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 Super Speed USB Camera (SM-Series) SDK Manual

Version 1.0.4

Dec.12, 2018

Relevant Products

Part Numbers
SMN-B050-U, SMN-C050-U, SME-B050-U, SME-C050-U SMN-B012-U, SMN-C012-U, SME-B012-U, SME-C012-U

Revision History

Revision	Date	Author	Description
1.0.0	Jul. 26, 2012	JT Zheng	Initial Revision
1.0.1	Dec.12,2012	JT Zheng	Add SetTransferSize
1.0.2	Jun. 26, 2013	JT Zheng	Adding B012/C012 camera supports
1.0.3	Jul. 2, 2014	JT Zheng	Adding USB Monitor API
1.0.4	Dec.12,2018	JT Zheng	New Logo

Mightex USB 3.0 SMX camera is mainly designed for low cost high speed machine vision applications. With high speed USB 3.0 interface and powerful PC camera engine, the camera delivers 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 3.0 for data collection, USB3.0 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 Super Speed Classic USB Camera User Manual*.**

The Mightex SMX camera can work with USB2.0 High Speed with lower frame rate.

SDK FILES:

The SDK includes the following files:

\LIB directory:

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

\Documents directory:

Mightex SMX Camera SDK Manual.pdf

\Examples directory

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

Note

1). The frame rate specified on camera's specification is based on the assumption that there're enough USB bandwidth available for data transferring, if there're multiple cameras connected, USB bandwidth has to be shared and that might cause slower frame rate.

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. These cameras might be different types.

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 “SSClassic_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
#define SDK_POINTER_API extern "C" __declspec(dllexport) unsigned char * _cdecl
#define SDK_POINTER_API2 extern "C" __declspec(dllexport) unsigned short * _cdecl
#define SDK_POINTER_API3 extern "C" __declspec(dllexport) unsigned long * _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
#define SDK_POINTER_API2 extern "C" __declspec(dllimport) unsigned short * _cdecl
#define SDK_POINTER_API3 extern "C" __declspec(dllimport) unsigned long * _cdecl
#endif

#define GRAB_FRAME_FOREVER                    0x8888


typedef struct {
    int CameraID;
    int WorkMode;                    // 0 - NORMAL mode, 1 - TRIGGER mode
    int SensorClock;                 // 24, 48, 96 for 24MHz, 48MHz and 96MHz
    int Row;                         // It's ColSize, in 1280x1024, it's 1024
    int Column;                      // It's RowSize, in 1280x1024, it's 1280
    int Bin;                         // 0, 1, 2 for no-decimation, 1:2 and 1:4 decimation
    int BinMode;// 0 - Skip, 1 - Bin
    int CameraBit; // 8 or 16.
    int XStart;
    int YStart;
    int ExposureTime; // in 50us unit, e.g. 100 means 5000us(5ms)
    int RedGain;
    int GreenGain;
    int BlueGain;
    int TimeStamp; // in 1ms unit, 0 - 0xFFFFFFFF and round back
    int SensorMode;// Bit0 is used for GRR mode, Bit1 is used for Strobe out enable
    int TriggerOccurred; // For NORMAL mode only, set when trigger occurred during the grabbing of last frame.
    int TriggerEventCount; // Reserved.
    int FrameSequenceNo; // Reserved.
    int IsFrameBad; // Is the current frame a bad one.

    int FrameProcessType; // 0 - RAW, 1 - BMP
    int FilterAcceptForFile; // Reserved
} TProcessedDataProperty;

 typedef struct {
    int FrameTotalRows;   // including V_Blanking Rows
    int VBlankingRows;    // Actual VBlanking rows (expanded if ET is longer)
    int VBlankingRowsInTriggerMode; // The rows set in VBlanking register.
    int RowClocks;        // including H_Blanking clocks
    int HBlankingClocks;  // setting in HBlanking register
    int ExposureTimeRows; // setting in ET registers
    int RowTimeinUs;      // including H_Blanking
    int FrameReadingTimeInUs;   // V_Blanking is not included, F_Valid time
    int VBlankingTimeInUs;      // Actual VBlanking time in "us"
    int VBlankingTimeInUsInTriggerMode; // VBlanking(in reg) time in "us"
    int DMABufferPreFetchWaitTimeInUs;  // Prefetching threshold in "us"
} TFrameParameter;
```

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

// Import functions:
SDK_API SSClassicUSB_InitDevice( void );
SDK_API SSClassicUSB_UnInitDevice( void );
SDK_API SSClassicUSB_GetModuleNoSerialNo( int DeviceID, char *ModuleNo, char *SerialNo);
SDK_API SSClassicUSB_AddDeviceToWorkingSet( int DeviceID );
SDK_API SSClassicUSB_RemoveDeviceFromWorkingSet( int DeviceID );
SDK_API SSClassicUSB_StartCameraEngine( HWND ParentHandle, int CameraBitOption, int ProcessThreads, int
IsCallBackInThread );
SDK_API SSClassicUSB_StopCameraEngine( void );
SDK_API SSClassicUSB_SetUSBConnectMonitor( int  DeviceID, int MonitorOn );
SDK_API SSClassicUSB_SetUSB30TransferSize( int TransferSizeLevel );
SDK_API SSClassicUSB_GetCameraFirmwareVersion( int DeviceID );
SDK_API SSClassicUSB_StartFrameGrab( int DeviceID, int TotalFrames );
SDK_API SSClassicUSB_StopFrameGrab( int DeviceID );
SDK_API SSClassicUSB_ShowFactoryControlPanel( int DeviceID, char *passWord );
SDK_API SSClassicUSB_HideFactoryControlPanel( void );
SDK_API SSClassicUSB_SetBayerFilterType( int DeviceID, int FilterType );
SDK_API SSClassicUSB_SetCameraWorkMode( int DeviceID, int WorkMode );
SDK_API SSClassicUSB_SetCustomizedResolution( int deviceID, int RowSize, int ColSize, int Bin, int BinMode );
SDK_API SSClassicUSB_SetExposureTime( int DeviceID, int exposureTime );
SDK_API SSClassicUSB_SetXYStart( int DeviceID, int XStart, int YStart );
SDK_API SSClassicUSB_SetGains( int DeviceID, int RedGain, int GreenGain, int BlueGain );
SDK_API SSClassicUSB_SetGainRatios( int DeviceID, int RedGainRatio, int BlueGainRatio);
SDK_API SSClassicUSB_SetColumnGain( int DeviceID, in; ColumnGain );
SDK_API SSClassicUSB_SetGamma( int DeviceID, int Gamma, int Contrast, int Bright, int Sharp );
SDK_API SSClassicUSB_SetBWMode( int DeviceID, int BWMode, int H_Mirror, int V_Flip );
SDK_API SSClassicUSB_SetMinimumFrameDelay( int IsMinimumFrameDelay );
SDK_API SSClassicUSB_SoftTrigger( int DeviceID );
SDK_API SSClassicUSB_SetSensorFrequency( int DeviceID, int Frequency );
SDK_API SSClassicUSB_SetSensorBlankings( int DeviceID, int HBlanking, int VBlanking );
SDK_API SSClassicUSB_SetSensorMode( int DeviceID, int SensorMode );
SDK_API SSClassicUSB_SetTriggerBurstCount( int DeviceID, int BurstCount );
SDK_API SSClassicUSB_ResetTimeStamp( int DeviceID );
SDK_API SSClassicUSB_InstallFrameHooker( int FrameType, FrameDataCallBack FrameHooker );
SDK_API SSClassicUSB_InstallUSBDeviceHooker( DeviceFaultCallBack USBDeviceHooker );
SDK_POINTER_API SSClassicUSB_GetCurrentFrame( int FrameType, int DeviceID, unsigned char* &FramePtr );
SDK_POINTER_API2 SSClassicUSB_GetCurrentFrame16bit( int FrameType, int DeviceID, unsigned short*
&FramePtr );
SDK_POINTER_API3 SSClassicUSB_GetCurrentFramePara( int DeviceID, unsigned long* &FrameParaPtr );
SDK_API SSClassicUSB_GetDevicesErrorState();
SDK_API SSClassicUSB_IsUSBSuperSpeed( int DeviceID );
SDK_API SSClassicUSB_SetGPIOConfig( int DeviceID, unsigned char ConfigByte );
SDK_API SSClassicUSB_SetGPIOOut( int DeviceID, unsigned char OutputByte );
SDK_API SSClassicUSB_SetGPIOInOut( int DeviceID, unsigned char OutputByte, unsigned char *InputBytePtr );
```

Important Notes for SMX USB3.0 camera SDK:

1). Sensor output bandwidth management:

For achieving maximum frame rate of a system, user has to consider the bandwidth of each major element in the data transfer chain, mainly, those elements include:

a). Sensor output bandwidth
b). On-camera processing bandwidth
c). Data transfer (USB communication) bandwidth
d). Host hardware bandwidth
e). Host software bandwidth, including the processing bandwidth of user's software.

Basically, as SMX camera uses a direct DMA which routes data directly from sensor to USB FIFO, Item (b) is not a concern, and for most current Host hardware (e.g. USB host controller is based PCIe v2.0+), host hardware (item (d)) is not a concern either.

However, the USB bandwidth (item(c)) and the Host processing bandwidth (item(e)) are very dynamic from application to application. E.g. there might be multiple cameras share the USB bandwidth, or some software needs complicated data processing for each grabbed frame.

To achieve maximum frame rate on various setups, a good management of the sensor output bandwidth (item(a)) is very critical to user's applications. As this is a key step for user to control the actual data bandwidth in the data chain to make sure there're no corrupted frames generated and dropped.

In this SDK we provide the follow parameters to allow user to control the sensor output bandwidth:

*. Sensor Clock: we provide low (24MHz), medium (48MHz) and high (96MHz) clocks.
*. HBlanking and VBlanking: VBlanking is the interval between frames. HBlanking is the internal between rows in a frame, setting to a longer interval will slow down the output data rate of a sensor.

The rule of controlling the sensor output bandwidth is to make it lower than the current available USB bandwidth, as otherwise corrupted frames might be generated and have to be dropped. We suggest user to do an USB bandwidth test with his actual application setup. For application with one camera on an USB3.0 port, this is usually not needed, as USB3.0 bandwidth is enough for one camera in most cases (however, if the camera is set to maximum resolution and in 16bit mode, it's still possible that the image outputting is faster than USB transferring). User might do the following:

*. If user connects a camera to an USB2.0 port, it's suggested to set the sensor clock to medium (typically 48MHz)
*. If user connects a camera to an USB3.0 port, user can set the sensor clock to high (typically 96MHz)
*. User might start with a conservative HBlanking and VBlanking value, the HBlanking value range is 0 – 48 which means the fastest to the slowest, while VBlanking is defined in a "50us" unit, e.g. set 20 to VBlanking means 1000us VBlanking interval time.

For example, with Mightex demo software ("SSClassicCameraApp.exe"), user might set HBlanking to 8 and VBlanking to 80 to start with and observe the frame rate (for a certain resolution which user intends to use, note that user should set a small ET, as otherwise the frame rate will be limited by the Exposure Time), then reducing the HBlanking to see the increasing of the frame rate, if the frame rate is not increased (or even reduced), user should not reduce the HBlanking anymore. And similarly, user might reduce the VBlanking and see the increasing of the frame rate, otherwise, user should not reduce VBlanking anymore.

Note that in above test, user might set "Grab Frame Type" (in "Global Controls" tab of the main window of SSClassicCameraApp.exe) to RAW, which will minimize the data processing. If user's software needs heavy data processing, user might have to either find a powerful host or slow down the sensor output further.

The current camera engine sets the clock to 48MHz and HBlanking/VBlanking to 1/60 as default, for user's own software user should set proper values for them.

2). Parameters for StartCameraEngine() API

Comparing with the StartCameraEngine()API of other Mightex cameras, the API for SMX cameras needs two more arguments: *ProcessThreads* and *IsCallBackInThread*, this is quite related to the design the of the SMX camera engine.

Basically, the camera engine is a multiple threads module which includes the thread of:

*. **Camera threads:** Those higher priority threads are responsible for communicate with cameras, the number of the threads equals to the active cameras in working set, e.g. for an application with 2 cameras, there're 2 camera threads. Thus for multiple camera application, it's highly recommended to use a multi-core CPU, e.g. for an application with 2 cameras, a dual core CPU has to be used, for an application with 4 cameras, a quad core CPU has to be used.

*. **Processing threads**: Those threads are used for convert grabbed RAW image data to DIB image data. The number of the threads is defined by *ProcessThreads*, basically, user should set it to at least 2, for a dual core CPU, it should be set to 4 or higher.

*. **Callback Thread**: This is a working thread for calling user's installed callback when the *IsCallBackInThread* is set to "1". Otherwise this thread is NOT created, and user's callback is invoked in the windows message handler of the factory panel window. So when *IsCallBackInThread* is set to "0", the callback really depends on the windows message loop. Note that when *IsCallBackInThread* is set to "1", the factory panel is NOT created, thus APIs of *SSClassicUSB_ShowFactoryControlPanel()* and *SSClassicUSB_HideFactoryControlPanel()* should not be used.

3). USB Connection Watch Dog

The firmware has an USB connection watch dog, when it's ON, it will monitor the USB communication between the camera and the host, if there's no any USB handshaking in a certain timeout interval (~3 seconds), it will disconnect the camera from the USB bus, connect back after 2 seconds and re-start.

By default, the watch dog is off when camera starts, host has to use the "**SSClassicUSB_SetUSBConnectMonitor(deviceID, 1)"** API to ask the camera to turn it on. And similarly, host can turn it off with the same API by setting the second argument to "0".

As the USB connection between host and camera only exists after the camera engine is active, so user might have the following calling sequence when starting the host software:

```
SSClassicUSB_InitDevice();      // Get the devices
SSClassicUSB_AddCameraToWorkingSet( deviceID); // Adding cameras to "working set".
SSClassicUSB _StartCameraEngine();
SSClassicUSB_SetUSBConnectMonitor( deviceID, 1 ); // Turn the watch dog on.
…… Operations ……
```

And when application terminates, it usually does:

```
SSClassicUSB_SetUSBConnectMonitor( deviceID, 0 ); // Turn the watch dog off.
SSClassicUSB _StopCameraEngin();
SSClassicUSB _UnInitDevice()
```

It's important to turn it off (if it's turned on) before stopping camera engine, otherwise the camera will reset itself because the camera engine isn't there anymore.

The watch dog might be helpful to recover from physical USB connection issues, when there's any USB connection issue, the camera engine might be blocked and in such case, the USB handshaking is stopped and that will trigger the watch dog to restart after ~3 seconds.. On host side, a device fault callback is recommended to be installed, thus the host software will be notified by the callback, user might do camera cleaning job in the device fault callback as following:

```
Void USBDeviceFaultCallBack( int deviceID, int deviceFault )
{
    SSClassicUSB_StopCameraEngin();
    SSClassicUSB _UnInitDevice()
}
```

In the DeviceFaultCallBack, the "SSClassicUSB_SetUSBConnectMonitor(deviceID, 0);" should not be invoked before stop camera engine, actually, in most cases, even user calls this API, the camera will still reset itself as the "turn off" USB command (and other USB commands) can't be sent to camera when there's low level error.

User's host software might have auto-recovery feature based on the watch dog and the device fault callback, e.g. after the device fault callback is invoked, user's codes might wait for a while (to let the camera to restart) and re-start the camera engine.

// Please check the header file included in the CDROM release for the latest information, we may add functions from time to time.

EXPORT Functions:

SSClassic_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 invoke **SSClassicUSB_ShowFactoryControlPanel()** and **SSClassicUSB_HideFactoryControlPanel()** to show and hide this window in user's application.

SDK_API SSClassicUSB_InitDevice(void);

This is first function user should invoke for his software, this function communicates with the installed device driver and reserves resources for all further operations.
Arguments: None
Return: The number of Mightex SMX cameras (SMN and SME) currently attached to the USB Bus (USB2.0 or USB3.0), if there's no Mightex SMX USB camera attached, the return value is 0.
Note: There's **NO** device handle needed for calling further SDK APIs, after invoking SSClassicUSB_InitDevice, camera engine reserves resources for all the attached SMX cameras. For example, if the returned value is 2, that 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 *SSClassicUSB_AddCameraToWorkingSet(deviceID)* to set the camera as active.
Important: The device drivers and camera engines are different for SMX cameras from other Mightex USB2.0 cameras(e.g. B-series, M-series or S-serires cameras..etc.), those USB2.0 cameras are **not** handled by the SMX camera engine, so they will not present in the return number.
Another important point is that re-invoking of this API while the first one is not ended with "*SSClassicUSB _UnInitDevice()*" will always return 0.

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

```
SSClassicUSB_InitDevice();     // Get the devices
SSClassicUSB_AddCameraToWorkingSet( deviceID); // Adding cameras to "working set".
SSClassicUSB _StartCameraEngine();
…… Operations ……
```

When application terminates, it usually does:

```
SSClassicUSB _StopCameraEngin();
SSClassicUSB _UnInitDevice()
```

SDK_API SSClassicUSB_UnInitDevice(void);

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

SDK_API SSClassicUSB_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 *SSClassicUSB_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)
-2: The camera engine is started already.

1: if the call succeeds.

Important: This API has to be invoked when the camera engine is NOT started (or is stopped), usually, user should use this API to get Module/Serial No of a camera right after invoking *SSClassicUSB_InitDevice()*, and keep the ModuleNo/SerialNo in variables for future uses.

SDK_API SSClassicUSB_AddDeviceToWorkingSet(int DeviceID);

For a present device, user might add it to the "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 *SSClassicUSB_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 cameras in "Working Set".

Important: This API has to be invoked when the camera engine is NOT started (or is stopped), When user adds a device in working set by invoking *SSClassicUSB_AddDeviceToWorkingSet(int DeviceID)***,** the camera is activated when the camera engine is started.

SDK_API SSClassicUSB_RemoveDeviceFromWorkingSet(int DeviceID);

User might remove the camera from the "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 *SSClassicUSB_InitDevice()* function for it.

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

Important: This API has to be invoked when the camera engine is NOT started (or is stopped), When user removes a device from working set by invoking *SSClassicUSB_RemoveDeviceFromWorkingSet(int DeviceID),* the camera is always "dummy" when the camera engine is started.

SDK_API SSClassicUSB_StartCameraEngine(HWND ParentHandle, int CameraBitOption, int ProcessThreads, int IsCallBackInThread) ;

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 before all the following camera related operations

Argument: ParentHandle – The window handle of the main form of user's application, as the engine might rely 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 can be 8 for 8bit mode and 16 for 16bit mode.

ProcessThreads – Number of processing threads created by camera engine, please refer to the description in the "**Important Notes for SMX USB3.0 camera SDK** Important Notes" for the explanation of this argument.

IsCallBackInThread – Whether user's installed callback is invoked from a working thread OR windows message handler, please refer to the description in the "**Important Notes for SMX USB3.0 camera SDK**" for the explanation of this argument.

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" after the camera engine has started. It's expected user to arrange camera working set properly and then start the camera engine.

SDK_API SSClassicUSB_StopCameraEngine(void);

This function stops the started camera engine.
Argument: None.
Return: -1 If the function fails (e.g. The camera engine is not started yet)
1 if the call succeeds.

SDK_API SSClassicUSB_SetUSBConnectMonitor(int DeviceID, int MonitorOn);

The firmware has an internal USB hand-shaking watch dog, it communicates with USB host in a certain interval (hand-shaking), if there's no hand shaking for more than 3 seconds, the camera will disconnect itself from the USB bus, re-connect back and starts (a hardware reset).
Argument: *DeviceID* – the device number which identifies the camera.
MonitorOn – 0: Handshaking watchdog is off, 1: Handshaking watchdog is on.
Return: Negative value: If the function fails (e.g. invalid device number)
1 if the call succeeds.
Important: The hand shaking part on host is implemented in the Camera Engine (DLL), when camera engine quits (StopCameraEngine()) and watch dog is still ON on camera side, the camera will restart.
This watch dog is turned off by default on camera when camera starts, user might turn it on with the above API, usually, user might do the following:

```
SSClassicUSB_InitDevice();     // Get the devices
SSClassicUSB_AddCameraToWorkingSet( deviceID); // Adding cameras to "working set".
SSClassicUSB _StartCameraEngine();
SSClassicUSB_SetUSBConnectMonitor( deviceID, 1 ); // Turn the watch dog on.
…… Operations ……
```

When application terminates, it usually does:

```
SSClassicUSB_SetUSBConnectMonitor( deviceID, 0 ); // Turn the watch dog off.
SSClassicUSB _StopCameraEngin();
SSClassicUSB _UnInitDevice()
```

It's important to turn it off (if it's turned on) before stopping camera engine, otherwise the camera will reset itself once because the camera engine isn't there anymore.
This API might be helpful to recover from USB connection issue, when there's any USB connection issue, the camera engine might be blocked and in such case, the handshaking is stopped and that will trigger the watch dog soon.. On host side, a device fault callback should be installed, and it will be invoked in such case, user might do camera cleaning job in the device fault callback as following:

```
Void USBDeviceFaultCallBack( int deviceID, int deviceFault )
{
    SSClassicUSB _StopCameraEngin();
    SSClassicUSB _UnInitDevice()
}
```

In the DeviceFaultCallBack, the "SSClassicUSB_SetUSBConnectMonitor(deviceID, 0);" should NOT be invoked, actually, in most cases, even user calls this API to turn off the watch dog, the camera will still reset itself as the "turn off" USB command can't be sent to camera.
In user's software, user might arrange the software to re-start the camera initialization and setups…etc. in this case and that will make the application recover from the USB connection error condition.

SDK_API SSClassicUSB_SetUSB30TransferSize(int TransferSizeLevel);

This function set the maximum transfer block size.
Argument: TransferSizeLevel -- it's from 0 to 4, 0 means minimum transfer size, 4 means maximum.
Return: -1 If the function fails (e.g. The camera engine is not started yet)
1 if the call succeeds.
Note: In most cases, it should be set to 4 (it's the default setting) which provides the maximum USB bandwidth efficiency, however, for some earlier USB3.0 Host controllers from **Etron** or **ASMedia**, the device driver of such

Host controller doesn't allow a bulk transfer block more than a certain limit (e.g. 1Mbytes), in such case, user might have to set the Level to 1 or even 0.
User might test such with the demo software first, in the "factory panel", there's a slide bar allow user to set different TransferSizeLevel, the default is maximum (4), if it doesn't work properly (usually the issue will show up when setting to high resolutions), user might try with the smaller level.

SDK_API SSClassicUSB_GetCameraFirmwareVersion(int DeviceID);

This function returns the firmware version of the specified camera.
Argument: DeviceID – the device number which identifies the camera
Return: -1 If the function fails (e.g. The camera engine is not started yet)
Other: the firmware version of the camera, it's in 0x00MMNNRR format, e.g. if the return value is 0x00010102, the firmware version is 1.1.2.

SDK_API SSClassicUSB_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 (by invoking *SSClassicUSB_StartFrameGrab()*).
TRIGGER mode – Camera will only grab a frame while there's an external trigger asserted OR host sends a Soft Trigger command (with the API of *SSClassicUSB_SoftTrigger()*). For Software with camera working in **TRIGGER** mode, it's recommended that user use *SSClassicUSB_StartFrameGrab(deviceID, 0x8888)* to grab frames forever.

SDK_API SSClassicUSB_StartFrameGrab(int DeviceID, int TotalFrames);
SDK_API SSClassicUSB_StopFrameGrab(int DeviceID);

When camera engine is started, the engine prepares all the resources, but it does **NOT** start the frame grabbing yet, until *SSClassicUSB_StartFrameGrab()* function is invoked. After it's successfully called, camera engine starts to grab frames from the specified cameras. User may call *SSClassicUSB_StopFrameGrab()* to stop the engine from grabbing frames from the specified cameras.
Argument:
DeviceID – the device number from which camera engine will start/stop to grab frames, user might set it to 0x88(ALL_DEVICES) for grabbing/stopping frames from all cameras in working set.
TotalFrames – This is for *SSClassicUSB_StartFrameGrab()* only, after grabbing frames of this number from the specified camera, the camera engine will automatically stop grabbing. If the DeviceID is ALL_DEVICES, the camera engine will grab the specified number from each camera in the working set. If user doesn't want it to be stopped, set this number to 0x8888, this means to grab frame forever, until user calls *SSClassicUSB_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 SSClassicUSB_ShowFactoryControlPanel(int DeviceID, char *passWord);

For user to access the factory features conveniently and easily, the library provides the dialog window which has all the features on it. Note that there's no Factory Control Panel created when user set the "*IsCallBackInThread*" to 1 when starting the camera engine. In such case, the call will still return 1 but doing nothing.
Argument: DeviceID – the device number.

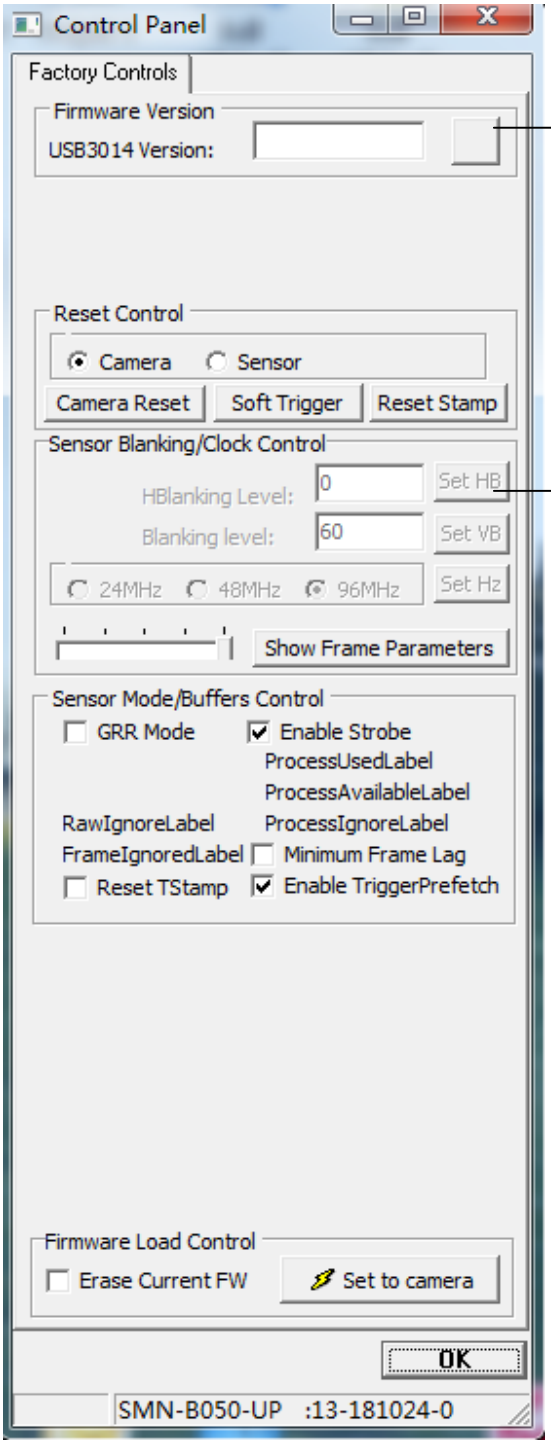
Password – A pointer to a string which is the password, “661016” is recommended in most cases. Two special password is that:

“123456” is used for showing the Hot Pixel Calibration panel.

“234567” is used for showing firmware upgrade control.

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.

The Slide bar in the left of the "Show Frame Parameters" button is used for setting Transfer Size, for **Etron** or **ASMedia** USB3.0 Host Controller, user should set it to 0 or 1 (it has 5 steps, from 0 – 4, the most right step as shown now is 4).
The "Show Frame Parameters" button will show a dialog box which has all the timing parameters of the current frame setting.

For B050/C050 Camera, User can also set GRR/ERS mode of the sensor shutter, GRR is only applicable when the camera is set to TRIGGER mode.
User can also disable/enable the Strobe out signal here
The "xxxLabel" items are for service purposes only

For upgrade the firmware, it's hidden in normal operations, it will show up when user follow the steps in the "firmware upgrade guide.pdf" (which will only be sent to user when there's a new firmware to be upgraded).

SDK_API SSClassicUSB_HideFactoryControlPanel(void);

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

SDK_API SSClassicUSB_SetCustomizedResolution(int deviceID, int RowSize, int ColumnSize, int Bin, int BinMode)

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, 2: 1:4 decimation mode
BinMode – 0: Skip mode for decimation, 1: Bin mode for decimation
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 multiple of a certain constants.
1 if the call succeeds.
Important:
RowSize and ColumnSize might be confused sometime, it can be described with an example, e.g. for a 1280x1024 resolution, the RowSize is 1280 and the ColumnSize is 1024.
The actual RowSize/ColumnSize are affected by the "Bin" parameter, so we have:
ActualRowSize = RowSize/BinFactor[Bin]; // BinFactor[3] = (1, 2, 4);
ActualColumnSize = ColumnSize/BinFactor[Bin];
There're several rules for the ActualRowSize and ActualColumnSize:
*. ActualRowSize must be multiple of 64, ActualColumnSize must be multiple of 16.
*. Minimum ActualColumnSize is 32 (for 16bit mode), or 64 (for 8bit mode).
`An example:`
SSClassicUSB_SetCustomizedResolution(deviceNO, 1280, 1024, 1, 0);
Set the camera (deviceNo) to 1280x1024, 1:2 skip decimation enabled.
Note： ForB012/C012 camera, the frame rate isn't improved when setting to 1:2 decimation mode.

SDK_API SSClassicUSB_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.

SDK_API SSClassicUSB_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 SSClassicUSB_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: DeviceID – 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: For **B050/C050** cameras, 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 **B012/C012** cameras, the gain value is from 1 – 64, represents 0.125 – 8x of digital amplifying multiples.
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 no more than 32 (4x).

SDK_API SSClassicUSB_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 **SSClassicUSB_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 SSClassicUSB_ColumnGain(int deviceID, int ColumnGain);

This API is applicable **for B012/C012 camera only**, this is analog gain applied to each column of the sensor, thus it's a global analog gain to the sensor
Argument: DeviceID – the device number which identifies the camera will be operated.
ColumnGain – 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 – 4, represents 1x, 2x, 4x and 8x of analog amplifying Multiples.
Note: In most applications, user should set the ColumnGain to 1, otherwise the noise is quite visible.

SDK_API SSClassicUSB_SetGamma(int DeviceID, 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:
DeviceID – Identifies the camera will be operated
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 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 SSClassicUSB_SetBWMode(int DeviceID, 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:

DeviceID – the device number which identifies the camera will be operated
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 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 SSClassicUSB_SetMinimumFrameDelay(int IsMinimumFrameDelay);

User may set minimum frame delay with this API. Note that the flag is global for all the active cameras.

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.

Note: User should NOT enable "Minimum Frame Delay" when the camera is in TRIGGER mode.

SDK_API SSClassicUSB_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 invalid)
1 if the call succeeds.

Note: User can use this API to simulate a trigger while the camera is in **TRIGGER** mode.

SDK_API SSClassicUSB_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 – 24: Slow clock, 48: Normal clock, 96: Fast clock.

Return: -1 If the function fails. (e.g. DeviceID is out of range)
1 if the call succeeds.

Note: Although the value is 24, 48 and 96, the actual sensor clock might not be exactly 24MHz, 48MHz and 96MHz. (e.g. for B012/C012 camera, the sensor clock is set to 19MHz, 37MHz and 74MHz really)

SDK_API SSClassicUSB_SetSensorBlankings(int DeviceID, int; hblanking, int vblanking);

User may user this API to set the HBlanking and VBlanking of the sensor.
Argument: DeviceNo – the device number which identifies the camera will be operated.
hblanking – 0 - 48 for hblanking levels, 0 means shortest and 48 means longest.
vblanking – the time in 50us unit for vblanking, e.g. 20 means 1000us.
Return: -1 If the function fails. (e.g. DeviceID is out of range)
1 if the call succeeds.

SDK_API SSClassicUSB_SetSensorMode(int DeviceID, int SensorMode);

User may user this API to set the Sensor Mode of the sensor.
Argument: DeviceNo – the device number which identifies the camera will be operated.
SensorMode – it's a bit map argument as following:
Bit0 – for GRR/ERS option, 0 – ERS shutter, 1 – GRR shutter
Bit1 – For Strobe Enable/Disable option, 0 – Disable, 1 – Enable
Bit2 – For "Trigger Reset Time Stamp", 0 – Disable (default), 1 - Enable
Return: -1 If the function fails. (e.g. DeviceID is out of range)
1 if the call succeeds.
Note: 1). GRR/ERS selection are for B050/C050 camera only, for B012/C012 camera, it's always Global Shutter. And note ***that GRR is only applicable when the camera is in TRIGGER mode.***
2). When Bit2 is set, the camera will reset the frame time stamp(in both "NORMAL" and "TRIGGER" mode) when there's a valid external trigger signal.

SDK_API SSClassicUSB_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 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 WorkMode; // 0 - NORMAL mode, 1 - TRIGGER mode
   int SensorClock;  // 24, 48, 96 for 24MHz, 48MHz and 96MHz
   int Row; // It's ColSize, in 1280x1024, it's 1024
   int Column; // It's RowSize, in 1280x1024, it's 1280
   int Bin; // 0, 1, 2 for no-decimation, 1:2 and 1:4 decimation
   int BinMode;// 0 - Skip, 1 - Bin
   int CameraBit; // 8 or 16.
   int XStart;
   int YStart;
   int ExposureTime; // in 50us unit, e.g. 100 means 5000us(5ms)
   int RedGain;
   int GreenGain;
   int BlueGain;
   int TimeStamp; // in 1ms unit, 0 - 0xFFFFFFFF and round back
   int SensorMode;// Bit0 is used for GRR mode, Bit1 is used for Strobe out enable
   int TriggerOccurred; // For NORMAL mode only, set when trigger occurred during the grabbing of last frame.
   int TriggerEventCount; // Reserved.
   int FrameSequenceNo; // Reserved.
   int IsFrameBad; // Is the current frame a bad one.

   int FrameProcessType; // 0 - RAW, 1 - BMP
```

```
    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 field identifies which camera in working set generates the frame.

WorkMode – The work mode (NORMAL or TRIGGER) when the frame is grabbed.

SensorClock – The clock of the sensor when the frame is grabbed.

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, BinMode – Bin shows 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 or 2 (means 1:2 or 1:4 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[3] = (1, 2, 4);
BinMode is used for "Skip" mode or "Average" mode when Bin is enabled.

CameraBit – 8 or 16, the camera bit mode.

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 – 4294967296 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.

SensorMode – Bit0 and Bit1 identifies the GRR/ERS shutter mode and Strobe Output Enable.

TriggerOccurred – For NORMAL mode only, set when trigger occurred during the grabbing of the frame..

TriggerEventCount – Reserved.

FrameSequenceNo – Reserved.

IsFrameBad – Set if the frame contains corrupted image data, if this field is set, this frame should be ignored.

FrameProcessType – 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).

FileterAcceptForFile – Reserved.

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 8bit Mode

OR

Unsigned char Pixels[Actual_Row][Actual_Column[2]; // For 16bit Mode

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).

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 figured 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 color camera it's RGB24 format, for Monochrome camera, the data returned in BMP mode is RGB8), the BMP mode data is as following:
(BytePtr points to a memory block which contains the frame data, the memory block is a continuous memory which is actually a 1-D array, the following 2-D or 3-D array is for description of the memory layout of the memory block only)

. 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 when the "IsCallBackInThread" is set to "0" in the StartCameraEngine() API, in this case, camera engine relies on Windows message loop to invoke the callback function. GUI operations are allowed in this callback function.
When "IsCallBackInThread " is set to "1", the callback is invoked in a working thread and usually GUI operations are not recommended. In both cases, blocking this callback will slow down the camera engine and 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 cameras.

Important: when the "IsCallBackInThread" is set to 0, as the frame callback is invoked actually by a windows timer (which relies on WM_TIMER message) in camera engine, it's important to let windows message loop running (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.
    }
}
```

OR user might set the "IsCallBackInThread" is set to 1 to have a working thread to invoke the callback, in such case, user has to pay attention to the possible data synchronizations of multiple threads.

SDK_API SSClassicUSB_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 device fault 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 DeviceBitMap, int FaultType);*
DeviceBitMap – the cameras which causes the device fault, it's bit OR for multi devices, Bit(0) means the first device...and so on.
FaultType – it is always ZERO in current version.

Note: The installed device callback might also be invoked if Host has very limited resources for grabbing image data from camera, in this case, it's recommended for user to set longer VBlanking and HBlanking time OR set slower clock.

SDK_POINTER_API SSClassicUSB_GetCurrentFrame(int FrameType, int DeviceID, unsigned char* &FramePtr);
SDK_POINTER_API2 SSClassicUSB_GetCurrentFrame16bit(int FrameType, int DeviceID, unsigned short* &FramePtr);
Argument: FrameType – 0: Raw Data **(RAW mode)**
1: Processed Data. **(BMP mode)**
DeviceID -- The device to be grabbed from.
FramePtr -- Pointer to a memory block (described below)
Return: Pointer to the same memory block pointed by FramePtr.
NULL if the API fails.

In some cases, user might **not** want to register a frame callback to get the frame data, for example, there're some tools in which callback mechanism is not fully supported, the above two APIs can be used in such cases.
Note that these two APIs is not functional if user installed the frame callback already, so user might first un-install the callback (pass NULL as the FrameHooker) when using these two APIs.
These APIs are NOT recommended for tools callback mechanism is supported, as these APIs are 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 *SSClassicUSB_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 Paddings[256-sizeof(TprocessedDataProperty)];
   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 these APIs. This API returns NULL when it fails, a common cause of failure is the time out error, which means there's no frame grabbed in the preset time (~2 seconds), this usually occurs when the camera is in **TRIGGER** mode, as in **TRIGGER** mode, there might be no frame to be grabbed all the time. So when using these APIs in **TRIGGER** mode, it's recommended for user to check the return value of the API, and if it's NULL, user might immediately invoking this API again.
Note that the second API "**SSClassicUSB_GetCurrentFrame16bit(int FrameType, int DeviceID, unsigned short* &FramePtr);"** is for returning 16bit image data when the camera is set to 16bit mode, the return value and FramePtr points to the 16bit image data.

SDK_POINTER_API3 SSClassicUSB_GetCurrentFramePara(int DeviceID, unsigned long* &FrameParaPtr);

User may call this function to configure GPIO pins.
Argument : DeviceID – The device number, which identifies the camera is being operated.
FrameParaPtr – The pointer points to a data block described by *TFrameParameter*, it has multiple fields describing the time parameters of an image frame with the current setting.
Return: -1 If the function fails (e.g. invalid device number)
1 if the call succeeds.

SDK_API SSClassicUSB_GetDevicesErrorState();

Note that this API won't work if user installed a device fault 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
>0: 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.

SDK_API SSClassicUSB_IsUSBSuperSpeed(int DeviceID);

User may call this function to get the actual connecting USB speed.
Return: 0 If the USB connection speed is High Speed.
1 if the USB connection speed is Super Speed.
Negative Value: The API invoking fails (e.g. invalid DeviceID OR the camera engine is not started yet).

SDK_API SSClassicUSB_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 SSClassicUSB_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.