STEMMER IMAGING

Application Note

Common Vision Blox Application Note

Interfacing Common Vision Blox to 3rd Party Imaging Libraries

Version 1.2.0 from 29/07/2019

How to use Common Vision Blox together with libraries like OpenCV, IPP or others.

ABSTRACT: Although Common Vision Blox comes with a variety of algorithms tailored to many different imaging and machine vision applications, there are sometimes good reasons to combine Common Vision Blox with other libraries in one application.
Luckily, Common Vision Blox provides the necessary utilities to do so – sometimes even at minimal or almost no CPU cost. This guide shows how to use these utilities.

Copyright

Content

Content

1. About this guide 4
2. Basic Concepts 5
   Image Models 5
   The Image Model of CVB 5
   Data Types 5
   Memory Layout 7
   Lifetime Management 7
   The Image Model of OpenCV 8
   Data Types 8
   Memory Layout 8
   Lifetime Management 8
   The Image Model of IPP 9
   Data Types 9
   Memory Layout 9
   Lifetime Management 9
3. Data Transfer from 3rd Party Libraries to CVB 10
4. Data Transfer from CVB to 3rd Party Libraries 15
   Transfer Without Copy 15
   OpenCV 15
   IPP 20
   Transfer With Copy 20
   OpenCV 21
   IPP 24
5. Other Programming Languages 25
6. Appendix/ Contact and Feedback 26
   Version History 26

CVB Interoperability

1. About this guide

To avoid lengthy wordings we will use the following abbreviations throughout this text:

CVB as always is short for Common Vision Blox

IPP Short for “Intel Performance Primitives”, a highly optimized and renowned imaging algorithm library

VPAT Virtual Pixel Access Table – an image data access scheme available in CVB.

Although we have chosen OpenCV and IPP as two exemplary, generally available libraries that have no affiliation to a company that is actively selling their own products on the machine vision markets, it is evident that the methods and approaches outlined in this document are applicable to any library that provides access to the necessary data structures and information. Therefore it is only reasonable to assume that with the information provided herein, it is also possible to interface CVB to e.g. Teledyne Dalsa’s Sapera® or MVTec’s Halcon®.

All products and libraries are property of their respective trademark holders and mention of these libraries/products does not imply an endorsement of Common Vision Blox by these companies.

Please note that it is assumed that the reader has a firm grasp of C/C++, especially the use of pointers and pointer arithmetic – both are going to be used extensively throughout this document, but will not be explained in detail. It is also assumed, that the reader is sufficiently familiar with Common Vision Blox and either OpenCV and/or IPP, as neither of those libraries will be explained beyond the extent immediately necessary for this guide.

With the exception of chapter 2, the chapters of this guide may be read independently or in sequence. Their summarized contents are:

- **Chapter 2** Explains some basic concepts that will be the foundation of the methods explained in chapter 3 and 4. It is recommended to read this chapter first, especially if you are not yet very familiar with the image model of CVB.
- **Chapter 3** Explains how images from a 3rd party library may be used inside CVB.
- **Chapter 4** Explains how images from CVB may be used inside a 3rd party library.
- **Chapter 5** Gives hints about transferring the recipes outlined in chapter 3 and 4 from C/C++ (which is the main focus of this guide) to other programming languages.

CVB Interoperability

2. Basic Concepts

Image Models

The fundamental reason why it is not possible to simply pass images/image data between any pair of two machine vision libraries (or sometimes even between the Windows SDK or the Common Language Runtime and a machine vision library) is that there is no commonly agreed format for the description of images that is understood and accepted by more than one library (or at best: by more than just a few libraries).

The main reasons for this are…

… the fact that different libraries emphasize different aspects of image processing; for example, OpenCV and IPP don't need to concern themselves too much with image acquisition and therefore do not need something like the ring buffer handling that is found e.g. in CVB or Sapera.

… different capabilities of different libraries; floating point valued images are for example of no practical use to the Windows SDK, whereas in CVB and in IPP they play an important role for a variety of operations like FFT or Wavelet Transforms.

… different approaches to the representation of image data in a particular library; for example the VPAT access interface in CVB allows for variable views to the same image data[1] and used to be a particularly valuable asset at a time when almost no frame grabber had the necessary hardware on board to sort image data from multi tap cameras into a proper memory layout.

Not surprisingly the key to bridging the gap between any two libraries is to translate and/or transform the image model of library A to fit that of library B. Often (but not always) this can even be achieved without copying the image data. Of course, doing so requires knowledge of the data structures involved as well as the capabilities of the libraries involved.

The Image Model of CVB

Data Types

Data type in this context refers to the data type of an individual pixel inside an image (i.e. if the memory address of a pixel is known, what data type will the pointer need to be cast to for dereferencing it properly).

CVB stores all the information it provides about the data type of an image's pixel in a 32 bit value that is composed as follows:

[1] This allows e.g. for RGB←→BGR transformation, simple deinterlacing or resizing of image data without actually transforming the image's pixel data in memory but through manipulating the VPATs that define the software's perception of the image data

CVB Interoperability

Bit	31	…		10	9	8	7	6	5	4	3	2	1	0
Purpose	reserved			overlay?	float?	signed?	bits per pixel							

- Bit 0 to 7 specify the number of bits per pixel that are being used[2]
- Bit 8 indicates whether or not the pixel data type is signed
- Bit 9 indicates whether or not the pixel data type is float[3]
- Bit 10 indicates whether or not bit 0 is to be interpreted as an overlay mask

Strictly speaking, the 32 bit value specifies the data type of an image plane's pixels. CVB in theory offers the possibility of having image planes with different data types, but this not commonly used and one would in fact need to go to great lengths to construct such an image using CVB functions.

It is obvious that in situations where the data type can be specified freely (for example when calling `CreateGenericImageDT` or `CreateImageFromPointer`), it is easily possible to build nonsensical combinations like 7 bit floating point – the functions will not return an error but do as they're told. Therefore common sense should be applied when constructing a data type descriptor.

If we ignore the reserved bit and the overlay mask bit (which has no representation in the OpenCV and IPP pixel data types), then we end up using the following data type descriptors for the most commonly used pixel data types[4],[5]:

Descriptor & 0x3FF	C/C++ data type	Descriptor & 0x3FF	C/C++ data type
0x00000008	`unsigned char`	0x00000108	`(signed) char`
0x00000010	`unsigned short`	0x00000110	`(signed) short`
0x00000020	`unsigned int`	0x00000120	`(signed) int`
0x00000220[6] 0x00000320	`float`	0x00000240 0x00000340	`double`

[2] Note that this number gives the number of bits per pixel that have actually been used. The number of padding bits that might have been added is not reflected in the data type descriptor. For example the "bits per pixel" value may10, but the number of bits consumed in memory by each pixel is in this case at least 16.

[3] If this bit is *not* set, it is safe to assume that the pixel data type is an integer type.

[4] Two additional descriptors that are being used occasionally are 0x0000000A and 0x0000000C which correspond to 10 and 12 bit unsigned integer data. As there are no built-in types with that size in any commonly used programming language it is up to the user to make sure that during operations on images of this type the range of valid values is not exceeded. Internally these formats are usually stored in 16 bits per pixel and can be accessed like 16 bit integer data (i.e. through an `unsigned short*`).

[5] The setting of the signed-ness bit is irrelevant with floating point valued data types as these are always assumed to be signed. [7] Note that the widespread notion of considering RGB a 24 bit per pixel format does not apply to CVB.

CVB Interoperability

CVB treats the pixel data type, the memory layout, and the colour/channel interpretation of an image completely separate. Of course monochrome images are just images with one channel (usually called "plane" in the CVB documentation). RGB images are simply images with three such channels, but so would be HSI or Lab images as well. That means that the user is responsible for keeping track of the colour format he is currently working on[6]. The Common Vision Blox Display, when confronted with an image with 3 or more planes, will simply assume that plane 0 contains the red pixels, 1 one contains the green pixels and plane 2 contains the blue pixels.

Memory Layout

CVB implements access to the image data through the so-called VPATs, two tables with offsets in x and y direction[7] that can be combined with a base pointer to calculate the address of a pixel in memory. The base pointer and VPAT for an image plane can be queried using the function `GetImageVPA`:

```
// get base pointer and VPAT address
intptr_t pBase = 0;
PVPAT    pVpat = nullptr;
GetImageVPA(cvbImg, 0, reinterpret_cast<void**>(&pBase), &pVpat);
// calculate address of pixel x = 19, y = 21:   intptr_t
pPixel = pBase + pVpat[19].XEntry + pVpat[21].YEntry;
```

This might look like an unnecessary overhead when compared to other libraries that often simply work on a base pointer and an increment in y direction to calculate a pixel's address. But the VPATs not only allow CVB to easily cover practically all memory layouts that a 3rd party library may use – they also allow for neat tricks like merging two separate images into one (see function `CreatePanoramicImageMap`), rotating an image (see function `CreateRotatedImageMap`) or extracting a scaled portion of an image (see function `CreateImageMap`) – all without copying a single pixel.

Lifetime Management

Defining the lifetime of a large data object (like an image acquired from a camera that may easily contain several megabytes of pixel data) is not as trivial as it may sound, and there is more than just one approach available for this issue[8]. In CVB we have opted for the implementation of a reference counting model:

- Every image object (i.e. every object access through a handle of type `IMG`[9]) carries its own reference counter.
- Every time the handle is passed to a consumer (a new function, a different thread or just a new copy of the handle) the reference count is/should be increased by calling `ShareObject` once.

---

[6] In situations where an image has been created through one of the color space conversion functions of the CVB Foundation Package, it is actually possible to determine the color format using the function `ImageColorModel`.

[7] Strictly speaking it is a table of structures that hold an x offset and a y offset.

[8] Probably the two most commonly encountered patterns here are alloc/free (i.e. there's a malloc-like allocation function and free-like cleanup function) and reference counting.

[9] In fact the reference counting model in CVB is not limited to image objects – it also applies to several other stateful objects in CVB.

- Every time a consumer stops using a handle, its reference count is/should be decreased by calling `ReleaseObject` once.
- If the reference count reaches 0, the image memory is freed.

This is important to be aware of, because other libraries typically follow different approaches. IPP for example simply provides functions to allocate and free image memory (and it is the caller's responsibility to make sure both are being called, and at the right time), while e.g. Sapera uses C++ objects and the image memory's lifetime is tied to the lifetime of the containing Sapera object instance.

The Image Model of OpenCV

Data Types

OpenCV typically describes the images it is working on using a structure called `IplImage` which was actually borrowed from the Intel Image Processing Library. The supported pixel data types for the OpenCV library are listed in the description of the `depth` member of the `IplImage` structure. The supported pixel data types are:

- 8 bit signed and unsigned integer
- 16 bit signed and unsigned integer
- 32 bit signed integer
- 32 bit float
- 64 bit float

OpenCV supports images with 1 to 4 channels (`nChannels` member of `IplImage`). All channels share the same data type, and the memory layout may in theory be interleaved or planar, with interleaved being the predominant alternative (see description of the `dataOrder` member of `IplImage`).

Memory Layout

The memory of an OpenCV image is accessible through the `imageData` pointer in the `IplImage` structure[10]. The data pointer together with the `widthStep` member (y increment) allow for the calculation of the address of each pixel in the image.

Lifetime Management

An OpenCV image comes into existence when it is created using one of the creation functions (like e.g. `cvCreateImage`) and can be destroyed using e.g. `cvReleaseImage`.

[10] This refers to the C interface of OpenCV. If you are using the C++ interface of OpenCV please to the documentation to map the description to the C++ interface.

CVB Interoperability

The Image Model of IPP

The image processing functions in the IPP (Intel® Integrated Performance Primitives for Intel® Architecture) are a little more basic than those of the OpenCV library. Unlike OpenCV, IPP does not use a structure to collect the information necessary to process image data – memorizing and organizing that data is solely the programmer's responsibility.

Data Types

On functions that process image data, IPP supports the following data types:

- 8 bit signed and unsigned integer
- 16 bit signed and unsigned integer
- 32 bit signed integer and 32 bit float
- 32 bit signed integer and 32 bit float complex[11]

Like OpenCV, IPP supports images with 1 to 4 channels. In case of 4 channels, one channel may be used for transparency information (alpha channel).

Memory Layout

Like OpenCV, IPP supports interleaved and planar data arrangement, with interleaved arrangement again being the predominant memory layout. The image data is being accessed through the base pointer and the vertical increment.

Note that due to the fact that IPP contains many functions that have been optimized for Intel's SSE units, it is necessary that a memory block for IPP to work on is aligned on a 32 byte boundary and that the start of each line of an image lies on a 4 byte boundary.

Lifetime Management

As IPP offers no structure to contain the image data, there is only a function available to allocate memory blocks that fit the requirements of IPP (`ippiMalloc)`, and a function to free these memory blocks (`ippiFree`).

[11] The complex data type is only used by a very limited set of functions.

3. Data Transfer from 3rd Party Libraries to CVB

The data/image transfer from 3rd party libraries to CVB is probably the easier direction – simply because thanks to the VPAT image model and the fairly versatile data type descriptor CVB covers all the memory layouts and data types encountered in libraries like OpenCV or IPP. It is also noteworthy that in practically all relevant cases the import can happen without copying the image data – all that is needed is a very thin VPAT wrapper (that can be built by the function `CreateImageFromPointer`) which takes only a fraction of a millisecond to construct. Probably the only potential obstacle is lifetime management.

As the function `CreateImageFromPointer` is pivotal for importing 3rd party images into CVB, let's have a detailed look at it:

```
IMPORT(cvbres_t) CreateImageFromPointer (void *pImageMem, size_t MemSize, cvbdim_t Width,
                    cvbdim_t Height, cvbdim_t NumPlanes, cvbdatatype_t DataType, intptr_t
                    PitchX, intptr_t PitchY, intptr_t PitchPlane, cvbval_t PlaneOrder[],
                    PFFINALRELEASE ReleaseCallback, void *pUserData, IMG &ImgOut);
```

The parameters are:

- `pImageMem` Pointer to the pixel data. When importing from OpenCV the pointer can be copied from `IplImage::imageData`.
- `MemSize`

  Gives the size of the memory block pointed to by `pImageMem`. This value will only be used for consistency checks and should be at least `Width` * `Height` * `NumPlanes` * `sizeof(`pixeldatatype`)`. When importing from OpenCV, the value `IplImage::imageSize` may be used here.
- `Width, Height`

  Specify the width and height of the image to be imported. When importing from OpenCV these can be copied from `IplImage::width` and `IplImage::height`.
- `NumPlanes`

  Specify the number of planes the CVB image should have. When importing from OpenCV, the value `IplImage::nChannels` may be used directly as it corresponds to the CVB plane count interpretation. With IPP, it is necessary to interpret the image type:

IPP image type	Number of Planes
`ipp`<data_type[12]>`C1`	1
`ipp`<data_type>`C2`	2
`ipp`<data_type>`C3`	3
`ipp`<data_type>`C4`	4
`ipp`<data_type>`AC4`	3
`ipp`<data_type>`P3`	3
`ipp`<data_type>`P4`	4

[12] One of the following: 8u, 8s, 16u, 16s, 32s, 32f, 32sc, 32fc. Note that in the unlikely case of dealing with 32sc or 32fc it might be a good idea to double the number of planes and treat e.g. a one-channel complex image as a 2 plane image with plane 0 being real and plane 1 being imaginary part.

CVB Interoperability

- `DataType`

  Build a CVB data type descriptor that best fits the pixel format of each channel. The available OpenCV and IPP data type descriptions map to CVB descriptors as follows:

OpenCV	IPP	CVB
`IPL_DEPTH_8U`	`ipp8u`	0x00000008
`IPL_DEPTH_8S`	`ipp8s`	0x00000108
`IPL_DEPTH_16U`	`ipp16u`	0x00000010
`IPL_DEPTH_16S`	`ipp16s`	0x00000110
`IPL_DEPTH_32S`	`ipp32s`	0x00000120
`IPL_DEPTH_32F`	`ipp32f`	0x00000320
`IPL_DEPTH_64F`		0x00000340

- `PitchX, PitchY, PitchPlane`

  These three pitch values will be needed by `CreateImageFromPointer` to build the VPAT wrapper around a 3rd party image. Although many users seem to struggle with these, their interpretation is fairly straightforward: If we know that `p` is the address of a pixel at the location `(x, y, i)` (with i being the plane index), then ○ `p + PitchX` gives the address of the pixel at the location `(x+1, y, i)` ○ `p + PitchY` gives the address of the pixel at the location `(x, y+1, i)` ○ `p + PitchPlane` gives the address of the pixel at the location `(x, y, i+1)`

  To determine the correct values for these pitches, it is necessary to look at the source image's pixel data type as well as the source image's memory layout! Luckily with OpenCV and IPP the number of possible permutations here is fairly limited:

  - OpenCV

    Although OpenCV does in principle inherit the possibility to describe planar memory layouts from the IPL, it usually uses interleaved memory layouts only. With interleaved layout the pitch values need to be constructed as follows:

```
PitchX = IplImage::nChannels * sizeof(pixel_type)
PitchY = IplImage::widthStep
PitchPlane = sizeof(pixel_type)
```

- IPP

  With IPP we need to take into account the possibility of planar memory layout (`P3` and `P4` variants of the image data types), as well as the special cases of the alpha channel (`AC4`) and the complex data types (`32sc` and `32fc`):

  - Interleaved Formats (`C1`, `C2`, `C3`, `C4`):

    `PitchX = NumPlanes`[13] `* sizeof(`pixel_type`)`

    `PitchY = stepBytes`[14]

    `PitchPlane = sizeof(`pixel_type`)`

  - Alpha channel formats (`AC4`)

    `PitchX = 4 * sizeof(`pixel_type`)`

    `PitchY = stepBytes`

    `PitchPlane = sizeof(`pixel_type`)`

  - Planar formats (`P3`, `P4`)

    Planar formats are only supported by `CreateImageFromPointer` if the distance in memory between the different planes is fixed[15].

    `PitchX = sizeof(`pixel_type`)`

    `PitchY = stepBytes`

    `PitchPlane = planeDistance`

  - Complex formats (`32sc`, `32fc`)

    To treat IPP's complex data formats properly one could come up with two valid approaches: One can either double the width and then simply access the image as if it were a regular image (keeping in mind that the even columns hold the real part and the odd columns hold the imaginary part of the pixel values), or one can construct an image with 2 planes and treat plane 0 as the real part and plane 1 as the imaginary part. The latter approach is only possible for complex formats with just one channel[16].

---

[13] `NumPlanes` refers to the value given to `CreateImageFromPointer` in the NumPlanes function parameter.

[14] `stepBytes` is the value returned by the `ippMalloc` function in the `pStepBytes` parameter (needs to memorized for later use) or generally speaking the y increment that is being used in IPP function calls.

[15] As there is currently no `ippMalloc` version for planar images it is the caller's responsibility to make sure that this condition is met and that the y increments and the plane increments are calculated correctly and take into account the necessary padding bytes.

[16] … at least without building multiple images with `CreateImageFromPointer` and then concatenating them with `CreateConcatenatedImage`. But this case is probably so exotic that we should spare the reader the full description of it.

The approach of doubling the width is fairly straightforward and the rules mentioned above the IPP apply (with `sizeof(`pixel_type`)` being 4 of course). Just remember to multiply the `Width` parameter with 2 in that case.

Building an image where the real part and the imaginary part are located in different planes requires – besides multiplying the `NumPlanes` parameter with 2 – a slight modification of the calculation of the increments:

```
PitchX = 8
PitchY = stepBytes
PitchPlane = 4
```

- `PlaneOrder`

  This parameter gives the caller the opportunity to easily reorder the numbering of the image planes. As mentioned before, CVB usually interprets plane 0 as the red channel, plane 1 as the green channel and plane 2 as the blue channel where color processing or display is involved. If that fits the data provided by the 3rd party library (typically the case with the IPP for example), then `PlaneOrder` can simply be set to `NULL`. But OpenCV by default loads color images in BGR format – in this case the PlaneOrder parameter should be set up like this to create a proper CVB image[17]:

  ```
  cvbval_t planeOrder[] = { 2, 1, 0 };
  ```

- `ReleaseCallback, pUserData`

  The callback and user data pointer provided here can help make sure that the 3rd party image's lifetime and the CVB image's lifetime are properly managed. As the CVB image constructed by `CreateImageFromPointer` shares the source image's memory, it is absolutely necessary that the source image exists at least as long as the CVB image constructed by `CreateImageFromPointer`. To make this easier, a callback pointer can be given in the `ReleaseCallback` parameter, and this callback will be invoked by CVB as soon as the reference counter of the image constructed by `CreateImageFromPointer` reaches zero.

  With this callback and the user data pointer it is even possible to transfer ownership (and thereby lifetime management) of the 3rd party image to CVB. For example it is possible to simply pass the `IplImage*` returned by e.g. `cvLoadImage` to the `pUserData` parameter and call `cvReleaseImage` on it in the release callback (see code example below).

  `ReleaseCallback` and/or `pUserData` may be set to `NULL` if the caller is not interested in knowing when the CVB image has been destroyed.

[17] This approach makes use of the VPAT to alter the perception of the memory layout. Pixel ordering in memory is still BGR in such a case, and those parts of CVB that absolutely require RGB ordering will change the byte order in memory. All others simply won't care.

CVB Interoperability

Of course it is also possible to decouple the CVB image completely from the source image it has been created from, but this requires copying the image data and is therefore a bit more time consuming. Probably the most convenient way to do it is to call `CreateImageFromPointer` first (as described above) and then using `CreateDuplicateImageEx`[18] to create a decoupled copy of the source image.

Code Example

As the OpenCV library provides a lot of convenience functionality that IPP does not offer, we'll only show a code sample for OpenCV here, but of course the principles apply to other 3rd party libraries as well. For the sake of brevity, error checking as well as using-, include- and linker statements have been omitted. Simply painting the image over the desktop is of course just a quick and dirty approach to displaying the image, but it reduces the amount of code necessary for displaying it to just one line.

```
void __stdcall final_release(void* pBufferBase, void* pUserData)
{
  IplImage* ocvImg = reinterpret_cast<IplImage*>(pUserData);
cvReleaseImage(&ocvImg);
}

int _tmain(int argc, _TCHAR* argv[])
{
  // load a sample image
string path = getenv("CVB");
  path.append("\\Tutorial\\FruitBowl.jpg");
  IplImage* ocvImg = cvLoadImage(path.c_str(),
CV_LOAD_IMAGE_UNCHANGED);
  // build CVB image from OpenCV image
IMG cvbFromOcv = nullptr;
  cvbval_t planeOrder[] = { 2, 1, 0 };
  CreateImageFromPointer(ocvImg->imageData, ocvImg->imageSize, ocvImg->width,
      ocvImg->height, ocvImg->nChannels, 8, ocvImg->nChannels * 1,  ocvImg-
      >widthStep, 1, planeOrder, final_release, ocvImg, cvbFromOcv);

  // display it (simply painting over the desktop)
  ImageToDC(cvbFromOcv, GetDC(NULL), 0, 0, 0, 0,
      ImageWidth(cvbFromOcv)-1, ImageHeight(cvbFromOcv)-1,
      ImageWidth(cvbFromOcv)-1, ImageHeight(cvbFromOcv)-1,
       0, 1, 2, 1.0, 0);
ReleaseObject(cvbFromOcv)
;   return 0; }
```

[18] Please Note that this function is not yet available in CVB 11.1 and older versions. In CVB 11.1 either use the significantly slower `CreateDuplicateImage` instead or copy the 3rd party library's image data by means of e.g. `memcpy` and build the `CreateImageFromPointer` wrapper around this copy of the image data (in this scenario it's recommended to free the data buffer in the release callback).

4. Data Transfer from CVB to 3rd Party Libraries

Transferring data from CVB images to 3rd party libraries is a lot more difficult than vice versa – especially if the transfer is supposed to be done without copying image data. Ironically it is the VPAT image model that made transferring 3rd party images into CVB so easy in the previous chapter. This makes the other direction potentially more complicated because a certain amount of work needs to be invested to make sure that the memory layout of the CVB image matches the requirements of the destination library.

Transfer Without Copy

For performance reasons, it is usually more desirable to interface CVB to another library without copying the image data. However, this is not always possible.

First of all the 3rd party library that is the target for the image data needs to support this approach, i.e. there needs to be some means of creating an image for that library around an already existing memory block (much

like the `CreateImageFromPointer` function in CVB or the combination of `cvCreateImageHeader` and `cvSetData` in OpenCV). This is not always a given – a library that concerns itself exclusively or mostly with the acquisition of images is likely to require the buffers it works on to have been allocated by its own memory management. In short: If you want to transfer image data without copying them, the first step should be to make sure that the destination library actually supports this.

Then there is the matter of the CVB image model. Because the VPAT interface of CVB images supports far more memory layouts than a library relying on e.g. a base pointer and a fixed y increment, it is necessary to analyse the memory layout of a CVB image to find out if a data transfer without copying the image data is possible. The function for this is `GetLinearAccess` (exported by the CVCUtilities.dll). It analyses the VPAT on a given image's plane and returns the x and y increments as well as the base pointer if the VPAT is organized linearly (i.e. if it uses fixed increments in x and y direction), otherwise the returned increments will be 0. The subsequent checks on the increments and the base pointer that are necessary of course depend on the 3rd party library's requirements, so it is not possible to give a complete and generally applicable account here but for our sample cases (OpenCV and IPP) it is possible to give a recipe.

OpenCV

First of all, it is recommendable to determine whether or not the image format is in principle compatible with OpenCV. For this to be the case…

- … all image planes will need to be of the same data type (usually the case)
- … the data type must be one of those supported by OpenCV (see chapter 2)
- … the number of planes must not exceed 4.

CVB Interoperability

If these conditions are met, GetLinearAccess will need to be called on each plane of the image, and the results (base pointers and increments) will need to be harvested for the following comparisons[19]:

- The x and y increments for all planes need to be identical
- The x increment must be equal to the number of bytes per pixel times the number of planes
- The y increment must be greater or equal to the x increment times the image width[20]
- For images with more than one plane the absolute difference between two consecutive base pointers must be equal to the number of bytes per pixel
- For images with three planes to be properly interpreted as BGR by OpenCV it is necessary that the three base pointers are in descending order (i.e. base pointer for plane 0 points to a higher address than base pointer for plane 1 and base pointer for plane 1 points to a higher address than base pointer for plane 2)[24].

If all the required conditions are met, attaching an OpenCV image to the CVB image data is fairly straightforward: First an empty `IplImage` structure needs to be created as the recipient by calling `cvCreateImageHeader`, then the data members need to be set using `cvSetData` (the input for `cvSetData` can be taken directly from `GetLinearAccess`).

The only remaining issue to be sorted out now is the object lifetime. The situation is a bit dangerous in that both, the source CVB image and the wrapping OpenCV image rely on the same image data to be available. CVB has no means of yielding ownership of its image memory to another entity, so calling e.g. `cvReleaseImage` or `cvReleaseData` will result in an access violation as soon as the CVB image is being accessed after that (even if it is just through `ReleaseObject`). Vice versa, calling `ReleaseObject` on the CVB image and accessing the OpenCV image afterward will also result in an access violation. The only safe way to get rid of both, the CVB image and the OpenCV image (without producing a memory leak) is to only call `cvReleaseImageHeader` on the OpenCV side and (as usual) `ReleaseObject` on the CVB side[21].

---

[19] In principle these conditions apply the logic behind the calculation of the increments for `CreateImageFromPointer` shown in chapter 3 to the CVB image.

[20] If the y increment is bigger than x increment times image width there simply is a number of padding bytes added in memory to each line which can be safely ignored. [24] This is an implication of the BGR ordering of the color planes that OpenCV expects. Note that the `IplImage` structure does in fact have a `channelSeq` member, but according to the documentation this member is ignored by OpenCV.

Of course, application of this criterion is optional: For those OpenCV functions that do not process color information the channel order is largely irrelevant.

[21] Of course the `ReleaseObject` call can happen a lot later than `cvReleaseImageHeader` – the latter will in fact destroy the the `IplImage` structure so that accidental continued use of the image is no longer possible.

CVB Interoperability

Expressing all this in C/C++ could look e.g. like this:

```
// ----------------------------------------------------------------------
/// Translate a cvb data type descriptor to an OpenCV data type.
// ----------------------------------------------------------------------
int ocv_depth(cvbdatatype_t cvbDt)
{
  // map compatible cvb data type descriptors to OpenCV data types
switch (cvbDt & 0xFF)
  {
case 8:
    return IsSignedDatatype(cvbDt) ? IPL_DEPTH_8S : IPL_DEPTH_8U;
case 16:
    return IsSignedDatatype(cvbDt) ? IPL_DEPTH_16S : IPL_DEPTH_16U;
case 32:
    return IsFloatDatatype(cvbDt) ? IPL_DEPTH_32F : IPL_DEPTH_32S;
case 64:
    return IsFloatDatatype(cvbDt) ? IPL_DEPTH_64F : 0;
default:     return 0;
  }
}

// ----------------------------------------------------------------------
/// Check if an image in principle is compatible with CVB.
// ----------------------------------------------------------------------
bool is_opencv_compatible(IMG cvbImg)
{   using namespace std;
if (cvbImg == nullptr)
return false;
  vector<cvbdatatype_t> dataTypes(ImageDimension(cvbImg));
for (cvbdim_t i = 0; i < ImageDimension(cvbImg); ++i)
dataTypes[i] = ImageDatatype(cvbImg, i);
  // all planes have identical data type?
for (size_t i = 1; i < dataTypes.size();
++i)
if (dataTypes[i] != dataTypes[0])
return false;
  // data type compatible with OpenCV?
if (ocv_depth(dataTypes[0]) == 0)
return false;
  // no more than 4 channels?   if ((ImageDimension(cvbImg) < 1) ||
(ImageDimension(cvbImg) > 4))     return false;
// all checks passed
return true;
}
```

CVB Interoperability

```
// ----------------------------------------------------------------------
/// Analyze the memory layout of a CVB image and see if it can be transferred
/// to opencv without copying the image data.
/// \param [in] cvbImg The image to be analyzed.
/// \param [in] bgrStrict When set to true, the function will return false if
///   the input image has 3 planes and the three planes are not arranged in
///   the sequence 2, 1, 0. If the image does not have 3 planes, this
///   parameter is ignored.
// ----------------------------------------------------------------------
bool is_opencv_shareable(IMG cvbImg, bool bgrStrict)
{
  using namespace std;
if(!is_opencv_compatible(cvbImg))
return false;
vector<intptr_t> basePtr(ImageDimension(cvbImg));
vector<intptr_t> xInc(ImageDimension(cvbImg));
vector<intptr_t> yInc(ImageDimension(cvbImg));
for (cvbdim_t i = 0; i < ImageDimension(cvbImg);
++i)
    GetLinearAccess(cvbImg, i, reinterpret_cast<void**>(&basePtr[i]), &xInc[i], &yInc[i]);
  // do all planes have same increment in x and y?
for (size_t i = 1; i < xInc.size(); ++i)
  {     if (xInc[i] != xInc[0])      return false;
        if (yInc[i] != yInc[0])      return false;
  }
  // do the increments hint at an interleaved memory layout?
size_t bytesPp = BytesPerPixel(ImageDatatype(cvbImg, 0));
if (xInc[0] != (bytesPp * ImageDimension(cvbImg)))     return false;
  // is the y increment ok?
if (yInc[0] < xInc[0] * ImageWidth(cvbImg))     return false;
  // is the plane pitch ok?
  if (ImageDimension(cvbImg) > 1)
  {
    for (size_t i = 0; i < basePtr.size() - 1; ++i)
    {
      ptrdiff_t iInc = abs(basePtr[i+1] - basePtr[i]);
      if (iInc != bytesPp)         return false;
    }
  }
  // if requested, make sure we have bgr arrangement
if ((bgrStrict) && (ImageDimension(cvbImg) == 3))
  {
    if (basePtr[0] <= basePtr[1])       return false;
    if (basePtr[1] <= basePtr[2])       return false;
  }
  // all test passed successfully
return true;
}
```

CVB Interoperability

```
 // ----------------------------------------------------------------------
/// If possible, create an OpenCV image from a CVB image without copying the
/// image data.
/// \param [in] cvbImg The image to be transferred.
/// \param [in] bgrStrict When set to true, the function will fail if
///   the input image has 3 planes and the three planes are not arranged in
///   the sequence 2, 1, 0. If the image does not have 3 planes, this
///   parameter is ignored.
// ----------------------------------------------------------------------
IplImage* cvb_to_ocv_nocopy(IMG cvbImg, bool bgrStrict)
{
  if (!is_opencv_shareable(cvbImg, bgrStrict))     return nullptr;
  // construct an appropriate OpenCV image
CvSize size;
size.width = ImageWidth(cvbImg);
size.height = ImageHeight(cvbImg);
IplImage* retval = cvCreateImageHeader(size, ocv_depth(ImageDatatype(cvbImg, 0)),
                                       ImageDimension(cvbImg));
  if (retval == nullptr)     return retval;
  // as we have made sure that the memory layout is compatible, it is
  // sufficient here to access plane 0 only
void* ppixels = nullptr;
intptr_t xInc = 0;
intptr_t yInc = 0;
  GetLinearAccess(cvbImg, 0, &ppixels, &xInc, &yInc);   cvSetData(retval, ppixels, yInc);
return retval; }
```

Use of these functions is fairly straightforward:

```
  // load a sample image
  string path = getenv("CVB");
  path.append("Tutorial\\FruitBowl.jpg");
  IMG cvbImg = nullptr;
  LoadImageFile(path.c_str(), cvbImg);

  // create an attached OpenCV image
  IplImage* ocvImg = cvb_to_ocv_nocopy(cvbImg, false);

  …

  // cleanup
  cvReleaseImageHeader(&ocvImg);
  ReleaseObject(cvbImg);
```

CVB Interoperability

IPP

In principle the same approach taken for OpenCV can be applied to IPP as well, therefore we will not re-iterate them in source code here again. First verify whether the image is in principle usable with IPP (the analysis is more less the same as for OpenCV, however IPP does not support 64 bit floating point images), analyse the memory layout (keeping in mind that some functions of IPP actually also support a planar layout) and use the information provided by `GetLinearAccess` to let IPP work on the image.

BGR arrangement is not an issue here, as most functions in IPP actually expect RGB layout, and lifetime management also is a non-issue as IPP uses no image structure of its own.

There is, however, one additional pitfall that did not apply to OpenCV: The IPP tries to use SSE optimized algorithms wherever possible. These do not only require that each line starts on a 4 byte boundary, they also require that the image starts on a 32 bit boundary. Both need to be verified using the data returned by `GetLinearAccess`, when calling an IPP function. If these conditions are not met, those IPP functions that expected them will return an error code[22].

Lifetime is also a non-issue with IPP. As there is – at least in the C API – no data structure for images (IPP usually works only on pointers and increments that the caller needs to keep inside his own data structures) there is nothing to keep in mind other than that the pointers and increments returned by `GetLinearAccess` should not be used any more once the image has been released with `ReleaseObject`.

Transfer With Copy

Although copying the image data from a CVB image to any 3rd party data structure will take significantly longer than the approach explained in the previous chapter (even on small images), there are typically two scenarios where that amount of time needs to be invested:

1. If the 3rd party library does not accept memory buffers that weren't allocated by it.
2. If the memory layout of the CVB image does not meet the requirements of the 3rd party library[23],[24].

These two cases can differ significantly in terms of implementation and time required for copying the data. If the data in the CVB image is already arranged in a format that is usable by the target library, copying the data line by line will be very efficient. Otherwise it may be necessary to copy each pixel individually through VPAT access, which obviously is more time-consuming.

---

[22] Two situations where this image data layout is usually given are CVB VIN drivers (as long as RotateImage has not been set) and BMP files.

[23] Keep in mind that this can often be prevented by fairly simple measures. For example make sure that the `RotateImage` parameter in the driver configuration file is (if available) set to 0 and avoid functions like GetLinearAccess, `CreatePanoramicImageMap` and `CreateRotatedImageMap`, as these functions will (most likely) generate VPATs that are not linearly accessible.

[24] Doing `memcpy` line by line as opposed to doing the whole image in one block is not much slower, but saves a great deal of checks that would be necessary to properly treat padding bytes and, at the same time, automatically takes care of potential discrepancies between top-down or bottom-up arrangement.

CVB Interoperability

OpenCV

To determine whether or not it becomes necessary to copy the image data through the VPAT, we can revert to the work that has been done in the previous chapter: If it would in principle be possible to transfer the data without copying, then the layout obviously is already ok and VPAT copy will not be necessary. Copying in this case is reduced to a loop over the image's lines .

For VPAT copy, there obviously needs to be two loops (one in vertical and one in horizontal direction). A bit of complexity is added by the fact that the amount of data to be copied for each pixel depends on the image's data type, but templates go a long way to implementing copy loops that automatically apply to all supported data types. They can also greatly help reduce overhead from looping through the image planes[25].

In practice the code will look something like this:

```
// ----------------------------------------------------------------------
/// Copy functions that copies a CVB image line by line into an OpenCV image.
// ----------------------------------------------------------------------
IplImage* cvb_to_ocv_copy_lines(IMG cvbImg)
{
  // consistency checks on input
  if (!is_opencv_compatible(cvbImg))     return nullptr;
  if (!(is_opencv_shareable(cvbImg, true) ||
       (is_opencv_shareable(cvbImg, false))))     return nullptr;
  // get data access pointers
  intptr_t pSrc, xInc, yInc;   pSrc = xInc = yInc = 0;
  GetLinearAccess(cvbImg, 0, reinterpret_cast<void**>(&pSrc), &xInc, &yInc);
  if ((xInc == 0) || (yInc == 0))     return nullptr;
  // create destination image
  IplImage* retval = cvCreateImage(cvSize(ImageWidth(cvbImg),
    ImageHeight(cvbImg)),  ocv_depth(ImageDatatype(cvbImg, 0)),
    ImageDimension(cvbImg));
  if (retval == nullptr)     return nullptr;
  // copy loop
  intptr_t pDst = reinterpret_cast<intptr_t>(retval->imageData);
  for (int y = 0; y < retval->height; ++y)
  {
    memcpy(reinterpret_cast<void*>(pDst), reinterpret_cast<void*>(pSrc), xInc * retval->width);
    pSrc += yInc;
    pDst += retval->widthStep;
  }
  return retval;
}
```

```
// ----------------------------------------------------------------------
/// The most versatile, but also most time-consuming part: copying the image
```

[25] Note, however, that this benefit can only be reaped in the release build.

CVB Interoperability

```
/// data through the VPAT. For speed reasons this has been implemented as a
/// template - the compiler's optimizer will discard all unnecessary code
/// paths and comparisons for #planes < 4 in the release build.
// ----------------------------------------------------------------------
template<typename t_pixel, int numPlanes> IplImage* cvb_to_ocv_copy_vpat(IMG cvbImg)
{ using namespace std;
  // consistency checks
  if (!is_opencv_compatible(cvbImg))      return nullptr;
  if (ImageDimension(cvbImg) < numPlanes) return nullptr;
  // get the VPAT pointers
  vector<intptr_t> pBase(numPlanes);
  vector<PVPAT>    pVpat(numPlanes);
  vector<intptr_t> pLine(numPlanes);
  for (int i = 0; i < numPlanes; ++i)
    if (!GetImageVPA(cvbImg, i, reinterpret_cast<void**>(&pBase[i]), &pVpat[i]))
       return nullptr;
  // create the destination image
  IplImage* retval = cvCreateImage(cvSize(ImageWidth(cvbImg), ImageHeight(cvbImg)),
    ocv_depth(ImageDatatype(cvbImg, 0)), ImageDimension(cvbImg));
  if (retval == nullptr)     return nullptr;
  // copy loops
  intptr_t pDstLine = reinterpret_cast<intptr_t>(retval->imageData);
  for (int y = 0; y < retval->height; ++y)
  { t_pixel* pDst = reinterpret_cast<t_pixel*>(pDstLine);
    for (int i = 0; i < numPlanes; ++i)
         pLine[i] = pBase[i] + pVpat[i][y].YEntry;
         for (int x = 0; x < retval->width; ++x)
          {
           if (numPlanes == 3)
            {
             // in case we have 4 planes, we switch R and B
             *pDst++ = *reinterpret_cast<t_pixel*>(pLine[2] + pVpat[2][x].XEntry);
             *pDst++ = *reinterpret_cast<t_pixel*>(pLine[1] + pVpat[1][x].XEntry);
            *pDst++ = *reinterpret_cast<t_pixel*>(pLine[0] + pVpat[0][x].XEntry);
            }
    else
      {
        *pDst++ = *reinterpret_cast<t_pixel*>(pLine[0] + pVpat[0][x].XEntry);
        if (numPlanes > 1)
          *pDst++ = *reinterpret_cast<t_pixel*>(pLine[1] + pVpat[1][x].XEntry);
        if (numPlanes > 2)
          *pDst++ = *reinterpret_cast<t_pixel*>(pLine[2] + pVpat[2][x].XEntry);
        if (numPlanes > 3)
          *pDst++ = *reinterpret_cast<t_pixel*>(pLine[3] + pVpat[3][x].XEntry);
      }
  }
    pDstLine += retval->widthStep;
  }
  return retval;
}
```

CVB Interoperability

```
// ----------------------------------------------------------------------------
/// Just a utility function to avoid a huge code cascade when interfacing the
/// template above to the code below.
// ----------------------------------------------------------------------------
template<typename t_pixel>
IplImage* cvb_to_ocv_copy_vpat_dispatcher(IMG cvbImg)
{
  int dim = ImageDimension(cvbImg);
  switch(dim)
  { case 1: return cvb_to_ocv_copy_vpat<t_pixel, 1>(cvbImg);
    case 2: return cvb_to_ocv_copy_vpat<t_pixel, 2>(cvbImg);
    case 3: return cvb_to_ocv_copy_vpat<t_pixel, 3>(cvbImg);
    case 4: return cvb_to_ocv_copy_vpat<t_pixel, 4>(cvbImg);
    default:  return nullptr;
  }
}

// ----------------------------------------------------------------------------
/// If possible, create an OpenCV image from a CVB image, copying the
/// image data.
/// \param [in] cvbImg The image to be copied.
/// \param [in] bgrStrict When set to true, and image with RGB layout will
///   automatically converted to BGR layout; if the image does not have 3
///   planes, this parameter is ignored.
// ----------------------------------------------------------------------------
IplImage* cvb_to_ocv_copy(IMG cvbImg, bool bgrStrict)
{
  // make sure the image is not fundamentally incompatible
  if (!is_opencv_compatible(cvbImg))     return nullptr;
  // find out if a line-by-line copy is possible
  if (is_opencv_shareable(cvbImg, bgrStrict))     return cvb_to_ocv_copy_lines(cvbImg);
  // VPAT copy is necessary to port the image to OpenCV
  switch(ocv_depth(ImageDatatype(cvbImg, 0)))
  {   case IPL_DEPTH_8U:
      case IPL_DEPTH_8S: return cvb_to_ocv_copy_vpat_dispatcher<unsigned char>(cvbImg);
      case IPL_DEPTH_16U:
      case IPL_DEPTH_16S: return cvb_to_ocv_copy_vpat_dispatcher<unsigned short>(cvbImg);
      case IPL_DEPTH_32S: return cvb_to_ocv_copy_vpat_dispatcher<int>(cvbImg);
      case IPL_DEPTH_32F: return cvb_to_ocv_copy_vpat_dispatcher<float>(cvbImg);
      case IPL_DEPTH_64F: return cvb_to_ocv_copy_vpat_dispatcher<double>(cvbImg);
      default:     return nullptr;
  }
}
```

With the OpenCV image created through `cvCreateImage` independent of the CVB image, it does not matter in what order they are destroyed. And unlike in the previous chapter, destruction of the OpenCV image must now happen through `cvReleaseImage`, otherwise a memory leak will occur.

CVB Interoperability

IPP

Like in the previous chapter, the copy of CVB image data to IPP buffers is again pretty much covered by what has been said and coded for OpenCV. Again the major differences are that the IPP does not support doublevalued images and is working well with RGB arrangement of color images (therefore the special treatment in the template `cvb_to_ocv_copy_vpat` for 3-plane images can just be omitted). Due to the special requirements (image start on a 32 byte boundary, line start on a 4 byte boundary), import of CVB images into IPP is probably a little more likely to require copying image data.

5. Other Programming Languages

C/C++ has been used as the programming language for this little application note – mostly because when looking at any 3rd party imaging library, C++ is the most likely language to be supported. It is also because the pointer support in C/C++ is very straightforward and versatile. Of course this does not mean that interopation with 3rd party libraries is limited to C/C++.

The methods explained in this application note should be easily transferable to any language that supports the notion of pointers such as C# or Delphi. With C# of course there's the added complexity of managed versus unmanaged domain, but even that is not really a problem – it just requires proper use of the `fixed` and the `unsafe` keyword. Keep in mind however, that it is beneficial to keep the managed-unmanaged transitions to the necessary minimum as these transitions tend to consume a fair amount of CPU time.

Visual Basic.Net programmers can in theory also implement anything that has been shown in this application note. However, because the language does not support pointers directly, doing so requires heavy use of the `System.Runtime.InteropServices.Marshal` class's methods and is certain to result in code that is much harder to implement and read, and runs a fair bit slower than the corresponding C# implementation. Therefore our recommendation to VB.Net programmers is to implement their interoperation with 3rd party libraries in C# into a DLL that can then be used from within a VB.Net application.

Probably the only language that is a complete and utter dead end in terms of this application note is Visual Basic 6 – this language has no inherent support for pointers, nor does it come with functions or constructs that let the programmer do any memory management worth mentioning, therefore Visual Basic 6 programmers do not have the option of mixing CVB and other libraries in their application with the methods described here.

6. Appendix/ Contact and Feedback

Version History

Rev. & Date	Who	Comments
1.0.0 (14.02.2013)	VGi	Initial draft.
1.0.1 (14.02.2013)	ABa	Proofread.
1.1.0 (19.02.2013)	MKe	Corrections.
1.2.0 (29.07.2019)	HGa	New design

We hope that these notes were useful for you and look forward to your feedback. In case of further questions, please do not hesitate to contact our technical support.

Additional information as well as frequently asked questions and a lot of valuable details regarding image processing, can also be found on our website.

Phone: +49 89 80902-200
E-Mail: support@stemmer-imaging.de

Web: http://www.commonvisionblox.com

forum.commonvisionblox.com

http://www.stemmer-imaging.de(menu *Service*)

Yours sincerely - STEMMER IMAGING Technical Team