Simply Brighter

(In Canada)
111 Railside Road
Suite 201
Toronto, ON M3A 1B2
CANADA
Tel: 1-416-840 4991
Fax: 1-416-840 6541

(In US)
1241 Quarry Lane
Suite 105
Pleasanton, CA 94566
USA
Tel: 1-925-218 1885
Email: sales@mightex.com

Mightex S-Series USB Camera SDK Manual

Version 1.0.4
Jan.2, 2019

Relevant Products

Part Numbers
SCN-BG04-U, SCE-BG04-U, SCN-CG04-U, SCE-CG04-U, SCN-B013-U, SCE-B013-U, SCN-C013-U, SCE-C013-U, SCN-C030-U, SCE-C030-U

Revision History

Revision	Date	Author	Description
1.0.0	Aug. 10, 2010	JT Zheng	Initial Revision
1.0.1	Jul. 28, 2011	JT Zheng	100us Minimum ET for BG04/CG04 in TRIGGER mode
1.0.2	Jan. 31, 2012	JT Zheng	Adding “xxx_DS.dll” description
1.0.3	May 16, 2012	JT Zheng	Adding SetGainRatios() API
1.0.4	Jan.2,2019	JT Zheng	New Mightex Logo

Mightex USB 2.0 SCX camera is mainly designed for low cost machine vision applications, With high speed USB 2.0 interface and powerful PC camera engine, the camera delivers CMOS image data at high frame rate. GUI demonstration application and SDK are provided for user's application developments.

IMPORTANT:

Mightex USB Camera is using USB 2.0 for data collection, USB 2.0 hardware MUST be present on user's PC and Mightex device driver MUST be installed properly before using Mightex demonstration application OR developing application with Mightex SDK. **For installation of Mightex device driver, please refer to *Mightex New Classic USB Camera User Manual.***

SDK FILES:

The SDK includes the following files:

\LIB directory:

NewClassic NewClassic_USBCamera_SDK.h --- Header files for all data prototypes and dll export functions.
NewClassic NewClassic_USBCamera_SDK.dll--- DLL file exports functions.
NewClassic NewClassic_USBCamera_SDK.lib--- Import lib file, user may use it for VC++ development.
NewClassicUsblib.dll --- DLL file used by "NewClassic_USBCamera_SDK.dll" .

\Documents directory:

Mightex SCX Camera SDK Manual.pdf

\Examples directory

\Delphi --- Delphi project example.
\VC++ --- VC++ 6.0 project example.
\VB_Application --- Contain "Stdcall" version of "NewClassic_USBCamera_SDK.dll", named as "NewClassic _USBCamera_SDK_Stdcall.dll" which is used for VB developers.
…more samples will be added under this sub-directory.

Note

1). Although the SCX Camera engine supports multiple cameras, the camera is mainly designed for grabbing frame from one camera at a certain moment. For application in which user needs to get frame from multiple cameras simultaneously, user might use Mightex Buffer camera.
When there're multiple cameras, user may invoke functions to get the number of cameras currently present on USB Bus and add the subset of the cameras into current "Working Set", all the cameras in "Working Set" are active cameras from which camera engine can grab frames. Up to 8 cameras (software limit) can be supported by the camera engine, and all of them can be added in working set although it's recommended to have one activated camera at a certain moment for most of the applications. These cameras might be different types (e.g. 1.3M Mono, 1.3M Color or 3M Color). While there're more than one camera in working set, the Camera engine has to grab frame from cameras one by one and the USB data bandwidth has to be shared among those cameras and the frame rate is reduced.
2). Although cameras are USB Devices, camera engine itself is not designed as full Plug&Play, it's NOT recommended to Plug or Unplug cameras while the camera engine is grabbing frames from the cameras.
3). The code examples are for demonstration of the DLL functions only, device fault conditions are not fully handled in these examples, user should handle those error conditions properly.

HEADER FILE:

The "NewClassic_USBCamera_SDK.h" is as following:

```
typedef int SDK_RETURN_CODE;
typedef unsigned int DEV_HANDLE;

#ifdef SDK_EXPORTS
#define SDK_API extern "C" __declspec(dllexport) SDK_RETURN_CODE _cdecl
#define SDK_HANDLE_API extern "C" __declspec(dllexport) DEV_HANDLE _cdecl
#else
#define SDK_API extern "C" __declspec(dllimport) SDK_RETURN_CODE _cdecl
#define SDK_HANDLE_API extern "C" __declspec(dllimport) DEV_HANDLE _cdecl
#endif
```

```
#define GRAB_FRAME_FOREVER                    0x8888

typedef struct {
    int CameraID;
    int Row;
    int Column;
    int Bin;
    int XStart;
    int YStart;
    int ExposureTime;
    int RedGain;
    int GreenGain;
    int BlueGain;
    int TimeStamp;
    int TriggerOccurred;
    int TriggerEventCount;
     int  ProcessFrameType；
} TProcessedDataProperty;

typedef void (* DeviceFaultCallBack)( int DeviceType );
typedef void (* FrameDataCallBack)( TProcessedDataProperty* Attributes, unsigned char *BytePtr );

// Import functions:
SDK_API NewClassicUSB_InitDevice( void );
SDK_API NewClassicUSB_UnInitDevice( void );
SDK_API NewClassicUSB_GetModuleNoSerialNo( int DeviceID, char *ModuleNo, char *SerialNo);
SDK_API NewClassicUSB_AddDeviceToWorkingSet( int DeviceID );
SDK_API NewClassicUSB_RemoveDeviceFromWorkingSet( int DeviceID );
SDK_API NewClassicUSB_ActiveDeviceInWorkingSet( int DeviceID, int Active );
SDK_API NewClassicUSB_StartCameraEngine( HWND ParentHandle, int CameraBitOption  );
SDK_API NewClassicUSB_StopCameraEngine( void );
SDK_API NewClassicUSB_SetCameraWorkMode( int DeviceID, int WorkMode );
SDK_API NewClassicUSB_StartFrameGrab( int TotalFrames );
SDK_API NewClassicUSB_StopFrameGrab( void );
SDK_API NewClassicUSB_ShowFactoryControlPanel( int DeviceID, char *passWord );
SDK_API NewClassicUSB_HideFactoryControlPanel( void );
SDK_API NewClassicUSB_SetCustomizedResolution( int deviceID, int RowSize, int ColSize, int Bin );
SDK_API NewClassicUSB_SetExposureTime( int DeviceID, int exposureTime );
SDK_API NewClassicUSB_SetXYStart( int deviceID, int XStart, int YStart );
SDK_API NewClassicUSB_SetGains( int deviceID, int RedGain, int GreenGain, int BlueGain );
SDK_API NewClassicUSB_SetGainRatios( int deviceID, int RedGainRatio, int BlueGainRatio);
SDK_API NewClassicUSB_SetGamma( int Gamma, int Contrast, int Bright, int Sharp );
SDK_API NewClassicUSB_SetBWMode( int BWMode, int H_Mirror, int V_Flip );
SDK_API  NewClassicUSB_SetMinimumFrameDelay( int IsMinimumFrameDelay);
SDK_API NewClassicUSB_SoftTrigger( int DeviceID );
SDK_API NewClassicUSB_SetSensorFrequency( int DeviceID, int sensorFrequency);
SDK_API NewClassicUSB_SetHBlankingExtension( int DeviceID, int; hblankingExtension);
SDK_API NewClassicUSB_InstallFrameHooker( int FrameType, FrameDataCallBack FrameHooker );
SDK_API NewClassicUSB_InstallUSBDeviceHooker( DeviceFaultCallBack USBDeviceHooker );
SDK_POINTER_API NewClassicUSB_GetCurrentFrame( int FrameType, int deviceID, byte* &FramePtr );
SDK_API NewClassicUSB_GetDevicesErrorState();
SDK_API NewClassicUSB_SetGPIOConfig( int DeviceID, unsigned char ConfigByte );
SDK_API NewClassicUSB_SetGPIOInOut( int DeviceID, unsigned char OutputByte,
                    unsigned char *InputBytePtr );
```

// **Please check the header file itself of latest information, we may add functions from time to time.**

Basically, only ONE data structure *TProcessedDataProperty* is defined and used for the all following functions. Note that “#pragma (1)” should be used (as above) for the definition of this structure, as DLL expects the variable of this data structure is “BYTE” alignment.

EXPORT Functions:

NewClassic_USBCamera_SDK.dll exports functions to allow user to easily and completely control the camera and get image frame. The factory features such as firmware version queries, firmware upgrade…etc. are provided by a built-in windows, User may simply invoke **NewClassicUSB_ShowFactoryControlPanel()** and **NewClassicUSB_HideFactoryControlPanel()** to show and hide this window in user's application.

SDK_API NewClassicUSB_InitDevice(void);

This is first function user should invoke for his own application, this function communicates with the installed device driver and reserves resources for all further operations.
Arguments: None
Return: The number of Mightex SCX cameras (SCN and SCE) currently attached to the USB 2.0 Bus, if there's no Mightex SCX USB camera attached, the return value is 0.
Note: There's **NO** device handle needed for calling further SDK APIs, after invoking NewClassicUSB_InitDevice, camera engine reserves resources for all the attached SCX cameras. For example, if the returned value is 2, which means there TWO cameras currently presented on USB, user may use "1" or "2" as DeviceID to call further device related functions, "1" means the first device and "2" is the second device (Note it's ONE based). By default, all the devices are in "inactive" state, user should invoke *NewClassicUSB_AddCameraToWorkingSet(deviceID)* to set the camera as active.
Important: The device drivers and camera engines are different for SCX cameras, BCN/BCE/BTN/BTE camera and Mightex MCN/MCE/GLN/MLE cameras, BXX and MCN/MCE/GLN/MLE camera will **not** be handled by the SCX camera engine, so they will not present in the return number. *Another point is that SCX camera engine does NOT run with BXX/MXX camera engine con-currently (due to the resource conflict), although they can be installed on the same PC without any problem…but only one of camera engines should be run at a certain time.*

For properly operating the cameras, usually the application should have the following sequence for device initialization and opening:

```
NewClassicUSB_InitDevice();     // Get the devices
NewClassicUSB_AddCameraToWorkingSet( deviceID); // Adding cameras to "working set".
NewClassicUSB _StartCameraEngine();
…… Operations ……
```

When application terminates, it usually does:

```
NewClassicUSB _StopCameraEngin();
NewClassicUSB _UnInitDevice()
```

SDK_API NewClassicUSB_UnInitDevice(void);

This is the function to release all the resources reserved by NewClassicUSB_InitDevice(), user should invoke it before application terminates.
Arguments: None
Return: Always return 0.

SDK_API NewClassicUSB_GetModuleNoSerialNo(int DeviceID, char *ModuleNo, char *SerialNo);

For any present device, user might get its Module Number and Serial Number by invoking this function.
Argument: DeviceID – the number (ONE based) of the device, Please refer to the notes of **NewClassicUSB_InitDevice()** function for it.
ModuleNo – the pointer to a character buffer, the buffer should be available for at least 16 characters.
SerialNo – the pointer to a character buffer, the buffer should be available for at least 16 characters.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Important: Usually, user should use this API to get Module/Serial No of a camera after invoking **NewClassicUSB_InitDevice().**

SDK_API NewClassicUSB_AddDeviceToWorkingSet(int DeviceID);

For a present device, user might add it to current "Working Set" of the camera engine, and the camera becomes active.
Argument: DeviceID – the number (ONE based) of the device, Please refer to the notes of **NewClassicUSB_InitDevice()** function for it.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Note: Camera Engine will only grab frames from active cameras in "Working Set".
Important: When user adds a device in working set by invoking **NewClassicUSB_AddDeviceToWorkingSet(int DeviceID),** the camera is activated by default. This should be done before the camera engine starts, after camera engine is started, user should use **NewClassicUSB_ActiveDeviceInWorkingSet**() to de-activate or re-activate a device in working set.

SDK_API NewClassicUSB_RemoveDeviceFromWorkingSet(int DeviceID);

User might remove the camera from the current "Working Set", after invoking this function, the camera become inactive.
Argument: DeviceID – the number (ONE based) of the device, Please refer to the notes of **NewClassicUSB_InitDevice()** function for it.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Note: Camera Engine will only grab frames from active cameras (the cameras in "Working Set").
Important: User can remove a device from working set by invoking **NewClassicUSB_AddDeviceToWorkingSet(int DeviceID), t**his should be done before the camera engine is started, after camera engine starts, user should use NewClassicUSB_ActiveDeviceInWorkingSet() to de-activate or re-activate a device in working set.

SDK_API NewClassicUSB_ActiveDeviceInWorkingSet(int DeviceID, int Active);

User might temporarily de-activate a camera in the working set and later re-activate it.
Argument: DeviceID – the number (ONE based) of the device, Please refer to the notes of **NewClassicUSB_InitDevice()** function for it.
Active: 1 – the camera is activated in working set, 0 – the camera is de-activated in working set.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Note: Camera Engine will only grab frames from active cameras in "Working Set".
Important: When user adds a device in working set by invoking **NewClassicUSB_AddDeviceToWorkingSet(int DeviceID),** the camera is activated by default, after that, user might use this function to de-activate or re-activate it later.
(User should add/remove cameras in working set before camera engine starts, after the camera engine is started, user should use this API for active/de-active cameras in working set). For SCX camera, we recommend only one camera is activated (e.g. user might add multiple cameras in Working Set, but de-activate them but leaving one active only.) at a certain moment for maximum frame rate.

SDK_API NewClassicUSB_StartCameraEngine(HWND ParentHandle, int CameraBitOption) ;

There's a multiple threads camera engine, which is responsible for all the frame grabbing, internal queue managements, raw data processing…etc. functions. User MUST start this engine for all the following camera related operations
Argument: ParentHandle – The window handle of the main form of user's application, as the engine relies on Windows Message Queue, it needs a parent window handle which mostly should be the handle of the main window of user's application. If user's application doesn't have parent window, user might use NULL for this argument. For console applications, user should use NULL for this argument too. (please refer to the application notes later.)
CameraBitOption – It must be 8 for all SCX cameras.
Return: -1 If the function fails (e.g. There's no camera added in working set)
1 if the call succeeds.

Important: ***It's* NOT allowed *to Add Camera to WorkingSet OR Remove Camera from "Working Set" while the camera engine has started already.. we expect user to arrange camera working set properly and then start the camera engine.***

SDK_API NewClassicUSB_StopCameraEngine(void);

This function stops the started camera engine.
Argument: None.
Return: it always returns 1.
Important: Users don't need to explicitly open and close the opened device, as NewClassicUSB_InitDevice() will actually open all the current attached cameras and reserve resources for them, however, by default, all of them are inactive, user needs to set them as active by invoking *NewClassicUSB_AddCameraToWorkingSet(deviceID).*
It's* NOT allowed *to Add Camera to WorkingSet OR Remove Camera from WorkingSet while the camera engine is started already.. we expect user to arrange camera working set properly and then start the camera engine.

SDK_API NewClassicUSB_SetCameraWorkMode(int DeviceID, int WorkMode);

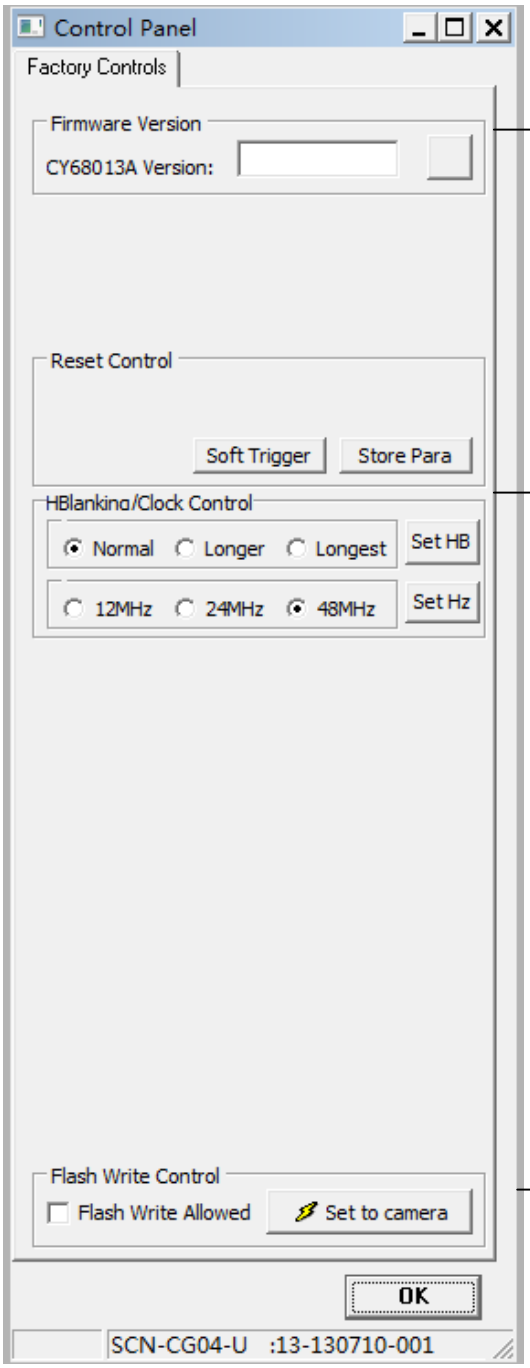
By default, the Camera is working in "**NORMAL"** mode in which camera deliver frames to Host continuously, however, in some applications, user may set it to **"TRIGGER"** Mode, in which the camera is waiting for an external trigger signal and capture ONE frame for each trigger signal.
Argument: *DeviceID* – the device number which identifies the camera.
WorkMode – 0: **NORMAL** Mode, 1: **TRIGGER** Mode.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Important:
NORMAL mode and **TRIGGER** mode have the same features, but:
NORMAL mode – Camera will always grab frame as long as Host asks for it.
TRIGGER mode – Camera will only grab a frame while there's an external trigger asserted (and Host asks for a frame), in TRIGGER mode, host actually polls for the trigger state of the camera and grab a frame while the trigger state is ON (there's an external trigger assertion).

SDK_API NewClassicUSB_StartFrameGrab(int TotalFrames);
SDK_API NewClassicUSB_StopFrameGrab(void);

When camera engine is started, the engine prepares all the resources, but it does **NOT** start the frame grabbing yet, until **NewClassicUSB_StartFrameGrab()** function is invoked. After it's successfully called, camera engine starts to grab frames from cameras in current "Working Set". User may call **NewClassicUSB_StopFrameGrab()** to stop the engine from grabbing frames from camera.
Argument: TotalFrames – This is for **NewClassicUSB_StartFrameGrab()** only, after grabbing frames of this number, the camera engine will automatically stop grabbing, if user doesn't want it to be stopped, set this number to 0x8888, this means to grab frame forever (in **NORMAL** mode), until user calls NewClassicUSB_StopFrameGrab().
Return: -1 If the function fails (e.g. invalid device number or if the engine is NOT started yet)
1 if the call succeeds.

SDK_API NewClassicUSB_ShowFactoryControlPanel(int DeviceID, char *passWord);

For user to access the factory features conveniently and easily, the library provides its second dialog window which has all the features on it.
Argument: DeviceID – the device number.
Password – A pointer to a string which is the password, "661016" is recommended in most cases.
Return: -1 If the function fails (e.g. invalid device number)
1 If the call succeeds.



Click to button will get firmware version information from camera.

User can select sensor clock and HBlanking control here, for most applications, it's recommended to use the default setting. Slower clock or Longer HBlanking might deliver better SNR image with lower frame rate.
For BG04/CG04 cameras, the SNR will be degraded when selecting 48MHz, however, the frame rate will go **~65fps** under this setting.

User can enable the firmware upgrading feature here, and then user might use the Mightex firmware upgrading tools (another software) to upgrade the firmware of the camera.

SDK_API NewClassicUSB_HideFactoryControlPanel(void);

This function hides the factory control panel, which is shown by invoking
NewClassicUSB_ShowFactoryControlPanel().
Argument: None.
Return: it always returns 1.

SDK_API NewClassicUSB_SetCustomizedResolution(int deviceID, int RowSize, int ColumnSize, int Bin)

User may set the customized Row/Column size by invoking this function.
Argument: DeviceNo – the device number which identifies the camera will be operated.
RowSize – the customer defined row size.
ColumnSize – the customer defined column size.
Bin – 0: Normal mode, 1: 1:2 decimation mode
Return: -1 : If the function fails (e.g. invalid device number)
-2 : The customer defined row or column size is out of range. The valid range for row/column sizes are:
Actual RowSize: 4 – Maximum Row Size (camera dependant, for 1.3M and 3M camera, it's 1280 and 2048)
Actual ColumnSize: 4 – Maximum Column Size (camera dependant, for 1.3M and 3M camera, it's 1024 and 1536)
Note that while in Bin mode (Bin is "1"), the minimum of RowSize and Column Size should be 8, while Bin is "0", RowSize and ColumnSize can be 4.
-3 : The customer defined row/column size must be a number of times of 4.
1 if the call succeeds.
An example:
NewClassicUSB_SetCustomizedResolution(deviceNO, 1280, 1024, 1);
Set the camera (deviceNo) to 1280x1024, 1:2 decimation enabled.

SDK_API NewClassicUSB_SetExposureTime(int DeviceID, int exposureTime);

User may set the exposure time by invoking this function.
Argument: DeviceNo – the device number which identifies the camera will be operated.
exposureTime – the Exposure Time is set in 50 Microsecond UNIT, e.g. if it's 4, the exposure time of the camera will be set to 200us. The valid range of exposure time is 50us – 750ms, So the exposure time value here is from 1 – 15000.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Note: For BG04/CG04, when the camera is in TRIGGER mode, the minimum ET the camera accepts is 100us, thus the range is 0.1 – 750ms for BG04/CG04 cameras when it's in TRIGGER mode.

SDK_API NewClassicUSB_SetXYStart(int deviceID, int XStart, int YStart);

User may set the X, Y Start position (at a certain resolution) by invoking this function.
Argument: DeviceNo – the device number which identifies the camera will be operated.
Xstart, YStart – the start position of the ROI (in pixel).
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Important: While the resolution is NOT set to the Maximum, the setting Region Of Interesting (ROI) is an active region of the full sensor pixels, user may use this function to set the left top point of the ROI. Note that under a certain ROI, the Xstart and Ystart have a valid settable range. If any value out of this range is set to device, device firmware will check it and set the closest valid value.

SDK_API NewClassicUSB_SetGains(int deviceID, int RedGain, int GreenGain, int BlueGain);

User may set the Red, Green and Blue pixels' gain of a certain camera by invoking this function.
Argument: DeviceNo – the device number which identifies the camera will be operated.
RedGain, GreenGain, BlueGain – the gain value to be set.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Important: The gain value should be from 1 – 64, represents 0.125x – 8x of analog amplifying Multiples. For any value of the valid range, device will handle it by setting the close valid value. For Monochrome sensor modules, only GreenGain value will be used by the camera firmware. For the BG/CG04 modules (with MT9V032 sensor), only one Global Gain is used for all pixels (even for color sensor with RGB pixels) and only value in 8 – 32 range is used (Gain from 1x – 4x), for any value less than 8, the firmware will take it as 8 (1x), for any value more than 32, the firmware will take it as 32 (4x).
Note: For setting proper exposure for an image, it's recommended to adjust exposure time prior to the gain, as setting high gain will increase the noise (Gain is similar to the ISO settings in a consumer camera), for applications which the SNR is important, it's recommended to set Gain not more than 32 (4x), for BG04/CG04 camera, it's recommended to set Gain not more than 16 (2x).

SDK_API NewClassicUSB_SetGainRatios(int deviceID, int RedGainRatio, int BlueGainRatio);

User may set the Red/Green and Blue/Green digital Ratio by invoking this function. Note it's for Color camera with only one global gain (CG04) only.
Argument: DeviceNo – the device number which identifies the camera will be operated.
RedGainRatio, BlueGainRatio – the gain adjust value to Red and Blue pixels.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succedds.
Important: On some types of Color CMOS sensor, there's only one global hardware gain for all R, G, B pixels (similar to the Monochrome camera), so when user invokes the above **NewClassicUSB_SetGains(…)** function, it applies to all R, G, B gains. For user wants to set different gains for Blue and Red pixels (e.g. for white balance), user should use this API. The RedGainRatio and BlueGainRatio are percentage adjustments to the global gain on the Red and Blue pixels, it can be from 1 – 200 (1% to 200%), for example, if RedGainRatio is 95, it means the final gain for Red pixel is 95% of the global gain. Note that when user invokes this API multiple times, the ratio will be applied to the previous setting ratio, e.g. user invokes this API with ratio set to 90 (90%) for the first time, and then 80 (80%) for the second time, the actual final ratio is 72 (72%).

SDK_API NewClassicUSB_SetGamma(int Gamma, int Contrast, int Bright, int Sharp);

User may set the Gamma, Contrast, Brightness and Sharpness level for **ALL** Cameras (in "working set") by invoking this function.
Argument: Gamma – Gamma value, valid range 1 – 20, represent 0.1 – 2.0 gamma value.
Contrast – Contrast value, valid range 0 – 100, represent 0% -- 100%.
Bright – Brightness value, valid range 0 – 100, represent 0% -- 100%.
Sharp – Sharp Level, valid range 0 – 3, 0: No Sharp, 1: Sharp, 2: Sharper, 3: Sharpest.
Return: -1 If the function fails,
1 if the call succeeds.
Important: For application needs BMP data (rather than RAW data) from camera engine, user can install frame hooker with the FrameType set to "1", in this case, camera engine will process the image frames (from **ALL** cameras in "working set") with "de-mosaic" algorithm, and the resulting Bitmap data can be got via the installed callback function. User might use this function to change the parameters of the algorithm in data processing, note that in most cases, the default values (Gamma = 1.0, Contrast and Brightness = 50%, Sharp = 0) are proper.

SDK_API NewClassicUSB_SetBWMode(int BWMode, int H_Mirror, int V_Flip);

User may set the processed Bitmap data as "Black and White" mode, "Horizontal Mirror" and "Vertical Flip" by invoking this function.

Argument: BWMoe : 0: Normal, 1: Black and White mode.
H_Mirror: 0: Normal, 1: Horizontal Mirror.
V_Flip: 0: Normal, 1: Vertical Flip.
Return: -1 If the function fails.
1 if the call succeeds.
Important: For application needs BMP data (rather than RAW data) from camera engine, user can install frame hooker with the FrameType set to "1", in this case, camera engine will process the image frames (from **ALL** cameras in "working set") with "de-mosaic" algorithm, and the resulting Bitmap data can be got via the installed callback function. User may use this function to change some attributes of the Bitmap image. Note that the BWMode is only applicable to Color sensor cameras.

SDK_API NewClassicUSB_SetMinimumFrameDelay(int IsMinimumFrameDelay);

User may set minimum frame delay with this API.
Argument: IsMinimumFrameDelay – 0: Disable Minimum Frame Delay,
1:Enable Minimum Frame Delay.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succedds.

SDK_API NewClassicUSB_SoftTrigger(int DeviceID);

User may user this API to trigger the camera while the camera is in TRIGGER mode.
Argument: DeviceNo – the device number which identifies the camera will be operated.
Return: -1 If the function fails. (e.g. DeviceID is out of range)
1 if the call succeeds.
Note: User can use this API to simulate a trigger while the camera is in TRIGGER mode.

SDK_API NewClassicUSB_SetSensorFrequency(int DeviceID, int sensorFrequency);

User may user this API to set the sensor clock.
Argument: DeviceNo – the device number which identifies the camera will be operated.
sensorFrequency – 0: Slow clock, 1: Normal clock, 2: Fast clock.
Return: -1 If the function fails. (e.g. DeviceID is out of range)
1 if the call succeeds.
Note: The default setting of the camera is "Fast" clock for B013/C030 cameras, and "Normal" clock for BG04/CG04 cameras. For BG04/CG04 cameras, when it's set to "Fast" clock, the frame rate can go ~65fps, however, the SNR is degraded.

SDK_API NewClassicUSB_SetHBlankingExtension(int DeviceID, int; hblankingExtension);

User may user this API to set the HBlanking of the sensor.
Argument: DeviceNo – the device number which identifies the camera will be operated.
hblankingExtension – 0 : Normal HBlanking, 1: Longer HBlanking, 2: Longest HBlanking
Return: -1 If the function fails. (e.g. DeviceID is out of range)
1 if the call succeeds.
Note: The default setting of the camera is Normal HBlanking.

SDK_API NewClassicUSB_InstallFrameHooker(int FrameType, FrameDataCallBack FrameHooker);
Argument: FrameType – 0: Raw Data **(RAW mode)**
1: Processed Data. **(BMP mode)**
FrameHooker – Callback function installed.
Return: -1 If the function fails (e.g. invalid Frame Type).
1 if the call succeeds.
Important: The call back function will only be invoked while the frame grabbing is started, host will be notified every time the camera engine get a new frame (from any cameras in current working set).
Note:
The callback has the following prototype:
typedef void (* FrameDataCallBack)(TProcessedDataProperty* Attributes, unsigned char *BytePtr);

The TProcessedDataPropertys defined as:

```
typedef struct {
   int CameraID;
   int Row;
   int Column;
   int Bin;
   int XStart;
   int YStart;
   int ExposureTime;
   int RedGain;
   int GreenGain;
   int BlueGain;
   int TimeStamp;
   int TriggerOccurred;
   int TriggerEventCount;
   int ProcessFrameType;
} TProcessedDataProperty;
```

Arguments of Call Back function:
Attributes – This is an important data structure which contains information of this particular frame, camera firmware fill this data structure while camera finishes the grabbing of a frame, with the real time parameters used for this frame. It has the following elements:

CameraID – This is the camera number (the same as the deviceID used in all the APIs), as camera engine might get frames from more than one cameras (there might be multiple cameras in current working set), this identifies which camera in working set generates the frame.

Row, Column – The Row Number and Column Number of this frame, this is also the resolution. Note that the actual size of the image is also depending on the "Bin". And note that Row and Column here are Row Number and Column Number, for example, in the resolution of 1280x1024, the Row Number is 1024, and the Column Number is 1280.

Bin – Whether decimation mode is enabled for this frame, if it's 0, it's NOT enabled, and it's enabled if it might be 1 (means 1:2 mode The actual size of the Row and Column should be as following:

Actual_Row = Row / DecimationFactor[Bin];
Actual_Column = Column/ DecimationFactor[Bin];
Where DecimationFactor[2] = (1, 2);

Xstart, Ystart – the actual (X,Y) start position of ROI of this frame.

Exposure Time – The exposure time camera was used for generating this frame, Note that it's in 50us unit, a value of "2" means 100us.

RedGain, GreenGain, BlueGain – The Gain Values for this frame, might be from 1 to 64.

TimeStamp – Camera firmware will mark each frame with a time stamp, this is a number from 0 – 65535 ms(and it's automatically round back) which is generated by the internal timer of the firmware, the unit of it is 1ms. For example, if one Frame's time stamp is 100 and the next frame's stamp is 120, the time interval between them is 200x100us = 20ms.

TriggerOccurred – Reserved.

TriggerEventCount – Reserved.

FrameType – the Frame type of this frame, this is the same as the value in invoking **InstallFrameHooker(FrameType, FrameHooker),** it can be either RAW (0) or BMP (1).

BytePtr – The pointer to the memory buffer which holds the frame data.
Please note that the data format is different in RAW and BMP mode.
RAW mode – While user installs frame hooker for RAW data, the BytePtr points to an array as following:

Unsigned char Pixels[Actual_Row][Actual_Column];

For color camera, a Bayer RGB filter is on top of the sensor, so the RAW data is as following (in the following example, it's 2048x1536 for C030 camera):

Pixels[0][0 ...2047] (First Row) G R G R ..... G R
Pixels[1][0 ...2047] (Second Row) B G B G ..... B G

....

Pixels[1534][0 ...2047] G R G R ..... G R
Pixels[1535][0 ...2047] (Last Row) B G B G ..... B G

Note that Acutal_Row, Actual_Column can be figure out according to Row, Column and Bin value as described above.

BMP mode – While user installs frame hooker for BMP data, camera engine will process the raw data from the sensor and generate BMP (note that this is actually only TRUE for color camera, for Monochrome camera, the data returned in BMP mode is the same as RAW mode), the BMP mode data is as following:

****. Monochrome camera***

The BytePtr points to

Unsigned char Data[Actual_Row][Actual_Column];

**. Color camera*

The BytePtr points to

Unsigned char Data[Actual_Row][Actual_Column][3];

Note:

Data[][][0] – 8bit value for **Blue** *at this pixel*
Data[][][1] – 8bit value for **Green** *at this pixel*
Data[][][2] – 8bit value for **Red** *at this pixel*

Camera engine has built-in interpolation algorithm to generate B, G and R value from the Raw Pixels[][] data.

Note that this callback function is invoked in the main thread (usually the GUI thread) of the host application, camera engine relies on Windows message loop to invoke the callback function. GUI operations are allowed in this callback function. However, blocking this callback will slow down the camera engine and thus is not recommended.
While installing the hooker, the FrameType is very important, the camera engine will bypass the complicated BMP generation algorithm if the FrameType is **RAW** mode, this will speed up the processing and usually get higher frame rate in most cases (depending on Host resources), this is especially true for color sensors.

Important: As the frame callback is invoked by a windows timer (which relies on WM_TIMER message) in camera engine, it's important to let windows message loop running (thus the WM_TIMER message is processed), for some console applications, user should pay attention to that, usually user can insert the following code somewhere in his main loop:

```
if(GetMessage(&msg,NULL,NULL,NULL))
{
    if(msg.message == WM_TIMER)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);   // Callback is actually called here.
    }
}
```

SDK_API NewClassicUSB_InstallUSBDeviceHooker(DeviceFaultCallBack USBDeviceHooker);
User may call this function to install a callback, which will be invoked by camera engine while camera engine encounter camera errors.
Argument: USBDeviceHooker – the callback function registered to camera engine.
Return: It always return 1.
Note:
1). The camera engine doesn't support Plug&Play of the cameras for the current version, while the camera engine is starting work, any plug or unplug of the cameras is **NOT** recommended. If plug or unplug occurs, the camera engine might stop its grabbing and invoke the installed callback function. This notifies the host the occurrence of the device configuration change and it's recommended for host to arrange all the "house clean" works. Host might simply do

house keeping and terminate OR host might let user to re-start the camera engine. (Please refer to the Delphi example code for this)
2). The callback function has the following prototype:
typedef void (DeviceFaultCallBack)(int FaultType);*
The FaultType is always ZERO in current version.
Note: The installed device callback will be invoked if Host doesn't have proper resources for grabbing image data from camera, in this case, it's recommended for user to set longer HBlanking time OR set slower clock.

SDK_API NewClassicUSB_SetGPIOConfig(int DeviceID, unsigned char ConfigByte);

User may call this function to configure GPIO pins.
Argument : DeviceID – The device number, which identifies the camera is being operated.
ConfigByte – Only the 4 LSB are used, bit0 is for GPIO1, bit1 is for GPIO2 and so on. Set a certain bit to 0 configure the corresponding GPIO to output, otherwise it's input.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.

SDK_API NewClassicUSB_SetGPIOInOut(int DeviceID, unsigned char OutputByte, unsigned char *InputBytePtr);

User may call this function to set GPIO output pin states and read the input pins states.
Argument : DeviceID – The device number, which identifies the camera is being operated.
OutputByte – Only the 4 LSB are used, bit0 is for GPIO1, bit1 is for GPIO2 and so on. Set a certain bit to 1 will output High on the corresponding GPIO pin, otherwise it outputs Low. Note that it's only working for those pins are configured as "Output".
InputBytePtr – the Address of a byte, which will contain the current Pin States, only the 4 LSB bits are used, note that even a certain pin is configured as "output", we can still get its current state.
Return: -1 If the function fails (e.g. invalid device number)handle or it's camera WITHOUT GPIO)
1 if the call succeeds.

Application Notes:

1). Getting frames from a specified camera when there're multiple cameras

The API NewClassicUSB_StartGrabFrames (frameNumber) will get frames from all the cameras in working set, for SCX cameras, in most applications, it's recommended to get frames from a specified camera only when there're multiple cameras connected.
In some cases, user might want to have a function like "int NewClassicUSB_StartGrabFrameFromCamera(int camId);", this allows user to get frame from the specified camera (identified by "camId").

Although we don't have this exact API from our DLL, but there's an API:

int **NewClassicUSB_ActiveDeviceInWorkingSet**(int DeviceID, int Active);

Note:
a>. While user adds a camera to working Set, it's "Active" by default.
b>. When there're multiple cameras in working set, but user only wants to grab frame from one camera at a certain moment, user might have a helper function first:

```
void ActivateOneCameraOnly( int cameraID )
{
    NewClassicUSB_ActiveDeviceInWorkingSet( cameraID, 1 );  // Activate the specified camera
    for ( i=1; i<MAX_CAMERA; i++)
     {
        if ( i != cameraID)
           NewClassicUSB_ActiveDeviceInWorkingSet( i,0);   // De-activate other cameras.
      }
}
```

c>.So user might implement the mentioned function:

```
int NewClassicUSB_StartGrabFrameFromCamera(int camId)
{
   ActivateOneCameraOnly( camId );
   NewClassicUSB_StartGrabFrames( TotalFrames);
}
```

Note the above codes activate ONE camera in the working set, user should also arrange proper callback for getting the frame data from this camera.

2). NewClassicUSB_AddDeviceToWorkingSet() and NewClassicUSB_ActiveDeviceInWorkingSet()

The first API **NewClassicUSB_AddDeviceToWorkingSet()** is for user to decide whether a camera should be put into the working set **BEFORE** the camera engine is started, after camera engine is started, user should **not** use this API to change the working set, so this is a "Permanent" map of the working set. (Similary, the NewClassicUSB_RemoveDeviceFromWorkingSet() should be used BEFORE the camera engine is started),

Generally, We expect users have the following APIs invoking sequence:

```
// Do USB Devices Initialization
```

```
TotalDevices = NewClassicUSB_InitDevice();
// Identify cameras by Module No. and Serial No.
for (i=1; i<=TotalDevices; i++)
{
    NewClassicUSB_GetModuleNoSerialNo( i, char *ModuleNo, char *SerialNo);
    NewClassicUSB_AddDeviceToWorkingSet(i);
}
NewClassicUSB_StartCameraEngine( );

//…after camera engine is started, user can use NewClassicUSB_ActivateCameraInWorkingSet() to
// activate/de-activate a camera, camera engine only grabs frame from activated camera. Note that for
// SCX cameras, we recommend only ONE camera is activated in the working set for optimized frame
// rate and minimum frame delay.
```

After Camera Engine is started, user might do camera operations to the cameras in working set. By default, while a camera is added to the working set, it's in "Active" state. However, user might temporarily de-activate (and activate later) it in his program. This usually makes sense while there're multiple cameras connected to the Host, and the software doesn't really need grabbing frames from those cameras simultaneously.

For example, If user has multiple cameras connected and added to working set (e.g. 4 cameras), user might set proper parameters (e.g. Working Mode, resolution, exposure time…etc.) to each camera, After that, user might want to get frame a certain camera only, user can active this camera only and after getting several frames from it. User can switch the active camera to another one.

3). For development tools without callback mechanism

There're some tools in which callback mechanism might be a problem to implement, the DLL provide reserved APIs as following:

byte *NewClassicUSB_GetCurrentFrame(int FrameType, int deviceID , byte * &FramePtr);

Note that this API is not functional if user installed the frame callback already, so user might first un-install the callback (pass NULL as the FrameHooker) in this case.
This API is NOT recommended for tools callback mechanism is supported, as this API is much slower than the callback mechanism for getting the frame data (it has more memory copies inside the camera engine).
For the FrameType argument, please refer to the API **NewClassicUSB_InstallFrameHooker(),** and it returns a pointer as the function's return value and its third argument to a block of memory which contains:

```
{
  TprocessedDataProperty  ImageProperty;
  Byte  ImageData[];
}
```

Note that the ImageProperty and ImageData[] are two blocks of memory which contains the Image Property and Image Data, please refer to the callback function:

```
FrameDataCallBack( TProcessedDataProperty* Attributes, unsigned char *BytePtr );
```

For the description of these two fields, user might refer to the "Attributes" and "BytePtr" description in callback function, these two pointers pointed to the same memory blocks as the "ImageProperty" and "ImageData" in this API. This API returns NULL when it fails.

For SCX cameras, the window camera engine (DLL) has multiple threads, in which two of them are important:
a). Camera Control Thread: it grabs frame from camera and setting camera parameters (e.g. exposure time) to camera, the grabbed frame is stored in a “RAW” frame queue.
b). Image Processing Thread: it gets frame from the “RAW” queue and applies bayer-filter algorithm to convert the “RAW” image to “RGB” image, the “RGB” images are stored in “RGB” frame queue.

Thus there’re some frame buffer (Queues) for the frames grabbed from the camera to the delivery image data to user, In most applications, user might want to the following: (instead of continuously grabbing frames)
*. Set Proper Camera Parameters for the following frames (e.g. Exposure Time/Gains…)
*. Start frame grabbing
*. Getting the frame (or frames)
As there might be old frames (in queues) with “Old” camera parameters, it’s always recommended to use frame callback mechanism to get frame data (which might be stored in queue for a while already) and check each frame’s frame property, the frame property has all the camera’s parameters which is used to generated THIS frame, user’s application might have to ignore some “Old” frames after setting new camera parameters.
****

Int NewClassicUSB_GetDevicesErrorState();

Note that this API won’t work if user installed a device callback already, it will return:
-1: If camera engine is NOT started or user installed a device fault callback already
0: No Device error occurred
1: There was devices error occurred

For tools doesn’t support callback mechanism, user might call this API to check if there’s any low level error occurrence in the camera engine.

4). For Console application

User might develop his console application based on the APIs, however, as our camera engine needs window message loop to run, user should let the message loop to be active in his code. Please refer to the following code example:

```
int main(int argc, char**argv)
{
   int ret;
   MSG   msg;

   ret = NewClassicUSB_InitDevice();
   assert(ret==1);
   ret = NewClassicUSB_AddDeviceToWorkingSet(1);
   ret = NewClassicUSB_InstallUSBDeviceHooker( CameraFaultCallBack );
   ret = NewClassicUSB_StartCameraEngine(NULL, 8);
   assert(ret);
   if (-1==NewClassicUSB_SetCustomizedResolution(1,1280,1024, 0))
   {
         StopCamera();
         return FALSE;
   }
   ret = NewClassicUSB_StartFrameGrab(GRAB_FRAME_FOREVER);
   assert(ret==1);
   ret = NewClassicUSB_InstallFrameHooker( 1, FrameCallBack );
   assert(ret==1);

   int quit = 0;
   while (!quit)
   {
       if (!_kbhit())
       {
             Sleep(20);
```

```
            // The following is to let camera engine to be active..it needs message loop.
            if(GetMessage(&msg,NULL,NULL,NULL))
            {
                //if(msg.message   ==   WM_TIMER)
                {
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                }
            }
        }
        else
        {
            char ch;
            ch = _getch();
            switch(toupper(ch))
            {
             case 'Q':
                    quit=1;
                    break;
             default:
                    break;
           }
        }
    }
    StopCamera();
    return 0;
}
```

6). For the callback function to be invoked in a working thread

With the standard DLL, the installed frame callback function is invoked in the main thread (usually the GUI thread) of the application, the advantage of this is that it's much easier for user to handle the data processing and GUI operations inside the frame callback, however, it needs windows message loop to be active in the software (see above console application as an example).

Software needs the frame callback to be invoked in a working thread, Mightex provides another DLL which has the completely same feature but with the following differences:

1) The following two APIs do nothing but return directly: (As those two APIs have GUI operations inside them)

NewClassicUSB_ShowFactoryControlPanel()

NewClassicUSB_HideFactoryControlPanel()

2). All APIs are in "stdcall" convention instead of "cdecl"

3). The installed frame callback and device callback are invoked in a working thread.

This DLL is named as "NewClassic_USBCamera_SDK_DS.dll" and it can be found in CDROM under "\DirectShow\x86\MightexClassicCameraEngine\" for x86 OS, and under "\DirectShow\x64\MightexClassicCameraEngine\" for x64 OS.