VRmUsbCam2 API

IMPORTANT NOTE: This documentation assumes the reader to be an experienced software developer who is familiar with state-of-the-art development platforms!

Table of Contents

1 Overview

The VRmUsbCam2 API offers fast and easy access to the VRmagic cameras and analog video converters. Main functionality of this interface is frame grabbing and device configuration. Supported host controllers depend on device type.

1.1.1 Streaming

Streaming devices require USB 2.0 (EHCI) host controllers.

- VRmC-3+(i)(/BW)
- VRmC-4(/BW)
- VRmC-4+(/BW)
- VRmC-8
- VRmC-8 with one external VRmS-8 sensor
- VRmC-8+
- VRmC-9/BW
- VRmC-9 with one external VRmS-9 sensor
- VRmC-9+/BW
- VRmC-12(/BW)
- VRmC-12(/BW) with one external VRmMS-12 sensor
- VRmC-12(/BW) with one external VRmS-12 sensor
- VRmC-12+(/BW)
- VRmC-14(/BW)
- VRmC-14(/BW) with one external VRmS-14 sensor
- VRmC-16(/BW)
- VRmC-16(/BW) with one external VRmS-16 sensor
- VRmC-18(/BW)
- VRmC-18(/BW) with one external VRmS-18 sensor
- VRmAVC-1
- VRmAVC-1+S
- VRmAVC-1+I
- VRmAVC-2

1.1.2 Smart

Smart devices support USB 1.x (UHCI/OHCI) and USB 2.0 (EHCI) host controllers.

- VRmFC-22(/BW)
- VRmFC-42(/BW)
- VRmMFC with VRmMSA2 frontend for up 4 external sensors
- VRmMFC with VRmMSARE1 frontend for up to 2 RE sensors

1.1.3 Intelligent

Intelligent devices support a local interface on the devices itself and support streaming over Ethernet using TCP and UDP.

- VRmDC-8
- VRmDC-9/BW

- VRmDC-12(/BW)
- VRmDC-12(/BW) with one external VRmS-12 sensor
- VRmDC-14(/BW)
- VRmDC-14(/BW) with one external VRmS-14 sensor
- VRmDC-16(/BW)
- VRmDC-18(/BW)
- VRmDFC-22(/BW)
- VRmDFC-42(/BW)
- VRmDMFC with VRmMSA2 frontend for up 4 external VRmMSC12 sensors
- VRmDC-X-E camera with one external sensor board
- VRmDAVC-2

2 VRmUsbCam2 C API

2.1 Project Handling

The most convenient way is to include `vrmusbcam2.h` and link against `vrmusbcam2.lib`.

In Microsoft Visual C++ you might use the following steps:

- In "Linker→Input→Additional Dependencies", add "vrmusbcam2.lib"
- In "Linker→General→Additional Library Directories", add the path "<InstallDir>\VRmUsbCam2 Library\lib", where <InstallDir> is the installation folder of the VRmagic USB Camera Development Kit, usually "C:\Program Files\VRmagic".
- In "C/C++→General→Additional Include Directories", add the path "<InstallDir>\VRmUsbCam2 Library\include", where <InstallDir> is the installation folder of the VRmagic USB Camera Development Kit, usually "C:\Program Files\VRmagic".

2.2 Error Handling

Every API function offers a return value of type `VRmRetVal` to indicate success or failure. If a function returns `VRM_FAILED` you can use `VRmUsbCamGetLastError()` to get an error description as C string. In order to obtain customer support you may also decide to enable the built-in logging features of the library by calling `VRmUsbCamEnableLogging()`. It is not recommended to use the logging feature in other situations.

2.3 Device Management

Two basic steps are required:

- **Device Scan**
  Before you are able to use a camera device you have to instruct the library to scan the USB bus for VRmagic devices using `VRmUsbCamUpdateDeviceKeyList()`. After this you can use `VRmUsbCamGetDeviceKeyListSize()` and `VRmUsbCamGetDeviceKeyListEntry()` to get a `VRmDeviceKey` for each device found. After usage the key has to be freed by using `VRmUsbCamFreeDeviceKey()`.

- **Open/Close Device**
  `VRmUsbCamDevice` is a handle to any single device. Use `VRmUsbCamOpenDevice()` with a `VRmDeviceKey` returned by `VRmUsbCamGetDeviceKeyListEntry()` to get a valid handle for a specific device. If this action is successful you can use the handle until you close it by using `VRmUsbCamCloseDevice()`.
  Note: *You can only open devices that are not busy (check* `!m_busy` *of* `VRmDeviceKey`).

2.4 Image Handling

The API supports multiple color formats (`VRmColorFormat`) and image modifier flags (`VRmImageModifier`). `VRmImageFormat` combines size (width and height), color format and bit combination of image modifier flags. The `VRmImage` is a struct built of a `VRmImageFormat` struct, an image buffer, pitch and a time stamp.
In addition, `VRmUsbCamGetFrameCounter()` can be used to obtain a frame counter for images originated by the frame grabber API.

Images can be created using one of the following functions:

- `VRmUsbCamNewImage()`
  allocates a new image of the given format.
- `VRmUsbCamCopyImage()`
  creates a new image as a copy of an existing image.
- `VRmUsbCamCropImage()`
  creates an image referring a rectangular section of an existing image. Note that the returned image is only valid as long as the image it originated from remains valid.
- `VRmUsbCamSetImage()`
  creates a container for an existing image represented by a format and a buffer.

Images obtained by one of these functions have to be freed by using `VRmUsbCamFreeImage()`.

A variety of color formats is supported:

- ARGB color format (32 bits/pixel)
- BGR color format (24 bits/pixel)
- RGB565 color format (16 bits/pixel)
- GRAY8 format (8 bits/pixel)
- YUYV color format (32 bits / 2 pixel)
- UYVY color format (32 bits / 2 pixel)
- GRAY10 format (16 bits/pixel) (S)
- BAYER8 color format (8 bits/pixel) (S)
- BAYER10 color format (16 bits/pixel) (S)

In addition, also compressed image formats exist:

- GRAY8/RLE (S)
  GRAY8 format (8 bits/pixel), byte-wise lossless RLE (see below)
- BGR/RLE (S)
  BGR color format (24 bits/pixel), byte-wise lossless RLE (see below)

Note: *Formats marked by (S) can only act as source formats (device raw formats) and will never be used as a conversion target format!*
`VRmUsbCamConvertImage()` copies the contents of the source to the target image, while converting each pixel to the target image format. Please note that the target image format must be one of the formats returned from `VRmUsbCamGetTargetFormatListSize[/Entry]()`, with the source format given as first parameter. Arbitrarily creating your own `VrmImageFormat` target format will probably result in an error when calling `VRmUsbCamConvertImage()`. The only parameters that you may safely change on an existing image format are the image format modifiers `VRM_HORIZONTAL_MIRRORED` and `VRM_VERTICAL_MIRRORED`.

Note: *Additionally you can get a* `VRmImage` *from the frame grabber API (see* `VRmUsbCamLockNextImage()`*) and apply the active converter Properties*

(`VRM_PROPID_CONVERTER_*`*) using* `VRmUsbCamGetTargetFormatListSize[/Entry]Ex()` *functions to create the* `VrmImageFormat` *target format.*

Note: *Conversions of source images of any "Bayer" Color Format can be accomplished by using a fast Low-Quality conversion or a slower High-Quality conversion. Default is Low-Quality, to enable High-Quality use the* `VRM_PROPID_CONVERTER_BAYER_HQ_B` *Property.*

Standard (rectangular) Image Format

By default, image data in memory is organized rectangular. The number of bytes belonging to one single image pixel thereby depends on the color format in use.
The image data buffer that is pointed to by the `mp_buffer` member of `VRmImage` stores each line of the image linearly, typically interrupted by padding bytes between them. The resulting line length (all data bytes + padding bytes) are referred to as image *pitch*.
So, offset #0 in the image data buffer stores the first byte of the top-left pixel of the image.

Note: *The top-left edge of the image data buffer may differ from the "physically top-left" edge of the image, this is indicated by the image modifiers* `VRM_HORIZONTAL_MIRRORED` *and* `VRM_VERTICAL_MIRRORED`.

Finding the data belonging to a pixel with coordinates (x,y) is easy: the corresponding byte offset in the image data buffer evaluates to *(x*pixeldepth+y*pitch)*, where *pixeldepth* is determined from the image's color format by `VRmUsbCamGetPixelDepthFromColorFormat()`.

Note: *The interlaced full-frame source formats of VRmAVC-1 and VRmFAVC-1, as indicated by the VRM_INTERLACED_FRAME image format modifier are organized slightly different: the upper half of the image refers to the first field and the lower half to the second field.*

Run-Length-Encoding (RLE) Format

RLE source formats are only available for smart cameras (VRmFC-x) with GRAY8 or BGR24 source formats. They are indicated by the `VRM_RUN_LENGTH_ENCODED` image modifier flag set.
The byte-wise lossless RLE compression is performed by the device, not by the host computer, thus it aids in reducing transfer data-rate and enables higher frame rates for suitable types of images.

Note: *The RLE compression algorithm in use is not guaranteed to produce less data than the uncompressed raw image consists of. Given a total amount of N data bytes, the resulting RLE image may grow up to 2*N data bytes. On the other hand, a uniformly colored image can be reduced to a total of 2*(N/255+1) data bytes.*

In general, images with extensive uniformly colored areas will result in good compression, whereas inhomogeneous images will more likely result in inflated data. The correction filter LUT is applied before the compression takes place, so one can use black-level, contrast or gamma settings to maximize compression gain.

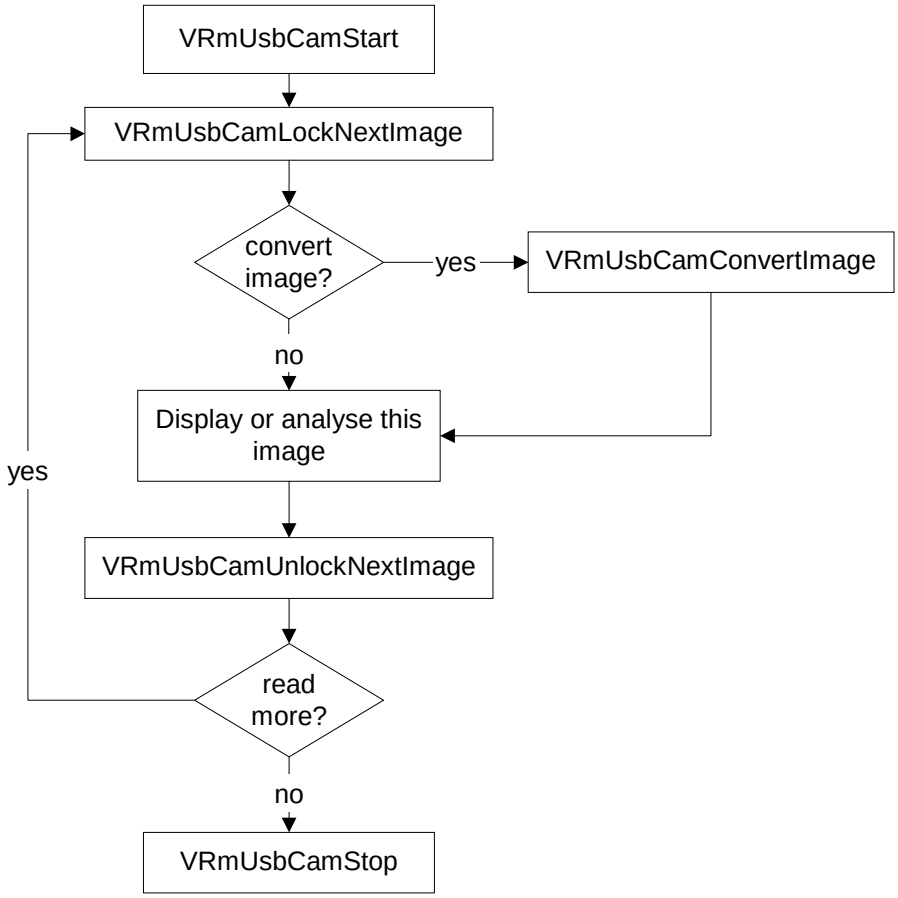
The RLE compressed data is grouped into byte pairs (V,L), starting from offset #0 in the image data buffer. Pitch information is ignored, so byte pairs are stored linearly without interruption until a special EOF (end-of-frame) marker.
Each byte pair consists of a "value" V and a "length information" L (number of repetitions of value V). The meaning of the value V depends on the color format in use.

Note: *The total amount of data bytes can be retrieved from* `VRmUsbCamGetImageBufferSize()`*. This can be used to check consistency of the RLE data by verifying that decoding ended after the given amount of data bytes.*

2.5 Frame Grabber

The VRmUsbCam2 library can grab continuous images from the camera devices. The grabbing process is started by the `VRmUsbCamStart()` function and stopped by `VRmUsbCamStop()`. Use `VRmUsbCamLockNextImage()`to access an image from the internal image queue (ring buffer). After you are done with processing the locked image, you have to use `VRmUsbCamUnlockNextImage()` to free the resources again.
The size of the internal ring buffer determines how many images can maximally be locked at a time. It is a property of the camera `VRM_PROPID_GRAB_HOST_RINGBUFFER_SIZE_I` (see section 2.7). The Smart devices additionally offer a ring buffer on the device the size of which can be modified using `VRM_PROPID_GRAB_DEVICE_RINGBUFFER_SIZE_I`.

A simple flow chart shows the basic process of grabbing one image at a time:



When grabbing, it is possible for the USB transfer from the device to stall. The result is that one or more images will be dropped and the transfer is restarted automatically. The library provides the `fp_frames_dropped` parameter of the `VRmUsbCamLockNextImage()` function to find out about such an event. Each image obtained by `VRmUsbCamLockNextImage()` has a time stamp given in milliseconds (`m_time_stamp`) and a frame counter (`VRmUsbCamGetFrameCounter`) which includes dropped frames and dropped triggers events.

This extract from the source code of the DirectDraw C++ Demo Application further clarifies the process:

```
// prepare image format for locked ddraw surface.
VRmImageFormat target_format;
if(!VRmUsbCamGetTargetFormatListEntryEx(device,0,&target_format))
    LogExit();

// start grabber at first
if(!VRmUsbCamStart(device))
    LogExit();

// and enter the loop
while (!g_quit)
{
    // lock the DirectDraw off-screen buffer to output the image to the screen.
    // The screen_buffer_pitch variable will receive the pitch (byte size of
    // one line) of the buffer.
    VRmDWORD screen_buffer_pitch;
    VRmBYTE* p_screen_buffer=DDrawLockBuffer(screen_buffer_pitch);

    // now, wrap a VRmImage around the locked screen buffer to receive the converted image
    VRmImage* p_target_img=0;
    VRmUsbCamSetImage(&p_target_img,target_format,p_screen_buffer,screen_buffer_pitch);

    // lock next (raw) image for read access, convert it to the desired
    // format and unlock it again, so that grabbing can
    // go on
    VRmImage* p_source_img=0;
    VRmBOOL frames_dropped;
    if(!VRmUsbCamLockNextImage(device,&p_source_img,&frames_dropped))
        LogExit();
    if(!VRmUsbCamConvertImage(p_source_img,p_target_img))
        LogExit();
    if(!VRmUsbCamUnlockNextImage(device,&p_source_img))
        LogExit();

    // see, if we had to drop some frames due to data transfer stalls. if so,
    // output a message
    if (frames_dropped)
        cout << "- frame(s) dropped -" << endl;

    // free the resources of the target image
    if(!VRmUsbCamFreeImage(&p_target_img))
        LogExit();
    // give the off-screen buffer back to DirectDraw
    DDrawUnlockBuffer(p_screen_buffer);

    // and update the screen
    DDrawUpdate();
}
// stop grabber
if(!VRmUsbCamStop(device))
    LogExit();
```

All devices support multiple source (“raw”) image formats. Use `VRmUsbCamGetSourceFormatListSize()` and `VRmUsbCamGetSourceFormatListEntry()` to retrieve the list of these formats. Use `VRmUsbCamSetSourceFormatIndex()` to activate one of them while the grabber is not running.

While for some applications it may be sufficient to process source images of the device, in most cases you will need to convert them to a well-known color format. This can be achieved by using the image conversion features of the API.
The functions `VRmUsbCamGetTargetFormatListSize()` and `VRmUsbCamGetTargetFormatListEntry()` provide access to the list of image formats the given source format can be converted to. Image conversion is then performed by `VRmUsbCamConvertImage()`.

Supported target image formats:
0: ARGB color format
1: BGR color format
2: RGB565 format
3: grayscale format
4: YUYV color format

Note: *When using the* `VRmUsbCamGetTargetFormatListSize[/Entry]Ex()` *functions the order of target image formats depends on the* `VRM_PROPID_CONVERTER_PREFER_GRAY_OUTPUT_B` *Property.*

2.6 Trigger Functions

Trigger functionality provides the ability to control the point of time of image exposure. There are two kinds of trigger signals a camera will accept:

- Soft Trigger
  `VRmUsbCamSoftTrigger()` triggers image exposure as fast as possible (this may require a firmware update, please contact VRmagic Support for details). Soft trigger can be used to synchronize several cameras without external wiring.

- External Trigger
  Here, image exposure is triggered in dependence of some external electrical signal. This is only supported by cameras with a corresponding external connector.

Not all devices support trigger functions. In order to see which trigger modes are available refer to the `VRM_PROPID_GRAB_MODE_E` Property for available options. To enable one or both trigger modes set the Property to the corresponding value.

The external trigger function can be configured as “edge” or “level” triggered with different polarities using the `VRM_PROPID_CAM_STROBE_POLARITY_E` Property.

2.7 Configuration Settings (Properties)

Various parameters can be changed to control each aspect of the device. Popular examples are “Exposure Time” or “Pixel Clock”. With v2.6.0.0, the VRmUsbCam2 API offers a consistent way of controlling these: the Property Interface. In previous versions of the API usage of structs like `VRmSettings1`, `VrmSettings2` etc. had been necessary to get similar results (for compatibility’s sake this is still possible, though *we strongly advise to use the advanced method described here*, since cross-device usage of code is somewhat encumbered when using the older scheme).
First of all, the Property Interface is capable of listing all parameters that the current device supports – along with a short description, a default value and (optionally) a range of allowable values. It also provides functions for setting or retrieving individual parameters.

A “Property” consists of an identifier (e. g. “`VRM_PROPID_CAM_EXPOSURE_TIME_F`”), a short description (e. g. “Exposure Time [ms]”), information about possible modification (i. e. if the property is writeable or read-only), the current value, the default value, and value range (i. e. information about minimum and maximum values). Some Properties also include a step-value, indicating the smallest possible modification of the value. Step-value of the Exposure Time Property for example is 0.1 [ms], which of course means that the smallest possible modification of exposure time is 0.1 ms.

Using the various information of a Property allows for easy handling. For example, changing the Exposure Time to 25ms can easily be implemented, as this extract from the C++ demo application shows:

```
float value=25.f;
if (!VRmUsbCamSetPropertyValueF(device,VRM_PROPID_CAM_EXPOSURE_TIME_F,&value))
    LogExit();
```

Properties come with different types of parameters. Parameter types are indicated in the identifier of the Property. There are integral types of parameters like *float* (thus, the identifier for the Exposure Time Property is "`VRM_PROPID_CAM_EXPOSURE_TIME_F`"; note the "`_F`"), *boolean* ("`_B`"), or *integer* ("`_I`") values as well as more complex types like *VRmPointI* ("`_POINT_I`"), *VRmRectI* ("`_RECT_I`"), or *VRmSizeI* ("`_SIZE_I`"). The set of supported parameter types can be found in the `VRmPropType` enumeration, included in the API headerfile "`vrmusbcam2.h`".
Every parameter type is addressed by a corresponding function. *Integer* Properties ("`_I`"), for example, can be modified by a function `VRmUsbCamSetPropertyValueI()`, whereas a *float* Property would need `VRmUsbCamSetPropertyValueF()`; again, the last character indicates the Property type.
A special parameter type is the *Enum* ("`_E`") type. Properties featuring this parameter type cannot include values of a certain range or of numeric quality. Instead, their value will be symbolic. Thus, the Property identified by `VRM_PROPID_CAM_VIDEO_STANDARD_E`, for example, can only have `VRM_PROPID_CAM_VIDEO_STANDARD_PAL` or `VRM_PROPID_CAM_VIDEO_STANDARD_NTSC` as valid values.

The „name" of a specific Property is represented by its identifier (such as `VRM_PROPID_CAM_EXPOSURE_TIME_F`). Note that the prefix "`VRM_PROPID_`" is specific to the C API and that it will be different in COM or .NET environments (see the relevant chapter below). A list of property identifiers is given in the `VRmPropId` enumeration included in the headerfile called "`vrmusbcam2props.h`".

Using the property identifier you can easily call for the current value of a specific Property. Again, this is illustrated by an example found in the C++ demo application:

```
float value;
if (!VRmUsbCamGetPropertyValueF(device,VRM_PROPID_CAM_EXPOSURE_TIME_F,&value))
    LogExit();
```

As you will see, `VRmUsbCamGetPropertyValue<x>()` is the function used, with `<x>` of course indicating the parameter type. In order to get any type-independent Property information use `VRmUsbCamGetPropertyInfo()`. This will provide a struct called `VRmPropInfo` containing the Property's id string, type, description and writeable flag. If you want to find out about the type-dependent information of a given Property, you may use `VRmUsbCamGetPropertyAttribs<x>()`. These will include default, min, max and step-value.

Note: *Not every device will support all possible Properties* (as they are given in the headerfile)!

An easy way to find out about the supported Properties of your device is to use the tool called *XmlDeviceInfo* included in this distribution[1]. It will be accessible via the start menu "VRmagic→VRmUsbCam2 Library→XmlDeviceInfo (VB.NET)" and generate a list of all supported Properties of a device connected to the computer and supported by the VRmUsbCam2 API. In case of multiple devices found, only Properties of the first are being captured.

[1] The .NET frameworks needs to be installed in order to use *XmlDeviceInfo*

A similar tool is also included as C++ (democ++deviceinfo) demonstrating the use of following C API calls: `VRmUsbCamGetPropertyListSize()`, `VRmUsbCamGetPropertyListEntry()`, `VRmUsbCamGetPropertySupported()`, and `VRmUsbCamGetPropertyInfo()`.

Almost all settings (current Property values) can be loaded from and saved to non-volatile memory on the device using `VRmUsbCamLoadConfig()` and `VRmUsbCamSaveConfig()` respectively. Naturally, the read-only Properties are excluded from loading.
Using `VRmUsbCamSaveConfig()`, you can create up to 9 different configurations, distinguished by a so-called *Configuration Id*. The first two Ids 0 and 1 are reserved, where 0 means "factory defaults" and 1 means "user defaults". Please note that you cannot overwrite the factory defaults.
At startup, the user defaults will be loaded automatically.
Once some specific configuration is no longer needed, use `VRmUsbCamDeleteConfig()` to remove it and release the memory occupied.
It is also possible to assign a custom description to each configuration, see Property `VRM_PROPID_GRAB_CONFIG_DESCRIPTION_S`.

With API version 2.6.1.0, the VRmUsbCam2 C API additionally provides you with a convenient way to configure the device by a graphical user interface[2] that is already known as an integral part of the VRmagic CamLab application and the VRmUsbCamDS Capture Source (see chapter 4).
The headerfile "`vrmusbcam2win32.h`" defines the functions necessary to do so: `VRmUsbCamCreateDevicePropertyPage()` and `VRmUsbCamDestroyDevicePropertyPage()`.

2.8 Callbacks

It is possible to register various callback hook functions at the VRmUsbCam2 API that will be called when certain events occur.

Currently there are two groups of callbacks:

- Static Callbacks (VRmUsbCam2 global)
  see `VRmUsbCamRegisterStaticCallback()` and `VRmUsbCamUnregisterStaticCallback()`
- Device Callbacks (related to a specific VRmUsbCam2 Device)
  see `VRmUsbCamRegisterDeviceCallback()` and `VRmUsbCamUnregisterDeviceCallback()`

The `VRmStaticCallbackType` and `VRmDeviceCallbackType` enumerations list all event types that can be registered for, respectively. Please note that you must register a certain callback function only once.
The static callbacks require a running GUI loop, i.e. usage of the Win32 functions `GetMessage()` and `DispatchMessage()`, meanwhile the device callbacks are called synchronously from within VRmUsbCam2 API calls.

2.9 User Data

In certain circumstances it may be convenient to store specific data not related to the device's operation as such (e. g. device location, maintenance procedures, pictures etc.) on the device itself. To address this need, every VRmUsbCam2 device offers the chance to store any such data in its non-volatile memory.
Within VRmUsbCam2 C API, this feature is accessible by four distinct functions:

[2] Currently only supported on Microsoft Windows

- `VRmUsbCamLoadUserData()` and `VRmUsbCamSaveUserData()` are used to load data from or store data to the device's non-volatile memory, respectively.
- `VRmUsbCamNewUserData()` and `VRmUsbCamFreeUserData()` are addressing the host's own memory management, (un)creating data needed for the load and store functions.

Only one unit of user data (i. e. one file) can be stored to any device at a time. Furthermore, memory space is limited. Older devices offer free user space of roughly 1KB, whereas current devices push the limit to around 25 KB. In order to find out about total memory and free memory (for user data, that is), call for the read-only Properties `VRM_PROPID_DEVICE_NV_MEM_TOTAL_I` or `VRM_PROPID_DEVICE_NV_MEM_FREE_I`, respectively.

Important Note: *Saving of user data via* `VRmUsbCamSaveUserData()` *might take up to 3 seconds. The device must not be disconnected from the host before the saving process is completed!*

3 VRmUsbCam2 COM and .NET API

Apart from the native VRmUsbCam2 C API that is provided by the `vrmusbcam2.dll`, COM (Component Object Model) classes have been registered by the Setup Wizard of the VRmagic USB Camera Development Kit.
These can be used to access all features of the VRmUsbCam2 C API 2 from within any programming language that is capable of dealing with COM objects. Simply instantiate[3] a `VRmUsbCam2` object of the COM library "`VRmagic VRmUsbCam COM API 2" (vrmusbcam2.dll)`.

Although .NET-based languages like *Microsoft Visual Basic .NET* or *Microsoft Visual C#* are capable of COM, yet another API is provided for this purpose: The *VRmUsbCamNET* assembly.
It will be installed by the Setup Wizard into the Global Assembly Cache if .NET Framework support is detected at the time of installation. **It is strongly recommended to be used from within .NET languages instead of the VRmUsbCam2 COM API.**
You can also find the assembly in the "VRmUsbCam2\wrappers\net" subdirectory of the installation folder of the Development Kit to easily access it from within your applications.

***Note*:** *When using the VRmUsbCamNET assembly, you should also import the System.Drawing assembly.*

The interfaces of the VRmUsbCam2 COM and .NET APIs are quite similar. Both are based on the interface of the VRmUsbCam2 C API described in the previous paragraphs. Hence the following part only points out the structural differences between using the COM/.NET API and the C API.

3.1 Naming Conventions

Whereas all type- and struct-identifiers of the C API start with the prefix "VRm" and all function-identifiers start with "VRmUsbCam2", the whole VRmUsbCam2 COM/.NET API resides in the VRmUsbCamCOM/VRmUsbCamNET namespace and lacks any prefixes, since they are not necessary.

3.2 Classes and Structs

The VRmUsbCam2 COM/.NET API arranges functions (methods) and data semantically, thus defining the following classes/objects within the respective namespace:

- `DeviceKey`
  a unique identifier of a VRmagic device; corresponds to VRmDeviceKey
- `ImageFormat`
  holds format information about a single image; corresponds to VRmImageFormat
- `Image`
  holds a single image, either read from the device directly or converted from another image read from the device; corresponds to VRmImage
- `Device`
  representation of a single VRmagic device; corresponds to VRmUsbCamDevice
- `DevicePropertyPage`
  provides a simple GUI allowing for configuration of common parameters of the Device; can be an independent parent or a child window
- `Exception` (.NET only)
  exception that is thrown by all classes/methods within the VRmUsbCam2 .NET assembly

[3] Microsoft Visual Basic 6.0 will auto-instantiate `VRmUsbCam2`, because it is declared as *Application Object*

- `VRmUsbCam2`
  container class for all "global" methods

Note: *In COM, ImageFormat and VRmUsbCam2 are the only user-creatable classes. All other classes can only by retrieved by member methods of VRmUsbCam2 COM API.*

In addition, structs are defined according to the C API:

- `PropInfo`
  receives general information about a device property; corresponds to VRmPropInfo. See section 2.7.
- `PropAttribs`
  receives default, min, max and step value of a device property; corresponds to VRmPropAttribs<x> structs. See section 2.7.
- `Size, Point, Rectangle` (.COM only[4])
  corresponds to VRmSizeI, VRmPointI and VRmRectI

3.3 Enums

Enumerations are defined in accordance with the C API, thereby all enumeration values lack the VRM_ prefix:

- `ColorFormat`
  corresponds to VRmColorFormat
- `ImageModifier`
  corresponds to VRmImageModifier
- `PropId`
  corresponds to VRmPropId. Note that .NET additionally lacks the PROPID_ *prefix of the enumeration constants*
- `PropType` (.COM only[5])
  corresponds to VRmPropType

3.4 Events

Callback definitions of the VRmUsbCam2 C API are mapped to the COM/.NET unified event model. The following classes are event sources:

- `VRmUsbCam2`
  provides `DeviceChange` events[6] in accordance to `VRmStaticCallbackType` enumeration. These events require a running GUI event loop.
- `Device`
  provides `PropertyListChanged`, `PropertyValueChanged`, `SourceFormatChanged`, `SourceFormatListChanged` and `TargetFormatListChanged` events in accordance to `VRmDeviceCallbackType` enumeration.
  These events are raised synchronously from within VRmUsbCam2 API calls.

[4] .NET uses built-in types
[5] .NET uses System.Type to distinguish device property types
[6] In .NET, these events are static/shared, ie. type members instead of instance members

3.5 Legacy Classes, Struct and Enums

For compatibility with older versions of the VRmUsbCam2 API, the COM and .NET interfaces still contain classes, structs, enums and methods that are not to be used in newer applications.
In COM, they are marked by "**OBSOLETE**" in their descriptions. Unfortunately, the VRmUsbCam2 .NET API currently does not support description/help strings for its methods, classes or structs. But instead, it provides two additional classes, namely `_obsolete_Device` and `_obsolete_VRmUsbCam` that contain the obsolete methods of the Device and VRmUsbCam2 class, respectively. In addition, these methods/properties are marked with an "Obsolete" attribute.

Most of the obsolete functionality has been replaced by Device Properties as described in section 2.7.

3.6 Error Handling

While error handling in the C API is performed solely by returning error codes (`VRmRetVal`) and supplying the `VRmUsbCamGetLastError()` function, the VRmUsbCam2 .NET API makes use of exceptions. Therefore, most methods do not offer a return value, but may throw a `VRmUsbCamNET.Exception`.
More similar to the C API, the VRmUsbCam2 COM API provides error handling by returning COM error codes (HRESULT). Additionally, all VRmUsbCam2 COM classes implement the IErrorInfo interface to offer written descriptions in case of an error, but most COM-capable programming languages use this interface automatically.

3.7 Properties

Since COM and .NET offer the property model, `VRmUsbCamGet<xxx>/Set<xxx>` functions of the C API are mapped to read-only or read-write properties of the classes `DeviceKey`, `ImageFormat`, `Image`, `Device` and `VRmUsbCam2`.

Note: *This is not to be mixed up with the Device Properties described in section 2.7!*

- `DeviceKey` class: read-only properties corresponding to members of `VRmDeviceKey` struct: `Busy, Manufacturer, Product, Serial.`
  In addition, `ProductId` and `VendorId` is provided which is obtained by `VRmUsbCamGetProduct/VendorId()` of the C API.

- `ImageFormat` class:
  - Array-type properties
    `TargetFormatList`
  - Properties corresponding to members of `VRmImageFormat` struct:
    `Size, ColorFormat, ImageModifier`
    Note that partial modification of the ImageModifier is possible by
    `FlipHorizontal()` and `FlipVertical()`

  In addition, the `PixelDepth` and, in case of Grabber Source Formats, a short `Description` is provided.
- `Image` class: according to the VRmImage struct of the C API, read-only properties are defined:
  `Buffer, ImageFormat, Pitch, TimeStamp`
  The `BufferSize` property presents the actual buffer size as obtained by `VRmUsbCamGetImageBufferSize()`. The `FrameCounter` is also provided as property. Direct access to the image buffer can be obtained by using `LockBits()` and `UnlockBits()` methods.
- `PropInfo` class: corresponding to VRmPropInfo struct, read-only properties are defined:
  `Description, Id, IdString, Type, Writeable`
- `Device` class:
  - Array-type properties
    `SourceFormatList, TargetFormatList`[7]`, PropertyList`
  - Read-only properties
    `DeviceKey, Running, PropertySupported, PropertyAttribs, PropertyInfo`
  - Read-write properties
    `SourceFormat, SourceFormatIndex, PropertyValue`
  - Events
    `PropertyListChanged, PropertyValueChanged, SourceFormatChanged, SourceFormatListChanged, TargetFormatListChanged`
- `VRmUsbCam2` class
  - Array-type properties
    `DeviceKeyList`
  - Read-only properties
    `CurrentTime, Version`

3.8 Interoperation with VRmUsbCamDS Capture Source

The VRmUsbCam2 COM/.NET API can be used to configure settings and to select the source format on a VRmUsbCamDS Capture Source filter.
The `VRmUsbCam2` class therefore provides the `AttachToVRmUsbCamDS()` method. Search the DirectShow demo application for an example.
When calling `AttachToVRmUsbCamDS()`, you obtain a `Device` object representing the VRmagic Device currently opened by the specified DirectShow Capture Source. Please note that some of the methods will not be usable on this object, since DirectShow retains full control over the device. Namely, the following `Device` methods are prohibited:
`Start(), Stop(), IsNextImageReady(), LockNextImage(), UnlockNextImage()`

When finished, simply release the `Device` object by the `Dispose()` method.

[7] In contrast to ImageFormat.TargetFormatList, this list is influenced by CONVERTER_xxx properties of the Device

4 DirectShow

VRmUsbCam2 DevKit will install two DirectShow Filters: VRmUsbCamDS and VRmImageConverter

4.1 VRmUsbCamDS

VRmUsbCamDS is a DirectShow Video Capture Source which can be instantiated once for every supported device. It features one output pin (use `FindPin` "1") providing images of different formats including timestamps for capturing. This output pin has a pin property page for device identification and source format selection.
In addition, various device parameters can be adjusted on the filter's property page.

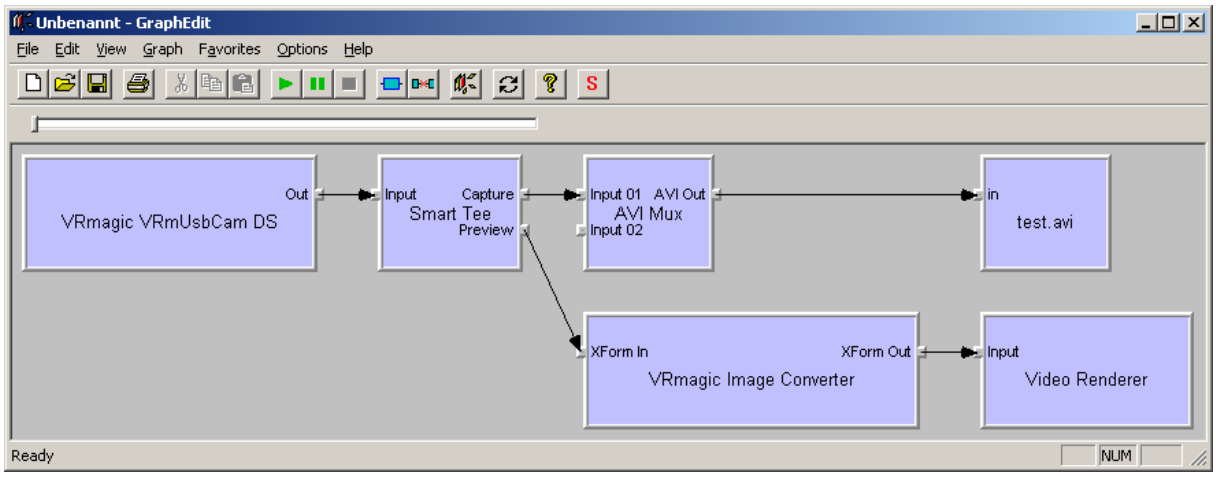
Supported output formats are:
#0 `MEDIASUBTYPE_VRMM` equivalent to source format (see section 2.5)
#1 `MEDIASUBTYPE_ARGB32` equivalent to target format ARGB32
#2 `MEDIASUBTYPE_RGB32` equivalent to target format RGB32
#3 `MEDIASUBTYPE_RGB24` equivalent to target format RGB24
#4 `MEDIASUBTYPE_RGB565` equivalent to target format RGB565
#5 `MEDIASUBTYPE_ARGB32` equivalent to target format ARGB32 (upside down*)
#6 `MEDIASUBTYPE_RGB32` equivalent to target format RGB32 (upside down*)
#7 `MEDIASUBTYPE_RGB24` equivalent to target format RGB24 (upside down*)
#8 `MEDIASUBTYPE_RGB565` equivalent to target format RGB565 (upside down*)

**upside down images (negative heights) are preferred by some video renderers*

In case of black and white cameras, an additional `MEDIASUBTYPE_RGB8` will be available. It will be inserted in the list of output formats either before each `ARGB32` or after each `RGB565` format, depending on the `CONVERTER_PREFER_GRAY_OUTPUT_B` device property.

In order to get a preview while capturing the usage of the Smart Tee Filter (installed by DirectX) is suggested.
For capture applications it is recommended to capture the native `MEDIASUBTYPE_VRMM`.

Sample graph for AVI capture including video preview:



In order to write a C++ program using these filters just include `vrmdshow.h` and use the defined GUIDs for filters and the VRMM mediasubtype.

It is possible to access the `VRmUsbCamDevice` of every VRmUsbCamDS instance using the `AttatchToVRmUsbCamDS()` methods of the VRmUsbCam2 COM or .NET API (see section 3.8).

4.2 VRmImageConverter

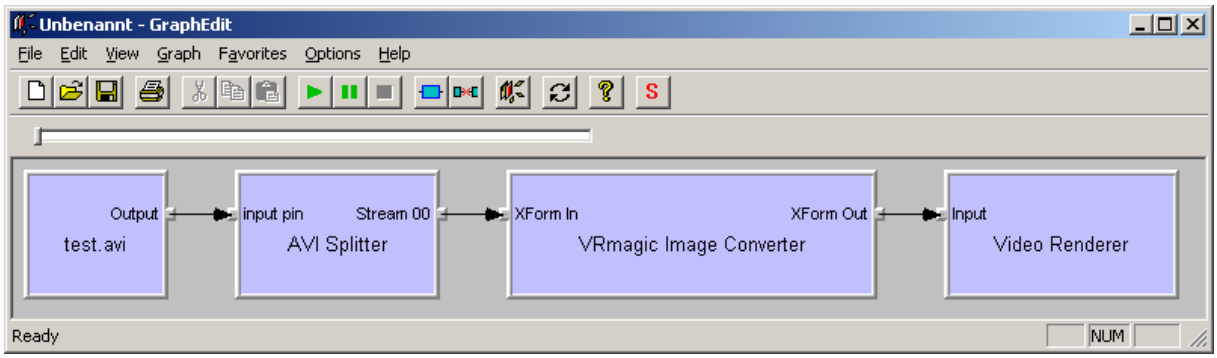
VRmImageConverter is a DirectShow Video Transform Filter which can convert a `MEDIASUBTYPE_VRMM` input to different RGB formats:

Supported output formats:
#0 `MEDIASUBTYPE_ARGB32`
#1 `MEDIASUBTYPE_RGB32`
#2 `MEDIASUBTYPE_RGB24`
#3 `MEDIASUBTYPE_RGB565`
#4 `MEDIASUBTYPE_ARGB32` (upside down*)
#5 `MEDIASUBTYPE_RGB32` (upside down*)
#6 `MEDIASUBTYPE_RGB24` (upside down*)
#7 `MEDIASUBTYPE_RGB565` (upside down*)

**upside down images (negative heights) are preferred by some video renderers*

In case of black and white cameras, an additional `MEDIASUBTYPE_RGB8` will be available. It will be inserted in the list of output formats either before each `ARGB32` or after each `RGB565` format, depending on the `CONVERTER_PREFER_GRAY_OUTPUT_B` device property.

Sample graph for displaying an AVI captured in `MEDIASUBTYPE_VRMM`:



5 History

API v2.8.1.x or later	see “VRmUsbCam2 Devkit Changelog.pdf” for details
API v2.7.2.7	fixed occasional startup problem of VRmAVC-1(+I/+S) with NXP chip (requires firmware version >= v21.99)
API v2.7.2.6	fixed image order and config import of VRmMFC
API v2.7.2.5	added bayer 10bit format to VRmMFC fixed multithreading issue in "UnlockNextImage"
API v2.7.2.4	added support for VRmC-4+ and VRmC-12+ added support for gray 16bit and BGR 48bit target formats improved startup behavior and added support for RLE compressed bayer format to VRmMFC
API v2.7.2.3	improved deserializer of VRmMFC
API v2.7.2.2	added support for VRmAVC-1+I added support for selection of TFF/ BFF formats to VRm(F)AVC-1 (might require a firmware update) fixed sync-freerunning mode of VRmDC-12
API v2.7.2.1	fixed minor documentation issues
API v2.7.2.0	added support for VRmMFC (Smart Multi-Sensor Camera) added support for VRmDC-8, VRmDC-9/BW, VRmDC-12 added support for ethernet receiving added support for VRmAVC-1+S added support for VRmSM-1 added possibility to lock mutiple images of host ring buffer
API v2.6.7.8	fixed non working AttachToVRmUsbCamDS fixed timing changes in freerunning sequential mode fixed trigger timeout for VRmFC-x devices fixed maximal pixel clock for VRmFC-6
API v2.6.7.7	added support for binning modes to VRm(F)C-12/BW added support for overclocking to VRmC-3+ and VRm(F)C-12 added support for external sensor models of VRmFC-12 added auto pixel clock for VRmFC-6
API v2.6.7.6	added support for external sensor models of VRmC-12 added support for TTL trigger/strobe of VRmC-8+/VRmC-9+ Rev 1.1 added channel balance plugin to improve image quality of VRmC-9(+) fixed maximum value of trigger timeout property reduced jitter for VRmFC-x cameras

API v2.6.7.5	added property for roi of auto exposure, with default = center 1/9 of image
API v2.6.7.4	added support for VRmC-9+ fixed incompatibility of VRmUsbCamDS with Adobe Flash Media Encoder
API v2.6.7.3	added support for synchronized free-running grabbing mode (VRmC-12) (might require a firmware update)
API v2.6.7.2	fixed auto-white balance that was accidentally disabled
API v2.6.7.1	added support for VRmC-8+ added subsampling, auto-exposure and optimized 10bit evaluation for VRmFC-x cameras
API v2.6.7.0	built with VS2005 SP1, Visual C++ Runtime 8.0 SP1, .NET Framework 2.0 (no longer supports Framework 1.x) fixed problem of unlit images in free-running and soft/edge triggered modes Device Property Page now runs in its own thread
API v2.6.6.3	added error handling for trigger stalls
API v2.6.6.2	fixed loading/switching between user ROI configs
API v2.6.6.1	added support for VRmFC-6(/BW) updated source format list of VRmFC-4/8/9
API v2.6.6.0	added support for VRmFC-4, VRmFC-8, VRmFC-9, VRmFC-12 added RLE compressed source format for VRmFC-x added device callbacks (C) and events (COM, .NET) fixed user roi of VRmC-3+ and VRmC-12
API v2.6.5.0	added support for VRmFAVC-1 added support for VRmCI enhanced user settings to support 9 different configs
API v2.6.4.0	added VRmUsbCamIsFirmwareCompressionRequired
API v2.6.3.0	added image analysis plugin (chessboard + concentric marker) added defective pixel management (DPM) added property for "images ready in host ringbuffer"
API v2.6.1.0	improved precision of strobe output and trigger timeout (might require a firmware update) added free-running sequential mode added support for VRmC-3+(/BW) added blacklevel property to filter adjustable pixel clock for VRmC-12 (5 to 26.6 MHz) fixed VRmUsbCamReloadUserSettings

API v2.6.0.0	added property based configuration interface added support for VRmC-9/BW and VRmC-12(/BW) added properties for converter (flips, BayerHQ, prefer gray), multi channel filter settings (R/G/B), plugins for auto exposure, auto white balance and auto reset level calibration (for VRmC-3 and VRmC-4) added frame counter information to image fixed negative luminance values
API v2.5.0.0	completed support for VRmC-8pro added shutter config (might require a firmware update) added trigger timeout added user data storage (in eeprom) improved user roi handling added YUYV as target format to image converter
API v2.4.0.0	added optional High-Quality Bayer Filter added basic support for VRmC-8pro
API v2.3.0.0	added VRmUsbCam2 COM API v2 added soft trigger added COM/.NET interoperability with VRmUsbCamDS Capture Source re-designed .NET API v2 to match COM API v2 added source format to load/save settings
API v2.2.0.0	added support for VRmC-4pro v2 and VRmC-6pro added trigger controls
API v2.1.0.1	added access to timer enhanced target format handling: added support for RGB565, fixed horizontal mirrored, target format handling is now device independent
API v2.0.2.6	added support for VRmC-OEM-1
API v2.0.2.5	initial API v2.x release

6 Migration Hints

C++ API v1.x to C API v2.x

- Replace the access via instances of `VRmUsbCam2` by the device handle `VRmUsbCamDevice`
- Replace `GetNextImageBGRA` (or `GetNextImageRAW`) by consecutive `VRmUsbCamLockNextImage` and `VRmUsbCamUnlockNextImage`, you may use `VRmUsbCamConvertImage` between these calls to convert this source image to different target formats
- Camera Configuration is now device type dependent: use `VRmUsbCamGetFeatures` to check for accessible camera settings via `VRmUsbCamSetSettingsX` / `VRmUsbCamGetSettingsX`

C API v2.0.x to C API v2.1.x

- Replace `VRmUsbCamGetDevice` in `VRmUsbCamGetTargetFormatListSize` by the active source format given in `VRmUsbCamGetSourceFormat`

C API v2.6.0.0 to C API v2.6.1.0

- Replace VRmUsbCamLastErrorWasTransferTimeout by VRmUsbCamLastErrorWasTriggerTimeout

.NET API v1.x to .NET API v2.x

- VRmUsbCam2.tDeviceId class
  - Rename: `VRmUsbCam2.tDeviceId` => `DeviceKey`, rename members:
    - `name` => `Product`
    - `serial` => `Serial`

- VRmUsbCam2 class
  - The following methods have to be remapped / renamed:
    - `CheckFramesDropped()`: use `framesDropped` parameter of `LockNextImage()`
    - `GetNextImage()`: replace by `LockNextImage()`, `ConvertImage()`, `UnlockNextImage()`
      Example code (VB.NET), where cam is an instance of VRmUsbCam2:
      ```
      Dim framesDropped As Boolean
      Dim convimg As New VRmagic.Image(cam.targetFormats(0))
      Dim rawimg As VRmagic.Image =
      cam.LockNextImage(framesDropped)
      cam.ConvertImage(rawimg, convimg)
      cam.UnlockNextImage(rawimg)
      ```
    - `GetTimeStamp()`: use `TimeStamp` property of `VRmagic.Image` class
    - `ScanForDevices()`: use `UpdateDeviceKeyList()` instead and examine the `DeviceKeyList` property
    - `xxxAdjustments()`: rename to `xxxSettings()`
    - `SkipNextImage()`: replace by `LockNextImage()`,`UnlockNextImage()` pair
  - The following properties have to be remapped/renamed:
    - `DefaultROI`: removed without replacement (but selecting a source format by the `SourceFormat` property restores the ROI to its defaults)

- `DeviceId`: rename to `DeviceKey`
- `Red/Green/BlueGain, ExposureTime, PixelBiasVoltage, PixelClockMhz, ResetLevel, ROI, SensorSize`:
  use Settings1.xxx property instead and rename:
  `ExposureTime` => `ExposureTimeMs`
  `ImageROI` => `ROI`
- `Gamma`: use `Settings3.Gamma` property instead
- `IlluminationIntensity`: use `Settings2.IlluminationIntensity` property instead
- `IsIlluminationAvailable`: check whether `FeaturesType.SETTINGS2_SUPPORTED` flag is set in `Features` property
- `SupportedPixelFormats`: use `TargetFormatList` property instead. The basic `System.Drawing.Imaging.PixelFormat` has been replaced by the more complex `VRmagic.ImageFormat`, but can be retrieved by its `ToPixelFormat()` method.

.NET API v2.0.x to .NET API v2.1.x

- `VRmUsbCam2.TargetFormatList`: use `VRmUsbCam2.SourceFormat.TargetFormatList` property instead

.NET API v2.1.x to .NET API v2.2.x

- `VRmUsbCam2.Bitmap`: removed, use `VRmUsbCam2.ToBitmap()` method instead. This reflects the fact that a pitch conversion may be performed

.NET API v2.2.x to .NET API v2.3.x

- The following namespaces have to be remapped/renamed:
  - `VRmagic`: rename to `VRmUsbCamNET`

- The following classes/types have to be remapped/renamed:
  - `(VRmagic.)VRmUsbCam2`: rename to `VRmUsbCamNET.Device`, except regarding the following methods and properties that still remain in `VRmUsbCamNET.VRmUsbCam2`:
    `EnableLogging(), UpdateDeviceKeyList(), RestartTimer(), CurrentTime, DeviceKeyList`
  - all `VRmUsbCam2.<xxxx>Type`: rename to `VRmUsbCamNET.<xxxx>` (note the missing "Type" suffix)
  - `VRmUsbCamException`: rename to `VRmUsbCamNET.Exception`
- The following methods have to be remapped/renamed:
  - `New ImageFormat(...)` (constructor): removed. Do not create ImageFormats on your own. Instead, only use the formats in the lists `Device.SourceFormatList` and `ImageFormat.TargetFormatList`
  - `New Image(...)` (constructors): use `VRmUsbCam2.NewImage()`, `VRmUsbCam2.SetImage()` and `Image.Clone()` methods
  - `Image.ConvertImage`: use `VRmUsbCam2.ConvertImage()` method instead
  - `New VRmUsbCam2(...)` (constructor): use `VRmUsbCam2.OpenDevice()` method instead

- The following properties have to be remapped/renamed:

- `Image.ImageModifier`: is no longer writable, use `FlipHorizontal()` and `FlipVertical()` instead

.NET API v2.6.6.x to .NET API v2.6.7.x

- **The assembly is now based on .NET Framework 2.0 and thus no longer compatible to applications based on earlier versions of the Framework.**

  In order to use VRmagic devices from within .NET applications based on **Framework 1.x**, you can use the **VRmUsbCam2 COM API via .NET COM Interop** instead, however functionality will then be limited:

  - The `LockBits()` and `UnlockBits()` methods of the Image object cannot be used, so image data cannot be modified.
    Note: read access to the actual image data is provided by the To…Array() methods.
  - The `PropertyAttribs` property of the Device object is inaccessible because of a bug in COM Interop.

  If you do not need these features, you can still use the VRmUsbCam2 API from your .NET 1.x application by applying the following adaptions to your existing .NET source code:

  - Replace all references to `VRmUsbCamNET` by `VRmUsbCamCOM`
  - Add a reference to `stdole` assembly
  - Statically instantiate an instance of `VRmUsbCamCOM.VRmUsbCam2` somewhere in your code and name it `VRmUsbCam2`
  - Replace all array types returned by any member method or property of any VRmUsbCam2 class by `System.Array`
  - Replace occurrences of `System.Drawing.Size`, `Point` or `Rectangle` by their corresponding classes in `VRmUsbCamCOM`, use `New…()` methods of the static `VRmUsbCam2` object for creating them
  - Catch `System.Runtime.InteropServices.COMException` instead of `VRmUsbCamNET.Exception` and investigate the `ErrorCode` member instead of the `Number` member
  - Add the `PROPID_` prefix to all property identifiers used
  - Replace calls to `Image.ToBitmap()` and `Image.ConvertToBitmap()` by calls to `Image.ToPicture()`, `ConvertToPicture()`…,
    if required, use the HDC of your target graphics object as argument to each method, finally, use the `Drawing.Bitmap.FromHbitmap()` static method to create a `Drawing.Bitmap` from the Handle of the Picture object.
    This C# example shows how to extract a `Bitmap` from a VRmUsbCam2 COM API `Image` object (`vrmImage`):

    ```
    stdole.IPictureDisp tmppic = vrmImage.ToPictureNoDC();
    Bitmap tmpbitmap = Bitmap.FromHbitmap(new IntPtr(tmppic.Handle));
    System.Runtime.InteropServices.Marshal.ReleaseComObject(tmppic);
    …do what you want with tmpbitmap here…
    tmpbitmap.Dispose();
    ```