

```
    DxStatus = DxGetGammaLut(dGammaParam, pGammaLut, &nLutLength);
    if (DxStatus != DX_OK)
    {
        if (pGammaLut!= NULL)
        {
            delete []pGammaLut;
            pGammaLut= NULL;
        }

        return;
    }


    // Image processing
    // ……

    if (pGammaLut!= NULL)
    {
        delete []pGammaLut;
        pGammaLut= NULL;
    }
```

8.4.17. DxImageImprovment

Declaration:

VxInt32 DHDECL DxImageImprovment (void *pInputBuffer, void *pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight, VxInt64 nColorCorrectionParam, void *pContrastLut, void *pGammaLut);

Descriptions:

This function is used for the image quality promotion of the input images, and only 24bitRGB images are supported.

Formal parameter:

pInputBuffer	The data buffer of original image.
pOutputBuffer	The data buffer of target image.
nWidth	The width of the image.
nHeight	The height of the image.
nColorCorrectionParam	The color correction value and can be get by the GX_INT_COLOR_CORRECTION_PARAM code in the GxIAPI library, and also it can be set to 0.
pContrastLut	The contrast look-up table, which can be calculated by DxGetContrastLut function, is only calculated once, and can be set to NULL.
pGammaLut	The Gamma look-up table, which can be calculated by DxGetGammatLut function, is only calculated once, and can be set to NULL.

nColorCorrectionParam, pContrastLut, pGammaLut, different combinations can be carried out to achieve different effects. The combination is shown in Table 3:

Table 3

NO.	nColorCorrectionParam	pContrastLut	pGammaLut	Effect
1	≠ 0	≠ NULL	≠ NULL	Color correction, contrast, Gamma adjustment (**At this point, the image quality is the best**)
2	≠ 0	NULL	≠ NULL	Color correction, Gamma adjustment
3	≠ 0	≠ NULL	NULL	Color correction, contrast adjustment
4	0	≠ NULL	≠ NULL	Contrast, Gamma adjustment
5	≠ 0	NULL	NULL	Color correction
6	0	≠ NULL	NULL	Contrast adjustment
7	0	NULL	≠ NULL	Gamma adjustment

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code sample:

```
// Gets the contrast adjustment parameter value.
GX_STATUS GxStatus = GXGetInt (hDevice, GX_INT_CONTRAST_PARAM, &nContrastParam);
if (GxStatus != GX_STATUS_SUCCESS)
{
    return;
}

// Gets the adjustment parameter value of the color correction.
GxStatus = GXGetInt (hDevice, GX_INT_COLOR_CORRECTION_PARAM, &nColorCorrectionParam);
if (GxStatus != GX_STATUS_SUCCESS)
{
    return;
}

// Gets the Gamma adjustment parameter.
GxStatus = GXGetFloat (hDevice, GX_FLOAT_GAMMA_PARAM, &dGammaParam);
if (GxStatus != GX_STATUS_SUCCESS)
{
    return;
}

do
{
   // Gets the length of the Gamma look-up table.
   VxInt32 DxStatus = DxGetGammaLut(dGammaParam, NULL, &nLutLength);
   if (DxStatus != DX_OK)
   {
      break;
   }

   // Applies memory for the Gamma look-up table.
   pGammaLut = new BYTE[nLutLength];
   if (pGammaLut == NULL)
   {
       DxStatus = DX_NOT_ENOUGH_SYSTEM_MEMORY;
       break;
   }

   // Calculates the Gamma look-up table.
   DxStatus = DxGetGammaLut(dGammaParam, pGammaLut, &nLutLength);
   if (DxStatus != DX_OK)
   {
       break;
   }

   // Gets the length of the contrast look-up table.
   DxStatus = DxGetContrastLut(nContrastParam, NULL, &nLutLength);
   if (DxStatus != DX_OK)
   {
      break;
   }
```

```
    // Applies memory for the contrast look-up table.
    pContrastLut = new BYTE[nLutLength];
    if (pContrastLut == NULL)
    {
        DxStatus = DX_NOT_ENOUGH_SYSTEM_MEMORY;
        break;
    }

    // Calculates the contrast look-up table.
    DxStatus = DxGetContrastLut(nContrastParam, pContrastLut, &nLutLength);
    if (DxStatus != DX_OK)
    {
       break;
    }
}while(0);

// Sets look-up table failed, and then release the resource.
if (nStatus != DX_OK)
{
    if (pGammaLut != NULL)
    {
        delete[] m_pGammaLut;
        pGammaLut = NULL;
    }
    if (pContrastLut != NULL)
    {
       delete[] pContrastLut;
       pContrastLut = NULL;
    }
    return;
}

// Improves the quality of the image.
DxStatus = DxImageImprovment(pInputBuffer,pOutputBuffer,nWidth,nHeight,
                             nColorCorrectionParam, pContrastLut, pGammaLut);

if (pGammaLut!= NULL)
{
    delete []pGammaLut;
    pGammaLut = NULL;
}

if (pContrastLut!= NULL)
{
    delete []pContrastLut;
    pContrastLut = NULL;
}
```

Effect images：

Figure 66 shows an image with no quality promotion. Figure 67 shows the image after quality promotion.

Figure 66: An image with no quality promotion

Figure 67: An image after quality promotion

8.4.18. DxImageMirror

Declaration:

VxInt32 DHDECLDxImageMirror (void *pInputBuffer, void *pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight, DX_IMAGE_MIRROR_MODE emMirrorMode)

Descriptions:

This function is used to generate a mirror image of the original image in the horizontal or vertical direction, and the input image is 8 bits raw image or 8 bits monochrome image.

Formal parameter:

pInputBuffer	The data buffer of original image.
pOutputBuffer	The data buffer of target image.
nWidth	The width of the image.
nHeight	The height of the image.
emMirrorMode	The mode of image mirror, HORIZONTAL_MIRROR and VERTICAL_MIRROR, that is horizontal mirror flip and vertical mirror flip.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code Sample:

```
// Horizontal mirror an image. (Inputs and outputs can not be the same
// buffer).
 VxInt32DxStatus=DxImageMirror((BYTE*)pIn8BitBuf,
                               (BYTE*)pOut8BitBuf, nWidth, nHeight,
                               HORIZONTAL_MIRROR);
if (DxStatus != DX_OK)
{
   return ;
}

 // Vertical mirror an image. (Inputs and outputs can not be the same
 // buffer).
 VxInt32DxStatus=DxImageMirror((BYTE*)pIn8BitBuf,
                               (BYTE*)pOut8BitBuf, nWidth, nHeight,
                                VERTICAL_MIRROR);
if (DxStatus != DX_OK)
{
   return ;
}
```

Effect images:

Figure 68 shows an 8 bits monochrome image. Figure 69 shows the image after horizontal flip. Figure 70 shows the image after vertical flip.

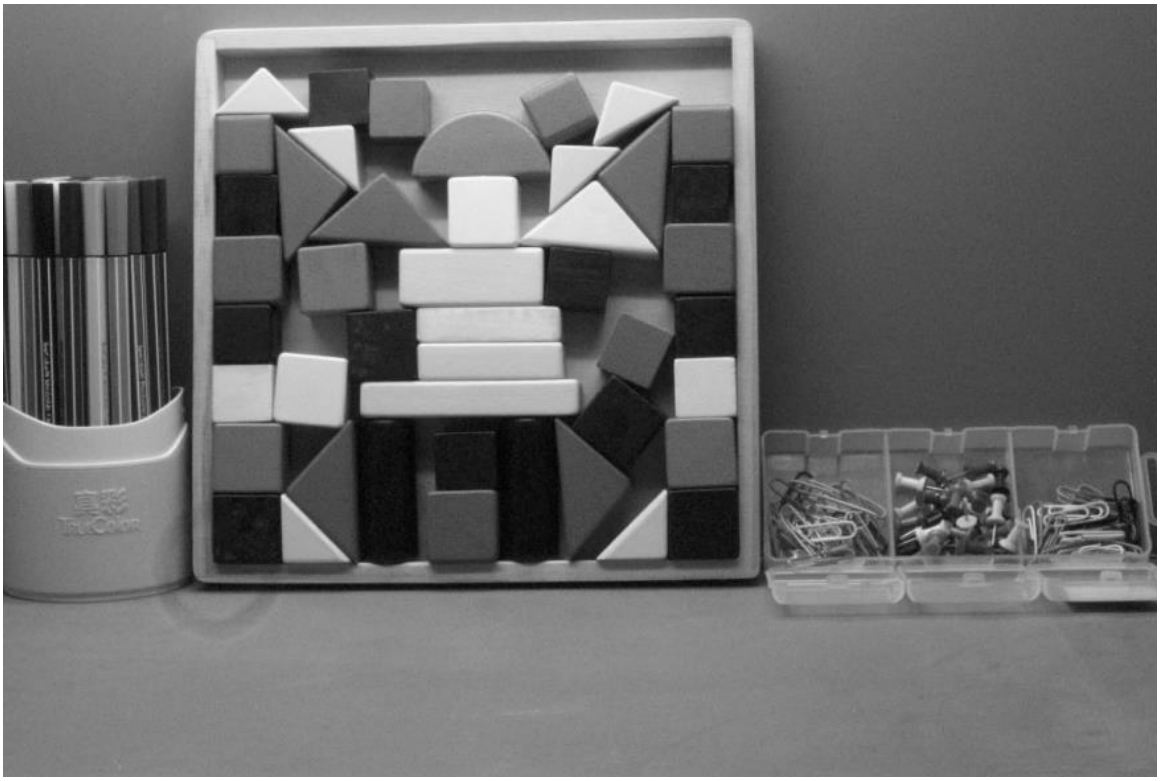

Figure 68: Original image

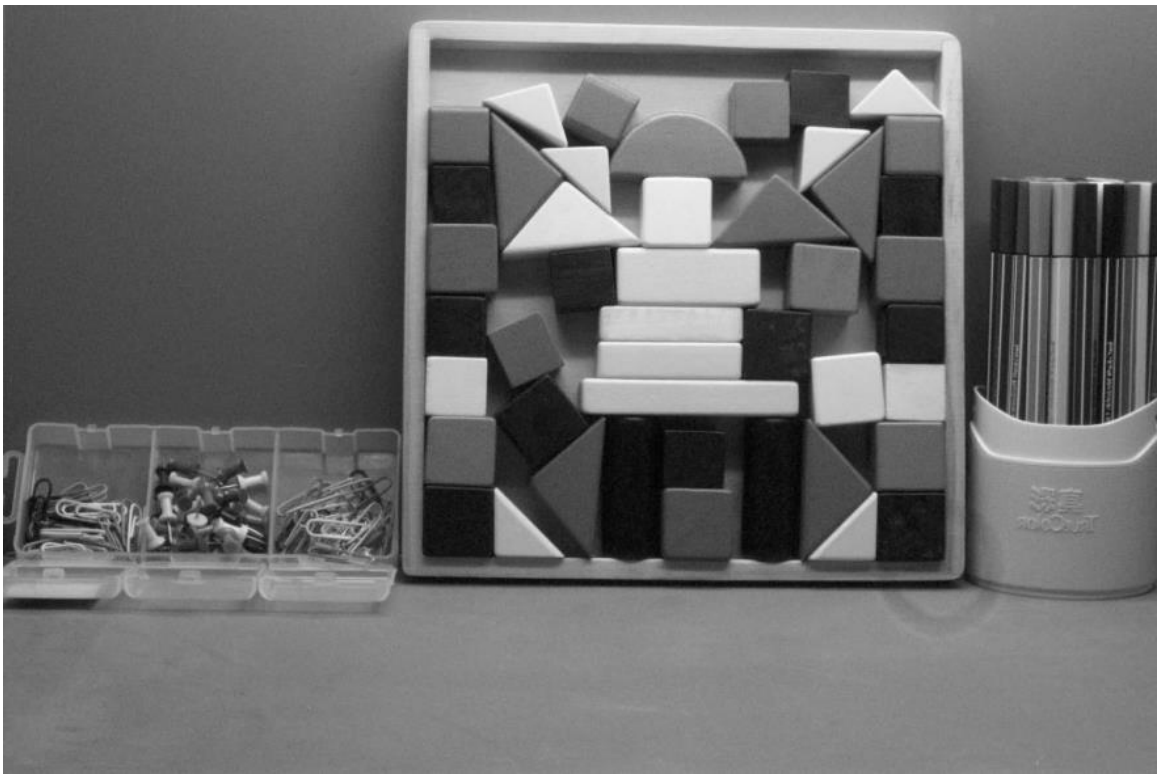

Figure 69: The image after horizontal flip

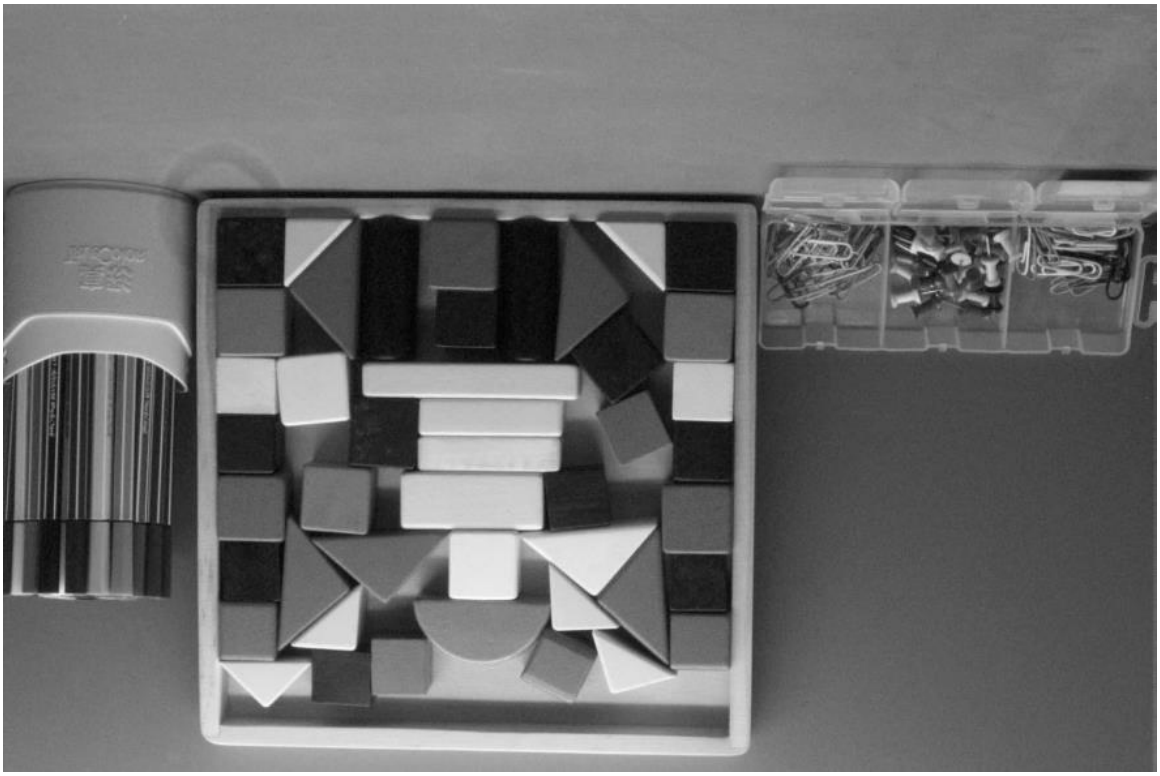

Figure 70: The image after vertical flip

8.4.19. DxGetLut

Declaration:

VxInt32 DHDECL DxGetLut (VxInt32 nContrastParam, double dGamma, VxInt32 nLightness, VxUint8 *pLut, VxUint16 *pLutLength);

Descriptions:

The function is used to calculate the 8bit look-up table of image processing.

Formal parameter:

nContrastParam	The parameter of contrast adjustment. Range: -50~100
nGamma	The parameter of Gamma adjustment. Range: 0.1~10
nLightness	The parameter of lightness adjustment. Range: -150~150
pLut	Look-up table. If the contrast, Gamma or lightness parameters change, it needs to be recalculated.
pLutLength	The length of look-up table, unit: byte.

If the length value inputted by the user not equal to the current actual length value, it will return errors, and the *pLutLength* returns the actual length value.

If the *pLut* inputted by the user is NULL, it returns success, the *pLutLength* returns the actual length value.

If the operation succeeds, the *pLutLength* returns the actual length value.

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code Sample:

```
// Gets the length of the look-up table.
VxInt32 DxStatus= DxGetLut (nContrastParam,
                            dGamma,nLightness,NULL,&nLutLength);
```

```
    if (DxStatus != DX_OK)
    {
        return;
    }

    // Applies memory for the look-up table.
    VxUint8*pLut = new VxUint8[nLutLength];
    if (pContrastLut == NULL)
    {
       return;
    }

    //Calculates and gets the look-up table.
    DxStatus = DxGetLut(nContrastParam, dGamma,nLightness, pLut,
                        &nLutLength);
    if (DxStatus != DX_OK)
    {
        if (pLut!= NULL)
       {
          delete []pLut;
          pLut = NULL;
       }
        return;
    }

    // Image processing.
    // ……

    if (pLut!= NULL)
    {
       delete []pLut;
       pLut = NULL;
    }
```

8.4.20. DxCalcCCParam

Declaration:

VxInt32 DHDECL DxCalcCCParam (VxInt64 nColorCorrectionParam, VxInt16nSaturation, VxInt16 *parrCC, VxUint8nLength);

Descriptions:

The function is used to calculate the array of image color correction.

Formal parameter:

nColorCorrectionParam	The color correction value, you can get it by the GX_INT_COLOR_CORRECTION_PARAM code of the GxIAPI library, and you can also set it to 0.
nSaturation	The parameter of saturation adjustment, range: 0~128
parrCC	The address of the array.
nLength	The length of array (sizeof (VxInt16 * 9)).

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code Sample:

```
// Applies memory for the image color adjustment array.
VxInt16*parrCC = new VxInt16[sizeof (VxInt16 * 9)];
if (parrCC== NULL)
{
   return;
}

//Gets the color correction parameter.
GxStatus = GXGetInt (hDevice, GX_INT_COLOR_CORRECTION_PARAM,
                     &nColorCorrectionParam);

//Calculates the color adjustment array.
DxStatus = DxCalcCCParam(nColorCorrectionParam, nSaturation, parrCC,
                         sizeof (VxInt16 * 9))
if (DxStatus != DX_OK)
{
    if (parrCC!= NULL)
   {
      delete []parrCC;
      parrCC= NULL;
   }
    return;
}

// Image processing.
// ......................

if (parrCC!= NULL)
{
   delete []parrCC;
   parrCC= NULL;
}
```

8.4.21. DxRaw8ImgProcess

Declaration:

VxInt32 DHDECL DxRaw8ImgProcess (void *pRaw8Buf, void *pRgbBuf, VxInt32 nWidth,
VxInt32 nHeight,
COLOR_IMG_PROCESS *pstClrImageProc);

Descriptions:

The function is used to process the raw 8bit image.

Formal parameter:

pRaw8Buf Point to the original image 8-bit data buffer.
pRgbBuf Point to the target image data buffer (RGB data), the size is image width *image height*3.
nWidth The width of the image.

nHeight The height of the image.

Note: If the current COLOR_IMG_PROCESS:: bAccelerate is set to true, to speed up, the *nHeight* must be the integer multiple of 4.

pstClrImageProc The structure pointer of color image processing method, and the define as follows:

```
typedef struct   COLOR_IMG_PROCESS
{
    bool                    bDefectivePixelCorrect;    /// The switch of bad pixel correction.
    bool                    bDenoise;                  /// The switch of denoise.
    bool                    bSharpness;            /// The switch of sharpness.
    bool                    bAccelerate;           /// The switch of acceleration.
    VxInt16                 *parrCC;           /// The array address of color processing parameter.
    VxUint8                 nCCBufLength;          /// The length of parrCC (sizeof (VInt16)*9).
    float                   fSharpFactor;          /// The factor of sharpness.
    VxUint8                 *pProLut;              /// The buffer of look-up table.
    VxUint16                nLutLength;            /// The length of look-up table.
    DX_BAYER_CONVERT_TYPE   cvType;                /// The interpolation method.
    DX_PIXEL_COLOR_FILTER   emLayOut;              /// The BAYER format.
    bool                    bFlip;                 /// The flip sign.
    VxUint8                 arrReserved[32];       /// Reserve 32byte.
} COLOR_IMG_PROCESS;
```

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code Sample:

```
// Initializes the structure parameters of the color image processing
// function settings.
COLOR_IMG_PROCESS stClrImageProc;

stClrImageProc.bAccelerate= false;
stClrImageProc.bDefectivePixelCorrect= false;
stClrImageProc.bDenoise= false;
stClrImageProc.bFlip= true;
stClrImageProc.bSharpness           = false;
stClrImageProc.fSharpFactor= fSharpen;
stClrImageProc.cvType= RAW2RGB_NEIGHBOUR;
stClrImageProc.emLayOut = (DX_PIXEL_COLOR_FILTER)nPixelColorFilter;

// Gets the look-up table length.
VxInt32DxStatus = DxGetLut(nContrastParam,dGamma,nLightness,NULL,
                           &stClrImageProc.nLutLenghth);
if (DxStatus != DX_OK)
{
    return;
}

// Applies memory for the look-up table.
stClrImageProc.pProLut = new VxUint8[stClrImageProc.nLutLenghth];
if (pContrastLut == NULL)
{
```

```
    return;
}

// Calculates and gets the look-up table.
DxStatus = DxGetLut(nContrastParam, dGamma,nLightness,
                    stClrImageProc.pProLut,
                    &stClrImageProc.nLutLenghth);
if (DxStatus != DX_OK)
{
    if (stClrImageProc.pProLut!= NULL)
    {
        delete []stClrImageProc.pProLut;
        stClrImageProc.pProLut= NULL;
    }
    return;
}

// Applies memory for the image color adjustment array.
stClrImageProc.nCCBufLength = sizeof (VxInt16 * 9);
stClrImageProc.parrCC = new VxInt16[stClrImageProc.nCCBufLength];
if (stClrImageProc.parrCC == NULL)
{
    if (stClrImageProc.pProLut!= NULL)
    {
        delete []stClrImageProc.pProLut;
        stClrImageProc.pProLut = NULL;
    }

    return;
}

// Gets the color correction parameter.
GxStatus = GXGetInt (hDevice, GX_INT_COLOR_CORRECTION_PARAM,
                     &nColorCorrectionParam);

// Calculates the color adjustment array.
DxStatus = DxCalcCCParam(nColorCorrectionParam, nSaturation,
                         stClrImageProc.parrCC,
                         &stClrImageProc.nCCBufLength)
if (DxStatus != DX_OK)
{
    if (stClrImageProc.pProLut!= NULL)
    {
        delete []stClrImageProc.pProLut;
        stClrImageProc.pProLut = NULL;
    }

    if (stClrImageProc.parrCC!= NULL)
    {
        delete []stClrImageProc.parrCC;
        stClrImageProc.parrCC= NULL;
    }
    return;
```

```
    }

    // Processes the 8 bits Raw images
    emStatus = DxRaw8ImgProcess(pRaw8Buf,pRgbBuf, nWidth, nHeight,
                                &stClrImageProc);

   if (stClrImageProc.pProLut!= NULL)
   {
      delete []stClrImageProc.pProLut;
      stClrImageProc.pProLut = NULL;
   }
    if (stClrImageProc.parrCC!= NULL)
   {
      delete []stClrImageProc.parrCC;
      stClrImageProc.parrCC= NULL;
   }
```

8.4.22. DxMonoImgProcess

Declaration:

VxInt32 DHDECL DxMonoImgProcess (void *pRaw8Buf, void *pRgbBuf, VxInt32 nWidth,
VxInt32 nHeight,
MONO_IMG_PROCESS *pstGrayImageProc);

Descriptions:

The function is used to process the mono 8bit image.

Formal parameter:

pRaw8Buf Point to the 8-bit data buffer of original image.

pRgbBuf Point to the data buffer of target image (RGB data), the size is image width * image height*3.

nWidth The width of the image.

nHeight The height of the image.

Note: If the current COLOR_IMG_PROCESS:: bAccelerate is set to true, to speed up, the *nHeight* must be the integer multiple of 4.

pstClrImageProc The structure pointer of mono image processing method, and the define as follows:

```
Typedefstruct MONO_IMG_PROCESS
{
    bool        bDefectivePixelCorrect;    /// The switch of bad pixel correction.
    bool        bSharpness;                /// The switch of sharpness.
    bool        bAccelerate;               /// The switch of acceleration.
    float       fSharpFactor;              /// The factor of sharpness.
    VxUint8     *pProLut;                  /// The buffer of look-up table.
    VxUint16    nLutLength;                /// The length of look-up table.
    VxUint8     arrReserved[32];           /// Reserve 32byte.
} MONO_IMG_PROCESS;
```

Return value:

If success, returns DX_OK, otherwise, see the DX_STATUS definition.

Code Sample:

```
// Initializes the structure parameters of the color image processing
// function settings.
MONO_IMG_PROCESS stGrayImageProc;

stGrayImageProc.bAccelerate= false;
stGrayImageProc.bDefectivePixelCorrect= false;
stGrayImageProc.bSharpness           = false;
stGrayImageProc.fSharpFactor= fSharpen;

// Gets the length of look-up table.
 VxInt32DxStatus = DxGetLut (nContrastParam,dGamma,nLightness,NULL,
                             &stGrayImageProc.nLutLenghth);
 if (DxStatus != DX_OK)
 {
    return;
 }

 // Applies memory for the look-up table.
stGrayImageProc.pProLut = new VxUint8[stGrayImageProc.nLutLenghth];
if (pContrastLut == NULL)
{
   return;
}

 // Calculates and gets the look-up table.
 DxStatus = DxGetLut(nContrastParam, dGamma,nLightness,
                     stGrayImageProc.pProLut,
                     &stGrayImageProc.nLutLenghth);
 if (DxStatus != DX_OK)
 {
    if (stGrayImageProc.pProLut!= NULL)
    {
       delete []stGrayImageProc.pProLut;
        stGrayImageProc.pProLut = NULL;
    }
    return;
 }

 // Processes the 8 bits mono images.
 emStatus = DxMono8ImgProcess(pRaw8Buf,pRgbBuf, nWidth, nHeight,
                              &stGrayImageProc);

if (stGrayImageProc.pProLut!= NULL)
{
   delete []stGrayImageProc.pProLut;
   stGrayImageProc.pProLut = NULL;
}
```

8.5. Function

8.5.1. Image Quality Enhancement Function

This function can realize the color correction function, the contrast adjustment and the Gamma adjustment function of any combination.

8.5.1.1. Related Functions

- Set the contrast look-up table function
  VxInt32 DHDECL *DxGetContrastLut* (int nContrastParam, void *pContrastLut, int *pLutLength);

- Set the Gamma look-up table function
  VxInt32 DHDECL *DxSetGammatLut* (double dGammaParam, void *pGammaLut, int *pLutLength);

- The function of image quality promotion
  VxInt32 DHDECL *DxImageImprovment* (void *pInputBuffer, void *pOutputBuffer, VxUint32 nWidth, VxUint32 nHeight, VxInt64 nColorCorrectionParam, void *pContrastLut, void *pGammaLut);

  For the use of the functions, refer to the interface section.

8.5.1.2. ColorTransformationControl

- Terms

Color correction (Color transformation): Improves the color reduction of the camera to make the image closer to human visual perception.

Color transformation mode: Sets the mode of the color transformation to be performed. 0: Set to the default mode, the color correction factor uses the coefficient provided from factory; 1: Set to the user-defined mode. The user can input the color correction factor according to the actual application. The user can modify the value of this function during the acquisition.

Color transformation matrix value selection: Sets the value to be inputted in the color transformation matrix to customize the color transformation. Note: Depending on the camera model, some values of the color transformation matrix may be pre-set and cannot be changed. The user can modify the value of this function during the acquisition.

Color transformation matrix value: Gets the value in the color transformation matrix, which is used to customize the current value of color transformation.

The user expects that the camera can output the precision color, but the color world is a dynamic one, everyone sees some difference in the color. Same to the sensor, different sensors have different interpretations of color. But what is the precision color and who determine it? So, the user needs a color template. The color template contains 24 colors, and each color has fixed RGB value, as shown in Figure 71:

Figure 71: Color template

With this color template, the user can base on it to shoot the color template with a camera, the RGB value of each color may be different from the standard RGB value of the color template, the vendor can use the software or hardware to convert the RGB value that is read to the standard RGB value. Because the color space is continuous, all the other RGB values read can be converted to the standard RGB values by using the 24 colors.

- Related Parameters

GX_ENUM_COLOR_TRANSFORMATION_MODE:	The color transformation mode， refer to GX_COLOR_TRANSFORMATION_MODE_ENTRY
GX_BOOL_COLOR_TRANSFORMATION_ENABLE:	Color transformation enable
GX_ENUM_COLOR_TRANSFORMATION_VALUE_SELECTOR:	The color transformation matrix value selection， refer to GX_COLOR_TRANSFORMATION_VALUE_SELECTOR_ENTRY
GX_FLOAT_COLOR_TRANSFORMATION_VALUE:	The color transformation matrix value
GX_INT_COLOR_CORRECTION_PARAM:	The color correction parameter

- Effect images

Figure 72:Before color correction

Figure 73:After color correction

- Sample code

```
GX_STATUS status = GX_STATUS_SUCCESS;

//Enables color transformation.
status = GXSetBool(hDevice, GX_BOOL_COLOR_TRANSFORMATION_ENABLE,
                   true);
```

```
//Sets the color transformation mode to user-defined mode.
GX_COLOR_TRANSFORMATION_MODE_ENTRY nValue;
nValue = GX_COLOR_TRANSFORMATION_SELECTOR_USER;
status = GXSetEnum(hDevice, GX_ENUM_COLOR_TRANSFORMATION_MODE,
                   nValue);

//Gets the color correction parameter value.
int64_t nColorParam = 0;
status = GXGetInt(hDevice, GX_INT_COLOR_CORRECTION_PARAM,
                  &nColorParam);
```

8.5.1.3. Contrast Adjustment

- Terms

  Contrast: The brightness ratio of the bright part and the dark part of the image is called contrast. The image with high contrast is clear and the profile of the object which is captured by the camera is clear. Conversely, the image with low contrast is not clear and the profile of the object is not clear.

- Related Parameters

  `GX_INT_CONTRAST_PARAM`：the contrast parameter.

- Effect images

Figure 74:Before contrast adjustment

Figure 75:After contrast adjustment

8.5.1.4. Gamma Adjustment

- Terms

Gamma adjustment： The Gamma adjustment is to make the output of the display as close as possible to the input.

Gamma： The value of Gamma adjustment. A nonlinear transformation of the pixel value is performed on the current image in the form of a power function.

Gamma mode: Manual adjustment mode 0: set to default mode, 1: set to user-defined mode.

- Related Parameters

GX_BOOL_GAMMA_ENABLE:	Gamma enable
GX_ENUM_GAMMA_MODE:	Gamma mode, refer to GX_GAMMA_MODE_ENTRY
GX_FLOAT_GAMMA:	Gamma
GX_FLOAT_GAMMA_PARAM:	Gamma parameter

- Effect images

Figure 76:Before Gamma adjustment

Figure 77: After Gamma adjustment

- Sample code

```
GX_STATUS status = GX_STATUS_SUCCESS;

//Enables Gamma.
status = GXSetBool(hDevice, GX_BOOL_GAMMA_ENABLE, true);

//Sets Gamma mode to user-defined mode.
GX_GAMMA_MODE_ENTRY nValue;
nValue = GX_GAMMA_SELECTOR_GAMMA;
status = GXSetEnum(hDevice, GX_ENUM_GAMMA_MODE, nValue);

//Gets the Gamma parameter value.
```

```
double dColorParam = 0.0;
status = GXGetFloat(hDevice, GX_FLOAT_GAMMA_PARAM, &dColorParam);
```

8.5.1.5. Sharpen

- Terms

Sharpen: Sharpening is to improve the definition of the image edges. The higher the definition, the clearer the outline of the image.

Sharpness: Adjust the sharpness value to adjust the camera's sharpness to the image. The adjustment range is 0-3.0. The larger the value, the higher the sharpness.

Sharpen mode: Decide whether to enable the sharpening function. ON means that the sharpening function is enabled; OFF means that the sharpening function is disabled.

- Related Parameters

GX_FLOAT_SHARPNESS:	Sharpness
GX_ENUM_SHARPNESS_MODE:	Sharpen mode, refer to GX_SHARPNESS_MODE_ENTRY

- Effect images

Figure 78: Before sharpen adjustment

Figure 79: After sharpen adjustment

- Sample code

```
GX_STATUS status = GX_STATUS_SUCCESS;

//Enables sharpening.
GX_SHARPNESS_MODE_ENTRY nValue;
nValue = GX_SHARPNESS_MODE_ON;
status = GXSetEnum(hDevice, GX_ENUM_SHARPNESS_MODE, nValue);

//Gets the value of sharpness.
double dColorParam = 0.0;
status = GXGetFloat(hDevice, GX_FLOAT_SHARPNESS, &dColorParam);
```

9. Revision History

No.	Version	Changes	Date
1	V1.0.0	Initial release	2013-03
2	V2.0.0	Initial release for Windows & Linux	2019-01-11
3	V2.0.1	Modify a bug of code sample	2019-01-23
4	V2.0.2	Modify the mistake found in the system test	2019-02-12
5	V2.0.3	Modify some descriptions	2019-02-14
6	V2.0.4	Modify some formats	2019-02-21