Frame Grabber SDK (Windows-C)

Developer Guide

Legal Information

Contents

Chapter 1 Overview

The Frame Grabber SDK is a software development kit, which provides unified APIs for the access and control of frame grabbers. It simplifies the API calling process, and supports operations of multiple types of frame grabbers at the same time.

1.1 Introduction

This manual mainly introduces the Frame Grabber SDK based on C language, which provides several APIs for controlling frame grabbers, cameras, and buffers, acquiring images, configuring device parameters, and processing images.

Note

Now the supported frame grabber types are: GigE, CoaXPress, and Camera Link.

1.2 Development Environment

The development environment of Frame Grabber SDK is shown in the table below.

Frame Grabber Type	Item	Required
GigE	Hardware	PCIe Gen2×4 bus
	Software	Microsoft® Windows 7 (32/64-bit)/Windows 10 (32/64-bit)
CoaXPress	Hardware	PCIe Gen2×8 bus
	Software	Microsoft® Windows 7 (32/64-bit)/Windows 10 (32/64-bit)
Camera Link	Hardware	PCIe Gen2×4 bus
	Software	Microsoft® Windows 7 (32/64-bit)/Windows 10 (32/64-bit)

1.3 Update History

The update history shows the summary of changes in Frame Grabber SDK with different versions.

Summary of Changes in Version 2.1.0_10/2022

Version	Content
	1. Added an API for opening a frame grabber according to the frame

Version	Content
Version 2.1.0_10/2022	grabber ID and specifying its access mode: ***MV_FG_OpenInterfaceByID***.
	2. Added an API for opening a device according to the device ID: ***MV_FG_OpenDeviceByID***.
	3. Added an API for saving the TIFF image: ***MV_FG_SaveTiffToFile***.
	4. Added an API for saving the PNG image: ***MV_FG_SavePngToFile***.
	5. Added an API for decoding the lossless compressed stream: ***MV_FG_HB_Decode***.
	6. Extended the enumeration about exception types: ***MV_FG_EXCEPTION_TYPE***
	7. Added the sample code for event receiving: ***Receive Events***.

Summary of Changes in Version 2.0.0_05/2022

New document.

Chapter 2 Basic Process

This chapter mainly introduces the API calling flow of basic process, including the frame grabber operation, device operation, and image acquisition.
In the figure of API calling flow, APIs in the white area are related to frame grabbers, APIs in the blue area are related to devices, APIs in orange area are related to stream operations.

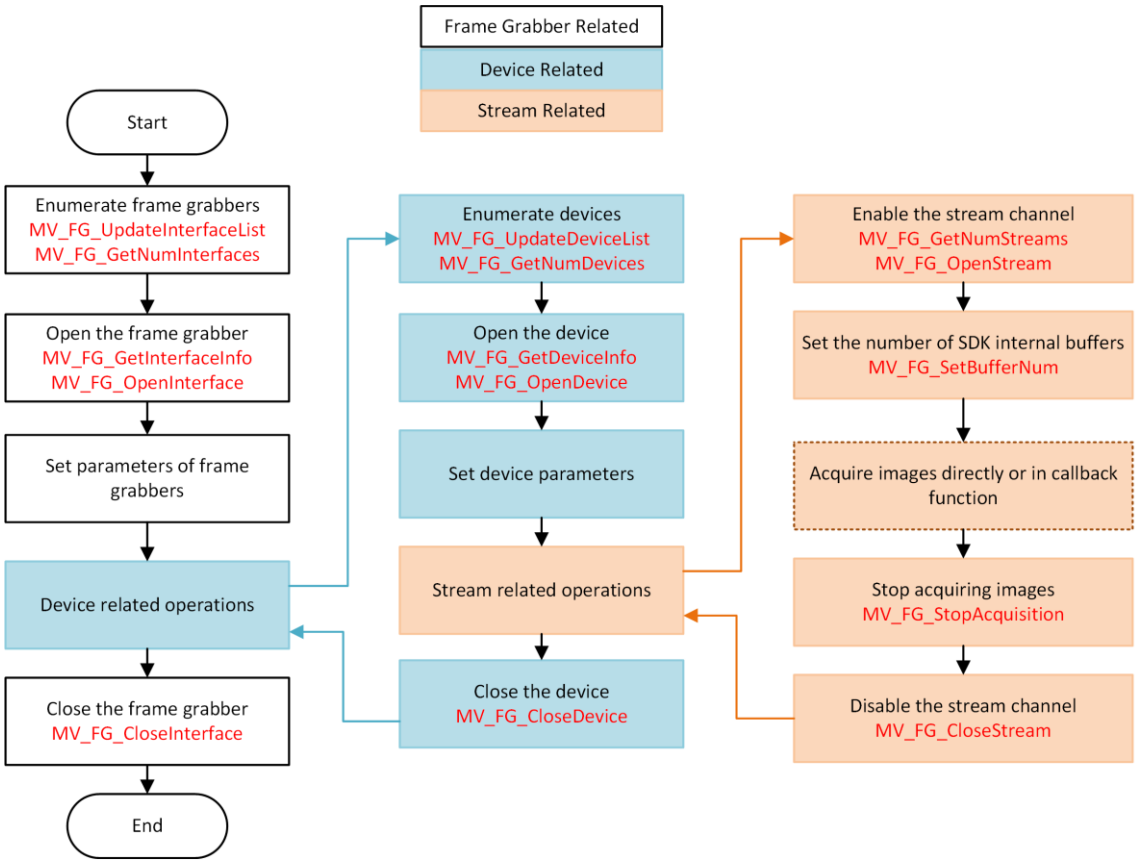

Figure 2-1 API Calling Flow of Basic Process

Image Acquisition

Two methods of image acquisition are provided: acquire images directly or acquire images in the callback function.

- Acquire images directly.
  - Start streaming: ***MV_FG_StartAcquisition***.
  - Get the image buffer information: ***MV_FG_GetFrameBuffer***.
  - Release the image buffer: ***MV_FG_ReleaseFrameBuffer***.
- Acquire images in the callback function.
  - Register a callback function for receiving frame buffer information: ***MV_FG_RegisterFrameCallBack***.
  - Start streaming: ***MV_FG_StartAcquisition***.

Note

- Above two methods of image acquisition cannot be used at the same time.
- In callback functions, time-consuming operations and thread locks are not recommended, which may cause blocking.
- The image data in the image buffer structure is a buffer pointer, it is recommended to copy the data of callback function and use it in another thread.

Chapter 3 API Reference

3.1 Version Information

API for getting the SDK version information.

3.1.1 MV_FG_GetSDKVersion

Get the SDK version information.

API Definition

```
unsigned char* MV_FG_GetSDKVersion( );
```

Return Value

Return the SDK version information in a string.

Remarks

Format: "Version number + Type + Compile time".

3.2 Frame Grabber

APIs for enumerating frame grabbers, acquiring relevant information, enabling/disabling frame grabbers, and so on.

3.2.1 MV_FG_UpdateInterfaceList

Update the frame grabber list.

API Definition

```
int MV_FG_UpdateInterfaceList(
   unsigned int      nTLayerType,
   bool8_t           *pbChanged
);
```

Parameters

nTLayerType

[IN] Frame grabber type. See ***Frame Grabber Type*** for details.

pbChanged

[OUT] Whether the frame grabber list is updated.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

This API must be called before any frame grabber operations are performed. The internal list of frame grabbers will only be updated when this API is called.

3.2.2 MV_FG_ReleaseTLayerResource

Release the frame grabber resource of the specified type.

API Definition

```
int MV_FG_ReleaseTLayerResource(
   unsigned int       nTLayerType
);
```

Parameters

nTLayerType

[IN] Frame grabber type. See ***Frame Grabber Type*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

- Before releasing the library MvFGControl.dll, you need to call this API to release all frame grabbers resources.
- All frame grabbers of the specified type should be disabled before calling this API.
- After disabling the used GigE Vision frame grabber, you need to call this API to release the CTI resources of frame grabber. Otherwise, the GigE Vision frame grabber cannot be enabled by another process since the CTI resources are occupied.

3.2.3 MV_FG_GetNumInterfaces

Get the number of frame grabbers.

API Definition

```
int MV_FG_GetNumInterfaces(
   unsigned int       *pnNumIfaces
);
```

Parameters

pnNumIfaces

[OUT] Number of frame grabbers.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.2.4 MV_FG_GetInterfaceInfo

Get the frame grabber information by frame grabber index.

API Definition

```
int MV_FG_GetInterfaceInfo(
   unsigned int                  nIndex,
   MV_FG_INTERFACE_INFO          *pstIfaceInfo
);
```

Parameters

nIndex

[IN] Frame grabber index, which starts from 0.

pstIfaceInfo

[OUT] Frame grabber information, see ***MV_FG_INTERFACE_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.2.5 MV_FG_OpenInterface

Open the frame grabber.

API Definition

```
int MV_FG_OpenInterface(
   unsigned int      nIndex,
   IFHANDLE          *phIface
);
```

Parameters

nIndex

[IN] Frame grabber index, which starts from 0.

phIface

[OUT] Frame grabber handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.2.6 MV_FG_OpenInterfaceEx

Open the frame grabber and specify its access mode.

API Definition

```
int MV_FG_OpenInterfaceEx(
   unsigned int      nIndex,
```

```
    int                 nAccess,
    IFHANDLE            *phIface
);
```

Parameters

nIndex

[IN] Frame grabber index, which starts from 0.

nAccess

[IN] Access mode. See ***Frame Grabber Access Mode*** for details.

phIface

[OUT] Frame grabber handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

- There are three types of access modes: *MV_FG_ACCESS_UNKNOWN*, *MV_FG_ACCESS_READONLY*, and *MV_FG_ACCESS_CONTROL*.
- CoaXPress frame grabbers and Camera Link frame grabbers can only be enabled with the access mode *MV_FG_ACCESS_CONTROL* (i.e., with permission to control).

3.2.7 MV_FG_OpenInterfaceByID

Open the frame grabber according to frame grabber ID and specify its access mode.

API Definition

```
int MV_FG_OpenInterfaceByID(
    char                *pcInterfaceID,
    int                 nAccess,
    IFHANDLE            *phIface
);
```

Parameters

pcInterfaceID

[IN] Frame grabber ID.

nAccess

[IN] Access mode. See ***Frame Grabber Access Mode*** for details.

phIface

[OUT] Frame grabber handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

- There are three types of access modes: *MV_FG_ACCESS_UNKNOWN*, *MV_FG_ACCESS_READONLY*,

and *MV_FG_ACCESS_CONTROL*.
- CoaXPress frame grabbers and Camera Link frame grabbers can only be enabled with the access mode *MV_FG_ACCESS_CONTROL* (i.e., with permission to control).

3.2.8 MV_FG_CloseInterface

Close the frame grabber.

API Definition

```
int MV_FG_CloseInterface(
   IFHANDLE          phIface
);
```

Parameters

phIface

[IN] Frame grabber handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.3 Device

APIs for enumerating devices, acquiring relevant information, and opening/closing devices.

3.3.1 MV_FG_UpdateDeviceList

Update the device list of a specified frame grabber.

API Definition

```
int MV_FG_UpdateDeviceList(
   IFHANDLE        hIface,
   bool8_t         *pbChanged
);
```

Parameters

hIface

[IN] Frame grabber handle.

pbChanged

[OUT] Whether the device list is updated.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

The internal list of devices will only be updated when this API is called.

3.3.2 MV_FG_GetNumDevices

Get the number of devices of a specified frame grabber.

API Definition

```
int MV_FG_GetNumDevices(
    IFHANDLE                hIface,
    unsigned int            *pnNumDevices
);
```

Parameters

hIface

[IN] Frame grabber handle.

pnNumDevices

[OUT] Number of devices.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.3.3 MV_FG_GetDeviceInfo

Get the device information.

API Definition

```
int MV_FG_GetDeviceInfo(
    IFHANDLE                hIface,
    unsigned int            nIndex,
    MV_FG_DEVICE_INFO       *pstDevInfo
);
```

Parameters

hIface

[IN] Frame grabber handle.

nIndex

[IN] Device index, which starts from 0.

pstDevInfo

[OUT] Device information, see ***MV_FG_DEVICE_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.3.4 MV_FG_OpenDevice

Open the device according to index.

API Definition

```
int MV_FG_OpenDevice(
   IFHANDLE              hIface,
   unsigned int          nIndex,
   DEVHANDLE*            phDevice
);
```

Parameters

hIface

[IN] Frame grabber handle.

nIndex

[IN] Device index, which starts from 0.

phDevice

[OUT] Device handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.3.5 MV_FG_OpenDeviceByID

Open the device according to device ID.

API Definition

```
int MV_FG_OpenDevice(
   IFHANDLE              hIface,
   char                  *pcDeviceID,
   DEVHANDLE             *phDevice
);
```

Parameters

hIface

[IN] Frame grabber handle.

pcDeviceID

[IN] Device ID.

phDevice

[OUT] Device handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.3.6 MV_FG_CloseDevice

Close the device module corresponding to the specified device handle.

API Definition

```
int MV_FG_CloseDevice(
   DEVHANDLE*          hDevice
);
```

Parameters

hDevice

[IN] Device handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.4 Stream

APIs for acquiring stream information, enabling/disabling stream channels, and so on.

3.4.1 MV_FG_GetNumStreams

Get the number of stream channels.

API Definition

```
int MV_FG_GetNumStreams(
   DEVHANDLE              hDevice,
   unsigned int          *pnNumStreams
);
```

Parameters

hDevice

[IN] Device handle.

pnNumStreams

[OUT] Number of stream channels.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.4.2 MV_FG_GetPayloadSize

Get the image size of the stream channel.

API Definition

```
int MV_FG_GetPayloadSize(
```

```
    STREAMHANDLE            hStream,
    unsigned int            *pnPayloadSize
);
```

Parameters

hStream

[IN] Stream channel handle.

pnPayloadSize

[OUT] Image size of the stream channel.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

The image size of the stream channel needs to be reacquired after the image-related parameters of the camera are changed, such as width, height, and pixel format.

3.4.3 MV_FG_OpenStream

Open the stream channel.

API Definition

```
int MV_FG_OpenStream(
    DEVHANDLE               hDevice,
    unsigned int            nIndex,
    STREAMHANDLE            *phStream
);
```

Parameters

hDevice

[IN] Device handle.

nIndex

[IN] Stream channel index.

phStream

[OUT] Stream channel handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.4.4 MV_FG_CloseStream

Close the stream channel.

API Definition

```
int MV_FG_CloseStream(
```

```
    STREAMHANDLE        hStream
);
```

Parameters

hStream

[IN] Stream channel handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.4.5 MV_FG_StartAcquisition

Start image acquisition.

API Definition

```
int MV_FG_StartAcquisition(
    STREAMHANDLE        hStream
);
```

Parameters

hStream

[IN] Stream channel handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.4.6 MV_FG_StopAcquisition

Stop image acquisition.

API Definition

```
int MV_FG_StopAcquisition(
    STREAMHANDLE        hStream
);
```

Parameters

hStream

[IN] Stream channel handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.4.7 MV_FG_GetFrameBuffer

Get the buffer information of a frame. This API is valid only when buffers are managed internally by the

SDK.

API Definition

```
int MV_FG_GetFrameBuffer(
   STREAMHANDLE                hStream,
   MV_FG_BUFFER_INFO           *pstBufferInfo,
   unsigned int                nTimeout
);
```

Parameters

hStream

[IN] Stream channel handle.

pstBufferInfo

[OUT] Buffer information, see ***MV_FG_BUFFER_INFO*** for details.

nTimeout

[IN] Timeout, unit: millisecond.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

The API ***MV_FG_ReleaseFrameBuffer*** should be called to release the buffer information.

3.4.8 MV_FG_RegisterFrameCallBack

Register the callback function for frame buffer information. This API is valid only when buffers are managed internally by the SDK.

API Definition

```
int MV_FG_RegisterFrameCallBack(
   STREAMHANDLE                hStream,
   MV_FG_FrameCallBack         cbFrame,
   void                        *pUser
);
```

Parameters

hStream

[IN] Stream channel handle.

cbFrame

[IN] Frame buffer information callback function.

```
void (__stdcall *MV_FG_FrameCallBack)(
   MV_FG_BUFFER_INFO        *pstBufferInfo,
   void                     *pUser
)
```

pstBufferInfo

[IN] Frame buffer information, see ***MV_FG_BUFFER_INFO*** for details.

pUser

[IN] User-defined data.

pUser

[IN] User-defined data.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

- Time-consuming operations in the callback function will block the access to the subsequent frame buffer information.
- This API and the API ***MV_FG_GetFrameBuffer*** are mutually exclusive.
- You need to call this API to register the callback function before calling API ***MV_FG_StartAcquisition***.

3.4.9 MV_FG_ReleaseFrameBuffer

Release the buffer information. This API is valid only when buffers are requested internally by the SDK.

API Definition

```
int MV_FG_ReleaseFrameBuffer(
   STREAMHANDLE              hStream,
   MV_FG_BUFFER_INFO         *pstBufferInfo
);
```

Parameters

hStream

[IN] Stream channel handle.

pstBufferInfo

[IN] Buffer information, see ***MV_FG_BUFFER_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

This API and ***MV_FG_GetFrameBuffer*** should be used in pair. The image data (**pstBufferInfo**) obtained via ***MV_FG_GetFrameBuffer*** should be released by this API.

3.4.10 MV_FG_GetImageBuffer

Get the buffer handle of a frame. This API is valid only when buffers are requested and registered to

stream channels by the user.

API Definition

```
int MV_FG_GetImageBuffer(
   STREAMHANDLE              hStream,
   BUFFERHANDLE              *phBuffer,
   unsigned int              nTimeout
);
```

Parameters

hStream

[IN] Stream channel handle.

phBuffer

[OUT] Buffer handle.

nTimeout

[IN] Timeout, unit: millisecond.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

By calling the API ***MV_FG_GetBufferInfo***, you can get the buffer information with the acquired buffer handle.

3.5 Buffer

APIs for managing buffers, acquiring relevant information, and so on.

3.5.1 MV_FG_SetBufferNum

Set the number of internal buffers for the SDK.

API Definition

```
int MV_FG_SetBufferNum(
   STREAMHANDLE              hStream,
   unsigned int              nBufferNum
);
```

Parameters

hStream

[IN] Stream channel handle.

nBufferNum

[IN] Number of buffers.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

- The SDK has no internal buffer if this API is not called or the value of parameter **nBufferNum** is set to 0. When there is no internal buffer, you need to request and register buffers to stream channels before starting image acquisition.
- When the parameter **nBufferNum** is set to a value greater than 0, buffers will the allocated internally by the SDK and it is not allowed to register buffers to stream channels at this time.
- The number of image buffers should be reasonably allocated.

3.5.2 MV_FG_GetBufferChunkData

Get the chunk data information of a buffer.

API Definition

```
int MV_FG_GetBufferChunkData(
   STREAMHANDLE                hStream,
   MV_FG_BUFFER_INFO           *pstBufferInfo,
   unsigned int                nIndex,
   MV_FG_CHUNK_DATA_INFO       *pstChunkDataInfo
);
```

Parameters

hStream

[IN] Stream channel handle.

pstBufferInfo

[IN] Buffer information, see ***MV_FG_BUFFER_INFO*** for details.

nIndex

[IN] Chunk data index.

pstChunkDataInfo

[OUT] Chunk data information, see ***MV_FG_CHUNK_DATA_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

After getting the buffer information, this API must be called before calling the APIs ***MV_FG_ReleaseFrameBuffer*** and ***MV_FG_QueueBuffer*** to get valid information of the chunk data.

3.5.3 MV_FG_AnnounceBuffer

Register buffer to the stream channel.

API Definition

```
int MV_FG_AnnounceBuffer(
   STREAMHANDLE          hStream,
   void                  *pBuffer,
   unsigned int          nSize,
   void                  *pPrivate,
   BUFFERHANDLE          *phBuffer
);
```

Parameters

hStream

[IN] Stream channel handle.

pBuffer

[IN] Image buffer address.

nSize

[IN] Image buffer size.
The image buffer size is acquired by calling the API ***MV_FG_GetPayloadSize***. The private information is user-defined.

pPrivate

[IN] Private information address.

phBuffer

[OUT] Buffer handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

This API must be called prior to acquiring images.

3.5.4 MV_FG_RevokeBuffer

Revoke buffer from the stream channel.

API Definition

```
int MV_FG_RevokeBuffer(
   STREAMHANDLE          hStream,
   BUFFERHANDLE          hBuffer,
   void                  **pBuffer,
   void                  **pPrivate
);
```

Parameters

hStream

[IN] Stream channel handle.

hBuffer

[IN] Buffer handle.

pBuffer

[OUT] Image buffer address.

pPrivate

[OUT] Private information address.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

Only buffers in unused queues can be revoked. You can allocate buffers to unused queues by calling the API ***MV_FG_FlushQueue***.

3.5.5 MV_FG_FlushQueue

Refresh the buffer queue.

API Definition

```
int MV_FG_FlushQueue(
   STREAMHANDLE                hStream,
   MV_FG_BUFFER_QUEUE_TYPE     enQueueType
);
```

Parameters

hStream

[IN] Stream channel handle.

enQueueType

[IN] Buffer queue type, see ***MV_FG_BUFFER_QUEUE_TYPE*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

The following queue types are not supported during image acquisition: **MV_FG_BUFFER_QUEUE_INPUT_TO_OUTPUT** and **MV_FG_BUFFER_QUEUE_ALL_DISCARD**.

3.5.6 MV_FG_GetBufferInfo

Get the buffer information by buffer handle.

API Definition

```
int MV_FG_GetBufferInfo(
   BUFFERHANDLE          hBuffer,
   MV_FG_BUFFER_INFO     *pstBufferInfo
);
```

Parameters

hBuffer

[IN] Buffer handle.

pstBufferInfo

[OUT] Buffer information, see ***MV_FG_BUFFER_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.5.7 MV_FG_QueueBuffer

Insert the buffer back to the input queue.

API Definition

```
int MV_FG_QueueBuffer(
   BUFFERHANDLE       hBuffer
);
```

Parameters

hBuffer

[IN] Buffer handle.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.6 Image Processing

APIs for displaying a frame of image, saving BMP/JPEG images, converting pixel formats, and so on.

3.6.1 MV_FG_DisplayOneFrame

Display one frame of image.

API Definition

```
int MV_FG_DisplayOneFrame(
```

```
  IMAGEHANDLE                    hImage,
  void                          *hWnd,
  MV_FG_DISPLAY_FRAME_INFO       *pstDisplayFrameInfo
);
```

Parameters

hImage

[IN] Image handle. You can use the handle of frame grabbers, devices, or stream channels.

hWnd

[IN] Window handle.

pstDisplayFrameInfo

[IN] Image information to be displayed, see ***MV_FG_DISPLAY_FRAME_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.6.2 MV_FG_SaveBitmap

Save the BMP image.

API Definition

```
int MV_FG_SaveBitmap(
  IMAGEHANDLE                hImage,
  MV_FG_SAVE_BITMAP_INFO      *pstSaveBitmapInfo
);
```

Parameters

hImage

[IN] Image handle. You can use the handle of frame grabbers, devices, or stream channels.

pstSaveBitmapInfo

[IN][OUT] BMP image information, see ***MV_FG_SAVE_BITMAP_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.6.3 MV_FG_SaveJpeg

Save the JPEG image.

API Definition

```
int MV_FG_SaveJpeg(
  IMAGEHANDLE                hImage,
  MV_FG_SAVE_JPEG_INFO        *pstSaveJpegInfo
);
```

Parameters

hImage

[IN] Image handle. You can use the handle of frame grabbers, devices, or stream channels.

pstSaveJpegInfo

[IN][OUT] JPEG image information, see ***MV_FG_SAVE_JPEG_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.6.4 MV_FG_SaveTiffToFile

Save the TIFF image.

API Definition

```
int MV_FG_SaveTiffToFile(
    IMAGEHANDLE                     hImage,
    MV_FG_SAVE_TIFF_TO_FILE_INFO    *pstSaveTiffInfo
);
```

Parameters

hImage

[IN] Image handle. You can use the handle of frame grabbers, devices, or stream channels.

pstSaveTiffInfo

[IN][OUT] TIFF image information, see ***MV_FG_SAVE_TIFF_TO_FILE_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.6.5 MV_FG_SavePngToFile

Save the PNG image.

API Definition

```
int MV_FG_SavePngToFile(
    IMAGEHANDLE                     hImage,
    MV_FG_SAVE_PNG_TO_FILE_INFO     *pstSavePngInfo
);
```

Parameters

hImage

[IN] Image handle. You can use the handle of frame grabbers, devices, or stream channels.

pstSavePngInfo

[IN][OUT] PNG image information, see ***MV_FG_SAVE_PNG_TO_FILE_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.6.6 MV_FG_ConvertPixelType

Convert the pixel format.

API Definition

```
int MV_FG_ConvertPixelType(
   IMAGEHANDLE                  hImage,
   MV_FG_CONVERT_PIXEL_INFO      *pstConvertPixelInfo
);
```

Parameters

hImage

[IN] Image handle. You can use the handle of frame grabbers, devices, or stream channels.

pstConvertPixelInfo

[IN][OUT] Pixel format conversion information, see ***MV_FG_CONVERT_PIXEL_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.6.7 MV_FG_ReconstructImage

Reconstruct images.

API Definition

```
int MV_FG_ReconstructImage(
   IMAGEHANDLE                  hImage,
   MV_FG_RECONSTRUCT_INFO       *pstReconstructInfo
);
```

Parameters

hImage

[IN] Image handle. You can use the handle of frame grabbers, devices, or stream channels.

pstReconstructInfo

[IN][OUT] Image reconstruction information, see ***MV_FG_RECONSTRUCT_INFO*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

- Image rotation, image flipping, and image splitting are supported, but the specific precondition is required.
- The supported pixel formats of image rotation and image flipping are: *MV_FG_PIXEL_TYPE_Mono8*,

MV_FG_PIXEL_TYPE_RGB8_Packed, and *MV_FG_PIXEL_TYPE_BGR8_Packed*.
- Image splitting supports all pixel formats.

3.6.8 MV_FG_HB_Decode

Decode the lossless compressed stream.

API Definition

```
int __stdcall MV_FG_HB_Decode(
    IMAGEHANDLE                 hImage,
    MV_FG_HB_DECODE_PARAM       *pstDecodeParam
);
```

Parameters

hImage

[IN] Image handle. You can use the handle of frame grabbers, devices, or stream channels.

pstDecodeParam

[IN][OUT] Structure about lossless decoding parameters. See ***MV_FG_HB_DECODE_PARAM*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

- This API is used to decode the lossless compressed stream obtained from cameras into raw data.
- This API also supports parsing the watermark information of real-time image for the current camera. If the input lossless stream is not from the current camera or not obtained in real time, the watermark parsing may fail. To parse the watermark information, you should input the handle of current camera, otherwise, only lossless decoding will be performed if you input the frame grabber handle or stream handle.

3.7 Parameters Control

APIs for getting XML files of frame grabbers and cameras, getting and setting the device parameters, and saving and loading the device features.

3.7.1 MV_FG_GetXMLFile

Get the XML file of frame grabbers / devices.

API Definition

```
int MV_FG_GetXMLFile(
    PORTHANDLE          hPort,
    unsigned char       *pData,
    unsigned int        nDataSize,
    unsigned int        *pnDataLen
```

```
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

pData

[IN][OUT] Address of the buffer in which the XML file is stored.

nDataSize

[IN] Size of the buffer in which the XML file is stored.

pnDataLen

[OUT] Length of the XML file.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.2 MV_FG_GetNodeAccessMode

Get the access mode of a node.

API Definition

```
int MV_FG_GetNodeAccessMode(
   PORTHANDLE                  hPort,
   const char                 *strName,
   MV_FG_NODE_ACCESS_MODE     *penAccessMode
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strName

[IN] Node name.

penAccessMode

[OUT] Access mode of the node, see ***MV_FG_NODE_ACCESS_MODE*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

To get the access mode of the EnumEntry type node, the input value for **strName** should be in the format "EnumEntry_NodeName_EnumEntryName". For example, to get the access mode of the node *MV_FG_PIXEL_TYPE_Mono8* of the enumeration ***MV_FG_PIXEL_TYPE***, the **strName** should be "EnumEntry_PixelFormat_Mono8".

3.7.3 MV_FG_GetNodeInterfaceType

Get the type of a node.

API Definition

```
int MV_FG_GetNodeInterfaceType(
   PORTHANDLE                    hPort,
   const char                   *strName,
   MV_FG_NODE_INTERFACE_TYPE     *penInterfaceType
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strName

[IN] Node name.

penInterfaceType

[OUT] Type of the node, see ***MV_FG_NODE_INTERFACE_TYPE*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.4 MV_FG_GetIntValue

Get the information about an integer type node.

API Definition

```
int MV_FG_GetIntValue(
   PORTHANDLE          hPort,
   const char         *strKey,
   MV_FG_INTVALUE      *pstIntValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

pstIntValue

[OUT] Information about the integer type node, see ***MV_FG_INTVALUE*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.5 MV_FG_SetIntValue

Set the information of an integer type node.

API Definition

```
int MV_FG_SetIntValue(
   PORTHANDLE          hPort,
   const char          *strKey,
   int64_t             nValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

nValue

[IN] Value to be set.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.6 MV_FG_GetEnumValue

Get the information about an enumerated type node.

API Definition

```
int MV_FG_GetEnumValue(
   PORTHANDLE          hPort,
   const char          *strKey,
   MV_FG_ENUMVALUE     *pstEnumValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

pstEnumValue

[OUT] Information about the enumerated type node, see ***MV_FG_ENUMVALUE*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.7 MV_FG_SetEnumValue

Set the information of an enumerated type node.

API Definition

```
int MV_FG_SetEnumValue(
   PORTHANDLE              hPort,
   const char              *strKey,
   unsigned int            nValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

nValue

[IN] Value to be set.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.8 MV_FG_SetEnumValueByString

Set the information of an enumeration type node by string.

API Definition

```
int MV_FG_SetEnumValueByString(
   PORTHANDLE              hPort,
   const char              *strKey,
   const char              *strValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

strValue

[IN] Values (input as strings) to be set.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.9 MV_FG_GetFloatValue

Get the information about a float type node.

API Definition

```
int MV_FG_GetFloatValue(
   PORTHANDLE              hPort,
   const char              *strKey,
   MV_FG_FLOATVALUE        *pstFloatValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

pstFloatValue

[OUT] Information about the float type node, see ***MV_FG_FLOATVALUE*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.10 MV_FG_SetFloatValue

Set the information of a float type node.

API Definition

```
int MV_FG_SetFloatValue(
   PORTHANDLE           hPort,
   const char           *strKey,
   float                fValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

fValue

[IN] Value to be set.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.11 MV_FG_GetBoolValue

Get the information about a boolean type node.

API Definition

```
int MV_FG_GetBoolValue(
   PORTHANDLE          hPort,
   const char          *strKey,
   bool8_t             *pbValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

pstIntValue

[OUT] Information about the boolean type node.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.12 MV_FG_SetBoolValue

Set the information of a boolean type node.

API Definition

```
int MV_FG_SetBoolValue(
   PORTHANDLE          hPort,
   const char          *strKey,
   bool8_t             bValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

bValue

[IN] Value to be set.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.13 MV_FG_GetStringValue

Get the information about a string type node.

API Definition

```
int MV_FG_GetStringValue(
   PORTHANDLE                hPort,
   const char                *strKey,
   MV_FG_STRINGVALUE         *pstStringValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

pstStringValue

[OUT] Information about the string type node, see ***MV_FG_STRINGVALUE*** for details.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.14 MV_FG_SetStringValue

Set the information of a string type node.

API Definition

```
int MV_FG_SetStringValue(
   PORTHANDLE           hPort,
   const char           *strKey,
   const char           *strValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strKey

[IN] Node name.

strValue

[IN] Value to be set.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.15 MV_FG_SetCommandValue

Execute the commands of a command type node.

API Definition

```
int MV_FG_SetCommandValue(
   PORTHANDLE          hPort,
   const char          *strKey
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be only a device.

strKey

[IN] Node name.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

This API is supported by Camera Link cameras.

3.7.16 MV_FG_SetConfigIntValue

Set a custom value for an integer type of node.

API Definition

```
int MV_FG_SetConfigIntValue(
   PORTHANDLE                hPort,
   MV_FG_CONFIG_CMD          enConfigCmd,
   int64_t                   nValue
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

enConfigCmd

[IN] Configuration command, see ***MV_FG_CONFIG_CMD*** for details.

nValue

[IN] Value to be set. Range: [MV_FG_BAUDRATE_9600, MV_FG_BAUDRATE_AUTOMAX], the default value is MV_FG_BAUDRATE_115200.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

3.7.17 MV_FG_FeatureLoad

Import the device features.

API Definition

```
int MV_FG_FeatureLoad(
   PORTHANDLE          hPort,
   const char          *strFileName
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strFileName

[IN] Name of the file in which the device attributes and features are stored (only supports English).

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

This API is not supported by Camera Link cameras.

3.7.18 MV_FG_FeatureSave

Save the device features.

API Definition

```
int MV_FG_FeatureSave(
   PORTHANDLE          hPort,
   const char          *strFileName
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be either a frame grabber or a device.

strFileName

[IN] Name of the file in which the device attributes and features will be saved (only supports English).

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

This API is not supported by Camera Link cameras.

3.8 Message Notification

APIs for registering the callback function for exceptions and events.

3.8.1 MV_FG_RegisterExceptionCallBack

Register the callback function for exception information.

API Definition

```
int MV_FG_RegisterExceptionCallBack(
   PORTHANDLE                  hPort,
   MV_FG_ExceptionCallBack     cbException,
   void                        *pUser
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be a frame grabber, device, or stream channel.

cbException

[IN] Exception information callback function.

```
void (__stdcall *MV_FG_ExceptionCallBack)(
   MV_FG_EXCEPTION_TYPE        enExceptionType,
   void                        *pUser
)
```

enExceptionType

[IN] Exception type information, see ***MV_FG_EXCEPTION_TYPE*** for details.

pUser

[IN] User-defined data.

pUser

[IN] User-defined data.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

- Processing time-consuming operations in the callback function will block the access to the subsequent exception information.
- This API is available for the exception callback of frame grabbers, devices, and stream channels. The type of registered callback function varies according to the handle type.

3.8.2 MV_FG_RegisterEventCallBack

Register the callback function for events.

API Definition

```
int MV_FG_RegisterEventCallBack(
   PORTHANDLE                hPort,
   const char                *strEventName,
   MV_FG_EventCallBack       cbEvent,
   void                      *pUser
);
```

Parameters

hPort

[IN] Handle of the object to which the parameter belongs; the object can be a frame grabber, device, or stream channel.

strEventName

[IN] Event name.

cbEvent

[IN] Events callback function.

```
void (__stdcall *MV_FG_EventCallBack)(
   MV_FG_EVENT_INFO          *pstEventInfo,
   void                      *pUser
)
```

pstEventInfo

[IN] Events information, see ***MV_FG_EVENT_INFO*** for details.

pUser

[IN] User-defined data.

pUser

[IN] User-defined data.

Return Value

Return ***MV_FG_SUCCESS*** on success, and return ***Error Code*** on failure.

Remarks

- Processing time-consuming operations in the callback function will block the access to the subsequent events information.
- This API is available for the event callback of frame grabbers, devices, and stream channels. The type of registered callback function varies according to the handle type.Now the frame grabber offline event is not supported. After upgrading the frame grabber, you need to restart the PC before use.

Chapter 4 Data Structure and Enumeration

4.1 Data Structure

4.1.1 MV_CML_DEVICE_INFO

Structure about Camera Link Device Information

Member	Data Type	Description
chVendorName	unsigned char[]	Device vendor name, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chModelName	unsigned char	Device model name, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chManufacturerInfo	unsigned char	Device manufacturer information, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chDeviceVersion	unsigned char	Device version, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chSerialNumber	unsigned char	Device serial number, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chUserDefinedName	unsigned char	User-defined name, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chDeviceID	unsigned char	Device ID, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
nReserved[48]	unsigned int[]	Reserved.

4.1.2 MV_CML_INTERFACE_INFO

Structure about Camera Link Frame Grabber Information

Member	Data Type	Description
chInterfaceID	unsigned char[]	Frame grabber ID, the length is defined by the macro "MV_FG_MAX_IF_INFO_SIZE" (the value is 64).
chDisplayName	unsigned char[]	Displayed name, the length is defined by the macro "MV_FG_MAX_IF_INFO_SIZE" (the value is 64).
chSerialNumber	unsigned char[]	Serial number, the length is defined by the macro "MV_FG_MAX_IF_INFO_SIZE" (the value is 64).
nPCIEInfo	unsigned int	Information about the PCIe slot of the frame grabber. The lower 16 bits are valid bits: bits 0 to 2 indicate function, bits 3 to 7 indicate device, and bits 8 to 15 indicate bus.
nReserved[64]	unsigned int[]	Reserved.

4.1.3 MV_CXP_DEVICE_INFO

Structure about CoaXPress Device Information

Member	Data Type	Description
chVendorName	unsigned char[]	Device vendor name, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chModelName	unsigned char[]	Device model name, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chManufacturerInfo	unsigned char[]	Device manufacturer information, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chDeviceVersion	unsigned char[]	Device version, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chSerialNumber	unsigned char[]	Device serial number, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chUserDefinedName	unsigned char[]	User-defined name, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chDeviceID	unsigned char[]	Device ID, the length is defined by the macro

Member	Data Type	Description
		"MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
nReserved[48]	unsigned int[]	Reserved.

4.1.4 MV_CXP_INTERFACE_INFO

Structure about CoaXPress Frame Grabber Information

Member	Data Type	Description
chInterfaceID	unsigned char[]	Frame grabber ID, the length is defined by the macro "MV_FG_MAX_IF_INFO_SIZE" (the value is 64).
chDisplayName	unsigned char[]	Displayed name, the length is defined by the macro "MV_FG_MAX_IF_INFO_SIZE" (the value is 64).
chSerialNumber	unsigned char[]	Serial number, the length is defined by the macro "MV_FG_MAX_IF_INFO_SIZE" (the value is 64).
nPCIEInfo	unsigned int	Information about the PCIe slot of the frame grabber. The lower 16 bits are valid bits: bits 0 to 2 indicate function, bits 3 to 7 indicate device, and bits 8 to 15 indicate bus.
nReserved[64]	unsigned int[]	Reserved.

4.1.5 MV_FG_BUFFER_INFO

Structure about Output Frame Buffer Information

Member	Data Type	Description
pBuffer	void*	Image buffer address.
nSize	unsigned int	Size of the image buffer address.
nFilledSize	unsigned int	Frame length.
pPrivate	void*	Private data.
nWidth	unsigned int	Image width.
nHeight	unsigned int	Image height.
enPixelType	***MV_FG_PIXEL_TYPE***	Pixel format.
bNewData	bool	Whether it is a new image.
bQueued	bool	Whether it is in the image acquisition queue.

Member	Data Type	Description
bAcquiring	bool	Whether the image is being acquired.
bIncomplete	bool	Whether the image acquisition is incomplete.
nFrameID	int64_t	Frame No.
nDevTimeStamp	int64_t	Device timestamp.
nHostTimeStamp	int64_t	Host timestamp.
nNumChunks	unsigned int	Number of chunks.
nChunkPayloadSize	unsigned int	Size of chunk payloads.
nSecondCount	unsigned int	Seconds (time scale).
nCycleCount	unsigned int	Cycle count (time scale).
nCycleOffset	unsigned int	Cycle offset (time scale).
fGain	float	Gain.
fExposureTime	float	Exposure time.
nAverageBrightness	unsigned int	Average brightness.
nFrameCounter	unsigned int	Total number of frames.
nTriggerIndex	unsigned int	Trigger count.
nInput	unsigned int	Input.
nOutput	unsigned int	Output.
nRed	unsigned int	Red (white balance).
nGreen	unsigned int	Green (white balance).
nBlue	unsigned int	Blue (white balance).
nOffsetX	unsigned int	ROI x-offset.
nOffsetY	unsigned int	ROI y-offset.
nChunkWidth	unsigned int	ROI width.
nChunkHeight	unsigned int	ROI height.
nReserved[45]	unsigned int[]	Reserved.

4.1.6 MV_FG_CCM_INFO

Structure about Color Correction Matrix (CCM) Information

Member	Data Type	Description
bCCMEnable	bool32_t	Whether to enable CCM.

Member	Data Type	Description
nCCMat[9]	int[]	Color correction matrix. Range of value: (-65536, 65536).
nCCMScale	unsigned int	Quantized coefficient, which is an integral power of 2 and the maximum value is 65536.
nReserved[4]	unsigned int[]	Reserved.

4.1.7 MV_FG_CHUNK_DATA_INFO

Structure about Chunk Data Information

Member	Data Type	Description
pChunkData	unsigned char*	Chunk data.
nChunkID	unsigned int	Chunk ID.
nChunkLen	unsigned int	Chunk length.
nReserved[4]	unsigned int[]	Reserved.

4.1.8 MV_FG_CONVERT_PIXEL_INFO

Structure about Pixel Format Conversion Information

Member	Data Type	Description
stInputImageInfo	***MV_FG_INPUT_IMAGE_INFO***	Input image information.
stOutputImageInfo	***MV_FG_OUTPUT_IMAGE_INFO***	Output image information.
enCfaMethod	***MV_FG_CFA_METHOD***	Interpolation method.
bFilterEnable	bool32_t	Whether smooth interpolation is enabled.
stGammaInfo	***MV_FG_GAMMA_INFO***	Gamma information.
stCCMInfo	***MV_FG_CCM_INFO***	Color Correction Matrix (CCM) information. It is supported by Windows only.
nReserved[4]	unsigned int[]	Reserved.

4.1.9 MV_FG_DEVICE_INFO

Structure about Device Information

Member	Data Type	Description
nDevType	unsigned int	Device type.
nReserved[3]	unsigned int[]	Reserved.

DevInfo	union {
	MV_CXP_DEVICE_INFO stCXPDevInfo;
	MV_GEV_DEVICE_INFO stGEVDevInfo;
	MV_CML_DEVICE_INFO stCMLDevInfo;
	unsigned int nReserved[256];
	}
		Device information. The member to be used is determined by **nDevType**.
		stCXPDevInfo: CoaXPress device information.
		stGEVDevInfo: GigE Vision device information.
		stCMLDevInfo: Camera Link device information.

4.1.10 MV_FG_ENUMVALUE

Structure about Enumeration Type Value

Member	Data Type	Description
nCurValue	unsigned int	Current value.
strCurSymbolic	char[]	The node name (property key) of the current value, the length is defined by the macro "MV_FG_MAX_SYMBOLIC_STRLEN" (the value is 64).
nSupportedNum	unsigned int	The number of supported enumeration types.
nSupportValue	unsigned int[]	The value of supported enumeration types, the number is defined by the macro "MV_FG_MAX_SYMBOLIC_NUM" (the value is 64).
strSymbolic	char[]	The node name (property key) of the value of supported enumeration types. The number is defined by the macro "MV_FG_MAX_SYMBOLIC_NUM" (the value is 64) and the length is defined by the macro "MV_FG_MAX_SYMBOLIC_STRLEN" (the value is 64).

Member	Data Type	Description
nReserved[4]	unsigned int[]	Reserved.

4.1.11 MV_FG_EVENT_INFO

Structure about Event Information

Member	Data Type	Description
EventName	char[]	Event name, the maximum length is 128 bytes (value of macro "MV_FG_MAX_EVENT_NAME_SIZE").
nEventID	unsigned int	Event ID.
nBlockID	uint64_t	Frame No., valid for stream-related events.
nTimestamp	uint64_t	Timestamp.
pEventData	void*	Event data, which is an internal buffer and needs to be processed in time.
nEventDataSize	unsigned int	Length of event data.
nReserved[4]	unsigned int[]	Reserved.

4.1.12 MV_FG_FLOATVALUE

Structure about Float Type Value

Member	Data Type	Description
fCurValue	float	Current value.
fMax	float	The maximum value.
fMin	float	The minimum value.
nReserved[4]	unsigned int[]	Reserved.

4.1.13 MV_FG_FRAME_SPEC_INFO

Structure about Watermark Information

Member	Data Type	Description
nSecondCount	unsigned int	Seconds.

Member	Data Type	Description
nCycleCount	unsigned int	Number of cycles.
nCycleOffset	unsigned int	The offset of a cycle.
fGain	float	Gain.
fExposureTime	float	Exposure time.
nAverageBrightness	unsigned int	Average brightness.
nRed	unsigned int	Red.
nGreen	unsigned int	Green.
nBlue	unsigned int	Blue.
nFrameCounter	unsigned int	Total number of frames.
nTriggerIndex	unsigned int	Trigger counting.
nInput	unsigned int	Input.
nOutput	unsigned int	Output.
nOffsetX	unsigned short	Offset in the x coordinate.
nOffsetY	unsigned short	Offset in the y coordinate.
nFrameWidth	unsigned short	Watermark width.
nFrameHeight	unsigned short	Watermark height.
nReserved[16]	unsigned int[]	Reserved.

4.1.14 MV_FG_GAMMA_INFO

Structure about Gamma Information

Note

When setting the Gamma curve correction, the valid corrected curve is required.

Member	Data Type	Description
enGammaType	***MV_FG_GAMMA_TYPE***	Gamma type.
fGammaValue	float	Gamma value. Value range: [0.1, 4.0]
pGammaCurveBuf	unsigned char*	Gamma curve buffer.
nGammaCurveBufLen	unsigned int	Length of gamma curve.
nReserved[4]	unsigned int[]	Reserved.

4.1.15 MV_FG_HB_DECODE_PARAM

Structure about Lossless Decoding Parameters

Member	Data Type	Description
pSrcBuf	unsigned char*	Input data buffer.
nSrcLen	unsigned int	Input data size.
stOutputImageInfo	***MV_FG_OUTPUT_IMAGE_INFO***	Output image information.
stFrameSpecInfo	***MV_FG_FRAME_SPEC_INFO***	Watermark information. (Not supported.)
nRes[8]	unsigned int[]	Reserved.

4.1.16 MV_FG_INPUT_IMAGE_INFO

Structure about Input Image Information

Member	Data Type	Description
nWidth	unsigned int	Image width.
nHeight	unsigned int	Image height.
enPixelType	***MV_FG_PIXEL_TYPE***	Pixel format.
pImageBuf	unsigned char*	Input image buffer.
nImageBufLen	unsigned int	Input image length.
nReserved[4]	unsigned int[]	Reserved.

4.1.17 MV_FG_DISPLAY_FRAME_INFO

Structure about the Displayed Image Information

Member	Data Type	Description
nWidth	unsigned int	Image width.
nHeight	unsigned int	Image height.
enPixelType	***MV_FG_PIXEL_TYPE***	Pixel format.
pImageBuf	unsigned char*	Input image buffer.
nImageBufLen	unsigned int	Input image length.

Member	Data Type	Description
nReserved[4]	unsigned int[]	Reserved.

4.1.18 MV_FG_INTERFACE_INFO

Structure about Frame Grabber Information

Member	Data Type	Description
nTLayerType	unsigned int	Frame grabber type. See ***Frame Grabber Type*** for details.
nReserved[4]	unsigned int[]	Reserved.

IfaceInfo	union {
	MV_CXP_INTERFACE_INFO stCXPIfaceInfo;
	MV_GEV_INTERFACE_INFO stGEVIfaceInfo;
	MV_GEV_INTERFACE_INFO stCMLIfaceInfo;
	unsigned int nReserved[256];
	}
		Frame grabber information. The member to be used is determined by **nTLayerType**.
		stCXPIfaceInfo: CoaXPress frame grabber information.
		stGEVIfaceInfo: GigE Vision frame grabber information.
		stCMLIfaceInfo: Camera Link frame grabber information.

4.1.19 MV_FG_INTVALUE

Structure about Integer Type Value

Member	Data Type	Description
nCurValue	int64_t	Current value.
nMax	int64_t	The maximum value.
nMin	int64_t	The minimum value.
nInc	int64_t	Increment.
nReserved[16]	unsigned int[]	Reserved.

4.1.20 MV_FG_OUTPUT_IMAGE_INFO

Structure about Output Image Information

Member	Data Type	Description
nWidth	unsigned int	Image width.
nHeight	unsigned int	Image height.
enPixelType	***MV_FG_PIXEL_TYPE***	Pixel format.
pImageBuf	unsigned char*	Image buffer.
nImageBufSize	unsigned int	Image buffer size.
nImageBufLen	unsigned int	Image length.
nReserved[4]	unsigned int[]	Reserved.

4.1.21 MV_FG_RECONSTRUCT_INFO

Structure about Image Reconstruction Information

Member	Data Type	Description
stInputImageInfo	***MV_FG_INPUT_IMAGE_INFO***	Input image information.
stOutputImageInfo	***MV_FG_OUTPUT_IMAGE_INFO***	Output image information, the maximum number of images is defined by the macro "MV_FG_MAX_SPLIT_NUM" (the value is 8).
enReconstructMode	***MV_FG_RECONSTRUCT_MODE***	Image reconstruction mode.
nReserved[4]	unsigned int[]	Reserved.

4.1.22 MV_FG_SAVE_BITMAP_INFO

Structure about BMP Image Saving Information

Member	Data Type	Description
stInputImageInfo	***MV_FG_INPUT_IMAGE_INFO***	Input image information.
pBmpBuf	unsigned char*	Buffer of output BMP image.
nBmpBufSize	unsigned int	Size of output buffer.

Member	Data Type	Description
nBmpBufLen	unsigned int	Length of output BMP image.
enCfaMethod	***MV_FG_CFA_METHOD***	Interpolation method.
nReserved[4]	unsigned int[]	Reserved.

4.1.23 MV_FG_SAVE_JPEG_INFO

Structure about JPEG Image Saving Information

Member	Data Type	Description
stInputImageInfo	***MV_FG_INPUT_IMAGE_INFO***	Input image information.
pJpgBuf	unsigned char*	Buffer of output JPEG image.
nJpgBufSize	unsigned int	Size of output buffer.
nJpgBufLen	unsigned int	Length of output JPEG image.
nJpgQuality	unsigned int	Encoding quality. Range of value: (0, 100].
enCfaMethod	***MV_FG_CFA_METHOD***	Interpolation method.
nReserved[4]	unsigned int[]	Reserved.

4.1.24 MV_FG_SAVE_PNG_TO_FILE_INFO

Structure about PNG Image Saving Information

Member	Data Type	Description
stInputImageInfo	***MV_FG_INPUT_IMAGE_INFO***	Input image information.
pcImagePath	char*	Path of the input image.
nPngCompression	unsigned int	Encoding compression rate, range: [0, 9].
enCfaMethod	***MV_FG_CFA_METHOD***	Interpolation method.
nReserved[4]	unsigned int[]	Reserved.

4.1.25 MV_FG_SAVE_TIFF_TO_FILE_INFO

Structure about TIFF Image Saving Information

Member	Data Type	Description
stInputImageInfo	***MV_FG_INPUT_IMAGE_INFO***	Input image information.
pcImagePath	char*	Path of the input image.
fXResolution	float	Horizontal resolution.
fYResolution	float	Vertical resolution.
enResolutionUnit	***MV_FG_RESOLUTION_UNIT***	Resolution unit.
enCfaMethod	***MV_FG_CFA_METHOD***	Interpolation method.
nReserved[4]	unsigned int[]	Reserved.

4.1.26 MV_FG_STRINGVALUE

Structure about String Type Value

Member	Data Type	Description
strCurValue[256]	char	Current value.
nMaxLength	int64_t	The maximum length.
nReserved[4]	unsigned int[]	Reserved.

4.1.27 MV_GEV_DEVICE_INFO

Structure about GigE Vision Device Information

Member	Data Type	Description
nIpCfgOption	unsigned int	IP configurations supported by the device.
nIpCfgCurrent	unsigned int	Current IP configuration.
nCurrentIp	unsigned int	Current IP address.
nCurrentSubNetMask	unsigned int	Current subnet mask.
nDefultGateWay	unsigned int	Current gateway.
nNetExport	unsigned int	Network interface IP address.
nMacAddress	uint64_t	MAC address.

Member	Data Type	Description
chVendorName	unsigned char[]	Device vendor name, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chModelName	unsigned char[]	Device model name, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chManufacturerInfo	unsigned char[]	Device manufacturer information, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chDeviceVersion	unsigned char[]	Device version, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chSerialNumber	unsigned char[]	Device serial number, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chUserDefinedName	unsigned char[]	User-defined name, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
chDeviceID	unsigned char[]	Device ID, the length is defined by the macro "MV_FG_MAX_DEV_INFO_SIZE" (the value is 64).
nReserved[48]	unsigned int[]	Reserved.

4.1.28 MV_GEV_INTERFACE_INFO

Structure about GigE Vision Frame Grabber Information

Member	Data Type	Description
chInterfaceID	unsigned char[]	Frame grabber ID, the length is defined by the macro "MV_FG_MAX_IF_INFO_SIZE" (the value is 64).
chDisplayName	unsigned char[]	Displayed name, the length is defined by the macro "MV_FG_MAX_IF_INFO_SIZE" (the value is 64).
chSerialNumber	unsigned char[]	Serial number, the length is defined by the macro "MV_FG_MAX_IF_INFO_SIZE" (the value is 64).
nPCIEInfo	unsigned int	Information about the PCIe slot of the

Member	Data Type	Description
		frame grabber. The lower 16 bits are valid bits: bits 0 to 2 indicate function, bits 3 to 7 indicate device, and bits 8 to 15 indicate bus.
nReserved[64]	unsigned int[]	Reserved.

4.2 Enumeration

4.2.1 MV_FG_BUFFER_QUEUE_TYPE

Enumeration about the types of buffer queues.

Enumeration Type	Value	Description
MV_FG_BUFFER_QUEUE_INPUT_TO_OUTPUT	0	Flush buffers from the input queue to the output queue.
MV_FG_BUFFER_QUEUE_OUTPUT_DISCARD	1	Discard buffers in the output queue.
MV_FG_BUFFER_QUEUE_ALL_TO_INPUT	2	Flush all buffers (including those in the output queue) to the input queue.
MV_FG_BUFFER_QUEUE_UNQUEUED_TO_INPUT	3	Flush unused buffers to the input queue.
MV_FG_BUFFER_QUEUE_ALL_DISCARD	4	Discard all queued buffers (those in the input queue and the output queue).

4.2.2 MV_FG_CFA_METHOD

Enumeration about the Color Filter Array (CFA) interpolation methods.

Enumeration Type	Value	Description
MV_FG_CFA_METHOD_QUICK	0	Quick interpolation.
MV_FG_CFA_METHOD_BALANCE	1	Balanced interpolation.
MV_FG_CFA_METHOD_OPTIMAL	2	Optimal interpolation.

4.2.3 MV_FG_CONFIG_CMD

Enumeration about the configuration command.

Enumeration Type	Value	Description
CONFIG_CMD_INT64_BAUDRATE	1	Baud rate (integer type).

4.2.4 MV_FG_EXCEPTION_TYPE

Enumeration about Exception Types

Enumeration Type	Value	Description
EXCEPTION_TYPE_SYSTEM_TEMPERATURE_UPLIMIT	0x0080	The temperature reached the upper limit.
EXCEPTION_TYPE_SYSTEM_TEMPERATURE_LOWLIMIT	0x0081	The temperature reached the lower limit.
EXCEPTION_TYPE_SYSTEM_DDR_INIT	0x0082	DDR initialization failed.
EXCEPTION_TYPE_CARD_PACKETBUF_ERR	0x0180	Packet buffer error.
EXCEPTION_TYPE_CARD_ACKPACKETBUF_ERR	0x0181	Response packet buffer error.
EXCEPTION_TYPE_LINK0_STREAM_CRC_ERR	0x1080	Link0 stream CRC verification error.
EXCEPTION_TYPE_LINK0_STREAM_PACKET_RESEND	0x1081	Link0 stream packet resending.
EXCEPTION_TYPE_LINK0_STREAM_CTRLPACKET_ERR	0x1082	Link0 control packet error.
EXCEPTION_TYPE_LINK0_PRETREATBUF_ERR	0x1090	Link0 pretreatment buffer error.
EXCEPTION_TYPE_LINK0_CAM_ACK_RECVBUF_ERR	0x1091	Buffer of receiving Link0 camera ack packet error.
EXCEPTION_TYPE_LINK0_CAM_ACK_TRANSMITBUF_ERR	0x1092	Buffer of transmitting Link0 camera packet error.
EXCEPTION_TYPE_LINK1_STREAM_CRC_ERR	0x1180	Link1 stream CRC verification error.
EXCEPTION_TYPE_LINK1_STREAM_PACKET_RESEND	0x1181	Link1 stream packet resending.
EXCEPTION_TYPE_LINK1_STREAM_CTRLPACKET_ERR	0x1182	Link1 control packet error.
EXCEPTION_TYPE_LINK1_PRETREATBUF_ERR	0x1190	Link1 pretreatment buffer error.

Enumeration Type	Value	Description
EXCEPTION_TYPE_LINK1_CAM_ACK_RECVBUF_ERR	0x1191	Buffer of receiving Link1 camera ack packet error.
EXCEPTION_TYPE_LINK1_CAM_ACK_TRANSMITBUF_ERR	0x1192	Buffer of transmitting Link1 camera packet error.
EXCEPTION_TYPE_LINK2_STREAM_CRC_ERR	0x1280	Link2 stream CRC verification error.
EXCEPTION_TYPE_LINK2_STREAM_PACKET_RESEND	0x1281	Link2 stream packet resending.
EXCEPTION_TYPE_LINK2_STREAM_CTRLPACKET_ERR	0x1282	Link2 control packet error.
EXCEPTION_TYPE_LINK2_PRETREATBUF_ERR	0x1290	Link2 pretreatment buffer error.
EXCEPTION_TYPE_LINK2_CAM_ACK_RECVBUF_ERR	0x1291	Buffer of receiving Link2 camera ack packet error.
EXCEPTION_TYPE_LINK2_CAM_ACK_TRANSMITBUF_ERR	0x1292	Buffer of transmitting Link2 camera packet error.
EXCEPTION_TYPE_LINK3_STREAM_CRC_ERR	0x1380	Link3 stream CRC verification error.
EXCEPTION_TYPE_LINK3_STREAM_PACKET_RESEND	0x1381	Link3 stream packet resending.
EXCEPTION_TYPE_LINK3_STREAM_CTRLPACKET_ERR	0x1382	Link3 control packet error.
EXCEPTION_TYPE_LINK3_PRETREATBUF_ERR	0x1390	Link3 pretreatment buffer error.
EXCEPTION_TYPE_LINK3_CAM_ACK_RECVBUF_ERR	0x1391	Buffer of receiving Link3 camera ack packet error.
EXCEPTION_TYPE_LINK3_CAM_ACK_TRANSMITBUF_ERR	0x1392	Buffer of transmitting Link3 camera packet error.
EXCEPTION_TYPE_STREAM0_DROP_FRAME_IMAGE	0x2080	Stream0 channel frame dropping.
EXCEPTION_TYPE_STREAM0_IMAGE_DATACOUNT_ERR	0x2081	Image data counting error of Stream0 channel.
EXCEPTION_TYPE_STREAM0_DROP_FRAME_TRIGGER	0x2082	Stream0 channel frame dropping triggered.
EXCEPTION_TYPE_STREAM0_QUEUEBUF_ERR	0x2090	Stream0 QUEUE buffer error.
EXCEPTION_TYPE_STREAM0_WDMABUF_ERR	0x2091	Stream0 WDMA buffer error.
EXCEPTION_TYPE_STREAM0_RDMABUF_ERR	0x2092	Stream0 RDMA buffer error.
EXCEPTION_TYPE_STREAM0_PACKETBUF_ERR	0x2093	Stream0 PACKET buffer error.
EXCEPTION_TYPE_STREAM0_WDMALENGTH_ERR	0x2094	Stream0 WDMA length error.

Enumeration Type	Value	Description
EXCEPTION_TYPE_STREAM0_RDMALENGTH_ERR	0x2095	Stream0 RDMA length error.
EXCEPTION_TYPE_STREAM1_DROP_FRAME_IMAGE	0x2180	Stream1 channel frame dropping.
EXCEPTION_TYPE_STREAM1_IMAGE_DATACOUNT_ERR	0x2181	Image data counting error of Stream1 channel.
EXCEPTION_TYPE_STREAM1_DROP_FRAME_TRIGGER	0x2182	Stream1 channel frame dropping triggered.
EXCEPTION_TYPE_STREAM1_QUEUEBUF_ERR	0x2190	Stream1 QUEUE buffer error.
EXCEPTION_TYPE_STREAM1_WDMABUF_ERR	0x2191	Stream1 WDMA buffer error.
EXCEPTION_TYPE_STREAM1_RDMABUF_ERR	0x2192	Stream1 RDMA buffer error.
EXCEPTION_TYPE_STREAM1_PACKETBUF_ERR	0x2193	Stream1 PACKET buffer error.
EXCEPTION_TYPE_STREAM1_WDMALENGTH_ERR	0x2194	Stream1 WDMA length error.
EXCEPTION_TYPE_STREAM1_RDMALENGTH_ERR	0x2195	Stream1 RDMA length error.
EXCEPTION_TYPE_STREAM2_DROP_FRAME_IMAGE	0x2280	Stream2 channel frame dropping.
EXCEPTION_TYPE_STREAM2_IMAGE_DATACOUNT_ERR	0x2281	Image data counting error of Stream2 channel.
EXCEPTION_TYPE_STREAM2_DROP_FRAME_TRIGGER	0x2282	Stream2 channel frame dropping triggered.
EXCEPTION_TYPE_STREAM2_QUEUEBUF_ERR	0x2290	Stream2 QUEUE buffer error.
EXCEPTION_TYPE_STREAM2_WDMABUF_ERR	0x2291	Stream2 WDMA buffer error.
EXCEPTION_TYPE_STREAM2_RDMABUF_ERR	0x2292	Stream2 RDMA buffer error.
EXCEPTION_TYPE_STREAM2_PACKETBUF_ERR	0x2293	Stream2 PACKET buffer error.
EXCEPTION_TYPE_STREAM2_WDMALENGTH_ERR	0x2294	Stream2 WDMA length error.
EXCEPTION_TYPE_STREAM2_RDMALENGTH_ERR	0x2295	Stream2 RDMA length error.
EXCEPTION_TYPE_STREAM3_DROP_FRAME_IMAGE	0x2380	Stream3 channel frame dropping.
EXCEPTION_TYPE_STREAM3_IMAGE_DATACOUNT_ERR	0x2381	Image data counting error of Stream3 channel.
EXCEPTION_TYPE_STREAM3_DROP_FRAME_TRIGGER	0x2382	Stream3 channel frame dropping triggered.
EXCEPTION_TYPE_STREAM3_QUEUEBUF_ERR	0x2390	Stream3 QUEUE buffer error.
EXCEPTION_TYPE_STREAM3_WDMABUF_ERR	0x2391	Stream3 WDMA buffer error.
EXCEPTION_TYPE_STREAM3_RDMABUF_ERR	0x2392	Stream3 RDMA buffer error.
EXCEPTION_TYPE_STREAM3_PACKETBUF_ERR	0x2393	Stream3 PACKET buffer error.

Enumeration Type	Value	Description
EXCEPTION_TYPE_STREAM3_WDMALENGTH_ERR	0x2394	Stream3 WDMA length error.
EXCEPTION_TYPE_STREAM3_RDMALENGTH_ERR	0x2395	Stream3 RDMA length error.
EXCEPTION_TYPE_PCIE_SCHEDULEBUF_ERR	0x3088	Scheduling module error.
EXCEPTION_TYPE_PCIE_SCHEDULE_ERR	0x3089	Scheduling result error.
EXCEPTION_TYPE_PCIE_LINK0_RECVBUF_ERR	0x3180	Link0 receiving buffer error.
EXCEPTION_TYPE_PCIE_LINK0_LENGTH_ERR	0x3181	Link0 control packet length error.
EXCEPTION_TYPE_PCIE_LINK0_SOFT_RECVBUF_ERR	0x3280	Link0 soft receiving buffer error.
EXCEPTION_TYPE_PCIE_LINK0_SOFT_LENGTH_ERR	0x3281	Link0 soft control packet length error.
EXCEPTION_TYPE_PCIE_LINK1_RECVBUF_ERR	0x3188	Link1 receiving buffer error.
EXCEPTION_TYPE_PCIE_LINK1_LENGTH_ERR	0x3189	Link1 control packet length error.
EXCEPTION_TYPE_PCIE_LINK1_SOFT_RECVBUF_ERR	0x3288	Link1 soft receiving buffer error.
EXCEPTION_TYPE_PCIE_LINK1_SOFT_LENGTH_ERR	0x3289	Link1 soft control packet length error.
EXCEPTION_TYPE_PCIE_LINK2_RECVBUF_ERR	0x3190	Link2 receiving buffer error.
EXCEPTION_TYPE_PCIE_LINK2_LENGTH_ERR	0x3191	Link2 control packet length error.
EXCEPTION_TYPE_PCIE_LINK2_SOFT_RECVBUF_ERR	0x3290	Link2 soft receiving buffer error.
EXCEPTION_TYPE_PCIE_LINK2_SOFT_LENGTH_ERR	0x3291	Link2 soft control packet length error.
EXCEPTION_TYPE_PCIE_LINK3_RECVBUF_ERR	0x3198	Link3 receiving buffer error.
EXCEPTION_TYPE_PCIE_LINK3_LENGTH_ERR	0x3199	Link3 control packet length error.
EXCEPTION_TYPE_PCIE_LINK3_SOFT_RECVBUF_ERR	0x3298	Link3 soft receiving buffer error.
EXCEPTION_TYPE_PCIE_LINK3_SOFT_LENGTH_ERR	0x3299	Link3 soft control packet length error.
EXCEPTION_TYPE_PCIE_STREAM0_RECVBUF_ERR	0x3382	FIFO error of stream in Stream0 buffer.
EXCEPTION_TYPE_PCIE_STREAM0_LIST_ERR	0x3383	Invalid list format of Stream0.
EXCEPTION_TYPE_PCIE_STREAM0_SIZE_ERR	0x3384	Stream0 image size and memory mismatched.
EXCEPTION_TYPE_PCIE_STREAM1_RECVBUF_ERR	0x338A	FIFO error of stream in Stream1 buffer.
EXCEPTION_TYPE_PCIE_STREAM1_LIST_ERR	0x338B	Invalid list format of Stream1.

Enumeration Type	Value	Description
EXCEPTION_TYPE_PCIE_STREAM1_SIZE_ERR	0x338C	Stream1 image size and memory mismatched.
EXCEPTION_TYPE_PCIE_STREAM2_RECVBUF_ERR	0x3392	FIFO error of stream in Stream2 buffer.
EXCEPTION_TYPE_PCIE_STREAM2_LIST_ERR	0x3393	Invalid list format of Stream2.
EXCEPTION_TYPE_PCIE_STREAM2_SIZE_ERR	0x3394	Stream2 image size and memory mismatched.
EXCEPTION_TYPE_PCIE_STREAM3_RECVBUF_ERR	0x339A	FIFO error of stream in Stream3 buffer.
EXCEPTION_TYPE_PCIE_STREAM3_LIST_ERR	0x339B	Invalid list format of Stream3.
EXCEPTION_TYPE_PCIE_STREAM3_SIZE_ERR	0x339C	Stream3 image size and memory mismatched.
EXCEPTION_TYPE_CAMERA_DISCONNECT_ERR	0x10000001	Camera disconnected.

4.2.5 MV_FG_GAMMA_TYPE

Enumeration about the Gamma types.

Enumeration Type	Value	Description
MV_FG_GAMMA_TYPE_NONE	0	Disabled.
MV_FG_GAMMA_TYPE_VALUE	1	Gamma value.
MV_FG_GAMMA_TYPE_USER_CURVE	2	Gamma curve, which has the following possibilities: ● When the output image is 8-bit, the curve length is 256 * size of (unsigned char); ● When the output image is 10-bit, the curve length varies according to the input image. ○ Source image format is 10-bit, the curve length is 1024*sizeof(unsigned short); ○ Source image format is 12-bit, the curve length is 4096*sizeof(unsigned short); ○ Source image format is 16-bit, the curve length is 65536*sizeof(unsigned short).

Enumeration Type	Value	Description
MV_FG_GAMMA_TYPE_LRGB2SRGB	3	Linear RGB to sRGB conversion.
MV_FG_GAMMA_TYPE_SRGB2LRGB	4	sRGB to linear RGB conversion. It is supported only for color interpolation and is invalid for color correction.

4.2.6 MV_FG_NODE_ACCESS_MODE

Enumeration about the access modes of a node.

Enumeration Type	Value	Description
ACCESS_MODE_NI	0	Not implemented.
ACCESS_MODE_NA	1	Not available.
ACCESS_MODE_WO	2	Write only.
ACCESS_MODE_RO	3	Read only.
ACCESS_MODE_RW	4	Read and write.
ACCESS_MODE_UNDEFINED	5	Undefined.

4.2.7 MV_FG_NODE_INTERFACE_TYPE

Enumeration about the types of a node.

Enumeration Type	Value	Description
INTERFACE_TYPE_Value	0	Value
INTERFACE_TYPE_Base	1	Base
INTERFACE_TYPE_Integer	2	Integer
INTERFACE_TYPE_Boolean	3	Boolean
INTERFACE_TYPE_Command	4	Command
INTERFACE_TYPE_Float	5	Float
INTERFACE_TYPE_String	6	String
INTERFACE_TYPE_Register	7	Register
INTERFACE_TYPE_Category	8	Category
INTERFACE_TYPE_Enumeration	9	Enumeration

Enumeration Type	Value	Description
INTERFACE_TYPE_EnumEntry	10	EnumEntry
INTERFACE_TYPE_Port	11	Port

4.2.8 MV_FG_PIXEL_TYPE

Enumeration about the pixel formats.

Enumeration Type	Value	Description
Undefined Format		
MV_FG_PIXEL_TYPE_Undefined	0xFFFFFFFF	Undefined format.
Mono Format		
MV_FG_PIXEL_TYPE_Mono8	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(8) | 0x0001	Mono8
MV_FG_PIXEL_TYPE_Mono10	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x0003	Mono10
MV_FG_PIXEL_TYPE_Mono10_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x0004	Mono10_Packed
MV_FG_PIXEL_TYPE_Mono12	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x0005	Mono12
MV_FG_PIXEL_TYPE_Mono12_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x0006	Mono12_Packed
MV_FG_PIXEL_TYPE_Mono16	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x0007	Mono16
Bayer Format		
MV_FG_PIXEL_TYPE_BayerGR8	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(8) | 0x0008	BayerGR8
MV_FG_PIXEL_TYPE_BayerRG8	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(8) | 0x0009	BayerRG8
MV_FG_PIXEL_TYPE_BayerGB8	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(8) | 0x000A	BayerGB8
MV_FG_PIXEL_TYPE_BayerBG8	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(8) | 0x000B	BayerBG8
MV_FG_PIXEL_TYPE_BayerGR10	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x000C	BayerGR10
MV_FG_PIXEL_TYPE_BayerRG10	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x000D	BayerRG10
MV_FG_PIXEL_TYPE_BayerGB10	MV_FG_PIXEL_MONO |	BayerGB10

Enumeration Type	Value	Description
	MV_FG_PIXEL_BIT_COUNT(16) | 0x000E	
MV_FG_PIXEL_TYPE_BayerBG10	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x000F	BayerBG10
MV_FG_PIXEL_TYPE_BayerGR12	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x0010	BayerGR12
MV_FG_PIXEL_TYPE_BayerRG12	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x0011	BayerRG12
MV_FG_PIXEL_TYPE_BayerGB12	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x0012	BayerGB12
MV_FG_PIXEL_TYPE_BayerBG12	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x0013	BayerBG12
MV_FG_PIXEL_TYPE_BayerGR10_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x0026	BayerGR10_Packed
MV_FG_PIXEL_TYPE_BayerRG10_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x0027	BayerRG10_Packed
MV_FG_PIXEL_TYPE_BayerGB10_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x0028	BayerGB10_Packed
MV_FG_PIXEL_TYPE_BayerBG10_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x0029	BayerBG10_Packed
MV_FG_PIXEL_TYPE_BayerGR12_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x002A	BayerGR12_Packed
MV_FG_PIXEL_TYPE_BayerRG12_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x002B	BayerRG12_Packed
MV_FG_PIXEL_TYPE_BayerGB12_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x002C	BayerGB12_Packed
MV_FG_PIXEL_TYPE_BayerBG12_Packed	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(12) | 0x002D	BayerBG12_Packed
MV_FG_PIXEL_TYPE_BayerGR16	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x002E	BayerGR16
MV_FG_PIXEL_TYPE_BayerRG16	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x002F	BayerRG16
MV_FG_PIXEL_TYPE_BayerGB16	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x0030	BayerGB16
MV_FG_PIXEL_TYPE_BayerBG16	MV_FG_PIXEL_MONO | MV_FG_PIXEL_BIT_COUNT(16) | 0x0031	BayerBG16
RGB Format		
MV_FG_PIXEL_TYPE_RGB8_Packed	MV_FG_PIXEL_COLOR | MV_FG_PIXEL_BIT_COUNT(24) | 0x0014	RGB8_Packed
MV_FG_PIXEL_TYPE_BGR8_Packe	MV_FG_PIXEL_COLOR |	BGR8_Packed

Enumeration Type	Value	Description
d	MV_FG_PIXEL_BIT_COUNT(24) | 0x0015	
MV_FG_PIXEL_TYPE_RGBA8_Packed	MV_FG_PIXEL_COLOR | MV_FG_PIXEL_BIT_COUNT(32) | 0x0016	RGBA8_Packed
MV_FG_PIXEL_TYPE_BGRA8_Packed	MV_FG_PIXEL_COLOR | MV_FG_PIXEL_BIT_COUNT(32) | 0x0017	BGRA8_Packed
MV_FG_PIXEL_TYPE_RGB16_Packed	MV_FG_PIXEL_COLOR | MV_FG_PIXEL_BIT_COUNT(48) | 0x0033	RGB16_Packed
YUV Format		
MV_FG_PIXEL_TYPE_YUV422_Packed	MV_FG_PIXEL_COLOR | MV_FG_PIXEL_BIT_COUNT(16) | 0x001F	YUV422_Packed
MV_FG_PIXEL_TYPE_YUV422_YUYV_Packed	MV_FG_PIXEL_COLOR | MV_FG_PIXEL_BIT_COUNT(16) | 0x0032	YUV422_YUYV_Packed

4.2.9 MV_FG_RECONSTRUCT_MODE

Enumeration about the image reconstruction modes.

Enumeration Type	Value	Description
Rotation Mode The supported pixel formats of rotation mode are: *MV_FG_PIXEL_TYPE_Mono8*, *MV_FG_PIXEL_TYPE_RGB8_Packed*, and *MV_FG_PIXEL_TYPE_BGR8_Packed*.		
RECONSTRUCT_MODE_ROTATE_90	MV_FG_ROTATE_MODE | 0x0001	Rotate 90 degrees.
RECONSTRUCT_MODE_ROTATE_180	MV_FG_ROTATE_MODE | 0x0002	Rotate 180 degrees.
RECONSTRUCT_MODE_ROTATE_270	MV_FG_ROTATE_MODE | 0x0003	Rotate 270 degrees.
Flip Mode The supported pixel formats of flip mode are: *MV_FG_PIXEL_TYPE_Mono8*, *MV_FG_PIXEL_TYPE_RGB8_Packed*, and *MV_FG_PIXEL_TYPE_BGR8_Packed*.		
RECONSTRUCT_MODE_FLIP_VERTICAL	MV_FG_FLIP_MODE | 0x0001	Vertical flip.
RECONSTRUCT_MODE_FLIP_HORIZONTAL	MV_FG_FLIP_MODE | 0x0002	Horizontal flip.
Split by Row This mode is supported by line scan cameras only.		
RECONSTRUCT_MODE_SPLIT_BY_LINE_2	MV_FG_SPLIT_BY_LINE_MODE | 0x0002	Split into 2 images by row.

Enumeration Type	Value	Description
RECONSTRUCT_MODE_SPLIT_BY_LINE_3	MV_FG_SPLIT_BY_LINE_MODE | 0x0003	Split into 3 images by row.
RECONSTRUCT_MODE_SPLIT_BY_LINE_4	MV_FG_SPLIT_BY_LINE_MODE | 0x0004	Split into 4 images by row.

4.2.10 MV_FG_RESOLUTION_UNIT

Enumeration About Resolution Unit

Enumeration Type	Macro Definition Value	Description
MV_FG_Resolution_Unit_None	1	No unit.
MV_FG_Resolution_Unit_Inch	2	Inch.
MV_FG_Resolution_Unit_CENTIMETER	3	Centimeter.

Chapter 5 Macro Definition

Table 5-1 Handle Type

Type Definition	Description
IFHANDLE	Frame grabber handle.
DEVHANDLE	Device handle.
STREAMHANDLE	Stream channel handle.
BUFFERHANDLE	Buffer handle.
PORTHANDLE	Parameter handle. It is the handle of the object to which the parameter belongs; the object can be a frame grabber, device, or stream channel.
IMAGEHANDLE	Image handle. It is the handle of the object to which the image belongs; the object can be a frame grabber, device, or stream channel.

Table 5-2 Frame Grabber Type

Macro	Value	Description
MV_FG_GEV_INTERFACE	0x00000001	GigE Vision frame grabber.
MV_FG_U3V_INTERFACE	0x00000002	USB3 Vision frame grabber.
MV_FG_CAMERALINK_INTERFACE	0x00000004	Camera Link frame grabber.
MV_FG_CXP_INTERFACE	0x00000008	CoaXPress frame grabber.

Table 5-3 Frame Grabber Access Mode

Macro	Value	Description
MV_FG_ACCESS_UNKNOWN	0x0	Permission undefined.
MV_FG_ACCESS_READONLY	0x1	Read only; no permission to set or get node values.
MV_FG_ACCESS_CONTROL	0x2	Permission to control.

Table 5-4 Device Type

Macro	Value	Description
MV_FG_GEV_DEVICE	0x00000001	GigE Vision device.
MV_FG_U3V_DEVICE	0x00000002	USB3 Vision device.
MV_FG_CAMERALINK_DEVICE	0x00000003	Camera Link device.
MV_FG_CXP_DEVICE	0x00000004	CoaXPress device.

Table 5-5 Pixel Format

Macro	Value	Description
MV_FG_PIXEL_MONO	0x01000000	Monochrome format.
MV_FG_PIXEL_COLOR	0x02000000	Color format.
MV_FG_PIXEL_CUSTOM	0x80000000	Custom format.
MV_FG_PIXEL_BIT_COUNT(n)	((n) << 16)	Location of the number of bits.

Table 5-6 Flag Bit of GigE Vision Device

Macro	Value	Description
MV_FG_GEV_IFCONFIG_LLA_BIT	0x00000004	Whether LLA is enabled.
MV_FG_GEV_IFCONFIG_DHCP_BIT	0x00000002	Whether DHCP is enabled.
MV_FG_GEV_IFCONFIG_PERSISTENT_BIT	0x00000001	Whether static IP is enabled.
MV_FG_GEV_IFCONFIG_PR_BIT	0x80000000	Whether pause frames can be processed.
MV_FG_GEV_IFCONFIG_PG_BIT	0x40000000	Whether pause frames can be generated.

Table 5-7 Maximum Value

Macro	Value	Description
MV_FG_MAX_IF_INFO_SIZE	64	The maximum length of string for frame grabber information.
MV_FG_MAX_DEV_INFO_SIZE	64	The maximum length of string for device information.
MV_FG_MAX_EVENT_NAME_SIZE	128	The maximum length of event name.
MV_FG_MAX_SYMBOLIC_NUM	64	The maximum number of node names (property keys) for the XML description file.
MV_FG_MAX_SYMBOLIC_STRLEN	64	The maximum length for node names (property keys) of the XML description file.

Table 5-8 Image Reconstruction

Macro	Value	Description
MV_FG_MAX_SPLIT_NUM	8	The maximum number of images to split a source image into.
MV_FG_ROTATE_MODE	0x1000	Rotation mode.
MV_FG_FLIP_MODE	0x2000	Flip mode.

Macro	Value	Description
MV_FG_SPLIT_BY_LINE_MODE	0x3000	Split by row.

Table 5-9 Baud Rate

Macro	Value	Description
MV_FG_BAUDRATE_9600	0x00000001	9600
MV_FG_BAUDRATE_19200	0x00000002	19200
MV_FG_BAUDRATE_38400	0x00000004	38400
MV_FG_BAUDRATE_57600	0x00000008	57600
MV_FG_BAUDRATE_115200	0x00000010	115200
MV_FG_BAUDRATE_230400	0x00000020	230400
MV_FG_BAUDRATE_460800	0x00000040	460800
MV_FG_BAUDRATE_921600	0x00000080	921600
MV_FG_BAUDRATE_AUTOMAX	0x40000000	Auto-negotiated maximum value supported by the camera.

Appendix A. Sample Code

A.1 Acquire Images with Callback Function

The following sample codes show how to acquire images using callback functions.

```
#include <stdio.h>
#include <Windows.h>
#include <process.h>
#include <conio.h>
#include "MVFGControl.h"

#define BUFFER_NUMBER            3          // Number of requested buffers
#define FILE_NAME_LEN            256        // The maximum length of file name

// User-defined parameters.
typedef struct _Callback_UserParam_
{
    DEVHANDLE                    hDevice;           // Device handle
    MV_FG_SAVE_JPEG_INFO         stSaveJpegInfo;    // JPEG image saving information
}Callback_User;

// Wait for key press.
void WaitForKeyPress(void)
{
    while(!_kbhit())
    {
        Sleep(10);
    }
    _getch();
}

// Clear residual data from stdin.
void ClearStdin(void)
{
    char c = '\0';

    while (1)
    {
        c = getchar();
        if ('\n' == c || EOF == c)
        {
            break;
        }
    }
}
```

```
// Exception information callback function.
void ExceptionCb(MV_FG_EXCEPTION_TYPE enExceptionType, void* pUser)
{
	switch(enExceptionType)
	{
	case EXCEPTION_TYPE_INTERFACE_DISCONNECT:
		{
			printf("Exception: Interface Disconnected!\n");
			break;
		}
	case EXCEPTION_TYPE_DEVICE_DISCONNECT:
		{
			printf("Exception: Device Disconnected!\n");
			break;
		}
	case EXCEPTION_TYPE_STREAM_ABNORMAL_IMAGE:
		{
			printf("Exception: Abnormal Image!\n");
			break;
		}
	case EXCEPTION_TYPE_STREAM_LIST_OVERFLOW:
		{
			printf("Exception: Buffer List Overflow, Clear the Oldest Frame!\n");
			break;
		}
	case EXCEPTION_TYPE_STREAM_DISCONNECTED:
		{
			printf("Exception: Stream Disconnected!\n");
			break;
		}
	default:
		{
			printf("Unknown Exception!\n");
			break;
		}
	}
}


// Save the original JPEG image data.
void SaveJpegImage(unsigned char* pJpgBuf, unsigned int nJpegSize)
{
	if (NULL != pJpgBuf && 0 < nJpegSize)
	{
		char szFileName[FILE_NAME_LEN] = { 0 };

		SYSTEMTIME sys;
		GetLocalTime(&sys);
		sprintf_s(szFileName, FILE_NAME_LEN, "Image_%04d%02d%02d%02d%02d%02d%04d.jpg",
sys.wYear, sys.wMonth,
			sys.wDay, sys.wHour, sys.wMinute, sys.wSecond, sys.wMilliseconds);
```

```
            FILE* pImageFile = NULL;
            if ((0 != fopen_s(&pImageFile, szFileName, "wb")) || (NULL == pImageFile))
            {
                return;
            }

            fwrite(pJpgBuf, 1, nJpegSize, pImageFile);
            fclose(pImageFile);
        }
}


// Frame buffer information callback function.
void FrameCb(MV_FG_BUFFER_INFO* pstBufferInfo, void* pUser)
{
    if (pstBufferInfo && pUser)
    {
        Callback_User* pstUser                  = (Callback_User*)pUser;
        int nRet                                = 0;
        DEVHANDLE hDevice                           = pstUser->hDevice;
        MV_FG_SAVE_JPEG_INFO stSaveJpegInfo = pstUser->stSaveJpegInfo;

        printf("FrameNumber:%2I64d%, Width:%d, Height:%d\n", pstBufferInfo->nFrameID,
pstBufferInfo->nWidth, pstBufferInfo->nHeight);
        stSaveJpegInfo.stInputImageInfo.pImageBuf = (unsigned char*)pstBufferInfo->pBuffer;
        stSaveJpegInfo.stInputImageInfo.nImageBufLen = pstBufferInfo->nFilledSize;
        stSaveJpegInfo.stInputImageInfo.nHeight = pstBufferInfo->nHeight;
        stSaveJpegInfo.stInputImageInfo.nWidth = pstBufferInfo->nWidth;
        stSaveJpegInfo.stInputImageInfo.enPixelType = pstBufferInfo->enPixelType;

        unsigned int nSize = pstBufferInfo->nHeight * pstBufferInfo->nWidth * 2;
        if (stSaveJpegInfo.nJpgBufSize < nSize)
        {
            if (stSaveJpegInfo.pJpgBuf)
            {
                free (stSaveJpegInfo.pJpgBuf);
                stSaveJpegInfo.pJpgBuf = NULL;
            }
            stSaveJpegInfo.pJpgBuf = (unsigned char*)malloc(nSize);
            if (NULL == stSaveJpegInfo.pJpgBuf)
            {
                printf("malloc pConvertData fail!\n");
                nRet = MV_FG_ERR_RESOURCE_EXHAUSTED;
                return;
            }
            stSaveJpegInfo.nJpgBufSize = nSize;
        }

        stSaveJpegInfo.nJpgBufLen = 0;
        stSaveJpegInfo.enCfaMethod = MV_FG_CFA_METHOD_OPTIMAL;
```

```
            stSaveJpegInfo.nJpgQuality = 60;

            nRet = MV_FG_SaveJpeg(hDevice, &stSaveJpegInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Save Jpeg failed! %#x\n", nRet);
            }
            else
            {
                SaveJpegImage(stSaveJpegInfo.pJpgBuf, stSaveJpegInfo.nJpgBufLen);
            }
        }
        return;
}

// Print frame grabber information.
bool PrintInterfaceInfo(unsigned int nInterfaceNum)
{
        int nRet = 0;

        for (unsigned int i = 0; i < nInterfaceNum; i++)
        {
            MV_FG_INTERFACE_INFO stInterfaceInfo = { 0 };

            nRet = MV_FG_GetInterfaceInfo(i, &stInterfaceInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get info of No.%d interface failed! %#x\n", i, nRet);
                return false;
            }

            switch (stInterfaceInfo.nTLayerType)
            {
            case MV_FG_CXP_INTERFACE:
                {
                    printf("[CXP]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                        stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chDisplayName,
                        stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chInterfaceID,
                        stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chSerialNumber);
                    break;
                }
            case MV_FG_GEV_INTERFACE:
                {
                    printf("[GEV]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                        stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chDisplayName,
                        stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chInterfaceID,
                        stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chSerialNumber);
                    break;
                }
```

```
            case MV_FG_CAMERALINK_INTERFACE:
                {
                    printf("[CML]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                        stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chDisplayName,
                        stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chInterfaceID,
                        stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chSerialNumber);
                    break;
                }
            default:
                {
                    printf("Unknown interface type.\n");
                    return false;
                }
            }
        }

        return true;
}

// Print device information.
bool PrintDeviceInfo(IFHANDLE hInterface, unsigned int nDeviceNum)
{
        int nRet = 0;

        for (unsigned int i = 0; i < nDeviceNum; i++)
        {
            MV_FG_DEVICE_INFO stDeviceInfo = { 0 };

            nRet = MV_FG_GetDeviceInfo(hInterface, i, &stDeviceInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get info of No.%d device failed! %#x\n", i, nRet);
                return false;
            }

            switch (stDeviceInfo.nDevType)
            {
            case MV_FG_CXP_DEVICE:
                {
                    printf("[CXP]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chSerialNumber);
                    break;
                }
            case MV_FG_GEV_DEVICE:
                {
                    printf("[GEV]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
```

```
                        stDeviceInfo.DevInfo.stGEVDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chSerialNumber);
                    break;
                }
            case MV_FG_CAMERALINK_DEVICE:
                {
                    printf("[CML]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                        stDeviceInfo.DevInfo.stCMLDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stCMLDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stCMLDevInfo.chSerialNumber);
                    break;
                }
            default:
                {
                    printf("Unknown device type.\n");
                    return false;
                }
            }
        }

        return true;
}

int main()
{
        int                     nRet = 0;
        IFHANDLE            hInterface = NULL;
        DEVHANDLE           hDevice = NULL;
        STREAMHANDLE        hStream = NULL;

        do
        {
            // Enumerate frame grabbers.
            bool bChanged = false;
            nRet = MV_FG_UpdateInterfaceList(MV_FG_CXP_INTERFACE | MV_FG_GEV_INTERFACE |
MV_FG_CAMERALINK_INTERFACE, &bChanged);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Update interface list failed! %#x\n", nRet);
                break;
            }

            // Get the number of frame grabbers.
            unsigned int nInterfaceNum = 0;
            nRet = MV_FG_GetNumInterfaces(&nInterfaceNum);
            if (MV_FG_SUCCESS != nRet || 0 == nInterfaceNum)
            {
                printf("No interface found! return = %d, number = %d\n", nRet, nInterfaceNum);
                break;
```

```
    }

    // Display frame grabber information.
    if (false == PrintInterfaceInfo(nInterfaceNum))
    {
        break;
    }

    // Select frame grabber.
    int nInterfaceIndex = -1;
    printf("Select an interface: ");
    scanf_s("%d", &nInterfaceIndex);
    ClearStdin();

    if (nInterfaceIndex < 0 || nInterfaceIndex >= (int)nInterfaceNum)
    {
        printf("Invalid interface index.\nQuit.\n");
        break;
    }

    // Enable the frame grabber and get the frame grabber handle.
    nRet = MV_FG_OpenInterface((unsigned int)nInterfaceIndex, &hInterface);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Open No.%d interface failed! %#x\n", nInterfaceIndex, nRet);
        break;
    }

    // Register the exception information callback function of the frame grabber.
    //nRet = MV_FG_RegisterExceptionCallBack(hInterface, ExceptionCb, hInterface);
    //if (MV_FG_SUCCESS != nRet)
    //{
    //    printf("Register interface exception callback failed!\n");
    //    break;
    //}

    // Enumerate cameras of the frame grabber.
    nRet = MV_FG_UpdateDeviceList(hInterface, &bChanged);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Update device list failed! %#x\n", nRet);
        break;
    }

    // Get the number of devices.
    unsigned int nDeviceNum = 0;
    nRet = MV_FG_GetNumDevices(hInterface, &nDeviceNum);
    if (MV_FG_SUCCESS != nRet || 0 == nDeviceNum)
    {
        printf("No device found! return = %d, number = %d\n", nRet, nDeviceNum);
        break;
```

```
    }

    // Display device information.
    if (false == PrintDeviceInfo(hInterface, nDeviceNum))
    {
        break;
    }

    // Select device.
    int nDeviceIndex = -1;
    printf("Select a device: ");
    scanf_s("%d", &nDeviceIndex);
    ClearStdin();

    if (nDeviceIndex < 0 || nDeviceIndex >= (int)nDeviceNum)
    {
        printf("Invalid device index.\nQuit.\n");
        break;
    }

    // Open the device and get the device handle.
    nRet = MV_FG_OpenDevice(hInterface, (unsigned int)nDeviceIndex, &hDevice);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Open No.%d device failed! %#x\n", nDeviceIndex, nRet);
        hDevice = NULL;
        break;
    }

    // Register the exception information callback function of the device.
    //nRet = MV_FG_RegisterExceptionCallBack(hDevice, ExceptionCb, hDevice);
    //if (MV_FG_SUCCESS != nRet)
    //{
    //    printf("Register device exception callback failed!\n");
    //    break;
    //}

    // Disable trigger mode.
    nRet = MV_FG_SetEnumValueByString(hDevice, "TriggerMode", "Off");
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Turn off trigger mode failed! %#x\n", nRet);
        break;
    }

    // Get the number of stream channels.
    unsigned int nStreamNum = 0;
    nRet = MV_FG_GetNumStreams(hDevice, &nStreamNum);
    if (MV_FG_SUCCESS != nRet || 0 == nStreamNum)
    {
        printf("No stream available! return = %d, number = %d\n", nRet, nStreamNum);
```

```
        break;
    }

    // Enable stream channel (currently only one stream channel is supported at a time).
    nRet = MV_FG_OpenStream(hDevice, 0, &hStream);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Open stream failed! %#x\n", nRet);
        break;
    }

    // Register the exception information callback function of the stream channel.
    //nRet = MV_FG_RegisterExceptionCallBack(hStream, ExceptionCb, hStream);
    //if (MV_FG_SUCCESS != nRet)
    //{
    //    printf("Register stream exception callback failed!\n");
    //    break;
    //}

    // Set the number of internal buffers for the SDK.
    nRet = MV_FG_SetBufferNum(hStream, BUFFER_NUMBER);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Set buffer number failed! %#x\n", nRet);
        break;
    }

    Callback_User stUerParam;
    memset(&stUerParam, 0, sizeof(Callback_User));
    stUerParam.hDevice = hDevice;

    // Register the frame buffer information callback function.
    nRet = MV_FG_RegisterFrameCallBack(hStream, FrameCb, &stUerParam);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Register frame callback failed! %#x\n", nRet);
        break;
    }

    // Start image acquisition.
    nRet = MV_FG_StartAcquisition(hStream);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Start acquistion failed! %#x\n", nRet);
        return nRet;
    }

    printf("Press any key to stop acquisition.\n");
    WaitForKeyPress();

    // Stop image acquisition.
```

```
        nRet = MV_FG_StopAcquisition(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Stop acquisition failed! %#x\n", nRet);
            if (stUerParam.stSaveJpegInfo.pJpgBuf)
            {
                free(stUerParam.stSaveJpegInfo.pJpgBuf);
                stUerParam.stSaveJpegInfo.pJpgBuf = NULL;
            }
            return nRet;
        }
        if (stUerParam.stSaveJpegInfo.pJpgBuf)
        {
            free(stUerParam.stSaveJpegInfo.pJpgBuf);
            stUerParam.stSaveJpegInfo.pJpgBuf = NULL;
        }
    } while (0);

    // Disable stream channel.
    if (NULL != hStream)
    {
        nRet = MV_FG_CloseStream(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close stream failed! %#x\n", nRet);
        }
        hStream = NULL;
    }

    // Close the device.
    if (NULL != hDevice)
    {
        nRet = MV_FG_CloseDevice(hDevice);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close device failed! %#x\n", nRet);
        }
        hDevice = NULL;
    }

    // Close the frame grabber.
    if (NULL != hInterface)
    {
        nRet = MV_FG_CloseInterface(hInterface);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close interface failed! %#x\n", nRet);
        }
        hInterface = NULL;
    }
```

```
    printf("Press any key to exit.\n");
    WaitForKeyPress();

    return 0;
}
```

A.2 Acquire Images with Internal Buffers

The following sample codes show how to acquire images with internal buffers of the SDK.

```
#include <stdio.h>
#include <Windows.h>
#include <process.h>
#include <conio.h>
#include "MVFGControl.h"

#define BUFFER_NUMBER          3          // Number of requested buffers
#define FILE_NAME_LEN          256        // The maximum length of file name
#define SAVE_IMAGE_NUM         10         // The maximum number of saved images
#define TIMEOUT                1000       // Timeout; unit: millisecond (ms)
bool g_bExit = false;                     // Stop acquisition

// Wait for key press.
void WaitForKeyPress(void)
{
    while(!_kbhit())
    {
        Sleep(10);
    }
    _getch();
}

// Clear residual data from stdin
void ClearStdin(void)
{
    char c = '\0';

    while (1)
    {
        c = getchar();
        if ('\n' == c || EOF == c)
        {
            break;
        }
    }
}

// Exception information callback function.
void ExceptionCb(MV_FG_EXCEPTION_TYPE enExceptionType, void* pUser)
{
    switch(enExceptionType)
```

```
    {
    case EXCEPTION_TYPE_INTERFACE_DISCONNECT:
        {
            printf("Exception: Interface Disconnected!\n");
            break;
        }
    case EXCEPTION_TYPE_DEVICE_DISCONNECT:
        {
            printf("Exception: Device Disconnected!\n");
            break;
        }
    case EXCEPTION_TYPE_STREAM_ABNORMAL_IMAGE:
        {
            printf("Exception: Abnormal Image!\n");
            break;
        }
    case EXCEPTION_TYPE_STREAM_LIST_OVERFLOW:
        {
            printf("Exception: Buffer List Overflow, Clear the Oldest Frame!\n");
            break;
        }
    case EXCEPTION_TYPE_STREAM_DISCONNECTED:
        {
            printf("Exception: Stream Disconnected!\n");
            break;
        }
    default:
        {
            printf("Unknown Exception!\n");
            break;
        }
    }
}



// Save the original BMP image data.
void SaveBitImage(unsigned char* pBitMapBuf, unsigned int nBufferSize)
{
    if (NULL != pBitMapBuf && 0    < nBufferSize)
    {
        char szFileName[FILE_NAME_LEN] = { 0 };
        FILE* pImageFile = NULL;
        SYSTEMTIME sys;
        GetLocalTime(&sys);

        sprintf_s(szFileName, FILE_NAME_LEN, "Image_%04d%02d%02d%02d%02d%02d%04d.bmp",
sys.wYear, sys.wMonth,
            sys.wDay, sys.wHour, sys.wMinute, sys.wSecond, sys.wMilliseconds);

        if ((0 != fopen_s(&pImageFile, szFileName, "wb")) || (NULL == pImageFile))
```

```
        {
            return;
        }

        fwrite(pBitMapBuf, 1, nBufferSize, pImageFile);
        fclose(pImageFile);
    }
}

// Image acquisition thread.
unsigned int __stdcall GrabbingThread(void* pUser)
{
    if (pUser)
    {
        STREAMHANDLE            hStream = (STREAMHANDLE)pUser;
        MV_FG_BUFFER_INFO    stFrameInfo = { 0 };    // Image information
        int                         nRet = 0;
        MV_FG_SAVE_BITMAP_INFO stSaveBitmapInfo = {0}; // BMP image saving information
        memset(&stSaveBitmapInfo, 0, sizeof(MV_FG_SAVE_BITMAP_INFO));

        // Start image acquisition.
        nRet = MV_FG_StartAcquisition(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Start acquistion failed! %#x\n", nRet);
            return nRet;
        }
        g_bExit = false;

        while (!g_bExit)
        {
            // Get the buffer information of a frame.
            nRet = MV_FG_GetFrameBuffer(hStream, &stFrameInfo, TIMEOUT);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get frame buffer info failed! %#x\n", nRet);
                continue;
            }
            else
            {
                printf("FrameNumber:%2I64d%, Width:%d, Height:%d\n", stFrameInfo.nFrameID,
stFrameInfo.nWidth, stFrameInfo.nHeight);
                if ((stFrameInfo.pBuffer) && (0 < stFrameInfo.nFilledSize))
                {
                    stSaveBitmapInfo.stInputImageInfo.pImageBuf = (unsigned
char*)stFrameInfo.pBuffer;
                    stSaveBitmapInfo.stInputImageInfo.nImageBufLen = stFrameInfo.nFilledSize;
                    stSaveBitmapInfo.stInputImageInfo.nHeight = stFrameInfo.nHeight;
                    stSaveBitmapInfo.stInputImageInfo.nWidth = stFrameInfo.nWidth;
                    stSaveBitmapInfo.stInputImageInfo.enPixelType = stFrameInfo.enPixelType;
                    unsigned int nSize = stFrameInfo.nHeight * stFrameInfo.nWidth * 4;
```

```
                        if (stSaveBitmapInfo.nBmpBufSize < nSize)
                        {
                            if (stSaveBitmapInfo.pBmpBuf)
                            {
                                free (stSaveBitmapInfo.pBmpBuf);
                                stSaveBitmapInfo.pBmpBuf = NULL;
                            }
                            stSaveBitmapInfo.pBmpBuf = (unsigned char*)malloc(nSize);
                            if (NULL == stSaveBitmapInfo.pBmpBuf)
                            {
                                printf("malloc pConvertData fail!\n");
                                nRet = MV_FG_ERR_RESOURCE_EXHAUSTED;
                                break;
                            }
                            stSaveBitmapInfo.nBmpBufSize = nSize;
                        }

                        stSaveBitmapInfo.nBmpBufLen = 0;
                        stSaveBitmapInfo.enCfaMethod = MV_FG_CFA_METHOD_OPTIMAL;
                        nRet = MV_FG_SaveBitmap(hStream, &stSaveBitmapInfo);
                        if (MV_FG_SUCCESS != nRet)
                        {
                            printf("MV_FG_SaveBitmap info failed! %#x\n", nRet);
                        }
                        else
                        {
                             SaveBitImage(stSaveBitmapInfo.pBmpBuf,
stSaveBitmapInfo.nBmpBufLen);
                        }
                    }
                }

                // Insert the buffer back to the input queue.
                nRet = MV_FG_ReleaseFrameBuffer(hStream, &stFrameInfo);
                if (MV_FG_SUCCESS != nRet)
                {
                    printf("Release frame buffer failed! %#x\n", nRet);
                    break;
                }
            }

            if (stSaveBitmapInfo.pBmpBuf)
            {
                free(stSaveBitmapInfo.pBmpBuf);
                stSaveBitmapInfo.pBmpBuf = NULL;
            }

            // Stop image acquisition.
            nRet = MV_FG_StopAcquisition(hStream);
            if (MV_FG_SUCCESS != nRet)
            {
```

```
            printf("Stop acquisition failed! %#x\n", nRet);
            return nRet;
        }

    }

    return MV_FG_SUCCESS;
}

// Print frame grabber information.
bool PrintInterfaceInfo(unsigned int nInterfaceNum)
{
    int nRet = 0;

    for (unsigned int i = 0; i < nInterfaceNum; i++)
    {
        MV_FG_INTERFACE_INFO stInterfaceInfo = { 0 };

        nRet = MV_FG_GetInterfaceInfo(i, &stInterfaceInfo);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Get info of No.%d interface failed! %#x\n", i, nRet);
            return false;
        }

        switch (stInterfaceInfo.nTLayerType)
        {
        case MV_FG_CXP_INTERFACE:
            {
                printf("[CXP]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chSerialNumber);
                break;
            }
        case MV_FG_GEV_INTERFACE:
            {
                printf("[GEV]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chSerialNumber);
                break;
            }
        case MV_FG_CAMERALINK_INTERFACE:
            {
                printf("[CML]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chInterfaceID,
```

```
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chSerialNumber);
                break;
            }
        default:
            {
                printf("Unknown interface type.\n");
                return false;
            }
        }
    }

    return true;
}

// Print device information.
bool PrintDeviceInfo(IFHANDLE hInterface, unsigned int nDeviceNum)
{
    int nRet = 0;

    for (unsigned int i = 0; i < nDeviceNum; i++)
    {
        MV_FG_DEVICE_INFO stDeviceInfo = { 0 };

        nRet = MV_FG_GetDeviceInfo(hInterface, i, &stDeviceInfo);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Get info of No.%d device failed! %#x\n", i, nRet);
            return false;
        }

        switch (stDeviceInfo.nDevType)
        {
        case MV_FG_CXP_DEVICE:
            {
                printf("[CXP]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                    stDeviceInfo.DevInfo.stCXPDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stCXPDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stCXPDevInfo.chSerialNumber);
                break;
            }
        case MV_FG_GEV_DEVICE:
            {
                printf("[GEV]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                    stDeviceInfo.DevInfo.stGEVDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stGEVDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stGEVDevInfo.chSerialNumber);
                break;
            }
        case MV_FG_CAMERALINK_DEVICE:
```

```
                {
                    printf("[CML]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                        stDeviceInfo.DevInfo.stCMLDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stCMLDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stCMLDevInfo.chSerialNumber);
                    break;
                }
            default:
                {
                    printf("Unknown device type.\n");
                    return false;
                }
            }
        }

        return true;
}

int main()
{
        int                 nRet = 0;
        IFHANDLE            hInterface = NULL;
        DEVHANDLE           hDevice = NULL;
        STREAMHANDLE        hStream = NULL;

        do
        {
            // Enumerate frame grabbers.
            bool bChanged = false;
            nRet = MV_FG_UpdateInterfaceList(MV_FG_CXP_INTERFACE | MV_FG_GEV_INTERFACE |
MV_FG_CAMERALINK_INTERFACE, &bChanged);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Update interface list failed! %#x\n", nRet);
                break;
            }

            // Get the number of frame grabbers.
            unsigned int nInterfaceNum = 0;
            nRet = MV_FG_GetNumInterfaces(&nInterfaceNum);
            if (MV_FG_SUCCESS != nRet || 0 == nInterfaceNum)
            {
                printf("No interface found! return = %d, number = %d\n", nRet, nInterfaceNum);
                break;
            }

            // Display frame grabber information.
            if (false == PrintInterfaceInfo(nInterfaceNum))
            {
                break;
```

```
    }

    // Select frame grabber.
    int nInterfaceIndex = -1;
    printf("Select an interface: ");
    scanf_s("%d", &nInterfaceIndex);
    ClearStdin();

    if (nInterfaceIndex < 0 || nInterfaceIndex >= (int)nInterfaceNum)
    {
        printf("Invalid interface index.\nQuit.\n");
        break;
    }

    // Open the frame grabber and get the frame grabber handle.
    nRet = MV_FG_OpenInterface((unsigned int)nInterfaceIndex, &hInterface);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Open No.%d interface failed! %#x\n", nInterfaceIndex, nRet);
        break;
    }

    // Register the exception information callback function of the frame grabber.
    //nRet = MV_FG_RegisterExceptionCallBack(hInterface, ExceptionCb, hInterface);
    //if (MV_FG_SUCCESS != nRet)
    //{
    //    printf("Register interface exception callback failed!\n");
    //    break;
    //}

    // Enumerate cameras of the frame grabber.
    nRet = MV_FG_UpdateDeviceList(hInterface, &bChanged);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Update device list failed! %#x\n", nRet);
        break;
    }

    // Get the number of devices.
    unsigned int nDeviceNum = 0;
    nRet = MV_FG_GetNumDevices(hInterface, &nDeviceNum);
    if (MV_FG_SUCCESS != nRet || 0 == nDeviceNum)
    {
        printf("No device found! return = %d, number = %d\n", nRet, nDeviceNum);
        break;
    }

    // Display device information.
    if (false == PrintDeviceInfo(hInterface, nDeviceNum))
    {
        break;
```

```
}

// Select device.
int nDeviceIndex = -1;
printf("Select a device: ");
scanf_s("%d", &nDeviceIndex);
ClearStdin();

if (nDeviceIndex < 0 || nDeviceIndex >= (int)nDeviceNum)
{
    printf("Invalid device index.\nQuit.\n");
    break;
}

// Open the device and get the device handle.
nRet = MV_FG_OpenDevice(hInterface, (unsigned int)nDeviceIndex, &hDevice);
if (MV_FG_SUCCESS != nRet)
{
    printf("Open No.%d device failed! %#x\n", nDeviceIndex, nRet);
    hDevice = NULL;
    break;
}

// Register the exception information callback function of the device.
//nRet = MV_FG_RegisterExceptionCallBack(hDevice, ExceptionCb, hDevice);
//if (MV_FG_SUCCESS != nRet)
//{
//    printf("Register device exception callback failed!\n");
//    break;
//}

// Disable the trigger mode.
nRet = MV_FG_SetEnumValueByString(hDevice, "TriggerMode", "Off");
if (MV_FG_SUCCESS != nRet)
{
    printf("Turn off trigger mode failed! %#x\n", nRet);
    break;
}

// Get the number of stream channels.
unsigned int nStreamNum = 0;
nRet = MV_FG_GetNumStreams(hDevice, &nStreamNum);
if (MV_FG_SUCCESS != nRet || 0 == nStreamNum)
{
    printf("No stream available! return = %d, number = %d\n", nRet, nStreamNum);
    break;
}

// Open stream channel (currently only one stream channel is supported at a time).
nRet = MV_FG_OpenStream(hDevice, 0, &hStream);
if (MV_FG_SUCCESS != nRet)
```

```
        {
            printf("Open stream failed! %#x\n", nRet);
            break;
        }

        // Register the exception information callback function of the stream channel.
        //nRet = MV_FG_RegisterExceptionCallBack(hStream, ExceptionCb, hStream);
        //if (MV_FG_SUCCESS != nRet)
        //{
        //    printf("Register stream exception callback failed!\n");
        //    break;
        //}

        // Set the number of internal buffers for the SDK.
        nRet = MV_FG_SetBufferNum(hStream, BUFFER_NUMBER);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Set buffer number failed! %#x\n", nRet);
            break;
        }

        // Create thread for image acquisition.
        void* hThreadHandle = (void*)_beginthreadex(NULL, 0, GrabbingThread, hStream, 0, NULL);
        if (NULL == hThreadHandle)
        {
            printf("Create thread failed!\n");
            break;
        }

        printf("Press any key to stop acquisition.\n");
        WaitForKeyPress();

        // Stop image acquisition thread.
        g_bExit = true;
        WaitForSingleObject(hThreadHandle, INFINITE);
        CloseHandle(hThreadHandle);
        hThreadHandle = NULL;
    } while (0);

    // Close stream channel.
    if (NULL != hStream)
    {
        nRet = MV_FG_CloseStream(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close stream failed! %#x\n", nRet);
        }
        hStream = NULL;
    }

    // Close the device.
```

```
    if (NULL != hDevice)
    {
        nRet = MV_FG_CloseDevice(hDevice);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close device failed! %#x\n", nRet);
        }
        hDevice = NULL;
    }

    // Close the frame grabber.
    if (NULL != hInterface)
    {
        nRet = MV_FG_CloseInterface(hInterface);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close interface failed! %#x\n", nRet);
        }
        hInterface = NULL;
    }

    printf("Press any key to exit.\n");
    WaitForKeyPress();

    return 0;
}
```

A.3 Acquire Images with User Registering Buffers

The following sample codes show how to acquire images using buffers registered to stream channels by the user.

```
#include <stdio.h>
#include <Windows.h>
#include <process.h>
#include <conio.h>
#include "MVFGControl.h"

#define BUFFER_NUMBER        3        // Number of requested buffers
#define FILE_NAME_LEN        256      // The maximum length of file name
#define SAVE_IMAGE_NUM       10       // The maximum number of saved images
#define TIMEOUT              1000     // Timeout; unit: millisecond (ms)

bool g_bExit = false;                 // Stop acquisition

// Wait for key press.
void WaitForKeyPress(void)
{
    while(!_kbhit())
    {
        Sleep(10);
```

```
    }
    _getch();
}

// Clear residual data from stdin.
void ClearStdin(void)
{
    char c = '\0';

    while (1)
    {
        c = getchar();
        if ('\n' == c || EOF == c)
        {
            break;
        }
    }
}

// Save the original image data.
void SaveRawImage(int nImageNo, MV_FG_BUFFER_INFO* pstImageInfo)
{
    if (pstImageInfo)
    {
        char szFileName[FILE_NAME_LEN] = { 0 };

        sprintf_s(szFileName, FILE_NAME_LEN, "Image_w%d_h%d_n%d.raw", pstImageInfo->nWidth,
pstImageInfo->nHeight, nImageNo);

        FILE* pImageFile = NULL;
        if ((0 != fopen_s(&pImageFile, szFileName, "wb")) || (NULL == pImageFile))
        {
            return;
        }

        fwrite(pstImageInfo->pBuffer, 1, pstImageInfo->nFilledSize, pImageFile);
        fclose(pImageFile);
    }
}

// Image acquisition thread.
unsigned int __stdcall GrabbingThread(void* pUser)
{
    if (pUser)
    {
        STREAMHANDLE            hStream = (STREAMHANDLE)pUser;
        BUFFERHANDLE            hBuffer = NULL;
        MV_FG_BUFFER_INFO       stFrameInfo = { 0 };       // Image information
        int                     nSaveImage = 0;            // Number of saved images
        int                     nRet = 0;
```

```
        // Start image acquisition.
        nRet = MV_FG_StartAcquisition(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Start acquistion failed! %#x\n", nRet);
            return nRet;
        }
        g_bExit = false;

        while (!g_bExit)
        {
            // Get the buffer handle of a frame.
            nRet = MV_FG_GetImageBuffer(hStream, &hBuffer, TIMEOUT);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get image buffer failed! %#x\n", nRet);
                continue;
            }

            // Get image information.
            nRet = MV_FG_GetBufferInfo(hBuffer, &stFrameInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get image info failed! %#x\n", nRet);
            }
            else
            {
                printf("FrameNumber:%2I64d%, Width:%d, Height:%d\n", stFrameInfo.nFrameID,
stFrameInfo.nWidth, stFrameInfo.nHeight);

                if (nSaveImage < SAVE_IMAGE_NUM)
                {
                    SaveRawImage(++nSaveImage, &stFrameInfo);
                }
            }

            // Insert the buffer back to the input queue.
            nRet = MV_FG_QueueBuffer(hBuffer);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Queue buffer failed! %#x\n", nRet);
                break;
            }
        }

        // Stop image acquisition.
        nRet = MV_FG_StopAcquisition(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Stop acquisition failed! %#x\n", nRet);
            return nRet;
```

```
            }
        }

        return MV_FG_SUCCESS;
}

// Print frame grabber information.
bool PrintInterfaceInfo(unsigned int nInterfaceNum)
{
        int nRet = 0;

        for (unsigned int i = 0; i < nInterfaceNum; i++)
        {
            MV_FG_INTERFACE_INFO stInterfaceInfo = { 0 };

            nRet = MV_FG_GetInterfaceInfo(i, &stInterfaceInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get info of No.%d interface failed! %#x\n", i, nRet);
                return false;
            }

            switch (stInterfaceInfo.nTLayerType)
            {
            case MV_FG_CXP_INTERFACE:
                {
                    printf("[CXP]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                        stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chDisplayName,
                        stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chInterfaceID,
                        stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chSerialNumber);
                    break;
                }
            case MV_FG_GEV_INTERFACE:
                {
                    printf("[GEV]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                        stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chDisplayName,
                        stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chInterfaceID,
                        stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chSerialNumber);
                    break;
                }
            case MV_FG_CAMERALINK_INTERFACE:
                {
                    printf("[CML]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                        stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chDisplayName,
                        stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chInterfaceID,
                        stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chSerialNumber);
                    break;
                }
```

```
            default:
                {
                    printf("Unknown interface type.\n");
                    return false;
                }
            }
        }

        return true;
}

// Print device information.
bool PrintDeviceInfo(IFHANDLE hInterface, unsigned int nDeviceNum)
{
        int nRet = 0;

        for (unsigned int i = 0; i < nDeviceNum; i++)
        {
            MV_FG_DEVICE_INFO stDeviceInfo = { 0 };

            nRet = MV_FG_GetDeviceInfo(hInterface, i, &stDeviceInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get info of No.%d device failed! %#x\n", i, nRet);
                return false;
            }

            switch (stDeviceInfo.nDevType)
            {
            case MV_FG_CXP_DEVICE:
                {
                    printf("[CXP]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chSerialNumber);
                    break;
                }
            case MV_FG_GEV_DEVICE:
                {
                    printf("[GEV]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chSerialNumber);
                    break;
                }
            case MV_FG_CAMERALINK_DEVICE:
                {
                    printf("[CML]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
```

```
                        stDeviceInfo.DevInfo.stCMLDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stCMLDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stCMLDevInfo.chSerialNumber);
                    break;
                }
            default:
                {
                    printf("Unknown device type.\n");
                    return false;
                }
            }
        }

        return true;
}

int main()
{
        int                 nRet = 0;
        IFHANDLE            hInterface = NULL;
        DEVHANDLE           hDevice = NULL;
        STREAMHANDLE        hStream = NULL;
        BUFFERHANDLE        hBuffer[BUFFER_NUMBER] = { 0 };
        void*               pBuffer[BUFFER_NUMBER] = { 0 };

        do
        {
            // Enumerate frame grabbers.
            bool bChanged = false;
            nRet = MV_FG_UpdateInterfaceList(MV_FG_CXP_INTERFACE | MV_FG_GEV_INTERFACE |
MV_FG_CAMERALINK_INTERFACE, &bChanged);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Update interface list failed! %#x\n", nRet);
                break;
            }

            // Get the number of frame grabbers.
            unsigned int nInterfaceNum = 0;
            nRet = MV_FG_GetNumInterfaces(&nInterfaceNum);
            if (MV_FG_SUCCESS != nRet || 0 == nInterfaceNum)
            {
                printf("No interface found! return = %d, number = %d\n", nRet, nInterfaceNum);
                break;
            }

            // Display frame grabber information.
            if (false == PrintInterfaceInfo(nInterfaceNum))
            {
                break;
            }
```

```
// Select frame grabber.
int nInterfaceIndex = -1;
printf("Select an interface: ");
scanf_s("%d", &nInterfaceIndex);
ClearStdin();

if (nInterfaceIndex < 0 || nInterfaceIndex >= (int)nInterfaceNum)
{
    printf("Invalid interface index.\nQuit.\n");
    break;
}

// Open the frame grabber and get the frame grabber handle.
nRet = MV_FG_OpenInterface((unsigned int)nInterfaceIndex, &hInterface);
if (MV_FG_SUCCESS != nRet)
{
    printf("Open No.%d interface failed! %#x\n", nInterfaceIndex, nRet);
    break;
}

// Enumerate cameras of the frame grabber.
nRet = MV_FG_UpdateDeviceList(hInterface, &bChanged);
if (MV_FG_SUCCESS != nRet)
{
    printf("Update device list failed! %#x\n", nRet);
    break;
}

// Get the number of devices.
unsigned int nDeviceNum = 0;
nRet = MV_FG_GetNumDevices(hInterface, &nDeviceNum);
if (MV_FG_SUCCESS != nRet || 0 == nDeviceNum)
{
    printf("No device found! return = %d, number = %d\n", nRet, nDeviceNum);
    break;
}

// Display device information.
if (false == PrintDeviceInfo(hInterface, nDeviceNum))
{
    break;
}

// Select the device.
int nDeviceIndex = -1;
printf("Select a device: ");
scanf_s("%d", &nDeviceIndex);
ClearStdin();

if (nDeviceIndex < 0 || nDeviceIndex >= (int)nDeviceNum)
```

```
{
    printf("Invalid device index.\nQuit.\n");
    break;
}

// Open the device and get the device handle.
nRet = MV_FG_OpenDevice(hInterface, (unsigned int)nDeviceIndex, &hDevice);
if (MV_FG_SUCCESS != nRet)
{
    printf("Open No.%d device failed! %#x\n", nDeviceIndex, nRet);
    hDevice = NULL;
    break;
}

// Disable trigger mode.
nRet = MV_FG_SetEnumValueByString(hDevice, "TriggerMode", "Off");
if (MV_FG_SUCCESS != nRet)
{
    printf("Turn off trigger mode failed! %#x\n", nRet);
    break;
}

// Get the number of stream channels.
unsigned int nStreamNum = 0;
nRet = MV_FG_GetNumStreams(hDevice, &nStreamNum);
if (MV_FG_SUCCESS != nRet || 0 == nStreamNum)
{
    printf("No stream available! return = %d, number = %d\n", nRet, nStreamNum);
    break;
}

// O stream channel (currently only one stream channel is supported at a time).
nRet = MV_FG_OpenStream(hDevice, 0, &hStream);
if (MV_FG_SUCCESS != nRet)
{
    printf("Open stream failed! %#x\n", nRet);
    break;
}

// Get the image size of the stream channel.
unsigned int nPayloadSize = 0;
nRet = MV_FG_GetPayloadSize(hStream, &nPayloadSize);
if (MV_FG_SUCCESS != nRet)
{
    printf("Get payload size failed! %#x\n", nRet);
    break;
}

// Register buffer to stream channel.
for (unsigned int i = 0; i < BUFFER_NUMBER; i++)
{
```

```
            // Allocate image buffers.
            pBuffer[i] = malloc(nPayloadSize);
            if (NULL == pBuffer[i])
            {
                printf("Allocate buffer failed!\n");
                nRet = MV_FG_ERR_OUT_OF_MEMORY;
                break;
            }

            // Register buffer to SDK.
            nRet = MV_FG_AnnounceBuffer(hStream, pBuffer[i], nPayloadSize, NULL, &(hBuffer[i]));
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Announce buffer failed! %#x\n", nRet);
                break;
            }
        }
        if (MV_FG_SUCCESS != nRet)
        {
            break;
        }

        // Refresh the buffer queue for image acquisition.
        nRet = MV_FG_FlushQueue(hStream, MV_FG_BUFFER_QUEUE_ALL_TO_INPUT);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Flush queue: all to input failed! %#x\n", nRet);
            break;
        }

        // Create thread for image acquisition.
        void* hThreadHandle = (void*)_beginthreadex(NULL, 0, GrabbingThread, hStream, 0, NULL);
        if (NULL == hThreadHandle)
        {
            printf("Create thread failed!\n");
            break;
        }

        printf("Press any key to stop acquisition.\n");
        WaitForKeyPress();

        // Stop image acquisition thread.
        g_bExit = true;
        WaitForSingleObject(hThreadHandle, INFINITE);
        CloseHandle(hThreadHandle);
        hThreadHandle = NULL;
    } while (0);

    // Release resources.
    if (NULL != hStream)
    {
```

```
        // Clear buffer queues.
        nRet = MV_FG_FlushQueue(hStream, MV_FG_BUFFER_QUEUE_ALL_DISCARD);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Flush buffer queue failed! %#x\n", nRet);
        }

        // Revoke and release registered buffers.
        for (unsigned int i = 0; i < BUFFER_NUMBER; i++)
        {
            if (NULL != hBuffer[i])
            {
                nRet = MV_FG_RevokeBuffer(hStream, hBuffer[i], NULL, NULL);
                if (MV_FG_SUCCESS != nRet)
                {
                    printf("Revoke No.%d buffer failed! %#x\n", i, nRet);
                }
                hBuffer[i] = NULL;
            }

            if (NULL != pBuffer[i])
            {
                free(pBuffer[i]);
                pBuffer[i] = NULL;
            }
        }

        // Close the stream channel.
        nRet = MV_FG_CloseStream(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close stream failed! %#x\n", nRet);
        }
        hStream = NULL;
    }

    // Close the device.
    if (NULL != hDevice)
    {
        nRet = MV_FG_CloseDevice(hDevice);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close device failed! %#x\n", nRet);
        }
        hDevice = NULL;
    }

    // Close the frame grabber.
    if (NULL != hInterface)
    {
        nRet = MV_FG_CloseInterface(hInterface);
```

```
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close interface failed! %#x\n", nRet);
        }
        hInterface = NULL;
    }

    printf("Press any key to exit.\n");
    WaitForKeyPress();

    return 0;
}
```

A.4 Convert Pixel Format

The following sample codes show how to convert the format of acquired images to a desired format.

```
#include <stdio.h>
#include <Windows.h>
#include <process.h>
#include <conio.h>
#include "MVFGControl.h"

#define BUFFER_NUMBER          3          // Number of requested buffers
#define FILE_NAME_LEN          256        // The maximum length of file name
#define SAVE_IMAGE_NUM         10         // The maximum number of saved images
#define TIMEOUT                1000       // Timeout; unit: millisecond (ms)
bool g_bExit = false;                     // Stop acquisition

// Wait for key press.
void WaitForKeyPress(void)
{
    while(!_kbhit())
    {
        Sleep(10);
    }
    _getch();
}

// Clear residual data from stdin.
void ClearStdin(void)
{
    char c = '\0';

    while (1)
    {
        c = getchar();
        if ('\n' == c || EOF == c)
        {
            break;
        }
```

```
    }
}

// Exception information callback function.
void ExceptionCb(MV_FG_EXCEPTION_TYPE enExceptionType, void* pUser)
{
    switch(enExceptionType)
    {
    case EXCEPTION_TYPE_INTERFACE_DISCONNECT:
        {
            printf("Exception: Interface Disconnected!\n");
            break;
        }
    case EXCEPTION_TYPE_DEVICE_DISCONNECT:
        {
            printf("Exception: Device Disconnected!\n");
            break;
        }
    case EXCEPTION_TYPE_STREAM_ABNORMAL_IMAGE:
        {
            printf("Exception: Abnormal Image!\n");
            break;
        }
    case EXCEPTION_TYPE_STREAM_LIST_OVERFLOW:
        {
            printf("Exception: Buffer List Overflow, Clear the Oldest Frame!\n");
            break;
        }
    case EXCEPTION_TYPE_STREAM_DISCONNECTED:
        {
            printf("Exception: Stream Disconnected!\n");
            break;
        }
    default:
        {
            printf("Unknown Exception!\n");
            break;
        }
    }
}

// Save the original BMP image data.
void SaveBitImage(unsigned char* pBitMapBuf, unsigned int nBufferSize)
{
    if (NULL != pBitMapBuf && 0    < nBufferSize)
    {
        char szFileName[FILE_NAME_LEN] = { 0 };
        FILE* pImageFile = NULL;
        SYSTEMTIME sys;
        GetLocalTime(&sys);
```

```
            sprintf_s(szFileName, FILE_NAME_LEN, "Image_%04d%02d%02d%02d%02d%02d%04d.bmp",
sys.wYear, sys.wMonth,
                  sys.wDay, sys.wHour, sys.wMinute, sys.wSecond, sys.wMilliseconds);

            if ((0 != fopen_s(&pImageFile, szFileName, "wb")) || (NULL == pImageFile))
            {
                  return;
            }

            fwrite(pBitMapBuf, 1, nBufferSize, pImageFile);
            fclose(pImageFile);
      }
}

bool IsColorPixelFormat(MV_FG_PIXEL_TYPE enPixelType)
{
      switch(enPixelType)
      {
      case MV_FG_PIXEL_TYPE_RGBA8_Packed:
      case MV_FG_PIXEL_TYPE_BGRA8_Packed:
      case MV_FG_PIXEL_TYPE_BayerGR8:
      case MV_FG_PIXEL_TYPE_BayerRG8:
      case MV_FG_PIXEL_TYPE_BayerGB8:
      case MV_FG_PIXEL_TYPE_BayerBG8:
      case MV_FG_PIXEL_TYPE_BayerGB10:
      case MV_FG_PIXEL_TYPE_BayerGB10_Packed:
      case MV_FG_PIXEL_TYPE_BayerBG10:
      case MV_FG_PIXEL_TYPE_BayerBG10_Packed:
      case MV_FG_PIXEL_TYPE_BayerRG10:
      case MV_FG_PIXEL_TYPE_BayerRG10_Packed:
      case MV_FG_PIXEL_TYPE_BayerGR10:
      case MV_FG_PIXEL_TYPE_BayerGR10_Packed:
      case MV_FG_PIXEL_TYPE_BayerGB12:
      case MV_FG_PIXEL_TYPE_BayerGB12_Packed:
      case MV_FG_PIXEL_TYPE_BayerBG12:
      case MV_FG_PIXEL_TYPE_BayerBG12_Packed:
      case MV_FG_PIXEL_TYPE_BayerRG12:
      case MV_FG_PIXEL_TYPE_BayerRG12_Packed:
      case MV_FG_PIXEL_TYPE_BayerGR12:
      case MV_FG_PIXEL_TYPE_BayerGR12_Packed:
      case MV_FG_PIXEL_TYPE_BayerGR16:
      case MV_FG_PIXEL_TYPE_BayerRG16:
      case MV_FG_PIXEL_TYPE_BayerGB16:
      case MV_FG_PIXEL_TYPE_BayerBG16:
      case MV_FG_PIXEL_TYPE_YUV422_Packed:
      case MV_FG_PIXEL_TYPE_YUV422_YUYV_Packed:
            return true;
      default:
            return false;
      }
}
```

```
// Determine whether the image is in monochrome format.
bool IsMonoPixelFormat(MV_FG_PIXEL_TYPE enPixelType)
{
	switch(enPixelType)
	{
	case MV_FG_PIXEL_TYPE_Mono10:
	case MV_FG_PIXEL_TYPE_Mono10_Packed:
	case MV_FG_PIXEL_TYPE_Mono12:
	case MV_FG_PIXEL_TYPE_Mono12_Packed:
	case MV_FG_PIXEL_TYPE_Mono16:
		return true;
	default:
		return false;
	}
}

// Image acquisition thread.
unsigned int __stdcall GrabbingThread(void* pUser)
{
	if (pUser)
	{
		STREAMHANDLE		hStream = (STREAMHANDLE)pUser;
		MV_FG_BUFFER_INFO	stFrameInfo = { 0 };	// Image information
		int			nRet = 0;
		MV_FG_CONVERT_PIXEL_INFO stConvertPixelInfo = {0}; // Image conversion information
		memset(&stConvertPixelInfo, 0, sizeof(MV_FG_CONVERT_PIXEL_INFO));

		// Start image acquisition.
		nRet = MV_FG_StartAcquisition(hStream);
		if (MV_FG_SUCCESS != nRet)
		{
			printf("Start acquistion failed! %#x\n", nRet);
			return nRet;
		}
		g_bExit = false;

		while (!g_bExit)
		{
			// Get the buffer information of a frame.
			nRet = MV_FG_GetFrameBuffer(hStream, &stFrameInfo, TIMEOUT);
			if (MV_FG_SUCCESS != nRet)
			{
				printf("Get frame buffer info failed! %#x\n", nRet);
				continue;
			}
			else
			{
				printf("FrameNumber:%2I64d%, Width:%d, Height:%d\n", stFrameInfo.nFrameID,
stFrameInfo.nWidth, stFrameInfo.nHeight);
				if ((stFrameInfo.pBuffer) && (0 < stFrameInfo.nFilledSize))
```

```
                {
                    MV_FG_PIXEL_TYPE enDstPixelType = MV_FG_PIXEL_TYPE_Undefined;
                    unsigned int nChannelNum = 0;
                    char szFileName[FILE_NAME_LEN] = { 0 };
                    FILE* pImageFile = NULL;
                    SYSTEMTIME sys;
                    GetLocalTime(&sys);

                    // If in color format, convert to RGB8.
                    if (IsColorPixelFormat(stFrameInfo.enPixelType))
                    {
                        nChannelNum = 3;
                        enDstPixelType = MV_FG_PIXEL_TYPE_RGB8_Packed;
                        sprintf_s(szFileName, FILE_NAME_LEN,
"After_%04d%02d%02d%02d%02d%02d%03d.rgb", sys.wYear, sys.wMonth,
                            sys.wDay, sys.wHour, sys.wMinute, sys.wSecond, sys.wMilliseconds);
                    }
                    //If in monochrome format, convert to Mono8.
                    else if (IsMonoPixelFormat(stFrameInfo.enPixelType))
                    {
                        nChannelNum = 1;
                        enDstPixelType = MV_FG_PIXEL_TYPE_Mono8;
                        sprintf_s(szFileName, FILE_NAME_LEN,
"After_%04d%02d%02d%02d%02d%02d%03d.gray", sys.wYear, sys.wMonth,
                            sys.wDay, sys.wHour, sys.wMinute, sys.wSecond, sys.wMilliseconds);
                    }
                    else
                    {
                        printf("Don't need to convert!\n");
                    }

                    if (enDstPixelType != MV_FG_PIXEL_TYPE_Undefined)
                    {

                        stConvertPixelInfo.stInputImageInfo.pImageBuf = (unsigned
char*)stFrameInfo.pBuffer;
                        stConvertPixelInfo.stInputImageInfo.nImageBufLen =
stFrameInfo.nFilledSize;
                        stConvertPixelInfo.stInputImageInfo.nHeight = stFrameInfo.nHeight;
                        stConvertPixelInfo.stInputImageInfo.nWidth = stFrameInfo.nWidth;
                        stConvertPixelInfo.stInputImageInfo.enPixelType =
stFrameInfo.enPixelType;

                        unsigned int nSize = stFrameInfo.nHeight * stFrameInfo.nWidth *
nChannelNum;
                        if (nSize > stConvertPixelInfo.stOutputImageInfo.nImageBufSize)
                        {
                            if (stConvertPixelInfo.stOutputImageInfo.pImageBuf)
                            {
                                free(stConvertPixelInfo.stOutputImageInfo.pImageBuf);
                                stConvertPixelInfo.stOutputImageInfo.pImageBuf = NULL;
```

```
                                    }
                                    stConvertPixelInfo.stOutputImageInfo.pImageBuf = (unsigned
char*)malloc(nSize);
                                    if (NULL == stConvertPixelInfo.stOutputImageInfo.pImageBuf)
                                    {
                                        printf("malloc pConvertData fail!\n");
                                        nRet = MV_FG_ERR_RESOURCE_EXHAUSTED;
                                        break;
                                    }
                                    stConvertPixelInfo.stOutputImageInfo.nImageBufSize = nSize;
                                }

                                stConvertPixelInfo.stOutputImageInfo.nImageBufLen = 0;
                                stConvertPixelInfo.stOutputImageInfo.enPixelType = enDstPixelType;
                                stConvertPixelInfo.enCfaMethod = MV_FG_CFA_METHOD_OPTIMAL;

                                nRet = MV_FG_ConvertPixelType(hStream, &stConvertPixelInfo);
                                if (MV_FG_SUCCESS != nRet)
                                {
                                    printf("Convert Pixel Type fail! nRet [0x%x]\n", nRet);
                                    continue;
                                }

                                if ((0 != fopen_s(&pImageFile, szFileName, "wb")) || (NULL == pImageFile))
                                {
                                    continue;
                                }

                                fwrite(stConvertPixelInfo.stOutputImageInfo.pImageBuf, 1,
stConvertPixelInfo.stOutputImageInfo.nImageBufLen, pImageFile);
                                fclose(pImageFile);
                                printf("Convert pixeltype succeed\n");
                            }
                        }
                    }

                    // Insert the buffer back to the input queue.
                    nRet = MV_FG_ReleaseFrameBuffer(hStream, &stFrameInfo);
                    if (MV_FG_SUCCESS != nRet)
                    {
                        printf("Release frame buffer failed! %#x\n", nRet);
                        break;
                    }
                }

                if (stConvertPixelInfo.stOutputImageInfo.pImageBuf)
                {
                    free(stConvertPixelInfo.stOutputImageInfo.pImageBuf);
                    stConvertPixelInfo.stOutputImageInfo.pImageBuf = NULL;
                }
```

```
        // Stop image acquisition.
        nRet = MV_FG_StopAcquisition(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Stop acquisition failed! %#x\n", nRet);
            return nRet;
        }

    }

    return MV_FG_SUCCESS;
}

// Print frame grabber information.
bool PrintInterfaceInfo(unsigned int nInterfaceNum)
{
    int nRet = 0;

    for (unsigned int i = 0; i < nInterfaceNum; i++)
    {
        MV_FG_INTERFACE_INFO stInterfaceInfo = { 0 };

        nRet = MV_FG_GetInterfaceInfo(i, &stInterfaceInfo);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Get info of No.%d interface failed! %#x\n", i, nRet);
            return false;
        }

        switch (stInterfaceInfo.nTLayerType)
        {
        case MV_FG_CXP_INTERFACE:
            {
                printf("[CXP]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chSerialNumber);
                break;
            }
        case MV_FG_GEV_INTERFACE:
            {
                printf("[GEV]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chSerialNumber);
                break;
            }
        case MV_FG_CAMERALINK_INTERFACE:
            {
```

```
                    printf("[CML]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                        stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chDisplayName,
                        stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chInterfaceID,
                        stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chSerialNumber);
                    break;
                }
            default:
                {
                    printf("Unknown interface type.\n");
                    return false;
                }
            }
        }

        return true;
}

// Print device information.
bool PrintDeviceInfo(IFHANDLE hInterface, unsigned int nDeviceNum)
{
        int nRet = 0;

        for (unsigned int i = 0; i < nDeviceNum; i++)
        {
            MV_FG_DEVICE_INFO stDeviceInfo = { 0 };

            nRet = MV_FG_GetDeviceInfo(hInterface, i, &stDeviceInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get info of No.%d device failed! %#x\n", i, nRet);
                return false;
            }

            switch (stDeviceInfo.nDevType)
            {
            case MV_FG_CXP_DEVICE:
                {
                    printf("[CXP]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chSerialNumber);
                    break;
                }
            case MV_FG_GEV_DEVICE:
                {
                    printf("[GEV]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chModelName,
```

```
                    stDeviceInfo.DevInfo.stGEVDevInfo.chSerialNumber);
                break;
            }
        case MV_FG_CAMERALINK_DEVICE:
            {
                printf("[CML]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chSerialNumber);
                break;
            }
        default:
            {
                printf("Unknown device type.\n");
                return false;
            }
        }
    }

    return true;
}

int main()
{
    int                 nRet = 0;
    IFHANDLE            hInterface = NULL;
    DEVHANDLE           hDevice = NULL;
    STREAMHANDLE        hStream = NULL;

    do
    {
        // Enumerate frame grabbers.
        bool bChanged = false;
        nRet = MV_FG_UpdateInterfaceList(MV_FG_CXP_INTERFACE | MV_FG_GEV_INTERFACE |
MV_FG_CAMERALINK_INTERFACE, &bChanged);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Update interface list failed! %#x\n", nRet);
            break;
        }

        // Get the number of frame grabbers.
        unsigned int nInterfaceNum = 0;
        nRet = MV_FG_GetNumInterfaces(&nInterfaceNum);
        if (MV_FG_SUCCESS != nRet || 0 == nInterfaceNum)
        {
            printf("No interface found! return = %d, number = %d\n", nRet, nInterfaceNum);
            break;
        }
```

```
// Display frame grabber information.
if (false == PrintInterfaceInfo(nInterfaceNum))
{
    break;
}

// Select frame grabber.
int nInterfaceIndex = -1;
printf("Select an interface: ");
scanf_s("%d", &nInterfaceIndex);
ClearStdin();

if (nInterfaceIndex < 0 || nInterfaceIndex >= (int)nInterfaceNum)
{
    printf("Invalid interface index.\nQuit.\n");
    break;
}

// Open the frame grabber and get the frame grabber handle.
nRet = MV_FG_OpenInterface((unsigned int)nInterfaceIndex, &hInterface);
if (MV_FG_SUCCESS != nRet)
{
    printf("Open No.%d interface failed! %#x\n", nInterfaceIndex, nRet);
    break;
}

// Register the exception information callback function of the frame grabber.
//nRet = MV_FG_RegisterExceptionCallBack(hInterface, ExceptionCb, hInterface);
//if (MV_FG_SUCCESS != nRet)
//{
//    printf("Register interface exception callback failed!\n");
//    break;
//}

// Enumerate cameras of the frame grabber.
nRet = MV_FG_UpdateDeviceList(hInterface, &bChanged);
if (MV_FG_SUCCESS != nRet)
{
    printf("Update device list failed! %#x\n", nRet);
    break;
}

// Get the number of devices.
unsigned int nDeviceNum = 0;
nRet = MV_FG_GetNumDevices(hInterface, &nDeviceNum);
if (MV_FG_SUCCESS != nRet || 0 == nDeviceNum)
{
    printf("No device found! return = %d, number = %d\n", nRet, nDeviceNum);
    break;
}
```

```
// Display device information.
if (false == PrintDeviceInfo(hInterface, nDeviceNum))
{
    break;
}

// Select the device.
int nDeviceIndex = -1;
printf("Select a device: ");
scanf_s("%d", &nDeviceIndex);
ClearStdin();

if (nDeviceIndex < 0 || nDeviceIndex >= (int)nDeviceNum)
{
    printf("Invalid device index.\nQuit.\n");
    break;
}

// Open the device and get the device handle.
nRet = MV_FG_OpenDevice(hInterface, (unsigned int)nDeviceIndex, &hDevice);
if (MV_FG_SUCCESS != nRet)
{
    printf("Open No.%d device failed! %#x\n", nDeviceIndex, nRet);
    hDevice = NULL;
    break;
}

// Register the exception information callback function of the device.
//nRet = MV_FG_RegisterExceptionCallBack(hDevice, ExceptionCb, hDevice);
//if (MV_FG_SUCCESS != nRet)
//{
//    printf("Register device exception callback failed!\n");
//    break;
//}

// Disable trigger mode.
nRet = MV_FG_SetEnumValueByString(hDevice, "TriggerMode", "Off");
if (MV_FG_SUCCESS != nRet)
{
    printf("Turn off trigger mode failed! %#x\n", nRet);
    break;
}

// Get the number of stream channels.
unsigned int nStreamNum = 0;
nRet = MV_FG_GetNumStreams(hDevice, &nStreamNum);
if (MV_FG_SUCCESS != nRet || 0 == nStreamNum)
{
    printf("No stream available! return = %d, number = %d\n", nRet, nStreamNum);
    break;
}
```

```
        // Open stream channel (currently only one stream channel is supported at a time).
        nRet = MV_FG_OpenStream(hDevice, 0, &hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Open stream failed! %#x\n", nRet);
            break;
        }

        // Register the exception information callback function of the stream channel.
        //nRet = MV_FG_RegisterExceptionCallBack(hStream, ExceptionCb, hStream);
        //if (MV_FG_SUCCESS != nRet)
        //{
        //    printf("Register stream exception callback failed!\n");
        //    break;
        //}

        // Set the number of internal buffers for the SDK.
        nRet = MV_FG_SetBufferNum(hStream, BUFFER_NUMBER);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Set buffer number failed! %#x\n", nRet);
            break;
        }

        // Create thread for image acquisition.
        void* hThreadHandle = (void*)_beginthreadex(NULL, 0, GrabbingThread, hStream, 0, NULL);
        if (NULL == hThreadHandle)
        {
            printf("Create thread failed!\n");
            break;
        }

        printf("Press any key to stop acquisition.\n");
        WaitForKeyPress();

        // Stop image acquisition thread.
        g_bExit = true;
        WaitForSingleObject(hThreadHandle, INFINITE);
        CloseHandle(hThreadHandle);
        hThreadHandle = NULL;
    } while (0);

    // Close the stream channel.
    if (NULL != hStream)
    {
        nRet = MV_FG_CloseStream(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close stream failed! %#x\n", nRet);
        }
```

```
        hStream = NULL;
    }

    // Close the device.
    if (NULL != hDevice)
    {
        nRet = MV_FG_CloseDevice(hDevice);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close device failed! %#x\n", nRet);
        }
        hDevice = NULL;
    }

    // Close the frame grabber.
    if (NULL != hInterface)
    {
        nRet = MV_FG_CloseInterface(hInterface);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close interface failed! %#x\n", nRet);
        }
        hInterface = NULL;
    }

    printf("Press any key to exit.\n");
    WaitForKeyPress();

    return 0;
}
```

A.5 Get Chunk Data

The following sample codes show how to enable and configure chunk data and get the chunk data information.

```
#include <stdio.h>
#include <Windows.h>
#include <process.h>
#include <conio.h>
#include "MVFGControl.h"

#define BUFFER_NUMBER                   3               // Number of requested buffers
#define CHUNK_ID_TIMESTAMP_LITTLE    0xa5a50101         // Timestamp
#define CHUNK_ID_EXPOSURE_LITTLE     0xa5a50103         // Exposure

// Wait for key press.
void WaitForKeyPress(void)
{
    while(!_kbhit())
    {
```

```
            Sleep(10);
        }
        _getch();
}

// Clear residual data from stdin.
void ClearStdin(void)
{
        char c = '\0';

        while (1)
        {
            c = getchar();
            if ('\n' == c || EOF == c)
            {
                break;
            }
        }
}

// Exception information callback function.
void ExceptionCb(MV_FG_EXCEPTION_TYPE enExceptionType, void* pUser)
{
        switch(enExceptionType)
        {
        case EXCEPTION_TYPE_INTERFACE_DISCONNECT:
            {
                printf("Exception: Interface Disconnected!\n");
                break;
            }
        case EXCEPTION_TYPE_DEVICE_DISCONNECT:
            {
                printf("Exception: Device Disconnected!\n");
                break;
            }
        case EXCEPTION_TYPE_STREAM_ABNORMAL_IMAGE:
            {
                printf("Exception: Abnormal Image!\n");
                break;
            }
        case EXCEPTION_TYPE_STREAM_LIST_OVERFLOW:
            {
                printf("Exception: Buffer List Overflow, Clear the Oldest Frame!\n");
                break;
            }
        case EXCEPTION_TYPE_STREAM_DISCONNECTED:
            {
                printf("Exception: Stream Disconnected!\n");
                break;
            }
        default:
```

```
        {
            printf("Unknown Exception!\n");
            break;
        }
    }
}

// Frame buffer information callback function.
void FrameCb(MV_FG_BUFFER_INFO* pstBufferInfo, void* pUser)
{
    if (pstBufferInfo)
    {
        // Print image information.
        printf("FrameNumber:%2I64d%, Width:%d, Height:%d, Chunk Num:%d\n",
            pstBufferInfo->nFrameID, pstBufferInfo->nWidth, pstBufferInfo->nHeight,
pstBufferInfo->nNumChunks);

        int                         nRet = 0;
        STREAMHANDLE                hStream = (STREAMHANDLE)pUser;
        unsigned int                nChunkNum = pstBufferInfo->nNumChunks;      // Number of
chunks
        MV_FG_CHUNK_DATA_INFO       stChunkDataInfo = { 0 };                    // Chunk data
information

        // Print chunk data information.
        printf("********************\n");
        for (unsigned int i = 0; i < nChunkNum; i++)
        {
            memset(&stChunkDataInfo, 0, sizeof(MV_FG_CHUNK_DATA_INFO));
            nRet = MV_FG_GetBufferChunkData(hStream, pstBufferInfo, i, &stChunkDataInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get No.%d chunk data failed! %#x\n", i, nRet);
                printf("********************\n");
                return;
            }

            switch (stChunkDataInfo.nChunkID)
            {
            case CHUNK_ID_TIMESTAMP_LITTLE:
                printf("Chunk ID[%#x], Chunk length[%d], Chunk data[%d]\n",
                    stChunkDataInfo.nChunkID, stChunkDataInfo.nChunkLen,
*((uint32_t*)stChunkDataInfo.pChunkData));
                break;
            case CHUNK_ID_EXPOSURE_LITTLE:
                printf("Chunk ID[%#x], Chunk length[%d], Chunk data[%f]\n",
                    stChunkDataInfo.nChunkID, stChunkDataInfo.nChunkLen,
*((float*)stChunkDataInfo.pChunkData));
                break;
            default:
                break;
```

```
                }
            }
            printf("********************\n");
        }
}

// Print frame grabber information.
bool PrintInterfaceInfo(unsigned int nInterfaceNum)
{
    int nRet = 0;

    for (unsigned int i = 0; i < nInterfaceNum; i++)
    {
        MV_FG_INTERFACE_INFO stInterfaceInfo = { 0 };

        nRet = MV_FG_GetInterfaceInfo(i, &stInterfaceInfo);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Get info of No.%d interface failed! %#x\n", i, nRet);
            return false;
        }

        switch (stInterfaceInfo.nTLayerType)
        {
        case MV_FG_CXP_INTERFACE:
            {
                printf("[CXP]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chSerialNumber);
                break;
            }
        case MV_FG_GEV_INTERFACE:
            {
                printf("[GEV]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chSerialNumber);
                break;
            }
        case MV_FG_CAMERALINK_INTERFACE:
            {
                printf("[CML]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chSerialNumber);
                break;
            }
```

```
            default:
                {
                    printf("Unknown interface type.\n");
                    return false;
                }
            }
        }

        return true;
}

// Print device information.
bool PrintDeviceInfo(IFHANDLE hInterface, unsigned int nDeviceNum)
{
        int nRet = 0;

        for (unsigned int i = 0; i < nDeviceNum; i++)
        {
            MV_FG_DEVICE_INFO stDeviceInfo = { 0 };

            nRet = MV_FG_GetDeviceInfo(hInterface, i, &stDeviceInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get info of No.%d device failed! %#x\n", i, nRet);
                return false;
            }

            switch (stDeviceInfo.nDevType)
            {
            case MV_FG_CXP_DEVICE:
                {
                    printf("[CXP]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stCXPDevInfo.chSerialNumber);
                    break;
                }
            case MV_FG_GEV_DEVICE:
                {
                    printf("[GEV]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chUserDefinedName,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chModelName,
                        stDeviceInfo.DevInfo.stGEVDevInfo.chSerialNumber);
                    break;
                }
            case MV_FG_CAMERALINK_DEVICE:
                {
                    printf("[CML]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
```

```
                    stDeviceInfo.DevInfo.stCMLDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chSerialNumber);
                break;
            }
        default:
            {
                printf("Unknown device type.\n");
                return false;
            }
        }
    }

    return true;
}

int main()
{
    int                 nRet = 0;
    IFHANDLE            hInterface = NULL;
    DEVHANDLE           hDevice = NULL;
    STREAMHANDLE        hStream = NULL;

    do
    {
        // Enumerate frame grabbers.
        bool bChanged = false;
        nRet = MV_FG_UpdateInterfaceList(MV_FG_CXP_INTERFACE | MV_FG_GEV_INTERFACE |
MV_FG_CAMERALINK_INTERFACE, &bChanged);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Update interface list failed! %#x\n", nRet);
            break;
        }

        // Get the number of frame grabbers.
        unsigned int nInterfaceNum = 0;
        nRet = MV_FG_GetNumInterfaces(&nInterfaceNum);
        if (MV_FG_SUCCESS != nRet || 0 == nInterfaceNum)
        {
            printf("No interface found! return = %d, number = %d\n", nRet, nInterfaceNum);
            break;
        }

        // Display frame grabber information.
        if (false == PrintInterfaceInfo(nInterfaceNum))
        {
            break;
        }

        // Select frame grabber.
```

```
int nInterfaceIndex = -1;
printf("Select an interface: ");
scanf_s("%d", &nInterfaceIndex);
ClearStdin();

if (nInterfaceIndex < 0 || nInterfaceIndex >= (int)nInterfaceNum)
{
    printf("Invalid interface index.\nQuit.\n");
    break;
}

// Enable the frame grabber with specified access mode and get the frame grabber handle.
const int nAccessMode = 2;          // 0 - Unknown, 1 - Read Only, 2 - Control, 3 - Exclusive
nRet = MV_FG_OpenInterfaceEx((unsigned int)nInterfaceIndex, nAccessMode, &hInterface);
if (MV_FG_SUCCESS != nRet)
{
    printf("Open No.%d interface failed! %#x\n", nInterfaceIndex, nRet);
    break;
}

// Register the exception information callback function of the frame grabber.
//nRet = MV_FG_RegisterExceptionCallBack(hInterface, ExceptionCb, hInterface);
//if (MV_FG_SUCCESS != nRet)
//{
//    printf("Register interface exception callback failed!\n");
//    break;
//}

// Enumerate cameras of the frame grabber.
nRet = MV_FG_UpdateDeviceList(hInterface, &bChanged);
if (MV_FG_SUCCESS != nRet)
{
    printf("Update device list failed! %#x\n", nRet);
    break;
}

// Get the number of devices.
unsigned int nDeviceNum = 0;
nRet = MV_FG_GetNumDevices(hInterface, &nDeviceNum);
if (MV_FG_SUCCESS != nRet || 0 == nDeviceNum)
{
    printf("No device found! return = %d, number = %d\n", nRet, nDeviceNum);
    break;
}

// Display device information.
if (false == PrintDeviceInfo(hInterface, nDeviceNum))
{
    break;
}
```

```
// Select device.
int nDeviceIndex = -1;
printf("Select a device: ");
scanf_s("%d", &nDeviceIndex);
ClearStdin();

if (nDeviceIndex < 0 || nDeviceIndex >= (int)nDeviceNum)
{
    printf("Invalid device index.\nQuit.\n");
    break;
}

// Open the device and get the device handle.
nRet = MV_FG_OpenDevice(hInterface, (unsigned int)nDeviceIndex, &hDevice);
if (MV_FG_SUCCESS != nRet)
{
    printf("Open No.%d device failed! %#x\n", nDeviceIndex, nRet);
    hDevice = NULL;
    break;
}

// Register the exception information callback function of the device.
//nRet = MV_FG_RegisterExceptionCallBack(hDevice, ExceptionCb, hDevice);
//if (MV_FG_SUCCESS != nRet)
//{
//    printf("Register device exception callback failed!\n");
//    break;
//}

// Enable chunk mode.
nRet = MV_FG_SetBoolValue(hDevice, "ChunkModeActive", true);
if (MV_FG_SUCCESS != nRet)
{
    printf("Set chunk mode failed! %#x\n", nRet);
    break;
}

// Set Chunk Selector to Exposure.
nRet = MV_FG_SetEnumValueByString(hDevice, "ChunkSelector", "Exposure");
if (MV_FG_SUCCESS != nRet)
{
    printf("Set exposure chunk failed! %#x\n", nRet);
    break;
}

// Enable chunk.
nRet = MV_FG_SetBoolValue(hDevice, "ChunkEnable", true);
if (MV_FG_SUCCESS != nRet)
{
    printf("Set exposure chunk enable failed! %#x\n", nRet);
    break;
```

```
}

// Set Chunk Selector to Timestamp.
nRet = MV_FG_SetEnumValueByString(hDevice, "ChunkSelector", "Timestamp");
if (MV_FG_SUCCESS != nRet)
{
    printf("Set timestamp chunk failed! %#x\n", nRet);
    break;
}

// Enable chunk.
nRet = MV_FG_SetBoolValue(hDevice, "ChunkEnable", true);
if (MV_FG_SUCCESS != nRet)
{
    printf("Set timestamp chunk enable failed! %#x\n", nRet);
    break;
}

// Disable trigger mode.
nRet = MV_FG_SetEnumValueByString(hDevice, "TriggerMode", "Off");
if (MV_FG_SUCCESS != nRet)
{
    printf("Turn off trigger mode failed! %#x\n", nRet);
    break;
}

// Get the number of stream channels.
unsigned int nStreamNum = 0;
nRet = MV_FG_GetNumStreams(hDevice, &nStreamNum);
if (MV_FG_SUCCESS != nRet || 0 == nStreamNum)
{
    printf("No stream available! return = %d, number = %d\n", nRet, nStreamNum);
    break;
}

// Open stream channel (currently only one stream channel is supported at a time).
nRet = MV_FG_OpenStream(hDevice, 0, &hStream);
if (MV_FG_SUCCESS != nRet)
{
    printf("Open stream failed! %#x\n", nRet);
    break;
}

// Register the exception information callback function of the stream channel.
//nRet = MV_FG_RegisterExceptionCallBack(hStream, ExceptionCb, hStream);
//if (MV_FG_SUCCESS != nRet)
//{
//    printf("Register stream exception callback failed!\n");
//    break;
//}
```

```
        // Set the number of internal buffers for the SDK.
        nRet = MV_FG_SetBufferNum(hStream, BUFFER_NUMBER);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Set buffer number failed! %#x\n", nRet);
            break;
        }

        // Register the frame buffer information callback function.
        nRet = MV_FG_RegisterFrameCallBack(hStream, FrameCb, hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Register frame callback failed! %#x\n", nRet);
            break;
        }

        // Start image acquisition.
        nRet = MV_FG_StartAcquisition(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Start acquistion failed! %#x\n", nRet);
            return nRet;
        }

        printf("Press any key to stop acquisition.\n");
        WaitForKeyPress();

        // Stop image acquisition.
        nRet = MV_FG_StopAcquisition(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Stop acquisition failed! %#x\n", nRet);
            return nRet;
        }
    } while (0);

    // Close stream channel.
    if (NULL != hStream)
    {
        nRet = MV_FG_CloseStream(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close stream failed! %#x\n", nRet);
        }
        hStream = NULL;
    }

    // Close the device.
    if (NULL != hDevice)
    {
        nRet = MV_FG_CloseDevice(hDevice);
```

```
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close device failed! %#x\n", nRet);
        }
        hDevice = NULL;
    }

    // Close the frame grabber.
    if (NULL != hInterface)
    {
        nRet = MV_FG_CloseInterface(hInterface);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close interface failed! %#x\n", nRet);
        }
        hInterface = NULL;
    }

    printf("Press any key to exit.\n");
    WaitForKeyPress();

    return 0;
}
```

A.6 Load Dynamic Link Library

The following sample codes show how to use the frame grabber SDK with dynamic calling.

```
#include <stdio.h>
#include <Windows.h>
#include <process.h>
#include <conio.h>
#include "MVFGDefines.h"
#include "MVFGErrorDefine.h"

typedef unsigned char*    (__stdcall * DLL_GetSDKVersion)              ();

typedef int               (__stdcall * DLL_UpdateInterfaceList)     (IN unsigned int nTLayerType, OUT
bool8_t *pbChanged);
typedef int               (__stdcall * DLL_GetNumInterfaces)         (OUT unsigned int *pnNumIfaces);
typedef int               (__stdcall * DLL_GetInterfaceInfo)         (IN unsigned int nIndex, OUT
MV_FG_INTERFACE_INFO *pstIfaceInfo);
typedef bool               (__stdcall * DLL_GetInterfaceInfoEx)       (IN unsigned int nIndex, IN
MV_FG_INTERFACE_INFO_CMD enIfaceInfoCmd, OUT MV_FG_INFO_VALUE *pstInfoValue);
typedef int               (__stdcall * DLL_OpenInterface)              (IN unsigned int nIndex, OUT
IFHANDLE* phIface);
typedef int               (__stdcall * DLL_OpenInterfaceEx)          (IN unsigned int nIndex, IN int
nAccess, OUT IFHANDLE* phIface);
typedef int               (__stdcall * DLL_CloseInterface)           (IN IFHANDLE hIface);
typedef int               (__stdcall * DLL_UpdateDeviceList)         (IN IFHANDLE hIface, OUT bool8_t
*pbChanged);
```

```
typedef int (__stdcall * DLL_GetNumDevices) (IN IFHANDLE hIface, OUT unsigned int *pnNumDevices);
typedef int (__stdcall * DLL_GetDeviceInfo) (IN IFHANDLE hIface, IN unsigned int nIndex, OUT MV_FG_DEVICE_INFO *pstDevInfo);
typedef int (__stdcall * DLL_OpenDevice) (IN IFHANDLE hIface, IN unsigned int nIndex, OUT DEVHANDLE* phDevice);
typedef int (__stdcall * DLL_CloseDevice) (IN DEVHANDLE hDevice);
typedef int (__stdcall * DLL_GetNumStreams) (IN DEVHANDLE hDevice, OUT unsigned int *pnNumStreams);
typedef int (__stdcall * DLL_OpenStream) (IN DEVHANDLE hDevice, IN unsigned int nIndex, OUT STREAMHANDLE* phStream);
typedef int (__stdcall * DLL_CloseStream) (IN STREAMHANDLE hStream);
typedef int (__stdcall * DLL_SetBufferNum) (IN STREAMHANDLE hStream, IN unsigned int nBufferNum);
typedef int (__stdcall * DLL_RegisterFrameCallBack) (IN STREAMHANDLE hStream, IN MV_FG_FrameCallBack cbFrame, IN void* pUser);
typedef int (__stdcall * DLL_GetFrameBuffer) (IN STREAMHANDLE hStream, OUT MV_FG_BUFFER_INFO* pstBufferInfo, IN unsigned int nTimeout);
typedef int (__stdcall * DLL_ReleaseFrameBuffer) (IN STREAMHANDLE hStream, IN MV_FG_BUFFER_INFO* pstBufferInfo);
typedef int (__stdcall * DLL_GetBufferChunkData) (IN STREAMHANDLE hStream, IN MV_FG_BUFFER_INFO* pstBufferInfo, IN unsigned int nIndex, OUT MV_FG_CHUNK_DATA_INFO* pstChunkDataInfo);
typedef int (__stdcall * DLL_GetPayloadSize) (IN STREAMHANDLE hStream, OUT unsigned int* pnPayloadSize);
typedef int (__stdcall * DLL_AnnounceBuffer) (IN STREAMHANDLE hStream, IN void *pBuffer, IN unsigned int nSize, IN void *pPrivate, OUT BUFFERHANDLE *phBuffer);
typedef int (__stdcall * DLL_RevokeBuffer) (IN STREAMHANDLE hStream, IN BUFFERHANDLE hBuffer, OUT void **pBuffer, OUT void **pPrivate);
typedef int (__stdcall * DLL_FlushQueue) (IN STREAMHANDLE hStream, IN MV_FG_BUFFER_QUEUE_TYPE enQueueType);
typedef int (__stdcall * DLL_StartAcquisition) (IN STREAMHANDLE hStream);
typedef int (__stdcall * DLL_StopAcquisition) (IN STREAMHANDLE hStream);
typedef int (__stdcall * DLL_GetImageBuffer) (IN STREAMHANDLE hStream, OUT BUFFERHANDLE *phBuffer, IN unsigned int nTimeout);
typedef int (__stdcall * DLL_GetBufferInfo) (IN BUFFERHANDLE hBuffer, OUT MV_FG_BUFFER_INFO* pstBufferInfo);
typedef int (__stdcall * DLL_QueueBuffer) (IN BUFFERHANDLE hBuffer);
typedef int (__stdcall * DLL_GetXMLFile) (IN PORTHANDLE hPort, IN OUT unsigned char* pData, IN unsigned int nDataSize, OUT unsigned int* pnDataLen);
typedef int (__stdcall * DLL_GetNodeAccessMode) (IN PORTHANDLE hPort, IN const char * strName, OUT MV_FG_NODE_ACCESS_MODE *penAccessMode);
typedef int (__stdcall * DLL_GetNodeInterfaceType) (IN PORTHANDLE hPort, IN const char * strName, OUT MV_FG_NODE_INTERFACE_TYPE *penInterfaceType);
typedef int (__stdcall * DLL_GetIntValue) (IN PORTHANDLE hPort, IN const char* strKey, OUT MV_FG_INTVALUE *pstIntValue);
typedef int (__stdcall * DLL_SetIntValue) (IN PORTHANDLE hPort, IN const char* strKey, IN int64_t nValue);
typedef int (__stdcall * DLL_GetEnumValue) (IN PORTHANDLE hPort, IN const char* strKey, OUT MV_FG_ENUMVALUE *pstEnumValue);
typedef int (__stdcall * DLL_SetEnumValue) (IN PORTHANDLE hPort, IN const
```

```
char* strKey, IN unsigned int nValue);
typedef int                 (__stdcall * DLL_SetEnumValueByString)    (IN PORTHANDLE hPort, IN const
char* strKey, IN const char* strValue);
typedef int                 (__stdcall * DLL_GetFloatValue)           (IN PORTHANDLE hPort, IN const
char* strKey, OUT MV_FG_FLOATVALUE *pstFloatValue);
typedef int                 (__stdcall * DLL_SetFloatValue)           (IN PORTHANDLE hPort, IN const
char* strKey, IN float fValue);
typedef int                 (__stdcall * DLL_GetBoolValue)            (IN PORTHANDLE hPort, IN const
char* strKey, OUT bool8_t *pbValue);
typedef int                 (__stdcall * DLL_SetBoolValue)            (IN PORTHANDLE hPort, IN const
char* strKey, IN bool8_t bValue);
typedef int                 (__stdcall * DLL_GetStringValue)          (IN PORTHANDLE hPort, IN const
char* strKey, OUT MV_FG_STRINGVALUE *pstStringValue);
typedef int                 (__stdcall * DLL_SetStringValue)          (IN PORTHANDLE hPort, IN const
char* strKey, IN const char* strValue);
typedef int                 (__stdcall * DLL_SetCommandValue)         (IN PORTHANDLE hPort, IN const
char* strKey);
typedef int                 (__stdcall * DLL_FeatureSave)             (IN PORTHANDLE hPort, IN const
char* strFileName);
typedef int                 (__stdcall * DLL_FeatureLoad)             (IN PORTHANDLE hPort, IN const
char* strFileName);
typedef int                 (__stdcall * DLL_RegisterExceptionCallBack)(IN PORTHANDLE hPort, IN
MV_FG_ExceptionCallBack cbException, IN void* pUser);
typedef int                 (__stdcall * DLL_ReleaseTLayerResource) (IN unsigned int nTLayerType);

#define BUFFER_NUMBER          3          // Number of requested buffers
#define FILE_NAME_LEN          256        // The maximum length of file name
#define SAVE_IMAGE_NUM         10         // The maximum number of saved images
#define TIMEOUT                1000       // Timeout; unit: millisecond (ms)

bool g_bExit = false;                     // Stop acquisition

// Thread parameters
struct MultiThrParam
{
    void*           pUser;          // User-defined parameter
    HINSTANCE       hDll;           // Dynamic link library handle
};

// Wait for key press.
void WaitForKeyPress(void)
{
    while(!_kbhit())
    {
        Sleep(10);
    }
    _getch();
}

// Clear residual data from stdin.
void ClearStdin(void)
```

```
{
    char c = '\0';

    while (1)
    {
        c = getchar();
        if ('\n' == c || EOF == c)
        {
            break;
        }
    }
}

// Exception information callback function.
void ExceptionCb(MV_FG_EXCEPTION_TYPE enExceptionType, void* pUser)
{
    switch(enExceptionType)
    {
    case EXCEPTION_TYPE_INTERFACE_DISCONNECT:
        {
            printf("Exception: Interface Disconnected!\n");
            break;
        }
    case EXCEPTION_TYPE_DEVICE_DISCONNECT:
        {
            printf("Exception: Device Disconnected!\n");
            break;
        }
    case EXCEPTION_TYPE_STREAM_ABNORMAL_IMAGE:
        {
            printf("Exception: Abnormal Image!\n");
            break;
        }
    case EXCEPTION_TYPE_STREAM_LIST_OVERFLOW:
        {
            printf("Exception: Buffer List Overflow, Clear the Oldest Frame!\n");
            break;
        }
    case EXCEPTION_TYPE_STREAM_DISCONNECTED:
        {
            printf("Exception: Stream Disconnected!\n");
            break;
        }
    default:
        {
            printf("Unknown Exception!\n");
            break;
        }
    }
}
```

```
// Save the original image data.
void SaveRawImage(int nImageNo, MV_FG_BUFFER_INFO* pstImageInfo)
{
	if (pstImageInfo)
	{
		char szFileName[FILE_NAME_LEN] = { 0 };

		sprintf_s(szFileName, FILE_NAME_LEN, "Image_w%d_h%d_n%d.raw", pstImageInfo->nWidth,
pstImageInfo->nHeight, nImageNo);

		FILE* pImageFile = NULL;
		if ((0 != fopen_s(&pImageFile, szFileName, "wb")) || (NULL == pImageFile))
		{
			return;
		}

		fwrite(pstImageInfo->pBuffer, 1, pstImageInfo->nFilledSize, pImageFile);
		fclose(pImageFile);
	}
}

// Image acquisition thread.
unsigned int __stdcall GrabbingThread(void* pUser)
{
	if (pUser)
	{
		MultiThrParam*		pstThreadParam	= (MultiThrParam*)pUser;
		MV_FG_BUFFER_INFO	stFrameInfo		= { 0 };		// Image information
		int			nSaveImage		= 0;			// Number of saved images
		int			nRet			= 0;

		DLL_StartAcquisition DLLStartAcquisition =
(DLL_StartAcquisition)GetProcAddress(pstThreadParam->hDll, "MV_FG_StartAcquisition");
		DLL_GetFrameBuffer DLLGetFrameBuffer =
(DLL_GetFrameBuffer)GetProcAddress(pstThreadParam->hDll, "MV_FG_GetFrameBuffer");
		DLL_ReleaseFrameBuffer DLLReleaseFrameBuffer =
(DLL_ReleaseFrameBuffer)GetProcAddress(pstThreadParam->hDll, "MV_FG_ReleaseFrameBuffer");
		DLL_StopAcquisition DLLStopAcquisition =
(DLL_StopAcquisition)GetProcAddress(pstThreadParam->hDll, "MV_FG_StopAcquisition");

		// Start image acquisition.
		nRet = DLLStartAcquisition(pstThreadParam->pUser);
		if (MV_FG_SUCCESS != nRet)
		{
			printf("Start acquistion failed! %#x\n", nRet);
			return nRet;
		}
		g_bExit = false;

		while (!g_bExit)
		{
```

```
            // Get the buffer information of a frame.
            nRet = DLLGetFrameBuffer(pstThreadParam->pUser, &stFrameInfo, TIMEOUT);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get frame buffer info failed! %#x\n", nRet);
                continue;
            }
            else
            {
                printf("FrameNumber:%2I64d%, Width:%d, Height:%d\n", stFrameInfo.nFrameID,
stFrameInfo.nWidth, stFrameInfo.nHeight);

                if (nSaveImage < SAVE_IMAGE_NUM)
                {
                    SaveRawImage(++nSaveImage, &stFrameInfo);
                }
            }

            // Insert the buffer back to the input queue.
            nRet = DLLReleaseFrameBuffer(pstThreadParam->pUser, &stFrameInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Release frame buffer failed! %#x\n", nRet);
                break;
            }
        }

        // Stop image acquisition.
        nRet = DLLStopAcquisition(pstThreadParam->pUser);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Stop acquisition failed! %#x\n", nRet);
            return nRet;
        }
    }

    return MV_FG_SUCCESS;
}

// Print frame grabber information.
bool PrintInterfaceInfo(HINSTANCE hDll, unsigned int nInterfaceNum)
{
    int nRet = 0;

    DLL_GetInterfaceInfo DLLGetInterfaceInfo = (DLL_GetInterfaceInfo)GetProcAddress(hDll,
"MV_FG_GetInterfaceInfo");

    for (unsigned int i = 0; i < nInterfaceNum; i++)
    {
        MV_FG_INTERFACE_INFO stInterfaceInfo = { 0 };
```

```
        nRet = DLLGetInterfaceInfo(i, &stInterfaceInfo);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Get info of No.%d interface failed! %#x\n", i, nRet);
            return false;
        }

        switch (stInterfaceInfo.nTLayerType)
        {
        case MV_FG_CXP_INTERFACE:
            {
                printf("[CXP]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chSerialNumber);
                break;
            }
        case MV_FG_GEV_INTERFACE:
            {
                printf("[GEV]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chSerialNumber);
                break;
            }
        case MV_FG_CAMERALINK_INTERFACE:
            {
                printf("[CML]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chSerialNumber);
                break;
            }
        default:
            {
                printf("Unknown interface type.\n");
                return false;
            }
        }
    }

    return true;
}

// Print device information.
bool PrintDeviceInfo(HINSTANCE hDll, IFHANDLE hInterface, unsigned int nDeviceNum)
{
    int nRet = 0;
```

```
    DLL_GetDeviceInfo DLLGetDeviceInfo = (DLL_GetDeviceInfo)GetProcAddress(hDll,
"MV_FG_GetDeviceInfo");

    for (unsigned int i = 0; i < nDeviceNum; i++)
    {
        MV_FG_DEVICE_INFO stDeviceInfo = { 0 };

        nRet = DLLGetDeviceInfo(hInterface, i, &stDeviceInfo);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Get info of No.%d device failed! %#x\n", i, nRet);
            return false;
        }

        switch (stDeviceInfo.nDevType)
        {
        case MV_FG_CXP_DEVICE:
            {
                printf("[CXP]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                    stDeviceInfo.DevInfo.stCXPDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stCXPDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stCXPDevInfo.chSerialNumber);
                break;
            }
        case MV_FG_GEV_DEVICE:
            {
                printf("[GEV]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                    stDeviceInfo.DevInfo.stGEVDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stGEVDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stGEVDevInfo.chSerialNumber);
                break;
            }
        case MV_FG_CAMERALINK_DEVICE:
            {
                printf("[CML]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chSerialNumber);
                break;
            }
        default:
            {
                printf("Unknown device type.\n");
                return false;
            }
        }
    }
```

```
    return true;
}

int main()
{
    HINSTANCE    MVFGCtrlDll = NULL;        // Handle of MVFGControl.dll

    // The default path for the dynamic link library is System Disk:\Program Files (x86)\Common
Files\MVS\Runtime
    MVFGCtrlDll = LoadLibrary("MVFGControl.dll");
    if (NULL == MVFGCtrlDll)
    {
        DWORD errCode = GetLastError();
        printf("Error code! [%ld]\n",errCode);
        printf("Press any key to exit.\n");
        WaitForKeyPress();
        return -1;
    }

    int                    nRet = 0;
    IFHANDLE               hInterface = NULL;
    DEVHANDLE              hDevice = NULL;
    STREAMHANDLE           hStream = NULL;

    do
    {
        // Enumerate frame grabbers.
        bool bChanged = false;
        DLL_UpdateInterfaceList DLLUpdateInterfaceList =
(DLL_UpdateInterfaceList)GetProcAddress(MVFGCtrlDll, "MV_FG_UpdateInterfaceList");
        nRet = DLLUpdateInterfaceList(MV_FG_CXP_INTERFACE | MV_FG_GEV_INTERFACE |
MV_FG_CAMERALINK_INTERFACE, &bChanged);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Update interface list failed! %#x\n", nRet);
            break;
        }

        // Get the number of frame grabbers.
        unsigned int nInterfaceNum = 0;
        DLL_GetNumInterfaces DLLGetNumInterfaces =
(DLL_GetNumInterfaces)GetProcAddress(MVFGCtrlDll, "MV_FG_GetNumInterfaces");
        nRet = DLLGetNumInterfaces(&nInterfaceNum);
        if (MV_FG_SUCCESS != nRet || 0 == nInterfaceNum)
        {
            printf("No interface found! return = %d, number = %d\n", nRet, nInterfaceNum);
            break;
        }

        // Display frame grabber information.
```

```
        if (false == PrintInterfaceInfo(MVFGCtrlDll, nInterfaceNum))
        {
            break;
        }

        // Select frame grabber.
        int nInterfaceIndex = -1;
        printf("Select an interface: ");
        scanf_s("%d", &nInterfaceIndex);
        ClearStdin();

        if (nInterfaceIndex < 0 || nInterfaceIndex >= (int)nInterfaceNum)
        {
            printf("Invalid interface index.\nQuit.\n");
            break;
        }

        // Open the frame grabber with specified access mode and get the frame grabber handle.
        const int nAccessMode = 2;          // 0 - Unknown, 1 - Read Only, 2 - Control, 3 - Exclusive
        DLL_OpenInterfaceEx DLLOpenInterfaceEx =
(DLL_OpenInterfaceEx)GetProcAddress(MVFGCtrlDll, "MV_FG_OpenInterfaceEx");
        nRet = DLLOpenInterfaceEx((unsigned int)nInterfaceIndex, nAccessMode, &hInterface);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Open No.%d interface failed! %#x\n", nInterfaceIndex, nRet);
            break;
        }

        // Enumerate cameras of the frame grabber.
        DLL_UpdateDeviceList DLLUpdateDeviceList =
(DLL_UpdateDeviceList)GetProcAddress(MVFGCtrlDll, "MV_FG_UpdateDeviceList");
        nRet = DLLUpdateDeviceList(hInterface, &bChanged);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Update device list failed! %#x\n", nRet);
            break;
        }

        // Get the number of devices.
        unsigned int nDeviceNum = 0;
        DLL_GetNumDevices DLLGetNumDevices =
(DLL_GetNumDevices)GetProcAddress(MVFGCtrlDll, "MV_FG_GetNumDevices");
        nRet = DLLGetNumDevices(hInterface, &nDeviceNum);
        if (MV_FG_SUCCESS != nRet || 0 == nDeviceNum)
        {
            printf("No device found! return = %d, number = %d\n", nRet, nDeviceNum);
            break;
        }

        // Display device information.
        if (false == PrintDeviceInfo(MVFGCtrlDll, hInterface, nDeviceNum))
```

```
        {
            break;
        }

        // Select device.
        int nDeviceIndex = -1;
        printf("Select a device: ");
        scanf_s("%d", &nDeviceIndex);
        ClearStdin();

        if (nDeviceIndex < 0 || nDeviceIndex >= (int)nDeviceNum)
        {
            printf("Invalid device index.\nQuit.\n");
            break;
        }

        // Open the device and get the device handle.
        DLL_OpenDevice DLLOpenDevice = (DLL_OpenDevice)GetProcAddress(MVFGCtrlDll,
"MV_FG_OpenDevice");
        nRet = DLLOpenDevice(hInterface, (unsigned int)nDeviceIndex, &hDevice);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Open No.%d device failed! %#x\n", nDeviceIndex, nRet);
            hDevice = NULL;
            break;
        }

        // Disable trigger mode.
        DLL_SetEnumValueByString DLLSetEnumValueByString =
(DLL_SetEnumValueByString)GetProcAddress(MVFGCtrlDll, "MV_FG_SetEnumValueByString");
        nRet = DLLSetEnumValueByString(hDevice, "TriggerMode", "Off");
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Turn off trigger mode failed! %#x\n", nRet);
            break;
        }

        // Get the number of stream channels.
        unsigned int nStreamNum = 0;
        DLL_GetNumStreams DLLGetNumStreams =
(DLL_GetNumStreams)GetProcAddress(MVFGCtrlDll, "MV_FG_GetNumStreams");
        nRet = DLLGetNumStreams(hDevice, &nStreamNum);
        if (MV_FG_SUCCESS != nRet || 0 == nStreamNum)
        {
            printf("No stream available! return = %d, number = %d\n", nRet, nStreamNum);
            break;
        }

        // Open stream channel (currently only one stream channel is supported at a time).
        DLL_OpenStream DLLOpenStream = (DLL_OpenStream)GetProcAddress(MVFGCtrlDll,
"MV_FG_OpenStream");
```

```
        nRet = DLLOpenStream(hDevice, 0, &hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Open stream failed! %#x\n", nRet);
            break;
        }

        // Set the number of internal buffers for the SDK.
        DLL_SetBufferNum DLLSetBufferNum = (DLL_SetBufferNum)GetProcAddress(MVFGCtrlDll,
"MV_FG_SetBufferNum");
        nRet = DLLSetBufferNum(hStream, BUFFER_NUMBER);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Set buffer number failed! %#x\n", nRet);
            break;
        }

        // Create thread for image acquisition.
        MultiThrParam stThreadParam = { 0 };
        stThreadParam.pUser = hStream;
        stThreadParam.hDll    = MVFGCtrlDll;
        void* hThreadHandle = (void*)_beginthreadex(NULL, 0, GrabbingThread,
(void*)&stThreadParam, 0, NULL);
        if (NULL == hThreadHandle)
        {
            printf("Create thread failed!\n");
            break;
        }

        printf("Press any key to stop acquisition.\n");
        WaitForKeyPress();

        // Stop image acquisition thread.
        g_bExit = true;
        WaitForSingleObject(hThreadHandle, INFINITE);
        CloseHandle(hThreadHandle);
        hThreadHandle = NULL;
    } while (0);

    // Close stream channel.
    if (NULL != hStream)
    {
        DLL_CloseStream DLLCloseStream = (DLL_CloseStream)GetProcAddress(MVFGCtrlDll,
"MV_FG_CloseStream");
        nRet = DLLCloseStream(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close stream failed! %#x\n", nRet);
        }
        hStream = NULL;
    }
```

```
    // Close the device.
    if (NULL != hDevice)
    {
        DLL_CloseDevice DLLCloseDevice = (DLL_CloseDevice)GetProcAddress(MVFGCtrlDll,
"MV_FG_CloseDevice");
        nRet = DLLCloseDevice(hDevice);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close device failed! %#x\n", nRet);
        }
        hDevice = NULL;
    }

    // Close the frame grabber.
    if (NULL != hInterface)
    {
        DLL_CloseInterface DLLCloseInterface = (DLL_CloseInterface)GetProcAddress(MVFGCtrlDll,
"MV_FG_CloseInterface");
        nRet = DLLCloseInterface(hInterface);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close interface failed! %#x\n", nRet);
        }
        hInterface = NULL;
    }

    FreeLibrary(MVFGCtrlDll);

    printf("Press any key to exit.\n");
    WaitForKeyPress();

    return 0;
}
```

A.7 Receive Events

The following sample codes show how to configure the frame grabber event, how to register event callback, and how to handle the event information received in the callback function.

```
#include <Windows.h>
#include <conio.h>
#include <stdio.h>
#include <process.h>
#include "MVFGControl.h"

bool g_bExit = FALSE;
#define BUFFER_NUMBER                3          // Number of requested buffers


void __stdcall EventCallBack(MV_FG_EVENT_INFO* pstEventInfo, void* pUser)
```

```
{
	static int nEventNum = 0;

	nEventNum++;

	if (NULL != pstEventInfo)
	{
		printf("%d Event: name %s	id 0x%x time %lld \r\n", nEventNum, pstEventInfo->EventName,
pstEventInfo->nEventID, pstEventInfo->nTimestamp);
	}
}

// Wait for key press.
void WaitForKeyPress(void)
{
	while(!_kbhit())
	{
		Sleep(10);
	}
	_getch();
}

// Clear residual data from stdin.
void ClearStdin(void)
{
	char c = '0';

	do
	{
		c = getchar();
		if (c == '\n' ||c == EOF)
		{
			break;
		}
	}
	while(TRUE);
}

// Stream acquiring thread.
unsigned int __stdcall GrabbingThread(void* pUser)
{
	if (pUser)
	{
		STREAMHANDLE		hStream = (STREAMHANDLE)pUser;
		BUFFERHANDLE		hBuffer = NULL;
		MV_FG_BUFFER_INFO	stFrameInfo = {0};
		int			nSaveImage = 10;
		int			nFrameNum	= 0;

		// Start acquiring images.
		int nRet = MV_FG_StartAcquisition(hStream);
```

```
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Start Acquisition failed, %#x\n", nRet);
            return nRet;
        }

        while(!g_bExit)
        {
            // Get the image buffer.
            nRet = MV_FG_GetImageBuffer(hStream, &hBuffer, 1000);
            if (MV_FG_SUCCESS == nRet)
            {
                nRet = MV_FG_GetBufferInfo(hBuffer, &stFrameInfo);
                if (MV_FG_SUCCESS == nRet)
                {
                    nFrameNum++;
                    printf("FrameNumber %8d: %-8I64d\tWidth: %d\tHeight: %d\n", nFrameNum,
stFrameInfo.nFrameID, stFrameInfo.nWidth, stFrameInfo.nHeight);
                }

                // Insert the image buffer back to the input queue.
                nRet = MV_FG_QueueBuffer(hBuffer);
                if (MV_FG_SUCCESS != nRet)
                {
                    printf("Queue Buffer error, %#x\n", nRet);
                    break;
                }
            }
            else
            {
                printf("Get image buffer failed, %#x\n", nRet);
            }

        }

        // Stop acquisition.
        nRet = MV_FG_StopAcquisition(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Stop Acquisition failed, %#x\n", nRet);
            return nRet;
        }
    }

    return MV_FG_SUCCESS;
}

int main(int argc, char** argv)
{
    int                 nRet = MV_FG_SUCCESS;
    IFHANDLE            hInterface = NULL;
```

```
    DEVHANDLE           hDevice = NULL;
    STREAMHANDLE        hStream = NULL;
    BUFFERHANDLE        hBuffer[BUFFER_NUMBER] = {0};
    void*               pBuffer[BUFFER_NUMBER] = {0};

    do
    {
        // Enumerate frame grabbers.
        bool bChanged = false;
        nRet = MV_FG_UpdateInterfaceList(MV_FG_CXP_INTERFACE |    MV_FG_GEV_INTERFACE |
MV_FG_CAMERALINK_INTERFACE, &bChanged);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Update Interface List error, %#x\n", nRet);
            break;
        }

        // Get the number of frame grabbers.
        unsigned int nInterfaceNumber = 0;
        nRet = MV_FG_GetNumInterfaces(&nInterfaceNumber);
        if (MV_FG_SUCCESS != nRet || 0 == nInterfaceNumber)
        {
            printf("No Interface found\n");
            break;
        }

        // Print the frame grabber information.
        for (unsigned int i = 0; i < nInterfaceNumber; i++)
        {
            MV_FG_INTERFACE_INFO stInterfaceInfo = {0};

            nRet = MV_FG_GetInterfaceInfo(i, &stInterfaceInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get info of No.%d Interface error, %#x\n", i, nRet);
                break;
            }

            if (stInterfaceInfo.nTLayerType == MV_FG_CXP_INTERFACE)
            {
                printf("[CXP]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stCXPIfaceInfo.chSerialNumber);
            }
            else if (stInterfaceInfo.nTLayerType == MV_FG_GEV_INTERFACE)
            {
                printf("[GEV]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chDisplayName,
```

```
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stGEVIfaceInfo.chSerialNumber);
            }
            else if (stInterfaceInfo.nTLayerType == MV_FG_CAMERALINK_INTERFACE)
            {
                printf("[CML]No.%d Interface:
\n\tDisplayName: %s\n\tInterfaceID: %s\n\tSerialNumber:%s\n", i,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chDisplayName,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chInterfaceID,
                    stInterfaceInfo.IfaceInfo.stCMLIfaceInfo.chSerialNumber);
            }
        }
        if (MV_FG_SUCCESS != nRet)
        {
            break;
        }

        // Select a frame grabber and get the index.
        int nSelectedInterfaceIndex = -1;
        printf("Select an interface: ");
        scanf_s("%d", &nSelectedInterfaceIndex);
        ClearStdin();

        if ((nSelectedInterfaceIndex < 0) || (nSelectedInterfaceIndex >= (int)nInterfaceNumber))
        {
            printf("invalid interface index, Quit\n");
            break;
        }

        // Open the frame grabber. The frame grabber handle will be returned.
        nRet = MV_FG_OpenInterface(nSelectedInterfaceIndex, &hInterface);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Open No.%d Interface error, %#x\n", nSelectedInterfaceIndex, nRet);
            break;
        }

        // Register the callback function for events
        nRet = MV_FG_RegisterEventCallBack(hInterface, "ReceiveImageFrameStart0", EventCallBack,
NULL);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("MV_FG_RegisterEventCallBack event %s error, %#x\n", "ReceiveImageFrameStart0",
nRet);
            break;
        }

        // Set the event type to stream event (you can set types of other events under the camera
node EventCategory)
        nRet = MV_FG_SetEnumValueByString(hInterface, "EventCategory", "StreamEvent");
        if (MV_FG_SUCCESS != nRet)
```

```
        {
            printf("MV_FG_SetEnumValueByString EventCategory %s error, %#x\n", "StreamEvent",
nRet);
            break;
        }

        // Set the event channel (Channel0-Channel3)
        nRet = MV_FG_SetEnumValueByString(hInterface, "ChannelSelector", "Channel0");
        if (MV_FG_SUCCESS != nRet)
        {
            printf("MV_FG_SetEnumValueByString ChannelSelector %s error, %#x\n", "Channel0",
nRet);
            break;
        }

        // Set the specific event (you can set other specific events under the camera node
EventSelector)
        nRet = MV_FG_SetEnumValueByString(hInterface, "EventSelector",
"ReceiveImageFrameStart0");
        if (MV_FG_SUCCESS != nRet)
        {
            printf("MV_FG_SetEnumValueByString EventSelector %s error, %#x\n",
"ReceiveImageFrameStart0", nRet);
            break;
        }

        // Enable the event notification
        nRet = MV_FG_SetEnumValueByString(hInterface, "EventNotification", "On");
        if (MV_FG_SUCCESS != nRet)
        {
            printf("MV_FG_SetEnumValueByString EventNotification %s error, %#x\n", "On", nRet);
            break;
        }

        // Enumerate cameras of the frame grabber.
        nRet = MV_FG_UpdateDeviceList(hInterface, &bChanged);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Update Device list error, %#x\n", nRet);
            break;
        }

        // Get and print the device information.
        unsigned int nDeviceNumber = 0;
        nRet = MV_FG_GetNumDevices(hInterface, &nDeviceNumber);
        if (MV_FG_SUCCESS != nRet || 0 == nDeviceNumber)
        {
            printf("No devices found, %#x\n", nRet);
            break;
        }
```

```
        for (unsigned int i = 0; i < nDeviceNumber; i++)
        {
            MV_FG_DEVICE_INFO stDeviceInfo = {0};

            nRet = MV_FG_GetDeviceInfo(hInterface, i, &stDeviceInfo);
            if (MV_FG_SUCCESS != nRet)
            {
                printf("Get Info of No.%u Device error, %#x\n", i, nRet);
                break;
            }

            if (stDeviceInfo.nDevType == MV_FG_CXP_DEVICE)
            {
                printf("[CXP]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                    stDeviceInfo.DevInfo.stCXPDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stCXPDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stCXPDevInfo.chSerialNumber);
            }
            else if (stDeviceInfo.nDevType == MV_FG_GEV_DEVICE)
            {
                printf("[GEV]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                    stDeviceInfo.DevInfo.stGEVDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stGEVDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stGEVDevInfo.chSerialNumber);
            }
            else if (stDeviceInfo.nDevType == MV_FG_CAMERALINK_DEVICE)
            {
                printf("[CML]No.%d Device:
\n\tUserDefinedName: %s\n\tModelName: %s\n\tSerialNumber: %s\n", i,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chUserDefinedName,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chModelName,
                    stDeviceInfo.DevInfo.stCMLDevInfo.chSerialNumber);
            }
        }
        if (MV_FG_SUCCESS != nRet)
        {
            break;
        }

        // Select the device.
        int nSelectedDeviceIndex = -1;
        printf("Select a device: ");
        scanf_s("%d", &nSelectedDeviceIndex);
        ClearStdin();

        if ((nSelectedDeviceIndex < 0) || (nSelectedDeviceIndex >= (int)nDeviceNumber))
        {
            printf("invalid device index, Quit\n");
            break;
```

```
    }

    // Open the device and get the device handle.
    nRet = MV_FG_OpenDevice(hInterface, nSelectedDeviceIndex, &hDevice);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Open device error, %#x\n", nRet);
        hDevice = NULL;
        break;
    }

    // Close the trigger mode.
    nRet = MV_FG_SetEnumValueByString(hDevice, "TriggerMode", "Off");
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Turn off TriggerMode failed, %#x\n", nRet);
        break;
    }

    // Get the number of streaming channels.
    unsigned int nStreamNumber = 0;
    nRet = MV_FG_GetNumStreams(hDevice, &nStreamNumber);
    if (MV_FG_SUCCESS != nRet || 0 == nStreamNumber)
    {
        printf("No Stream available\n");
        break;
    }

    // Open the streaming channel (now only a single streaming channel is supported)
    nRet = MV_FG_OpenStream(hDevice, 0, &hStream);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Open Stream error, %#x\n", nRet);
        break;
    }

    // Get the image size.
    unsigned int nImagePayloadSize = 0;
    nRet = MV_FG_GetPayloadSize(hStream, &nImagePayloadSize);
    if (MV_FG_SUCCESS != nRet)
    {
        printf("Get payload size error, %#x\n", nRet);
        break;
    }

    // Allocate and register the buffer of images.
    for (unsigned int i = 0; i < BUFFER_NUMBER; i++)
    {
        // Allocate the image buffer.
        pBuffer[i] = malloc(nImagePayloadSize);
        if (NULL == pBuffer[i])
```

```
                {
                    printf("Allocate Buffer error\n");
                    nRet = MV_FG_ERR_OUT_OF_MEMORY;
                    break;
                }

                // Register the buffer for SDK.
                nRet = MV_FG_AnnounceBuffer(hStream, pBuffer[i], nImagePayloadSize, NULL,
&(hBuffer[i]));
                if (MV_FG_SUCCESS != nRet)
                {
                    printf("Announce Buffer error, %#x\n", nRet);
                    break;
                }
            }

            if (MV_FG_SUCCESS != nRet)
            {
                break;
            }

            // Insert all buffers back to the input queue.
            nRet = MV_FG_FlushQueue(hStream, MV_FG_BUFFER_QUEUE_ALL_TO_INPUT);
            if (MV_FG_SUCCESS != nRet)
            {
                break;
            }

            // Create a thread for streaming.
            void* hThreadHandle = (void*) _beginthreadex(NULL , 0 , GrabbingThread , hStream, 0 , NULL);
            if (NULL == hThreadHandle)
            {
                printf("Create Thread Error\n");
                break;
            }

            printf("Press any key to stop acquisition.\n");
            WaitForKeyPress();

            g_bExit = true;
            WaitForSingleObject(hThreadHandle, INFINITE);
            CloseHandle(hThreadHandle);
            hThreadHandle = NULL;
        } while (0);

        // Release relative resources.
        if (NULL != hStream)
        {
            // Clear the buffer queue.
            nRet = MV_FG_FlushQueue(hStream, MV_FG_BUFFER_QUEUE_ALL_DISCARD);
            if (MV_FG_SUCCESS != nRet)
```

```
        {
            printf("Flush Buffer Queue error, %#x\n", nRet);
        }

        // Release the registered buffer.
        for (unsigned int i = 0; i < BUFFER_NUMBER; i++)
        {
            if (NULL != hBuffer[i])
            {
                nRet = MV_FG_RevokeBuffer(hStream, hBuffer[i], NULL, NULL);
                if (MV_FG_SUCCESS != nRet)
                {
                    printf("Revoke Buffer failed, %#x\n", nRet);
                }
                hBuffer[i] = NULL;
            }

            if (NULL != pBuffer[i])
            {
                free(pBuffer[i]);
                pBuffer[i] = NULL;
            }
        }

        // Close the streaming channel.
        nRet = MV_FG_CloseStream(hStream);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close Stream error, %#x\n", nRet);
        }
        hStream = NULL;
    }

    // Close the device.
    if (NULL != hDevice)
    {
        nRet = MV_FG_CloseDevice(hDevice);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close device error, %#x\n", nRet);
        }
        hDevice = NULL;
    }

    // Close the frame grabber.
    if (NULL != hInterface)
    {
        nRet = MV_FG_CloseInterface(hInterface);
        if (MV_FG_SUCCESS != nRet)
        {
            printf("Close Interface error, %#x\n", nRet);
```

```
            }
            hInterface = NULL;
        }

        printf("Press any key to exit.\n");
        WaitForKeyPress();

        return 0;
}
```

Appendix B. Error Code

You can search for the detailed error descriptions according to returned error codes or error names.

SDK Error Code

Error Code	Error Name	Description
Normal Status Code		
0x00000000	MV_FG_SUCCESS	Succeeded.
General Error Codes (from 0x80190001 to 0x801900FF)		
0x80190001	MV_FG_ERR_ERROR	Unknown error.
0x80190002	MV_FG_ERR_NOT_INITIALIZED	Not initialized.
0x80190003	MV_FG_ERR_NOT_IMPLEMENTED	Not implemented.
0x80190004	MV_FG_ERR_RESOURCE_IN_USE	The requested resource is in use.
0x80190005	MV_FG_ERR_ACCESS_DENIED	No access permission.
0x80190006	MV_FG_ERR_INVALID_HANDLE	Invalid handle.
0x80190007	MV_FG_ERR_INVALID_ID	Invalid ID.
0x80190008	MV_FG_ERR_NO_DATA	No data.
0x80190009	MV_FG_ERR_INVALID_PARAMETER	Invalid parameter.
0x80190010	MV_FG_ERR_IO	IO error.
0x80190011	MV_FG_ERR_TIMEOUT	Timed out.
0x80190012	MV_FG_ERR_ABORT	Operation interrupted.
0x80190013	MV_FG_ERR_INVALID_BUFFER	Invalid buffer.
0x80190014	MV_FG_ERR_NOT_AVAILABLE	Unreachable.
0x80190015	MV_FG_ERR_INVALID_ADDRESS	Invalid address.
0x80190016	MV_FG_ERR_BUFFER_TOO_SMALL	Buffer too small.
0x80190017	MV_FG_ERR_INVALID_INDEX	Invalid index.
0x80190018	MV_FG_ERR_PARSING_CHUNK_DATA	Failed to parse chunk.
0x80190019	MV_FG_ERR_INVALID_VALUE	Invalid value.
0x80190020	MV_FG_ERR_RESOURCE_EXHAUSTED	Resource exhausted.
0x80190021	MV_FG_ERR_OUT_OF_MEMORY	Failed to request memory.
0x80190022	MV_FG_ERR_BUSY	Busy.
0x80190023	MV_FG_ERR_LOADLIBRARY	Failed to load dynamic link library.
0x80190024	MV_FG_ERR_CALLORDER	Incorrect order of API calls.

Error Code	Error Name	Description
GenICam Related Error Codes (from 0x80190100 to 0x801901FF)		
0x80190100	MV_FG_ERR_GC_GENERIC	Generic error.
0x80190101	MV_FG_ERR_GC_ARGUMENT	Parameter error.
0x80190102	MV_FG_ERR_GC_RANGE	The value is out of range.
0x80190103	MV_FG_ERR_GC_PROPERTY	Attribute error.
0x80190104	MV_FG_ERR_GC_RUNTIME	Running environment error.
0x80190105	MV_FG_ERR_GC_LOGICAL	Logic error.
0x80190106	MV_FG_ERR_GC_ACCESS	Access permission error.
0x80190107	MV_FG_ERR_GC_TIMEOUT	Timed out.
0x80190108	MV_FG_ERR_GC_DYNAMICCAST	Conversion exception.
0x801901FF	MV_FG_ERR_GC_UNKNOW	GenICam unknown error.
Image Processing Related Error Codes (from 0x80190200 to 0x801902FF)		
0x80190200	MV_FG_ERR_IMG_HANDLE	Handle error.
0x80190201	MV_FG_ERR_IMG_SUPPORT	Not supported.
0x80190202	MV_FG_ERR_IMG_PARAMETER	Parameter error.
0x80190203	MV_FG_ERR_IMG_OVERFLOW	Out of memory.
0x80190204	MV_FG_ERR_IMG_INITIALIZED	Operation not initialized.
0x80190205	MV_FG_ERR_IMG_RESOURCE	Failed to release resource.
0x80190206	MV_FG_ERR_IMG_ENCRYPT	Image encryption error.
0x80190207	MV_FG_ERR_IMG_FORMAT	Incorrect image format or image format not supported.
0x80190208	MV_FG_ERR_IMG_SIZE	Incorrect image width/height or image width/height out of range.
0x80190209	MV_FG_ERR_IMG_STEP	The value of image width/height and step parameter mismatch.
0x80190210	MV_FG_ERR_IMG_DATA_NULL	The address for storing the image data is empty.
0x80190211	MV_FG_ERR_IMG_ABILITY_ARG	The image algorithm capability contains invalid parameters.
0x801902FF	MV_FG_ERR_IMG_UNKNOW	Unknown error occurred during image processing.

GenTL Error Code

Error Code	Error Name	Description
-1001	GC_ERR_ERROR	Unknown error.
-1002	GC_ERR_NOT_INITIALIZED	Module or resource is not initialized.
-1003	GC_ERR_NOT_IMPLEMENTED	The requested operation is not implemented.
-1004	GC_ERR_RESOURCE_IN_USE	The requested resource is already occupied.
-1005	GC_ERR_ACCESS_DENIED	The requested resource is not allowed.
-1006	GC_ERR_INVALID_HANDLE	The given handle does not support the operation.
-1007	GC_ERR_INVALID_ID	Failed to connect to the specified resource by the invalid ID.
-1008	GC_ERR_NO_DATA	The function has no data to be processed.
-1009	GC_ERR_INVALID_PARAMETER	Invalid given parameter(s).
-1010	GC_ERR_IO	I/O communication error.
-1011	GC_ERR_TIMEOUT	Operation timed out.
-1012	GC_ERR_ABORT	The operation is aborted before being completed.
-1013	GC_ERR_INVALID_BUFFER	The registered buffer is not enough to acquire images in the current acquisition mode.
-1014	GC_ERR_NOT_AVAILABLE	Resource or information is not available within the given time in the current status.
-1015	GC_ERR_INVALID_ADDRESS	The given address is invalid or out of range due to internal reasons.
-1016	GC_ERR_BUFFER_TOO_SMALL	The provided buffer is too small to receive the expected amount of data.
-1017	GC_ERR_INVALID_INDEX	The index is out of range.
-1018	GC_ERR_PARSING_CHUNK_DATA	An error occurred when parsing a buffer containing chunk data.
-1019	GC_ERR_INVALID_VALUE	Trying to write an invalid value to the register.
-1020	GC_ERR_RESOURCE_EXHAUSTED	The requested resource is exhausted.
-1021	GC_ERR_OUT_OF_MEMORY	No enough memory for the system or hardware of the system.

Error Code	Error Name	Description
-1022	GC_ERR_BUSY	Failed to perform the requested operation. The current module or instance is busy.