OpenVX (Open Computer Vision Acceleration API) is a low-level programming framework domain to access computer vision hardware acceleration with both functional and performance portability. OpenVX supports modern hardware architectures, such as mobile and embedded SoCs as well as desktop systems.

[n.n] refers to sections in the OpenVX 1.1 specification available at khronos.org/registry/vx/
Parameter color coding: ***Output***, ***Input***, and ***Input/Output*** parameters.
Parameters marked as [Opt.] are optional. If omitted, NULL.
In descriptions, curly braces {} mean "one of," and square braces [] mean "optional."
- Indicates the overflow policy used is VX_CONVERT_POLICY_SATURATE.
- Refer to list of enumerations on page 7 of this reference guide.

OpenVX Functions and Objects

VX_API_CALL

In every OpenVX function and object, insert VX_API_CALL before the function name, as shown in the example below. It was omitted from the functions on this reference card to save space.

<return_type> VX_API_CALL **vxFunctionName**(T *arg1* , . . . , T *argN*);

Vision Functions

Vision functions in OpenVX may be graph mode or immediate mode.

- **Graph mode functions** have "Node" in the function name. They may be created and linked together, verified by the implementation, then executed as often as needed.
- **Immediate mode functions** are executed on a context immediately, as if they were single node graphs, with no leaking side-effects.

In the vision functions, the parameter ***graph*** is the reference to the graph, and the parameter ***context*** is the reference to the overall context.

Vision Functions

Absolute Difference [3.2]

$out(x, y) = |in_1(x, y) - in_2(x, y)|$

vx_node **vxAbsDiffNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuAbsDiff** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1, in2, out: Image of VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Accumulate [3.3]

- $accum(x, y) = accum(x, y) + input(x, y)$

vx_node **vxAccumulateImageNode** (vx_graph *graph*, vx_image *input*, vx_image *accum*);

vx_status **vxuAccumulateImage** (vx_context *context*, vx_image *input*, vx_image *accum*);

input: The input image of VX_DF_IMAGE_U8 format.
accum: The accumulation image of VX_DF_IMAGE_S16 format.

Accumulate Squared [3.4]

- Accumulates a squared input image to an output image.

vx_node **vxAccumulateSquareImageNode** (vx_graph *graph*, vx_image *input*, vx_scalar *shift*, vx_image *accum*);

vx_status **vxuAccumulateSquareImage** (vx_context *context*, vx_image *input*, vx_scalar *shift*, vx_image *accum*);

input: The input image of VX_DF_IMAGE_U8 format.
shift: Shift amount of type VX_TYPE_UINT32 ($0 \le shift \le 15$).
accum: The accumulation image of VX_DF_IMAGE_S16 format.

Accumulate Weighted [3.5]

$accum(x, y) = (1 - \alpha) * accum(x, y) + \alpha * input(x, y)$

vx_node **vxAccumulateWeightedImageNode** (vx_graph *graph*, vx_image *input*, vx_scalar *alpha*, vx_image *accum*);

vx_status **vxuAccumulateWeightedImage** (vx_context *context*, vx_image *input*, vx_scalar *alpha*, vx_image *accum*);

input: The input image of VX_DF_IMAGE_U8 format.
alpha: The scale of type VX_TYPE_FLOAT32 ($0.0 \le \alpha \le 1.0$).
accum: The accumulation image of VX_DF_IMAGE_U8 format.

Arithmetic Addition [3.6]

$out(x, y) = in_1(x, y) + in_2(x, y)$

vx_node **vxAddNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*);

vx_status **vxuAdd** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*);

in1, in2, out: Image of VX_DF_IMAGE_{U8, S16} format.
policy: VX_CONVERT_POLICY_{WRAP, SATURATE}.

Arithmetic Subtraction [3.7]

$out(x, y) = in_1(x, y) - in_2(x, y)$

vx_node **vxSubtractNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*);

vx_status **vxuSubtract** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*);

in1, in2, out: Image of VX_DF_IMAGE_{U8, S16} format.
policy: VX_CONVERT_POLICY_{WRAP, SATURATE}.

Bitwise AND [3.8]

$out(x, y) = in_1(x, y) \wedge in_2(x, y)$

vx_node **vxAndNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuAnd** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1, in2: The input image of VX_DF_IMAGE_U8 format.
out: The output image of VX_DF_IMAGE_U8 format.

Bitwise EXCLUSIVE OR [3.9]

$out(x, y) = in_1(x, y) \oplus in_2(x, y)$

vx_node **vxXorNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuXor** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1, in2: The input image of VX_DF_IMAGE_U8 format.
out: The output image of VX_DF_IMAGE_U8 format.

Bitwise INCLUSIVE OR [3.10]

$out(x, y) = in_1(x, y) \vee in_2(x, y)$

vx_node **vxOrNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*);

vx_status **vxuOr** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*);

in1, in2: The input image of VX_DF_IMAGE_U8 format.
out: The output image of VX_DF_IMAGE_U8 format.

Bitwise NOT [3.11]

$out(x, y) = \overline{in}(x, y)$

vx_node **vxNotNode** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuNot** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Box Filter [3.12]

Computes a Box filter over a window of the input image.

vx_node **vxBox3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuBox3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Canny Edge Detector [3.13]

Provides a Canny edge detector kernel.

vx_node **vxCannyEdgeDetectorNode** (vx_graph *graph*, vx_image *input*, vx_threshold *hyst*, vx_int32 *gradient_size*, vx_enum *norm_type*, vx_image *output*);

vx_status **vxuCannyEdgeDetector** (vx_context *context*, vx_image *input*, vx_threshold *hyst*, vx_int32 *gradient_size*, vx_enum *norm_type*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.
hyst: The double threshold for hysteresis. VX_TYPE_{U8, U16}.
gradient_size: Size of Sobel filter window; supports at least 3, 5, 7.
norm_type: Norm used to compute the gradient. VX_NORM_{L1, L2}.

Channel Combine [3.14]

Combines multiple image planes.

vx_node **vxChannelCombineNode** (vx_graph *graph*, vx_image *plane0*, vx_image *plane1*, vx_image *plane2*, vx_image *plane3*, vx_image *output*);

vx_status **vxuChannelCombine** (vx_context *context*, vx_image *plane0*, vx_image *plane1*, vx_image *plane2*, vx_image *plane3*, vx_image *output*);

plane{0, 1}: Plane forming channel {0, 1}. VX_DF_IMAGE_U8.
plane{2, 3}: [Opt.] Plane that forms channel {2, 3}. VX_DF_IMAGE_U8.
output: The output image.

Channel Extract [3.15]

Extracts a plane from a multi-planar or interleaved image.

vx_node **vxChannelExtractNode** (vx_graph *graph*, vx_image *input*, vx_enum *channel*, vx_image *output*);

vx_status **vxuChannelExtract** (vx_context *context*, vx_image *input*, vx_enum *channel*, vx_image *output*);

input: One of the defined vx_df_image_e multi-channel formats.
channel: Channel to extract. VX_CHANNEL_{0, 1, 2, 3, R, G, B, A, Y, U, V}.
output: The output image of VX_DF_IMAGE_U8 format.

Color Convert [3.16]

Converts the format of an image.

vx_node **vxColorConvertNode** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuColorConvert** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: The image {to, from} which to convert.

Convert Bit Depth [3.17]

- Converts image bit depth.

vx_node **vxConvertDepthNode** (vx_graph *graph*, vx_image *input*, vx_image *output*, vx_enum *policy*, vx_scalar *shift*);

vx_status **vxuConvertDepth** (vx_context *context*, vx_image *input*, vx_image *output*, vx_enum *policy*, vx_int32 *shift*);

input, output: The {input, output} image.
policy: VX_CONVERT_POLICY_{WRAP, SATURATE}.
shift: The shift value of type VX_TYPE_INT32.

Custom Convolution [3.18]

- Convolves an image with a user-defined matrix.

vx_node **vxConvolveNode** (vx_graph *graph*, vx_image *input*, vx_convolution *conv*, vx_image *output*);

vx_status **vxuConvolve** (vx_context *context*, vx_image *input*, vx_convolution *conv*, vx_image *output*);

input: An input image of VX_DF_IMAGE_U8 format.
conv: The vx_int16 convolution matrix.
output: The output image of VX_DF_IMAGE_{S16, U8} format.

Dilate Image [3.19]

Grows the white space in a VX_DF_IMAGE_U8 Boolean image.

vx_node **vxDilate3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuDilate3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Equalize Histogram [3.20]

Normalizes brightness and contrast of a grayscale image.

vx_node **vxEqualizeHistNode** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuEqualizeHist** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: The grayscale image of VX_DF_IMAGE_U8 format.

Erode Image [3.21]

Shrinks the white space in a VX_DF_IMAGE_U8 Boolean image.

vx_node **vxErode3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuErode3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Fast Corners [3.22]

Finds corners in an image.

vx_node **vxFastCornersNode** (vx_graph *graph*, vx_image *input*, vx_scalar *strength_thresh*, vx_bool *nonmax_suppression*, vx_array *corners*, vx_scalar *num_corners*);

vx_status **vxuFastCorners** (vx_context *context*, vx_image *input*, vx_scalar *strength_thresh*, vx_bool *nonmax_suppression*, vx_array *corners*, vx_scalar *num_corners*);

input: The input image of VX_DF_IMAGE_U8 format.
strength_thresh: (VX_TYPE_FLOAT32) Intensity difference threshold.
nonmax_suppression: Boolean specifying if non-maximum suppression is applied to detected corners before being placed in the vx_array.
corners: Output corner vx_array of type VX_TYPE_KEYPOINT.
num_corners: [Opt.] Number of detected corners in image. (VX_TYPE_SIZE)

(Continued on next page) ▶

◀ Vision Functions (cont.)

Gaussian Filter [3.23]

Computes a Gaussian filter over a window of the input image.

vx_node **vxGaussian3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuGaussian3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Non-linear Filter [3.24]

Computes a non-linear filter over a window of the input image.

vx_node **vxNonLinearFilterNode** (vx_graph *graph*, vx_enum *function*, vx_image *input*, vx_matrix *mask*, vx_image *output*);

vx_status **vxuNonLinearFilter** (vx_context *context*, vx_enum *function*, vx_image *input*, vx_matrix *mask*, vx_image *output*);

function: The non-linear function of type VX_NONLINEAR_FILTER_{MEDIAN, MIN, MAX}.
input: An input image of VX_DF_IMAGE_U8 format.
mask: The mask to apply (See **vxCreateMatrixFromPattern**).
output: The output image of VX_DF_IMAGE_U8 format.

Harris Corners [3.25]

Computes the Harris Corners of an image.

vx_node **vxHarrisCornersNode** (vx_graph *graph*, vx_image *input*, vx_scalar *strength_thresh*, vx_scalar *min_distance*, vx_scalar *sensitivity*, vx_int32 *gradient_size*, vx_int32 *block_size*, vx_array *corners*, vx_scalar *num_corners*);

vx_status **vxuHarrisCorners** (vx_context *context*, vx_image *input*, vx_scalar *strength_thresh*, vx_scalar *min_distance*, vx_scalar *sensitivity*, vx_int32 *gradient_size*, vx_int32 *block_size*, vx_array *corners*, vx_scalar *num_corners*);

input: An input image of VX_DF_IMAGE_U8 format.
strength_thresh: The minimum threshold of type VX_TYPE_FLOAT32 with which to eliminate Harris corner scores.
min_distance: The radial Euclidean distance of type VX_TYPE_FLOAT32 for non-maximum suppression.
sensitivity: The scalar sensitivity threshold *k* of type VX_TYPE_FLOAT32.
gradient_size: The gradient window size to use on the input.
block_size: Block window size used to compute the Harris corner score.
corners: The array of objects of type VX_TYPE_KEYPOINT.
num_corners: [Opt.] Num. of detected corners in image. (VX_TYPE_SIZE)

Histogram [3.26]

Generates a distribution from an image.

vx_node **vxHistogramNode** (vx_graph *graph*, vx_image *input*, vx_distribution *distribution*);

vx_status **vxuHistogram** (vx_context *context*, vx_image *input*, vx_distribution *distribution*);

input: The input image of VX_DF_IMAGE_U8 format.
distribution: The output distribution.

Gaussian Image Pyramid [3.27]

Computes a Gaussian image pyramid using a 5x5 kernel.

vx_node **vxGaussianPyramidNode** (vx_graph *graph*, vx_image *input*, vx_pyramid *gaussian*);

vx_status **vxuGaussianPyramid** (vx_context *context*, vx_image *input*, vx_pyramid *gaussian*);

input: The input image of VX_DF_IMAGE_U8 format.
gaussian: The Gaussian pyramid of VX_DF_IMAGE_U8 format.

Laplacian Image Pyramid [3.28]

Computes a Laplacian Image Pyramid from an input image.

vx_node **vxLaplacianPyramidNode** (vx_graph *graph*, vx_image *input*, vx_pyramid *laplacian*, vx_image *output*);

vx_status **vxuLaplacianPyramid** (vx_context *context*, vx_image *input*, vx_pyramid *laplacian*, vx_image *output*);

input: The input image of VX_DF_IMAGE_U8 format.
laplacian: The Laplacian pyramid of VX_DF_IMAGE_S16 format.
output: The lowest-resolution image of VX_DF_IMAGE_S16 necessary to reconstruct the input image from the pyramid.

Reconstruction from Laplacian Image Pyramid [3.29]

Reconstructs the original image from a Laplacian Image Pyramid.

vx_node **vxLaplacianReconstructNode** (vx_graph *graph*, vx_pyramid *laplacian*, vx_image *input*, vx_image *output*);

vx_status **vxLaplacianReconstruct** (vx_context *context*, vx_pyramid *laplacian*, vx_image *input*, vx_image *output*);

laplacian: The Laplacian pyramid of VX_DF_IMAGE_S16 format.
input: The lowest-resolution image of VX_DF_IMAGE_S16 format.
output: The image of VX_DF_IMAGE_U8 with the highest possible resolution reconstructed from the pyramid.

Integral Image [3.30]

Computes the integral image of the input.

vx_node **vxIntegralImageNode** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuIntegralImage** (vx_context *context*, vx_image *input*, vx_image *output*);

input: The input image of VX_DF_IMAGE_U8 format.
output: The output image of VX_DF_IMAGE_U32 format.

Magnitude [3.31]

$mag(x,y) = \sqrt{grad_x(x,y)^2 + grad_y(x,y)^2}$

vx_node **vxMagnitudeNode** (vx_graph *graph*, vx_image *grad_x*, vx_image *grad_y*, vx_image *mag*);

vx_status **vxuMagnitude** (vx_context *context*, vx_image *grad_x*, vx_image *grad_y*, vx_image *mag*);

grad_{x, y}: The input {x, y} image of VX_DF_IMAGE_S16 format.
mag: The magnitude image of VX_DF_IMAGE_S16 format.

Mean and Standard Deviation [3.32]

Computes the mean and standard deviation of the input pixels.

vx_node **vxMeanStdDevNode** (vx_graph *graph*, vx_image *input*, vx_scalar *mean*, vx_scalar *stddev*);

vx_status **vxuMeanStdDev** (vx_context *context*, vx_image *input*, vx_float32 **mean*, vx_float32 **stddev*);

input: The input image. VX_DF_IMAGE_U8 is supported.
mean: Average pixel value of type VX_TYPE_FLOAT32.
stddev: Standard deviation of the pixel values of VX_TYPE_FLOAT32.

Median Filter [3.33]

Computes median values over a window of the input image.

vx_node **vxMedian3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output*);

vx_status **vxuMedian3x3** (vx_context *context*, vx_image *input*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.

Min, Max Location [3.34]

Locates the minima and maxima in an image.

vx_node **vxMinMaxLocNode** (vx_graph *graph*, vx_image *input*, vx_scalar *minVal*, vx_scalar *maxVal*, vx_array *minLoc*, vx_array *maxLoc*, vx_scalar *minCount*, vx_scalar *maxCount*);

vx_status **vxuMinMaxLoc** (vx_context *context*, vx_image *input*, vx_scalar *minVal*, vx_scalar *maxVal*, vx_array *minLoc*, vx_array *maxLoc*, vx_scalar *minCount*, vx_scalar *maxCount*);

input: The input image of VX_DF_IMAGE_{U8, S16} format.
{min, max}Val: The {min, max} value in the image.
{min, max}Loc: [Opt.] The {min, max} locations of type VX_TYPE_COORDINATES2D.
{min, max}Count: [Opt.] The number of detected {mins, maxes} in image. Type VX_TYPE_UINT32.

Optical Flow Pyramid (LK) [3.35]

Computes the optical flow between two pyramid images.

vx_node **vxOpticalFlowPyrLKNode** (vx_graph *graph*, vx_pyramid *old_images*, vx_pyramid *new_images*, vx_array *old_points*, vx_array *new_points_estimates*, vx_array *new_points*, vx_enum *termination*, vx_scalar *epsilon*, vx_scalar *num_iterations*, vx_scalar *use_initial_estimate*, vx_size *window_dimension*);

vx_status **vxuOpticalFlowPyrLK** (vx_context *context*, vx_pyramid *old_images*, vx_pyramid *new_images*, vx_array *old_points*, vx_array *new_points_estimates*, vx_array *new_points*, vx_enum *termination*, vx_scalar *epsilon*, vx_scalar *num_iterations*, vx_scalar *use_initial_estimate*, vx_size *window_dimension*);

{old, new}_images: The {old, new} image pyramid of VX_DF_IMAGE_U8.
old_points: Array of key points in a vx_array of VX_TYPE_KEYPOINT.
new_points_estimates: An array of estimation on what is the output key points in a vx_array of type VX_TYPE_KEYPOINT.
new_points: Array of key points in vx_array of type VX_TYPE_KEYPOINT.
termination: VX_TERM_CRITERIA_{ITERATIONS, EPSILON, BOTH}.
epsilon: The vx_float32 error for terminating the algorithm.
num_iterations: The number of iterations of type VX_TYPE_UINT32.
use_initial_estimate: VX_TYPE_BOOL set to vx_false_e or vx_true_e.
window_dimension: The size of the window.

Phase [3.36]

$\varphi = \tan^{-1}(grad_y(x,y) / grad_x(x,y)), \; 0 \le \varphi \le 255$

vx_node **vxPhaseNode** (vx_graph *graph*, vx_image *grad_x*, vx_image *grad_y*, vx_image *output*);

vx_status **vxuPhase** (vx_context *context*, vx_image *grad_x*, vx_image *grad_y*, vx_image *output*);

grad_{x, y}: The input {x, y} image of VX_DF_IMAGE_S16 format.
output: The phase image of VX_DF_IMAGE_U8 format.

Pixel-wise Multiplication [3.37]

$out(x,y) = in_1(x,y)\, in_2(x,y)\, scale$

vx_node **vxMultiplyNode** (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_scalar *scale*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_image *out*);

vx_status **vxuMultiply** (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_float32 *scale*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_image *out*);

in{1, 2}, out: Image of VX_DF_IMAGE_{U8, S16} format.
rounding_policy: VX_ROUND_POLICY_TO_{ZERO, NEAREST_EVEN}.
overflow_policy: VX_CONVERT_POLICY_{WRAP, SATURATE}.
scale: A non-negative value of type VX_TYPE_FLOAT32.

Remap [3.38]

$output(x,y) = input(map_x(x,y), map_y(x,y))$

vx_node **vxRemapNode** (vx_graph *graph*, vx_image *input*, vx_remap *table*, vx_enum *policy*, vx_image *output*);

vx_status **vxuRemap** (vx_context *context*, vx_image *input*, vx_remap *table*, vx_enum *policy*, vx_image *output*);

input, output: Image of type VX_DF_IMAGE_U8.
table: The remap table object.
policy: An interpolation type. VX_INTERPOLATION_{NEAREST_NEIGHBOR, BILINEAR}.

Scale Image [3.39]

vx_node **vxScaleImageNode** (vx_graph *graph*, vx_image *src*, vx_image *dst*, vx_enum *type*);

vx_status **vxuScaleImage** (vx_context *context*, vx_image *src*, vx_image *dst*, vx_enum *type*);

src, dst: The {source, destination} image of type VX_DF_IMAGE_U8.
type: An interpolation type. VX_INTERPOLATION_{NEAREST_NEIGHBOR, BILINEAR}.

Half Scale Gaussian [3.39]

Performs a Gaussian blur on an image then half-scales it.

vx_node **vxHalfScaleGaussianNode** (vx_graph *graph*, vx_image *input*, vx_image *output*, vx_int32 *kernel_size*);

vx_status **vxuHalfScaleGaussian** (vx_context *context*, vx_image *input*, vx_image *output*, vx_int32 *kernel_size*);

input, output: Image of VX_DF_IMAGE_U8 format.
kernel_size: Gaussian filter input size. Supported values are 1, 3, and 5.

Sobel3x3 [3.40]

Sobel filter, produces *X* and *Y* output planes.

vx_node **vxSobel3x3Node** (vx_graph *graph*, vx_image *input*, vx_image *output_x*, vx_image *output_y*);

vx_status **vxuSobel3x3** (vx_context *context*, vx_image *input*, vx_image *output_x*, vx_image *output_y*);

input: The input of VX_DF_IMAGE_U8 format.
output_{x, y}: [Opt.] The output gradient in the {x, y} direction of VX_DF_IMAGE_S16 format.

Table Lookup [3.41]

Uses a LUT to convert pixel values.

vx_node **vxTableLookupNode** (vx_graph *graph*, vx_image *input*, vx_lut *lut*, vx_image *output*);

vx_status **vxuTableLookup** (vx_context *context*, vx_image *input*, vx_lut *lut*, vx_image *output*);

input: Image of VX_DF_IMAGE_{U8, S16} format.
lut: The LUT of type VX_DF_IMAGE_{U8, S16}.
output: Image of VX_DF_IMAGE_{U8, S16} format.

Thresholding [3.42]

Produces a Boolean image by thresholding the input image.

vx_node **vxThresholdNode** (vx_graph *graph*, vx_image *input*, vx_threshold *thresh*, vx_image *output*);

vx_status **vxuThreshold** (vx_context *context*, vx_image *input*, vx_threshold *thresh*, vx_image *output*);

input: The input image of VX_DF_IMAGE_U8 format.
output: The output Boolean image.
thresh: Thresholding object.

Warp Affine [3.43]

Performs an affine transform on an image.

vx_node **vxWarpAffineNode** (vx_graph *graph*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*);

vx_status **vxuWarpAffine** (vx_context *context*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.
matrix: The affine matrix. Must be 2x3 of type VX_TYPE_FLOAT32.
type: The interpolation type. VX_INTERPOLATION_{NEAREST_NEIGHBOR, BILINEAR}.

(Continued on next page) ▶

◀ Vision Functions (cont.)

Warp Perspective [3.44]

Performs a perspective transform on an image.

vx_node **vxWarpPerspectiveNode** (vx_graph *graph*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*);

vx_status **vxuWarpPerspective** (vx_context *context*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*);

input, output: Image of VX_DF_IMAGE_U8 format.
matrix: Perspective matrix. Must be 3x3 of type VX_TYPE_FLOAT32.
type: The interpolation type. VX_INTERPOLATION_{NEAREST_NEIGHBOR, BILINEAR}.

Get Status [3.45.6]

Return status values from Object constructors if they fail.

vx_status **vxGetStatus** (vx_reference *reference*);

reference: The reference to check for construction errors.

Context Objects [3.48]

Attributes: enum vx_context_attribute_e: VX_CONTEXT_{VENDOR[_ID], UNIQUE_KERNELS, MODULES, REFERENCES, IMPLEMENTATION, EXTENSIONS[_SIZE], UNIQUE_KERNEL_TABLE, IMMEDIATE_BORDER[_POLICY], {CONVOLUTION, NONLINEAR}_MAX_DIMENSION, OPTICAL_FLOW_MAX_WINDOW_DIMENSION}

Create a vx_context.

vx_context **vxCreateContext** ();

Retrieve the context from any reference from within a context.

vx_context **vxGetContext** (vx_reference *reference*);

reference: The reference from which to extract the context.

Query the context for some specific information.

vx_status **vxQueryContext** (vx_context *context*, vx_enum *attribute*, void **ptr*, vx_size *size*);

attribute: Attribute to query of type vx_context_attribute_e.
ptr: Pointer to where to store the result.
size: Size of the container to which *ptr* points.

Release the OpenVX object context.

vx_status **vxReleaseContext** (vx_context **context*);

Set an attribute on the context.

vx_status **vxSetContextAttribute** (vx_context *context*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

attribute: Attribute to set of type vx_context_attribute_e.
ptr: Pointer to the data to which to set the attribute.
size: Size in bytes of the container to which *ptr* points.

Sets the default target of the immediate mode.

vx_status **vxSetImmediateModeTarget** (vx_context *context*, vx_enum *target_enum*, const char **target_string*);

context: The reference to the implementation context.
target_enum: Default immediate mode target enum to be set to vx_context object. VX_TARGET_{ANY, STRING, VENDOR_BEGIN}.
target_string: The target name ASCII string.

Convolution Objects [3.52]

Attributes: enum vx_convolution_attribute_e: VX_CONVOLUTION_{ROWS, COLUMNS, SCALE, SIZE}

Copy coefficients from/into a convolution object.

vx_status **vxCopyConvolutionCoefficients** (vx_convolution *conv*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

conv: The reference to the source/destination convolution object.
user_ptr: The address of the memory location at which to store the coefficient data.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create a reference to a convolution matrix object.

vx_convolution **vxCreateConvolution** (vx_context *context*, vx_size *columns*, vx_size *rows*);

columns, rows: Must be odd and >= 3 and less than VX_CONTEXT_CONVOLUTION_MAX_DIMENSION.

Queries an attribute on the convolution matrix object.

vx_status **vxQueryConvolution** (vx_convolution *conv*, vx_enum *attribute*, void **ptr*, vx_size *size*);

vxQueryConvolution parameters:

conv: The convolution matrix object to set.
attribute: VX_CONVOLUTION_{ROWS, COLUMNS, SCALE, SIZE}.
ptr: The location at which to store the resulting value.
size: The size in bytes of the container or data to which *ptr* points.

Release the reference to a convolution matrix.

vx_status **vxReleaseConvolution** (vx_convolution **conv*);

conv: The pointer to the convolution matrix to release.

Set attributes on the convolution object.

vx_status **vxSetConvolutionAttribute** (vx_convolution *conv*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

conv: The coordinates object to set.
attribute: VX_CONVOLUTION_{ROWS, COLUMNS, SCALE, SIZE}.
ptr: The pointer to the value to which to set the attribute.
size: The size in bytes of the container or data to which *ptr* points.

Array Objects [3.51]

Attributes: enum vx_array_attribute_e: VX_ARRAY_{ITEMTYPE, NUMITEMS, CAPACITY, ITEMSIZE}

Add items to the Array.

vx_status **vxAddArrayItems** (vx_array *arr*, vx_size *count*, const void **ptr*, vx_size *stride*);

arr: The reference to the array.
count: The total number of elements to insert.
ptr: Location from which to read the input values.
stride: The stride in bytes between elements.

Allows the application to copy a range from/into an array object.

ret **vxCopyArrayRange** (vx_array *array*, vx_size *range_start*, vx_size *range_end*, vx_size *user_stride*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

array: The reference to the source/destination array object.
range_start: The index of the first item of the array object to copy.
range_end: Index of item following last item of array object to copy.
user_stride: Number of bytes between the beginning of two consecutive items in the user memory pointed by *user_ptr*.
user_ptr: Address of memory location to store the requested data, or from where to get the data to store into the array object.
usage: Declares the effect of the copy with regard to the array object. VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create a reference to an Array object.

vx_array **vxCreateArray** (vx_context *context*, vx_enum *item_type*, vx_size *capacity*);

context: The reference to the overall Context.
item_type: The type of objects to hold.
capacity: The maximal number of items that the array can hold.

Delay Objects [3.67]

Attributes: enum vx_delay_attribute_e: VX_DELAY_{TYPE, SLOTS}

Age the internal delay ring by one.

vx_status **vxAgeDelay** (vx_delay *delay*);

delay: The pointer to the delay object.

Create a Delay object.

vx_delay **vxCreateDelay** (vx_context *context*, vx_reference *exemplar*, vx_size *slots*);

context: The reference to the system context.
exemplar: The exemplar object.
slots: The number of reference in the delay.

Retrieve a reference from a delay object.

vx_reference **vxGetReferenceFromDelay** (vx_delay *delay*, vx_int32 *index*);

delay: The reference to the delay object.
index: The index into the delay from which to extract the reference.

Query a vx_delay object attribute.

vx_status **vxQueryDelay** (vx_delay *delay*, vx_enum *attribute*, void **ptr*, vx_size *size*);

delay: The pointer to the delay object.
attribute: The attribute to query from VX_DELAY_{TYPE, SLOTS}.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

Release a reference to a delay object.

vx_status **vxReleaseDelay** (vx_delay **delay*);

delay: The pointer to the delay object.

Creates an opaque reference to a virtual Array with no direct user access.

vx_array **vxCreateVirtualArray** (vx_graph *graph*, vx_enum *item_type*, vx_size *capacity*);

graph: The reference to the parent graph.
item_type: The type of objects to hold, or zero to indicate an unspecified item type.
capacity: The maximal number of items that the array can hold, or zero to indicate an unspecified capacity.

Gives the application direct access to a range of an array object.

vx_status **vxMapArrayRange** (vx_array *array*, vx_size *range_start*, vx_size *range_end*, vx_map_id **map_id*, vx_size **stride*, void **ptr*, vx_enum *usage*, vx_enum *mem_type*, vx_uint32 *flags*);

array: The array object that contains the range to map.
range_start: The index of the first item of the array object to map.
range_end: The index of the item following the last item of the array object to map.
map_id: The address of a vx_map_id variable where the function returns a map identifier.
stride: The address of a vx_size variable where the function returns the memory layout of the mapped array range.
ptr: The address of a pointer that the function sets to the address where the requested data can be accessed.
usage: VX_{READ, WRITE}_ONLY or VX_READ_AND_WRITE.
mem_type: VX_MEMORY_TYPE_{NONE, HOST}.
flags: VX_NOGAP_X.

Query the Array for some specific information.

vx_status **vxQueryArray** (vx_array *arr*, vx_enum *attribute*, void **ptr*, vx_size *size*);

arr: The Array to release.
attribute: An attribute from vx_array_attribute_e.
ptr: Location at which to store the result.
size: The size in bytes of the container to which *ptr* points.

Release a reference of an Array object.

vx_status **vxReleaseArray** (vx_array **arr*);

arr: The Array to release.

Truncate an Array (remove items from the end).

vx_status **vxTruncateArray** (vx_array *arr*, vx_size *new_num_items*);

arr: The Array to release.
new_num_items: The new number of items for the Array.

Unmap and commit potential changes to an array object range.

vx_status **vxUnmapArrayRange** (vx_array *array*, vx_map_id *map_id*);

array: The reference to the array object to unmap.
map_id: The unique map identifier returned by **vxMapArrayRange**.

Object: Array (Advanced) [3.64]

Registers user-defined structures to the context.

vx_enum **vxRegisterUserStruct** (vx_context *context*, vx_size *size*);

size: The size of user struct in bytes.

Distribution Objects [3.53]

Attributes: enum vx_distribution_attribute_e: VX_DISTRIBUTION_{DIMENSIONS, OFFSET, RANGE, BINS, WINDOW, SIZE}

Allows the application to copy from/into a distribution object.

vx_status **vxCopyDistribution** (vx_distribution *distribution*, void * *user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

distribution: The reference to the source/destination distribution object.
user_ptr: The address of the memory location to store or get the data.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create a reference to a 1D Distribution.

vx_distribution **vxCreateDistribution** (vx_context *context*, vx_size *numBins*, vx_int32 *offset*, vx_int32 *range*);

context: The reference to the overall context.
numBins: The number of bins in the distribution.
offset: The start offset into the range value.
range: Total number of consecutive values of the distribution interval.

(Continued on next page) ▶

◀ Distribution Objects (cont.)

Allows application direct access to the distribution object.

vx_status **vxMapDistribution** (vx_distribution *distribution*, vx_map_id *map_id, void **ptr*, vx_enum *usage*, vx_enum *mem_type*, vx_bitfield *flags*);

distribution: The reference to the distribution object to map.
map_id: Address of variable where function returns a map identifier.
ptr: Address of a pointer that the function sets to the address where the requested data can be accessed.
usage: VX_READ_ONLY, VX_WRITE_ONLY, or VX_READ_AND_WRITE.
mem_type: VX_MEMORY_TYPE_{NONE, HOST}.
flags: Must be 0.

Query a Distribution object.

vx_status **vxQueryDistribution** (vx_distribution *distribution*, vx_enum *attribute*, void **ptr*, vx_size *size*);

distribution: Reference to the distribution.
attribute: Attribute to query. From vx_distribution_attribute_e.
ptr: The location at which to store the result.
size: The size in bytes of the container to which *ptr* points.

Release a reference to a distribution object.

vx_status **vxReleaseDistribution** (vx_distribution **distribution*);

distribution: The reference to the distribution.

Set the Distribution back to the memory.

vx_status **vxUnmapDistribution** (vx_distribution *distribution*, vx_map_id *map_id*);

distribution: The reference to the distribution object to unmap.
map_id: The unique map identifier that was returned when calling **vxMapDistribution**.

Image Objects [3.54]

Attributes: enum vx_image_attribute_e: VX_IMAGE_{WIDTH, HEIGHT, FORMAT, PLANES, SPACE, RANGE, SIZE, MEMORY_TYPE}

Computes size needed to retrieve an image patch from an image.

vx_size **vxComputeImagePatchSize** (vx_image *image*, const vx_rectangle_t **rect*, vx_uint32 *plane_index*);

image: The reference to the image from which to extract the patch.
rect: Coordinates. Must be 0 <= start < end <= dimension.
plane_index: The plane index from which to get the data.

Copies a rectangular patch from/into an image object plane.

vx_status **vxCopyImagePatch** (vx_image *image*, const vx_rectangle_t **image_rect*, vx_uint32 *image_plane_index*, const vx_imagepatch_addressing_t **user_addr*, void * *user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

image: The reference to the source/destination image object.
image_rect: The coordinates of the image patch.
image_plane_index: The plane index of the image object.
user_addr: The address of a structure describing the layout of the user memory location pointed by *user_ptr*.
user_ptr: The address of the memory location to store or get the data.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create an opaque reference to an image buffer.

vx_image **vxCreateImage** (vx_context *context*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *color*);

context: The reference to the implementation context.
width, height: The image {width, height} in pixels.
color: The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space.

Create a sub-image from a single plane channel of another image.

vx_image **vxCreateImageFromChannel** (vx_image *img*, vx_enum *channel*);

img: The reference to the parent image.
channel: The channel. VX_CHANNEL_{0, 1, 2, 3, R, G, B, A, Y, U, V}.

Create a reference to an externally allocated image object.

vx_image **vxCreateImageFromHandle** (vx_context *context*, vx_df_image *color*, const vx_imagepatch_addressing_t *addrs[ ], void *const *ptrs[ ], vx_enum *memory_type*);

context: The reference to the implementation context.
color: The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space.
addrs[ ]: The array of image patch addressing structures that define the dimension and stride of the array of pointers.
ptrs[ ]: The array of platform-defined references to each plane.
memory_type: VX_MEMORY_TYPE_{NONE, HOST}.

Graph Objects [3.49]

Attributes: enum vx_graph_attribute_e: VX_GRAPH_{NUMNODES, PERFORMANCE, NUMPARAMETERS, STATE}

Create an empty graph.

vx_graph **vxCreateGraph** (vx_context *context*);

context: The reference to the implementation context.

Returns a Boolean to indicate the state of graph verification.

vx_bool **vxIsGraphVerified** (vx_graph *graph*);

graph: The pointer to the graph to verify.

Cause the synchronous processing of a graph.

vx_status **vxProcessGraph** (vx_graph *graph*);

graph: The graph to execute.

Query attributes of the Graph.

vx_status **vxQueryGraph** (vx_graph *graph*, vx_enum *attribute*, void **ptr*, vx_size *size*);

graph: The reference to the created graph.
attribute: An attribute from vx_graph_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

Register a delay for auto-aging.

vx_status **vxRegisterAutoAging** (vx_graph *graph*, vx_delay *delay*);

graph: The graph to which the delay is registered for auto-aging.
delay: The delay to automatically age.

Release a reference to a graph.

vx_status **vxReleaseGraph** (vx_graph **graph*);

graph: The pointer to the graph to release.

Schedule a graph for future execution.

vx_status **vxScheduleGraph** (vx_graph *graph*);

graph: The graph to schedule.

Allows the user to set attributes on the graph.

vx_status **vxSetGraphAttribute** (vx_graph *graph*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

graph: The reference to the created graph.
attribute: VX_GRAPH_{NUMNODES, PERFORMANCE, NUMPARAMETERS, STATE}.
ptr: The location from which to read the value.
size: The size in bytes of the container to which *ptr* points.

Verify the state of the graph before it is executed.

vx_status **vxVerifyGraph** (vx_graph *graph*);

graph: The reference to the graph to verify.

Wait for a specific graph to complete.

vx_status **vxWaitGraph** (vx_graph *graph*);

graph: The graph on which to wait.

Create an image from another image given a rectangle.

vx_image **vxCreateImageFromROI** (vx_image *img*, vx_rectangle_t **rect*);

img: The reference to the parent image.
rect: The region of interest rectangle.

Create a reference to an image object that has a singular, uniform value in all pixels.

vx_image **vxCreateUniformImage** (vx_context *context*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *color*, const vx_pixel_value_t **value*);

context: The reference to the implementation context.
width, height: The image {width, height} in pixels.
color: The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space.
value: The pointer to the pixel value to which to set all pixels.

Create opaque reference to image buffer with no direct user access.

vx_image **vxCreateVirtualImage** (vx_graph *graph*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *color*);

graph: The reference to the parent graph.
width, height: The image {width, height} in pixels.
color: The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and color space. VX_DF_IMAGE_VIRT means the format is unspecified.

Access a specific indexed pixel in an image patch.

void * **vxFormatImagePatchAddress1d** (void **ptr*, vx_uint32 *index*, const vx_imagepatch_addressing_t **addr*);

ptr: The base pointer.
index: The 0-based index of the pixel count in the patch.
addr: Pointer to addressing mode information.

Access a specific pixel at a 2d coordinate in an image patch.

void * **vxFormatImagePatchAddress2d** (void **ptr*, vx_uint32 *x*, vx_uint32 *y*, const vx_imagepatch_addressing_t **addr*);

ptr: The base pointer.
addr: Pointer to addressing mode information.

Retrieve the valid region of the image as a rectangle.

vx_status **vxGetValidRegionImage** (vx_image *image*, vx_rectangle_t **rect*);

image: The image from which to retrieve the valid region.
rect: The destination rectangle.

Gives the application direct access to a rectangular patch of an image object plane.

vx_status **vxMapImagePatch** (vx_image *image*, const vx_rectangle_t * *rect*, vx_uint32 *plane_index*, vx_map_id * *map_id*, vx_imagepatch_addressing_t * *addr*, void **ptr*, vx_enum *usage*, vx_enum *mem_type*, vx_uint32 *flags*);

vxMapImagePatch parameters:

image: Reference to the image object containing the patch to map.
rect: The coordinates of the image patch.
plane_index: The plane index of the image object to be accessed.
map_id: Address of a vx_map_id variable where function returns a map identifier.
addr: Address of a structure describing memory layout of image patch.
ptr: The address of a pointer that the function sets to the address where the requested data can be accessed.
usage: Declares the access mode for image patch. VX_{READ, WRITE}_ONLY or VX_READ_AND_WRITE.
mem_type: Type of image patch memory. VX_MEMORY_TYPE_{NONE, HOST}.
flags: VX_NOGAP_X.

Retrieve various attributes of an image.

vx_status **vxQueryImage** (vx_image *image*, vx_enum *attribute*, void **ptr*, vx_size *size*);

image: The reference to the image to query.
attribute: The attribute to query. From vx_image_attribute_e.
ptr: The pointer to the location at which to store the result.
size: The size of the container to which *ptr* points.

Release a reference to an image object.

vx_status **vxReleaseImage** (vx_image **image*);

image: The reference to the image to release.

Set attributes on the image.

vx_status **vxSetImageAttribute** (vx_image *image*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

image: The reference to the image on which to set the attribute.
attribute: The attribute to set. From vx_image_attribute_e.
ptr: The pointer to the location from which to read the value.
size: The size in bytes of the container to which *ptr* points.

Sets valid rectangle for image according to a supplied rectangle.

vx_status **vxSetImageValidRectangle** (vx_image *image*, const vx_rectangle_t **rect*);

image: The reference to the image.
rect: The value to be set to the image valid rectangle.

Create a reference to an externally allocated image object.

vx_status **vxSwapImageHandle** (vx_image *image*, void *const *new_ptrs[ ], void **prev_ptrs[ ], vx_size *num_planes*);

image: The reference to the image created from handle.
new_ptrs[ ]: The pointer to a caller-owned array that contains the new image handle.
prev_ptrs[ ]: The pointer to a caller-owned array in which the application returns the previous image handle.
num_planes: Number of planes in the image.

Unmap and commit potential changes to an image object patch.

vx_status **vxUnmapImagePatch** (vx_image *image*, vx_map_id *map_id*);

image: The reference to the image object to unmap.
map_id: The unique map identifier returned by **vxMapImagePatch**.

Kernel Objects [3.68]

Attributes: enum vx_kernel_attribute_e: VX_KERNEL_{PARAMETERS, NAME, ENUM, LOCAL_DATA_SIZE}

Obtain a reference to kernel using vx_kernel_e enumeration.

vx_kernel **vxGetKernelByEnum** (vx_context *context*, vx_enum *kernel*);

context: The reference to the implementation context.
kernel: • Value from vx_kernel_e or vendor or client-defined value.

Obtain a reference to a kernel using a string to specify the name.

vx_kernel **vxGetKernelByName** (vx_context *context*, const vx_char **name*);

context: The reference to the implementation context.
name: The string of the name of the kernel to get.

Query the kernel to get information about the number of parameters, enum values, etc.

vx_status **vxQueryKernel** (vx_kernel *kernel*, vx_enum *attribute*, void **ptr*, vx_size *size*);

kernel: The kernel reference to query.
attribute: The attribute to query. From vx_kernel_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

Release the reference to the kernel.

vx_status **vxReleaseKernel** (vx_kernel **kernel*);

kernel: The pointer to the kernel reference to release.

Matrix Objects [3.56]

Attributes: enum vx_matrix_attribute_e: VX_MATRIX_{TYPE, ROWS, COLUMNS, SIZE, ORIGIN, PATTERN}

Copy from or to a matrix data.

vx_status **vxCopyMatrix** (vx_matrix *matrix*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

matrix: The reference to the source/destination matrix object.
user_ptr: The address of the memory location for storage or retrieval.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Creates a reference to a matrix object.

vx_matrix **vxCreateMatrix** (vx_context *c*, vx_enum *data_type*, vx_size *columns*, vx_size *rows*);

vxCreateMatrix parameters:

c: The reference to the overall context.
data_type: Unit format of matrix. VX_TYPE_{UINT8, INT32, FLOAT32}.
columns: The first dimension.
rows: The second dimension.

Creates a reference to a matrix object from a Boolean pattern.

vx_matrix **vxCreateMatrixFromPattern** (vx_context *c*, vx_enum *pattern*, vx_size *columns*, vx_size *rows*);

c: The reference to the overall context.
pattern: From vx_matrix_attribute_e.
columns: The first dimension.
rows: The second dimension.

Queries an attribute on the matrix object.

vx_status **vxQueryMatrix** (vx_matrix *mat*, vx_enum *attribute*, void **ptr*, vx_size *size*);

mat: The reference to the matrix.
attribute: The attribute to query. From vx_matrix_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

Releases a reference to a matrix object.

vx_status **vxReleaseMatrix** (vx_matrix **mat*);

mat: The reference to the matrix.

ObjectArray Objects [3.61]

Attributes: enum vx_object_array_attribute_e: VX_OBJECT_ARRAY_{ITEMTYPE, NUMITEMS}

Creates a reference to an ObjectArray of count objects.

vx_object_array **vxCreateObjectArray** (vx_context *context*, vx_reference *exemplar*, vx_size *count*);

context: The reference to the overall Context.
exemplar: The exemplar object that defines the metadata of the created objects in the ObjectArray.
count: The number of Objects to create in the ObjectArray.

Creates an opaque reference to a virtual ObjectArray.

vx_object_array **vxCreateVirtualObjectArray** (vx_graph *graph*, vx_reference *exemplar*, vx_size *count*);

graph: The reference to the graph in which to create the ObjectArray.
exemplar: The exemplar object that defines the type of object in the ObjectArray.
count: The number of Objects to create in the ObjectArray.

Retrieves the reference to the OpenVX Object.

vx_reference **vxGetObjectArrayItem** (vx_object_array *arr*, vx_uint32 *index*);

arr: The ObjectArray.
index: The index of the object in the ObjectArray.

Queries an attribute from the ObjectArray.

vx_status **vxQueryObjectArray** (vx_object_array *arr*, vx_enum *attribute*, void **ptr*, vx_size *size*);

arr: The reference to the ObjectArray.
attribute: VX_OBJECT_ARRAY_{ITEMTYPE, NUMITEMS}
ptr: The location at which to store the resulting value.
size: The size in bytes of the container to which *ptr* points.

Releases a reference of an ObjectArray Object.

vx_status **vxReleaseObjectArray** (vx_object_array **arr*);

arr: The pointer to the ObjectArray to release.

LUT Objects [3.55]

Attributes: enum vx_lut_attribute_e: VX_LUT_{TYPE, COUNT, SIZE, OFFSET}

Allows the application to copy from/into a LUT object.

vx_status **vxCopyLUT** (vx_lut *lut*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

lut: The reference to the source/destination LUT object.
user_ptr: The address of the memory location to store or get data.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: VX_MEMORY_TYPE_{NONE, HOST}.

Create LUT object of a given type.

vx_lut **vxCreateLUT** (vx_context *context*, vx_enum *data_type*, vx_size *count*);

context: The reference to the context.
data_type: The type of data stored in the LUT.
count: The number of entries desired.

Allows the application to copy from/into a LUT object.

vx_status **vxMapLUT** (vx_lut *lut*, vx_map_id **map_id*, void ***ptr*, vx_enum *usage*, vx_enum *mem_type*, vx_bitfield *flags*);

lut: The reference to the LUT object to map.
map_id: Address where the function returns a map identifier.
ptr: The address of a pointer that the function sets to the address where the requested data can be accessed.
usage: Declares the access mode for the LUT. VX_{READ, WRITE}_ONLY or VX_READ_AND_WRITE.
mem_type: Specifies the type of the memory where the LUT is requested to be mapped. VX_MEMORY_TYPE_{NONE, HOST}.
flags: Must be 0.

Query attributes from a LUT.

vx_status **vxQueryLUT** (vx_lut *lut*, vx_enum *attribute*, void **ptr*, vx_size *size*);

lut: The LUT to query.
attribute: Attribute to query. From vx_lut_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

Release a reference to a LUT object.

vx_status **vxReleaseLUT** (vx_lut **lut*);

lut: The pointer to the LUT to release.

Allows the application to copy from/into a LUT object.

vx_status **vxUnmapLUT** (vx_lut *lut*, vx_map_id *map_id*);

lut: The reference to the LUT object to unmap.
map_id: The unique map identifier returned by **vxMapLUT**.

Parameter Objects [3.69]

Attributes: enum vx_parameter_attribute_e: VX_PARAMETER_{INDEX, DIRECTION, TYPE, STATE, REF}

Retrieve a vx_parameter from a vx_kernel.

vx_parameter **vxGetKernelParameterByIndex** (vx_kernel *kernel*, vx_uint32 *index*);

kernel: The reference to the kernel.
index: The index of the parameter.

Retrieve a vx_parameter from a vx_node.

vx_parameter **vxGetParameterByIndex** (vx_node *node*, vx_uint32 *index*);

node: The node from which to extract the parameter.
index: The index of the parameter.

Query a parameter to determine its meta-information.

vx_status **vxQueryParameter** (vx_parameter *param*, vx_enum *attribute*, void **ptr*, vx_size *size*);

param: The reference to the parameter.
attribute: The attribute to query from vx_parameter_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

Release a reference to a parameter object.

vx_status **vxReleaseParameter** (vx_parameter **param*);

param: The pointer to the parameter.

Node Objects [3.50]

Attributes: enum vx_node_attribute_e: VX_NODE_{STATUS, PERFORMANCE, BORDER, LOCAL_DATA_{SIZE, PTR}, PARAMETERS, IS_REPLICATED, REPLICATE_FLAGS, VALID_RECT_RESET}

Queries information out of a node.

vx_status **vxQueryNode** (vx_node *node*, vx_enum *attribute*, void **ptr*, vx_size *size*);

node: The reference to the node to query.
attribute: The vx_node_attribute_e value to query.
ptr: The location at which to store the resulting value.
size: The size in bytes of the container to which *ptr* points.

Release a reference to a Node object.

vx_status **vxReleaseNode** (vx_node **node*);

node: The reference of the node to release.

Remove a Node from its parent Graph and release it.

void **vxRemoveNode** (vx_node **node*);

node: The reference to the node to remove.

Creates replicas of the same node to process a set of objects.

vx_status **vxReplicateNode** (vx_graph *graph*, vx_node *first_node*, vx_bool *replicate*[ ], vx_uint32 *number_of_parameters*);

graph: The reference to the graph.
first_node: The reference to the graph node to replicate.
***replicate*[ ]:** An array indicating the parameters to replicate.
number_of_parameters: Number of elements in the replicate array.

Set attributes of a node before Graph Validation.

vx_status **vxSetNodeAttribute** (vx_node *node*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

node: The reference to the node.
attribute: The vx_node_attribute_e value to set.
ptr: Pointer to desired value of the attribute.
size: The size in bytes of the objects to which *ptr* points.

Set attributes of a node before Graph Validation.

vx_status **vxSetNodeTarget** (vx_node *node*, vx_enum *target_enum*, const char **target_string*);

node: The reference to the vx_node object.
target_enum: The target enum to be set to the vx_node object. VX_TARGET_{ANY, STRING, VENDOR_BEGIN}.
target_string: The target name ASCII string.

Object: Node (Advanced) [3.65]

Defines the advanced features of the Node Interface.

vx_node **vxCreateGenericNode** (vx_graph *graph*, vx_kernel *kernel*);

graph: The reference to the graph in which this node exists.
kernel: The kernel reference to associate with this new node.

Set the specified parameter data for a kernel on the node.

vx_status **vxSetParameterByIndex** (vx_node *node*, vx_uint32 *index*, vx_reference *value*);

node: The node that contains the kernel.
index: The index of the parameter.
value: The desired value of the parameter.

Associate parameter and data references with a kernel on a node.

vx_status **vxSetParameterByReference** (vx_parameter *parameter*, vx_reference *value*);

parameter: The reference to the parameter.
value: The value to associate with the parameter.

Pyramid Objects [3.57]

Attributes: enum vx_pyramid_attribute_e: VX_PYRAMID_{LEVELS, SCALE, WIDTH, HEIGHT, FORMAT}

Create a reference to a pyramid object.

vx_pyramid **vxCreatePyramid** (vx_context *context*, vx_size *levels*, vx_float32 *scale*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *format*);

context: The reference to the overall context.
levels: The number of levels desired.
scale: This must be a non-zero positive value.
width: The width of the 0th-level image in pixels.
height: The height of the 0th-level image in pixels.
format: Format of all images in the pyramid or VX_DF_IMAGE_VIRT.

Create a reference to a virtual pyramid object.

vx_pyramid **vxCreateVirtualPyramid** (vx_graph *graph*, vx_size *levels*, vx_float32 *scale*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *format*);

graph: The reference to the parent graph.
levels: The number of levels desired.
scale: This must be a non-zero positive value.
width: The width of the 0th-level image in pixels.
height: The height of the 0th-level image in pixels.
format: Format of all images in the pyramid or VX_DF_IMAGE_VIRT.

Retrieve a level of the pyramid as a vx_image.

vx_image **vxGetPyramidLevel** (vx_pyramid *pyr*, vx_uint32 *index*);

pyr: The pointer to the pyramid.
index: The index of the level, such that *index* is less than *levels*.

Query an attribute from an image pyramid.

vx_status **vxQueryPyramid** (vx_pyramid *pyr*, vx_enum *attribute*, void **ptr*, vx_size *size*);

pyr: The pointer to the pyramid to query.
attribute: The attribute to query from vx_pyramid_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

Release a reference to a pyramid object.

vx_status **vxReleasePyramid** (vx_pyramid **pyr*);

pyr: The pointer to the pyramid to release.

Remap Objects [3.58]

Attributes: enum vx_remap_attribute_e: VX_REMAP_{SOURCE_{WIDTH, HEIGHT}, DESTINATION_{WIDTH, HEIGHT}}

Create a remap table object.

vx_remap **vxCreateRemap** (vx_context *context*, vx_uint32 *src_width*, vx_uint32 *src_height*, vx_uint32 *dst_width*, vx_uint32 *dst_height*);

context: The reference to the overall context.
src_{width, height}: The {width, height} of the source image in pixels.
dst_{width, height}: The {width, height} of destination image in pixels.

Retrieve the source pixel point from a destination pixel.

vx_status **vxGetRemapPoint** (vx_remap *table*, vx_uint32 *dst_x*, vx_uint32 *dst_y*, vx_float32 **src_x*, vx_float32 **src_y*);

table: Remap table reference.
dst_{x, y}: The destination {x, y} coordinate.
src_{x, y}: Pointer to location where to store the source coordinates in float representation to allow interpolation.

Query attributes from a remap table.

vx_status **vxQueryRemap** (vx_remap *r*, vx_enum *attribute*, void **ptr*, vx_size *size*);

r: The remap to query.
attribute: An attribute from vx_remap_attribute_e.
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

Release a reference to a remap table object.

vx_status **vxReleaseRemap** (vx_remap **table*);

table: A pointer to the remap table to release.

Assign a destination pixel mapping to the source pixel.

vx_status **vxSetRemapPoint** (vx_remap *table*, vx_uint32 *dst_x*, vx_uint32 *dst_y*, vx_float32 *src_x*, vx_float32 *src_y*);

table: Remap table reference.
dst_{x, y}: The destination {x, y} coordinate.
src_{x, y}: The source {x, y} coordinate in float to allow interpolation.

Advanced Framework

Node Callbacks [3.71]

Assign a callback to a node.

vx_status **vxAssignNodeCallback** (vx_node *node*, vx_nodecomplete_f *callback*);

node: The reference to the node.
callback: Callback function pointer.

Retrieve the current node callback function pointer.

vx_nodecomplete_f **vxRetrieveNodeCallback** (vx_node *node*);

node: The reference to the node.

Callback prototype.

typedef vxAction (***vx_nodecomplete_f**)(vx_node *node*);

node: The reference to the vx_node object.

Scalar Objects [3.59]

Attribute: enum vx_scalar_attribute_e: VX_SCALAR_TYPE

Allows the application to copy from/into a scalar object.

vx_status **vxCopyScalar**(vx_scalar *scalar*, void **user_ptr*, vx_enum *usage*, vx_enum *user_mem_type*);

scalar: The reference to the source/destination scalar object.
user_ptr: Address of the memory location where to store the requested data, or from where to get the data to store into the scalar object.
usage: VX_READ_ONLY or VX_WRITE_ONLY.
user_mem_type: Type of the memory referenced by the *user_addr*. VX_MEMORY_TYPE_{NONE, HOST}.

Create a reference to a scalar object.

vx_scalar **vxCreateScalar** (vx_context *context*, vx_enum *data_type*, const void **ptr*);

context: The reference to the system context.
data_type: • The type of the scalar, from vx_type_e (on page 8).
ptr: Pointer to the initial value of the scalar.

Query attributes from a scalar.

vx_status **vxQueryScalar** (vx_scalar *scalar*, vx_enum *attribute*, void **ptr*, vx_size *size*);

scalar: The scalar object.
attribute: VX_SCALAR_TYPE.
ptr: Location at which to store the result.
size: The size of the container to which *ptr* points.

Gets the scalar value out of a reference.

vx_status **vxReadScalarValue** (vx_scalar *ref*, void **ptr*);

ref: The reference from which to get the scalar value.
ptr: A pointer to a location to which to copy the scalar value.

Release a reference to a scalar object.

vx_status **vxReleaseScalar** (vx_scalar **scalar*);

scalar: The pointer to the scalar to release.

Sets the scalar value in a reference.

vx_status **vxWriteScalarValue** (vx_scalar *ref*, const void **ptr*);

ref: The reference to which to set the scalar value.
ptr: A pointer to a location from where to copy the scalar value.

Log [3.73]

Add a line to the log.

void **vxAddLogEntry** (vx_reference *ref*, vx_status *status*, const char **message*, ...);

ref: The reference to add the log entry against.
status: The status code. VX_SUCCESS status entries are ignored.
message: The human readable message to add to the log.

Register a callback facility to receive error logs.

void **vxRegisterLogCallback** (vx_context *context*, vx_log_callback_f *callback*, vx_bool *reentrant*);

context: The overall context to OpenVX.
callback: The callback function or NULL.
reentrant: Boolean reentrancy flag indicating whether the callback may be entered from multiple simultaneous tasks or threads.

Callback prototype for vxRegisterLogCallback.

typedef void (***vx_log_callback_f**) (vx_context *context*, vx_reference *ref*, vx_status *status*, vx_char *string*[ ]);

context: The overall context to OpenVX.
ref: The reference to add the log entry against.
status: The status code.

Reference Objects [3.47]

Attributes: enum vx_reference_attribute_e: VX_REF_ATTRIBUTE_{COUNT, TYPE, NAME}

Query any reference type for basic information (count, type).

vx_status **vxQueryReference** (vx_reference *ref*, vx_enum *attribute*, void **ptr*, vx_size *size*);

ref: Reference to the object to query.
attribute: The value to query. VX_REF_ATTRIBUTE_{COUNT, TYPE, NAME}
ptr: The location at which to store the resulting value.
size: The size of the container to which *ptr* points.

Release a reference.

vx_status **vxReleaseReference** (vx_reference **ref_ptr*);

ref_ptr: The pointer to the reference of the object to release.

Increments the reference counter of an object.

vx_status **vxRetainReference** (vx_reference *ref*);

ref: The reference to retain.

Name a reference.

vx_status **vxSetReferenceName** (vx_reference *ref*, const vx_char **name*);

ref: The reference to the object to be named.
name: NULL if not named, or a pointer to NUL-terminated name.

Threshold Objects [3.60]

Attributes: enum vx_threshold_attribute_e: VX_REMAP_{TYPE, THRESHOLD_{VALUE, LOWER, UPPER}, TRUE_VALUE, FALSE_VALUE, DATA_TYPE}

Create a reference to a threshold object of a given type.

vx_threshold **vxCreateThreshold** (vx_context *c*, vx_enum *thresh_type*, vx_enum *data_type*);

c: The reference to the overall context.
thresh_type: The type of threshold to create.
data_type: The data type of the threshold's value(s).

Query an attribute on the threshold object.

vx_status **vxQueryThreshold** (vx_threshold *thresh*, vx_enum *attribute*, void **ptr*, vx_size *size*);

thresh: Threshold object to query.
attribute: The attribute to modify from vx_threshold_attribute_e.
ptr: The location at which to store the result.
size: The size of the container pointed to by *ptr*.

Release a reference to a threshold object.

vx_status **vxReleaseThreshold** (vx_threshold **thresh*);

thresh: A pointer to a threshold object to release.

Set attributes on the threshold object.

vx_status **vxSetThresholdAttribute** (vx_threshold *thresh*, vx_enum *attribute*, const void **ptr*, vx_size *size*);

thresh: Threshold object to set.
attribute: The attribute to modify from vx_threshold_attribute_e.
ptr: The pointer to the value to which to set the attribute.
size: The size of the data pointed to by *ptr*.

Hints [3.74]

Provide a generic API to give platform-specific hints.

vx_status **vxHint** (vx_reference *reference*, vx_enum *hint*, const void **data*, vx_size *data_size*);

reference: The reference to the object to hint at.
hint: • See vx_hint_e.
data: Optional vendor specific data.
data_size: Size of the data structure *data*.

Directives [3.75]

Provide a generic API to give platform-specific directives.

vx_status **vxDirective** (vx_reference *reference*, vx_enum *directive*);

reference: The reference to the object to set the directive on.
directive: • The directive to set. See vx_directive_e.

User Kernels [3.76]

Set the signatures of the custom kernel.

vx_status **vxAddParameterToKernel** (vx_kernel *kernel*, vx_uint32 *index*, vx_enum *dir*, vx_enum *data_type*, vx_enum *state*);

kernel: The reference to the kernel.
index: The index of the parameter to add.
dir: VX_INPUT or VX_OUTPUT.
data_type: • Type of parameter, from vx_type_e (on page 8).
state: • Parameter state. See vx_parameter_state_e.

(Continued on next page) ▶

◀ Advanced Framework (cont.)

Allows users to add custom kernels to the known kernel database.

vx_kernel **vxAddUserKernel** (vx_context *context*,
const vx_char *name*[VX_MAX_KERNEL_NAME],
vx_enum *enumeration*, vx_kernel_f *func_ptr*,
vx_uint32 *numParams*, vx_kernel_validate_f *validate*,
vx_kernel_initialize_f *init*, vx_kernel_deinitialize_f *deinit*);

context: The reference to the implementation context.
name: The string to use to match the kernel.
enumeration: Enumerated value of the kernel to be used by clients.
func_ptr: The process-local function pointer to be invoked.
numParams: The number of parameters for this kernel.
validate: The pointer to vx_kernel_validate_f, which validates parameters to this kernel.
init, deinit: The kernel {initialization, deinitialization} function.

Allocates/registers user-defined kernel enumeration to a context.

vx_status **vxAllocateUserKernelId** (vx_context *context*,
vx_enum **pKernelEnumId*);

context: The reference to the implementation context.
pKernelEnumId: The pointer to return vx_enum for user-defined kernel.

Allocates/registers user-defined kernel library ID to a context.

vx_status **vxAllocateUserKernelLibraryId** (vx_context *context*,
vx_enum **pLibraryId*);

context: The reference to the implementation context.
pLibraryId: The pointer to vx_enum for user-kernel libraryId.

Called after parameters have been added and kernel is ready.

vx_status **vxFinalizeKernel** (vx_kernel *kernel*);

kernel: The reference to the kernel.

Load one or more kernels into the OpenVX context.

vx_status **vxLoadKernels** (vx_context *context*,
const vx_char **module*);

context: The reference to the implementation context.
module: The short name of the module to load.

Remove a vx_kernel from the vx_context.

vx_status **vxRemoveKernel** (vx_kernel *kernel*);

kernel: The reference to the kernel.

Set kernel attributes.

vx_status **vxSetKernelAttribute** (vx_kernel *kernel*,
vx_enum *attribute*, const void **ptr*, vx_size *size*);

kernel: The reference to the kernel.
attribute: The attribute to set, from vx_kernel_attribute_e. {VX_KERNEL_{PARAMETERS, NAME, ENUM, LOCAL_DATA_SIZE}}
ptr: Pointer to the attribute.
size: The size in bytes of the container to which *ptr* points.

Set the attributes of a vx_meta_format object.

vx_status **vxSetMetaFormatAttribute** (vx_meta_format *meta*,
vx_enum *attribute*, const void **ptr*, vx_size *size*);

meta: The reference to the vx_meta_format object to set.
attribute: Use the subset of data object attributes that define the meta data of this object or attributes from vx_meta_format_attribute_e.
ptr: The input pointer of the value to set on the meta format object.
size: The size in bytes of the container to which *ptr* points.

Set the attributes of a vx_meta_format object.

vx_status **vxSetMetaFormatFromReference** (
vx_meta_format *meta*, vx_reference *exemplar*);

meta: The meta format object to set.
exemplar: The exemplar data object.

Unload one or more kernels from the module.

vx_status **vxUnloadKernels** (vx_context *context*,
const vx_char **module*);

context: The reference to the implementation context.
module: The short name of the module to unload.

Graph Parameters [3.77]

Add the given parameter extracted from a vx_node to the graph.

vx_status **vxAddParameterToGraph** (vx_graph *graph*,
vx_parameter *parameter*);

graph: The graph reference.
parameter: Parameter reference to add to the graph from the node.

Retrieve a vx_parameter from a vx_graph.

vx_parameter **vxGetGraphParameterByIndex** (
vx_graph *graph*, vx_uint32 *index*);

graph: The graph reference.
index: The parameter index.

Set a reference to the parameter on the graph.

vx_status **vxSetGraphParameterByIndex** (vx_graph *graph*,
vx_uint32 *index*, vx_reference *value*);

graph: The graph reference.
index: The parameter index.
value: The reference to set to the parameter.

Enumerators

vx_enum_e

VX_ENUM_ACCESSOR	vx_accessor_e
VX_ENUM_ACTION	vx_action_e
VX_ENUM_BORDER	vx_border_e
VX_ENUM_BORDER_POLICY	vx_border_policy_e
VX_ENUM_CHANNEL	vx_channel_e
VX_ENUM_COLOR_RANGE	vx_channel_range_e
VX_ENUM_COLOR_SPACE	vx_color_space_e
VX_ENUM_CONVERT_POLICY	vx_convert_policy_e
VX_ENUM_DIRECTION	vx_direction_e
VX_ENUM_DIRECTIVE	vx_directive_e
VX_ENUM_GRAPH_STATE	vx_graph_state_e
VX_ENUM_HINT	vx_hint_e
VX_ENUM_INTERPOLATION	vx_interpolation_type_e
VX_ENUM_MEMORY_TYPE	vx_memory_type_e
VX_ENUM_NONLINEAR	vx_non_linear_filter_e
VX_ENUM_NORM_TYPE	vx_norm_type_e
VX_ENUM_PARAMETER_STATE	vx_parameter_state_e
VX_ENUM_PATTERN	vx_pattern_e
VX_ENUM_ROUND_POLICY	vx_round_policy_e
VX_ENUM_TARGET	vx_target_e
VX_ENUM_TERM_CRITERIA	vx_termination_criteria_e
VX_ENUM_THRESHOLD_TYPE	vx_threshold_type_e

vx_accessor_e
VX_{READ, WRITE}_ONLY
VX_READ_AND_WRITE

vx_action_e
VX_ACTION_{CONTINUE, ABANDON}

vx_border_e
VX_BORDER_{UNDEFINED, CONSTANT, REPLICATE}

vx_border_policy_e
VX_BORDER_POLICY_DEFAULT_TO_UNDEFINED
VX_BORDER_POLICY_RETURN_ERROR

vx_channel_e
VX_CHANNEL_{0, 1, 2, 3}
VX_CHANNEL_{R, G, B, A}
VX_CHANNEL_{Y, U , V}

vx_channel_range_e
VX_CHANNEL_RANGE_{FULL, RESTRICTED}

vx_color_space_e
VX_COLOR_SPACE_{NONE, DEFAULT}
VX_COLOR_SPACE_BT601_{525, 625}
VX_COLOR_SPACE_BT709

vx_convert_policy_e
VX_CONVERT_POLICY_{WRAP, SATURATE}

vx_df_image_e
VX_DF_IMAGE_VIRT
VX_DF_IMAGE_RGB
VX_DF_IMAGE_RGBX
VX_DF_IMAGE_NV12
VX_DF_IMAGE_NV21
VX_DF_IMAGE_UYVY
VX_DF_IMAGE_YUYV
VX_DF_IMAGE_IYUV
VX_DF_IMAGE_YUV4
VX_DF_IMAGE_U8
VX_DF_IMAGE_U16
VX_DF_IMAGE_S16
VX_DF_IMAGE_U32
VX_DF_IMAGE_S32

vx_direction_e
VX_INPUT, VX_OUTPUT, VX_BIDIRECTIONAL

vx_directive_e
VX_DIRECTIVE_{DISABLE, ENABLE}_LOGGING
VX_DIRECTIVE_{DISABLE, ENABLE}_PERFORMANCE

vx_graph_state_e
VX_GRAPH_STATE_{UNVERIFIED, VERIFIED}
VX_GRAPH_STATE_{RUNNING, ABANDONED, COMPLETED}

vx_hint_e
VX_HINT_PERFORMANCE_DEFAULT
VX_HINT_PERFORMANCE_LOW_POWER
VX_HINT_PERFORMANCE_HIGH_SPEED

vx_interpolation_type_e
VX_INTERPOLATION_NEAREST_NEIGHBOR
VX_INTERPOLATION_BILINEAR
VX_INTERPOLATION_AREA

vx_map_flag_e
VX_NOGAP_X

vx_memory_type_e
VX_MEMORY_TYPE_NONE
VX_MEMORY_TYPE_ HOST

vx_non_linear_filter_e
VX_NONLINEAR_FILTER_MEDIAN
VX_NONLINEAR_FILTER_MIN
VX_NONLINEAR_FILTER_ MAX

vx_norm_type_e
VX_NORM_{L1, L2}

vx_parameter_state_e
VX_PARAMETER_STATE_REQUIRED
VX_PARAMETER_STATE_OPTIONAL

vx_pattern_e
VX_PATTERN_BOX
VX_PATTERN_CROSS
VX_PATTERN_DISK
VX_PATTERN_OTHER

vx_round_policy_e
VX_ROUND_POLICY_TO_ZERO
VX_ROUND_POLICY_TO_NEAREST_EVEN

vx_status_e
VX_STATUS_MIN
VX_ERROR_REFERENCE_NONZERO
VX_ERROR_MULTIPLE_WRITERS
VX_ERROR_GRAPH_ABANDONED
VX_ERROR_GRAPH_SCHEDULED
VX_ERROR_INVALID_{SCOPE, NODE, GRAPH, TYPE, VALUE, DIMENSION, FORMAT, LINK, REFERENCE, MODULE, PARAMETERS}
VX_ERROR_OPTIMIZED_AWAY
VX_ERROR_NO_MEMORY
VX_ERROR_NO_RESOURCES
VX_ERROR_NOT_COMPATIBLE
VX_ERROR_NOT_ ALLOCATED
VX_ERROR_NOT_ SUFFICIENT
VX_ERROR_NOT_ SUPPORTED
VX_ERROR_NOT_ IMPLEMENTED
VX_FAILURE
VX_SUCCESS

vx_target_e
VX_TARGET_ANY
VX_TARGET_STRING
VX_TARGET_VENDOR_BEGIN

vx_termination_criteria_e
VX_TERM_CRITERIA_ITERATIONS
VX_TERM_CRITERIA_EPSILON
VX_TERM_CRITERIA_BOTH

vx_threshold_type_e
VX_THRESHOLD_TYPE_BINARY
VX_THRESHOLD_TYPE_ RANGE

Macros [3.45.3]

Defines the major version number macro.

```
#define VX_VERSION_MAJOR( x ) ((x & 0xFF) << 8)
```

Defines the minor version number macro.

```
#define VX_VERSION_MINOR( x ) ((x & 0xFF) << 0)
```

Defines the OpenVX Version Number.

```
#define VX_VERSION VX_VERSION_1_1
```

A type mask removes the scalar/object type from the attribute.

```
#define VX_TYPE_MASK (0x000FFF00)
```

Converts a set of four chars into a uint32_t container of a VX_DF_IMAGE code.

```
#define VX_DF_IMAGE( a, b, c, d ) ((a) | (b << 8) | (c << 16) | (d << 24))
```

Defines the manner in which to combine the Vendor and Object IDs to get the base value of the enumeration.

```
#define VX_ENUM_BASE( vendor, id ) (((vendor) << 20) | (id << 12))
```

Use to aid in debugging values in OpenVX.

```
#define VX_FMT_REF "%p"
```

Use to aid in debugging values in OpenVX.

```
#define VX_FMT_SIZE "%zu"
```

Use to indicate the 1:1 ratio in Q22.10 format.

```
#define VX_SCALE_UNITY (1024u)
```

Image Access Example [2.15.2]

```
vx_status status = VX_SUCCESS;
void *base_ptr = NULL;
vx_uint32 width = 640, height = 480, plane = 0;
vx_image image = vxCreateImage(context, width, height, VX_DF_IMAGE_U8);
vx_rectangle_t rect;
vx_imagepatch_addressing_t addr;

rect.start_x = rect.start_y = 0;
rect.end_x = rect.end_y = PATCH_DIM;

status = vxAccessImagePatch(image, &rect, plane,&addr,
                            &base_ptr, VX_READ_AND_WRITE);
if (status == VX_SUCCESS)
{
    vx_uint32 x,y,i,j;
    vx_uint8 pixel = 0;

    /* addressing options */

    /* use linear addressing function/macro */
    for (i = 0; i < addr.dim_x*addr.dim_y; i++) {
        vx_uint8 *ptr2 = vxFormatImagePatchAddress1d(base_ptr, i, &addr);
        *ptr2 = pixel;
    }

    /* 2d addressing option */
    for (y = 0; y < addr.dim_y; y+=addr.step_y) {
        for (x = 0; x < addr.dim_x; x+=addr.step_x) {
            vx_uint8 *ptr2 = vxFormatImagePatchAddress2d(base_ptr, x, y, &addr);
            *ptr2 = pixel;
        }
    }

    /* direct addressing by client. for subsampled planes, scale will change. */
    for (y = 0; y < addr.dim_y; y+=addr.step_y) {
        j = (addr.stride_y*y*addr.scale_y)/VX_SCALE_UNITY;
        for (x = 0; x < addr.dim_x; x+=addr.step_x) {
            vx_uint8 *tmp = (vx_uint8 *)base_ptr;
            i = j + (addr.stride_x*x*addr.scale_x) / VX_SCALE_UNITY;
            tmp[i] = pixel;
        }
    }
    /* commits the data back to the image. If rect were 0 or empty, it would just decrement
     *  the reference (used when reading an image only).
     */
    status = vxCommitImagePatch(image, &rect, plane, &addr, base_ptr);
}
vxReleaseImage(&image);
```

OpenVX Types (vx_type_e) [3.45.5]

VX_TYPE_INVALID	Invalid type value.
VX_TYPE_CHAR	vx_char
VX_TYPE_INT8	vx_int8
VX_TYPE_UINT8	vx_uint8
VX_TYPE_INT16	vx_int16
VX_TYPE_UINT16	vx_uint16
VX_TYPE_INT32	vx_int32
VX_TYPE_UINT32	vx_uint32
VX_TYPE_INT64	vx_int64
VX_TYPE_UINT64	vx_uint64
VX_TYPE_FLOAT32	vx_float32
VX_TYPE_FLOAT64	vx_float64
VX_TYPE_ENUM	vx_enum
VX_TYPE_SIZE	vx_size
VX_TYPE_DF_IMAGE	vx_df_image
VX_TYPE_BOOL	vx_bool
VX_TYPE_SCALAR_MAX	Floating value for comparison between OpenVX scalars and OpenVX structs.
VX_TYPE_RECTANGLE	vx_rectangle_t
VX_TYPE_KEYPOINT	vx_keypoint_t
VX_TYPE_COORDINATES2D	vx_coordinates2d_t
VX_TYPE_COORDINATES3D	vx_coordinates3d_t
VX_TYPE_USER_STRUCT_START	User-defined struct base index.
VX_TYPE_VENDOR_STRUCT_START	Vendor-defined struct base index.
VX_TYPE_KHRONOS_OBJECT_START	Khronos-defined object base index.
VX_TYPE_VENDOR_OBJECT_START	Vendor-defined object base index.
VX_TYPE_KHRONOS_STRUCT_MAX	Value for comparison between Khronos-defined structs and user structs.
VX_TYPE_USER_STRUCT_END	Value for comparison between user structs and vendor structs.
VX_TYPE_VENDOR_STRUCT_END	Value for comparison between vendor structs and Khronos-defined objects.
VX_TYPE_KHRONOS_OBJECT_END	Value for comparison between Khronos-defined objects and vendor structs.
VX_TYPE_VENDOR_OBJECT_END	Value used for bound checking of vendor objects.
VX_TYPE_REFERENCE	vx_reference
VX_TYPE_CONTEXT	vx_context
VX_TYPE_GRAPH	vx_graph
VX_TYPE_NODE	vx_node
VX_TYPE_KERNEL	vx_kernel
VX_TYPE_PARAMETER	vx_parameter
VX_TYPE_DELAY	vx_delay
VX_TYPE_LUT	vx_lut
VX_TYPE_DISTRIBUTION	vx_distribution
VX_TYPE_PYRAMID	vx_pyramid
VX_TYPE_THRESHOLD	vx_threshold
VX_TYPE_MATRIX	vx_matrix
VX_TYPE_CONVOLUTION	vx_convolution
VX_TYPE_SCALAR	vx_scalar
VX_TYPE_ARRAY	vx_array
VX_TYPE_IMAGE	vx_image
VX_TYPE_REMAP	vx_remap
VX_TYPE_ERROR	Error object which has no type.
VX_TYPE_META_FORMAT	vx_meta_format
VX_TYPE_OBJECT_ARRAY	vx_object_array