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 Buffer USB CCD Camera SDK Manual

Version 1.0.9

Dec. 24, 2018

Relevant Products

Part Numbers
CCN-B013-U, CCE-B013-U,CCN-C013-U, CCE-C013-U, CGN-B013-U, CGE-B013-U,CGN-C013-U, CGE-C013-U, CXN-B013-U, CXE-B013-U,CXN-C013-U, CXE-C013-U, CCN-B020-U, CCE-B020-U, CCN-C020-U, CCE-C020-U

Revision History

Revision	Date	Author	Description
1.0.0	Aug. 28, 2008	JT Zheng	Initial Revision
1.0.1	Oct. 21, 2008	JT Zheng	C-Mount Modals only
1.0.2	May 16, 2009	JT Zheng	Adding CGX Modules
1.0.3	Jun 2, 2009	JT Zheng	Adding P&P supporting
1.0.4	Jan. 16, 2010	JT Zheng	Adding CXX Modules
1.0.5	Apr. 16, 2010	JT Zheng	Red, Blue ratios on frame property
1.0.6	Apr. 26, 2010	JT Zheng	Adding CCX-X020-U Modules
1.0.7	Jan. 31, 2012	JT Zheng	Adding “xx_DS.dll” description, Adding 16bit GetCurrentFrame() description
1.0.8	Aug.6,2013	JT Zheng	Adding “SuspendCamera()” API
1.0.9	Dec. 24,2018	JT Zheng	New Mightex Logo

Mightex USB 2.0 Buffer CCD 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 CCD image data at high frame rate. GUI demonstration application and SDK are provided for user's application developments.

IMPORTANT:

Mightex USB CCD 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 Buffer USB CCD Camera User Manual*.**

SDK FILES:

The SDK includes the following files:

\LIB directory:

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

\Documents directory:

MighTex BUF CCD Camera SDK Manual.pdf

\Examples directory

\Delphi --- Delphi project example.
\VC++ --- VC++ 6.0 project example.
\VB_Application --- Contain "Stdcall" version of "BUF_USBCCDCamera_SDK.dll", named as "Buf_USBCCDCamera_SDK_Stdcall.dll" which is used for VB developers.
\CSharp_Application – C# example, it also must use Stdcall library.

Note

1). The Camera engine supports **simultaneous frame grabbing from 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 grabs frames. Up to 8 cameras (software limit) are supported by the camera engine, and all of them can be added in working set. These cameras might be different types. However, while there're more than one camera in working set, the USB data bandwidth has to be shared among those cameras, so the overall bandwidth for a certain camera is the one-Nth (suppose there're N cameras) of the USB2.0 available bandwidth.
[USB2.0 Bandwidth: For a Bulk EP, the theoretical maximum data rate is ~53Mbytes/s, assume each Micro-frame contains 13 Bulk blocks. However, as there're usually 10 – 20 % of bandwidth will be reserved for Control packages, and there're overheads for Bulk transfer itself, usually, user might get 20M – 40M byte/s data rate on a USB2.0 Bulk EP, and it's very dependant on the USB2.0 Host controller, the drivers and the applications which are currently sharing the USB data bandwidth at a certain moment]
2). 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.
3). Although the camera engine supports P&P, it's not recommended to plug out a working camera from usb port while host is grabbing image from it.

HEADER FILE:

The "BUF_USBCCDCamera_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
#define SDK_POINTER_API extern "C" __declspec(dllexport) unsigned char * _cdecl
```

```
#else
#define SDK_API extern "C" __declspec(dllimport) SDK_RETURN_CODE _cdecl
#define SDK_HANDLE_API extern "C" __declspec(dllimport) DEV_HANDLE _cdecl
#define SDK_POINTER_API extern "C" __declspec(dllimport) unsigned char * _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 UserMark;              /* Reserved */
    int  FrameTime;
    int  CCDFrequency;
    int FrameProcessType;   /* Raw or BMP */
    int FilterAcceptForFile; /* Reserved */
} TProcessedDataProperty;


typedef TImageControl *PImageCtl;

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

// Import functions:
SDK_API BUFCCDUSB_InitDevice( void );
SDK_API BUFCCDUSB_UnInitDevice( void );
SDK_API BUFCCDUSB_GetModuleNoSerialNo( int DeviceID, char *ModuleNo, char *SerialNo);
SDK_API BUFCCDUSB_GetUserSerialNo( int DeviceID,  char *UserSerialNo);
SDK_API BUFCCDUSB_SetUserSerialNo( int DeviceID,  char *UserSerialNo, int IsStoreToNVMemory);
SDK_API BUFCCDUSB_AddDeviceToWorkingSet( int DeviceID );
SDK_API BUFCCDUSB_RemoveDeviceFromWorkingSet( int DeviceID );
SDK_API BUFCCDUSB_ActiveDeviceInWorkingSet( int DeviceID, int Active);
SDK_API BUFCCDUSB_StartCameraEngine( HWND ParentHandle，  int CameraBitOptiont  );
SDK_API BUFCCDUSB_StopCameraEngine( void );
SDK_API BUFCCDUSB_SetCameraWorkMode( int DeviceID, int WorkMode );
SDK_API BUFCCDUSB_StartFrameGrab( int TotalFrames );
SDK_API BUFCCDUSB_StopFrameGrab( void );
SDK_API BUFCCDUSB_ShowFactoryControlPanel( int DeviceID, char *passWord );
SDK_API BUFCCDUSB_HideFactoryControlPanel( void );
SDK_API BUFCCDUSB_SetFrameTime( int deviceID, int  frameTime);
SDK_API BUFCCDUSB_SetCustomizedResolution( int deviceID, int RowSize, int ColSize, int Bin, int BufferCnt );
SDK_API BUFCCDUSB_SetExposureTime( int DeviceID, int exposureTime );
SDK_API BUFCCDUSB_SetXYStart( int deviceID, int XStart, int YStart );
SDK_API BUFCCDUSB_SetGains( int deviceID, int RedGain, int GreenGain, int BlueGain );
SDK_API BUFCCDUSB_SetGainRatios( int deviceID, int RedGainRatio, int; BlueGainRatio);
SDK_API BUFCCDUSB_SetGamma( int Gamma, int Contrast, int Bright, int Sharp );
SDK_API BUFCCDUSB_SetBWMode( int BWMode, int H_Mirror, int V_Flip );
SDK_API BUFCCDUSB_SetSoftTrigger( int deviceID  );
SDK_API BUFCCDUSB_SetCCDFrequency( int deviceID, int Frequency);
```

SDK_API BUFCCDUSB_SetMinimumFrameDelay(int IsMinimumFrameDelay);
SDK_API BUFCCDUSB_InstallFrameHooker(int FrameType, FrameDataCallBack FrameHooker);
SDK_API BUFCCDUSB_InstallUSBDeviceHooker(DeviceFaultCallBack USBDeviceHooker);
SDK_POINTER_API BUFCCDUSB_GetCurrentFrame(int FrameType, int deviceID, Byte &FramePtr);*
SDK_POINTER_API BUFCCDUSB_GetCurrentFrame_16Bit(int FrameType, int deviceID, unsigned short &FramePtr);*
SDK_API BUFCCDUSB_GetDevicesErrorState();
SDK_API BUFCCDUSB_ResetCameraTimer(int deviceID);
SDK_API BUFCCDUSB_SuspendCamera(int deviceID, int isSuspendCamera);
SDK_API BUFCCDUSB_SetGPIOConifg(int DeviceID, unsigned char ConfigByte);
*SDK_API BUFCCDUSB_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.**

EXPORT Functions:

BUF_USBCCDCamera_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, which is:

Mightex BUF CCD Camera Factory Control Window

User may simply invoke **BUFCCDUSB_ShowFactoryControlPanel()** and **BUFCCDUSB_HideFactoryControlPanel()** to show and hide this window.

SDK_API BUFCCDUSB_InitDevice(void);

This is first function user should invoke for his own application, this function communicates with the installed device driver and reserve resources for all further operations.
Arguments: None
Return: The number of Mightex Buffer CCD cameras (e.g. CCN/CCE/CGN/CGE/CXN/CXE…etc.) currently attached to the USB 2.0 Bus, if there's no Mightex CCD USB camera attached, the return value is 0.
Note: There's **NO** device handle needed for calling further SDK APIs, after invoking BUFCCDUSB_InitDevice, camera engine reserves resources for all the attached cameras. For example, if the returned value is 2, which means there TWO cameras currently presented on USB bus, 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 *BUFCCDUSB_AddCameraToWorkingSet(deviceID)* to set the camera as active.
Important: The device drivers and camera engines are different for Mightex CCD camera and Mightex CMOS (Buffer or Non-Buffer) cameras, CMOS camera will **not** be managed by the CCD camera engine, so they will not present in the return number. *Another point is that CCD camera engine does NOT run with CMOS camera engine con-currently, although they can be installed on the same PC without any problem…but only one of them 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:

```
BUFCCDUSB_InitDevice();        // Get the devices
BUFCCDUSB_AddCameraToWorkingSet( deviceID); // Adding cameras to "working set".
BUFCCDUSB_StartCameraEngine();
…… Operations ……              // including BUFCCDUSB_ActiveDeviceInWorkingSet()
```

When application terminates, it usually does:

```
MTUSB_StopCameraEngin();
MTUSB_UnInitDevice()
```

Note that users don't need to explicitly open and close the opened device, as BUFCCDUSB_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 *BUFCCDUSB_AddCameraToWorkingSet(deviceID).*

SDK_API BUFUSB_UnInitDevice(void);

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

SDK_API BUFCCDUSB_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 **BUFCCDUSB_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 ID)
-2 If the Camera engine is started already.
1 if the call succeeds.
Important: Usually, user might use this API to get Module/Serial No of a camera after invoking **BUFCCDUSB_InitDevice().** But **Must** before the camera engine is started.

SDK_API BUFCCDUSB_GetUserSerialNo(int DeviceID, char *UserSerialNo);
For any present device, user might get the user defined string by invoking this function.
Argument: DeviceID – the number (ONE based) of the device, Please refer to the notes of **BUFCCDUSB_InitDevice()** function for it.
UserSerialNo – the pointer to a character buffer, the buffer should be available for 16 characters.
Return: -1 If the function fails (e.g. invalid device ID)
-2 If the Camera engine is started already.
1 if the call succeeds.
Important: Usually, user might use this API to get the user defined string of a camera which was set by the **BUFCCDUSB_SetUserSerialNo()**. Note that this API must be invoked **Before** the camera engine is started.(or after the camera engine is stopped)
Note: The camera firmware reserves a 14 characters space for user defined string, as the last space must be EOS (End Of String, which is 0), actually, there're only 13 effective characters. User might set any content (e.g. user defined serial no.) to it by the **BUFCCDUSB_SetUserSerialNo()** API and later read it via this API.

SDK_API BUFCCDUSB_SetUserSerialNo(int DeviceID, char *UserSerialNo, int IsStoreToNVMemory);
For any present device, user might set the user defined string by invoking this function.
Argument: DeviceID – the number (ONE based) of the device, Please refer to the notes of **BUFCCDUSB_InitDevice()** function for it.
UserSerialNo – the pointer to a character buffer which contains the string to be written to camera, there can be at most 13 effective characters and it should be ended with EOS.
IsStoreToNVMemory – 0: The UserSerialNo will be written to camera, but **NOT** stored to NV memory
1: The UserSerialNo is written to camera and stored in NV memory.
Return: -1 If the function fails (e.g. invalid device ID)
-2 If the Camera engine is started already.
1 if the call succeeds.
Important: Usually, user might use this API to set the user defined string which later might be read by **BUFCCDUSB_GetUserSerialNo()**. Note that this API must be invoked **Before** the camera engine is started (or after the camera engine is stopped).
Note: The camera firmware reserves a 14 characters space for user defined string, as the last space must be EOS (End Of String, which is 0), actually, there're only 13 effective characters. User might set any content (e.g. user defined serial no.) to it by this API and later read it via **BUFCCDUSB_GetUserSerialNo()**.

SDK_API BUFCCDUSB_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 **BUFCCDUSB_InitDevice()** function for it.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Note: This API should be only invoked before the camera engine is started, after camera engine is started, user should user BUFCCDUSB_ActiveDeviceInWorkingSet() to enable/disable an added camera. Camera Engine will only grab frames from active cameras in "Working Set".

SDK_API BUFCCDUSB_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 **BUFCCDUSB_InitDevice()** function for it.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Note: This API should be only invoked before the camera engine is started, after camera engine is started, user should user BUFCCDUSB_ActiveDeviceInWorkingSet() to enable/disable an added camera. Camera Engine will only grab frames from active(enabled) cameras.

SDK_API BUFCCDUSB_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 **BUFCCDUSB_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, or the device specified by DeviceID is not in working set)
1: 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 **BUFCCDUSB_AddDeviceToWorkingSet(int DeviceID),** the camera is activated by default, However, user should only invokes AddDeviceToWorkingSet() and RemoveDeviceFromWorkingSet() before the camera engine is started. After the camera engine is started, user must use this API to activate or de-activate a certain camera.

SDK_API BUFCCDUSB_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…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, In application there's no GUI, this argument can be NULL.
CameraBitOption – 8 or 12, 8 for 8bit operation and 12 for 12bit operation, Please note that this setting is Global all the cameras in working set, so it's **NOT** possible to set some cameras in 8bit mode and others in 12bit mode when there're multiple cameras active in an application.
Return: -1 If the function fails (e.g. invalid device handle)
1 if the call succeeds.
2 When CameraBitOption is 12, but Camera Engine finds there's cameras in working set which don't support 12bit mode, camera engine will still start the engine normally but setting all camera to 8bit mode. It's recommended for user to always check the return value of this function, especially while setting to 12bit mode.
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. If user wants to activate or de-activate a certain camera after the camera engine is started, user should invoke BUFCCDUSB_ActiveDeviceInWorkingSet().***

SDK_API BUFCCDUSB_StopCameraEngine(void);

This function stops the started camera engine.
Argument: None.
Return: it always returns 1.

SDK_API BUFCCDUSB_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 there's available memory for a new frame, For example, when host (in most cases, it's a PC) is keeping to get frame from the camera, the camera is continuously grabbing frames from CMOS sensor.
TRIGGER mode – Camera will only grab a frame from CMOS sensor while there's an external trigger asserted (and there's available memory for a new frame).
Note: While FrameCallBack is installed, the callback function is invoked for each frame grabbed by the camera engine, so basically, there's no big difference for high level applications for these two modes.
Note: On important side effect of mode setting is that camera will clean the on-camera frame buffer, user might use this API to set camera to NORMAL mode even when the camera was in NORMAL mode, the camera will still be in NORMAL mode but all the on-camera frame buffer are cleaned.
User might use this function in some scenarios, as on-camera frame buffer is a problem when user wants to get a "fresh" frame, in these applications, user might do:

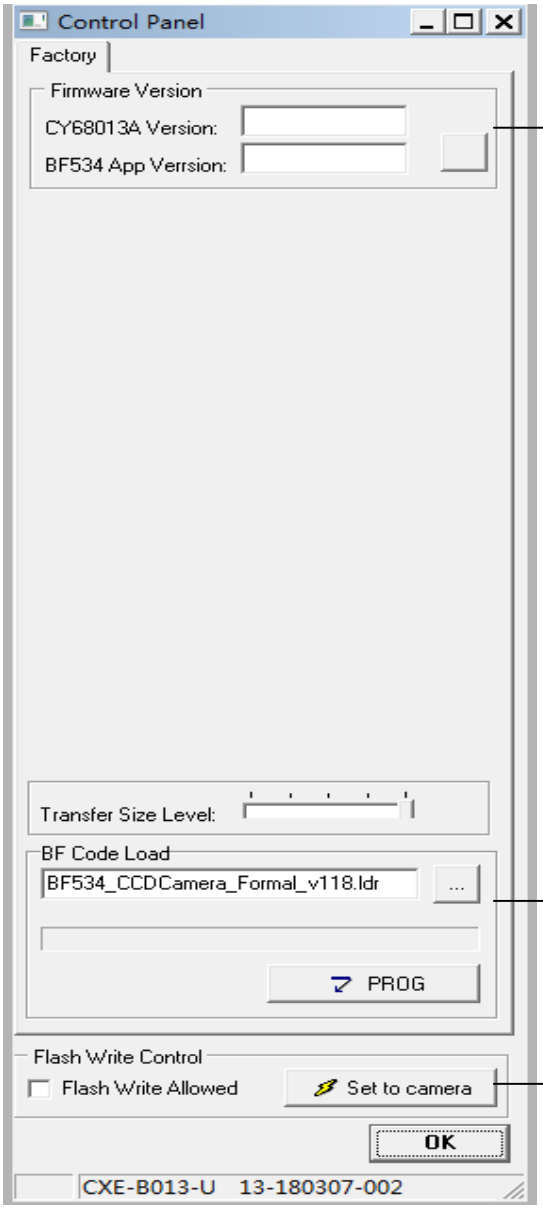
```
BUFCCDUSB_SetCameraWorkMode(); // this clear all the frames in buffer.
BUFCCDUSB_StartFrameGrab(1);   // Get one frame only, but it should be a "fresh" frame, not a frame in buffer.
```

SDK_API BUFCCDUSB_StartFrameGrab(int TotalFrames);
SDK_API BUFCCDUSB_StopFrameGrab(void);

When camera engine is started, the engine prepares all the resources, but it does **NOT** start the frame grabbing until **BUFCCDUSB_StartFrameGrab()** function is invoked. After it's successfully invoked, camera engine starts to grab frames from cameras in current "Working Set". User may call **BUFCCDUSB_StopFrameGrab()** to stop the engine from grabbing frames from camera.
Argument: TotalFrames – This is for BUFCCDUSB_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 the camera engine will always grab available frame from camera until user calls BUFCCDUSB_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 BUFCCDUSB_ShowFactoryControlPanel(int DeviceID, char *passWord);

For user to access the factory features conveniently and easily, the library provides a 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" should be used in most cases.
Return: -1 If the function fails (e.g. invalid device number)
1 If the call succeeds.
Note: Before user invokes this API to show the factory panel, there must be camera in the working set and camera engine is started.



Click to button will get firmware version information from camera.

User can upgrade firmware with those controls, for details, please refer to the user manual.

User might check the “Flash Write Allowed” checkbox and click the [Set to camera] button, this enable the flash Writing, it must be done before update the USB firmware (*.iic file), note this is **not** needed for upgrade DSP firmware.

SDK_API BUFCCDUSB_HideFactoryControlPanel(void);

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

SDK_API BUFCCDUSB_SetCustomizedResolution(int deviceID, int RowSize, int ColumnSize, int Bin, int BufferCnt);

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, for CCD Camera, the row size is fixed actually for a certain camera type, e.g. For CCX module cameras, it MUST be 1392, for CGX modules, it must be 1280.

ColumnSize – the customer defined column size.

Bin – We have the following definitions for it:

*. **CCX/CXX Modules:**

0: Normal mode,
0x81: 1:2 BIN mode,
0x82: 1:3 BIN mode,
0x83: 14: BIN mode,
0x03: 1:4 SKIP mode

*. **CGX Modules:**

0: Normal mode,
0x81: 1:2 BIN mode,
0x82: 1:3 BIN mode,
0x83: 14: BIN mode,
0x03: 1:4 Bin mode2

BufferCnt – the frame number of the on camera buffer (for this resolution).

Return: -1 : If the function fails (e.g. invalid device number)
-2 : The customer defined row or column size is out of range.
-3 : The customer defined row/column size must be a number of times of 8, otherwise it return –3.
1 if the call succeeds.

Note:

1). RowSize and ColumnSize should be set properly according to the camera type, For existing CCD Cameras, we have the following available settings:

*. CCX/CXX Modules

1.3M CCN/CCE/CXN/CXE Mono Modules: (CCN-B013-U, CCE-B013-U, CXN-B013-U and CXE-B013-U)

BinMode:
0 : No Bin/Skip, it's 1392x1040 as full resolution, ROI is only possible on column size:
0x81 : 1:2 Bin mode, it's pre-defined as: 1392 x 520
0x82 : 1:3 Bin mode, it's pre-defined as: 1392 x 344
0x83 : 1:4 Bin mode, it's pre-defined as: 1392 x 256
0x03 : 1:4 Skip mode, it's pre-defined as: 1392 x 256

1.3M CCN/CCE/CXN/CXE Color Modules: (CCN-C013-U, CCE-C013-U, CXN-C013-U and CXE-C013-U)

BinMode:
0 : No Bin/Skip, it's 1392x1040 as full resolution, ROI is only possible on column size:
0x03 : 1:4 Skip mode, it's pre-defined as: 1392 x 256
Note: for C013 Camera, Bin modes (1:2Bin, 1:3Bin and 1:4Bin) are not supported.

2M CCN/CCE Mono Modules: (CCN-B020-U, CCE-B020-U)

BinMode:
0 : No Bin/Skip, it's 1616x1232 as full resolution, ROI is only possible on column size:
0x81 : 1:2 Bin mode, it's pre-defined as: 1616 x616
0x82 : 1:3 Bin mode, it's pre-defined as: 1616 x410
0x83 : 1:4 Bin mode, it's pre-defined as: 1616 x 308
0x03 : 1:4 Skip mode, it's pre-defined as: 1616 x 308

2M CCN/CCE Color Modules: (CCN-C020-U, CCE-C020-U)

BinMode:
0 : No Bin/Skip, it's 1616x1232 as full resolution, ROI is only possible on column size:
0x03 : 1:4 Skip mode, it's pre-defined as: 1616 x 308
Note: for C020 Camera, Bin modes (1:2Bin, 1:3Bin and 1:4Bin) are not supported.

So, the ROI is only possible while BinMode is “0”, please also note that the Row Size is actually fixed (to 1392 for CCX-X013-U modules, for example) in all cases, actually only the ColumnSize can be set to a customized number when BinMode is 0, but it must be multiples of 8. (e.g. 64, 128…etc.)
It’s important to set Row Size, Column Size and BinMode correctly according to above available options to a certain Camera.
2). Buffer Count should not exceed Max buffer count (maximum is 8 for CCX cameras), it’s important for setting the buffer count, if the buffer count is too small, the frame rate might be affected, as device might have to ignore frames while the buffer is full (and host is not fetching frames on time). While setting buffer count too big, it’s maximum optimized for frame rate, but frame delay might be an issue. The setting of buffer count is very dependent on the Host bandwidth.
3). Buffer Option should always be 0 currently.

Example1: Set camera to 1392x1040 (default), No Bin/Skip mode, 4 buffer count, we have:
BUFCCDUSB_SetCustomizedResolution(deviceNO, 1392, 1040, 0, 4);

Example2: Set camera to 1392x480 (ROI), No Bin/Skip mode, 4 buffer count, we have:
BUFCCDUSB_SetCustomizedResolution(deviceNO, 1392, 480, 0, 4);

Example3: Set camera to 1392x256, 1:4 Bin, 4 buffer count, we have:
BUFCCDUSB_SetCustomizedResolution(deviceNO, 1392, 256, 0x83, 4);

*. CGX Modules

CGN/CGE Mono Modules: (CGN-B013-U and CGE-B013-U)
CCN-C013-U/CCE-C013-U: (CCN-C013-U and CCE-C013-U)

BinMode:
0 : No Bin/Skip, it’s 1280x960 as full resolution, ROI is only possible on column size:
0x81 : 1:2 Bin mode, it’s pre-defined as: 1280 x 480
0x82 : 1:3 Bin mode, it’s pre-defined as: 1280 x 320
0x83 : 1:4 Bin mode, it’s pre-defined as: 1280 x 240
0x03 : 1:4 Bin mode2, it’s pre-defined as: 1280 x 240
For Bin mode, the pixel is the sum of the bin area (e.g. for 1:2 bin, the pixel value is the sum of the 2x2 area, which is actually 4 pixels, it’s Row Bin and Column Bin), the **Bin mode2** *is slightly different, it’s sum of the first column pixel of the rows of the bin area, thus it’s sum of 2 pixels for a 2x2 area (in 1:2 bin mode), so it’s Row Bin and Column Skip. Note that for CGN and CGE C013 modules (color camera), we have:*
1). Color information is ***NOT*** *present in the frame data any more when the resolution is set to Bin mode, although the returned BMP data format is still RGB24(please refer to the frame callback described below), the R, G, B pixels are with the same value.*
2). “Sharp” algorithm should NOT be set while the resolution is set to 0x03.(Bin Mode2)

So, the ROI is only possible while BinMode is “0”, please also note that the Row Size is actually fixed (to 1280) in all cases, actually only the ColumnSize can be set to a customized number when BinMode is 0, but it must be multiples of 8. (e.g. 64, 128…etc.)
It’s important to set Row Size, Column Size and BinMode correctly according to above available options to a certain Camera.
2). Buffer Count should not exceed Max buffer count (maximum is 24 for CGX cameras), it’s important for setting the buffer count, if the buffer count is too small, the frame rate might be affected, as device might have to ignore frames while the buffer is full (and host is not fetching frames on time). While setting buffer count too big, it’s maximum optimized for frame rate, but frame delay might be an issue. The setting of buffer count is very dependent on the Host bandwidth.
3). Buffer Option should always be 0 currently.

Example1: Set camera to 1280x960 (default), No Bin/Skip mode, 4 buffer count, we have:
BUFCCDUSB_SetCustomizedResolution(deviceNO, 1280, 960, 0, 4);

Example2: Set camera to 1280x480 (ROI), No Bin/Skip mode, 4 buffer count, we have:
BUFCCDUSB_SetCustomizedResolution(deviceNO, 1280, 480, 0, 4);

Example3: Set camera to 1280x240, 1:4 Bin, 4 buffer count, we have:
BUFCCDUSB_SetCustomizedResolution(deviceNO, 1280, 240, 0x83, 4);

SDK_API BUFCCDUSB_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 – 200,000ms, So the exposure time value here is from 1 – 4000,000.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Note: Although the minimum exposure time can be set to 50us, camera might revise it to a minimum achievable exposure time in firmware, the minimum achievable exposure time of a certain camera is related to its current mode (NORMAL or TRIGGER) and its current CCD main clock (frequency). Basically, for CCX cameras, when CCD frequency is 28MHz (default), the minimum ET is:
NORMAL Mode: 1 Row Time (~68us)
TRIGGER Mode: 3 Row Times (~200us)
With slower CCD frequency, the minimum achievable ET is increased proportionally.

SDK_API BUFCCDUSB_SetFrameTime(int DeviceID, int frameTime);

User may set the frame time by invoking this function, this tries to set camera's frame rate precisely.
Argument: DeviceNo – the device number which identifies the camera will be operated.
frameTime – the frame Time is set in 100 Microsecond UNIT, e.g. if it's 400, the frame time of the camera will be set to 40ms (40000us). Camera will do**"best effort"** to achieve this frame time, however, under a certain resolution, the minimum and maximum frame time is also affected by some other factors, such as resolution itself, exposure time, and even the available USB bandwidth…etc. The actual frame time (frame rate) might not be this set time.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.
Note: By default, the frame time is set to Minimum internally to make the camera generate maximum frame rate under current resolution at current exposure time, for example, for CCX-B013-U camera, at 1392x1040 resolution, the frame rate is 15fps @ET<68ms, user might set frame time to 100ms, so the frame rate will be set to 10fps precisely. User can't get frame rate more than 15fps even with the frame time set to less than 68ms as 15fps is the physical output rate of the CCD sensor at this resolution (under 28MHz main clock). Similarly if the ET is longer than 100ms, in the above example, set frame time to shorter than 100ms won't generate 10fps rate. So setting frame time works in a "best effort" way.

SDK_API BUFCCDUSB_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), Actually, the XStart MUST be 0, as the ROI on row size is not allowed. (refer to above **BUFCCDUSB_SetCustomizedResolution()** description)
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 sensor pixel area, user may use this function to set the left top point of the ROI (when XStart is 0, it's actually the starting Row). Note that under a certain ROI resolution, the 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. Also note that the YStart must be a value of multiples of 8.

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

User may set the **global** gain (for Red, Green and Blue pixels) 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 in dB.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.

Important: The gain value should be from 6dB – 41dB, represents 2x – 112x of analog amplifying Multiples. For all curnent Monochrome sensor and color sensor modules, **only Green Gain value is used by the camera firmware** as Global gain for all pixels. Thus for color camera, user might need another API "**BUFCCDUSB_SetGainRatios()**" to Set "digital" gain of Red and Blue pixels.

Note:

1). 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 on consumer camera), for applications which the SNR is important, it's recommended to set Gain not more than 16dB.

2). For **CCX** modules, although the minimum Gain is 6dB, user might have to set it to 14dB when the camera is NOT in BIN mode. With the current hardware/firmware design, the CCD output (Sony ICX205 or ICX274) is only up to ~0.45V as its saturation voltage, even with 6dB gain (2x), it's ~0.9V signal, while the CCD processor is with a 2V reference ADC, only set the Gain to 14dB will let the ADC generate full range data. Here, we leave this feasibility to users, as in some cases, user might still want to set Gain to 6dB to get optimized SNR (rather than the ADC range). In most of the applications, the **Minimum Gain** recommended for CCX-B013-U/CCX-C013-U is as following:

No Bin mode (Bin = 0, or Bin = 0x03<Skip mode>), Gain = 14 (dB)
1:2 Bin mode (Bin = 0x81), Gain = 8 (dB)
1:3 Bin mode (Bin = 0x82), Gain = 6 (dB)
1:4 Bin mode (Bin = 0x83), Gain = 6 (dB)

3). For **CXX** modules, although the minimum Gain is 6dB, user might have to set it to 9dB when the camera is NOT in BIN mode. With the current hardware/firmware design, the CCD output (Sony ICX285) is only up to 0.8V as its saturation voltage, even with 6dB gain (2x), it's ~1.6V signal, while the CCD processor is with a 2V reference ADC, only set the Gain to 9dB will let the ADC generate full range data. Here, we leave this feasibility to users, as in some cases, user might still want to set Gain to 6dB to get optimized SNR (rather than the ADC range). In most of the applications, the **Minimum Gain** recommended for CXX-B013-U is as following:

No Bin mode (Bin = 0, or Bin = 0x03<Skip mode>), Gain = 9 (dB)
1:2 Bin mode (Bin = 0x81), Gain = 6 (dB)
1:3 Bin mode (Bin = 0x82), Gain = 6 (dB)
1:4 Bin mode (Bin = 0x83), Gain = 6 (dB)

4). For **CGX** modules, although the minimum Gain is 6dB, user might have to set it to 15dB when the camera is NOT in BIN mode. With the current hardware/firmware design, the CCD output (Sony ICX445) is only up to 0.38V as its saturation voltage, even with 6dB gain (2x), it's ~0.76V signal, while the CCD processor is with a 2V reference ADC, only set the Gain to 15dB will let the ADC generate full range data. Here, we leave this feasibility to users, as in some cases, user might still want to set Gain to 6dB to get optimized SNR (rather than the ADC range). In most of the applications, the **Minimum Gain** recommended for CGX-B013-U/CGX-C013-U is as following:

No Bin mode (Bin = 0) , Gain = 15 (dB) for B013 module and Gains = 17(dB) for C013 module.
1:2 Bin mode (Bin = 0x81), Gain = 8 (dB)
1:3 Bin mode (Bin = 0x82), Gain = 6 (dB)
1:4 Bin mode (Bin = 0x83), Gain = 6 (dB)
1:4 Bin mode2 (Bin = 0x03), Gain = 8 (dB)

SDK_API BUFCCDUSB_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 (CXX-C013-U) 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 succeeds.

Important: On some types of Color CCD Cameras, there's only one global hardware gain for all R, G, B pixels (similar to the Monochrome camera), so when user invokes the above **BUFCCDUSB_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, 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 BUFCCDUSB_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
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 BUFCCDUSB_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 BUFCCDUSB_SetSoftTrigger(int deviceID);

User may simulate an external trigger via this function.

Argument: DeviceNo – the device number which identifies the camera will be operated.

Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.

Important:
When the camera is set in TRIGGER mode, camera will grab one frame upon each external trigger assertion, User might use this API to simulate an external trigger assertion, each invoking of this function will let camera to grab one frame while it's in TRIGGER mode.

SDK_API BUFCCDUSB_SetCCDFrequency(int deviceID, int Frequency);

User may set CCD Sensor's working frequency (main clock) by invoking this function.

Argument: DeviceNo – the device number which identifies the camera will be operated.
Frequency – the working frequency of CCD sensor.

Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.

Important:
1). The Frequency has the following definitions:

For **CCX** Modules, we have:
For B013/C013 Modules: 0 -- 28MHz (default), 1 -- 14MHz, 2 -- 7MHz, 3 -- 3.5MHz, 4 -- 1.75MHz
For B020/C020 Modules: 0 -- 32MHz (default), 1 -- 16MHz, 2 -- 8MHz, 3 -- 4MHz, 2 -- 1.75MHz
For **CXX** Modules, we have:
0 -- 28MHz (default), 1 – 18MHz, 2 -- 14MHz, 3 -- 7MHz, 4 -- 3.5MHz
For **CGX** Modules, we have:
0 -- 32MHz (default), 1 -- 16MHz, 2 -- 8MHz, 3 -- 4MHz, 4 -- 2MHz
2). In most cases, CCD is working at default frequency, however, user might want to set it to a lower frequency in some cases as following:
*. If the frame rate is not a concern, lower CCD clock may deliver better SNR.
*. If user wants to get better frame rate at a certain exposure time, as the CCD sensor's output frame is very related to its frame time and current exposure time, for example, for CCX camera, at 1392x1040@28MHz, the frame time is ~67ms, while ET is less than the frame time (67ms), the frame rate is 15fps, however, if the ET is bigger than frame time, the frame rate is down to 5fps only (due to the hardware/firmware design of the camera). User might set CCD Frequency to 14MHz in this case, so the frame time is ~136ms at the same resolution (1392x1040), and the frame rate when ET is less than 136ms will be (15/2)=7.5fps. So user can see that in this example (e.g. exposure time = 100ms @1392x1040), setting clock to 14MHz will actually deliver higher frame rate (7.5fps) than 28MHz (5fps)
Note: In most applications, user should use the default (maximum) CCD clock, and if user invokes this API to change the main clock, the camera itself will automatically adjusts its register to keep all the pre-set camera parameters (e.g. exposure time) unchanged.

SDK_API BUFCCDUSB_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 succeeds.
Note: Set Minimum frame delay to "1" will let the camera engine ignore the "old" frames in frame buffer, so this will Show lower frame rate, but the frame delay (from the camera grabs a frame to the frame is delivered to user via the callback function) is minimized.

SDK_API BUFCCDUSB_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 UserMark;               /* Reserved */
    int  FrameTime;
    int  CCDFrequency;
    int FrameProcessType;   /* Raw or DIB */
    int FilterAcceptForFile; /* Reserved */
} 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 user set. Note that the Row and Column here is Number, NOT Size, this is a little confuse sometimes, e.g. For CCX modules @1392x1040 resolution, the Row Size is 1392, the Column Size is 1040, however the Row Number is 1040 (equals to Column Size), the Column Number is 1392 (equals to Row Size). Please also note that the Row/Column should also corresponds to the Bin value, e.g. while Bin is 1 (1:2 bin mode), the Row Number should be 520, while the Column Number should be 1392.

Bin – Refer to the Set resolution API for Bin definition.

Xstart, Ystart – the actual (X,Y) start position of ROI of this frame, note the Xstart is always 0, refer to SetXYStart API.

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 – For Green Gain it's the value of Global gain for this frame, might be from 6 to 41. For RedGain and BlueGain, they're final ratio for Red and Blue pixels (refer to the above "**BUFCCDUSB_SetGainRatios()"** API)

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 – While the camera is in **NORMAL** mode, camera is grabbing frames continuously (as long as host is fetching frames from it), If an external trigger signal asserted during a frame grabbing, camera will set this flag to ONE (otherwise it's ZERO). This gives host a choice to know the trigger assertion occurred for a certain frame even it's in **NORMAL** mode. This flag should NOT be used while it's in **TRIGGER** mode.

TriggerEventCount – While the camera is in **TRIGGER** mode, camera will grab ONE frame after it's triggered by the external signal, each effective trigger will increase this count by ONE, this gives host a clue for the relation of the frame and its trigger signal event. Note that this count is reset to ZERO whenever host set the work mode to TRIGGER (so host can reset the count by invoking **BUFCCDUSB_SetCameraWorkMode(DeviceID, 1),** even the camera is already in **TRIGGER** mode). Also note that the frame time can be from micro-seconds to hundreds of seconds (depending on exposure time), if trigger signals are generated more then once during a frame time, the count is still physically reflect the trigger signal assertions. For example, the TriggerEventCount for the first frame is 1, and the next frame's TriggerEventCount might be 6, this means assertion 1 generates the first frame, assertion 2 to 5 are occurred during the exposure or reading time of first frame…and they're actually ignored. The 6th assertion generates the second frame.

UserMark – Reserved.

FrameTime – User set Frame Time, refer to SetFrameTime API.

CCDFrequency – Current CCD Frequency Setting, See description of API **"BUFCCDUSB_SetCCDFrequency()"**

FrameProcessType – Current Frame Type (RAW or BMP)

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[RowNum][ColumnNum]; // For 8bit Camera, e.g. for 1392x1040 resolution, it's // Pixels[1040][1392].

OR

Unsigned char Pixels[RowNum][ColumnNum][2]; // For 12bit Camera

Here, Pixels[][][0] is the 8bit MSB (e.g D11—D4 for 12bit camera), and Pixels[][][1] contains 4 LSB (D3 – D0 for 12bit camera), all unused MSB are padded with 0).

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[RowNum][ColumnNum]; // For Mono camera, it's actually the same as RAW format.

**. Color camera*

The BytePtr points to

Unsigned char Data[RowNum][ColumnNum][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 in BMP mode, the return data format is ***NOT*** *affected by the bit set of the camera (8bit or 12bit)…Camera Engine will ignore the 4 LSB of camera cell even it's in 12bit mode and the return data is always 8bit. So for applications want to have 12bit data, please always install hooker with RAW mode if user wants to use 12bit pixel value.*
One more notice is that for CGX-C013 camera modules (CGN and CGE), while the resolution is set to Bin (0x81, 0x82, 0x83 or 0x03), color information is not present (from CCD sensor), so the above Blue, Green and Red pixel values are the same.

Note that this callback is invoked in the main thread of the host application, GUI operations are allowed in this callback, 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 some cases (depending on Host resources), this is especially true for color sensors…and usually user can set bigger buffer count in this case for higher frame rate**.**

Important:
1). As the frame callback is invoked by a windows timer 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, and 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);
    }
}
```

2). It's recommended to un-hook the user installed frame callback by invoking the "**BUFCCDUSB_InstallFrameHooker()"** with hooker function set to NULL (or nullptr) before stopping the frame grabbing when user's application is terminated.

SDK_API BUFCCDUSB_InstallUSBDeviceHooker(DeviceFaultCallBack USBDeviceHooker);
User may call this function to install a callback, which will be invoked by camera engine while a new camera is plugged in, an existing camera is plugged out or camera engine encounter camera errors.
Argument: USBDeviceHooker – the callback function registered to camera engine.
Return: It always return 1.
Note:
1). The camera engine supports Plug&Play of the cameras now, but user must install device hooker with this API to be notified by the camera engine, when a new camera is plugged in, the installed callback will be invoked with NotifyType is set to 1. Similarly, when an existing camera is plugged out, the callback will be invoked with NotifyType is set to 0.
2). If an error occurs during frame grabbing (e.g. hardware error of the camera), the camera engine will stop its grabbing from this camera (the camera is disabled internally) and invoke the installed callback function with the NotifyType set to 2.
3). The callback function has the following prototype:
typedef void (DeviceFaultCallBack)(int NofityType);*

The NotifyType is set by camera engine as following:

0 – An existing camera is plugged out.

1 – A new camera is plugged in.

2 – Hardware error occurs in camera frame grabbing.

When the callback is invoked by camera engine, user software should do proper house keeping and terminate itself (if the NotifyType is 2) OR it might let user to re-start the camera engine to enumerate the existing cameras (If the NotifyType is 0 or 1).

4). Although the camera engine supports P&P, it's **NOT** recommended to plug out a camera from USB while host is grabbing image from it.

SDK_API BUFCCDUSB_ResetCameraTimer(int deviceID);

User may call this function to reset the internal timer of a certain camera.

Argument : DeviceID – The device number, which identifies the camera is being operated.

Return: -1 If the function fails (e.g. invalid device number)

1 if the call succeeds.

Note: This API resets the camera's internal timer to "0", thus the time stamp (in the frame property) will restart (from 0).

SDK_API BUFCCDUSB_SuspendCamera(int deviceID, int isSuspendCamera);

User may call this function to suspend/resume the camera..

Argument : DeviceID – The device number, which identifies the camera is being operated.

isSuspendCamera – 0: Set the camera in Normal working mode.

1: Set the camera to Suspend mode

Return: -1 If the function fails (e.g. invalid device number OR Camera engine isn't started yet)

1 if the call succeeds.

Note: Although the API returns immediately, it might take ~2 seconds to resume a camera (from suspend mode), thus the application software should wait for some time (e.g. sleep(2000)) after resuming a camera.

SDK_API BUFCCDUSB_GetCurrentFrame(int FrameType, int deviceID, Byte* FramePtr);
SDK_API BUFCCDUSB_GetCurrentFrame_16Bit(int FrameType, int deviceID, unsigned short* FramePtr);
SDK_API BUFCCDUSB_GetDevicesErrorState();

These two APIs are mainly for develop tools in which callback mechanism is NOT supported, please refer to the application notes for the description of them.

SDK_API BUFCCDUSB_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 BUFCCDUSB_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). How to Minimize Frame Delay

User might see obvious frame delay in some cases, this is because the camera itself is always grabbing the frame whether or not host asks frame from it...the camera put the frames in its on-camera buffer, if user wants to minimize the frame delay, user might do:

a>. When setting resolution by "BUFCCDUSB_SetCustomizedResolution()", set the BufferCnt to "1", that sets the on-camera buffer to 1.

b>. User might invoke BUFCCDUSB_SetCameraWorkMode(CameraID, WorkMode) before invoking BUFCCDUSB_StartGrabFrames(), note that the WorkMode can be the same one as current mode of the camera (e.g. it's always "0", means NORMAL mode), as firmware will always ignore the old frames in buffer (even the buffer count is 1, it's possible the one is the OLD one) while it gets this command, and always re-start to grab a new frame from sensor.

c>. User should invoke the *BUFCCDUSB_SetMinimumFrameDelay(int IsMinimumFrameDelay);* with IsMinimumFrameDelay set to 1.

2). How to get frames from a specified camera only

The API BUFCCDUSB_StartGrabFrames (frameNumber) will get frames from all the cameras in working set, some time it's desirable to get a certain number of frames from a specified camera only if there're multiple cameras connected. For achieving this, user might use do the following:

User might want to have a function "int BUFCCDUSB_StartGrabFrameFromCamera(int TotalFrames, int camId);" Although we don't have this exact API from our DLL, but we have an API as following:

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

Note:

a>. While user add 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 )
{
    BUFCCDUSB_ActiveDeviceInWorkingSet( cameraID, 1 );  // Activate the specified camera
    for ( i=1; i<MAX_CAMERA; i++)
     {
        if ( i != cameraID)
          BUFCCDUSB_ActiveDeviceInWorkingSet( i,0);   // De-activate other cameras.
      }
}
```

c>.So user might implement the mentioned function:

```
int BUFCCDUSB_StartGrabFrameFromCamera(int TotalFrames, int camId)
{
   ActivateOneCameraOnly( camId );
   BUFCCDUSB_StartGrabFrames( TotalFrames);
}
```

Combined the above (1) and (2), if user wants to get frames from a certain camera with minimized frame delay, user might do:

```
BUFCCDUSB_SetCameraWorkMode( cameraID, 0) ; // Assume using "NORMAL" mode
BUFCCDUSB_StartGrabFrameFromCamera(1, cameraID); // assume get 1 frame, can be more.
```

3). Why we have BUFCCDUSB_AddDeviceToWorkingSet() and BUFCCDUSB_ActiveDeviceInWorkingSet()

The first API **BUFCCDUSB_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 for "Permanent" map of the working set. (Similary, the BUFCCDUSB_RemoveDeviceFromWorkingSet() should be used BEFORE the camera engine is started),

Generally, We expect user have the following software architecture:

```
// Do USB Devices Initialization
 TotalDevices = BUFCCDUSB_InitDevice();
 // Identify cameras by Module No. and Serial No.
 for (i=1; i<=TotalDevices; i++)
 {
      BUFCCDUSB_GetModuleNoSerialNo( i, char *ModuleNo, char *SerialNo);
      BUFCCDUSB_AddDeviceToWorkingSet(i);
 }
BUFCCDUSB_StartCameraEngine( );

//…after camera engine is started, user can use BUFCCDUSB_ActivateCameraInWorkingSet() to temporarily
//   Enable/Disable a camera.
```

After Camera Engine is started, user might operate 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 parallel at a certain point.
For example, If user has multiple cameras connected and added to working set (e.g. 4 cameras), user might and set proper parameters (e.g. Working Mode, resolution, exposure time…etc.) for all cameras, and user might set all cameras in "Trigger" mode, which makes it to wait for an external trigger to grab a frame…as camera has on-camera frame buffer, a "de-active" camera will still grab a frame after it's triggered, the frame is stored on its frame buffer…later on, user might activate cameras one by one and get frame from them, the frames are synchronized by the same external trigger signal in this case.

4). For development tools without callback mechanism

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

```
SDK_POINTER_API  BUFCCDUSB_GetCurrentFrame( int FrameType, int  deviceID, Byte*  &FramePtr );
SDK_POINTER_API  BUFCCDUSB_GetCurrentFrame_16Bit( int FrameType, int  deviceID, unsigned short*
                                                  &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 **BUFCCDUSB_InstallFrameHooker(),** this API returns a pointer (as its return value and its third argument) to a block of memory which contains:

```
{
   TprocessedDataProperty  ImageProperty;
   Byte Paddings[512-sizeof(TprocessedDataProperty)];
   Byte  ImageData[]; // OR unsigned short ImageData[];  for 10/12bit image
}
```

Note that it points to a block of memory which has two parts:
1). A 512 bytes block which contains TprocessedDataProperty at the starting of it…and paddings.
2). The Image Data.

For the description of these two fields, user might refer to the "Attributes" and "BytePtr" description in callback function *FrameDataCallBack(TProcessedDataProperty* Attributes, unsigned char *BytePtr);*
these two pointers pointed to the same memory blocks as the "ImageProperty" and "ImageData" above. Especially for the "ImageData", it has different memory map for RAW or BMP mode, for BMP mode, it has various maps for Mono/Color.
The API will return NULL when it fails.

SDK_API BUFCCDUSB_GetDevicesErrorState();
User might use this API to poll the camera engine if there's a low level engine error, this API only works while the Device Fault Callback is NOT installed via BUFCCDUSB_InstallUSBDeviceHooker(). It returns 0 means there's NO error, otherwise, there's error occurred.

5). 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:

```
Static void StopCamera()
{
    BUFCCDUSB_StopFrameGrab();
    BUFCCDUSB_StopCameraEngine();
    BUFCCDUSB_UnInitDevice();
}

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

   ret = BUFCCDUSB_InitDevice();
   assert(ret==1);
   ret = BUFCCDUSB_AddDeviceToWorkingSet(1);
   ret = BUFCCDUSB_InstallUSBDeviceHooker( CameraFaultCallBack );
   ret = BUFCCDUSB_StartCameraEngine(NULL, 8);
   assert(ret);
   if (-1==BUFCCDUSB_SetResolution(1, 8, 0, 4))
   {
        StopCamera();
        return FALSE;
   }
   ret = BUFCCDUSB_StartFrameGrab(GRAB_FRAME_FOREVER);
   assert(ret==1);
   ret = BUFCCDUSB_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 windows message loop to be active
        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)

BUFCCDUSB_ShowFactoryControlPanel()

BUFCCDUSB_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 "BUF_USBCCDCamera_SDK_DS.dll" and it can be found in CDROM under "\DirectShow\x86\MightexBufferCCDCameraEngine\" for x86 OS, and under "\DirectShow\x64\MightexBufferCCDCameraEngine\" for x64 OS.