

Mightex USB Classic Camera SDK Manual

Version 1.1.4

Mar. 18, 2009

Relevant Products

Part Numbers
MCN-B013-U, GLN-B013-U, MLE-B013-U, MCE-B013-U, MCN-C013-U, GLN-C013-U, MLE-C013-U, MCE-C013-U, MCN-C030-U, GLN-C030-U, MLE-C030-U, MCE-C030-U MCN-B013-US, GLN-B013-US, MLE-B013-US, MCE-B013-US, MCN-C013-US, GLN-C013-US, MLE-C013-US, MCE-C013-US, MCN-C030-US, GLN-C030-US, MLE-C030-US, MCE-C030-US

Revision History

Revision	Date	Author	Description
1.0.0	2006.6.26	JT Zheng	Initial Revision
1.0.1	2006.7.18	JT Zheng	External Trigger Mode
1.0.2	2006.8.17	JT Zheng	GPIO feature added
1.0.3	2006.8.27	JT Zheng	TWAIN Support, BG/FG support
1.0.4	2006.9.11	JT Zheng	Multiple Cameras support
1.0.5	2006.9.22	JT Zheng	Description of LED pins
1.0.6	2006.10.9	JT Zheng	Add Module table, “About...” item
1.0.7	2006.11.30	JT Zheng	Maximum Rate Option
1.0.8	2007.2.27	JT Zheng	Recommended Intel USB Host Controller
1.0.9	2007.3.8	JT Zheng	Strobe Signal for 1.3M Camera
1.1.0	2007.4.9	JT Zheng	“No Module” description
1.1.1	2007.7.19	JT Zheng	Add “-US” modules
1.1.2	2008.11.28	JT Zheng	Limits for Open device
1.1.3	2009.1.6	JT Zheng	Add PnP Event supporting
1.1.4	2009.3 18	JT Zheng	Improved description for Exposure Time

Mightex USB 2.0 color camera is mainly designed for microscopy and other scientific applications, in which cost-effective and ease of use are important. With USB 2.0 high speed interface and powerful PC software processing, the camera delivers excellent quality images at high frame rate. GUI application and SDK are provided for user's application developments.

IMPORTANT:

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

SDK FILES:

The SDK includes the following files:

\LIB directory:

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

\Documents directory:

MighTex USB Camera SDK Manual.pdf

\Examples directory

\Delphi	--- Delphi 5.0 project example.
\VC++	--- VC++ 6.0 project example.

\Firmware directory: The latest firmware.

Note that these examples are for demonstration of the DLL functions only, device fault situations are not fully handled in these examples, user should handle them properly.

HEADER FILE:

The "MT_USBCamera_SDK.h" is as following:

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

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

#define RAWDATA_IMAGE 0
#define BMPDATA_IMAGE 1

#pragma pack(1)
typedef struct {
    int Revision;
    // For Image Capture
```

```

int Resolution;
int BinMode;
int XStart;
int YStart;
int GreenGain;
int BlueGain;
int RedGain;
int MaxExposureTimeIndex;
int ExposureTime;
// For Image Rendor
bool ImageRendorFitWindow;
int Gamma;
int Contrast;
int Bright;
int SharpLevel;
bool BWMode;
bool HorizontalMirror;
bool VerticalFlip;
// For Capture Files.
int CatchFrames;
bool IsAverageFrame;
bool IsCatchRAW;
bool IsRawGraph;
bool IsCatchJPEG;
bool CatchIgnoreSkip;
} TImageControl;
#pragma pack()

typedef TImageControl *PImageCtl;
typedef void (* CallBackFunc)( int ImageSequenceNo, char *FileName );

// Export functions:
SDK_API MTUSB_InitDevice( void );
SDK_API MTUSB_UnInitDevice( void );
SDK_HANDLE_API MTUSB_OpenDevice( int deviceID );
SDK_HANDLE_API MTUSB_ShowOpenDeviceDialog( void );
SDK_API MTUSB_GetModuleNo( DEV_HANDLE DevHandle, char *ModuleNo );
SDK_API MTUSB_GetSerialNo( DEV_HANDLE DevHandle, char *SerialNo );
SDK_API MTUSB_StartCameraEngine( HWND ParentHandle, DEV_HANDLE DevHandle );
SDK_API MTUSB_StopCameraEngine( DEV_HANDLE DevHandle );
SDK_API MTUSB_SetCameraWorkMode( DEV_HANDLE DevHandle, int WorkMode );
SDK_API MTUSB_SetExternalParameters( DEV_HANDLE DevHandle, bool AutoLoop, bool IsRawGraph;
    bool IsJPEG, char *FilePath, char *FileName);
SDK_API MTUSB_WaitingExternalTrigger( DEV_HANDLE DevHandle, bool StartWait, CallBackFunc Aproc );
SDK_API MTUSB_ShowFrameControlPanel( DEV_HANDLE DevHandle, bool IsTriggerModeAllow, bool
    CloseParent, char *Title, int Left, int Top);
SDK_API MTUSB_HideFrameControlPanel( DEV_HANDLE DevHandle );
SDK_API MTUSB_ShowVideoWindow( DEV_HANDLE DevHandle, int Top, int Left, int Width, int Height );
SDK_API MTUSB_StartFrameGrab( DEV_HANDLE DevHandle );
SDK_API MTUSB_StopFrameGrab( DEV_HANDLE DevHandle );
SDK_API MTUSB_GetFrameSetting( DEV_HANDLE DevHandle, PImageCtl SettingPtr);
SDK_API MTUSB_SetFrameSetting( DEV_HANDLE DevHandle, PImageCtl SettingPtr);
SDK_API MTUSB_SetResolution( DEV_HANDLE DevHandle, PImageCtl SettingPtr);
SDK_API MTUSB_SetStartPosition( DEV_HANDLE DevHandle, PImageCtl SettingPtr);
SDK_API MTUSB_SetGain( DEV_HANDLE DevHandle, PImageCtl SettingPtr);
SDK_API MTUSB_SetExposureTime( DEV_HANDLE DevHandle, PImageCtl SettingPtr);
SDK_API MTUSB_SetGammaValue( DEV_HANDLE DevHandle, PImageCtl SettingPtr);
SDK_API MTUSB_SetGammaTable( DEV_HANDLE DevHandle, unsigned char *GammaTable );
SDK_API MTUSB_SetShowMode( DEV_HANDLE DevHandle, PImageCtl SettingPtr);
SDK_API MTUSB_SetWhiteBalance( DEV_HANDLE DevHandle );
SDK_API MTUSB_SetFrameRateLevel( DEV_HANDLE DevHandle, int RateLevel );
SDK_API MTUSB_GetCurrentFrameRate( DEV_HANDLE DevHandle );

```

```

SDK_API MTUSB_GetLastBMPFrame( DEV_HANDLE DevHandle, char *FileName );
SDK_API MTUSB_GetCurrentFrame( DEV_HANDLE DevHandle, int FrameType, unsigned char *Buffer );
SDK_API MTUSB_SaveFramesToFiles( DEV_HANDLE DevHandle, PImageCtl SettingPtr,
                                char *FilePath, char *FileName );

```

....

// Please check the header file itself of latest information, as we're adding functions from time to time.

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

EXPORT Functions:

MT_USBCamera_SDK.dll exports functions to allow user to easily and completely control the various parameters of frame grabbing, image render and snapshot catching. For user's quick development of application, the DLL has three built in windows, which are:

- Mightex USB Camera Device Open Dialog Window
- Mightex USB Camera Full Control Panel Dialog Window
- Mightex USB Video Window

The first two windows are not necessarily used if user wants to have his own GUI for similar purposes, there're sets of other functions which provides equivalent features, however, by using these two windows, especially the “Control Panel” window, it's extremely easy and quick to develop an application, this is a time-saving solution.

SDK_API MTUSB_InitDevice(void);

This is first function user should call for his own application, this function communicates with the installed device driver and reserve resources for further operations.

Arguments: None

Return: The number of Mightex USB cameras currently attached to the USB 2.0 Bus, if there's no Mightex USB camera attached, the return value is 0.

SDK_API MTUSB_UnInitDevice(void);

This is the function to release all the resources reserved by MTUSB_InitDevice(), user should invoke it before application terminates.

Arguments: None

Return: Always return 0.

SDK_HANDLE_API MTUSB_ShowOpenDeviceDialog(void);

User may call this function to show the Device Open Dialog, which lets user to select the camera will be operated. The dialog is as following:



It will show all the attached cameras in ModuleNo:SerialNo format in the pull down combo box, user may select the one he wants to operate and click the [OK] button.

Argument: None.

Return: The handle of the opened device.

Note that the device handle returned is an index to the internal data structure, and it will be used as the first parameter for all other device operation functions. It returns 0xFFFFFFFF if this API is invoked while the camera engine is started already.

Important: The current device driver only support **ONE** opened device, opening a device will close the previous opened device automatically (if there's an opened one), so if there's more than one Mightex USB cameras attached, user have to switch between them for operations.

This API can only be invoked with camera engine stopped (or not started yet).

SDK_HANDLE_API MTUSB_OpenDevice(int deviceID);

If user doesn't want to use the previous Open Device Dialog for opening a selected device, user may use This function to open the device.

Argument: deviceID – this is the index of the device, it's a ZERO based index, for example, while invoking MTUSB_InitDevice() and it returns 2 (the number of devices currently attached), deviceID can be 0 or 1, means the first and the second device. It returns 0xFFFFFFFF if this API is invoked while the camera engine started.

Return: Device Handle

Important: See the notes on the previous function description for the device handle.

This API can only be invoked with camera engine stopped (or not started yet).

SDK_API MTUSB_GetModuleNo(DEV_HANDLE DevHandle, char *ModuleNo);

For an opened device, user might get its Module Number by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

ModuleNo – 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 handle), or if the camera engine is already started.
1 if the call succeeds.

Important: This API can only be invoked with camera engine stopped (or not started yet).

SDK_API MTUSB_GetSerialNo(DEV_HANDLE DevHandle, char *SerialNo);

For an opened device, user might get its Serial Number by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

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 handle), or if the camera engine is already started.
1 if the call succeeds.

Important: This API can only be invoked with camera engine stopped (or not started yet).

SDK_API MTUSB_StartCameraEngine(HWND ParentHandle, DEV_HANDLE DevHandle);

We have a multiple threads camera engine internally, which is responsible for all the frame grabbing, raw data to RGB data conversion...etc. functions. User MUST start this engine for all the following camera related operations

Argument: ParentHandle – The window handle of the main form of user's application, as the engine relies on Windows Message Queue, it needs a parent window handle which mostly should be the handle of the main window of user's application.

DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

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

Important: After starting the camera engine, the application will use considerable amount of the system resources (e.g. RAM), For the PC running the application, it's recommended to be Pentium IV, 1.5G or up and have 512M memory. (Please refer to product spec. for the minimum requirement of the PC).

SDK_API MTUSB_StopCameraEngine(DEV_HANDLE DevHandle);

This function stops the started camera engine.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

Return: -1 If the function fails (e.g. invalid device handle or if the engine is NOT started yet)
1 if the call succeeds.

Important: For properly operate the camera, usually the application should have the following sequence for device initialization and opening:

```
MTUSB_InitDevice(); // Get the devices
MTUSB_OpenDevice(); // Using the device index returned by the previous MTUSB_InitDevice() call.
MTUSB_GetModuleNo();
MTUSB_GetSerialNo();
MTUSB_StartCameraEngine(); // Using the device handle returned by MTUSB_OpenDevice()
..... Operations .....
MTUSB_StopCameraEngin();
MTUSB_UnInitDevice()
```

Note that we don't need to explicitly close the opened device, because:

- 1). If user want to open another device, open device will automatically close the previous opened device,
- 2). MTUSB_UnInitDevice() will close the opened device, and release all other resources.

SDK_API MTUSB_SetCameraWorkMode(DEV_HANDLE DevHandle, int WorkMode);

By default, the Camera is working in “**Video**” mode in which camera deliver frames to PC continuously, however, user may set it to “**External Trigger**” Mode, in which the camera is waiting for an external trigger signal and capture ONE frame of image.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

WorkMode – 0: Video Mode, 1: External Trigger Mode.

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

SDK_API MTUSB_SetExternalParameters(DEV_HANDLE DevHandle, bool AutoLoop, bool IsRawGraph, bool IsJPEG, char *FilePath, char *FileName);

While the camera is in “External Trigger” Mode, invoking this function set the parameters for external trigger mode.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

AutoLoop – Whether is “Automatic Loop” or only One Shot Wait.

IsRawGraph – If it's true, the saved BMP or JPEG file is the raw graph which is not adjusted by Gamma, Contrast, Bright or Sharp settings.

IsJPEG – Is the saved file in JPEG format (other than BMP format).

FilePath, FileName – the directory and name of the saved file. Note that for path, the ending “\” is NOT needed, for filename, the extension (jpg or bmp) is NOT needed.

Return: -1: If the function fails (e.g. invalid device handle or camera is NOT in External Trigger Mode or it's during waiting for external trigger)
1: Call succeeds.

Important: Invoking this function won't really start the waiting of the external trigger, this function only set the parameters for trigger mode, a sequential invoking of the following **MTUSB_WaitingExternalTrigger()** is needed to fulfill the whole external trigger services.

SDK_API MTUSB_WaitingExternalTrigger(DEV_HANDLE DevHandle, bool StartWaiting, CallBackFunc Aproc);

While the camera is in “External Trigger” Mode, invoking this function starting the waiting for external signal.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or

MTUSB_OpenDevice()

StartWaiting – True: Start to wait for external trigger signal, .

False: Abort the waiting state if it’s in it.

Aproc – Call back installed by caller, it’s invoked after a frame is caught.

Return: -1: If the function fails (e.g. invalid device handle or camera is NOT in External Trigger Mode)

1: Call succeeds.

Important: This function will return immediately, while an external trigger occurs and a frame is caught, the call back function will be invoked. If “AutoLoop” is NOT set, user has to invoke this function again (with StartWaiting set to True) to start next catch, If “AutoLoop” is set, camera engine will still in “waiting” state and the call back function will be invoked repeatedly after each frame is caught. The call back function has two output arguments: Frame Sequential number(make sense only when autoloop is set) and image file name (for the image just caught, it’s actually set by the above **MTUSB_SetExternalParameters()** function).

Calling this function with StartWaiting set to False will notify camera engine to abort the waiting if it’s still in “waiting” state (this can be either the “AutoLoop” is set, or although it’s one-shot mode, but there’s no image caught yet), in this case the call back will still be invoked with first argument (Frame sequential number) set to ZERO.

A brief summary for External mode:

The camera can be set in “External Trigger” Mode, the camera engine has the following behaviors in this mode:

AutoLoop is set to TRUE:

*In this case, the camera engine will always stay in “waiting” state, each time it captured a frame (triggered by external signal), camera engine invokes the callback (if it’s installed). The only way out of “waiting” state is to Invoke the **MTUSB_WaitingExternalTrigger()** function with the StartWaiting set to FALSE, this notifies The engine to exit from “waiting” state. Camera engine will also call the callback with the first argument set to ZERO in this case. After it’s aborted, the callback function will be unhooked from camera engine automatically, which means next time Host must re-install the hooker by calling **MTUSB_WaitingExternalTrigger()**.*

AutoLoop is set to FALSE

*In this case, Camera engine is in “One-Shot Waiting” state, which means it will wait until a frame is captured (triggered by an external signal) or user abort it. In both cases, the callback (if installed) will be invoked, and the callback will be unhooked from camera engine, user has to reinstall the callback next time by call the **MTUSB_WaitingExternalTrigger()** function.*

*If user wants to get frame in buffer (instead of saving in a file), user may install a frame callback hooker by **MTUSB_InstallFrameHooker()** function, and also doing the following:*

*1). While invoking **MTUSB_SetExternalParameters()** function, set the FileName to “DoNotSaveFile”, that will let camera engine NOT to save the captured image to a file.*

*2). While invoking **MTUSB_WaitingExternalTrigger()** function, set the Callback Aproc to NULL, which doesn’t install a hooker, as in the case (file isn’t saved), it’s no need to install a hooker here.*

SDK_API MTUSB_ShowFrameControlPanel(DEV_HANDLE DevHandle, bool IsTriggerModeAllow, bool CloseParent, char *Title, int Left, int Top);

For user to develop application conveniently and easily, the library provides its second dialog window which has all the camera controls on it, if user use this window in his application, it’s NOT necessarily to use most of other functions.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or

MTUSB_OpenDevice()

IsTriggerModeAllow – Set to control whether the Trigger Mode Selection is visible on control panel. We provide this parameter for user doesn’t want to have “External Trigger” mode available on control panel.

CloseParent – Set to TRUE if user wants to close the Parent Window of the control panel, while user click the [x] button of the panel, note that this usually closes the whole application.

Title – The Title will be displayed on the control panel.

Left, Top – the Top-Left position of the control panel.

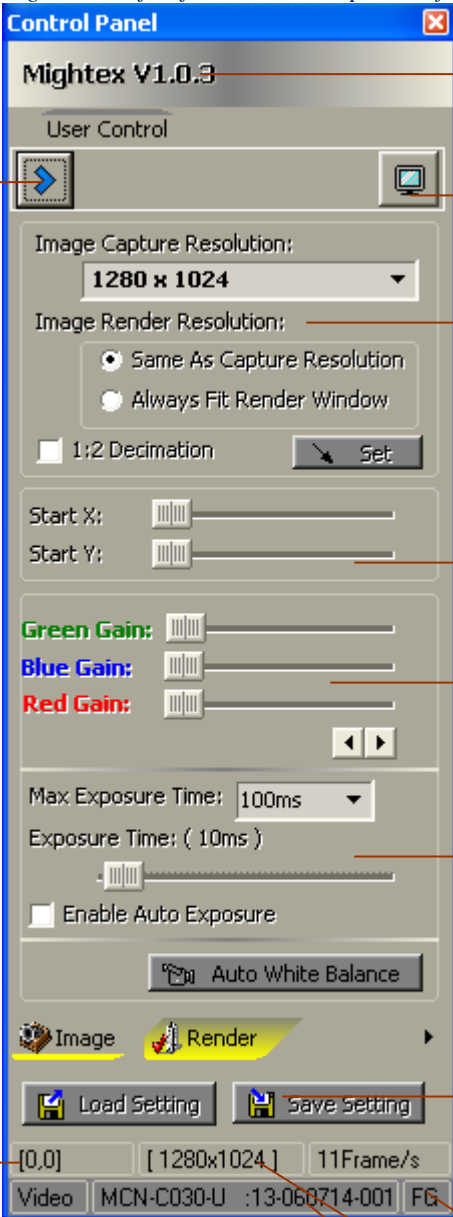
Return: -1 If the function fails (e.g. invalid device handle)

1 If the call succeeds.

Important: Close this control panel will close the whole application (it will post a message of SC_CLOSE to it’s parent window), so if user wants to have this control panel shown in application, the parent window is NOT necessarily visible. If user wants to hide this panel, don’t close it, but invoke **MTUSB_HideFrameControlPanel()** function instead.

The panel is as following*, in the “Image” page, it has the controls:

**Note that the panel might be modified for late versions, please refer to camera user manual for latest panel.*



The screenshot shows the 'Control Panel' window for 'Mightex V1.0.3'. The interface includes a 'User Control' section with a 'Start (and Stop) Frame Grab' button (a blue square with a right arrow) and a 'Show Video Window' button (a monitor icon). Below these are resolution settings: 'Image Capture Resolution' set to '1280 x 1024' and 'Image Render Resolution' with radio buttons for 'Same As Capture Resolution' (selected) and 'Always Fit Render Window'. There is a '1:2 Decimation' checkbox and a 'Set' button. Further down are 'Start X' and 'Start Y' sliders. Below those are 'Green Gain', 'Blue Gain', and 'Red Gain' sliders, along with a proportional adjustment button. The 'Max Exposure Time' is set to '100ms', and 'Exposure Time (10ms)' has a slider. An 'Enable Auto Exposure' checkbox is present. An 'Auto White Balance' button is also shown. At the bottom, there are 'Image' and 'Render' tabs, 'Load Setting' and 'Save Setting' buttons, and a status bar showing '[0,0]', '[1280x1024]', '11Frame/s', 'Video', 'MCN-C030-U', ':13-060714-001', and 'FG'.

Start (and Stop) Frame Grab button

The Title, I use “Mightex V1.0.0” as example

The “Show Video Window” button, click it show the Video window.

User may select the resolution here, currently, the provided resolution includes 32x32, ..., 640x480, 800x600, 1024x768 and 1280x1024(for 1.3M series), And the selectable “1:2 Decimation”(2x skip) mode. User may also select the resolution of rendering, it can be always fit the video window, OR always keep the same resolution as the capture image. User must use **Set** button to set the settings to camera engine. (Note that for minimum resolution 32x32, the 1:2 Decimation is not allowed)

User may use these two slider to select the start position of the capture image. (ROI feature) While it's NOT in full resolution.

User may use these three slider to manually adjust the RGB gains (0x – 16x). The **◀ ▶** is used for adjusting all gains (RGB gains) proportionally.

User may use these controls to select the maximum exposure time range and the current exposure time*, the **Auto White Balance** button is used for Automatic White Balance (AWB) set, user needs to set proper exposure time and put a white paper as the object, click this button will automatically set the RGB gains to get ideal white color. The “**Enable Auto Exposure**” checkbox allows user to enable auto exposure feature.

These two buttons are used to save/load all the current settings to/from a user defined file, user may set proper parameters (exposure time, gains...etc.) under a certain environment and save the parameters to a file named this environment, e.g. “Sunny Outside.set” or “In Room.set”, And user may load them back later.

Start Position of ROI

The Software is running in “Foreground”**.

Current selected resolution and frame rate.
The opening device’s Module No. and Serial No.

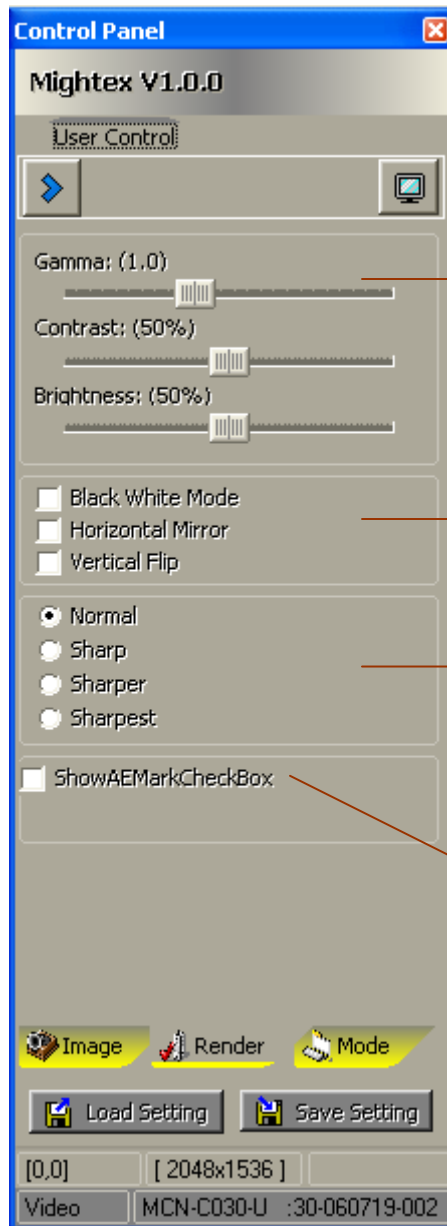
While the Video window is showing, user might choose “Same as capture resolution” or “Always fit the render window”, If it is chosen as “Same as Capture Resolution”, the Video window may not bigger enough to show the whole frame, user can move the image around by moving the mouse with the left button down.

*. *Exposure Time: There're 4 ranges totally, they're 0.05ms – 5ms, 0.1ms – 10ms, 1ms – 100ms and 7.5ms – 750ms, we expect in most cases, user should use 1ms – 100ms range. And while it's in 7.5ms – 750 ms range, the resolution is 7.5ms.*

Another point is that under certain lighting condition, there might be light flick from AC power ([220V@50Hz](#), and [110V@60Hz](#)), in this case, user should select proper Exposure time to avoid these flicks appearing in the Video window (though actually it won't be in still image anyway). Under 50Hz lighting condition, the exposure time should be a multiple of 1/100 second (10ms), and similarly, under 60Hz lighting condition, the exposure time should be set as a multiple of 1/120 second. Of course, it's recommended to use lighting source WITHOUT AC flicking

***. “Foreground” and “Background”: As the camera application does all the image processing on PC, it might use most of PC's CPU and Memory resources, however, this is only the case while it's in “FG” way, which means this application is the windows application get the focus (user's key or mouse input). While user operates on other applications, this application automatically turns to “BG”, which uses much less resources. Click on any of its windows will bring it back to “FG”.*

The control panel has the second page of “Render” as following:



Gamma, Contrast and Brightness control for the video window.

User may set the display frame (in video window) in “Black and White” mode, “Horizontal Mirror” mode and “Vertical Flip” mode by checking these boxes.

User may select the “Sharp” level, however, as the Sharp algorithm needs considerable PC resources, while selecting Sharp, Sharper or Sharpest, PC’s resource will be almost 100% occupied by the it. Not only the frame rate will reduce significantly and that will affect the running speed of other PC applications.

So it’s recommended to use “Normal” in most cases.

ShowAEMarkCheckBox will show the auto exposure detecting area on the image, while auto exposure feature is enabled, camera will detect the illumination of this area and figure the optimized exposure time.

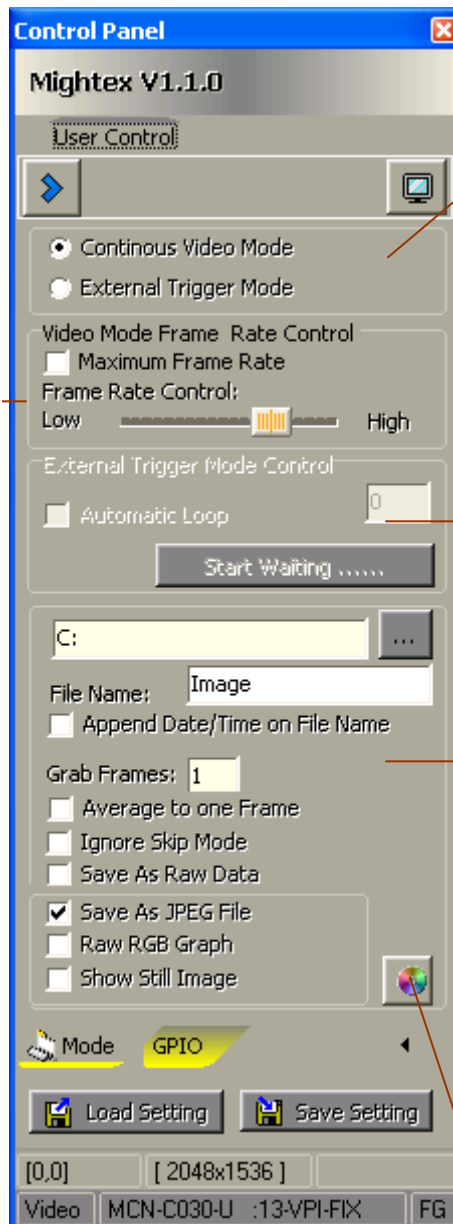
The control panel has the third page of “Mode” as following:

In Video Mode, the actual frame rate is very depending on the PC resources, while we set the default of each resolution to a moderate frame rate, that will give user the proper frame rate, it might use some CPU’s resources (for data collection from USB and image data processing), user may use this slider to adjust the frame rate to another level, to let PC has some more (or less) bandwidths for other applications. As different PC may have different frame rates under the same settings of this slider, this slider gives user a easy way to control on a particular PC..

The actual frame rate is shown on the pane below.

Note that this is only effective while the application is in Foreground, while it’s in Background, the application will release most of the resources for other Foreground application and the frame rate if very low.

“**Maximum Rate Control**” is an aggressive option to let user to get maximum frame rate, it’s **NOT** recommended to be turned on in most cases.



Camera Mode selection, the default is “Video” mode, user may also select “External Trigger” mode.

While camera is in “External Trigger” mode, user can click **Start Waiting** button, that will make software keeps to wait for an external signal (hard connected to the external trigger pins on camera’s connector), a falling edge of the trigger signal will start ONE frame of snapshot catching, and PC will save it to the specified location. If “Automatic Loop” is checked, the software will remain in the “waiting” state even after a frame is captured, waiting for the next capture. If “Automatic Loop” is NOT checked, it finishes the “waiting” after ONE capture, and user has to re-click the button for next capture.

The directory to store the captured files. While in “Video” mode, the camera engine continuously grabbing frames from camera, and user may ask it to save the grabbed frames to file in “Raw”, “JPEG” or “BMP” format. In “External Trigger” mode, only **ONE** frame in “JPEG” or “BMP” format can be saved for each external trigger.

User can specifies directory, filename and file number, as well as other settings:

“**Append Date/Time on File Name**” – Automatically append Date/Tim to file name.

“**Average to one frame**” – Save one image only, but it’s the average of all grabbing frames.

“**Ignore Skip Mode**” – user can check it if user wants to get a full resolution frame (e.g. 1280x1024), while the current video resolution is in 1:2 Decimation mode (e.g. 1280x1024 with 1:2 decimation).

“**Save As Raw Data**” – the Frame is save as RAW data file (Only valid for “Video” mode).

“**Save As JPEG file**” – Save file as JPEG image.

“**Raw RGB Graph**” – Ignore the Gamma/Contrast/Bright adjustments)

“**Show Still Image in Form**” – If this is checked, a still image window will be display after frames grabbing, and user can view them instantly. (Valid for “Video” mode only)

In “Video” mode, user can use this button for taking snapshots.

The GPIO page has the following:




For camera with LED driver, this LED brightness control will be shown, User can slide these bars to control the brightness of each LED channel or all 4 channels.

For camera with GPIO, this GPIO control will be shown, user can select each certain pin as Input or output (check the checkbox in “Set As Output” group), for the pin set as output, check it in the OutX box will set this pin to High, otherwise it’s output Low. The InX box shows the current input level, checked means input High, otherwise It’s Low.

Get Firmware Version Info.

“Trigger to Grab Frames in Video Mode” – while this is checked, a trigger signal occurrence will be an

equivalent to the  button click, this enables user to use an external device (e.g. push button) to grab frames in “Video” mode.

Important: After the camera engine is started, the control panel is created and hided, user can use this function to show it up, and it’s always a good idea to show it during the development time, even user don’t want it to be shown in final application GUI.

Please refer to the examples (VC++ or Delphi examples) for the using of the control panel, as well as other APIs.

SDK_API MTUSB_HideFrameControlPanel(DEV_HANDLE DevHandle);

This function hides the control panel, note the control panel is always there once the camera engine is started, hiding it only make it invisible.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

Return: -1 If the function fails (e.g. invalid device handle or if the engine is NOT started yet)
1 if the call succeeds.

SDK_API MTUSB_ShowVideoWindow(DEV_HANDLE DevHandle, int Top, int Left, int Width, int Height);


This function shows the video window, user may customize it's position and size with the input arguments.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

Left, Top – the Top-Left position of the video window.

Width, Height – The width and height of the video window.

Return: -1 If the function fails (e.g. invalid device handle or if the engine is NOT started yet)
1 if the call succeeds.

Important: On control panel, there's a button  corresponding to this function. The video window is shown as following:



Note that while the render resolution is chosen as "Same as Capture Resolution", the Video window may not be big enough to show the whole frame, user can move the image around by moving the mouse with the left button down.



SDK_API MTUSB_StartFrameGrab(DEV_HANDLE DevHandle, int TotalFrames);
SDK_API MTUSB_StopFrameGrab(DEV_HANDLE DevHandle);

When camera engine is started, in Video mode, the engine prepares all the resources, but it does NOT start the frame grabbing , until MTUSB_StartFrameGrab() function is invoked. After it's successfully called, user should see video on the video window (if it's showed). User may call MTUSB_StopFrameGrab() to stop the engine from grabbing frames from camera.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

TotalFrames – This is used in MTUSB_StartFrameGrab() only, after grabbing this frames, the camera engine will automatically stop frame grabbing, if user doesn't want it to be stopped, set this number to 0x8888, which will do frame grabbing forever, until user calls MTUSB_StopFrameGrab().

Return: -1 If the function fails (e.g. invalid device handle or if the engine is NOT started yet)
1 if the call succeeds.

Important: On control panel, there's a button  corresponding to "MTUSB_StartFrameGrab()", and after it's started the button becomes , click it invoking "MTUSB_StopFrameGrab()".

SDK_API MTUSB_GetFrameSetting(DEV_HANDLE DevHandle, PImageCtl SettingPtr);

User may get the current set of parameters by invoking this function, please note that the TImageControl data structure contains all the parameters for controlling Frame Grabbing, Video rendering and File savings.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

SettingPtr – the Pointer to variable of TImageControl structure.

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

Important:

- 1). There're two ways to change those settings, from the control panel OR by calling the following MTUSB_Setxxx functions, either way, the settings returned from this function are the current latest settings of the camera engine.
- 2). The TImageControl data structure has the following elements, please refer to the c style comments in /* */ for their definition:

```
typedef struct {
    int Revision; /* Reserved for internal use only */

    // For Image Capture
    int Resolution; /* This is an index to resolution settings, we have the following definition for it:
        0 – 32 x 32
        1 – 64 x 64
        2 – 160 x 120
        3 – 320 x 240
        4 – 640 x 480
        5 – 800 x 600
        6 – 1024 x 768
        7 – 1280 x 1024
        8 – 1600 x 1200 For 3M Camera only
        9 – 2048 x 1536 For 3M Camera only
        */
    int BinMode; /* 1 – No Skip mode, 2 – 2X skip or binning mode (1:2 decimation) */
    int XStart; /* Start Column of the ROI, should be even number or a value of multiple of 4 when it's in Skip mode */
    int YStart; /* Start Row of the ROI, should be even number or a value of multiple of 4 when it's in Skip mode */

    int GreenGain; /* Green Gain Value: 0 – 128, the actual gain is GreenGain/8 */
    int BlueGain; /* Blue Gain Value: 0 – 128, the actual gain is BlueGain/8 */
    int RedGain; /* Red Gain Value: 0 – 128, the actual gain is RedGain/8 */
    int MaxExposureTimeIndex; /* The index for maximum exposure time:
        0 – 5ms
```

```

1 – 10ms
2 – 100ms
3 – 750ms
*/
int ExposureTime; /* The current exposure time in Micro second, e.g. 10000 means 10ms */

// For Video image render
bool ImageRenderFitWindow; /* True if the image always fit video window, False if the image will keep the same
                             resolution as the grabbing resolution
                             */
int Gamma; /* Gamma value: 0 – 20, means 0.0 – 2.0 */
int Contrast; /* Contrast value: 0 – 100, means 0% -- 100% */
int Bright; /* Brightness : 0 – 100, means 0% -- 100% */
int SharpLevel; /* SharpLevel: 0 – 3, means Normal, Sharp, Sharper and Sharpest */
bool BWMode; /* Black White mode? */
bool HorizontalMirror; /* Horizontal Mirror? */
bool VerticalFlip; /* Vertical Flip? */

// For Capture Files.
int CatchFrames; /* Number of frames to be captured */
bool IsAverageFrame; /* Save only one frame, but it's the average of all grabbed frames */
bool IsCatchRAW; /* Save as RAW Data File? */
bool IsRawGraph; /* Save as JPG or BMP, but not corrected by Gamma, contrast, bright and sharp algorithm */
bool IsCatchJPEG; /* Save as JPEG File? */
bool CatchIgnoreSkip; /* Always capture full resolution? */
} TImageControl;

```

SDK_API MTUSB_SetFrameSetting(DEV_HANDLE DevHandle, PImageCtl SettingPtr);

User may set the all parameters by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
 SettingPtr – the Pointer to variable of TImageControl structure.

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

Important: This function set all of the parameters of SettingPtr to camera engine, and is effective immediately after the call, if the frame grabbing is started, it's immediately affected by those settings.

SDK_API MTUSB_SetResolution(DEV_HANDLE DevHandle, PImageCtl SettingPtr);

User may set the resolution (including capture and render) parameters by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
 SettingPtr – the Pointer to variable of TImageControl structure.

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

Important: Although the second input argument is a pointer to TimageControl structure, only three elements “Resolution”, “BinMode” and “ImageRenderFitWindow “ are used by this function, all others are ignored.

SDK_API MTUSB_SetStartPosition(DEV_HANDLE DevHandle, PImageCtl SettingPtr);

User may set the start position of ROI parameters by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
 SettingPtr – the Pointer to variable of TImageControl structure.

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

Important: Although the second input argument is a pointer to TimageControl structure, only two elements “XStart” and “YStart “ are used by this function, all others are ignored.

SDK_API MTUSB_SetGain(DEV_HANDLE DevHandle, PImageCtl SettingPtr);

User may set RGB Gains parameters by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
SettingPtr – the Pointer to variable of TImageControl structure.

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

Important: Although the second input argument is a pointer to TimageControl structure, only three elements “GreenGain”, “BlueGain” and “RedGain “ are used by this function, all others are ignored.

SDK_API MTUSB_SetExposureTime(DEV_HANDLE DevHandle, PImageCtl SettingPtr);

User may set the exposure time parameters by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
SettingPtr – the Pointer to variable of TImageControl structure.

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

Important: Although the second input argument is a pointer to TimageControl structure, only two elements “MaxExposureTimeIndex” and “ExposureTime “ are used by this function, all others are ignored.

MaxExposureTimeIndex: User should set it correctly as the exposure time range, there’re 4 ranges:

- 0: 0.05ms – 5ms,
- 1: 0.1ms – 10ms,
- 2: 1ms – 100ms,
- 3: 7.5ms – 750ms

ExposureTime: The setting exposure time in “us”, e.g. 1000 means 1000us (1ms).

For example, if user wants to set exposure time to 15ms, user might do:

```
ImageCtl.MaxExpousreTimeIndex = 2; // 1ms – 100ms range
ImageCtl.ExposureTime = 15000; // 15000us = 15ms
MTUSB_SetExposureTime( CamHandle, &ImageCtl);
```

Here, ImageCtl is a data variable of *TImageControl*, user might invoke “**MTUSB_GetFrameSetting(CamHandle, &ImageCtl)**” to get ImageCtl initialized in the startup codes.

SDK_API MTUSB_SetGammaValue(DEV_HANDLE DevHandle, PImageCtl SettingPtr);

User may set the Gamma, Contrast and Brightness parameters by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
SettingPtr – the Pointer to variable of TImageControl structure.

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

Important: Although the second input argument is a pointer to TimageControl structure, only four elements “Gamma”, “Contrast”, “Bright “ and “SharpLevel” are used by this function, all others are ignored.

SDK_API MTUSB_SetGammaTable(DEV_HANDLE DevHandle, unsigned char *GammaTable);

User may set the internal Gamma Table by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
GammaTable – the Pointer to 256 bytes array which contains the Gamma table.

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

Important: Camera engine has an internal Gamma table to do the Gamma correction, all the output value is get as GammaTable[InputValue], while InputValue is the ADC value read from CMOS sensor.

SDK_API MTUSB_SetShowMode(DEV_HANDLE DevHandle, PImageCtl SettingPtr);

User may set the BWMode, HorizontalMirror and VerticalFlip parameters by invoking this function.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
SettingPtr – the Pointer to variable of TImageControl structure.

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


Important: Although the second input argument is a pointer to TimageControl structure, only three elements “BWMode”, “HorizontalMirror” and “VerticalFlip “ are used by this function, all others are ignored.

SDK_API MTUSB_SetWhiteBalance(DEV_HANDLE DevHandle);

User may call this function for Automatic White Balance set.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

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

Important: This is the equivalent to the button of  Auto White Balance in control panel, note that user should set proper exposure time and put a white paper at proper distance before this function is invoked.

SDK_API MTUSB_SetFrameRateLevel(DEV_HANDLE DevHandle, int RateLevel);

User may call this function to set the current frame grabbing rate.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
RateLevel – Can be from 0 – 10, while 0 means the lowest frame rate, 10 means the highest rate.

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

Important: In the current design, the actual frame rate is mainly depending on the PC resources. The frame rate might be different on a slow PC and a fast PC, the default setting of the camera engine is to set the Maximum frame rate, however, that might not be ideal as almost all the CPU resources will be used by camera engine, which will make other PC applications “hunger” of CPU time, user might want to reduce the frame rate a little bit to politely give other application time to run.

SDK_API MTUSB_SetAutoExposure(DEV_HANDLE DevHandle, bool AutoExposureOn, bool ShowExposureMark);

User may call this function to set the current frame grabbing rate.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
AutoExposureOn – True/False to turn the “Auto Exposure” feature On/Off.
ShowExposureMark – True/False to show/hide the Exposure Mark.

Return: -1 If the function fails (e.g. invalid device handle or camera engine is in “External Trigger” mode)
1 if the call succeeds.

SDK_API MTUSB_GetCurrentFrameRate(DEV_HANDLE DevHandle);

User may call this function to get the current frame grabbing rate.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()

Return: -1 : If the function fails (e.g. invalid device handle)
-2 : Camera engine is not grabbing frames.
-3 : Camera engine is grabbing frames, but the current camera is unplugged from the USB bus.
1 : if the call succeeds.

Important: This is an average frame rate of the current grabbing, as PC is not a real time system, which might switch to other applications, so the actual frame rate is vary a little bit from time to time. This function returns the frame rate at the calling moment. With the control panel, frame rate is also shown on the bottom right corner.

SDK_API MTUSB_GetLastBMPFrame(DEV_HANDLE DevHandle, char *FileName);

User may call this function to get the bitmap format frame of the last captured frame.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
FileName – the full file name for the bitmap file.

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

Important: While the frame grabbing is running OR it's stopped, we can always get the last (for the time this function is invoking) frame of the Video window in Bitmap format. Note that this function may mainly be used in situation of user stop the video as the frame is exactly the user's interesting. Note that this bitmap frame is adjusted with user's setting of Gamma, contrast, bright and sharp level, if user wants to get un-adjusted image data, user might invoke **MTUSB_SetGammaValue()** function with Gamma set to 10 (mean 1.0), contrast and bright set to 50 (mean 50%) and SharpLevel set to 0 (mean Normal).

SDK_API MTUSB_GetCurrentFrame(DEV_HANDLE DevHandle, int FrameType, unsigned char *Buffer);

User may call this function to get a frame.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
FrameType – 0: Raw Data
1: 24bit Bitmap Data (DIB)
Buffer – Byte buffer to hold the whole frame of image.

Return: -1 If the function fails (e.g. invalid device handle or the frame grabbing is NOT running)
1 if the call succeeds.

Important: This function can only be invoked while the frame grabbing is started, user might want to get a frame of image in memory, instead of in a file. This function will put the current frame into Buffer, either in Raw data format, or in bitmap format, note that it's caller's responsibility to have big enough buffer to hold the image data, the buffer size should be:

In Raw Data format: At least (Row x Column) Bytes,

In Bitmap Data Format, for Mono camera, At least (Row x Column) Bytes, for Color Camera, At least 3 x (Row x Column) Bytes.

Note that this buffer is from current frame flow, so it's affected with user's setting of Gamma, contrast, bright and sharp level, if user wants to get un-adjusted image data, user might invoke **MTUSB_SetGammaValue()** function with Gamma set to 10 (mean 1.0), contrast and bright set to 50 (mean 50%) and SharpLevel set to 0 (mean Normal).

For the data structure for Raw and DIB data, please refer to next function.

The frame returned is the frame grabbed at the moment of this function is invoking, it's possible to return the same frame twice if user invokes this function sequentially, or it's possible to miss some frames if user invoke this function in a long interval. For getting each grabbing frame once, it's recommended to use the next function, which installs a callback hooker.

For the format of memory buffer, please refer to the next function.

SDK_API MTUSB_InstallFrameHooker(DEV_HANDLE DeviceHandle, bool IsMaxFrameRate, int FrameType, GetFrameCallBack FrameHooker)

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
IsMaxFrameRate – Set to True if user wants to get Maximum Frame Rate, otherwise set to False.
FrameType – 0: Raw Data
1: 24bit Bitmap Data (DIB)
FrameHooker – Callback function installed.

Return: -1 If the function fails (e.g. invalid device handle or the frame grabbing is NOT running)
1 if the call succeeds.

Important: This function can only be invoked while the frame grabbing is started, user might want to be notified every time the camera engine get a new frame, user might use this function to install a callback function, and the callback will be invoked once for each new frame.

Note

- 1). User may use this function to install a callback, or use the previous function to get a frame directly, it's recommended to use only one of these two functions.
- 2). For some applications need maximum frame rate of a ROI, user might set IsMaxFrameRate to True, which will give user optimized maximum frame rate on a certain PC. The actual frame rate is depending on the Exposure time and the PC resources. [Note that you might see frame rate always changes, this is due to the Windows' resource sharing]
- 3). The callback has the following prototype:

```
typedef void (* GetFrameCallBack)(int FrameType, int Row, int Col,  
    TImageAttachData* Attributes, unsigned char *BytePtr);
```

The TImageAttachData is defined as:

```
typedef struct {  
    int XStart;  
    int YStart;  
    int GreenGain;  
    int BlueGain;  
    int RedGain;  
    int ExposureTime;
```

// The following three parameters are increased from SDK version V1.4.0.0, 2006/10/9

```
    int TriggerOccurred;  
    int Reserve1;  
    int Reserve2;
```

```
} TImageAttachData;
```

While it's invoked, the FrameType is the same as the FrameType we set, while the BytePtr points to a buffer holds the image data. Row and Col are the size of the image. The TImageAttachData has the Image attributes for this particular frame, this is a very useful information if user want to have a Close Loop Control on the Image Analysis...etc. Note that the installed callback should not include any Blocking functions and it should BE as fast as possible, otherwise it will block the camera engine and reduce the frame rate, and it's also very important that there's no GUI operations in this callback hooker, as it's actually invoked in a working thread, which is NOT sync with the main GUI thread (The main GUI thread is written in Delphi, the VCL of Delphi is not supporting multi-thread invoking), So if user has to use messages (e.g. postmessage(...)) for showing any messages on GUI. **From SDK V1.4.0.0, we increase the "TriggerOccurred" parameter which is "1" if a falling edge occurred on camera's trigger pin before the completing of the frame grabbing. Note this is a "One Shot" flag, an occurrence(or more than one occurrences, but all occurred during one frame grabbing) only set this parameter to "1" for this frame. This gives user a chance to know the occurrence of the external trigger while the camera is working in "Video" mode.**

Caution: The external trigger should be filtered properly so that there's no glitches on this pin.

From firmware V1.1.3 and later, the Reserve1 field is used to provide a "Time Stamp" for each frame, note that the Time stamp is a number from 0 – 65535, it represents 0ms – 65535ms(and round back), with this field, user may know the time interval between two frames.

4). The callback function is invoked from a working thread, other than the main thread of the application (usually the UI thread), so attention must be paid for synchronize issues and data sharing.

5). While it's in DIB format, the image data is from current frame flow, so it may be corrected with user's setting of Gamma, contrast, bright and sharp level, if user wants to get un-adjusted image data, user might invoke

MTUSB_SetGammaValue() function with Gamma set to 10 (mean 1.0), contrast and bright set to 50 (mean 50%) and SharpLevel set to 0 (mean Normal). While the Raw data is definitely un-modified, it's ADC value from sensor.

6). For Raw data, the BytePtr points to a buffer which contains a raw image data, it has the following format:

```
typedef struct {  
    unsigned char RawImageData[Rows][Columns];  
} tRawImageData;
```

Here, Rows and Columns are set by user (Resolution, e.g. in case of 1280x1024, its RawImageData[1024][1280].

Also note that the **Rows are from top to bottom of the image, Columns are from Right to Left.**

For DIB data, it's different for Mono camera and Color camera:

For Mono camera, as internally we generate a 8bit bmp, so the buffer points to

```
typedef struct {  
    unsigned char Bitmap[Rows][Columns];  
} tDIBImageData;
```

Each pixel contain a byte, which represents the gray level (0 – 255), please note that for BMP, **the Rows are from bottom to top of the image, Columns are from Left to Right.**

For Color camera, we're using 24bit bmp internally so the BytePtr points to a buffer which is actually a tDIBImageData structure as following:

```
Typedef struct {  
    unsigned char Blue;  
    unsigned char Green;  
    unsigned char Red;  
} tRGBTriple;
```

```
typedef struct {  
    tRGBTriple DIBImageData[Rows][Columns];  
} tDIBImageData;
```

The Rows are from bottom to top of the image, Columns are from Left to Right.

Note that all above data structure is "Byte" aligned.

As you can notice, for Mono camera, it's actually the same for Raw data and Bmp data, except for the orientation of the row and column.

**SDK_API MTUSB_SaveFramesToFiles(DEV_HANDLE DevHandle, PImageCtl SettingPtr,
char *FilePath, char *FileName);**

User may call this function to save one or more frames to files.

Argument: DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
SettingPtr – the Pointer to variable of TImageControl structure.
FilePath – the directory to store the saved files.
FileName – the Filename used for file saving, note that the actual file name will be FileName_x.bmp (or FileName_x.jpg).

Return: -1 If the function fails (e.g. invalid device handle or the frame grabbing is NOT running)
1 if the call succeeds.

Important: Note that This function can only be invoked while the frame grabbing is started, user might want to get one or more frames and save them into a specified location, the "CatchFrames", "IsAverageFrame", "IsCatchRAW", "IsRawGraph", "IsCatchJPEG" and "CatchIgnoreSkip" in the data structure pointed by SettingPtr will be used for number of frames, frame format (Raw, Bmp or Jpeg) and ignore skip mode options.

Note:

IsAverageFrame gives user an option to get only ONE frame but it's the average of all the captured frames.

IsCatchRAW gives user an option to save the files as CMOS sensor raw data, currently, we generate a Text file for user's ease of observation

IsRawGraph gives user an option to save the graphic file (jpg or bmp) with the data which are NOT corrected with the current Gamma, contrast, bright or sharp algorithm.

**SDK_API MTUSB_SetLEDBrightness(DEV_HANDLE DevHandle, unsigned char LEDChannel,
unsigned char Brightness);**

User may call this function to set the Brightness of LED Channels

Argument : DevHandle -- DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
LED Channel – 1 – 5, while 1 – 4 means the channel1 – channel4 of LED light head, 5 means global.
Brightness – 0 – 100, it's the percentage of the brightness, while 0 mean OFF, 100 mean all ON.

Return: -1 If the function fails (e.g. invalid device handle or it's a camera WITHOUT LED Driver.)
1 if the call succeeds.

Important: Note that This function can only be invoked if the camera is with LED driver built in (GLN or MLE).

SDK_API MTUSB_SetGPIOConifg(DEV_HANDLE DevHandle, unsigned char ConfigByte);

User may call this function to configure GPIO pins.

Argument : DevHandle -- DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
ConfigByte – Only the 4 LSB are used, bit0 is for GPIO1, bit1 is for GPIO2 and so on. Set a certain bit to 1 configure the corresponding GPIO to output, otherwise it's input.

Return: -1 If the function fails (e.g. invalid device handle or it's a camera WITHOUT GPIO)
1 if the call succeeds.

Important: Note that This function can only be invoked if the camera is with built in GPIO (MCN, GLN and MCE).

SDK_API MTUSB_SetGPIOInOut(DEV_HANDLE DevHandle, unsigned char OutputByte, unsigned char *InputBytePtr);

User may call this function to set GPIO output pin states and read the input pins states.

Argument : DevHandle -- DevHandle – the device handle returned by either MTUSB_ShowOpenDeviceDialog() or MTUSB_OpenDevice()
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 handle or it's camera WITHOUT GPIO)
1 if the call succeeds.

Important: Note that This function can only be invoked if the camera is with built in GPIO (MCN, GLN and MCE).

For the above SDK APIs, it's recommended to invoke them in the main thread of an application (usually this is the UI thread), as those APIs actually modify the UI of the main control panel (even the panel is not shown). For user needs to have working threads, it's recommended to use working thread for signal/image processing purpose, and notify Main thread (with Windows Thread Sync mechanisms) for the SDK API invoking. Please refer to the Delphi and VC++ examples for the using of those APIs.

SDK_API MTUSB_InstallDeviceHooker(DeviceCallBack DeviceHooker);

User may call this function to install a callback, which will be invoked by camera engine while a Mightex camera is added or removed from the USB bus, note that the USB device is not necessarily a Mightex camera, it might a USB device other than the standard USB devices (e.g. USB HID devices, USB Mass Storage Devices....etc.)

Argument: DeviceHooker – the callback function registered to camera engine.

Return: It always return 1.

Note:

1). If plug or unplug occurs, the camera engine will stop its grabbing and invoke the installed callback function. This notifies the host the occurrence of the device configuration change and it's recommended for host to arrange all the "cleaning" works. Host might simply do house keeping and terminate OR host might let user to re-start the camera engine.

2). The callback function has the following prototype:

typedef void (DeviceFaultCallBack)(int FaultType);*

The FaultType is as following:

- 0 – A camera is removed from USB Bus
- 1 – A camera is attached to USB Bus
- 2 – Low level USB communication error occurred.

3). Some times defected USB connection (caused by the USB connector or cable) might also cause the Plug/Unplug event.

In the VC++ example code, this API is used for installing a callback for P&P event.