

SIPS Camera Driver Interface

Scientific Image Processing System (SIPS) is designed to work with any camera, providing the camera driver DLL is available. Even Moravian Instruments Cx (CMOS-based) and Gx (CCD-based) series of cameras, with which the SIPS package is shipped as a default camera control software, use a regular driver DLLs to interact with SIPS like any other camera.

This document describes the interface between SIPS and camera drivers. It can be used for both implementing a new camera driver for SIPS and for using Gx/Cx camera drivers DLLs to control Moravian cameras from other software packages.

Three driver DLLs are used to control all current Moravian cameras:

- **'gxusb.dll'** handles all CCD-based Gx cameras (G0 to G4, Mark I and Mark II) connected directly to the host computer through USB 2.0 hi-speed lines (480 Mbps).

Also, all CMOS-based C0, C1, and global-shutter sensor based C1+ and C2 cameras, connected using USB 3.0 super-speed lines (5 Gbps), are handled.

Please note this driver handles G2 cameras revision 3 and higher, as well as all G0, G1, G3 and G4 cameras. Older G2 cameras with USB 1.1 interface (revision 1) and the first USB 2 version (revision 2) are no longer supported. All Gx Mark II CCD cameras are supported.

The 'gxusb' DLL offers two image acquisition interfaces (see the next chapter for details):

1. **Camera timing synchronous** interface, used typically to perform precisely timed short exposure (typically shorter than ~0.25 s).
 2. **Computer timing** interface, typically used to perform long exposures (typically above ~0.25s).
- **'gxeth.dll'** handles all Cx and Gx cameras (C0 to C5, G0 to G4) connected to the host computer over TCP/IP network. In such cases the Moravian Camera Ethernet Adapter device is necessary. Up to four Gx/Cx cameras of arbitrary type can be connected to this device using standard USB lines on one side and the standard 1 Gbps Ethernet interface (compatible with older 10/100 Mbps networks) can be attached on the other side. Because the TCP/IP protocol can be routed, the distance between cameras and the host PC is virtually unlimited.

Please note camera support depends on the firmware version of the Camera Ethernet Adapter device. Refer to the Camera Ethernet Adapter User's Guide for procedures to check and/or update the device firmware.

The 'gxusb' DLL offers only one image acquisition interface:

1. **Camera timing asynchronous** interface.
- **'cxusb.dll'** driver handles large cooled C1x, C3, C4 and C5 camera lines. Also, latest C1+ and C2 cameras with rolling-shutter sensors, as well as C1+ and C2 cameras with global shutter sensors and large onboard memory, are handled by this driver.

These cameras use different image acquisition interface compared to smaller C0, C1, and global-shutter sensor based C1+ and C2 lines, and thus it is necessary to introduce new driver, as single driver DLL cannot handle different camera lines, each requiring different handling of image acquisition.

The 'cxusb' DLL offers only one image acquisition interface:

1. **Camera timing asynchronous** interface.

Please note the USB connected cameras require installation of the kernel driver on the host computer. The USB kernel drivers are identified by USB Vendor and Product identifiers. Some cameras are software compatible, so they share the Product ID (and thus use the same driver). The operating system takes care of these drivers and

the user usually needs not install them manually (refer to the **Installing and Using Drivers and Software** manual for kernel driver installation instructions, please). Of course, no kernel drivers are installed when the Moravian Camera Ethernet Adapter is used to communicate with cameras.

Driver DLL Interface

SIPS drivers are Dynamic Link Libraries (DLLs) with specific set of exported functions. All exported functions use standard “C” calling conventions¹, so they can be called from arbitrary software package without problems with name decorations etc.

Some functions are mandatory and the DLL must implement them to be accepted as a SIPS driver. Other functions may be optional, and the particular functionality could be used only if the driver exports (implements) these functions.

A certain exception from this rule are image acquisition functions. There are three sets of image acquisition functions defined:

1. **Camera timing synchronous** interface.
2. **Camera timing asynchronous** interface.
3. **Computer timing** interface.

Hint:

The **camera timing synchronous** and **computer timing** interfaces are legacy and deprecated for new drivers. Also, all new Moravian cameras use the **camera timing asynchronous** interface.

Each driver must implement at least one set of these functions, otherwise there is no way to acquire images, and the driver is rejected by SIPS. E.g. the 'gxusb.dll' driver implements the first and third interfaces while the 'gxeth.dll' driver implements only the second one.

The **camera timing asynchronous** and **computer timing** interfaces cannot be combined in one driver. Every driver should implement one or another interface, but not both.

Remark:

When SIPS can use both **camera timing synchronous** and the **computer timing** interfaces, it uses 0.25 seconds threshold. Exposures shorter than 0.25 seconds use camera timing interface and longer exposures are counted in computer, so they can be of arbitrary length. Computer timing is somewhat less precise compared to camera timing (computer-timing uncertainty is in the order of milliseconds, while camera-timing exposure time precision is better than 0.1 millisecond), but there is no limit in exposure time length.

Camera timing synchronous interface

The camera timing synchronous interface consists of single function **cameraGetFrameExposure** (and 16-bit and 8-bit variants of the same function), performing all necessary steps, beginning with exposure start and ending with image data download. It relies on the camera hardware to count the exposure time, but this function is blocking; it does not return from the call until the exposure time passes and image is read.

Hint:

The blocking nature of the **cameraGetFrameExposure** function is the reason why using of this interface is recommended for short exposures only.

¹ The calling conventions matter in 32-bit code only. The 64-bit Windows versions support only single universal calling convention, and compiler ignores calling convention specifications in the source code, which eliminates all compatibility issues.

Camera timing asynchronous interface

This interface also relies on the camera hardware to count exposure time, but is designed to operate asynchronously, without a possibly long blocking call, waiting until desired exposure time expires and image is downloaded.

The exposure sequence consists of the following steps:

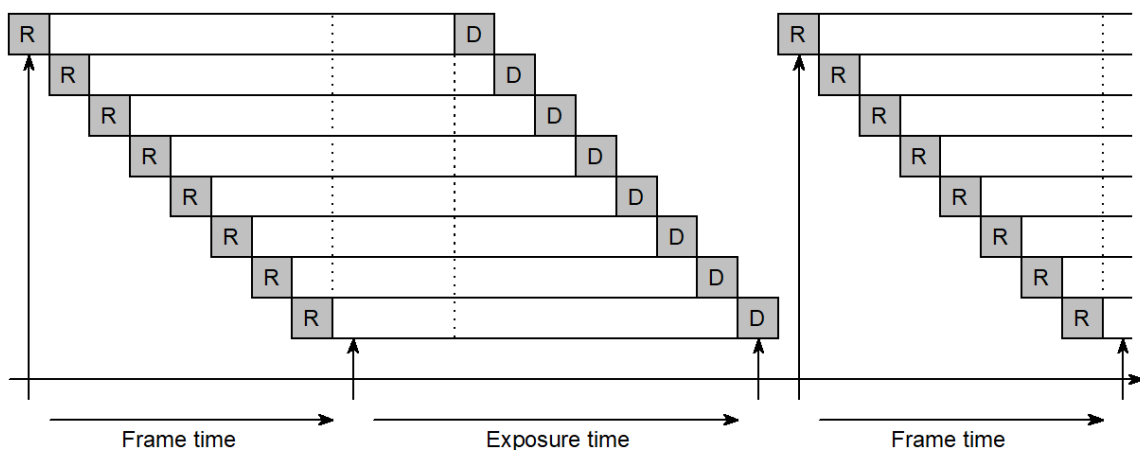
1. call **cameraStartExposure(...)**
2. ... (wait for exposure time, but can proceed to next step earlier)
3. call **cameraImageReady()**, if TRUE, continue to next step, if FALSE repeat this step
4. call **cameraReadImage(...)**

If the exposure is to be terminated earlier than the exposure time expires, it is possible to call **cameraAbortExposure** function. This function contains a parameter, informing the camera if the client application is interested in downloading image data of the shortened exposure.

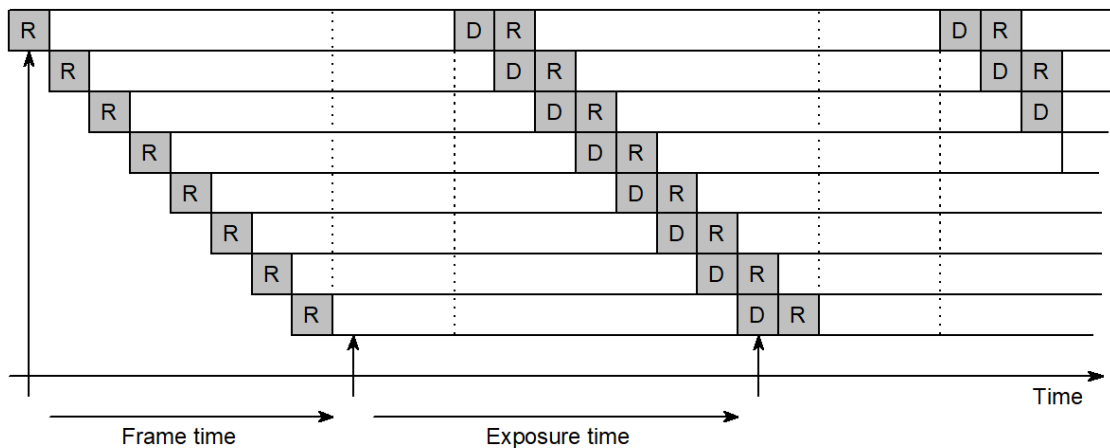
Camera timing asynchronous interface—continuous (serial) read

Cameras with an on-board RAM buffer for multiple frames can be capable of performing exposures in serial mode, in which the next exposure initiates immediately after the previous exposure finished and image is digitized. The delay between two subsequent exposures is virtually zero and all time is consumed by exposure time only.

Single-frame exposure of rolling-shutter camera timing:



Serial read mode eliminates the “Frame time” delay between individual exposures:



The **cameraGetBoolean (gbContinuousExposures)** call can be used to determine if this function can be used with connected camera.

If the exposure time is long enough, so the camera interface is capable to transfer images to host PC faster than individual frames are stored into camera memory, the on-board RAM works just like FIFO buffer, capable to bridge potential delays in the read commands issued by the host PC.

For very short exposures, images can be stored into camera RAM faster than the host PC is capable to download them. In such cases camera performs a fast sequence of exposures until the on-board RAM is filled and then pauses until the PC reads the oldest image. When the oldest frame is read and the memory slot occupied by it is freed, camera automatically performs next exposure.

Continuous image read begins the same way as normal image read by calling the **cameraStartExposure** function. Only if the camera has to perform continuous read, image is not read using the **cameraReadImage** function, but its alternative **cameraReadImageExposure**. This informs the camera should continue with next exposure. Last image in the sequence should be read using standard **cameraReadImage** function, or alternatively the series can be aborted using **cameraAbortExposure** call.

The continuous image commands can be as follows:

1. call **cameraStartExposure(...)**
2. ... (wait for exposure time, but can proceed to next step earlier)
3. call **cameraImageReady()**, if TRUE, continue to next step, if FALSE repeat this step
4. call **cameraReadImageExposure(...)**
5. ... (wait for exposure time, but can proceed to next step earlier)
6. call **cameraImageReady()**, if TRUE, continue to next step, if FALSE repeat this step
7. call **cameraReadImageExposure(...)**
8. ... (wait for exposure time, but can proceed to next step earlier)
9. ...
10. call **cameraImageReady()**, if FALSE repeat this step
11. call **cameraReadImage(...)**, this informs the camera, that the next serial exposure should not be started

Number of frames, which can be stored in the camera on-board memory, naturally depends on the memory size, but also on the image resolution, defined sub-frame, on the binning (if hardware binning is used) etc.

Camera timing asynchronous interface—triggered exposures

Cameras can be equipped with hardware input, which allows external devices to define exact time of exposure start. The **cameraGetBoolean(gbTrigger)** call can be used to determine if the connected camera is equipped with hardware trigger input.

If the exposure should wait for the trigger input, it should be started with the **cameraStartExposureTrigger**, call. The **cameraImageReady** call can be used to determine if the trigger was activated and exposure already finished or the image is still not acquired. The **cameraAbortExposure** can be used to cancel the started exposure, regardless if the camera waits for the trigger or exposure was already started and exposure is in progress.

If the used camera is not equipped with hardware trigger input, the **cameraStartExposureTrigger** function is identical to **cameraStartExposure**, which means the exposure starts immediately upon the call.

Computer timing interface

The exposure sequence controlled by the computer consists of the following steps:

1. call **cameraBeginExposure(...)**
2. ... (wait for exposure time)
3. call **cameraEndExposure(...)**
4. call **cameraGetImage(...)**

Providing the optional **cameraBeginExposure** and **cameraEndExposure** functions are not exported from the driver DLL, computer timing exposure sequence is a bit more complex:

1. call **cameraClearSensor()**
2. call **cameraOpen()**
3. ... (wait for exposure time)
4. call **cameraClose()**
5. call **cameraGetImage(...)**

Note steps 2 (**cameraOpen**) and 4 (**cameraClose**) are skipped if dark or bias frame is exposed. Also, these steps should be skipped if the used camera is not equipped with mechanical shutter.

Driver DLL interface

All driver public API function names are prefixed with the keyword “camera”. This allows multiple drivers of different device classes to co-exist in single DLL. For instance, every device class offers functions like **Enumerate**, **Initialize**, **Release**, etc. When the function names related to a particular driver are prefixed, their names become unique and they can be exported from single DLL. So, for instance, functions **cameraEnumerate** and **filtersEnumerate** can be exported from single DLL without name conflict.

However, the “camera” prefix was added to individual function names only recently. Originally, functions were exported without the prefixed name. To maintain compatibility with existing software, also the latest versions of camera driver DLLs export both variants of the name—with and without prefix.

Hint:

API functions without prefix are legacy and deprecated, all new software should use the functions with the “camera” prefixed names.

The function names after the “camera” prefix are identical with the original names.

Warning:

There are three exceptions from the “same name, only with camera prefix” naming rule.

The **cameraGetBoolean** corresponds to legacy **GetBooleanParameter**, and similarly **cameraGetInteger** and **cameraGetString** correspond to legacy **GetIntegerParameter** and **GetStringParameter** respectively.

This change was also motivated to use same names for functions with same functionality; drivers of other device classes use names **filtersGetBoolean**, **focuserGetBoolean**, **telescopeGetBoolean** etc.

Driver API functions

Exported functions are grouped according to functionality. Individual function description follows this list of all exported functions.

- Functions marked * are optional, SIPS does not require them to be implemented (exported from the driver DLL).
- Functions marked ** are part of image acquisition interfaces; at least one interface must be present. All functions from the particular interface must be present for the interface to be considered as implemented.

Driver life-cycle functions:

```
cameraEnumerate
cameraInitialize
cameraConfigure*
cameraRelease
cameraRegisterNotifyHWND
```

Driver information functions:

```
cameraGetBoolean
cameraGetInteger
cameraGetString
cameraGetValue
cameraGetLastErrorString
```

Image acquisition parameter functions:

```
cameraAdjustSubFrame*
cameraSetBinning*
cameraSetTemperature*
cameraSetTemperatureRamp*
cameraSetFan*
cameraEnumerateReadModes*
cameraSetReadMode*
cameraSetGain*
cameraConvertGain*
cameraSetWindowHeating*
cameraSetPreflash*
```

Image acquisition **camera timing synchronous** interface functions:

```
cameraGetImageExposure**
```

Image acquisition **camera timing asynchronous** interface functions:

```
cameraStartExposure**
cameraStartExposureTrigger**
cameraAbortExposure**
cameraImageReady**
cameraReadImage**
cameraReadImageExposure**
```

Image acquisition **computer timing** interface functions:

```
cameraBeginExposure*,**
cameraEndExposure*,**
cameraClearSensor**
cameraOpen**
```

```
cameraClose**
cameraGetImage**
```

Filter wheel functions:

```
cameraEnumerateFilters*
cameraEnumerateFilters2*
cameraSetFilter*
cameraReinitFilterWheel*
```

Guiding functions:

```
cameraMoveTelescope*
cameraMoveInProgress*
```

GPS functions:

```
cameraGetImageTimeStamp*
cameraGetGPSData*
cameraGetGPSData2*
```

Data types used in SFW driver Application Programming Interface:

```
typedef int          INTEGER;
typedef short        INT16;
typedef unsigned int  CARDINAL;
typedef unsigned char CARD8;
typedef float         REAL;
typedef double        LONGREAL;
typedef unsigned char CHAR;
typedef unsigned char BOOLEAN;
typedef void *        ADDRESS;
struct               CCamera;
```

Driver life-cycle functions

```
void __cdecl cameraEnumerate(
    void ( __cdecl *CallbackProc )( CARDINAL )
)
```

Mandatory function.

Enumerate allows discovering of all cameras currently connected to the host PC. Its argument is a pointer to callback function **CallbackProc** with single unsigned integer argument. This callback function is called for each connected camera, and the camera identifier² is passed as an argument. Application can then offer the user a list of all connected cameras to choose the one with which the user wants to work etc.

If the application is designed to work with one camera only or the camera identifier is known (for instance the application scans some .INI file where the identifier is defined), 'Enumerate' needs not to be called at all.

Please note the callback function uses the "C" calling conventions, as opposed to "stdcall" conventions often used in the Windows OS.

```
CCamera* __cdecl cameraInitialize(
    CARDINAL Id
)
```

Mandatory function.

User-space driver is designed to handle multiple cameras at once. Driver distinguishes individual cameras using handles to individual instances. Handle is defined as unsigned integer of the size corresponding to the size of the

² Identifier is an unsigned integer number, unique to every camera. The value is always 32-bit on both 32-bit and 64-bit driver versions.

address (32 bits or 64 bits). This allows the driver implementation to allocate class/structure and return a pointer to it as a handle.

This handle is returned by the **cameraInitialize** function. It is necessary to pass the camera identifier to **cameraInitialize** (see the **cameraEnumerate** function description). If the **cameraInitialize** returns 0xFFFFFFFF (INVALID_HANDLE_VALUE), the particular camera cannot be used. It is either not connected or is already used by some other application.

```
BOOLEAN __cdecl cameraConfigure(  
    CCamera *PCamera,  
    ADDRESS ParentHwnd  
)
```

Optional function.

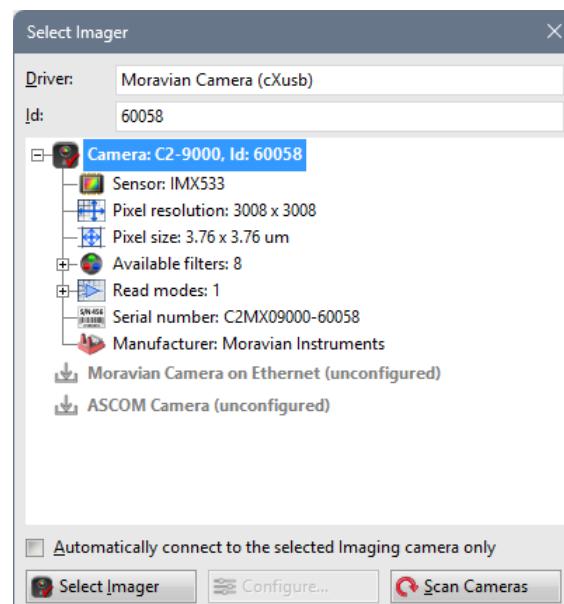
If the driver requires configuration (for instance TCP/IP based camera requires definition of IP address), it should implement this function. This function can open a dialog box (and use the ParentHwnd passed from the user application).

This function can use already enumerated **PCamera** handle to re-configure existing instance of driver or the **PCamera** can be -1, in which case the configuration affects all instances of the particular driver. User application should allow calling of **cameraConfigure** function with PCamera = -1 also in the case **cameraEnumerate** did not return any camera instance.

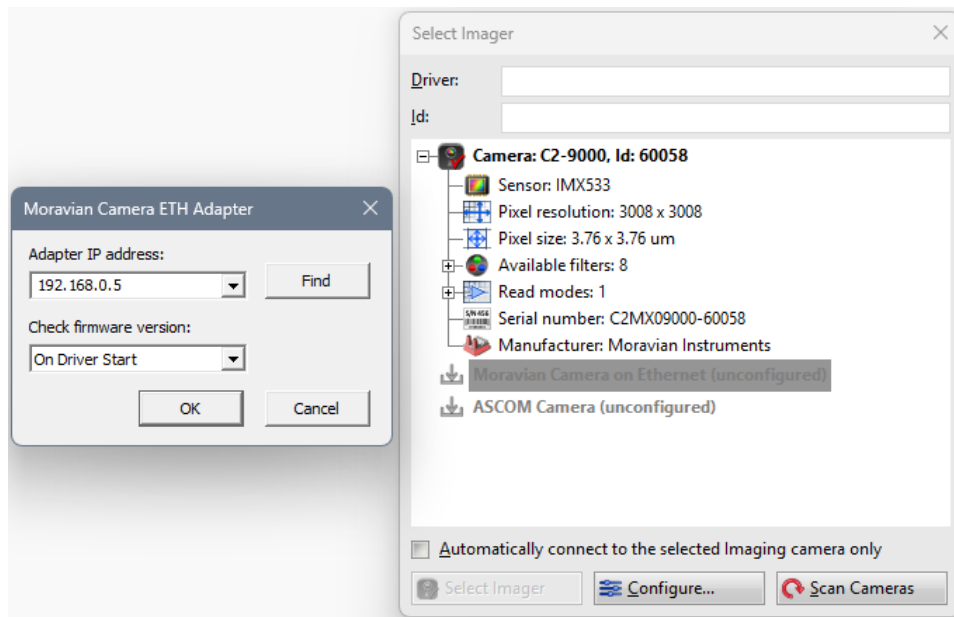
Remark:

SIPS always offer an empty line marked “unconfigured” for the drive exporting this function. After the configuration is finished, SIPS calls **cameraEnumerate** to get a fresh list of available cameras (e.g. from new Moravian Camera Ethernet Adapter device).

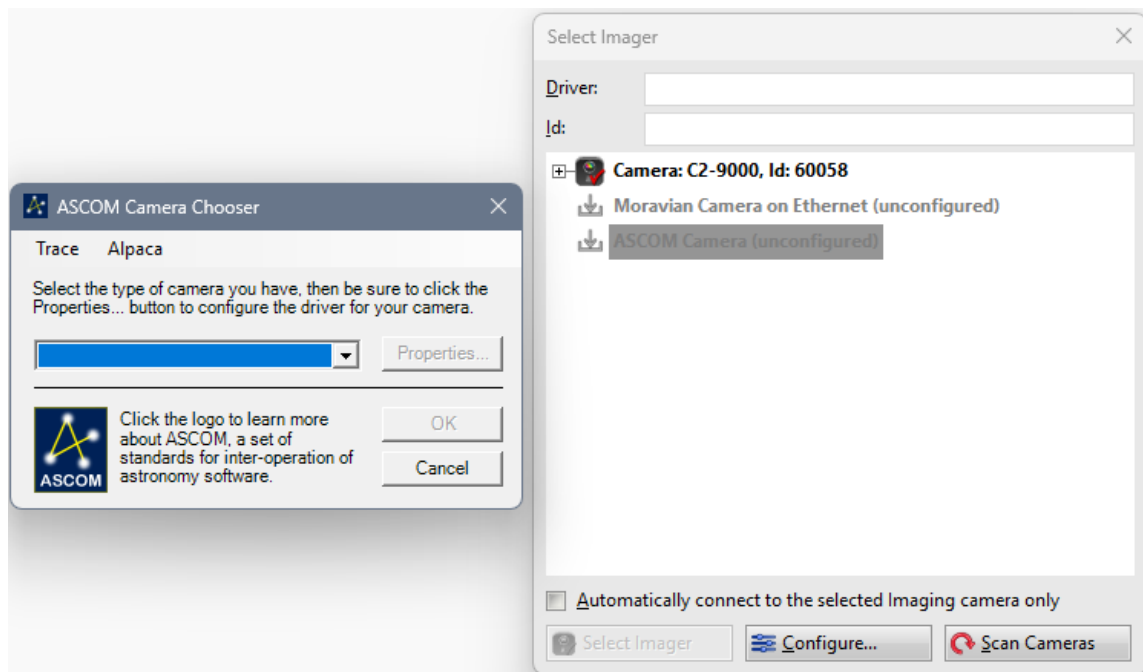
Below is an example of driver selection tool in SIPS, showing a C2 camera connected over USB, which does not require any configuration. Another two grayed lines indicate presence of drivers (Moravian Camera Ethernet Adapter and ASCOM) marked “unconfigured”, which allow call of **cameraConfigure** function.



The **cameraConfigure** function of the Moravian Camera Ethernet Adapter driver opens the **Ethernet Adapter IP address** selection dialog box:



while the same function of the ASCOM driver opens the standard **ASCOM Camera Chooser** dialog box:



```
void __cdecl cameraRelease(
    CCamera *PCamera
)
```

Mandatory function.

When the camera is no longer used, the handle must be released by the **cameraRelease** call. No other function (with the exception of **cameraEnumerate** and **cameraInitialize**) may be called after the **cameraRelease** call.

```
void __cdecl RegisterNotifyHwnd(
    CCamera *PCamera,
    ADDRESS NotifyHwnd
)
```

Mandatory function.

The driver can notify the application whether the camera was plugged or unplugged. Notifications are sent as Windows messages to the windows, which HWND was passed as an argument to the **cameraRegisterNotifyHWND** function.

Notification messages are:

```
#define WM_CAMERA_CONNECT      1034
#define WM_CAMERA_DISCONNECT  1035
```

When the application is no longer interested in this notification, it can call **cameraRegisterNotifyHWND** with NULL as a handle.

Calling **cameraRelease** automatically calls **cameraRegisterNotifyHWND** with NotifyHWND = NULL.

Driver information functions

```
BOOLEAN __cdecl cameraGetBoolean(
    CCamera *PCamera,
    CARDINAL Index,
    BOOLEAN *Boolean
)
```

Mandatory function.

Hint:

The legacy name of this function is **GetBooleanParameter**.

Function **cameraGetBoolean** returns boolean value depending on the Index parameter. If the function does not “understand” passed Index, it returns FALSE.

gbConnected	= 0	TRUE if camera currently connected
gbSubFrame	= 1	TRUE if camera supports sub-frame read
gbReadModes	= 2	TRUE if camera supports multiple read modes
gbShutter	= 3	TRUE if camera is equipped with mechanical shutter
gbCooler	= 4	TRUE if camera is equipped with active CCD cooler
gbFan	= 5	TRUE if camera fan can be controlled
gbFilters	= 6	TRUE if camera controls filter wheel
gbGuide	= 7	TRUE if camera is capable to guide the telescope mount
gbWindowHeating	= 8	TRUE if camera can control the CCD window heating
gbPreflash	= 9	TRUE if camera can use CCD preflash
gbAsymmetricBinning	= 10	TRUE if camera horizontal and vertical binning can differ
gbMicrometerFilterOffsets	= 11	TRUE if filter focusing offsets are in micrometers
gbPowerUtilization	= 12	TRUE if camera can return power utilization in GetValue
gbGain	= 13	TRUE if camera can return used gain in GetValue
gbElectronicShutter	= 14	TRUE if the sensor is equipped with electronic shutter
gbGPS	= 16	TRUE if the camera is equipped with GPS receiver
gbContinuousExposures	= 17	TRUE if the camera is capable of serial exposures
gbTrigger	= 18	TRUE if the sensor is equipped with hardware trigger port
gbConfigured	= 127	TRUE if camera is configured
gbRGB	= 128	TRUE if camera has Bayer RGBG filters on sensor
gbCMY	= 129	TRUE if camera has CMY filters on sensor
gbCMYG	= 130	TRUE if camera has CMYG filters on sensor
gbDebayerXOdd	= 131	TRUE if camera Bayer masks starts on horizontal odd pixel
gbDebayerYOdd	= 132	TRUE if camera Bayer masks starts on vertical odd pixel
gbInterlaced	= 133	TRUE if CCD detector is interlaced (else progressive)

Hint:

Similarly to changes of function name, legacy drivers used gbp* instead of gb* one.

```

BOOLEAN __cdecl cameraGetInteger(
    CCamera *PCamera,
    CARDINAL Index,
    INTEGER *Num
)

```

Mandatory function.

Hint:

The legacy name of this function is **GetIntegerParameter**.

Function **cameraGetInteger** returns integer value depending on the Index parameter. If the function does not “understand” passed Index, it returns FALSE.

giCameraId	= 0	Identifier of the current camera
giChipWidth	= 1	Sensor width in pixels
giChipDepth	= 2	Sensor depth in pixels
giPixelWidth	= 3	Sensor pixel width in nanometers
giPixelDepth	= 4	Sensor pixel depth in nanometers
giMaxBinningX	= 5	Maximum binning in horizontal direction
giMaxBinningY	= 6	Maximum binning in vertical direction
giReadModes	= 7	Number of read modes offered by the camera
giFilters	= 8	Number of filters offered by the camera
giMinimalExposure	= 9	Shortest exposure time in microseconds (µs)
giMaximalExposure	= 10	Longest exposure time in milliseconds (ms)
giMaximalMoveTime	= 11	Longest time to move the telescope in milliseconds (ms)
giDefaultReadMode	= 12	Read mode to be used as default
giPreviewReadMode	= 13	Read mode to be used for preview (fast read)
giMaxWindowHeating	= 14	Maximal value for cameraSetWindowHeating call
giMaxFan	= 15	Maximal value for cameraSetFan call
giMaxGain	= 16	Maximum value for cameraSetGain call
giMaxPossiblePixelValue	= 17	Maximum value of (saturated) pixel
Note the value may vary with read mode and binning, read only after cameraSetReadMode and cameraSetBinning calls		
giLineTime	= 18	Time to digitize one line of rolling-shutter cameras equipped with GPS receiver in picoseconds (ps)
giBiasPixelValue	= 19	Minimum (bias) value of pixel
Note the value may vary with read mode and binning, read only after cameraSetReadMode and cameraSetBinning calls		
giFirmwareMajor	= 128	Camera firmware version (optional)
giFirmwareMinor	= 129	
giFirmwareBuild	= 130	
giDriverMajor	= 131	Driver version (optional)
giDriverMinor	= 132	
giDriverBuild	= 133	
giFlashMajor	= 134	Flash version (optional)
giFlashMinor	= 135	
giFlashBuild	= 136	

Hint:

Similarly to changes of function name, legacy drivers used gbi* instead of gi* one.

```

BOOLEAN __cdecl cameraGetString(
    CCamera *PCamera,
    CARDINAL Index,
    CARDINAL String_HIGH,
    CHAR *String
)

```

Mandatory function.

Hint:

The legacy name of this function is **GetStringParameter**.

Function **cameraGetString** returns string value depending on the Index parameter. If the function does not “understand” passed Index, it returns FALSE.

The string is copied to the buffer pointed by the **String** parameter. The function checks the maximum buffer size not to cause buffer overrun. The highest character index (index starts with 0) of the buffer must be passed as **String_HIGH** parameter. If the buffer is longer than the passed string, terminating zero character is also copied.

gsCameraDescription	= 0	Camera description
gsManufacturer	= 1	Manufacturer name
gsCameraSerial	= 2	Camera serial number
gsChipDescription	= 3	Used sensor description

Hint:

Similarly to changes of function name, legacy drivers used gsi* instead of gs* one.

```
BOOLEAN __cdecl cameraGetValue(  
    CCamera *PCamera,  
    CARDINAL Index,  
    REAL *Value  
)
```

Mandatory function.

Function **cameraGetValue** returns float value depending on the Index parameter. If the function does not “understand” passed Index, it returns FALSE.

It is like **cameraGetBoolean/Integer/String**, but while the three previous functions return static values (independent on current camera state), the **cameraGetValue** returns numbers reflecting current state of the camera (e.g. sensor temperature etc.). The difference is that it is typically not necessary to communicate with attached camera to get **cameraGetBoolean/Integer/String**, but **cameraGetValue** typically needs to perform communication before it returns.

gvChipTemperature	= 0	Temperature of the sensor in deg. Celsius
gvHotTemperature	= 1	Temperature of the cooler hot side deg. Celsius
gvCameraTemperature	= 2	Temperature inside the camera in deg. Celsius
gvEnvironmentTemperature	= 3	Temperature of the environment air in deg. Celsius
gvSupplyVoltage	= 10	Voltage of the camera power supply
gvPowerUtilization	= 11	Sensor cooling utilization (number 0 to 1)
gvADCGain	= 20	A/D converter gain in electrons/ADU

Not every camera is equipped with all sensors. If the camera hardware cannot measure requested value, the function returns FALSE.

```
BOOLEAN __cdecl cameraGetLastErrorString(  
    CCamera *PCamera,  
    CARDINAL ErrorString_HIGH,  
    CHAR *ErrorString  
);
```

Mandatory function.

If any call fails (returns FALSE), **cameraGetLastErrorString** return failure description. **ErrorString** is passed like **String** in the **cameraGetString** function.

Image acquisition parameter functions

```
BOOLEAN __cdecl cameraAdjustSubFrame(  
    CCamera *PCamera,  
    INTEGER *x,  
    INTEGER *y,  
    INTEGER *w,  
    INTEGER *d,  
)
```

Optional function.

Digitization of individual pixels of CCD sensors is performed by camera electronics, outside of the sensor itself. So, the sub-frame reading of a CCD is performed in the camera and the sensor itself has little or no influence on this function. Limiting of image read to a sub-frame is naturally available in all CCD based Gx cameras without any restrictions on sub-frame position and/or size. So, the `cameraAdjustSubFrame` function does not alter the sub-frame coordinates if used with CCD camera.

Situation with CMOS sensors is just opposite—reading of a sub-frame³ has to be implemented in the sensor hardware/firmware and camera electronics cannot affect it. But sub-frame support by CMOS sensor hardware may impose number of limitations on the sub-frame position and size. What's more, these limitations differ among various sensors and also between 8-bit and 12-bit read modes of the same sensor etc. For instance, sub-frame origin must be aligned to certain multiply of pixels (typically 4 or 8 pixels). The same limitation is valid for sub-frame width and height. Also, minimal sub-frame width is often several hundred pixels etc.

But SIPS camera drivers as well as user interface were not designed to impose any sub-frame position/size restrictions. So, when a sub-frame read is requested by the user, the driver must combine two approaches:

1. Cx camera driver checks if the sub-frame position and size comply to sensor-imposed limitations. If yes, camera hardware is programmed to limit read to specific ROI only and resulting image is directly returned to the user.
2. If the requested sub-frame position and/or size does not fit sensor limitations, the driver enlarges the sub-frame size and position to satisfy the sensor and at the same time to be the smallest possible one, but containing the entire user requested sub-frame. This greater sub-frame is then read from camera and cropped by driver to the size requested by the user.

In most cases, but not always, the differences in sub-frame read from camera and sub-frame returned to the user are typically very small, a few pixels necessary to align position and size to multiply of 4 or 8 pixels etc. So, amount of data transferred from camera is almost the same, only the software cropping consumes some time.

The purpose of the **`cameraAdjustSubFrame`** function is to adjust sub-frame position and/or size to comply with sensor-imposed limitations. The application may offer the user to adjust requested sub-frame coordinates, so the driver can return images directly without software cropping and thus to achieve higher performance.

Remark:

The support for hardware sub-frames must be implemented also with camera firmware. For C1, C1+ and C2 cameras with global shutter CMOS sensors, make sure the firmware is of version 7.6 or higher. Otherwise, the driver always reads full-size images from the camera and implements sub-frame by software cropping. The impact on FPS, when only a small portion of image is requested, is obvious.

```
BOOLEAN __cdecl cameraSetBinning(  
    CCamera *PCamera,  
    CARDINAL x,  
    CARDINAL y  
)
```

Optional function.

Sets the required read binning. If the camera does not support binning, this function has no effect.

```
BOOLEAN __cdecl cameraSetTemperature(  
    CCamera *PCamera,  
    REAL Temperature  
)
```

Optional function.

³ Sub-frames are often denoted as ROI—Region Of Interest

Sets the required sensor temperature, **Temperature** parameter is expressed in degrees Celsius. If the camera has no cooler, this function has no effect.

```
BOOLEAN __cdecl cameraSetTemperatureRamp(  
    CCamera *PCamera,  
    REAL    Ramp  
)
```

Optional function.

Sets the maximum speed with which the driver changes chip temperature. The **Ramp** parameter is expressed in degrees Celsius per minute. If the camera has no cooler, this function has no effect.

Hint:

As this function is optional, the driver may not implement it. In such case, it is a task of the user's application to implement ramping.

```
BOOLEAN __cdecl cameraSetFan(  
    CCamera *PCamera,  
    CARD8    Speed  
)
```

Optional function.

If the camera is equipped with cooling fan and allows its control, this function sets the fan rotation speed. The maximum value of the **Speed** parameter should be determined by **cameraGetInteger(giMaxFan)** parameter index. If the camera supports only on/off switching, the maximum value should be 1 (fan on), while value 0 means fan off.

```
BOOLEAN __cdecl cameraEnumerateReadModes(  
    CCamera *PCamera,  
    CARDINAL Index,  
    CARDINAL Description_HIGH,  
    CHAR     *Description  
);
```

Optional function.

Enumerates all read modes provided by the camera. This enumeration does not use any callback, the caller passes **Index** beginning with 0 and repeats the call with incremented Index until the call returns FALSE. **Description** string is passed similarly to the **String** parameter of the **cameraGetString** function.

```
BOOLEAN __cdecl cameraSetReadMode(  
    CCamera *PCamera,  
    CARDINAL mode  
);
```

Optional function.

Sets required read mode.

```
BOOLEAN __cdecl cameraSetGain(  
    CCamera *PCamera,  
    CARDINAL gain  
);
```

Optional function.

Sets required gain. Range of parameter gain depends on camera hardware, as it typically represents directly a register value. This method is chosen to allow to control gain as precisely as each camera allows.

Low limit is always 0, high limit is returned by function **cameraGetInteger(giMaxGain)**.

```

BOOLEAN __cdecl cameraConvertGain(
    CCamera *PCamera,
    CARDINAL gain,
    LONGREAL *dB,
    LONGREAL *times
);

```

Optional function.

As the **cameraSetGain** function accepts camera-dependent parameter gain, which typically does not represent actual gain as signal multiply or value in dB, a helper function **cameraConvertGain** is provided to convert register value into gain in logarithmic dB units as well as in linear times-signal units.

```

BOOLEAN __cdecl cameraSetWindowHeating(
    CCamera *PCamera,
    CARD8 Heating
)

```

Optional function.

If the camera is equipped with sensor cold chamber front window heater and allows its control, this function sets heating intensity. The maximum value of the **Heating** parameter should be determined by **cameraGetInteger(giMaxWindowHeating)** call. If the camera supports only on/off switching, the maximum value should be 1 (heating on), while value 0 means heating off.

```

BOOLEAN __cdecl cameraSetPreflash(
    CCamera *PCamera,
    LONGREAL PreflashTime,
    CARDINAL ClearNum
)

```

Optional function.

If the camera is equipped with preflash electronics, this function controls it. Parameter **PreflashTime** defines time for which the preflash LED inside the camera is switched on. Second parameter **ClearNum** defines how many times the sensor has be cleared after the preflash.

Actual values of these parameters depend on the particular camera model (e.g. number and luminance of the preflash NIR LEDs used etc.). Gx series of CCD cameras typically need less than 1 second to completely saturate the CCD (**PreflashTime**). Number of subsequent clears should be at last 2, but more than 4 or 5 clears is not useful, too.

Image acquisition camera timing synchronous interface functions

```

BOOLEAN __cdecl cameraGetImageExposure(
    CCamera *PCamera,
    LONGREAL ExpTime,
    BOOLEAN UseShutter,
    INTEGER x,
    INTEGER y,
    INTEGER w,
    INTEGER d,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);

```

Optional function.

This function implements the entire **Camera timing synchronous** interface.

Warning:

Note this function is blocking, it does not return from the call until the exposure time passes and image is read.

For exposures of tens of seconds or minutes, the uncertainty of computer timing is not important (depending on the PC hardware, the uncertainty can be in the order of milliseconds). For short exposures, the PC host timing may not be good enough. This is why the **Camera timing synchronous** interface is introduced, in addition to legacy **Computer timing asynchronous** one. This function lets the camera itself count the exposure with much higher precision, limited by the camera hardware only.

The **ExpTime** parameter is the required exposure time in seconds. The **UseShutter** parameter tells the driver if the dark frame (without light) has to be acquired, or the shutter is to be opened and closed to acquire normal light image. Sub-frame coordinates are passed in the **x**, **y**, **w**, and **d** parameters. If the camera does not support sub-frame read, **x** and **y** must be 0 and **w** and **d** must be the sensor resolution.

It is expected the driver returns 16 bits per pixel (2 bytes) **w * d** matrix copied to **BufferAdr** address. The buffer must be allocated by the caller, driver does not allocate any memory. The **BufferLen** parameter informs the driver about allocated memory block length in bytes (not in pixels!). It must be greater or equal to image size in bytes else the function fails.

Maximum exposure time depends on the particular camera model. Another image exposition interface has to be used if longer than maximum exposure time is desired.

Remark:

Please note the maximum exposure time is ~65.5 seconds for the G2, G3 and G4 cameras and 8.192 seconds for the G0 and G1 cameras. If longer exposures are necessary, use the **Computer asynchronous timing** image acquisition interface.

CMOS cameras of all types (Cx series) can pass exposure time as long as approximately 150 hours, which is way above any reasonable exposure time.

```
BOOLEAN __cdecl cameraGetImageExposure16b(
    CCamera *PCamera,
    LONGREAL ExpTime,
    BOOLEAN UseShutter,
    INTEGER x,
    INTEGER y,
    INTEGER w,
    INTEGER d,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);
```

Optional function.

This function is equivalent to normal **cameraGetImageExposure** call, which also returns 16-bit pixels.

```
BOOLEAN __cdecl cameraGetImageExposure8b(
    CCamera *PCamera,
    LONGREAL ExpTime,
    BOOLEAN UseShutter,
    INTEGER x,
    INTEGER y,
    INTEGER w,
    INTEGER d,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);
```

Optional function.

Providing the used camera supports 8-bit read mode, using **cameraGetImageExposure** or **cameraGetImageExposure16b** call forces the driver to internally expand the buffer from 8-bit pixel to 16-bit pixel format, which consumes some time. If the client application can handle image frames with single byte per pixel, using the **cameraGetImageExposure8b** function can save time.

Remark:

Note the camera hardware must be able to return 8-bit pixels and the read mode must also be set to 8-bit per pixel (if the camera offers multiple read modes and only some of them are 8-bit). Otherwise, this call returns FALSE without even performing any exposure. In such cases, normal **cameraGetImageExposure** must be called to return 16-bit pixel frame.

Image acquisition camera timing asynchronous interface functions

```
BOOLEAN __cdecl cameraStartExposure(  
    CCamera *PCamera,  
    LONGREAL ExpTime,  
    BOOLEAN UseShutter,  
    INTEGER x,  
    INTEGER y,  
    INTEGER w,  
    INTEGER d  
);
```

Optional function.

When the **Camera timing asynchronous** interface is used, every exposure starts with **cameraStartExposure** call. Driver accepts exposure time, whether to open and close shutter (light or dark image) and sub-frame coordinates. If the camera does not support sub-frame read, **x** and **y** must be 0 and **w** and **d** must be the sensor resolution. The exposure is started upon executing of this call.

```
BOOLEAN __cdecl cameraStartExposureTrigger(  
    CCamera *PCamera,  
    LONGREAL ExpTime,  
    BOOLEAN UseShutter,  
    INTEGER x,  
    INTEGER y,  
    INTEGER w,  
    INTEGER d  
);
```

Optional function.

This function works similarly to **cameraStartExposure**, only the exposure does not start upon function call, but waits for hardware trigger. The **cameraImageReady** function can be used to determine if the trigger was activated and exposure already finished or the image is still not acquired. The **cameraAbortExposure** can be used to cancel the started exposure, regardless if the camera waits for the trigger or exposure was already started and exposure is in progress.

If the used camera is not equipped with hardware trigger input, the **cameraStartExposureTrigger** function is identical to **cameraStartExposure**, which means the exposure starts immediately upon function call. The **cameraGetBoolean(gbpTrigger)** call can be used to determine if the used camera has trigger input.

```
BOOLEAN __cdecl cameraAbortExposure(  
    CCamera *PCamera,  
    BOOLEAN Download  
);
```

Optional function.

When the exposure, already started by **cameraStartExposure** or **cameraStartExposureTrigger** function, is to be terminated before the exposure time expires, the **cameraAbortExposure** should be used. The **Download** parameter indicates whether the image should be digitized, because the user will call **cameraReadImage** later or the image should be discarded.

```

BOOLEAN __cdecl cameraImageReady(
    CCamera *PCamera,
    BOOLEAN *Ready
);

```

Optional function.

When the exposure is started by **cameraStartExposure** function, the **cameraImageReady** function returns FALSE in the **Ready** parameter if the exposure is still running. When the exposure finishes and it is possible to call **cameraReadImage**, the **cameraImageReady** function returns TRUE.

Hint:

It is recommended to count the exposure time within the user application despite the fact the exact exposure time is determined by the camera/driver and to start calling of **cameraImageReady** only after the exposure time expired. Starting to call **cameraImageReady** in the “infinite” loop immediately after **cameraStartExposure** and call it for the whole exposure time (and thus keeping at last one CPU core at high utilization) is a bad programming manner (politely expressed).

```

BOOLEAN __cdecl cameraReadImage(
    CCamera *PCamera,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);

```

Optional function.

When the **cameraImageReady** returns **Ready = TRUE**, it is possible to call **cameraReadImage**. It is expected the driver returns 16 bits per pixel (2 bytes) matrix copied to **BufferAdr** address. The buffer must be allocated by the caller, driver does not allocate any memory. The **BufferLen** parameter specifies allocated memory block length in bytes (not in pixels!). It has to be greater or equal to image size in bytes else the function fails.

```

BOOLEAN __cdecl cameraReadImageExposure(
    CCamera *PCamera,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);

```

Optional function.

The **cameraReadImageExposure** function initiates exposed image read exactly like the **cameraReadImage** function but lets the camera immediately start the subsequent exposure. Individual exposures follow immediately one after another, without the overhead of clearing the sensor.

If the function is called and the connected camera does not support serial read, functions returns FALSE. The **cameraGetBoolean(gbContinuousExposures)** call can be used to determine if this function can be used.

```

BOOLEAN __cdecl cameraReadImage16b(
    CCamera *PCamera,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);

```

Optional function.

This function is equivalent to normal **cameraReadImage** call, which also returns 16-bit pixels.

```

BOOLEAN __cdecl cameraReadImage8b(
    CCamera *PCamera,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);

```

Optional function.

Providing the used camera supports 8-bit read mode, using **cameraReadImage** or **cameraReadImage16b** call forces the driver to internally expand the buffer from 8-bit pixel to 16-bit pixel format, which consumes some time. If the client application can handle image frames with single byte per pixel, using **cameraReadImage8b** function can save time.

Note the camera hardware must be able to return 8-bit pixels, and the read mode must also be set to 8-bit per pixel (if the camera offers multiple read modes and only some of them are 8-bit). Otherwise, this call returns FALSE without even performing any exposure. In such case, normal **cameraReadImage** must be called to return 16-bit pixel frame.

Image acquisition computer timing interface functions

```
BOOLEAN cameraBeginExposure(  
    CCamera *PCamera,  
    BOOLEAN UseShutter  
);
```

Optional function.

This function, if implemented, is used instead of **cameraClearSensor** and **cameraOpen** calls (note the **cameraOpen** is to be called only if no dark or bias frame is opened, this option is replaced by the **UseShutter** parameter of the **cameraBeginExposure** function).

```
BOOLEAN cameraEndExposure(  
    CCamera *PCamera,  
    BOOLEAN UseShutter,  
    BOOLEAN AbortData  
);
```

Optional function.

This function complements the **cameraBeginExposure** function and if present, it is to be called instead of **cameraClose**. Parameter **UseShutter** indicates shutter operation (should be FALSE if it was FALSE in the corresponding **cameraBeginExposure** call).

Typically, the **cameraGetImage** call follows the **cameraEndExposure** to actually read the image frame. If the exposure was canceled by the user and no image is to be read from camera, parameter **AbortData** must be set to TRUE.

```
BOOLEAN __cdecl cameraClearSensor(  
    CCamera *PCamera  
);
```

Optional function.

The **cameraClearSensor** functions clears the sensor from accumulated charge, which is necessary before each exposure has to be started in the case optional cameraBeginExposure and cameraEndExpopsure functions are missing.

```
BOOLEAN __cdecl cameraOpen(  
    CCamera *PCamera  
);
```

Optional function.

Opens camera shutter. If the camera does have mechanical shutter, this function has no effect.

```
BOOLEAN __cdecl cameraClose(  
    CCamera *PCamera  
);
```

Optional function.

Closes camera shutter. If the camera does have mechanical shutter, this function has no effect.

```

BOOLEAN __cdecl cameraGetImage(
    CCamera *PCamera,
    INTEGER x,
    INTEGER y,
    INTEGER w,
    INTEGER d,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);

```

Optional function.

Reads image from the camera. Image is a **w * d** matrix of 16-bit pixels. The **BufferAdr** parameter points to the buffer to which the image is to be copied. **BufferLen** is the size of the buffer in bytes (not pixels!). If the passed size is not big enough to hold the image, the call fails.

. If the camera does not support sub-frame read, **x** and **y** must be 0 and **w** and **d** must be the sensor resolution.

```

BOOLEAN __cdecl cameraGetImage16b(
    CCamera *PCamera,
    INTEGER x,
    INTEGER y,
    INTEGER w,
    INTEGER d,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);

```

Optional function.

This function is equivalent to the **cameraGetImage** function, which also returns 16-bit pixels.

```

BOOLEAN __cdecl cameraGetImage8b(
    CCamera *PCamera,
    INTEGER x,
    INTEGER y,
    INTEGER w,
    INTEGER d,
    CARDINAL BufferLen,
    ADDRESS BufferAdr
);

```

Optional function.

Providing the used camera supports 8-bit read mode, using **cameraGetImage** or **cameraGetImage16b** call forces the driver to internally expand the buffer from 8-bit pixel to 16-bit pixel format, which consumes some time. If the client application can handle image frames with single byte per pixel, using the **cameraGetImage8b** function can save time.

Note the camera hardware must be able to return 8-bit pixels, and the read mode must also be set to 8-bit per pixel (if the camera offers multiple read modes and only some of them are 8-bit). Otherwise, this call returns FALSE without even performing any exposure. In such case, normal **cameraGetImage** must be called to return 16-bit pixel frame.

Filter wheel functions

```

BOOLEAN __cdecl cameraEnumerateFilters(
    CCamera *PCamera,
    CARDINAL Index,
    CARDINAL Description_HIGH,
    CHAR *Description,
    CARDINAL *Color
);

```

Optional function.

Enumerates all filters provided by the camera. This enumeration does not use any callback, but the caller passes index beginning with **Index** 0 and repeats the call with incremented **Index** until the call returns FALSE. Description string is passed similarly to the String parameter of the **cameraGetString** function. Color parameter hints the Windows color, which can be used to draw the filter name in the application.

```
BOOLEAN __cdecl cameraEnumerateFilters2(  
    CCamera *PCamera,  
    CARDINAL Index,  
    CARDINAL Description_High,  
    CHAR *Description,  
    CARDINAL *Color,  
    INTEGER *Offset  
);
```

Optional function.

This function is the same as **cameraEnumerateFilters** function but returns one more parameter **Offset**. This parameter indicates the focuser shift when each filter is selected.

Units of the **Offset** can be micrometers or arbitrary focuser specific units (steps). If the units used are micrometers, driver should return TRUE from **cameraGetBoolean(gbMicrometerFilterOffsets, ...)** call.

```
BOOLEAN __cdecl cameraSetFilter(  
    CCamera *PCamera,  
    CARDINAL FilterIndex  
);
```

Optional function.

Sets the required filter. If the camera is not equipped camera-controlled with filter wheel, this function has no effect.

Hint:

Even if the camera is equipped with filter wheel, the **cameraSetFilter** function may return FALSE. This typically indicates a (mechanical) failure of the filter wheel.

```
BOOLEAN __cdecl cameraReinitFilterWheel(  
    CCamera *PCamera  
);
```

Optional function.

The filter wheel performs the initialization, during which the zero-filter position is found and set.

Guiding functions

```
BOOLEAN __cdecl cameraMoveTelescope(  
    CCamera *PCamera,  
    INT16 RADurationMs,  
    INT16 DecDurationMs  
);
```

Optional function.

Instructs the camera to initiate telescope movement in the R.A. and/or Dec. axis for the defined period (in milliseconds). Parameters are passed as 16-bit integers and the sign define the movement direction in the respective coordinate.

Remark:

16-bit integer, holding time in milliseconds, limits the maximum pulse length to approx. 32.7 seconds.

If the camera is not equipped with autoguider port, this function has no effect.

```

BOOLEAN __cdecl cameraMoveInProgress(
    CCamera *PCamera,
    BOOLEAN *Moving
);

```

Optional function.

The function sets the **Moving** variable to TRUE if the movement started with **cameraMoveTelescope** call is still in progress.

If the camera is not equipped with autoguider port, this function has no effect.

GPS functions

```

BOOLEAN __cdecl cameraGetImageTimeStamp(
    CCamera *PCamera,
    INTEGER *Year,
    INTEGER *Month,
    INTEGER *Day,
    INTEGER *Hour,
    INTEGER *Minute,
    LONGREAL *Second
);

```

Optional function.

Obtains the exposure start time of the last acquired image, based on GPS time. The information about the time is available only after the **cameraReadImage** or **cameraReadImageExposure** call. The time information for the last read image remains valid until the next **cameraReadImage** or **cameraReadImageExposure** call (time instances are stored into internal FIFO the same way as entire frames are stored into FIFO created in camera on-board memory).

Warning:

Please note this function must be called after the **cameraReadImage** call beginning with the camera firmware v6.6 and later. Previous versions of camera firmware allowed the **cameraGetImageTimeStamp** call immediately after the **cameraImageReady** function returned TRUE, but prior to next **cameraStartExposure** call.

If the function returns FALSE, either the camera is not equipped with GPS receiver, or the GPS receiver did not have a valid time fix at the instance of exposure begin.

The presence of the GPS receiver in the camera can be tested using the **cameraGetBoolean(gbGPS)** call.

The exposure time stamp precision is better than 1 μ s. However, the GPS receiver must acquire at least 5 satellites to achieve such precision.

```

BOOLEAN __cdecl cameraGetGPSData(
    CCamera *PCamera,
    LONGREAL *Lat,
    LONGREAL *Lon,
    LONGREAL *MSL,
    INTEGER *Year,
    INTEGER *Month,
    INTEGER *Day,
    INTEGER *Hour,
    INTEGER *Minute,
    LONGREAL *Second,
    CARDINAL *Satellites,
    BOOLEAN *Fix
);

```

```

BOOLEAN __cdecl cameraGetGPSData2(
    CCamera *PCamera,

```

```

LONGREAL *Lat,
LONGREAL *Lon,
LONGREAL *MSL,
LONGREAL *GeoidSep,
INTEGER *Year,
INTEGER *Month,
INTEGER *Day,
INTEGER *Hour,
INTEGER *Minute,
LONGREAL *Second,
CARDINAL *Satellites,
BOOLEAN *Fix
);

```

Optional functions.

Read GPS location and current time in the instance of the call. Location coordinates **Lat** and **Lon** are provided in decimal degrees. North and East are positive directions, South and West are marked by negative numbers. MSL (mean sea level) is provided in meters.

The **cameraGetGPSData2** also returns Geoid separation value in addition to MSL.

The **Satellites** parameter indicates number of satellites tracked by the GPS receiver and the **Fix** variable indicates if the receiver acquired position fix.

The frequency of position updates can be between 15 seconds and 1 minute, depending on the GPS receiver module used (it is supposed the observing site location does not move and thus the location does not need to be updated very frequently). However, the time information, also provided by this call, does not contain the time of location information acquisition (it can be up to one minute old), but uses the same mechanism as the **cameraGetImageTimeStamp** call to get precise time. As the USB3 packet round-trip-time is typically around 0.1 ms, the time information can be obtained with better than 1 ms precision. But while the location information is available from only 3 acquired satellites (despite with limited precision), time information requires at least 5 satellites. So, if the location information is available, but the number of satellites is not high enough to provide precision time lock, the time-related parameters are returned all zeros.

Document updates

2019-05-14: Version 4.0 of driver DLLs released.

- Added preliminary support for C1 CMOS cameras.

2019-10-10: Version 4.1 of driver DLLs released.

- Added final support for C1 CMOS cameras.
- Added optional API functions **cameraSetGain** and **cameraConvertGain**.
- Added software binning and cropping (sub-frame) for Cx cameras.

2020-02-05: Version 4.2 of driver DLLs released.

- Modified the behavior of the optional **cameraBeginExposure** and **cameraEndExposure** functions, used as part of the Computer Timing Interface. The **cameraBeginExposure** call, if present, should not be called in addition to the **cameraClearSensor** and **cameraOpen** calls, but instead of them. Similarly, if the **cameraEndExposure** is present, it should be called instead of **cameraClose**, not in addition to **cameraClose** call.
- Added support for optional frame reading functions with 8-bit per pixel format **cameraGetFrameExposure8b/16b**, **cameraReadImage8b/16b** and **cameraGetFrame8b/16b**. The ***16b** versions of respective functions are only aliases to functions without bit size specifications, which always return 16-bit per pixel format.
- Added support for C2 line of cooled CMOS cameras.

- Added support for C1+ line of cooled CMOS cameras.

2020-04-23: Version 4.3 of driver DLLs released.

- Fixed bug causing wrong position of the sub-frame. This bug occurred only when reading image from Cx (CMOS) camera in 12-bit read mode and other than 1×1 binning. Other cameras, 8-bit read modes and unbinned images were unaffected.

2020-10-08: Version 4.4 of driver DLL released.

- Added new '**cxusb.dll**' driver file, supporting new C4-16000 scientific CMOS camera.
- Added hardware sub-frame (region-of-interest) handling for C1, C1+ and C2 CMOS cameras into '**gxusb**' driver.
- Please note the hardware sub-frame read requires camera firmware version 7.6 or higher.
- Added new optional API function **cameraAdjustSubFrame**, used to adjust coordinates of sub-frames of CMOS cameras to comply to limitations imposed by used sensors.

2020-12-16: Version 4.5 of driver DLL released.

- The '**cxusb.dll**' driver now supports C4-16000v2 scientific CMOS camera (PID 0C41H).

2021-04-30: Version 4.6 of driver DLL released.

- The '**cxusb.dll**' driver now supports 4th read mode **LoGain "16b"** for C4-16000 scientific CMOS cameras.

2021-05-28: Version 4.7 of driver DLL released.

- The '**cxusb.dll**' driver now supports hardware windowing on C4-16000 scientific CMOS cameras. The C4-16000 camera must use firmware version 4.4 or later to enable this feature.
- Preliminary support for the C1x and C3 cameras included in '**cxusb.dll**' driver DLL.

2021-09-10: Version 4.8 of driver DLL released.

- Fixed missing **giMaximalExposure** parameter in the **cameraGetInteger** function of the '**cxusb.dll**' driver DLL.
- Added **cameraSetGain** and **cameraConvertGain** function headers to the '**cxusb.h**' file.

2022-02-22: Version 4.9 of driver DLL released.

- The '**cxusb.dll**' driver newly supports hardware 2×2 binning of the C4-16000 scientific CMOS cameras. The C4-16000 camera must use firmware version 6.6 or later to enable this feature.
- The hardware binning is an optional featured, turned on/off by the '**HWBinning**' flag in the '**[device]**' section of the '**cxusb.ini**' configuration file.

2022-03-23: Version 4.10 of driver DLL released.

- The hardware 2×2 binning of the C4-16000 camera driver '**cxusb.dll**' is supported from firmware version 7.7, instead of 6.6 to fix some issues in sub-frame read.

2022-08-09: Version 4.11 of driver DLL released.

- The C1x and C3 camera driver '**cxusb.dll**' now supports hardware 2×2 binning. Note this functionality requires camera C1x and C3 firmware version 3.3 or later.
- The '**cxusb.dll**' driver now supports the new C5 camera line.
- Added '**BinningSaturate**' and '**BinningSum**' flags to the '**[driver]**' section of the '**cxusb.ini**' driver configuration file. These flags allow modification of the default binning behavior of all C1x, C3 and C5 cameras. Note this functionality requires camera firmware version 5.5 or later.

2022-09-03: Version 4.12 of driver DLL released.

- Added index '**gbGPS**' to '**GetBoolean**' call.
- Added functions **cameraGetImageTimeStamp** and **cameraGetGPSData**. These functions are available in the '**cxusb.dll**' and '**gxeth.dll**' drivers, which can handle new C5 cameras with GPS receiver.

2023-03-24: Version 4.13 of driver DLL released.

- Added support to C2-9000 camera.
- Added support for continuous (serial) image read. This mode is supported for C2-9000, all C1x, C3 and C5 cameras, providing the camera firmware is updated to version 6.6 or higher.
- Added **cameraReadImageExposure** function.
- Added **gbContinuousRead** index of the **cameraGetBoolean** function.
- Added support for hardware trigger input. The hardware trigger is supported by C1x cameras with GPS support.
- Added **cameraStartImageTrigger** function.
- Added **gbTrigger** index of the **cameraGetBoolean** function.
- Fixed the **cameraGetImageTimeStamp** for exposures shorter than frame digitization time. The returned exposure time is now corrected by all offsets, caused by exposure length, used sub-frame etc. and corresponds to the exact time of the first line exposure start.

2023-05-17: Version 4.14 of driver DLL released.

- Fixed problem in **cameraReadImageExposure** function, combined with hardware 2x2 binning.

2023-07-04: Version 4.15 of driver DLL released.

- The **cameraGetGPSData** function, used on cameras with new GPS receiver module version 2, could falsely return the Fix parameter set to FALSE, when the receiver reported the GPS satellite network in Differential mode in position fix. If the GPS satellite network remained in the Autonomous (non-differential) mode, the Fix parameter was properly set to TRUE.
- The **cameraOpen** and **cameraClose** functions are exported from the '**gxeth.dll**' and '**cxusb.dll**' driver libraries, despite they are not normally used in the **Camera Timing Asynchronous** interface, used by these drivers. The user's software can use these functions e.g. to temporarily close the sensor when a satellite passes over a field of view etc.

2023-08-03: Documentation update.

- Updated the **cameraGetImageTimeStamp** function description, the validity of the image GPS time stamp for the read image in regard to camera firmware version is specified.

2023-11-02: Version 4.16 of driver DLL released.

- Added serial image read support for C4-16000 cameras.
- Added support for GPS exposure timing for C3 and C2 cameras.

2023-12-08: Version 4.17 of driver DLL released.

- Added **cameraGetInteger** function index **giLineTime**.
- Added support for C0 cameras.
- Added support for C1 v3 (compact) cameras.
- Added support for C1+ v3 cameras.
- Added support for C2 v3 cameras.

2024-07-09: Version 4.18 of camera driver DLL released.

- The '**cxusb.dll**' and '**gxeth.dll**' now support the C2 cameras with global-shutter sensors and GPS receiver.
- The '**cxusb.dll**' and '**gxeth.dll**' now support C1+ cameras with rolling shutter sensors (C1+9000).

2024-09-19: Version 4.19 of camera driver DLL released.

- The '**cxusb.dll**' and '**gxeth.dll**' drivers fix the wrong image orientation of the C1+9000 camera.

2025-04-15: Version 4.20 of camera driver DLL released.

- The '**cxusb.dll**' and '**gxeth.dll**' drivers added support for the C2-46000 camera.
- The '**cxusb.dll**' and '**gxeth.dll**' drivers added support for GPS receiver on the C4-16000 camera.

2025-06-25: Version 4.21 of camera driver DLL released.

- The '**cxusb.dll**' and '**gxeth.dll**' drivers added support for the C1+46000 camera.
- Added **cameraGetInteger** function index **giBiasPixelValue**.
- The time returned by the **cameraGetInteger** function call with **giLineTime** index is newly updated with every GPS related call (**cameraGetImageTimeStamp** or **cameraGetGPSData**). The GPS receiver may increase the timing signal precision over time, e.g. with more acquired satellites, and thus the measured line digitization time can be determined more precisely.

2026-01-20: Version 4.22 of camera driver DLL released.

- The '**cxusb.dll**' and '**gxeth.dll**' drivers updated support for the C1+46000 and C2-46000 cameras to allow usage of the entire gain range available 0 to 1957.

2026-06-08: Version 4.23 of camera driver DLL released.

- The '**cxusb.dll**', '**gxusb.dll**', and '**gxeth.dll**' driver DLLs newly include an extended version of the API function **cameraGetGPSData2**, which returns also Geoid separation in addition in Mean sea level.

2026-08-11: Version 5.00 of camera driver DLL released.

- The new SDK documentation properly describes new driver API naming conventions (including of the “camera” prefix to every function name but keeping of the original names of API functions for compatibility reasons).
- The '**cxusb.dll**' and '**gxeth.dll**' drivers updated support for the C2 cameras with global shutter sensors and large onboard RAM (C2-7000R).