The **OpenVX™** Specification

Version 1.1

Document Revision: 2b213f9
Generated on Tue Sep 20 2016 15:46:13

Khronos Vision Working Group

Editor: Susheel Gautam

Contents

Chapter 1

Introduction

1.1 Abstract

OpenVX is a low-level programming framework domain to enable software developers to efficiently access computer vision hardware acceleration with both functional and performance portability. OpenVX has been designed to support modern hardware architectures, such as mobile and embedded SoCs as well as desktop systems. Many of these systems are parallel and heterogeneous: containing multiple processor types including multi-core CPUs, DSP subsystems, GPUs, dedicated vision computing fabrics as well as hardwired functionality. Additionally, vision system memory hierarchies can often be complex, distributed, and not fully coherent. OpenVX is designed to maximize functional and performance portability across these diverse hardware platforms, providing a computer vision framework that efficiently addresses current and future hardware architectures with minimal impact on applications.

OpenVX contains:

- a library of predefined and customizable vision functions,
- a graph-based execution model to combine function enabling both task and data-independent execution, and;
- a set of memory objects that abstract the physical memory.

OpenVX defines a C Application Programming Interface (API) for building, verifying, and coordinating graph execution, as well as for accessing memory objects. The graph abstraction enables OpenVX implementers to optimize the execution of the graph for the underlying acceleration architecture.

OpenVX also defines the `vxu` utility library, which exposes each OpenVX predefined function as a directly callable C function, without the need for first creating a graph. Applications built using the `vxu` library do not benefit from the optimizations enabled by graphs; however, the vxu library can be useful as the simplest way to use OpenVX and as first step in porting existing vision applications.

As the computer vision domain is still rapidly evolving, OpenVX provides an extensibility mechanism to enable developer-defined functions to be added to the application graph.

1.2 Purpose

The purpose of this document is to detail the Application Programming Interface (API) for OpenVX.

1.3 Scope of Specification

The document contains the definition of the OpenVX API. The conformance tests that are used to determine whether an implementation is consistent to this specification are defined separately.

1.4 Normative References

The section “Module Documentation” forms the normative part of the specification. Each API definition provided in that chapter has certain preconditions and post conditions specified that are normative. If these normative conditions are not met, the behavior of the function is undefined.

1.5 Version/Change History

- OpenVX 1.0 Provisional - November, 2013
- OpenVX 1.0 Provisional V2 - June, 2014
- OpenVX 1.0 - September 2014
- OpenVX 1.0.1 - April 2015
- OpenVX 1.1 - May 2016

1.6 Deprecation

Certain items that are deprecated through the evolution of this specification document are removed from it. However, to provide a backward compatibility for such items for a certain time period these items are made available via a compatibility header file available with the release of this specification document (vx_compatibility.h). The items listed in this compatibility header file are temporary only and are removed permanently when the backward compatibility is no longer supported for those items.

1.7 Requirements Language

In this specification, the words *shall* or *must* express a requirement that is binding, *should* expresses design goals or recommended actions, and *may* expresses an allowed behavior.

1.8 Typographical Conventions

The following typographical conventions are used in this specification.

- **Bold** words indicate warnings or strongly communicated concepts that are intended to draw attention to the text.
- `Monospace` words signify an API element (i.e., class, function, structure) or a filename.
- *Italics* denote an emphasis on a particular concept, an abstraction of a concept, or signify an argument, parameter, or member.
- Throughout this specification, code examples given to highlight a particular issue use the format as shown below:
-
```
/* Example Code Section */
int main(int argc, char *argv[])
{
   return 0;
}
```
- Some "mscgen" message diagrams are included in this specification. The graphical conventions for this tool can be found on its website.

  See Also

  `http://www.mcternan.me.uk/mscgen/`

1.8.1 Naming Conventions

The following naming conventions are used in this specification.

- Opaque objects and atomics are named as `vx_`*`object`*, e.g., `vx_image` or `vx_uint8`, with an underscore separating the object name from the "vx" prefix.
- Defined Structures are named as `vx_`*`struct`*`_t`, e.g., `vx_imagepatch_addressing_t`, with underscores separating the structure from the "vx" prefix and a "t" to denote that it is a structure.

- Defined Enumerations are named as `vx_enum_e`, e.g., `vx_type_e`, with underscores separating the enumeration from the "vx" prefix and an "e" to denote that it is an enumerated value.
- Application Programming Interfaces are named `vxsomeFunction()` using camel case, starting with lower-case, and no underscores, e.g., `vxCreateContext()`.
- Vision functions also have a naming convention that follows a lower-case, inverse dotted hierarchy similar to Java Packages, e.g.,

  ```
  "org.khronos.openvx.color_convert".
  ```

  This minimizes the possibility of name collisions and promotes sorting and readability when querying the namespace of available vision functions. Each vision function should have a unique dotted name of the style: *tld.vendor.library.function*. The hierarchy of such vision function namespaces is undefined outside the subdomain "org.khronos", but they do follow existing international standards. For OpenVX-specified vision functions, the "function" section of the unique name does not use camel case and uses underscores to separate words.

1.9 Glossary and Acronyms

- Atomic: The specification mentions *atomics*, which means a C primitive data type. Usages that have additional wording, such as *atomic operations* do not carry this meaning.
- API: Application Programming Interface that specifies how a software component interacts with another.
- Framework: A generic software abstraction in which users can override behaviors to produce application-specific functionality.
- Engine: A purpose-specific software abstraction that is tunable by users.
- Run-time: The execution phase of a program.
- Kernel: OpenVX uses the term *kernel* to mean an abstract *computer vision function*, not an Operating System kernel. Kernel may also refer to a set of convolution coefficients in some computer vision literature (e.g., the Sobel "kernel"). OpenVX does not use this meaning. OpenCL uses kernel (specifically `cl_kernel`) to qualify a function written in "CL" which the OpenCL may invoke directly. This is close to the meaning OpenVX uses; however, OpenVX does not define a language.

1.10 Acknowledgements

This specification would not be possible without the contributions from this partial list of the following individuals from the Khronos Working Group and the companies that they represented at the time:

- Erik Rainey - Amazon
- Susheel Gautam - QUALCOMM
- Victor Erukhimov - Itseez
- Doug Knisely - QUALCOMM
- Frank Brill - Samsung
- Kari Pulli - NVIDIA
- Thierry Lepley - NVIDIA
- Neil Trevett - NVIDIA
- Tomer Schwartz - Intel
- Shervin Emami - NVIDIA

- Olivier Pothier - STMicroelectronics International NV
- Andy Kuzma - Intel
- Mostafa Hagog - Intel
- Shorin Kyo - Huawei
- Renato Grottesi - ARM Limited
- Dave Schreiner - ARM Limited
- Chris Tseng - Texas Instruments, Inc.
- Daniel Laroche - NXP Semiconductors
- Andrew Garrard - Samsung Electronics
- Tomer Yanir - Samsung Electronics
- Erez Natan - Samsung Electronics
- Chang-Hyo Yu - Samsung Electronics
- Hans-Peter Nilsson - Axis Communications
- Stephen Neuendorffer - Xilinx, Inc.
- Amit Shoham - BDTi
- Paul Buxton - Imagination Technologies
- Yuki Kobayashi - Renesas Electronics
- Cormac Brick - Movidius Ltd
- Mikael Bourges-Sevenier - Aptina Imaging Corporation
- Tao Zhang - QUALCOMM
- Jesse Villareal - Texas Instruments, Inc.
- Vadim Pisarevsky - Itseez
- Andrey Kamaev - Itseez
- Vlad Vinogradov - Itseez
- Roman Donchenko - Itseez
- Alexander Alekhin - Itseez
- Radha Giduthuri - AMD
- Xin Wang - Vivante Corporation
- Anshu Arya - MulticoreWare
- Steve Ramm - Imagination Technologies

Chapter 2

Design Overview

2.1 Software Landscape

OpenVX is intended to be used either directly by applications or as the acceleration layer for higher-level vision frameworks, engines or platform APIs.

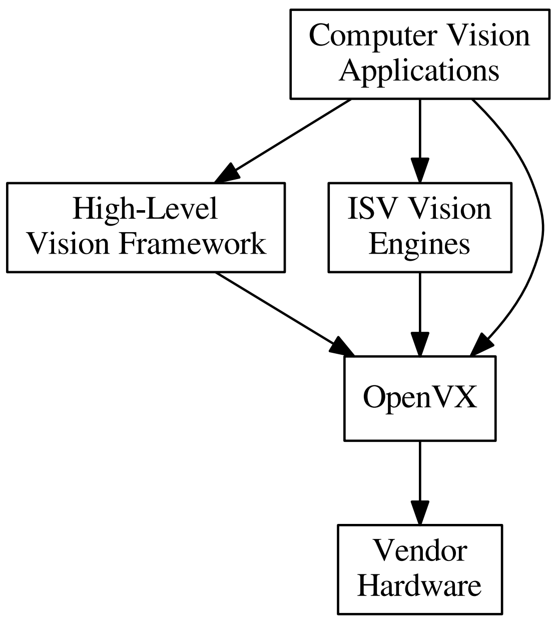


Figure 2.1: OpenVX Usage Overview

2.2 Design Objectives

OpenVX is designed as a framework of standardized computer vision functions able to run on a wide variety of platforms and potentially to be accelerated by a vendor's implementation on that platform. OpenVX can improve the performance and efficiency of vision applications by providing an abstraction for commonly-used vision functions and an abstraction for aggregations of functions (a "graph"), thereby providing the implementer the opportunity to minimize the run-time overhead.

The functions in OpenVX are intended to cover common functionality required by many vision applications.

2.2.1 Hardware Optimizations

This specification makes no statements as to which acceleration methodology or techniques may be used in its implementation. Vendors may choose any number of implementation methods such as parallelism and/or specialized hardware offload techniques.

This specification also makes no statement or requirements on a "level of performance" as this may vary significantly across platforms and use cases.

2.2.2 Hardware Limitations

The OpenVX focuses on vision functions that can be significantly accelerated by diverse hardware. Future versions of this specification may adopt additional vision functions into the core standard when hardware acceleration for those functions becomes practical.

2.3 Assumptions

2.3.1 Portability

OpenVX has been designed to maximize functional and performance portability wherever possible, while recognizing that the API is intended to be used on a wide diversity of devices with specific constraints and properties. Tradeoffs are made for portability where possible: for example, portable Graphs constructed using this API should work on any OpenVX implementation and return similar results within the precision bounds defined by the OpenVX conformance tests.

2.3.2 Opaqueness

OpenVX is intended to address a very broad range of devices and platforms, from deeply embedded systems to desktop machines and distributed computing architectures. The OpenVX API addresses this range of possible implementations without forcing hardware-specific requirements onto any particular implementation via the use of *opaque* objects for most program data.

All data, except client-facing structures, are opaque and hidden behind a reference that may be as thin or thick as an implementation needs. Each implementation provides the standardized interfaces for accessing data that takes care of specialized hardware, platform, or allocation requirements. Memory that is *imported* or *shared* from other APIs is not subsumed by OpenVX and is still maintained and accessible by the originator.

OpenVX does not dictate any requirements on memory allocation methods or the layout of opaque memory objects and it does not dictate byte packing or alignment for structures on architectures.

2.4 Object-Oriented Behaviors

OpenVX objects are both strongly typed at compile-time for safety critical applications and are strongly typed at run-time for dynamic applications. Each object has its typedef'd type and its associated enumerated value in the `vx_type_e` list. Any object may be down-cast to a `vx_reference` safely to be used in functions that require this, specifically `vxQueryReference`, which can be used to get the `vx_type_e` value using an `vx_enum`.

2.5 OpenVX Framework Objects

This specification defines the following OpenVX framework objects.

- Object: Context - The OpenVX context is the object domain for all OpenVX objects. All data objects *live* in the context as well as all framework objects. The OpenVX context keeps reference counts on all objects and must do garbage collection during its deconstruction to free lost references. While multiple clients may connect to the OpenVX context, all data are private in that the references that refer to data objects are given only to the creating party. The results of calling an OpenVX function on data objects created in different contexts are undefined.

- Object: Kernel - A Kernel in OpenVX is the abstract representation of a computer vision function, such as a “Sobel Gradient” or “Lucas Kanade Feature Tracking”. A vision function may implement many similar or identical features from other functions, but it is still considered a single, unique kernel as long as it is named by the same string and enumeration and conforms to the results specified by OpenVX. Kernels are similar to function signatures in this regard.
- Object: Parameter - An abstract input, output, or bidirectional data object passed to a computer vision function. This object contains the signature of that parameter’s usage from the kernel description. This information includes:
  - *Signature Index* - The numbered index of the parameter in the signature.
  - *Object Type* - e.g. `VX_TYPE_IMAGE`, or `VX_TYPE_ARRAY`, or some other object type from `vx_type_e`.
  - *Usage Model* - e.g. `VX_INPUT`, `VX_OUTPUT`, or `VX_BIDIRECTIONAL`.
  - *Presence State* - e.g. `VX_PARAMETER_STATE_REQUIRED`, or `VX_PARAMETER_STATE_OPTIONAL`.
- Object: Node - A node is an instance of a kernel that will be paired with a specific set of references (the parameters). Nodes are created from and associated with a single graph only. When a `vx_parameter` is extracted from a Node, an additional attribute can be accessed:
  - *Reference* - The `vx_reference` assigned to this parameter index from the Node creation function (e.g., `vxSobel3x3Node`).
- Object: Graph - A set of nodes connected in a directed (only goes one-way) acyclic (does not loop back) fashion. A Graph may have sets of Nodes that are unconnected to other sets of Nodes within the same Graph. See Graph Formalisms.

2.6 OpenVX Data Objects

Data objects are object that are processed by graphs in nodes.

- Object: Array An opaque array object that could be an array of primitive data types or an array of structures.
- Object: Convolution An opaque object that contains MxN matrix of `vx_int16` values. Also contains a scaling factor for normalization. Used specifically with `vxuConvolve` and `vxConvolveNode`.
- Object: Delay An opaque object that contains a manually controlled, temporally-delayed list of objects.
- Object: Distribution An opaque object that contains a frequency distribution (e.g., a histogram).
- Object: Image An opaque image object that may be some format in `vx_df_image_e`.
- Object: LUT An opaque lookup table object used with `vxTableLookupNode` and `vxuTableLookup`.
- Object: Matrix An opaque object that contains MxN matrix of some scalar values.
- Object: Pyramid An opaque object that contains multiple levels of scaled `vx_image` objects.
- Object: Remap An opaque object that contains the map of source points to destination points used to transform images.
- Object: Scalar An opaque object that contains a single primitive data type.
- Object: Threshold An opaque object that contains the thresholding configuration.
- Object: ObjectArray An opaque array object that could be an array of any data-object (not data-type) of OpenVX except Delay and ObjectArray objects.

2.7 Error Objects

Error objects are specialized objects that may be returned from other object creator functions when serious platform issue occur (i.e., out of memory or out of handles). These can be checked at the time of creation of these objects, but checking also may be put-off until usage in other APIs or verification time, in which case, the implementation must return appropriate errors to indicate that an invalid object type was used.

```
vx_<object> obj = vxCreate<Object>(context, ...);
vx_status status = vxGetStatus((vx_reference)obj);
if (status == VX_SUCCESS) {
    // object is good
}
```

2.8 Graphs Concepts

The *graph* is the central computation concept of OpenVX. The purpose of using graphs to express the Computer Vision problem is to allow for the possibility of any implementation to maximize its optimization potential because all the operations of the graph and its dependencies are known ahead of time, before the graph is processed.

Graphs are composed of one or more *nodes* that are added to the graph through node creation functions. Graphs in OpenVX must be created ahead of processing time and verified by the implementation, after which they can be processed as many times as needed.

2.8.1 Linking Nodes

Graph Nodes are linked together via data dependencies with *no explicitly-stated ordering*. The same reference may be linked to other nodes. Linking has a limitation, however, in that only one node in a graph may output to any specific data object reference. That is, only a single writer of an object may exist in a given graph. This prevents indeterminate ordering from data dependencies. All writers in a graph shall produce output data before any reader of that data accesses it.

2.8.2 Virtual Data Objects

Graphs in OpenVX depend on data objects to link together nodes. When clients of OpenVX know that they do not need access to these *intermediate* data objects, they may be created as `virtual`. Virtual data objects can be used in the same manner as non-virtual data objects to link nodes of a graph together; however, virtual data objects are different in the following respects.

- Inaccessible - No calls to an Map/Unmap or Copy APIs shall succeed given a reference to an object created through a virtual create function from a Graph external perspective. Calls to Map/Unmap or Copy APIs from within client-defined node that belongs to the same graph as the virtual object will succeed as they are Graph internal.

- Scoped - Virtual data objects are scoped within the Graph in which they are created; they cannot be shared outside their scope. The live range of the data content of a virtual data object is limited to a single graph execution. In other word, data content of a virtual object is undefined before graph execution and no data of a virtual object should be expected to be preserved across successive graph executions by the application.

- Intermediates - Virtual data objects should be used only for intermediate operations within Graphs, because they are fundamentally inaccessible to clients of the API.

- Dimensionless or Formatless - Virtual data objects may have dimensions and formats partially or fully undefined at creation time. For instance, a virtual image can be created with undefined or partially defined dimensions (0x0, Nx0 or 0xN where N is not null) and/or without defined format (VX_DF_IMAGE_VIRT). The undefined property of the virtual object at creation time is undefined with regard to the graph and mutable at graph verification time; it will be automatically adjusted at each graph verification, deduced from the node that outputs the virtual object. Dimensions and format properties that are well defined at virtual object creation time are immutable and can't be adjusted automatically at graph verification time. The Dimensionless or Formatless aspect of virtual data is a commodity that allows creating graphs generic with regard to dimensions or format, but there are restrictions:

1. Nodes may require the dimensions and/or the format to be defined for a virtual output object when it can't be deduced from its other parameters. For example, a Scale node requires well defined dimensions for the output image, while ColorConvert and ChannelCombine nodes require a well defined format for the output image.
2. An image created from ROI must always be well defined (vx_rectangle_t parameter) and can't be created from a dimensionless virtual image.
3. A ROI of a formatless virtual image shouldn't be a node output.
4. Levels of a dimensionless or formatless virtual pyramid shouldn't be a node output.

- Inheritance - A sub-object inherits from the virtual property of its parent. A sub-object also inherits from the Dimensionless or Formatless property of its parent with restrictions:

  1. it is adjusted automatically at graph verification when the parent properties are adjusted (the parent is the output of a node)
  2. it can't be adjusted at graph verification when the sub-object is itself the output of a node.

- Optimizations - Virtual data objects do not have to be created during Graph validation and execution and therefore may be of zero *size*.

These restrictions enable vendors the ability to optimize some aspects of the data object or its usage. Some vendors may not allocate such objects, some may create intermediate sub-objects of the object, and some may allocate the object on remote, inaccessible memories. OpenVX does not proscribe *which* optimization the vendor does, merely that it *may* happen.

2.8.3 Node Parameters

Parameters to node creation functions are defined as either atomic types, such as `vx_int32`, `vx_enum`, or as objects, such as `vx_scalar`, `vx_image`. The atomic variables of the Node creation functions shall be converted by the framework into `vx_scalar` references for use by the Nodes. A node parameter of type `vx_scalar` can be changed during the graph execution; whereas, a node parameter of an atomic type (`vx_int32` etc.) require at least a graph revalidation if changed. All node parameter objects may be modified by retrieving the reference to the `vx_parameter` via `vxGetParameterByIndex`, and then passing that to `vxQueryParameter` to retrieve the reference to the object.

```
vx_parameter param = vxGetParameterByIndex(node, p);
vx_reference ref;
vxQueryParameter(param, VX_PARAMETER_REF, &ref, sizeof(ref));
```

If the type of the parameter is unknown, it may be retrieved with the same function.

```
    vx_enum type;
    vxQueryParameter(param, VX_PARAMETER_TYPE, &type, sizeof(type)
);
    /* cast the ref to the correct vx_<type>. Atomics are now vx_scalar */
```

2.8.4 Graph Parameters

Parameters may exist on Graphs, as well. These parameters are defined by the author of the Graph and each Graph parameter is defined as a specific parameter from a Node within the Graph using `vxAddParameterToGraph`. Graph parameters communicate to the implementation that there are specific Node parameters that may be modified by the client between Graph executions. Additionally, they are parameters that the client may set without the reference to the Node but with the reference to the Graph using `vxSetGraphParameterByIndex`. This allows for the Graph authors to construct *Graph Factories*. How these factories work falls outside the scope of this document.
See Also

Framework: Graph Parameters

2.8.5 Execution Model

Graphs must execute in both:

- *Synchronous blocking mode* (in that `vxProcessGraph` will block until the graph has completed), and in
- *Asynchronous single-issue-per-reference mode* (via `vxScheduleGraph` and `vxWaitGraph`).

Asynchronous Mode

In asynchronous mode, Graphs must be single-issue-per-reference. This means that given a constructed graph reference G, it may be scheduled multiple times but only executes sequentially with respect to itself. Multiple graphs references given to the asynchronous graph interface do not have a defined behavior and may execute in parallel or in series based on the behavior or the vendor's implementation.

2.8.6 Graph Formalisms

To use graphs several rules must be put in place to allow deterministic execution of Graphs. The behavior of a `processGraph(` G`)` call is determined by the structure of the Processing Graph G. The Processing Graph is a bipartite graph consisting of a set of Nodes $N_1 \dots N_n$ and a set of data objects $d_1 \dots d_i$. Each edge (N_x, D_y) in the graph represents a data object D_y that is written by Node N_x and each edge (D_x, N_y) represents a data object D_x that is read by Node N_y. Each edge e has a name `Name(` e`)`, which gives the parameter name of the node that references the corresponding data object. Each Node Parameter also has a type `Type(node, name)` in `{INPUT, OUTPUT, INOUT}`. Some data objects are *Virtual*, and some data objects are *Delay*. Delay data objects are just collections of data objects with indexing (like an image list) and known linking points in a graph. A node may be classified as a *head node*, which has no backward dependency. Alternatively, a node may be a *dependent node*, which has a backward dependency to the head node. In addition, the Processing Graph has several restrictions:

1. *Output typing* - Every output edge (N_x, D_y) requires `Type(` N_x, `Name(` N_x, D_y)) in `{OUTPUT, INOUT}`
2. *Input typing* - Every input edge (N_x, D_y) requires `Type(` N_y, `Name(` D_x, N_y)) in `{INPUT}` or `{INOUT}`
3. *Single Writer* - Every data object is the target of at most one output edge.
4. *Broken Cycles* - Every cycle in G must contain at least input edge (D_x, N_y) where D_x is Delay.
5. *Virtual images must have a source* - If D_y is Virtual, then there is at least one output edge that writes D_y (N_x, D_y)
6. *Bidirectional data objects shall not be virtual* - If `Type(` N_x, `Name(` N_x, D_y)) is INOUT implies D_y is non-Virtual.
7. *Delay data objects shall not be virtual* - If D_x is Delay then it shall not be Virtual.
8. *A uniform image cannot be output or bidirectional*.

The execution of each node in a graph consists of an atomic operation (sometimes referred to as *firing*) that consumes data representing each input data object, processes it, and produces data representing each output data object. A node may execute when all of its input edges are marked *present*. Before the graph executes, the following initial marking is used:

- All input edges (D_x, N_y) from non-Virtual objects Dx are marked (parameters must be set).
- All input edges (D_x, N_y) with an output edge (N_z, D_x) are unmarked.
- All input edges (D_x, N_y) where D_x is a Delay data object are marked.

Processing a node results in unmarking all the corresponding input edges and marking all its output edges; marking an output edge (N_x, D_y) where D_y is not a Delay results in marking all of the input edges (D_y, N_z). Following these rules, it is possible to statically schedule the nodes in a graph as follows: Construct a precedence graph P, including all the nodes $N_1 \dots N_x$, and an edge (N_x, N_z) for every pair of edges (N_x, D_y) and (D_y, N_z) where D_y is not a Delay. Then unconditionally fire each node according to any topological sort of P.

The following assertions should be verified:

- P is a Directed Acyclic Graph (DAG), implied by 4 and the way it is constructed.
- Every data object has a value when it is executed, implied by 5, 6, 7, and the marking.
- Execution is deterministic if the nodes are deterministic, implied by 3, 4, and the marking.
- Every node completes its execution exactly once.

The execution model described here just acts as a formalism. For example, independent processing is allowed across multiple depended and depending nodes and edges, provided that the result is invariant with the execution model described here.

Contained & Overlapping Data Objects

There are cases in which two different data objects referenced by an output parameter of node N_1 and input parameter of node N_2 in a graph induce a dependency between these two nodes: For example, a pyramid and its level images, image and the sub-images created from it by `vxCreateImageFromROI`, or overlapping sub-images of the same image. Following figure show examples of this dependency. To simplify subsequent definitions and requirements a limitation is imposed that if a sub-image I' has been created from image *I* and sub-image I'' has been created from I', then I'' is still considered a sub-image of *I* and not of I'. In these cases it is expected that although the two nodes reference two different data objects, any change to one data object might be reflected in the other one. Therefore it implies that N_1 comes before N_2 in the graph's topological order. To ensure that, following definitions are introduced.

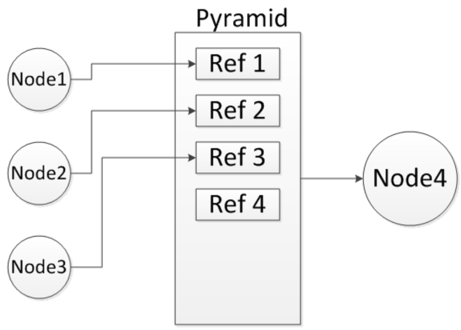


Figure 2.2: Pyramid Example

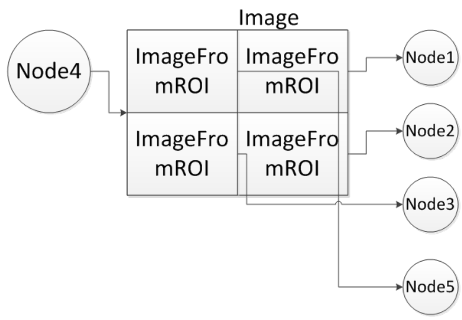


Figure 2.3: Image Example

1. *Containment Set* - *C(d)*, the set of recursively contained data objects of *d*, named *Containment Set*, is defined as follows:
   - C_0(*d*)={*d*}
   - C_1(*d*) is the set of all data objects that are *directly contained* by *d*:
     (a) If *d* is an image, all images created as an ROI of *d* are directly contained by *d*.
     (b) If *d* is a pyramid, all pyramid levels of *d* are directly contained by *d*.
     (c) If *d* is an object array, all elements of *d* are directly contained by *d*.
     (d) If *d* is a delay object, all slots of *d* are directly contained by *d*.
   - For i>1, C_i(*d*) is the set of all data objects that are contained by *d* at the i^{th} order

$$C_i(d) = \bigcup_{d' \in C_{i-1}(d)} C_1(d') \tag{2.1}$$

   - C(*d*) is the set that contains *d* itself, the data objects *contained* by *d*, the data objects that are contained by the data objects contained by *d* and so on. Formally:

$$C(d) = \bigcup_{i=0}^{\infty} C_i(d) \tag{2.2}$$

2. *I(d)* is a predicate that equals true if and only if *d* is an image.

3. *Overlapping Relationship* - The overlapping relation R_{ov} is a relation defined for images, such that if i_1 and i_2 in *C(i)*, *i* being an image, then i_1 R_{ov} i_2 is true if and only if i_1 and i_2 overlap, i.e there exists a point (x,y) of *i* that is contained in both i_1 and i_2 . Note that this relation is reflexive and symmetric, but not transitive: i_1 overlaps i_2 and i_2 overlaps i_3 does not necessarily imply that i_1 overlaps i_3, as illustrated in the following figure:

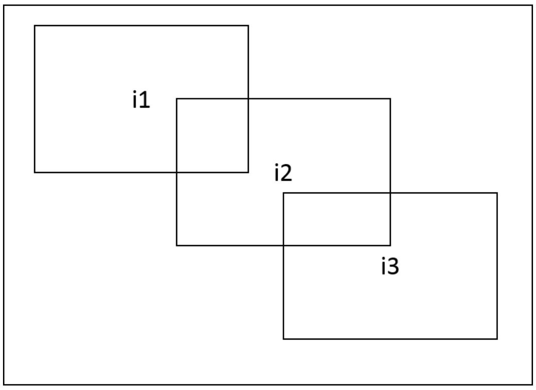


Figure 2.4: Overlap Example

4. *Dependency Relationship* - The dependency relationship N_1 -> N_2, is a relation defined for nodes. N_1 -> N_2 means that N_2 depends on N_1 and then implies that N_2 must be executed after the completion of N_1.

5. N_1 -> N_2 if N_1 writes to a data object d_1 and N_2 reads from a data object d_2 and:

$$d_1 \in C(d_2) \ or \ d_2 \in C(d_1) \ or \ (I(d_1) \ and \ I(d_2) \ and \ d_1 R_{ov} d_2) \tag{2.3}$$

2.8.7 Node Execution Independence

In the following example a client computes the gradient magnitude and gradient phase from a blurred input image. The `vxMagnitudeNode` and `vxPhaseNode` are *independently* computed, in that each does not depend on the output of the other. OpenVX does not mandate that they are run simultaneously or in parallel, but it could be implemented this way by the OpenVX vendor.

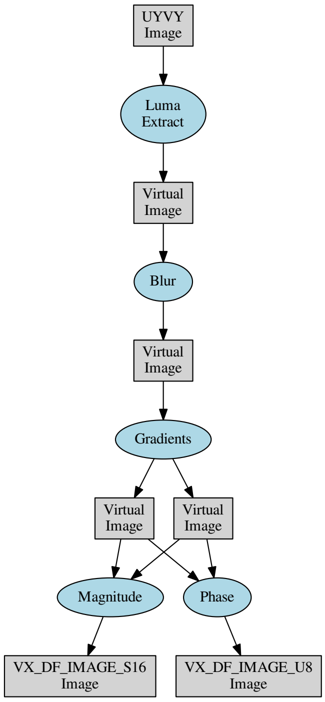


Figure 2.5: A simple graph with some independent nodes.

The code to construct such a graph can be seen below.

```
vx_context context = vxCreateContext();
vx_image images[] = {
        vxCreateImage(context, 640, 480, VX_DF_IMAGE_UYVY),
        vxCreateImage(context, 640, 480, VX_DF_IMAGE_S16),
        vxCreateImage(context, 640, 480, VX_DF_IMAGE_U8),
};
vx_graph graph = vxCreateGraph(context);
vx_image virts[] = {
        vxCreateVirtualImage(graph, 0, 0,
  VX_DF_IMAGE_VIRT),
        vxCreateVirtualImage(graph, 0, 0,
  VX_DF_IMAGE_VIRT),
        vxCreateVirtualImage(graph, 0, 0,
  VX_DF_IMAGE_VIRT),
        vxCreateVirtualImage(graph, 0, 0,
  VX_DF_IMAGE_VIRT),
};

vxChannelExtractNode(graph, images[0], VX_CHANNEL_Y, virts[0]),
vxGaussian3x3Node(graph, virts[0], virts[1]),
vxSobel3x3Node(graph, virts[1], virts[2], virts[3]),
vxMagnitudeNode(graph, virts[2], virts[3], images[1]),
vxPhaseNode(graph, virts[2], virts[3], images[2]),

status = vxVerifyGraph(graph);
if (status == VX_SUCCESS)
{
    status = vxProcessGraph(graph);
}
vxReleaseContext(&context); /* this will release everything */
```

2.8.8 Verification

Graphs within OpenVX must go through a rigorous validation process before execution to satisfy the design concept of eliminating run-time overhead (parameter checking) that guarantees safe execution of the graph. OpenVX must check for (but is not limited to) these conditions:

- Parameters To Nodes:
  - Each required parameter is given to the node (vx_parameter_state_e). Optional parameters may not be present and therefore are not checked when absent. If present, they are checked.
  - Each parameter given to a node must be of the right *direction* (a value from `vx_direction_e`).
  - Each parameter given to a node must be of the right *object type* (from the object range of `vx_type_e`).
  - Each parameter attribute or value must be verified. In the case of a scalar value, it may need to be range checked (e.g., $0.5 <= k <= 1.0$). The implementation is not required to do run-time range checking of scalar values. If the value of the scalar changes at run time to go outside the range, the results are undefined. The rationale is that the potential performance hit for run-time range checking is too large to be enforced. It will still be checked at graph verification time as a time-zero sanity check. If the scalar is an output parameter of another node, it must be initialized to a legal value. In the case of `vxScaleImageNode`, the relation of the input image dimensions to the output image dimensions determines the scaling factor. These values or attributes of data objects must be checked for compatibility on each platform.
  - Graph Connectivity - the `vx_graph` must be a Directed Acyclic Graph (DAG). No cycles or feedback is allowed. The `vx_delay` object has been designed to explicitly address feedback between Graph executions.
  - Resolution of Virtual Data Objects - Any changes to *Virtual* data objects from unspecified to specific format or dimensions, as well as the related creation of objects of specific type that are observable at processing time, takes place at Verification time.

2.9 Callbacks

Callbacks are a method to control graph flow and to make decisions based on completed work. The `vxAssignNodeCallback` call takes as a parameter a callback function. This function will be called after the execution of the particular node, but prior to the completion of the graph. If nodes are arranged into independent sets, the order of the callbacks is unspecified. Nodes that are arranged in a serial fashion due to data dependencies perform callbacks in order. The callback function may use the node reference first to extract parameters from the node, and then extract the data references. Data outputs of Nodes with callbacks shall be available (via Map/Unmap/Copy methods) when the callback is called.

2.10 User Kernels

OpenVX supports the concept of *client-defined functions* that shall be executed as *Nodes* from inside the Graph or are Graph *internal*. The purpose of this paradigm is to:

- Further exploit independent operation of nodes within the OpenVX platform.
- Allow componentized functions to be reused elsewhere in OpenVX.
- Formalize strict verification requirements (i.e., Contract Programming).

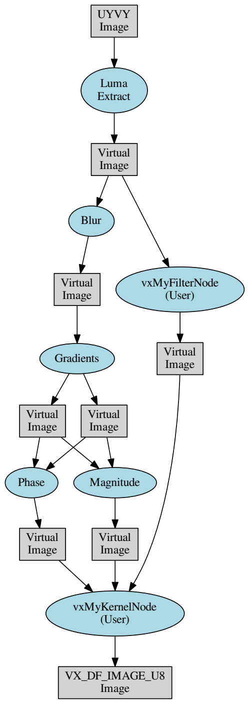


Figure 2.6: A graph with User Kernel nodes which are independent of the “base” nodes.

In this example, to execute client-supplied functions, the graph does not have to be halted and then resumed. These nodes shall be executed in an independent fashion with respect to independent base nodes within OpenVX. This allows implementations to further minimize execution time if hardware to exploit this property exists.

2.10.1 Parameter Validation

User Kernels must aid in the Graph Verification effort by providing an explicit validation function for each vision function they implement. Each parameter passed to the instanced Node of a User Kernel is validated using the client-supplied validation function. The client must check these attributes and/or values of each parameter:

- Each attribute or value of the parameter must be checked. For example, the size of array, or the value of a scalar to be within a range, or a dimensionality constraint of an image such as width divisibility. (Some implementations may have restrictions, such as an image width be evenly divisible by some fixed number).
- If the output parameters depend on attributes or values from input parameters, those relationships must be checked.

The Meta Format Object

The Meta Format Object is an opaque object used to collect requirements about the output parameter, which then the OpenVX implementation will check. The Client must manually set relevant object attributes to be checked against output parameters, such as dimensionality, format, scaling, etc.

2.10.2 User Kernels Naming Conventions

User Kernels must be exported with a unique name (see Naming Conventions for information on OpenVX conventions) and a unique enumeration. Clients of OpenVX may use either the name or enumeration to retrieve a kernel, so collisions due to non-unique names will cause problems. The kernel enumerations may be extended by following this example:

```
#define VX_KERNEL_NAME_KHR_XYZ "org.khronos.example.xyz"

#define VX_LIBRARY_XYZ (0x3) // assigned from Khronos, vendors control their own

enum vx_kernel_xyz_ext_e {
    VX_KERNEL_KHR_XYZ = VX_KERNEL_BASE(VX_ID_DEFAULT, VX_LIBRARY_XYZ) + 0x0,
    // up to 0xFFF kernel enums can be created.
};
```

Each vendor of a vision function or an implementation must apply to Khronos to get a unique identifier (up to a limit of $2^{12}-1$ vendors). Until they obtain a unique ID vendors must use VX_ID_DEFAULT.

To construct a kernel enumeration, a vendor must have both their ID and a *library* ID. The library ID's are completely *vendor* defined (however when using the VX_ID_DEFAULT ID, many libraries may collide in namespace).

Once both are defined, a kernel enumeration may be constructed using the VX_KERNEL_BASE macro and an offset. (The offset is optional, but very helpful for long enumerations.)

2.11 Immediate Mode Functions

OpenVX also contains an interface defined within `<VX/vxu.h>` that allows for immediate execution of vision functions. These interfaces are prefixed with `vxu` to distinguish them from the Node interfaces, which are of the form `vx<Name>Node`. Each of these interfaces replicates a Node interface with some exceptions. Immediate mode functions are defined to *behave* as *Single Node Graphs*, which have no leaking side-effects (e.g., no Log entries) within the Graph Framework after the function returns. The following tables refer to both the Immediate Mode and Graph Mode vision functions. The Module documentation for each vision function draws a distinction on each API by noting that it is either an immediate mode function with the tag `[Immediate]` or it is a Graph mode function by the tag `[Graph]`.

2.12 Targets

A 'Target' specifies a physical or logical devices where a node or an immediate mode function is executed. This allows the use of different implementations of vision functions on different targets. The existence of allowed Targets is exposed to the applications by the use of defined APIs. The choice of a Target allows for different levels of control on where the nodes can be executed. An OpenVX implementation must support at least one target. Additional supported targets are specified using the appropriate enumerations. See `vxSetNodeTarget`, `vxSetImmediateModeTarget`, and `vx_target_e`. An OpenVX implementation must support at least one target VX_TARGET_ANY as well as VX_TARGET_STRING enumerates. An OpenVX implementation may also support more than these two to indicate the use of specific devices. For example, an implementation may add VX_TARGET_CPU and VX_TARGET_GPU enumerates to indicate the support of two possible targets to assign a nodes to (or to excute an immediate mode function). Another way an implementation can indicate the existence of multiple targets, for example CPU and GPU, is by specifying the target as VX_TARGET_STRING and using strings 'CPU' and 'GPU'. Thus defining targets using names rather than enumerates. The specific naming of string or enumerates is not enforced by the specification and it is up to the vendors to document and communicate the Target naming. Once available in a given implementation Applications can assign a Target to a node to specify the target that must execute that node by using the API `vxSetNodeTarget`. For immediate mode functions the target specifies the physical or logical device where the future execution of that function will be attempted. When an immediate mode function is not supported on the selected target the execution falls back to VX_TARGET_ANY.

2.13 Base Vision Functions

OpenVX comes with a standard or *base* set of vision functions. The following table lists the supported set of vision functions, their input types (first table) and output types (second table), and the version of OpenVX in which they are supported.

2.13.1 Inputs

Vision Function	**U8**	**U16**	**S16**	**U32**	**S32**	**F32**	**color**
AbsDiff	1.0		1.0.1				
Accumulate	1.0						
AccumulateSquared	1.0						
AccumulateWeighted	1.0						
Add	1.0		1.0				
And	1.0						
Box3x3	1.0						
CannyEdgeDetector	1.0						
ChannelCombine	1.0						
ChannelExtract							1.0
ColorConvert							1.0
ConvertDepth	1.0		1.0				
Convolve	1.0						
Dilate3x3	1.0						
EqualizeHistogram	1.0						
Erode3x3	1.0						
FastCorners	1.0						
Gaussian3x3	1.0						
HarrisCorners	1.0						
HalfScaleGaussian	1.0						
Histogram	1.0						
IntegralImage	1.0						
TableLookup	1.0		1.1				
LaplacianPyramid	1.1						
LaplacianReconstruct			1.1				
Magnitude			1.0				
MeanStdDev	1.0						

Median3x3	1.0						
MinMax-Loc	1.0		1.0				
Multiply	1.0		1.0				
Non-Linear-Filter	1.1						
Not	1.0						
Optical-FlowPyrLK	1.0						
Or	1.0						
Phase			1.0				
Gaussian-Pyramid	1.0						
Remap	1.0						
Scale-Image	1.0						
Sobel3x3	1.0						
Subtract	1.0		1.0				
Threshold	1.0						
WarpAffine	1.0						
Warp-Perspective	1.0						
Xor	1.0						

2.13.2 Outputs

Vision Function	U8	U16	S16	U32	S32	F32	color
AbsDiff	1.0		1.0.1				
Accumu-late			1.0				
Accumulate-Squared			1.0				
Accumulate-Weighted	1.0						
Add	1.0		1.0				
And	1.0						
Box3x3	1.0						
Canny-Edge-Detector	1.0						
Channel-Combine							1.0
Channel-Extract	1.0						
Color-Convert							1.0
Convert-Depth	1.0		1.0				
Convolve	1.0		1.0				
Dilate3x3	1.0						

EqualizeHistogram	1.0						
Erode3x3	1.0						
FastCorners	1.0						
Gaussian3x3	1.0						
HarrisCorners	1.0						
HalfScaleGaussian	1.0						
Histogram				1.0			
IntegralImage				1.0			
TableLookup	1.0		1.1				
LaplacianPyramid			1.1				
LaplacianReconstruct	1.1						
Magnitude			1.0				
MeanStdDev						1.0	
Median3x3	1.0						
MinMaxLoc	1.0		1.0	1.0			
Multiply	1.0		1.0				
NonLinearFilter	1.1						
Not	1.0						
OpticalFlowPyrLK							
Or	1.0						
Phase	1.0						
GaussianPyramid	1.0						
Remap	1.0						
ScaleImage	1.0						
Sobel3x3			1.0				
Subtract	1.0		1.0				
Threshold	1.0						
WarpAffine	1.0						
WarpPerspective	1.0						
Xor	1.0						

2.14 Lifecycles

2.14.1 OpenVX Context Lifecycle

The lifecycle of the context is very simple.

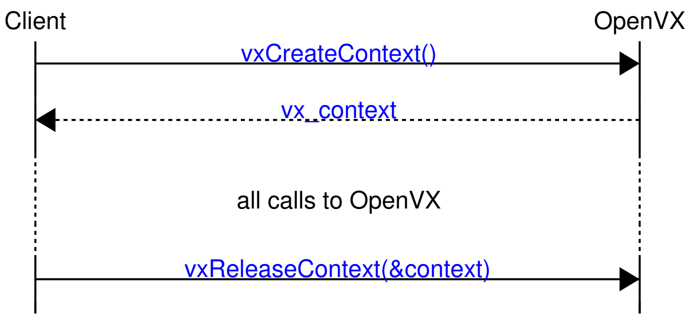


Figure 2.7: The lifecycle model for an OpenVX Context.

2.14.2 Graph Lifecycle

OpenVX has four main phases of graph lifecycle:

- Construction - Graphs are created via `vxCreateGraph`, and Nodes are connected together by data objects.
- Verification - The graphs are checked for consistency, correctness, and other conditions. Memory allocation may occur.
- Execution - The graphs are executed via `vxProcessGraph` or `vxScheduleGraph`. Between executions data may be updated by the client or some other external mechanism. The client of OpenVX may change reference of input data to a graph, but this may require the graph to be validated again by checking `vxIsGraphVerified`.
- Deconstruction - Graphs are released via `vxReleaseGraph`. All Nodes in the Graph are released.

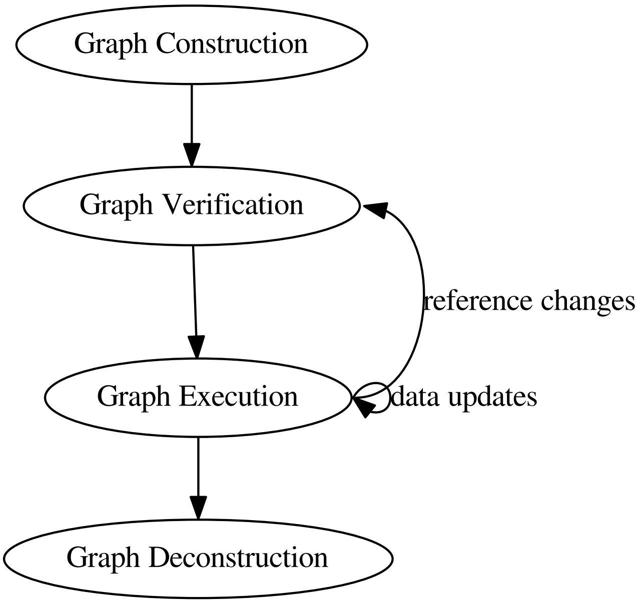


Figure 2.8: Graph Lifecycle

2.14.3 Data Object Lifecycle

All objects in OpenVX follow a similar lifecycle model. All objects are

- Created via `vxCreate<Object><Method>` or retreived via `vxGet<Object><Method>` from the parent object if they are internally created.
- Used within Graphs or immediate functions as needed.
- Then objects must be released via `vxRelease<Object>` or via `vxReleaseContext` when all objects are released.

OpenVX Image Lifecycle

This is an example of the Image Lifecycle using the OpenVX Framework API. This would also apply to other data types with changes to the types and function names.

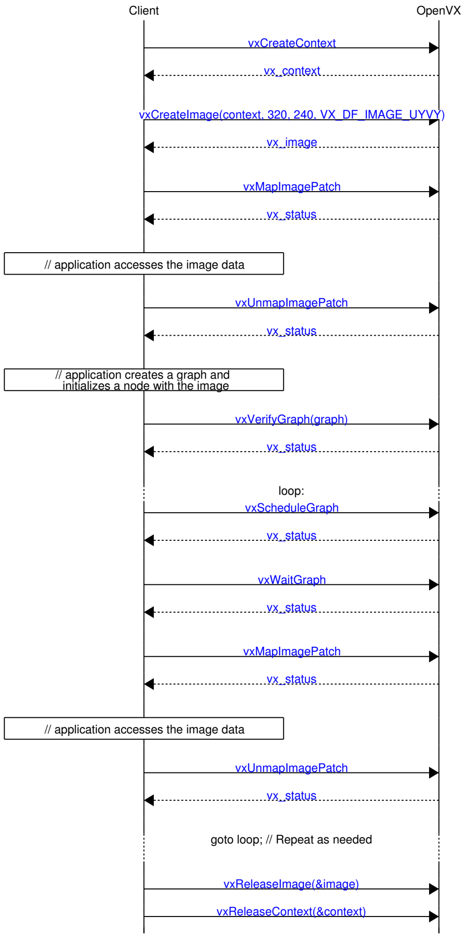


Figure 2.9: Image Object Lifecycle

2.15 Host Memory Data Object Access Patterns

For objects retrieved from OpenVX that are 2D in nature, such as `vx_image`, `vx_matrix`, and `vx_convolution`, the manner in which the host-side has access to these memory regions is well-defined. OpenVX uses a row-major storage (that is each unit in a column is memory-adjacent to its row adjacent unit). Two-dimensional objects are always created (using `vxCreateImage` or `vxCreateMatrix`) in width (columns) by height (rows) notation, with the arguments in that order. When accessing these structures in “C” with two-dimensional arrays of declared size, the user must therefore provide the array dimensions in the reverse of the order of the arguments to the Create function. This layout ensures *row-wise* storage in C on the host. A pointer could also be allocated for the matrix data and would have to be indexed in this row-major method.

2.15.1 Matrix Access Example

```
    const vx_size columns = 3;
    const vx_size rows = 4;
    vx_matrix matrix = vxCreateMatrix(context,
      VX_TYPE_FLOAT32, columns, rows);
    vx_status status = vxGetStatus((vx_reference)matrix);
    if (status == VX_SUCCESS)
    {
        vx_int32 j, i;
#if defined(OPENVX_USE_C99)
        vx_float32 mat[rows][columns]; /* note: row major */
#else
        vx_float32 *mat = (vx_float32 *)malloc(rows*columns*sizeof(
      vx_float32));
#endif
        if (vxCopyMatrix(matrix, mat, VX_READ_ONLY,
      VX_MEMORY_TYPE_HOST) == VX_SUCCESS) {
            for (j = 0; j < (vx_int32)rows; j++)
```

```
                for (i = 0; i < (vx_int32)columns; i++)
#if defined(OPENVX_USE_C99)
                    mat[j][i] = (vx_float32)rand()/(vx_float32)RAND_MAX;
#else
                    mat[j*columns + i] = (vx_float32)rand()/(
      vx_float32)RAND_MAX;
#endif
            vxCopyMatrix(matrix, mat, VX_WRITE_ONLY,
      VX_MEMORY_TYPE_HOST);
        }
#if !defined(OPENVX_USE_C99)
        free(mat);
#endif
    }
```

2.15.2 Image Access Example

Images and Array differ slightly in how they are accessed due to more complex memory layout requirements.

```
    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;
    vx_map_id map_id;

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

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

        /* a couple 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) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = ((addr.stride_y*y*addr.scale_y) /
                      VX_SCALE_UNITY) +
                    ((addr.stride_x*x*addr.scale_x) /
                      VX_SCALE_UNITY);
                tmp[i] = pixel;
            }
        }

        /* more efficient 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;
            }
        }
```

```
        /* this commits the data back to the image.
         */
        status = vxUnmapImagePatch(image, map_id);
    }
    vxReleaseImage(&image);
```

2.15.3 Array Access Example

Arrays only require a single value, the stride, instead of the entire addressing structure that images need.

```
        vx_size i, stride = sizeof(vx_size);
        void *base = NULL;
        vx_map_id map_id;
        /* access entire array at once */
        vxMapArrayRange(array, 0, num_items, &map_id, &stride, &base,
     VX_READ_AND_WRITE, VX_MEMORY_TYPE_HOST, 0);
        for (i = 0; i < num_items; i++)
        {
            vxArrayItem(mystruct, base, i, stride).some_uint += i;
            vxArrayItem(mystruct, base, i, stride).some_double = 3.14f;
        }
        vxUnmapArrayRange(array, map_id);
```

Map/Unmap pairs can also be called on individual elements of array using a method similar to this:

```
        /* access each array item individually */
        for (i = 0; i < num_items; i++)
        {
            mystruct *myptr = NULL;
            vxMapArrayRange(array, i, i+1, &map_id, &stride, (void **)&myptr,
     VX_READ_AND_WRITE, VX_MEMORY_TYPE_HOST, 0);
            myptr->some_uint += 1;
            myptr->some_double = 3.14f;
            vxUnmapArrayRange(array, map_id);
        }
```

2.16 Concurrent Data Object Access

Accessing OpenVX data-objects using the functions Map, Copy, Read concurrently to an execution of a graph that is accessing the same data objects is permitted only if all accesses are read-only. That is, for Map, Copy to have a read-only access mode and for nodes in the graph to have that data-object as an input parameter only. In all other cases, including write or read-write modes and Write access function, as well as a graph nodes having the data-object as output or bidirectional, the application must guarantee that the access is not performed concurrently with the graph execution. That can be achieved by calling un-map following a map and commit following access before calling `vxScheduleGraph` or `vxProcessGraph`. In addition, the application must call `vxWaitGraph` after `vxScheduleGraph` before calling Map, Read, Write or Copy to avoid restricted concurrent access. An application that fails to follow the above might encounter an undefined behavior and/or data loss without being notified by the OpenVX framework. Accessing images created from ROI (`vxCreateImageFromROI`) must be treated in this respect as if the entire image is being accessed.

- Setting an attribute is considered as writing to a data object in this respect.
- For concurrent execution of several graphs please see Execution Model
- Also see the graph formalism section for guidance on accessing ROIs of the same image within a graph.

2.17 Valid Image Region

The valid region mechanism informs the application as to which pixels of the output images of a graph's execution have valid values (see valid pixel definition below). The mechanism also applies to immediate mode (VXU) calls, and supports the communication of the valid region between different graph executions. Some vision functions, mainly those providing statistics and summarization of image information, use the valid region to ignore pixels that are not valid on their inputs (potentially bad or unstable pixel values). A good example of such a function is Min/Max Location. Formalization of the valid region mechanism is given below.

- Valid Pixels - All output pixels of an OpenVX function are considered valid by default, unless their calculation depends on input pixels that are not valid. An input pixel is not valid in one of two situations:

1. The pixel is outside of the image border and the border mode in use is VX_BORDER_UNDEFINED
2. The pixel is outside the valid region of the input image.

- Valid Region - The region in the image that contains all the valid pixels. Theoretically this can be of any shape. OpenVX currently only supports rectangular valid regions. In subsequent text the term 'valid rectangle' denotes a valid region that is rectangular in shape.
- Valid Rectangle Reset - In some cases it is not possible to calculate a valid rectangle for the output image of a vision function (for example, warps and remap). In such cases, the vision function is said to reset the valid Region to the entire image. The attribute VX_NODE_VALID_RECT_RESET is a read only attribute and is used to communicate valid rectangle reset behavior to the application. When it is set to vx_true_e for a given node the valid rectangle of the output images will reset to the full image upon execution of the node, when it is set to vx_false_e the valid rectangle will be calculated. All standard OpenVX functions will have this attribute set to vx_false_e by default, except for Warp and Remap where it will be set to vx_true_e.
- Valid Rectangle Initialization - Upon the creation of an image, its valid rectangle is the entire image. One exception to this is when creating an image via vxCreateImageFromROI; in that case, the valid region of the ROI image is the subset of the valid region of the parent image that is within the ROI. In other words, the valid region of an image created using an ROI is the largest rectangle that contains valid pixels in the parent image.
- Valid Rectangle Calculation - The valid rectangle of an image changes as part of the graph execution, the correct value is guaranteed only when the execution finishes. The valid rectangle of an image remains unchanged between graph executions and persists between graph executions as long as the application doesn't explicitly change the valid region via vxSetImageValidRectangle. Notice that using vxMapImagePatch, vxUnmapImagePatch or vxSwapImageHandle does not change the valid region of an image.
- Valid Rectangle for Immediate mode (VXU) - VXU is considered a single node graph execution, thus the valid rectangle of an output of VXU will be propagated for an input to a consequent VXU call (when using the same output image from one call as input to the consecutive call).
- Valid Region Usage - For all standard OpenVX functions, the framework must guarantee that all pixel values inside the valid rectangle of the output images are valid. The framework does not guarantee that input pixels outside of the valid rectangle are processed. For the following vision functions, the framework guarantees that pixels outside of the valid rectangle do not participate in calculating the vision function result: Equalize Histogram, Integral Image, Fast Corners, Histogram, Mean and Standard Deviation, Min Max Location, Optical Flow Pyramid (LK) and Canny Edge Detector. An application can get the valid rectangle of an image by using vxGetValidRegionImage.
- User kernels - User kernels may change the valid rectangles of their output images. To change the valid rectangle, the programmer of the user kernel must provide a call-back function that sets the valid rectangle. The output validator of the user kernel must provide this callback by setting the value of the vx_meta_format attribute VX_VALID_RECT_CALLBACK during the output validator. The callback function must be callable by the OpenVX framework during graph validation and execution. Assumptions must not be made regarding the order and the frequency by which the valid rectangle callback is called. The framework will recalculate the valid region when a change in the input valid regions is detected. For user nodes, the default value of VX_NODE_VALID_RECT_RESET is vx_true_e. Setting VX_VALID_RECT_CALLBACK during parameter validation to a value other than NULL will result in setting VX_NODE_VALID_RECT_RESET to vx_false_e. Note: the above means that when VX_VALID_RECT_CALLBACK is not set or set to NULL the user-node will reset the valid rectangle to the entire image.
- In addition, valid rectangle reset occurs in the following scenarios:
  1. A reset of the valid rectangle of a parent image when a node writes to one of its ROIs. The only case where the reset does not occur is when the child ROI image is identical to the parent image.
  2. For nodes that have the VX_NODE_VALID_RECT_RESET set to vx_true_e

2.18 Extending OpenVX

Beyond User Kernels there are other mechanisms for vendors to extend features in OpenVX. These mechanisms are not available to User Kernels. Each OpenVX official extension has a unique identifier, comprised of capital letters, numbers and the underscore character, prefixed with "KHR_", for example "KHR_NEW_FEATURE".

2.18.1 Extending Attributes

When extending attributes, vendors *must* use their assigned ID from `vx_vendor_id_e` in conjunction with the appropriate macros for creating new attributes with `VX_ATTRIBUTE_BASE`. The typical mechanism to extend a new attribute for some object type (for example a `vx_node` attribute from `VX_ID_TI`) would look like this:

```
enum {
   VX_NODE_TI_NEWTHING = VX_ATTRIBUTE_BASE(VX_ID_TI,
      VX_TYPE_NODE) + 0x0,
}
```

2.18.2 Vendor Custom Kernels

Vendors wanting to add more kernels to the base set supplied to OpenVX should provide a header of the form

```
#include <VX/vx_ext_<vendor>.h>
```

that contains definitions of each of the following.

- New Node Creation Function Prototype per function.

```
vx_node vxXYZNode(vx_graph graph, vx_image input,
      vx_uint32 value, vx_image output, vx_array temp);
```

- A new Kernel Enumeration(s) and Kernel String per function.

```
#define VX_KERNEL_NAME_KHR_XYZ "org.khronos.example.xyz"

#define VX_LIBRARY_XYZ (0x3) // assigned from Khronos, vendors control their own

enum vx_kernel_xyz_ext_e {
    VX_KERNEL_KHR_XYZ = VX_KERNEL_BASE(VX_ID_DEFAULT, VX_LIBRARY_XYZ) + 0x0,
    // up to 0xFFF kernel enums can be created.
};
```

- A new VXU Function per function.

```
vx_status vxuXYZ(vx_context context, vx_image input,
      vx_uint32 value, vx_image output, vx_array temp);
```

  This should come with good documentation for each new part of the extension. Ideally, these sorts of extensions should not require linking to new objects to facilitate usage.

2.18.3 Vendor Custom Extensions

Some extensions affect *base* vision functions and thus may be invisible to most users. In these circumstances, the vendor must report the supported extensions to the base nodes through the `VX_CONTEXT_EXTENSIONS` attribute on the context.

```
vx_char *tmp, *extensions = NULL;
vx_size size = 0;
vxQueryContext(context,VX_CONTEXT_EXTENSIONS_SIZE,&size,sizeof(
  size));
extensions = malloc(size);
vxQueryContext(context,VX_CONTEXT_EXTENSIONS,
               extensions, size);
```

Extensions in this list are dependent on the extension itself; they may or may not have a header and new kernels or framework feature or data objects. The common feature is that they are implemented and supported by the implementation vendor.

2.18.4 Hinting

The specification defines a Hinting API that allows Clients to feed information to the implementation for *optional* behavior changes. See Framework: Hints. It is assumed that most of the hints will be vendor- or implementation-specific. Check with the OpenVX implementation vendor for information on vendor-specific extensions.

2.18.5 Directives

The specification defines a Directive API to control implementation behavior. See Framework: Directives. This *may* allow things like disabling parallelism for debugging, enabling cache writing-through for some buffers, or any implementation-specific optimization.

2.19 Known Extensions to OpenVX

2.19.1 User Kernel Tiling

The User Kernel Tiling facility enables optimizations of the user kernels (e.g., locality of execution or parallelism) when performing computation on the image data. Modern processors have a diverse memory hierarchy that varies from relatively small but fast and expensive memory to relatively large but slow and inexpensive memory. Image data are typically too large to fit into the fast but small memory. The ability to break the image data into smaller sized units allows for optimized computation on these smaller units with fast memory access or parallel execution of a user kernel on multiple image tiles simultaneously. The OpenVX Graph Manager possesses the knowledge about the memory hierarchy of the platform and is hence in a position to break the image data into smaller units for memory optimization. Knowledge of the memory access pattern of an algorithm is key for the graph manager to enable optimizations.

The Khronos OpenVX Working Group will include this extension as part of the future version of this specification, contingent on community feedback.

Chapter 3

Module Documentation

3.1 Vision Functions

3.1.1 Detailed Description

These are the base vision functions supported in OpenVX 1.1. These functions were chosen as a subset of a larger pool of possible functions that fall under the following criteria:

- Applicable to Acceleration Hardware
- Very Common Usage
- Encumbrance Free

Modules

- Absolute Difference

  Computes the absolute difference between two images.
- Accumulate

  Accumulates an input image into output image.
- Accumulate Squared

  Accumulates a squared value from an input image to an output image.
- Accumulate Weighted

  Accumulates a weighted value from an input image to an output image.
- Arithmetic Addition

  Performs addition between two images.
- Arithmetic Subtraction

  Performs subtraction between two images.
- Bitwise AND

  Performs a bitwise AND operation between two `VX_DF_IMAGE_U8` *images.*
- Bitwise EXCLUSIVE OR

  Performs a bitwise EXCLUSIVE OR (XOR) operation between two `VX_DF_IMAGE_U8` *images.*
- Bitwise INCLUSIVE OR

  Performs a bitwise INCLUSIVE OR operation between two `VX_DF_IMAGE_U8` *images.*
- Bitwise NOT

  Performs a bitwise NOT operation on a `VX_DF_IMAGE_U8` *input image.*
- Box Filter

  Computes a Box filter over a window of the input image.
- Canny Edge Detector

  Provides a Canny edge detector kernel.
- Channel Combine

  Implements the Channel Combine Kernel.

- Channel Extract

  Implements the Channel Extraction Kernel.
- Color Convert

  Implements the Color Conversion Kernel.
- Convert Bit depth

  Converts image bit depth.
- Custom Convolution

  Convolves the input with the client supplied convolution matrix.
- Dilate Image

  Implements Dilation, which grows the white space in a `VX_DF_IMAGE_U8` Boolean image.
- Equalize Histogram

  Equalizes the histogram of a grayscale image.
- Erode Image

  Implements Erosion, which shrinks the white space in a `VX_DF_IMAGE_U8` Boolean image.
- Fast Corners

  Computes the corners in an image using a method based upon FAST9 algorithm suggested in [3] and with some updates from [4] with modifications described below.
- Gaussian Filter

  Computes a Gaussian filter over a window of the input image.
- Non Linear Filter

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

  Computes the Harris Corners of an image.
- Histogram

  Generates a distribution from an image.
- Gaussian Image Pyramid

  Computes a Gaussian Image Pyramid from an input image.
- Laplacian Image Pyramid

  Computes a Laplacian Image Pyramid from an input image.
- Reconstruction from a Laplacian Image Pyramid

  Reconstructs the original image from a Laplacian Image Pyramid.
- Integral Image

  Computes the integral image of the input.
- Magnitude

  Implements the Gradient Magnitude Computation Kernel.
- Mean and Standard Deviation

  Computes the mean pixel value and the standard deviation of the pixels in the input image (which has a dimension width and height).
- Median Filter

  Computes a median pixel value over a window of the input image.
- Min, Max Location

  Finds the minimum and maximum values in an image and a location for each.
- Optical Flow Pyramid (LK)

  Computes the optical flow using the Lucas-Kanade method between two pyramid images.
- Phase

  Implements the Gradient Phase Computation Kernel.
- Pixel-wise Multiplication

  Performs element-wise multiplication between two images and a scalar value.
- Remap

  Maps output pixels in an image from input pixels in an image.
- Scale Image

Implements the Image Resizing Kernel.

- Sobel 3x3

  Implements the Sobel Image Filter Kernel.

- TableLookup

  Implements the Table Lookup Image Kernel.

- Thresholding

  Thresholds an input image and produces an output Boolean image.

- Warp Affine

  Performs an affine transform on an image.

- Warp Perspective

  Performs a perspective transform on an image.

3.2 Absolute Difference

3.2.1 Detailed Description

Computes the absolute difference between two images. Absolute Difference is computed by:

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

The output image can be VX_DF_IMAGE_U8 only if both source images are VX_DF_IMAGE_U8 and the output image is explicitly set to VX_DF_IMAGE_U8. When the two input parameters have type s16, the conceptual definition describing the overflow is:
uint16 uresult = (uint16) abs((int32) (a) - (int32) (b));
int16 result = uresult > 32767 ? 32767 : (int16) uresult;

Functions

- vx_node VX_API_CALL vxAbsDiffNode (vx_graph graph, vx_image in1, vx_image in2, vx_image out)

  [Graph] Creates an AbsDiff node.
- vx_status VX_API_CALL vxuAbsDiff (vx_context context, vx_image in1, vx_image in2, vx_image out)

  [Immediate] Computes the absolute difference between two images.

3.2.2 Function Documentation

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

[Graph] Creates an AbsDiff node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	An input image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.
in	*in2*	An input image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.
out	*out*	The output image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Computes the absolute difference between two images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	An input image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.
in	*in2*	An input image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.
out	*out*	The output image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.3 Accumulate

3.3.1 Detailed Description

Accumulates an input image into output image. Accumulation is computed by:

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

The overflow policy used is `VX_CONVERT_POLICY_SATURATE`.

Functions

- vx_node VX_API_CALL vxAccumulateImageNode (vx_graph graph, vx_image input, vx_image accum)

  [Graph] Creates an accumulate node.
- vx_status VX_API_CALL vxuAccumulateImage (vx_context context, vx_image input, vx_image accum)

  [Immediate] Computes an accumulation.

3.3.2 Function Documentation

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

[Graph] Creates an accumulate node.

Parameters

`in`	*graph*	The reference to the graph.
`in`	*input*	The input `VX_DF_IMAGE_U8` image.
`in,out`	*accum*	The accumulation image in `VX_DF_IMAGE_S16`.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes an accumulation.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*input*	The input `VX_DF_IMAGE_U8` image.
`in,out`	*accum*	The accumulation image in `VX_DF_IMAGE_S16`

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
$*$	An error occurred. See `vx_status_e`.

3.4 Accumulate Squared

3.4.1 Detailed Description

Accumulates a squared value from an input image to an output image. Accumulate squares is computed by:

$$accum(x,y) = saturate_{int16}((uint16)accum(x,y) + (((uint16)(input(x,y)^2)) >> (shift)))$$

Where $0 \leq shift \leq 15$
The overflow policy used is VX_CONVERT_POLICY_SATURATE.

Functions

- vx_node VX_API_CALL vxAccumulateSquareImageNode (vx_graph graph, vx_image input, vx_scalar shift, vx_image accum)

  [Graph] Creates an accumulate square node.
- vx_status VX_API_CALL vxuAccumulateSquareImage (vx_context context, vx_image input, vx_scalar shift, vx_image accum)

  [Immediate] Computes a squared accumulation.

3.4.2 Function Documentation

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

[Graph] Creates an accumulate square node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*shift*	The input VX_TYPE_UINT32 with a value in the range of $0 \leq shift \leq 15$.
in,out	*accum*	The accumulation image in VX_DF_IMAGE_S16.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Computes a squared accumulation.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*shift*	A VX_TYPE_UINT32 type, the input value with the range $0 \leq shift \leq 15$.
in,out	*accum*	The accumulation image in VX_DF_IMAGE_S16

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.5 Accumulate Weighted

3.5.1 Detailed Description

Accumulates a weighted value from an input image to an output image. Weighted accumulation is computed by:

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

Where $0 \leq \alpha \leq 1$ Conceptually, the rounding for this is defined as:

$$output(x,y) = uint8((1-\alpha) * float32(int32(output(x,y))) + \alpha * float32(int32(input(x,y))))$$

Functions

- vx_node VX_API_CALL vxAccumulateWeightedImageNode (vx_graph graph, vx_image input, vx_scalar alpha, vx_image accum)

  [Graph] Creates a weighted accumulate node.
- vx_status VX_API_CALL vxuAccumulateWeightedImage (vx_context context, vx_image input, vx_scalar alpha, vx_image accum)

  [Immediate] Computes a weighted accumulation.

3.5.2 Function Documentation

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

[Graph] Creates a weighted accumulate node.

Parameters

`in`	*graph*	The reference to the graph.
`in`	*input*	The input `VX_DF_IMAGE_U8` image.
`in`	*alpha*	The input `VX_TYPE_FLOAT32` scalar value with a value in the range of $0.0 \leq \alpha \leq 1.0$.
`in,out`	*accum*	The `VX_DF_IMAGE_U8` accumulation image.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes a weighted accumulation.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*input*	The input `VX_DF_IMAGE_U8` image.
`in`	*alpha*	A `VX_TYPE_FLOAT32` type, the input value with the range $0.0 \leq \alpha \leq 1.0$.
`in,out`	*accum*	The `VX_DF_IMAGE_U8` accumulation image.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.6 Arithmetic Addition

3.6.1 Detailed Description

Performs addition between two images. Arithmetic addition is performed between the pixel values in two VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 images. The output image can be VX_DF_IMAGE_U8 only if both source images are VX_DF_IMAGE_U8 and the output image is explicitly set to VX_DF_IMAGE_U8. It is otherwise VX_DF_IMAGE_S16. If one of the input images is of type VX_DF_IMAGE_S16, all values are converted to VX_DF_IMAGE_S16. The overflow handling is controlled by an overflow-policy parameter. For each pixel value in the two input images:

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

Functions

- vx_node VX_API_CALL vxAddNode (vx_graph graph, vx_image in1, vx_image in2, vx_enum policy, vx_image out)

  [Graph] Creates an arithmetic addition node.
- vx_status VX_API_CALL vxuAdd (vx_context context, vx_image in1, vx_image in2, vx_enum policy, vx_image out)

  [Immediate] Performs arithmetic addition on pixel values in the input images.

3.6.2 Function Documentation

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

[Graph] Creates an arithmetic addition node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16.
in	*in2*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16.
in	*policy*	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
out	*out*	The output image, a VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 image.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Performs arithmetic addition on pixel values in the input images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image.
in	*in2*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image.
in	*policy*	A vx_convert_policy_e enumeration.

out	*out*	The output image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.7 Arithmetic Subtraction

3.7.1 Detailed Description

Performs subtraction between two images. Arithmetic subtraction is performed between the pixel values in two VX_DF_IMAGE_U8 or two VX_DF_IMAGE_S16 images. The output image can be VX_DF_IMAGE_U8 only if both source images are VX_DF_IMAGE_U8 and the output image is explicitly set to VX_DF_IMAGE_U8. It is otherwise VX_DF_IMAGE_S16. If one of the input images is of type VX_DF_IMAGE_S16, all values are converted to VX_DF_IMAGE_S16. The overflow handling is controlled by an overflow-policy parameter. For each pixel value in the two input images:

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

Functions

- vx_node VX_API_CALL vxSubtractNode (vx_graph graph, vx_image in1, vx_image in2, vx_enum policy, vx_image out)

  [Graph] Creates an arithmetic subtraction node.
- vx_status VX_API_CALL vxuSubtract (vx_context context, vx_image in1, vx_image in2, vx_enum policy, vx_image out)

  [Immediate] Performs arithmetic subtraction on pixel values in the input images.

3.7.2 Function Documentation

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

[Graph] Creates an arithmetic subtraction node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16, the minuend.
in	*in2*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16, the subtrahend.
in	*policy*	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
out	*out*	The output image, a VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 image.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Performs arithmetic subtraction on pixel values in the input images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image, the minuend.
in	*in2*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image, the subtrahend.

in	*policy*	A vx_convert_policy_e enumeration.
out	*out*	The output image in `VX_DF_IMAGE_U8` or `VX_DF_IMAGE_S16` format.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.8 Bitwise AND

3.8.1 Detailed Description

Performs a *bitwise AND* operation between two VX_DF_IMAGE_U8 images. Bitwise AND is computed by the following, for each bit in each pixel in the input images:

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

Or expressed as C code:

```
out(x,y) = in_1(x,y) & in_2(x,y)
```

Functions

- vx_node VX_API_CALL vxAndNode (vx_graph graph, vx_image in1, vx_image in2, vx_image out)

  [Graph] Creates a bitwise AND node.
- vx_status VX_API_CALL vxuAnd (vx_context context, vx_image in1, vx_image in2, vx_image out)

  [Immediate] Computes the bitwise and between two images.

3.8.2 Function Documentation

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

[Graph] Creates a bitwise AND node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	A VX_DF_IMAGE_U8 input image.
in	*in2*	A VX_DF_IMAGE_U8 input image.
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Computes the bitwise and between two images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 input image
in	*in2*	A VX_DF_IMAGE_U8 input image
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.9 Bitwise EXCLUSIVE OR

3.9.1 Detailed Description

Performs a *bitwise EXCLUSIVE OR* (XOR) operation between two `VX_DF_IMAGE_U8` images. Bitwise XOR is computed by the following, for each bit in each pixel in the input images:

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

Or expressed as C code:

```
out(x,y) = in_1(x,y) ^ in_2(x,y)
```

Functions

- vx_status VX_API_CALL vxuXor (vx_context context, vx_image in1, vx_image in2, vx_image out)

  [Immediate] Computes the bitwise exclusive-or between two images.
- vx_node VX_API_CALL vxXorNode (vx_graph graph, vx_image in1, vx_image in2, vx_image out)

  [Graph] Creates a bitwise EXCLUSIVE OR node.

3.9.2 Function Documentation

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

[Graph] Creates a bitwise EXCLUSIVE OR node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	A `VX_DF_IMAGE_U8` input image.
in	*in2*	A `VX_DF_IMAGE_U8` input image.
out	*out*	The `VX_DF_IMAGE_U8` output image.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes the bitwise exclusive-or between two images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A `VX_DF_IMAGE_U8` input image
in	*in2*	A `VX_DF_IMAGE_U8` input image
out	*out*	The `VX_DF_IMAGE_U8` output image.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.10 Bitwise INCLUSIVE OR

3.10.1 Detailed Description

Performs a *bitwise INCLUSIVE OR* operation between two VX_DF_IMAGE_U8 images. Bitwise INCLUSIVE OR is computed by the following, for each bit in each pixel in the input images:

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

Or expressed as C code:

```
out(x,y) = in_1(x,y) | in_2(x,y)
```

Functions

- vx_node VX_API_CALL vxOrNode (vx_graph graph, vx_image in1, vx_image in2, vx_image out)

  [Graph] Creates a bitwise INCLUSIVE OR node.
- vx_status VX_API_CALL vxuOr (vx_context context, vx_image in1, vx_image in2, vx_image out)

  [Immediate] Computes the bitwise inclusive-or between two images.

3.10.2 Function Documentation

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

[Graph] Creates a bitwise INCLUSIVE OR node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	A VX_DF_IMAGE_U8 input image.
in	*in2*	A VX_DF_IMAGE_U8 input image.
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Computes the bitwise inclusive-or between two images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 input image
in	*in2*	A VX_DF_IMAGE_U8 input image
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
$*$	An error occurred. See `vx_status_e`.

3.11 Bitwise NOT

3.11.1 Detailed Description

Performs a *bitwise NOT* operation on a VX_DF_IMAGE_U8 input image. Bitwise NOT is computed by the following, for each bit in each pixel in the input image:

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

Or expressed as C code:

```
out(x,y) = ~in_1(x,y)
```

Functions

- vx_node VX_API_CALL vxNotNode (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a bitwise NOT node.
- vx_status VX_API_CALL vxuNot (vx_context context, vx_image input, vx_image output)

  [Immediate] Computes the bitwise not of an image.

3.11.2 Function Documentation

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

[Graph] Creates a bitwise NOT node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	A VX_DF_IMAGE_U8 input image.
out	*output*	The VX_DF_IMAGE_U8 output image.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Computes the bitwise not of an image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The VX_DF_IMAGE_U8 input image
out	*output*	The VX_DF_IMAGE_U8 output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.12 Box Filter

3.12.1 Detailed Description

Computes a Box filter over a window of the input image. This filter uses the following convolution matrix:

$$\mathbf{K}_{box} = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{vmatrix} * \frac{1}{9}$$

Functions

- vx_node VX_API_CALL vxBox3x3Node (vx_graph graph, vx_image input, vx_image output)

    [Graph] Creates a Box Filter Node.
- vx_status VX_API_CALL vxuBox3x3 (vx_context context, vx_image input, vx_image output)

    [Immediate] Computes a box filter on the image by a 3x3 window.

3.12.2 Function Documentation

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

[Graph] Creates a Box Filter Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
out	*output*	The output image in `VX_DF_IMAGE_U8` format.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes a box filter on the image by a 3x3 window.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
out	*output*	The output image in `VX_DF_IMAGE_U8` format.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.13 Canny Edge Detector

3.13.1 Detailed Description

Provides a Canny edge detector kernel. This function implements an edge detection algorithm similar to that described in [2]. The main components of the algorithm are:

- Gradient magnitude and orientation computation using a noise resistant operator (Sobel).
- Non-maximum suppression of the gradient magnitude, using the gradient orientation information.
- Tracing edges in the modified gradient image using hysteresis thresholding to produce a binary result.

The details of each of these steps are described below.

- **Gradient Computation:** Conceptually, the input image is convolved with vertical and horizontal Sobel kernels of the size indicated by the *gradient_size* parameter. The Sobel kernels used for the gradient computation shall be as shown below. The two resulting directional gradient images (dx and dy) are then used to compute a gradient magnitude image and a gradient orientation image. The norm used to compute the gradient magnitude is indicated by the *norm_type* parameter, so the magnitude may be $|dx|+|dy|$ for VX_NORM_L1 or $\sqrt{dx^2+dy^2}$ for VX_NORM_L2. The gradient orientation image is quantized into 4 values: 0, 45, 90, and 135 degrees.
- For gradient size 3:

$$\mathbf{sobel}_x = \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}$$

$$\mathbf{sobel}_y = transpose(sobel_x) = \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix}$$

- For gradient size 5:

$$\mathbf{sobel}_x = \begin{vmatrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 4 \\ -1 & -2 & 0 & 2 & 1 \end{vmatrix}$$

$$\mathbf{sobel}_y = transpose(sobel_x)$$

- For gradient size 7:

$$\mathbf{sobel}_x = \begin{vmatrix} -1 & -4 & -5 & 0 & 5 & 4 & 1 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -15 & -60 & -75 & 0 & 75 & 60 & 15 \\ -20 & -80 & -100 & 0 & 100 & 80 & 20 \\ -15 & -60 & -75 & 0 & 75 & 60 & 15 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -1 & -4 & -5 & 0 & 5 & 4 & 1 \end{vmatrix}$$

$$\mathbf{sobel}_y = transpose(sobel_x)$$

- **Non-Maximum Suppression:** This is then applied such that a pixel is retained as a potential edge pixel if and only if its magnitude is greater than or equal to the pixels in the direction perpendicular to its edge orientation. For example, if the pixel's orientation is 0 degrees, it is only retained if its gradient magnitude is larger than that of the pixels at 90 and 270 degrees to it. If a pixel is suppressed via this condition, it must not appear as an edge pixel in the final output, i.e., its value must be 0 in the final output.
- **Edge Tracing:** The final edge pixels in the output are identified via a double thresholded hysteresis procedure. All retained pixels with magnitude above the *high* threshold are marked as known edge pixels (valued 255) in the final output image. All pixels with magnitudes less than or equal to the *low* threshold must not be marked as edge pixels in the final output. For the pixels in between the thresholds, edges are traced and marked as edges (255) in the output. This can be done by starting at the known edge pixels and moving in all eight directions recursively until the gradient magnitude is less than or equal to the low threshold.

- **Caveats:** The intermediate results described above are conceptual only; so for example, the implementation may not actually construct the gradient images and non-maximum-suppressed images. Only the final binary (0 or 255 valued) output image must be computed so that it matches the result of a final image constructed as described above.

Enumerations

- enum vx_norm_type_e {
VX_NORM_L1 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NORM_TYPE << 12)) + 0x0,
VX_NORM_L2 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NORM_TYPE << 12)) + 0x1 }

  A normalization type.

Functions

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

  [Graph] Creates a Canny Edge Detection Node.
- vx_status VX_API_CALL vxuCannyEdgeDetector (vx_context context, vx_image input, vx_threshold hyst, vx_int32 gradient_size, vx_enum norm_type, vx_image output)

  [Immediate] Computes Canny Edges on the input image into the output image.

3.13.2 Enumeration Type Documentation

enum vx_norm_type_e

A normalization type.

See Also

Canny Edge Detector

Enumerator

VX_NORM_L1 The L1 normalization.

VX_NORM_L2 The L2 normalization.

Definition at line 1293 of file vx_types.h.

3.13.3 Function Documentation

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

[Graph] Creates a Canny Edge Detection Node.

Parameters

`in`	*graph*	The reference to the graph.
`in`	*input*	The input `VX_DF_IMAGE_U8` image.
`in`	*hyst*	The double threshold for hysteresis. The threshold data_type shall be either `VX_TYPE_UINT8` or `VX_TYPE_INT16`. The `VX_THRESHOLD_TRUE_VALUE` and `VX_THRESHOLD_FALSE_VALUE` of `vx_threshold` are ignored.

in	*gradient_size*	The size of the Sobel filter window, must support at least 3, 5, and 7.
in	*norm_type*	A flag indicating the norm used to compute the gradient, VX_NORM_L1 or VX_NORM_L2.
out	*output*	The output image in VX_DF_IMAGE_U8 format with values either 0 or 255.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Computes Canny Edges on the input image into the output image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*hyst*	The double threshold for hysteresis. The threshold data_type shall be either VX_TYPE_UINT8 or VX_TYPE_INT16. The VX_THRESHOLD_TRUE_VALUE and VX_THRESHOLD_FALSE_VALUE of vx_threshold are ignored.
in	*gradient_size*	The size of the Sobel filter window, must support at least 3, 5 and 7.
in	*norm_type*	A flag indicating the norm used to compute the gradient, VX_NORM_L1 or VX_NORM_L2.
out	*output*	The output image in VX_DF_IMAGE_U8 format with values either 0 or 255.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.14 Channel Combine

3.14.1 Detailed Description

Implements the Channel Combine Kernel. This kernel takes multiple `VX_DF_IMAGE_U8` planes to recombine them into a multi-planar or interleaved format from `vx_df_image_e`. The user must specify only the number of channels that are appropriate for the combining operation. If a user specifies more channels than necessary, the operation results in an error. For the case where the destination image is a format with subsampling, the input channels are expected to have been subsampled before combining (by stretching and resizing).

Functions

- vx_node VX_API_CALL vxChannelCombineNode (vx_graph graph, vx_image plane0, vx_image plane1, vx_image plane2, vx_image plane3, vx_image output)

  [Graph] Creates a channel combine node.
- vx_status VX_API_CALL vxuChannelCombine (vx_context context, vx_image plane0, vx_image plane1, vx_image plane2, vx_image plane3, vx_image output)

  [Immediate] Invokes an immediate Channel Combine.

3.14.2 Function Documentation

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

[Graph] Creates a channel combine node.

Parameters

`in`	*graph*	The graph reference.
`in`	*plane0*	The plane that forms channel 0. Must be `VX_DF_IMAGE_U8`.
`in`	*plane1*	The plane that forms channel 1. Must be `VX_DF_IMAGE_U8`.
`in`	*plane2*	[optional] The plane that forms channel 2. Must be `VX_DF_IMAGE_U8`.
`in`	*plane3*	[optional] The plane that forms channel 3. Must be `VX_DF_IMAGE_U8`.
`out`	*output*	The output image. The format of the image must be defined, even if the image is virtual.

See Also

`VX_KERNEL_CHANNEL_COMBINE`

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Invokes an immediate Channel Combine.

Parameters

`in`	*context*	The reference to the overall context.

in	*plane0*	The plane that forms channel 0. Must be VX_DF_IMAGE_U8.
in	*plane1*	The plane that forms channel 1. Must be VX_DF_IMAGE_U8.
in	*plane2*	[optional] The plane that forms channel 2. Must be VX_DF_IMAGE_U8.
in	*plane3*	[optional] The plane that forms channel 3. Must be VX_DF_IMAGE_U8.
out	*output*	The output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.15 Channel Extract

3.15.1 Detailed Description

Implements the Channel Extraction Kernel. This kernel removes a single VX_DF_IMAGE_U8 channel (plane) from a multi-planar or interleaved image format from vx_df_image_e.

Functions

- vx_node VX_API_CALL vxChannelExtractNode (vx_graph graph, vx_image input, vx_enum channel, vx_image output)

  [Graph] Creates a channel extract node.
- vx_status VX_API_CALL vxuChannelExtract (vx_context context, vx_image input, vx_enum channel, vx_image output)

  [Immediate] Invokes an immediate Channel Extract.

3.15.2 Function Documentation

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

[Graph] Creates a channel extract node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image. Must be one of the defined vx_df_image_e multi-channel formats.
in	*channel*	The vx_channel_e channel to extract.
out	*output*	The output image. Must be VX_DF_IMAGE_U8.

See Also

VX_KERNEL_CHANNEL_EXTRACT

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Invokes an immediate Channel Extract.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image. Must be one of the defined vx_df_image_e multi-channel formats.
in	*channel*	The vx_channel_e enumeration to extract.
out	*output*	The output image. Must be VX_DF_IMAGE_U8.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.16 Color Convert

3.16.1 Detailed Description

Implements the Color Conversion Kernel. This kernel converts an image of a designated `vx_df_image_e` format to another `vx_df_image_e` format for those combinations listed in the below table, where the columns are output types and the rows are input types. The API version first supporting the conversion is also listed.

I/O	RGB	RGBX	NV12	NV21	UYVY	YUYV	IYUV	YUV4
RGB		1.0	1.0				1.0	1.0
RGBX	1.0		1.0				1.0	1.0
NV12	1.0	1.0					1.0	1.0
NV21	1.0	1.0					1.0	1.0
UYVY	1.0	1.0	1.0				1.0	
YUYV	1.0	1.0	1.0				1.0	
IYUV	1.0	1.0	1.0					1.0
YUV4								

The `vx_df_image_e` encoding, held in the `VX_IMAGE_FORMAT` attribute, describes the data layout. The interpretation of the colors is determined by the `VX_IMAGE_SPACE` (see `vx_color_space_e`) and `VX_IMAGE_RANGE` (see `vx_channel_range_e`) attributes of the image. OpenVX 1.1 implementations are required only to support images of `VX_COLOR_SPACE_BT709` and `VX_CHANNEL_RANGE_FULL`.

If the channel range is defined as `VX_CHANNEL_RANGE_FULL`, the conversion between the real number and integer quantizations of color channels is defined for red, green, blue, and Y as:

$$value_{real} = \frac{value_{integer}}{256.0}$$
$$value_{integer} = max(0, min(255, floor(value_{real} * 256.0)))$$

For the U and V channels, the conversion between real number and integer quantizations is:

$$value_{real} = \frac{(value_{integer} - 128.0)}{256.0}$$
$$value_{integer} = max(0, min(255, floor((value_{real} * 256.0) + 128)))$$

If the channel range is defined as `VX_CHANNEL_RANGE_RESTRICTED`, the conversion between the integer quantizations of color channels and the continuous representations is defined for red, green, blue, and Y as:

$$value_{real} = \frac{(value_{integer} - 16.0)}{219.0}$$
$$value_{integer} = max(0, min(255, floor((value_{real} * 219.0) + 16.5)))$$

For the U and V channels, the conversion between real number and integer quantizations is:

$$value_{real} = \frac{(value_{integer} - 128.0)}{224.0}$$
$$value_{integer} = max(0, min(255, floor((value_{real} * 224.0) + 128.5)))$$

The conversions between nonlinear-intensity Y'PbPr and R'G'B' real numbers are:

$$R' = Y' + 2(1 - K_r)Pr$$
$$B' = Y' + 2(1 - K_b)Pb$$
$$G' = Y' - \frac{2(K_r(1 - K_r)Pr + K_b(1 - K_b)Pb)}{1 - K_r - K_b}$$

$$Y' = (K_r * R') + (K_b * B') + (1 - K_r - K_b)G'$$
$$Pb = \frac{B'}{2} - \frac{(R' * K_r) + G'(1 - K_r - K_b)}{2(1 - K_b)}$$
$$Pr = \frac{R'}{2} - \frac{(B' * K_b) + G'(1 - K_r - K_b)}{2(1 - K_r)}$$

The means of reconstructing Pb and Pr values from chroma-downsampled formats is implementation-defined.

In VX_COLOR_SPACE_BT601_525 or VX_COLOR_SPACE_BT601_625:

$$K_r = 0.299$$
$$K_b = 0.114$$

In VX_COLOR_SPACE_BT709:

$$K_r = 0.2126$$
$$K_b = 0.0722$$

In all cases, for the purposes of conversion, these colour representations are interpreted as nonlinear in intensity, as defined by the BT.601, BT.709, and sRGB specifications. That is, the encoded colour channels are nonlinear R', G' and B', Y', Pb, and Pr.

Each channel of the R'G'B' representation can be converted to and from a linear-intensity RGB channel by these formulae:

$$value_{nonlinear} = 1.099 * value_{linear}{}^{0.45} - 0.099 \quad for \quad 1 \geq value_{linear} \geq 0.018$$
$$value_{nonlinear} = 4.500 * value_{linear} \quad for \quad 0.018 > value_{linear} \geq 0$$

$$value_{linear} = \left(\frac{value_{nonlinear} + 0.099}{1.099}\right)^{\frac{1}{0.45}} \quad for \quad 1 \geq value_{nonlinear} > 0.081$$
$$value_{linear} = \frac{value_{nonlinear}}{4.5} \quad for \quad 0.081 \geq value_{nonlinear} \geq 0$$

As the different color spaces have different RGB primaries, a conversion between them must transform the color coordinates into the new RGB space. Working with linear RGB values, the conversion formulae are:

$$R_{BT601\_525} = R_{BT601\_625} * 1.112302 + G_{BT601\_625} * -0.102441 + B_{BT601\_625} * -0.009860$$
$$G_{BT601\_525} = R_{BT601\_625} * -0.020497 + G_{BT601\_625} * 1.037030 + B_{BT601\_625} * -0.016533$$
$$B_{BT601\_525} = R_{BT601\_625} * 0.001704 + G_{BT601\_625} * 0.016063 + B_{BT601\_625} * 0.982233$$

$$R_{BT601\_525} = R_{BT709} * 1.065379 + G_{BT709} * -0.055401 + B_{BT709} * -0.009978$$
$$G_{BT601\_525} = R_{BT709} * -0.019633 + G_{BT709} * 1.036363 + B_{BT709} * -0.016731$$
$$B_{BT601\_525} = R_{BT709} * 0.001632 + G_{BT709} * 0.004412 + B_{BT709} * 0.993956$$

$$R_{BT601\_625} = R_{BT601\_525} * 0.900657 + G_{BT601\_525} * 0.088807 + B_{BT601\_525} * 0.010536$$
$$G_{BT601\_625} = R_{BT601\_525} * 0.017772 + G_{BT601\_525} * 0.965793 + B_{BT601\_525} * 0.016435$$
$$B_{BT601\_625} = R_{BT601\_525} * -0.001853 + G_{BT601\_525} * -0.015948 + B_{BT601\_525} * 1.017801$$

$$R_{BT601\_625} = R_{BT709} * 0.957815 + G_{BT709} * 0.042185$$
$$G_{BT601\_625} = G_{BT709}$$
$$B_{BT601\_625} = G_{BT709} * -0.011934 + B_{BT709} * 1.011934$$

$$R_{BT709} = R_{BT601\_525} * 0.939542 + G_{BT601\_525} * 0.050181 + B_{BT601\_525} * 0.010277$$
$$G_{BT709} = R_{BT601\_525} * 0.017772 + G_{BT601\_525} * 0.965793 + B_{BT601\_525} * 0.016435$$
$$B_{BT709} = R_{BT601\_525} * -0.001622 + G_{BT601\_525} * -0.004370 + B_{BT601\_525} * 1.005991$$

$$R_{BT709} = R_{BT601\_625} * 1.044043 + G_{BT601\_625} * -0.044043$$
$$G_{BT709} = G_{BT601\_625}$$
$$B_{BT709} = G_{BT601\_625} * 0.011793 + B_{BT601\_625} * 0.988207$$

A conversion between one YUV color space and another may therefore consist of the following transformations:

1. Convert quantized Y'CbCr ("YUV") to continuous, nonlinear Y'PbPr.
2. Convert continuous Y'PbPr to continuous, nonlinear R'G'B'.
3. Convert nonlinear R'G'B' to linear-intensity RGB (gamma-correction).
4. Convert linear RGB from the first color space to linear RGB in the second color space.
5. Convert linear RGB to nonlinear R'G'B' (gamma-conversion).
6. Convert nonlinear R'G'B' to Y'PbPr.
7. Convert continuous Y'PbPr to quantized Y'CbCr ("YUV").

   The above formulae and constants are defined in the ITU `BT.601` and `BT.709` specifications. The formulae for converting between RGB primaries can be derived from the specified primary chromaticity values and the specified white point by solving for the relative intensity of the primaries.

Functions

- vx_node VX_API_CALL vxColorConvertNode (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a color conversion node.
- vx_status VX_API_CALL vxuColorConvert (vx_context context, vx_image input, vx_image output)

  [Immediate] Invokes an immediate Color Conversion.

3.16.2 Function Documentation

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

[Graph] Creates a color conversion node.

Parameters

`in`	*graph*	The reference to the graph.
`in`	*input*	The input image from which to convert.
`out`	*output*	The output image to which to convert.

See Also

`VX_KERNEL_COLOR_CONVERT`

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Invokes an immediate Color Conversion.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*input*	The input image.
`out`	*output*	The output image.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.17 Convert Bit depth

3.17.1 Detailed Description

Converts image bit depth. This kernel converts an image from some source bit-depth to another bit-depth as described by the table below. If the input value is unsigned the shift must be in zeros. If the input value is signed, the shift used must be an arithmetic shift. The columns in the table below are the output types and the rows are the input types. The API version on which conversion is supported is also listed. (An *X* denotes an invalid operation.)

I/O	U8	U16	S16	U32	S32
U8	X		1.0		
U16		X	X		
S16	1.0	X	X		
U32				X	X
S32				X	X

Conversion Type The table below identifies the conversion types for the allowed bith depth conversions.

From	To	Conversion Type
U8	S16	Up-conversion
S16	U8	Down-conversion

Convert Policy Down-conversions with `VX_CONVERT_POLICY_WRAP` follow this equation:

```
output(x,y) = ((uint8)(input(x,y) >> shift));
```

Down-conversions with `VX_CONVERT_POLICY_SATURATE` follow this equation:

```
int16 value = input(x,y) >> shift;
value = value < 0 ? 0 : value;
value = value > 255 ? 255 : value;
output(x,y) = (uint8)value;
```

Up-conversions ignore the policy and perform this operation:

```
output(x,y) = ((int16)input(x,y)) << shift;
```

The valid values for 'shift' are as specified below, all other values produce undefined behavior.

```
0 <= shift < 8;
```

Functions

- vx_node VX_API_CALL vxConvertDepthNode (vx_graph graph, vx_image input, vx_image output, vx_enum policy, vx_scalar shift)

  [Graph] Creates a bit-depth conversion node.
- vx_status VX_API_CALL vxuConvertDepth (vx_context context, vx_image input, vx_image output, vx_enum policy, vx_int32 shift)

  [Immediate] Converts the input images bit-depth into the output image.

3.17.2 Function Documentation

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

[Graph] Creates a bit-depth conversion node.

Parameters

in	*graph*	The reference to the graph.

in	*input*	The input image.
out	*output*	The output image.
in	*policy*	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
in	*shift*	A scalar containing a VX_TYPE_INT32 of the shift value.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Converts the input images bit-depth into the output image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image.
out	*output*	The output image.
in	*policy*	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
in	*shift*	A scalar containing a VX_TYPE_INT32 of the shift value.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e..

3.18 Custom Convolution

3.18.1 Detailed Description

Convolves the input with the client supplied convolution matrix. The client can supply a vx_int16 typed convolution matrix $C_{m,n}$. Outputs will be in the VX_DF_IMAGE_S16 format unless a VX_DF_IMAGE_U8 image is explicitly provided. If values would have been out of range of U8 for VX_DF_IMAGE_U8, the values are clamped to 0 or 255.

$$k_0 = \frac{m}{2} \tag{3.1}$$

$$l_0 = \frac{n}{2} \tag{3.2}$$

$$sum = \sum_{k=0,l=0}^{k=m-1,l=n-1} input(x+k_0-k, y+l_0-l)C_{k,l} \tag{3.3}$$

Note

The above equation for this function is different than an equivalent operation suggested by the OpenCV Filter2-D function.

This translates into the C declaration:

```
// A horizontal Scharr gradient operator with different scale.
vx_int16 gx[3][3] = {
    {  3, 0, -3},
    { 10, 0,-10},
    {  3, 0, -3},
};
vx_uint32 scale = 8;
vx_convolution scharr_x = vxCreateConvolution(context, 3, 3);
vxCopyConvolutionCoefficients(scharr_x, (
  vx_int16*)gx, VX_WRITE_ONLY, VX_MEMORY_TYPE_HOST);
vxSetConvolutionAttribute(scharr_x,
  VX_CONVOLUTION_SCALE, &scale, sizeof(scale));
```

For VX_DF_IMAGE_U8 output, an additional step is taken:

$$output(x,y) = \begin{cases} 0 & \text{if } sum < 0 \\ 255 & \text{if } sum/scale > 255 \\ sum/scale & \text{otherwise} \end{cases}$$

For VX_DF_IMAGE_S16 output, the summation is simply set to the output

$$output(x,y) = sum/scale$$

The overflow policy used is VX_CONVERT_POLICY_SATURATE.

Functions

- vx_node VX_API_CALL vxConvolveNode (vx_graph graph, vx_image input, vx_convolution conv, vx_image output)

  [Graph] Creates a custom convolution node.
- vx_status VX_API_CALL vxuConvolve (vx_context context, vx_image input, vx_convolution conv, vx_image output)

  [Immediate] Computes a convolution on the input image with the supplied matrix.

3.18.2 Function Documentation

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

[Graph] Creates a custom convolution node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
in	*conv*	The vx_int16 convolution matrix.
out	*output*	The output image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Computes a convolution on the input image with the supplied matrix.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
in	*conv*	The vx_int16 convolution matrix.
out	*output*	The output image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.19 Dilate Image

3.19.1 Detailed Description

Implements Dilation, which *grows* the white space in a VX_DF_IMAGE_U8 Boolean image. This kernel uses a 3x3 box around the output pixel used to determine value.

$$dst(x,y) = \max_{\substack{x-1 \leq x' \leq x+1 \\ y-1 \leq y' \leq y+1}} src(x',y')$$

Note

For kernels that use other structuring patterns than 3x3 see vxNonLinearFilterNode or vxuNonLinearFilter.

Functions

- vx_node VX_API_CALL vxDilate3x3Node (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a Dilation Image Node.
- vx_status VX_API_CALL vxuDilate3x3 (vx_context context, vx_image input, vx_image output)

  [Immediate] Dilates an image by a 3x3 window.

3.19.2 Function Documentation

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

[Graph] Creates a Dilation Image Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Dilates an image by a 3x3 window.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.20 Equalize Histogram

3.20.1 Detailed Description

Equalizes the histogram of a grayscale image. This kernel uses Histogram Equalization to modify the values of a grayscale image so that it will automatically have a standardized brightness and contrast.

Functions

- vx_node VX_API_CALL vxEqualizeHistNode (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a Histogram Equalization node.
- vx_status VX_API_CALL vxuEqualizeHist (vx_context context, vx_image input, vx_image output)

  [Immediate] Equalizes the Histogram of a grayscale image.

3.20.2 Function Documentation

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

[Graph] Creates a Histogram Equalization node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The grayscale input image in `VX_DF_IMAGE_U8`.
out	*output*	The grayscale output image of type `VX_DF_IMAGE_U8` with equalized brightness and contrast.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Equalizes the Histogram of a grayscale image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The grayscale input image in `VX_DF_IMAGE_U8`
out	*output*	The grayscale output image of type `VX_DF_IMAGE_U8` with equalized brightness and contrast.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.21 Erode Image

3.21.1 Detailed Description

Implements Erosion, which *shrinks* the white space in a `VX_DF_IMAGE_U8` Boolean image. This kernel uses a 3x3 box around the output pixel used to determine value.

$$dst(x,y) = \min_{\substack{x-1 \le x' \le x+1 \\ y-1 \le y' \le y+1}} src(x',y')$$

Note

For kernels that use other structuring patterns than 3x3 see `vxNonLinearFilterNode` or `vxuNonLinearFilter`.

Functions

- vx_node VX_API_CALL vxErode3x3Node (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates an Erosion Image Node.
- vx_status VX_API_CALL vxuErode3x3 (vx_context context, vx_image input, vx_image output)

  [Immediate] Erodes an image by a 3x3 window.

3.21.2 Function Documentation

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

[Graph] Creates an Erosion Image Node.

Parameters

`in`	*graph*	The reference to the graph.
`in`	*input*	The input image in `VX_DF_IMAGE_U8` format.
`out`	*output*	The output image in `VX_DF_IMAGE_U8` format.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Erodes an image by a 3x3 window.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*input*	The input image in `VX_DF_IMAGE_U8` format.
`out`	*output*	The output image in `VX_DF_IMAGE_U8` format.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.22 Fast Corners

3.22.1 Detailed Description

Computes the corners in an image using a method based upon FAST9 algorithm suggested in [3] and with some updates from [4] with modifications described below. It extracts corners by evaluating pixels on the Bresenham circle around a candidate point. If N contiguous pixels are brighter than the candidate point by at least a threshold value t or darker by at least t , then the candidate point is considered to be a corner. For each detected corner, its strength is computed. Optionally, a non-maxima suppression step is applied on all detected corners to remove multiple or spurious responses.

3.22.2 Segment Test Detector

The FAST corner detector uses the pixels on a Bresenham circle of radius 3 (16 pixels) to classify whether a candidate point p is actually a corner, given the following variables.

$$
\begin{aligned}
I &= \text{input image} && (3.4)\\
p &= \text{candidate point position for a corner} && (3.5)\\
I_p &= \text{image intensity of the candidate point in image } I && (3.6)\\
x &= \text{pixel on the Bresenham circle around the candidate point } p && (3.7)\\
I_x &= \text{image intensity of the candidate point} && (3.8)\\
t &= \text{intensity difference threshold for a corner} && (3.9)\\
N &= \text{minimum number of contiguous pixel to detect a corner} && (3.10)\\
S &= \text{set of contiguous pixel on the Bresenham circle around the candidate point} && (3.11)\\
C_p &= \text{corner response at corner location } p && (3.12)\\
& && (3.13)
\end{aligned}
$$

The two conditions for FAST corner detection can be expressed as:

- C1: A set of N contiguous pixels S, $\forall x$ in S, $I_x > I_p + t$
- C2: A set of N contiguous pixels S, $\forall x$ in S, $I_x < I_p - t$

So when either of these two conditions is met, the candidate p is classified as a corner.
In this version of the FAST algorithm, the minimum number of contiguous pixels N is 9 (FAST9).
The value of the intensity difference threshold *strength_thresh*. of type `VX_TYPE_FLOAT32` must be within:

$$UINT8_{MIN} < t < UINT8_{MAX}$$

These limits are established due to the input data type `VX_DF_IMAGE_U8`.

Corner Strength Computation Once a corner has been detected, its strength (response, saliency, or score) shall be computed if nonmax_suppression is set to true, otherwise the value of strength is undefined. The corner response C_p function is defined as the largest threshold t for which the pixel p remains a corner.

Non-maximum suppression If the `nonmax_suppression` flag is true, a non-maxima suppression step is applied on the detected corners. The corner with coordinates (x, y) is kept if and only if

$$
\begin{aligned}
C_p(x,y) \geq C_p(x-1,y-1) \textit{ and } C_p(x,y) \geq C_p(x,y-1) \textit{ and}\\
C_p(x,y) \geq C_p(x+1,y-1) \textit{ and } C_p(x,y) \geq C_p(x-1,y) \textit{ and}\\
C_p(x,y) > C_p(x+1,y) \textit{ and } C_p(x,y) > C_p(x-1,y+1) \textit{ and}\\
C_p(x,y) > C_p(x,y+1) \textit{ and } C_p(x,y) > C_p(x+1,y+1)
\end{aligned}
$$

See Also

http://www.edwardrosten.com/work/fast.html
http://en.wikipedia.org/wiki/Features_from_accelerated_segment_test

Functions

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

  [Graph] Creates a FAST Corners Node.
- vx_status VX_API_CALL vxuFastCorners (vx_context context, vx_image input, vx_scalar strength_thresh, vx_bool nonmax_suppression, vx_array corners, vx_scalar num_corners)

  [Immediate] Computes corners on an image using FAST algorithm and produces the array of feature points.

3.22.3 Function Documentation

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

[Graph] Creates a FAST Corners Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input `VX_DF_IMAGE_U8` image.
in	*strength_thresh*	Threshold on difference between intensity of the central pixel and pixels on Bresenham's circle of radius 3 (`VX_TYPE_FLOAT32` scalar).
in	*nonmax_-suppression*	If true, non-maximum suppression is applied to detected corners before being placed in the `vx_array` of `VX_TYPE_KEYPOINT` objects.
out	*corners*	Output corner `vx_array` of `VX_TYPE_KEYPOINT`. The order of the key-points in this array is implementation dependent.
out	*num_corners*	The total number of detected corners in image (optional). Use a VX_TYPE_-SIZE scalar.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes corners on an image using FAST algorithm and produces the array of feature points.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input `VX_DF_IMAGE_U8` image.
in	*strength_thresh*	Threshold on difference between intensity of the central pixel and pixels on Bresenham's circle of radius 3 (`VX_TYPE_FLOAT32` scalar)
in	*nonmax_-suppression*	If true, non-maximum suppression is applied to detected corners before being places in the `vx_array` of `VX_TYPE_KEYPOINT` structs.
out	*corners*	Output corner `vx_array` of `VX_TYPE_KEYPOINT`. The order of the key-points in this array is implementation dependent.
out	*num_corners*	The total number of detected corners in image (optional). Use a VX_TYPE_-SIZE scalar.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.23 Gaussian Filter

3.23.1 Detailed Description

Computes a Gaussian filter over a window of the input image. This filter uses the following convolution matrix:

$$\mathbf{K}_{gaussian} = \begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix} * \frac{1}{16}$$

Functions

- vx_node VX_API_CALL vxGaussian3x3Node (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a Gaussian Filter Node.
- vx_status VX_API_CALL vxuGaussian3x3 (vx_context context, vx_image input, vx_image output)

  [Immediate] Computes a gaussian filter on the image by a 3x3 window.

3.23.2 Function Documentation

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

[Graph] Creates a Gaussian Filter Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Computes a gaussian filter on the image by a 3x3 window.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.24 Non Linear Filter

3.24.1 Detailed Description

Computes a non-linear filter over a window of the input image. The attribute VX_CONTEXT_NONLINEAR_MAX_DIMENSION enables the user to query the largest nonlinear filter supported by the implementation of vxNonLinearFilterNode. The implementation must support all dimensions (height or width, not necessarily the same) up to the value of this attribute. The lowest value that must be supported for this attribute is 9.

Functions

- vx_node VX_API_CALL vxNonLinearFilterNode (vx_graph graph, vx_enum function, vx_image input, vx_matrix mask, vx_image output)

  [Graph] Creates a Non-linear Filter Node.
- vx_status VX_API_CALL vxuNonLinearFilter (vx_context context, vx_enum function, vx_image input, vx_matrix mask, vx_image output)

  [Immediate] Creates a Non-linear Filter Node.

3.24.2 Function Documentation

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

[Graph] Creates a Non-linear Filter Node.

Parameters

in	*graph*	The reference to the graph.
in	*function*	The non-linear filter function. See `vx_non_linear_filter_e`.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
in	*mask*	The mask to be applied to the Non-linear function. `VX_MATRIX_ORIGIN` attribute is used to place the mask appropriately when computing the resulting image. See `vxCreateMatrixFromPattern`.
out	*output*	The output image in `VX_DF_IMAGE_U8` format.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Creates a Non-linear Filter Node.

Parameters

in	*context*	The reference to the overall context.
in	*function*	The non-linear filter function. See `vx_non_linear_filter_e`.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
in	*mask*	The mask to be applied to the Non-linear function. `VX_MATRIX_ORIGIN` attribute is used to place the mask appropriately when computing the resulting image. See `vxCreateMatrixFromPattern`.

out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.25 Harris Corners

3.25.1 Detailed Description

Computes the Harris Corners of an image. The Harris Corners are computed with several parameters

$$
\begin{aligned}
I &= \text{input image} && (3.14)\\
T_c &= \text{corner strength threshold} && (3.15)\\
r &= \text{euclidean radius} && (3.16)\\
k &= \text{sensitivity threshold} && (3.17)\\
w &= \text{window size} && (3.18)\\
b &= \text{block size} && (3.19)\\
& && (3.20)
\end{aligned}
$$

The computation to find the corner values or scores can be summarized as:

$$
\begin{aligned}
G_x &= Sobel_x(w,I) && (3.21)\\
G_y &= Sobel_y(w,I) && (3.22)\\
A &= window_{G_{x,y}}(x-b/2, y-b/2, x+b/2, y+b/2) && (3.23)\\
trace(A) &= \sum^{A} G_x^2 + \sum^{A} G_y^2 && (3.24)\\
det(A) &= \sum^{A} G_x^2 \sum^{A} G_y^2 - \left(\sum^{A} (G_x G_y)\right)^2 && (3.25)\\
M_c(x,y) &= det(A) - k * trace(A)^2 && (3.26)\\
V_c(x,y) &= \begin{cases} M_c(x,y) \text{ if } M_c(x,y) > T_c \\ 0 \text{ otherwise} \end{cases} && (3.27)
\end{aligned}
$$

where V_c is the thresholded corner value.

The normalized Sobel kernels used for the gradient computation shall be as shown below:

- For gradient size 3:

$$\mathbf{Sobel}_x(Normalized) = \frac{1}{4*255*b} * \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}$$

$$\mathbf{Sobel}_y(Normalized) = \frac{1}{4*255*b} * transpose(sobel_x) = \frac{1}{4*255*b} * \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix}$$

- For gradient size 5:

$$\mathbf{Sobel}_x(Normalized) = \frac{1}{16*255*b} * \begin{vmatrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 4 \\ -1 & -2 & 0 & 2 & 1 \end{vmatrix}$$

$$\mathbf{Sobel}_y(Normalized) = \frac{1}{16*255*b} * transpose(sobel_x)$$

- For gradient size 7:

$$\mathbf{Sobel}_x(Normalized) = \frac{1}{64*255*b} * \begin{vmatrix} -1 & -4 & -5 & 0 & 5 & 4 & 1 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -15 & -60 & -75 & 0 & 75 & 60 & 15 \\ -20 & -80 & -100 & 0 & 100 & 80 & 20 \\ -15 & -60 & -75 & 0 & 75 & 60 & 15 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -1 & -4 & -5 & 0 & 5 & 4 & 1 \end{vmatrix}$$

$$\mathbf{Sobel}_y(Normalized) = \frac{1}{64*255*b} * transpose(sobel_x)$$

V_c is then non-maximally suppressed using the following algorithm:

- Filter the features using the non-maximum suppression algorithm defined for vxFastCornersNode.
- Create an array of features sorted by V_c in descending order: $V_c(j) > V_c(j+1)$.
- Initialize an empty feature set $F = \{\}$
- For each feature j in the sorted array, while $V_c(j) > T_c$:
  - If there is no feature i in F such that the Euclidean distance between pixels i and j is less than r, add the feature j to the feature set F.

An implementation shall support all values of Euclidean distance r that satisfy:

```
0 <= max_dist <= 30
```

The feature set F is returned as a `vx_array` of `vx_keypoint_t` structs.

Functions

- vx_node VX_API_CALL 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)

  [Graph] Creates a Harris Corners Node.
- vx_status VX_API_CALL 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)

  [Immediate] Computes the Harris Corners over an image and produces the array of scored points.

3.25.2 Function Documentation

vx_node VX_API_CALL 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*)

[Graph] Creates a Harris Corners Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input `VX_DF_IMAGE_U8` image.
in	*strength_thresh*	The `VX_TYPE_FLOAT32` minimum threshold with which to eliminate Harris Corner scores (computed using the normalized Sobel kernel).
in	*min_distance*	The `VX_TYPE_FLOAT32` radial Euclidean distance for non-maximum suppression.
in	*sensitivity*	The `VX_TYPE_FLOAT32` scalar sensitivity threshold k from the Harris-Stephens equation.
in	*gradient_size*	The gradient window size to use on the input. The implementation must support at least 3, 5, and 7.
in	*block_size*	The block window size used to compute the Harris Corner score. The implementation must support at least 3, 5, and 7.
out	*corners*	The array of `VX_TYPE_KEYPOINT` objects. The order of the keypoints in this array is implementation dependent.

out	*num_corners*	The total number of detected corners in image (optional). Use a VX_TYPE_-SIZE scalar.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

vx_status VX_API_CALL 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*)

[Immediate] Computes the Harris Corners over an image and produces the array of scored points.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input `VX_DF_IMAGE_U8` image.
in	*strength_thresh*	The `VX_TYPE_FLOAT32` minimum threshold which to eliminate Harris Corner scores (computed using the normalized Sobel kernel).
in	*min_distance*	The `VX_TYPE_FLOAT32` radial Euclidean distance for non-maximum suppression.
in	*sensitivity*	The `VX_TYPE_FLOAT32` scalar sensitivity threshold k from the Harris-Stephens equation.
in	*gradient_size*	The gradient window size to use on the input. The implementation must support at least 3, 5, and 7.
in	*block_size*	The block window size used to compute the harris corner score. The implementation must support at least 3, 5, and 7.
out	*corners*	The array of `VX_TYPE_KEYPOINT` structs. The order of the keypoints in this array is implementation dependent.
out	*num_corners*	The total number of detected corners in image (optional). Use a VX_TYPE_-SIZE scalar

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.26 Histogram

3.26.1 Detailed Description

Generates a distribution from an image. This kernel counts the number of occurrences of each pixel value within the window size of a pre-calculated number of bins. A pixel with intensity 'I' will result in incrementing histogram bin 'i' where

$$i = (I - offset) * numBins/range for I >= offset$$

and

$$I < offset + range.$$

Pixels with intensities that don't meet these conditions will have no effect on the histogram. Here offset, range and numBins are values of histogram attributes (see `VX_DISTRIBUTION_OFFSET`, `VX_DISTRIBUTION_RANGE`, `VX_DISTRIBUTION_BINS`).

Functions

- vx_node VX_API_CALL vxHistogramNode (vx_graph graph, vx_image input, vx_distribution distribution)

  [Graph] Creates a Histogram node.
- vx_status VX_API_CALL vxuHistogram (vx_context context, vx_image input, vx_distribution distribution)

  [Immediate] Generates a distribution from an image.

3.26.2 Function Documentation

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

[Graph] Creates a Histogram node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in `VX_DF_IMAGE_U8`.
out	*distribution*	The output distribution.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Generates a distribution from an image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in `VX_DF_IMAGE_U8`
out	*distribution*	The output distribution.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.27 Gaussian Image Pyramid

3.27.1 Detailed Description

Computes a Gaussian Image Pyramid from an input image. This vision function creates the Gaussian image pyramid from the input image using the particular 5x5 Gaussian Kernel:

$$\mathbf{G} = \frac{1}{256} * \begin{vmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{vmatrix}$$

on each level of the pyramid then scales the image to the next level using `VX_INTERPOLATION_NEAREST_NEIGHBOR`. Level 0 shall be the same resolution as the input image. The pyramids must be configured with one of the following level scaling:

- `VX_SCALE_PYRAMID_HALF`
- `VX_SCALE_PYRAMID_ORB`

Functions

- vx_node VX_API_CALL vxGaussianPyramidNode (vx_graph graph, vx_image input, vx_pyramid gaussian)
  [Graph] Creates a node for a Gaussian Image Pyramid.
- vx_status VX_API_CALL vxuGaussianPyramid (vx_context context, vx_image input, vx_pyramid gaussian)
  [Immediate] Computes a Gaussian pyramid from an input image.

3.27.2 Function Documentation

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

[Graph] Creates a node for a Gaussian Image Pyramid.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
out	*gaussian*	The Gaussian pyramid with `VX_DF_IMAGE_U8` to construct.

See Also

Object: Pyramid

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes a Gaussian pyramid from an input image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8
out	*gaussian*	The Gaussian pyramid with VX_DF_IMAGE_U8 to construct.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.28 Laplacian Image Pyramid

3.28.1 Detailed Description

Computes a Laplacian Image Pyramid from an input image. This vision function creates the Laplacian image pyramid from the input image. First, a Gaussian pyramid with `VX_SCALE_PYRAMID_HALF` is created. Then, for each level i, the corresponding image I_i is blurred with Gaussian 5x5 filter, and the difference between the two images is the corresponding level L_i of the Laplacian pyramid:

$$L_i = I_i - Gaussian5x5(I_i).$$

Level 0 shall always have the same resolution as the input image.

Functions

- vx_node VX_API_CALL vxLaplacianPyramidNode (vx_graph graph, vx_image input, vx_pyramid laplacian, vx_image output)

  [Graph] Creates a node for a Laplacian Image Pyramid.
- vx_status VX_API_CALL vxuLaplacianPyramid (vx_context context, vx_image input, vx_pyramid laplacian, vx_image output)

  [Immediate] Computes a Laplacian pyramid from an input image.

3.28.2 Function Documentation

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

[Graph] Creates a node for a Laplacian Image Pyramid.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
out	*laplacian*	The Laplacian pyramid with `VX_DF_IMAGE_S16` to construct.
out	*output*	The lowest resolution image of type `VX_DF_IMAGE_S16` necessary to reconstruct the input image from the pyramid.

See Also

Object: Pyramid

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes a Laplacian pyramid from an input image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*laplacian*	The Laplacian pyramid with VX_DF_IMAGE_S16 to construct.
out	*output*	The lowest resolution image of type VX_DF_IMAGE_S16 necessary to re-construct the input image from the pyramid.

See Also

Object: Pyramid

Returns

A vx_status enumeration.

Return values

VX_SUCCESS	Success.
*	An error occured. See vx_status_e

3.29 Reconstruction from a Laplacian Image Pyramid

3.29.1 Detailed Description

Reconstructs the original image from a Laplacian Image Pyramid. This vision function reconstructs the image of the highest possible resolution from a Laplacian pyramid. The input image is added to the last level of the Laplacian pyramid L_{n-2}, the resulting image is upsampled to the resolution of the next pyramid level:

$$I_{n-2} = upsample(input + L_{n-1})$$

Correspondingly, for each pyramid level i, except for the first $i = 0$ and the last $i = n-1$:

$$I_{i-1} = upsample(I_i + L_i)$$

Finally, the output image is:

$$output = I_0 + L_0$$

Functions

- vx_node VX_API_CALL vxLaplacianReconstructNode (vx_graph graph, vx_pyramid laplacian, vx_image input, vx_image output)

  [Graph] Reconstructs an image from a Laplacian Image pyramid.
- vx_status VX_API_CALL vxuLaplacianReconstruct (vx_context context, vx_pyramid laplacian, vx_image input, vx_image output)

  [Immediate] Reconstructs an image from a Laplacian Image pyramid.

3.29.2 Function Documentation

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

[Graph] Reconstructs an image from a Laplacian Image pyramid.

Parameters

in	*graph*	The reference to the graph.
in	*laplacian*	The Laplacian pyramid with `VX_DF_IMAGE_S16` format.
in	*input*	The lowest resolution image of type `VX_DF_IMAGE_S16` for the Laplacian pyramid
out	*output*	The output image of type `VX_DF_IMAGE_U8` with the highest possible resolution reconstructed from the Laplacian pyramid.

See Also

Object: Pyramid

Returns

`vx_node`.

Return values

0	Node could not be created.
$*$	Node handle.

vx_status VX_API_CALL vxuLaplacianReconstruct (vx_context *context*, vx_pyramid *laplacian*, vx_image *input*, vx_image *output*)

[Immediate] Reconstructs an image from a Laplacian Image pyramid.

Parameters

in	*context*	The reference to the overall context.
in	*laplacian*	The Laplacian pyramid with VX_DF_IMAGE_S16 format.
in	*input*	The lowest resolution image of type VX_DF_IMAGE_S16 for the Laplacian pyramid
out	*output*	The output image of type VX_DF_IMAGE_U8 with the highest possible resolution reconstructed from the Laplacian pyramid.

See Also

Object: Pyramid

Returns

A vx_status enumeration.

Return values

VX_SUCCESS	Success.
*	An error occured. See vx_status_e

3.30 Integral Image

3.30.1 Detailed Description

Computes the integral image of the input. Each output pixel is the sum of the corresponding input pixel and all other pixels above and to the left of it.

$$dst(x,y) = sum(x,y)$$

where, for x>=0 and y>=0

$$sum(x,y) = src(x,y) + sum(x-1,y) + sum(x,y-1) - sum(x-1,y-1)$$

otherwise,

$$sum(x,y) = 0$$

The overflow policy used is `VX_CONVERT_POLICY_WRAP`.

Functions

- vx_node VX_API_CALL vxIntegralImageNode (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates an Integral Image Node.
- vx_status VX_API_CALL vxuIntegralImage (vx_context context, vx_image input, vx_image output)

  [Immediate] Computes the integral image of the input.

3.30.2 Function Documentation

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

[Graph] Creates an Integral Image Node.

Parameters

`in`	*graph*	The reference to the graph.
`in`	*input*	The input image in `VX_DF_IMAGE_U8` format.
`out`	*output*	The output image in `VX_DF_IMAGE_U32` format.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes the integral image of the input.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*input*	The input image in `VX_DF_IMAGE_U8` format.
`out`	*output*	The output image in `VX_DF_IMAGE_U32` format.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.31 Magnitude

3.31.1 Detailed Description

Implements the Gradient Magnitude Computation Kernel. This kernel takes two gradients in `VX_DF_IMAGE_S16` format and computes the `VX_DF_IMAGE_S16` normalized magnitude. Magnitude is computed as:

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

The conceptual definition describing the overflow is given as:
uint16 z = uint16(sqrt(double(uint32(int32(x) $*$ int32(x)) + uint32(int32(y) $*$ int32(y)))) + 0.5);
int16 mag = z $>$ 32767 ? 32767 : z;

Functions

- vx_node VX_API_CALL vxMagnitudeNode (vx_graph graph, vx_image grad_x, vx_image grad_y, vx_image mag)

  [Graph] Create a Magnitude node.
- vx_status VX_API_CALL vxuMagnitude (vx_context context, vx_image grad_x, vx_image grad_y, vx_image mag)

  [Immediate] Invokes an immediate Magnitude.

3.31.2 Function Documentation

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

[Graph] Create a Magnitude node.

Parameters

in	*graph*	The reference to the graph.
in	*grad_x*	The input x image. This must be in `VX_DF_IMAGE_S16` format.
in	*grad_y*	The input y image. This must be in `VX_DF_IMAGE_S16` format.
out	*mag*	The magnitude image. This is in `VX_DF_IMAGE_S16` format.

See Also

`VX_KERNEL_MAGNITUDE`

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Invokes an immediate Magnitude.

Parameters

in	*context*	The reference to the overall context.

in	*grad_x*	The input x image. This must be in VX_DF_IMAGE_S16 format.
in	*grad_y*	The input y image. This must be in VX_DF_IMAGE_S16 format.
out	*mag*	The magnitude image. This will be in VX_DF_IMAGE_S16 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.32 Mean and Standard Deviation

3.32.1 Detailed Description

Computes the mean pixel value and the standard deviation of the pixels in the input image (which has a dimension width and height). The mean value is computed as:

$$\mu = \frac{\left(\sum_{y=0}^{h} \sum_{x=0}^{w} src(x,y)\right)}{(width * height)}$$

The standard deviation is computed as:

$$\sigma = \sqrt{\frac{\left(\sum_{y=0}^{h} \sum_{x=0}^{w} (\mu - src(x,y))^2\right)}{(width * height)}}$$

Functions

- vx_node VX_API_CALL vxMeanStdDevNode (vx_graph graph, vx_image input, vx_scalar mean, vx_scalar stddev)

    [Graph] Creates a mean value and standard deviation node.
- vx_status VX_API_CALL vxuMeanStdDev (vx_context context, vx_image input, vx_float32 *mean, vx_float32 *stddev)

    [Immediate] Computes the mean value and standard deviation.

3.32.2 Function Documentation

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

[Graph] Creates a mean value and standard deviation node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image. `VX_DF_IMAGE_U8` is supported.
out	*mean*	The `VX_TYPE_FLOAT32` average pixel value.
out	*stddev*	The `VX_TYPE_FLOAT32` standard deviation of the pixel values.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes the mean value and standard deviation.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image. `VX_DF_IMAGE_U8` is supported.

out	*mean*	The average pixel value.
out	*stddev*	The standard deviation of the pixel values.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.33 Median Filter

3.33.1 Detailed Description

Computes a median pixel value over a window of the input image. The median is the middle value over an odd-numbered, sorted range of values.

Note

For kernels that use other structuring patterns than 3x3 see `vxNonLinearFilterNode` or `vxuNonLinearFilter`.

Functions

- vx_node VX_API_CALL vxMedian3x3Node (vx_graph graph, vx_image input, vx_image output)
  [Graph] Creates a Median Image Node.
- vx_status VX_API_CALL vxuMedian3x3 (vx_context context, vx_image input, vx_image output)
  [Immediate] Computes a median filter on the image by a 3x3 window.

3.33.2 Function Documentation

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

[Graph] Creates a Median Image Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
out	*output*	The output image in `VX_DF_IMAGE_U8` format.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Computes a median filter on the image by a 3x3 window.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
out	*output*	The output image in `VX_DF_IMAGE_U8` format.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.34 Min, Max Location

3.34.1 Detailed Description

Finds the minimum and maximum values in an image and a location for each. If the input image has several minimums/maximums, the kernel returns all of them.

$$minVal = \min_{\substack{0 \le x' \le width \\ 0 \le y' \le height}} src(x', y')$$

$$maxVal = \max_{\substack{0 \le x' \le width \\ 0 \le y' \le height}} src(x', y')$$

Functions

- vx_node VX_API_CALL 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)

  [Graph] Creates a min,max,loc node.
- vx_status VX_API_CALL 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)

  [Immediate] Computes the minimum and maximum values of the image.

3.34.2 Function Documentation

vx_node VX_API_CALL 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*)

[Graph] Creates a min,max,loc node.

Parameters

in	*graph*	The reference to create the graph.
in	*input*	The input image in `VX_DF_IMAGE_U8` or `VX_DF_IMAGE_S16` format.
out	*minVal*	The minimum value in the image, which corresponds to the type of the input.
out	*maxVal*	The maximum value in the image, which corresponds to the type of the input.
out	*minLoc*	The minimum `VX_TYPE_COORDINATES2D` locations (optional). If the input image has several minimums, the kernel will return up to the capacity of the array.
out	*maxLoc*	The maximum `VX_TYPE_COORDINATES2D` locations (optional). If the input image has several maximums, the kernel will return up to the capacity of the array.
out	*minCount*	The total number of detected minimums in image (optional). Use a `VX_TYPE_UINT32` scalar.
out	*maxCount*	The total number of detected maximums in image (optional). Use a `VX_TYPE_UINT32` scalar.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

vx_status VX_API_CALL 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*)

[Immediate] Computes the minimum and maximum values of the image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.
out	*minVal*	The minimum value in the image, which corresponds to the type of the input.
out	*maxVal*	The maximum value in the image, which corresponds to the type of the input.
out	*minLoc*	The minimum VX_TYPE_COORDINATES2D locations (optional). If the input image has several minimums, the kernel will return up to the capacity of the array.
out	*maxLoc*	The maximum VX_TYPE_COORDINATES2D locations (optional). If the input image has several maximums, the kernel will return up to the capacity of the array.
out	*minCount*	The total number of detected minimums in image (optional). Use a VX_TYPE_UINT32 scalar.
out	*maxCount*	The total number of detected maximums in image (optional). Use a VX_TYPE_UINT32 scalar.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.35 Optical Flow Pyramid (LK)

3.35.1 Detailed Description

Computes the optical flow using the Lucas-Kanade method between two pyramid images. The function is an implementation of the algorithm described in [1]. The function inputs are two `vx_pyramid` objects, old and new, along with a `vx_array` of `vx_keypoint_t` structs to track from the old `vx_pyramid`. The function outputs a `vx_array` of `vx_keypoint_t` structs that were tracked from the old `vx_pyramid` to the new `vx_pyramid`. Each element in the `vx_array` of `vx_keypoint_t` structs in the new array may be valid or not. The implementation shall return the same number of `vx_keypoint_t` structs in the new `vx_array` that were in the older `vx_array`.

In more detail: The Lucas-Kanade method finds the affine motion vector V for each point in the old image tracking points array, using the following equation:

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x{}^2 & \sum_i I_x * I_y \\ \sum_i I_x * I_y & \sum_i I_y{}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x * I_t \\ -\sum_i I_y * I_t \end{bmatrix}$$

Where I_x and I_y are obtained using the Scharr gradients on the input image:

$$G_x = \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix}$$

$$G_y = \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

I_t is obtained by a simple difference between the same pixel in both images. I is defined as the adjacent pixels to the point $p(x,y)$ under consideration. With a given window size of M, I is M^2 points. The pixel $p(x,y)$ is centered in the window. In practice, to get an accurate solution, it is necessary to iterate multiple times on this scheme (in a Newton-Raphson fashion) until:

- the residual of the affine motion vector is smaller than a threshold
- And/or maximum number of iteration achieved. Each iteration, the estimation of the previous iteration is used by changing I_t to be the difference between the old image and the pixel with the estimated coordinates in the new image. Each iteration the function checks if the pixel to track was lost. The criteria for lost tracking is that the matrix above is invertible. (The determinant of the matrix is less than a threshold : 10^{-7} .) Or the minimum eigenvalue of the matrix is smaller then a threshold (10^{-4}). Also lost tracking happens when the point tracked coordinate is outside the image coordinates. When `vx_true_e` is given as the input to `use_initial_estimates`, the algorithm starts by calculating I_t as the difference between the old image and the pixel with the initial estimated coordinates in the new image. The input `vx_array` of `vx_keypoint_t` structs with `tracking_status` set to zero (lost) are copied to the new `vx_array`.

Clients are responsible for editing the output `vx_array` of `vx_keypoint_t` structs array before applying it as the input `vx_array` of `vx_keypoint_t` structs for the next frame. For example, `vx_keypoint_t` structs with `tracking_status` set to zero may be removed by a client for efficiency.

This function changes just the *x*, *y*, and *tracking_status* members of the `vx_keypoint_t` structure and behaves as if it copied the rest from the old tracking `vx_keypoint_t` to new image `vx_keypoint_t`.

Functions

- vx_node VX_API_CALL 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)

  [Graph] Creates a Lucas Kanade Tracking Node.
- vx_status VX_API_CALL 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)

  [Immediate] Computes an optical flow on two images.

3.35.2 Function Documentation

vx_node VX_API_CALL 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*)

[Graph] Creates a Lucas Kanade Tracking Node.

Parameters

in	*graph*	The reference to the graph.
in	*old_images*	Input of first (old) image pyramid in VX_DF_IMAGE_U8.
in	*new_images*	Input of destination (new) image pyramid VX_DF_IMAGE_U8.
in	*old_points*	An array of key points in a vx_array of VX_TYPE_KEYPOINT; those key points are defined at the *old_images* high resolution pyramid.
in	*new_points_-estimates*	An array of estimation on what is the output key points in a vx_array of VX_TYPE_KEYPOINT; those keypoints are defined at the *new_images* high resolution pyramid.
out	*new_points*	An output array of key points in a vx_array of VX_TYPE_KEYPOINT; those key points are defined at the *new_images* high resolution pyramid.
in	*termination*	The termination can be VX_TERM_CRITERIA_ITERATIONS or VX_TE-RM_CRITERIA_EPSILON or VX_TERM_CRITERIA_BOTH.
in	*epsilon*	The vx_float32 error for terminating the algorithm.
in	*num_iterations*	The number of iterations. Use a VX_TYPE_UINT32 scalar.
in	*use_initial_-estimate*	Use a VX_TYPE_BOOL scalar.
in	*window_-dimension*	The size of the window on which to perform the algorithm. See VX_CONTE-XT_OPTICAL_FLOW_MAX_WINDOW_DIMENSION

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

vx_status VX_API_CALL 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*)

[Immediate] Computes an optical flow on two images.

Parameters

in	*context*	The reference to the overall context.
in	*old_images*	Input of first (old) image pyramid in VX_DF_IMAGE_U8.
in	*new_images*	Input of destination (new) image pyramid in VX_DF_IMAGE_U8
in	*old_points*	an array of key points in a vx_array of VX_TYPE_KEYPOINT those key points are defined at the old_images high resolution pyramid
in	*new_points_-estimates*	an array of estimation on what is the output key points in a vx_array of VX_TYPE_KEYPOINT those keypoints are defined at the new_images high resolution pyramid
out	*new_points*	an output array of key points in a vx_array of VX_TYPE_KEYPOINT those key points are defined at the new_images high resolution pyramid
in	*termination*	termination can be VX_TERM_CRITERIA_ITERATIONS or VX_TERM_-CRITERIA_EPSILON or VX_TERM_CRITERIA_BOTH
in	*epsilon*	is the vx_float32 error for terminating the algorithm
in	*num_iterations*	is the number of iterations. Use a VX_TYPE_UINT32 scalar.
in	*use_initial_-estimate*	Can be set to either vx_false_e or vx_true_e.

in	*window_-dimension*	The size of the window on which to perform the algorithm. See VX_CONTEXT_OPTICAL_FLOW_MAX_WINDOW_DIMENSION

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.36 Phase

3.36.1 Detailed Description

Implements the Gradient Phase Computation Kernel. This kernel takes two gradients in `VX_DF_IMAGE_S16` format and computes the angles for each pixel and stores this in a `VX_DF_IMAGE_U8` image.

$$\phi = \tan^{-1}\frac{grad_y(x,y)}{grad_x(x,y)}$$

Where ϕ is then translated to $0 \leq \phi < 2\pi$. Each ϕ value is then mapped to the range 0 to 255 inclusive.

Functions

- vx_node VX_API_CALL vxPhaseNode (vx_graph graph, vx_image grad_x, vx_image grad_y, vx_image orientation)

  [Graph] Creates a Phase node.
- vx_status VX_API_CALL vxuPhase (vx_context context, vx_image grad_x, vx_image grad_y, vx_image orientation)

  [Immediate] Invokes an immediate Phase.

3.36.2 Function Documentation

vx_node VX_API_CALL vxPhaseNode (vx_graph *graph,* vx_image *grad_x,* vx_image *grad_y,* vx_image *orientation*)

[Graph] Creates a Phase node.

Parameters

`in`	*graph*	The reference to the graph.
`in`	*grad_x*	The input x image. This must be in `VX_DF_IMAGE_S16` format.
`in`	*grad_y*	The input y image. This must be in `VX_DF_IMAGE_S16` format.
`out`	*orientation*	The phase image. This is in `VX_DF_IMAGE_U8` format.

See Also

`VX_KERNEL_PHASE`

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

vx_status VX_API_CALL vxuPhase (vx_context *context,* vx_image *grad_x,* vx_image *grad_y,* vx_image *orientation*)

[Immediate] Invokes an immediate Phase.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*grad_x*	The input x image. This must be in `VX_DF_IMAGE_S16` format.

in	*grad_y*	The input y image. This must be in VX_DF_IMAGE_S16 format.
out	*orientation*	The phase image. This will be in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.37 Pixel-wise Multiplication

3.37.1 Detailed Description

Performs element-wise multiplication between two images and a scalar value. Pixel-wise multiplication is performed between the pixel values in two VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 images and a scalar floating-point number *scale*. The output image can be VX_DF_IMAGE_U8 only if both source images are VX_DF_IMAGE_U8 and the output image is explicitly set to VX_DF_IMAGE_U8. It is otherwise VX_DF_IMAGE_S16. If one of the input images is of type VX_DF_IMAGE_S16, all values are converted to VX_DF_IMAGE_S16.

The scale with a value of $1/2^n$, where n is an integer and $0 \leq n \leq 15$, and 1/255 (0x1.010102p-8 C99 float hex) must be supported. The support for other values of scale is not prohibited. Furthermore, for scale with a value of 1/255 the rounding policy of VX_ROUND_POLICY_TO_NEAREST_EVEN must be supported whereas for the scale with value of $1/2^n$ the rounding policy of VX_ROUND_POLICY_TO_ZERO must be supported. The support of other rounding modes for any values of scale is not prohibited.

The rounding policy VX_ROUND_POLICY_TO_ZERO for this function is defined as:

$$reference(x,y,scale) = truncate(((int32\_t)in_1(x,y)) * ((int32\_t)in_2(x,y)) * (double)scale)$$

The rounding policy VX_ROUND_POLICY_TO_NEAREST_EVEN for this function is defined as:

$$reference(x,y,scale) = round\_to\_nearest\_even(((int32\_t)in_1(x,y)) * ((int32\_t)in_2(x,y)) * (double)scale)$$

The overflow handling is controlled by an overflow-policy parameter. For each pixel value in the two input images:

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

Functions

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

  [Graph] Creates an pixelwise-multiplication node.
- vx_status VX_API_CALL vxuMultiply (vx_context context, vx_image in1, vx_image in2, vx_float32 scale, vx_enum overflow_policy, vx_enum rounding_policy, vx_image out)

  [Immediate] Performs elementwise multiplications on pixel values in the input images and a scale.

3.37.2 Function Documentation

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

[Graph] Creates an pixelwise-multiplication node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16.
in	*in2*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16.
in	*scale*	A non-negative VX_TYPE_FLOAT32 multiplied to each product before overflow handling.
in	*overflow_policy*	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
in	*rounding_policy*	A VX_TYPE_ENUM of the vx_round_policy_e enumeration.
out	*out*	The output image, a VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 image.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Performs elementwise multiplications on pixel values in the input images and a scale.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*in1*	A `VX_DF_IMAGE_U8` or `VX_DF_IMAGE_S16` input image.
`in`	*in2*	A `VX_DF_IMAGE_U8` or `VX_DF_IMAGE_S16` input image.
`in`	*scale*	A non-negative `VX_TYPE_FLOAT32` multiplied to each product before overflow handling.
`in`	*overflow_policy*	A `vx_convert_policy_e` enumeration.
`in`	*rounding_policy*	A `vx_round_policy_e` enumeration.
`out`	*out*	The output image in `VX_DF_IMAGE_U8` or `VX_DF_IMAGE_S16` format.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.38 Remap

3.38.1 Detailed Description

Maps output pixels in an image from input pixels in an image. Remap takes a remap table object vx_remap to map a set of output pixels back to source input pixels. A remap is typically defined as:

$$output(x,y) = input(mapx(x,y), mapy(x,y));$$

for every (x,y) in the destination image

However, the mapping functions are contained in the vx_remap object.

Functions

- vx_node VX_API_CALL vxRemapNode (vx_graph graph, vx_image input, vx_remap table, vx_enum policy, vx_image output)

  [Graph] Creates a Remap Node.
- vx_status VX_API_CALL vxuRemap (vx_context context, vx_image input, vx_remap table, vx_enum policy, vx_image output)

  [Immediate] Remaps an output image from an input image.

3.38.2 Function Documentation

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

[Graph] Creates a Remap Node.

Parameters

in	*graph*	The reference to the graph that will contain the node.
in	*input*	The input `VX_DF_IMAGE_U8` image.
in	*table*	The remap table object.
in	*policy*	An interpolation type from `vx_interpolation_type_e`. `VX_INTERPOLATION_AREA` is not supported.
out	*output*	The output `VX_DF_IMAGE_U8` image.

Note

The border modes `VX_NODE_BORDER` value `VX_BORDER_UNDEFINED` and `VX_BORDER_CONSTANT` are supported.

Returns

A `vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Remaps an output image from an input image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*table*	The remap table object.
in	*policy*	The interpolation policy from vx_interpolation_type_e. VX_INTERPOLATION-_AREA is not supported.
out	*output*	The output VX_DF_IMAGE_U8 image.

Returns

A vx_status_e enumeration.

3.39 Scale Image

3.39.1 Detailed Description

Implements the Image Resizing Kernel. This kernel resizes an image from the source to the destination dimensions. The supported interpolation types are currently:

- VX_INTERPOLATION_NEAREST_NEIGHBOR
- VX_INTERPOLATION_AREA
- VX_INTERPOLATION_BILINEAR

The sample positions used to determine output pixel values are generated by scaling the outside edges of the source image pixels to the outside edges of the destination image pixels. As described in the documentation for vx_interpolation_type_e, samples are taken at pixel centers. This means that, unless the scale is 1:1, the sample position for the top left destination pixel typically does not fall exactly on the top left source pixel but will be generated by interpolation.

That is, the sample positions corresponding in source and destination are defined by the following equations:

$$x_{input} = \left((x_{output} + 0.5) * \frac{width_{input}}{width_{output}}\right) - 0.5$$

$$y_{input} = \left((y_{output} + 0.5) * \frac{height_{input}}{height_{output}}\right) - 0.5$$

$$x_{output} = \left((x_{input} + 0.5) * \frac{width_{output}}{width_{input}}\right) - 0.5$$

$$y_{output} = \left((y_{input} + 0.5) * \frac{height_{output}}{height_{input}}\right) - 0.5$$

- For VX_INTERPOLATION_NEAREST_NEIGHBOR, the output value is that of the pixel whose centre is closest to the sample point.
- For VX_INTERPOLATION_BILINEAR, the output value is formed by a weighted average of the nearest source pixels to the sample point. That is:

$$x_{lower} = \lfloor x_{input} \rfloor$$

$$y_{lower} = \lfloor y_{input} \rfloor$$

$$s = x_{input} - x_{lower}$$

$$t = y_{input} - y_{lower}$$

$$output(x_{input}, y_{input}) = (1-s)(1-t) * input(x_{lower}, y_{lower}) + s(1-t) * input(x_{lower}+1, y_{lower})$$
$$+(1-s)t * input(x_{lower}, y_{lower}+1) + s * t * input(x_{lower}+1, y_{lower}+1)$$

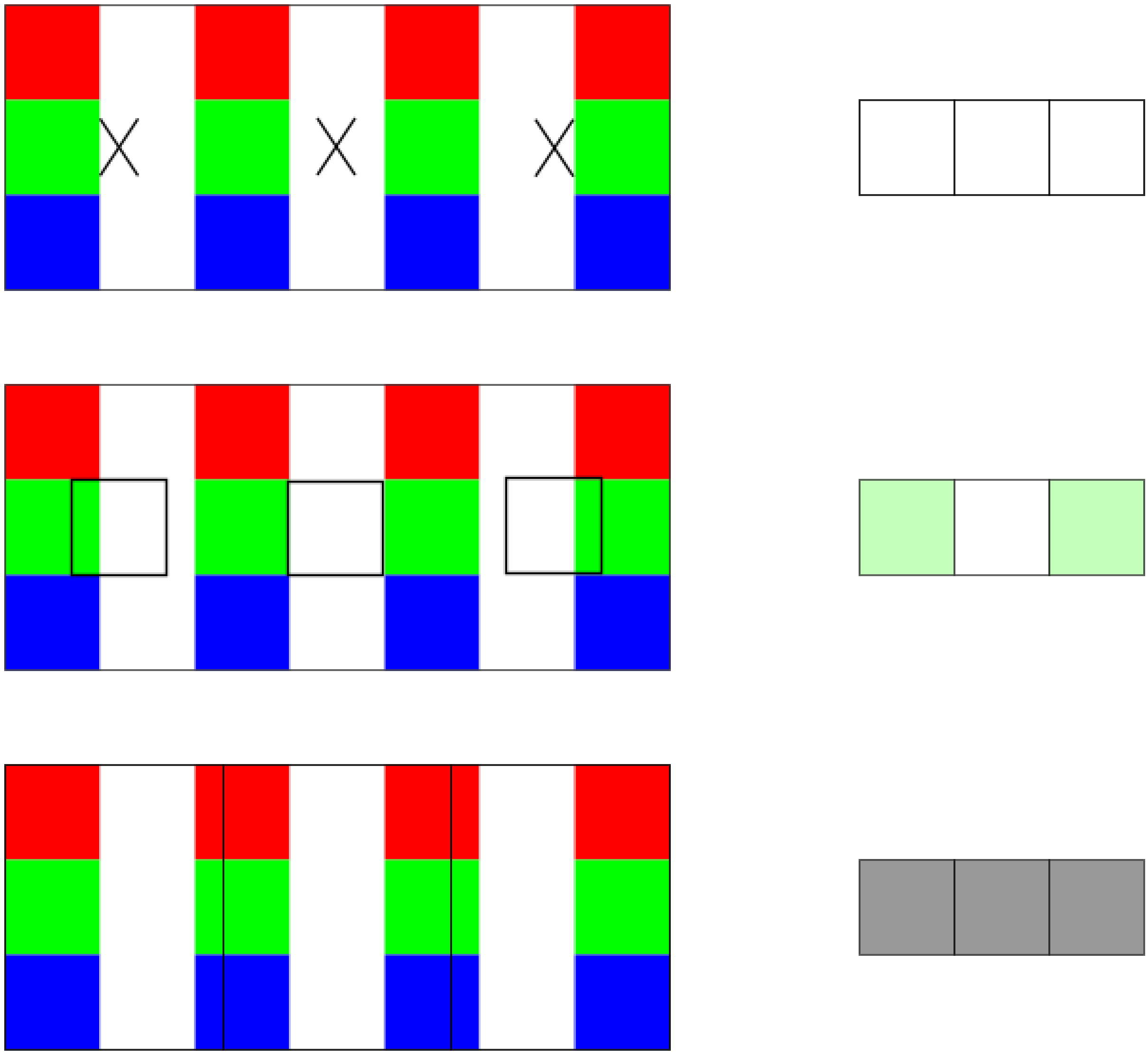
- For VX_INTERPOLATION_AREA, the implementation is expected to generate each output pixel by sampling all the source pixels that are at least partly covered by the area bounded by:

$$\left(x_{output} * \frac{width_{input}}{width_{output}}\right) - 0.5, \left(y_{output} * \frac{height_{input}}{height_{output}}\right) - 0.5$$

and

$$\left((x_{output}+1) * \frac{width_{input}}{width_{output}}\right) - 0.5, \left((y_{output}+1) * \frac{height_{input}}{height_{output}}\right) - 0.5$$

The details of this sampling method are implementation-defined. The implementation should perform enough sampling to avoid aliasing, but there is no requirement that the sample areas for adjacent output pixels be disjoint, nor that the pixels be weighted evenly.



The above diagram shows three sampling methods used to shrink a 7x3 image to 3x1.

The topmost image pair shows nearest-neighbor sampling, with crosses on the left image marking the sample positions in the source that are used to generate the output image on the right. As the pixel centre closest to the sample position is white in all cases, the resulting 3x1 image is white.

The middle image pair shows bilinear sampling, with black squares on the left image showing the region in the source being sampled to generate each pixel on the destination image on the right. This sample area is always the size of an input pixel. The outer destination pixels partly sample from the outermost green pixels, so their resulting value is a weighted average of white and green.

The bottom image pair shows area sampling. The black rectangles in the source image on the left show the bounds of the projection of the destination pixels onto the source. The destination pixels on the right are formed by averaging at least those source pixels whose areas are wholly or partly contained within those rectangles. The manner of this averaging is implementation-defined; the example shown here weights the contribution of each source pixel by the amount of that pixel's area contained within the black rectangle.

Functions

- vx_node VX_API_CALL vxHalfScaleGaussianNode (vx_graph graph, vx_image input, vx_image output, vx_int32 kernel_size)

  [Graph] Performs a Gaussian Blur on an image then half-scales it. The interpolation mode used is nearest-neighbor.
- vx_node VX_API_CALL vxScaleImageNode (vx_graph graph, vx_image src, vx_image dst, vx_enum type)

  [Graph] Creates a Scale Image Node.

- vx_status VX_API_CALL vxuHalfScaleGaussian (vx_context context, vx_image input, vx_image output, vx_int32 kernel_size)

  [Immediate] Performs a Gaussian Blur on an image then half-scales it. The interpolation mode used is nearest-neighbor.
- vx_status VX_API_CALL vxuScaleImage (vx_context context, vx_image src, vx_image dst, vx_enum type)

  [Immediate] Scales an input image to an output image.

3.39.2 Function Documentation

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

[Graph] Creates a Scale Image Node.

Parameters

in	*graph*	The reference to the graph.
in	*src*	The source image of type `VX_DF_IMAGE_U8`.
out	*dst*	The destination image of type `VX_DF_IMAGE_U8`.
in	*type*	The interpolation type to use.

See Also

vx_interpolation_type_e.

Note

The destination image must have a defined size and format. The border modes `VX_NODE_BORDER` value `VX_BORDER_UNDEFINED`, `VX_BORDER_REPLICATE` and `VX_BORDER_CONSTANT` are supported.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Graph] Performs a Gaussian Blur on an image then half-scales it. The interpolation mode used is nearest-neighbor. The output image size is determined by:

$$W_{output} = \frac{W_{input}+1}{2}, H_{output} = \frac{H_{input}+1}{2}$$

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input `VX_DF_IMAGE_U8` image.
out	*output*	The output `VX_DF_IMAGE_U8` image.
in	*kernel_size*	The input size of the Gaussian filter. Supported values are 1, 3 and 5.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Scales an input image to an output image.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*src*	The source image of type `VX_DF_IMAGE_U8`.
`out`	*dst*	The destintation image of type `VX_DF_IMAGE_U8`.
`in`	*type*	The interpolation type.

See Also

vx_interpolation_type_e.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

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

[Immediate] Performs a Gaussian Blur on an image then half-scales it. The interpolation mode used is nearest-neighbor.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*input*	The input `VX_DF_IMAGE_U8` image.
`out`	*output*	The output `VX_DF_IMAGE_U8` image.
`in`	*kernel_size*	The input size of the Gaussian filter. Supported values are 1, 3 and 5.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.40 Sobel 3x3

3.40.1 Detailed Description

Implements the Sobel Image Filter Kernel. This kernel produces two output planes (one can be omitted) in the x and y plane. The Sobel Operators G_x, G_y are defined as:

$$\mathbf{G}_x = \begin{vmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{vmatrix}, \mathbf{G}_y = \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{vmatrix}$$

Functions

- vx_node VX_API_CALL vxSobel3x3Node (vx_graph graph, vx_image input, vx_image output_x, vx_image output_y)

  [Graph] Creates a Sobel3x3 node.
- vx_status VX_API_CALL vxuSobel3x3 (vx_context context, vx_image input, vx_image output_x, vx_image output_y)

  [Immediate] Invokes an immediate Sobel 3x3.

3.40.2 Function Documentation

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

[Graph] Creates a Sobel3x3 node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
out	*output_x*	[optional] The output gradient in the x direction in `VX_DF_IMAGE_S16`.
out	*output_y*	[optional] The output gradient in the y direction in `VX_DF_IMAGE_S16`.

See Also

`VX_KERNEL_SOBEL_3x3`

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Invokes an immediate Sobel 3x3.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in `VX_DF_IMAGE_U8` format.
out	*output_x*	[optional] The output gradient in the x direction in `VX_DF_IMAGE_S16`.

out	*output_y*	[optional] The output gradient in the y direction in VX_DF_IMAGE_S16.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.41 TableLookup

3.41.1 Detailed Description

Implements the Table Lookup Image Kernel. This kernel uses each pixel in an image to index into a LUT and put the indexed LUT value into the output image. The formats supported are `VX_DF_IMAGE_U8` and `VX_DF_IMAGE_S16`.

Functions

- vx_node VX_API_CALL vxTableLookupNode (vx_graph graph, vx_image input, vx_lut lut, vx_image output)

    [Graph] Creates a Table Lookup node. If a value from the input image is not present in the lookup table, the result is undefined.
- vx_status VX_API_CALL vxuTableLookup (vx_context context, vx_image input, vx_lut lut, vx_image output)

    [Immediate] Processes the image through the LUT.

3.41.2 Function Documentation

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

[Graph] Creates a Table Lookup node. If a value from the input image is not present in the lookup table, the result is undefined.

Parameters

`in`	*graph*	The reference to the graph.
`in`	*input*	The input image in `VX_DF_IMAGE_U8` or `VX_DF_IMAGE_S16`.
`in`	*lut*	The LUT which is of type `VX_TYPE_UINT8` or `VX_TYPE_INT16`.
`out`	*output*	The output image of the same type as the input image.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

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

[Immediate] Processes the image through the LUT.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*input*	The input image in `VX_DF_IMAGE_U8` or `VX_DF_IMAGE_S16`.
`in`	*lut*	The LUT which is of type `VX_TYPE_UINT8` or `VX_TYPE_INT16`.
`out`	*output*	The output image of the same type as the input image.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.42 Thresholding

3.42.1 Detailed Description

Thresholds an input image and produces an output Boolean image. In VX_THRESHOLD_TYPE_BINARY, the output is determined by:

$$dst(x,y) = \begin{cases} true\ value & \text{if } src(x,y) > threshold \\ false\ value & \text{otherwise} \end{cases}$$

In VX_THRESHOLD_TYPE_RANGE, the output is determined by:

$$dst(x,y) = \begin{cases} false\ value & \text{if } src(x,y) > upper \\ false\ value & \text{if } src(x,y) < lower \\ true\ value & \text{otherwise} \end{cases}$$

Where 'false value' is the value indicated by the VX_THRESHOLD_FALSE_VALUE attribute of the *thresh* parameter, and the 'true value' is the value indicated by the VX_THRESHOLD_TRUE_VALUE attribute of the *thresh* parameter.

Functions

- vx_node VX_API_CALL vxThresholdNode (vx_graph graph, vx_image input, vx_threshold thresh, vx_image output)

  [Graph] Creates a Threshold node.
- vx_status VX_API_CALL vxuThreshold (vx_context context, vx_image input, vx_threshold thresh, vx_image output)

  [Immediate] Threshold's an input image and produces a VX_DF_IMAGE_U8 ∗ boolean image.

3.42.2 Function Documentation

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

[Graph] Creates a Threshold node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image. VX_DF_IMAGE_U8 is supported.
in	*thresh*	The thresholding object that defines the parameters of the operation. The VX_THRESHOLD_TRUE_VALUE and VX_THRESHOLD_FALSE_VALUE are taken into account.
out	*output*	The output Boolean image with values either VX_THRESHOLD_TRUE_VALUE or VX_THRESHOLD_FALSE_VALUE from the *thresh* parameter.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using vxGetStatus

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

[Immediate] Threshold's an input image and produces a VX_DF_IMAGE_U8 ∗ boolean image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image. VX_DF_IMAGE_U8 is supported.
in	*thresh*	The thresholding object that defines the parameters of the operation. The VX_THRESHOLD_TRUE_VALUE and VX_THRESHOLD_FALSE_VALUE are taken into account.
out	*output*	The output Boolean image with values either VX_THRESHOLD_TRUE_VALUE or VX_THRESHOLD_FALSE_VALUE from the *thresh* parameter.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.43 Warp Affine

3.43.1 Detailed Description

Performs an affine transform on an image. This kernel performs an affine transform with a 2x3 Matrix M with this method of pixel coordinate translation:

$$x0 = M_{1,1} * x + M_{1,2} * y + M_{1,3} \quad (3.28)$$
$$y0 = M_{2,1} * x + M_{2,2} * y + M_{2,3} \quad (3.29)$$
$$output(x,y) = input(x0,y0) \quad (3.30)$$

This translates into the C declaration:

```
// x0 = a x + b y + c;
// y0 = d x + e y + f;
vx_float32 mat[3][2] = {
    {a, d}, // 'x' coefficients
    {b, e}, // 'y' coefficients
    {c, f}, // 'offsets'
};
vx_matrix matrix = vxCreateMatrix(context,
  VX_TYPE_FLOAT32, 2, 3);
vxCopyMatrix(matrix, mat, VX_WRITE_ONLY,
  VX_MEMORY_TYPE_HOST);
```

Functions

- vx_status VX_API_CALL vxuWarpAffine (vx_context context, vx_image input, vx_matrix matrix, vx_enum type, vx_image output)

  [Immediate] Performs an Affine warp on an image.
- vx_node VX_API_CALL vxWarpAffineNode (vx_graph graph, vx_image input, vx_matrix matrix, vx_enum type, vx_image output)

  [Graph] Creates an Affine Warp Node.

3.43.2 Function Documentation

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

[Graph] Creates an Affine Warp Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*matrix*	The affine matrix. Must be 2x3 of type VX_TYPE_FLOAT32.
in	*type*	The interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_AREA is not supported.
out	*output*	The output VX_DF_IMAGE_U8 image.

Note

The border modes VX_NODE_BORDER value VX_BORDER_UNDEFINED and VX_BORDER_CONSTANT are supported.

Returns

vx_node.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Performs an Affine warp on an image.

Parameters

`in`	*context*	The reference to the overall context.
`in`	*input*	The input `VX_DF_IMAGE_U8` image.
`in`	*matrix*	The affine matrix. Must be 2x3 of type VX_TYPE_FLOAT32.
`in`	*type*	The interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_-AREA is not supported.
`out`	*output*	The output `VX_DF_IMAGE_U8` image.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.44 Warp Perspective

3.44.1 Detailed Description

Performs a perspective transform on an image. This kernel performs an perspective transform with a 3x3 Matrix M with this method of pixel coordinate translation:

$$x0 = M_{1,1} * x + M_{1,2} * y + M_{1,3} \tag{3.31}$$
$$y0 = M_{2,1} * x + M_{2,2} * y + M_{2,3} \tag{3.32}$$
$$z0 = M_{3,1} * x + M_{3,2} * y + M_{3,3} \tag{3.33}$$
$$output(x,y) = input(\frac{x0}{z0}, \frac{y0}{z0}) \tag{3.34}$$

This translates into the C declaration:

```
// x0 = a x + b y + c;
// y0 = d x + e y + f;
// z0 = g x + h y + i;
vx_float32 mat[3][3] = {
    {a, d, g}, // 'x' coefficients
    {b, e, h}, // 'y' coefficients
    {c, f, i}, // 'offsets'
};
vx_matrix matrix = vxCreateMatrix(context,
  VX_TYPE_FLOAT32, 3, 3);
vxCopyMatrix(matrix, mat, VX_WRITE_ONLY,
  VX_MEMORY_TYPE_HOST);
```

Functions

- vx_status VX_API_CALL vxuWarpPerspective (vx_context context, vx_image input, vx_matrix matrix, vx_enum type, vx_image output)

  [Immediate] Performs an Perspective warp on an image.
- vx_node VX_API_CALL vxWarpPerspectiveNode (vx_graph graph, vx_image input, vx_matrix matrix, vx_enum type, vx_image output)

  [Graph] Creates a Perspective Warp Node.

3.44.2 Function Documentation

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

[Graph] Creates a Perspective Warp Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input `VX_DF_IMAGE_U8` image.
in	*matrix*	The perspective matrix. Must be 3x3 of type `VX_TYPE_FLOAT32`.
in	*type*	The interpolation type from `vx_interpolation_type_e`. `VX_INTERPOLATION_AREA` is not supported.
out	*output*	The output `VX_DF_IMAGE_U8` image.

Note

The border modes `VX_NODE_BORDER` value `VX_BORDER_UNDEFINED` and `VX_BORDER_CONSTANT` are supported.

Returns

`vx_node`.

Return values

vx_node	A node reference. Any possible errors preventing a successful creation should be checked using `vxGetStatus`

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

[Immediate] Performs an Perspective warp on an image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input `VX_DF_IMAGE_U8` image.
in	*matrix*	The perspective matrix. Must be 3x3 of type VX_TYPE_FLOAT32.
in	*type*	The interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_-AREA is not supported.
out	*output*	The output `VX_DF_IMAGE_U8` image.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See `vx_status_e`.

3.45 Basic Features

3.45.1 Detailed Description

The basic parts of OpenVX needed for computation. Types in OpenVX intended to be derived from the C99 Section 7.18 standard definition of fixed width types.

Modules

- Objects

  Defines the basic objects within OpenVX.

Data Structures

- struct vx_coordinates2d_t

  The 2D Coordinates structure. More...
- struct vx_coordinates3d_t

  The 3D Coordinates structure. More...
- struct vx_keypoint_t

  The keypoint data structure. More...
- struct vx_rectangle_t

  The rectangle data structure that is shared with the users. The area of the rectangle can be computed as (end_x-start_x)∗(end_y-start_y). More...

Macros

- #define VX_API_CALL

  Defines calling convention for OpenVX API.
- #define VX_ATTRIBUTE_BASE(vendor, object) (((vendor) << 20) | (object << 8))

  Defines the manner in which to combine the Vendor and Object IDs to get the base value of the enumeration.
- #define VX_ATTRIBUTE_ID_MASK (0x000000FF)

  An object's attribute ID is within the range of $[0, 2^8 - 1]$ *(inclusive).*
- #define VX_CALLBACK

  Defines calling convention for user callbacks.
- #define VX_DF_IMAGE(a, b, c, d) ((a) | (b << 8) | (c << 16) | (d << 24))

  Converts a set of four chars into a `uint32_t` *container of a VX_DF_IMAGE code.*
- #define VX_ENUM_BASE(vendor, id) (((vendor) << 20) | (id << 12))

  Defines the manner in which to combine the Vendor and Object IDs to get the base value of the enumeration.
- #define VX_ENUM_MASK (0x00000FFF)

  A generic enumeration list can have values between $[0, 2^{12} - 1]$ *(inclusive).*
- #define VX_ENUM_TYPE(e) (((vx_uint32)e & VX_ENUM_TYPE_MASK) >> 12)

  A macro to extract the enum type from an enumerated value.
- #define VX_ENUM_TYPE_MASK (0x000FF000)

  A type of enumeration. The valid range is between $[0, 2^8 - 1]$ *(inclusive).*
- #define VX_FMT_REF "%p"

  Use to aid in debugging values in OpenVX.
- #define VX_FMT_SIZE "%zu"

  Use to aid in debugging values in OpenVX.
- #define VX_KERNEL_BASE(vendor, lib) (((vendor) << 20) | (lib << 12))

  Defines the manner in which to combine the Vendor and Library IDs to get the base value of the enumeration.
- #define VX_KERNEL_MASK (0x00000FFF)

  An individual kernel in a library has its own unique ID within $[0, 2^{12} - 1]$ *(inclusive).*
- #define VX_LIBRARY(e) (((vx_uint32)e & VX_LIBRARY_MASK) >> 12)

A macro to extract the kernel library enumeration from a enumerated kernel value.

- #define VX_LIBRARY_MASK (0x000FF000)

  A library is a set of vision kernels with its own ID supplied by a vendor. The vendor defines the library ID. The range is $[0, 2^8 - 1]$ *inclusive.*
- #define VX_MAX_LOG_MESSAGE_LEN (1024)

  Defines the length of a message buffer to copy from the log, including the trailing zero.
- #define VX_SCALE_UNITY (1024u)

  Use to indicate the 1:1 ratio in Q22.10 format.
- #define VX_TYPE(e) (((vx_uint32)e & VX_TYPE_MASK) $>>$ 8)

  A macro to extract the type from an enumerated attribute value.
- #define VX_TYPE_MASK (0x000FFF00)

  A type mask removes the scalar/object type from the attribute. It is 3 nibbles in size and is contained between the third and second byte.
- #define VX_VENDOR(e) (((vx_uint32)e & VX_VENDOR_MASK) $>>$ 20)

  A macro to extract the vendor ID from the enumerated value.
- #define VX_VENDOR_MASK (0xFFF00000)

  Vendor IDs are 2 nibbles in size and are located in the upper byte of the 4 bytes of an enumeration.
- #define VX_VERSION VX_VERSION_1_1

  Defines the OpenVX Version Number.
- #define VX_VERSION_1_0 (VX_VERSION_MAJOR(1) | VX_VERSION_MINOR(0))

  Defines the predefined version number for 1.0.
- #define VX_VERSION_1_1 (VX_VERSION_MAJOR(1) | VX_VERSION_MINOR(1))

  Defines the predefined version number for 1.1.
- #define VX_VERSION_MAJOR(x) ((x & 0xFF) $<<$ 8)

  Defines the major version number macro.
- #define VX_VERSION_MINOR(x) ((x & 0xFF) $<<$ 0)

  Defines the minor version number macro.

Typedefs

- typedef char vx_char

  An 8 bit ASCII character.
- typedef uint32_t vx_df_image

  Used to hold a VX_DF_IMAGE code to describe the pixel format and color space.
- typedef int32_t vx_enum

  Sets the standard enumeration type size to be a fixed quantity.
- typedef float vx_float32

  A 32-bit float value.
- typedef double vx_float64

  A 64-bit float value (aka double).
- typedef int16_t vx_int16

  A 16-bit signed value.
- typedef int32_t vx_int32

  A 32-bit signed value.
- typedef int64_t vx_int64

  A 64-bit signed value.
- typedef int8_t vx_int8

  An 8-bit signed value.
- typedef size_t vx_size

  A wrapper of `size_t` to keep the naming convention uniform.
- typedef vx_enum vx_status

A formal status type with known fixed size.

- typedef uint16_t vx_uint16

  A 16-bit unsigned value.
- typedef uint32_t vx_uint32

  A 32-bit unsigned value.
- typedef uint64_t vx_uint64

  A 64-bit unsigned value.
- typedef uint8_t vx_uint8

  An 8-bit unsigned value.

Enumerations

- enum vx_bool {
  vx_false_e = 0,
  vx_true_e }

  A Boolean value. This allows 0 to be FALSE, as it is in C, and any non-zero to be TRUE.
- enum vx_channel_e {
  VX_CHANNEL_0 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x0,
  VX_CHANNEL_1 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x1,
  VX_CHANNEL_2 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x2,
  VX_CHANNEL_3 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x3,
  VX_CHANNEL_R = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x10,
  VX_CHANNEL_G = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x11,
  VX_CHANNEL_B = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x12,
  VX_CHANNEL_A = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x13,
  VX_CHANNEL_Y = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x14,
  VX_CHANNEL_U = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x15,
  VX_CHANNEL_V = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x16 }

  The channel enumerations for channel extractions.
- enum vx_convert_policy_e {
  VX_CONVERT_POLICY_WRAP = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVERT_POLICY << 12)) + 0x0,
  VX_CONVERT_POLICY_SATURATE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVERT_POLICY << 12)) + 0x1 }

  The Conversion Policy Enumeration.
- enum vx_df_image_e {
  VX_DF_IMAGE_VIRT = (('V') | ('I' << 8) | ('R' << 16) | ('T' << 24)),
  VX_DF_IMAGE_RGB = (('R') | ('G' << 8) | ('B' << 16) | ('2' << 24)),
  VX_DF_IMAGE_RGBX = (('R') | ('G' << 8) | ('B' << 16) | ('A' << 24)),
  VX_DF_IMAGE_NV12 = (('N') | ('V' << 8) | ('1' << 16) | ('2' << 24)),
  VX_DF_IMAGE_NV21 = (('N') | ('V' << 8) | ('2' << 16) | ('1' << 24)),
  VX_DF_IMAGE_UYVY = (('U') | ('Y' << 8) | ('V' << 16) | ('Y' << 24)),
  VX_DF_IMAGE_YUYV = (('Y') | ('U' << 8) | ('Y' << 16) | ('V' << 24)),
  VX_DF_IMAGE_IYUV = (('I') | ('Y' << 8) | ('U' << 16) | ('V' << 24)),
  VX_DF_IMAGE_YUV4 = (('Y') | ('U' << 8) | ('V' << 16) | ('4' << 24)),
  VX_DF_IMAGE_U8 = (('U') | ('0' << 8) | ('0' << 16) | ('8' << 24)),
  VX_DF_IMAGE_U16 = (('U') | ('0' << 8) | ('1' << 16) | ('6' << 24)),
  VX_DF_IMAGE_S16 = (('S') | ('0' << 8) | ('1' << 16) | ('6' << 24)),
  VX_DF_IMAGE_U32 = (('U') | ('0' << 8) | ('3' << 16) | ('2' << 24)),
  VX_DF_IMAGE_S32 = (('S') | ('0' << 8) | ('3' << 16) | ('2' << 24)) }

  Based on the VX_DF_IMAGE definition.
- enum vx_enum_e {

VX_ENUM_DIRECTION = 0x00,
VX_ENUM_ACTION = 0x01,
VX_ENUM_HINT = 0x02,
VX_ENUM_DIRECTIVE = 0x03,
VX_ENUM_INTERPOLATION = 0x04,
VX_ENUM_OVERFLOW = 0x05,
VX_ENUM_COLOR_SPACE = 0x06,
VX_ENUM_COLOR_RANGE = 0x07,
VX_ENUM_PARAMETER_STATE = 0x08,
VX_ENUM_CHANNEL = 0x09,
VX_ENUM_CONVERT_POLICY = 0x0A,
VX_ENUM_THRESHOLD_TYPE = 0x0B,
VX_ENUM_BORDER = 0x0C,
VX_ENUM_COMPARISON = 0x0D,
VX_ENUM_MEMORY_TYPE = 0x0E,
VX_ENUM_TERM_CRITERIA = 0x0F,
VX_ENUM_NORM_TYPE = 0x10,
VX_ENUM_ACCESSOR = 0x11,
VX_ENUM_ROUND_POLICY = 0x12,
VX_ENUM_TARGET = 0x13,
VX_ENUM_BORDER_POLICY = 0x14,
VX_ENUM_GRAPH_STATE = 0x15,
VX_ENUM_NONLINEAR = 0x16,
VX_ENUM_PATTERN = 0x17 }

The set of supported enumerations in OpenVX.

- enum vx_interpolation_type_e {
  VX_INTERPOLATION_NEAREST_NEIGHBOR = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_INTERPOLATION << 12)) + 0x0,
  VX_INTERPOLATION_BILINEAR = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_INTERPOLATION << 12)) + 0x1,
  VX_INTERPOLATION_AREA = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_INTERPOLATION << 12)) + 0x2 }

  The image reconstruction filters supported by image resampling operations.
- enum vx_non_linear_filter_e {
  VX_NONLINEAR_FILTER_MEDIAN = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NONLINEAR << 12)) + 0x0,
  VX_NONLINEAR_FILTER_MIN = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NONLINEAR << 12)) + 0x1,
  VX_NONLINEAR_FILTER_MAX = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NONLINEAR << 12)) + 0x2 }

  An enumeration of non-linear filter functions.
- enum vx_pattern_e {
  VX_PATTERN_BOX = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_PATTERN << 12)) + 0x0,
  VX_PATTERN_CROSS = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_PATTERN << 12)) + 0x1,
  VX_PATTERN_DISK = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_PATTERN << 12)) + 0x2,
  VX_PATTERN_OTHER = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_PATTERN << 12)) + 0x3 }

  An enumeration of matrix patterns. See `vxCreateMatrixFromPattern`
- enum vx_status_e {

VX_STATUS_MIN = -25,
VX_ERROR_REFERENCE_NONZERO = -24,
VX_ERROR_MULTIPLE_WRITERS = -23,
VX_ERROR_GRAPH_ABANDONED = -22,
VX_ERROR_GRAPH_SCHEDULED = -21,
VX_ERROR_INVALID_SCOPE = -20,
VX_ERROR_INVALID_NODE = -19,
VX_ERROR_INVALID_GRAPH = -18,
VX_ERROR_INVALID_TYPE = -17,
VX_ERROR_INVALID_VALUE = -16,
VX_ERROR_INVALID_DIMENSION = -15,
VX_ERROR_INVALID_FORMAT = -14,
VX_ERROR_INVALID_LINK = -13,
VX_ERROR_INVALID_REFERENCE = -12,
VX_ERROR_INVALID_MODULE = -11,
VX_ERROR_INVALID_PARAMETERS = -10,
VX_ERROR_OPTIMIZED_AWAY = -9,
VX_ERROR_NO_MEMORY = -8,
VX_ERROR_NO_RESOURCES = -7,
VX_ERROR_NOT_COMPATIBLE = -6,
VX_ERROR_NOT_ALLOCATED = -5,
VX_ERROR_NOT_SUFFICIENT = -4,
VX_ERROR_NOT_SUPPORTED = -3,
VX_ERROR_NOT_IMPLEMENTED = -2,
VX_FAILURE = -1,
VX_SUCCESS = 0 }

The enumeration of all status codes.

- enum vx_target_e {
VX_TARGET_ANY = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_TARGET << 12)) + 0x0000,
VX_TARGET_STRING = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_TARGET << 12)) + 0x0001,
VX_TARGET_VENDOR_BEGIN = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_TARGET << 12)) + 0x1000 }

  The Target Enumeration.

- enum vx_type_e {

VX_TYPE_INVALID = 0x000,
VX_TYPE_CHAR = 0x001,
VX_TYPE_INT8 = 0x002,
VX_TYPE_UINT8 = 0x003,
VX_TYPE_INT16 = 0x004,
VX_TYPE_UINT16 = 0x005,
VX_TYPE_INT32 = 0x006,
VX_TYPE_UINT32 = 0x007,
VX_TYPE_INT64 = 0x008,
VX_TYPE_UINT64 = 0x009,
VX_TYPE_FLOAT32 = 0x00A,
VX_TYPE_FLOAT64 = 0x00B,
VX_TYPE_ENUM = 0x00C,
VX_TYPE_SIZE = 0x00D,
VX_TYPE_DF_IMAGE = 0x00E,
VX_TYPE_BOOL = 0x010,
VX_TYPE_SCALAR_MAX,
VX_TYPE_RECTANGLE = 0x020,
VX_TYPE_KEYPOINT = 0x021,
VX_TYPE_COORDINATES2D = 0x022,
VX_TYPE_COORDINATES3D = 0x023,
VX_TYPE_USER_STRUCT_START = 0x100,
VX_TYPE_VENDOR_STRUCT_START = 0x400,
VX_TYPE_KHRONOS_OBJECT_START = 0x800,
VX_TYPE_VENDOR_OBJECT_START = 0xC00,
VX_TYPE_KHRONOS_STRUCT_MAX = VX_TYPE_USER_STRUCT_START - 1,
VX_TYPE_USER_STRUCT_END = VX_TYPE_VENDOR_STRUCT_START - 1,
VX_TYPE_VENDOR_STRUCT_END = VX_TYPE_KHRONOS_OBJECT_START - 1,
VX_TYPE_KHRONOS_OBJECT_END = VX_TYPE_VENDOR_OBJECT_START - 1,
VX_TYPE_VENDOR_OBJECT_END = 0xFFF,
VX_TYPE_REFERENCE = 0x800,
VX_TYPE_CONTEXT = 0x801,
VX_TYPE_GRAPH = 0x802,
VX_TYPE_NODE = 0x803,
VX_TYPE_KERNEL = 0x804,
VX_TYPE_PARAMETER = 0x805,
VX_TYPE_DELAY = 0x806,
VX_TYPE_LUT = 0x807,
VX_TYPE_DISTRIBUTION = 0x808,
VX_TYPE_PYRAMID = 0x809,
VX_TYPE_THRESHOLD = 0x80A,
VX_TYPE_MATRIX = 0x80B,
VX_TYPE_CONVOLUTION = 0x80C,
VX_TYPE_SCALAR = 0x80D,
VX_TYPE_ARRAY = 0x80E,
VX_TYPE_IMAGE = 0x80F,
VX_TYPE_REMAP = 0x810,
VX_TYPE_ERROR = 0x811,
VX_TYPE_META_FORMAT = 0x812,
VX_TYPE_OBJECT_ARRAY = 0x813 }

The type enumeration lists all the known types in OpenVX.

- enum vx_vendor_id_e {

VX_ID_KHRONOS = 0x000,
VX_ID_TI = 0x001,
VX_ID_QUALCOMM = 0x002,
VX_ID_NVIDIA = 0x003,
VX_ID_ARM = 0x004,
VX_ID_BDTI = 0x005,
VX_ID_RENESAS = 0x006,
VX_ID_VIVANTE = 0x007,
VX_ID_XILINX = 0x008,
VX_ID_AXIS = 0x009,
VX_ID_MOVIDIUS = 0x00A,
VX_ID_SAMSUNG = 0x00B,
VX_ID_FREESCALE = 0x00C,
VX_ID_AMD = 0x00D,
VX_ID_BROADCOM = 0x00E,
VX_ID_INTEL = 0x00F,
VX_ID_MARVELL = 0x010,
VX_ID_MEDIATEK = 0x011,
VX_ID_ST = 0x012,
VX_ID_CEVA = 0x013,
VX_ID_ITSEEZ = 0x014,
VX_ID_IMAGINATION =0x015,
VX_ID_NXP = 0x016,
VX_ID_VIDEANTIS = 0x017,
VX_ID_SYNOPSYS = 0x018,
VX_ID_CADENCE = 0x019,
VX_ID_HUAWEI = 0x01A,
VX_ID_USER = 0xFFE,
VX_ID_MAX = 0xFFF,
VX_ID_DEFAULT = VX_ID_MAX }

The Vendor ID of the Implementation. As new vendors submit their implementations, this enumeration will grow.

Functions

- vx_status VX_API_CALL vxGetStatus (vx_reference reference)

  Provides a generic API to return status values from Object constructors if they fail.

3.45.2 Data Structure Documentation

struct vx_coordinates2d_t

The 2D Coordinates structure.

Definition at line 1483 of file vx_types.h.

Data Fields

vx_uint32	x	The X coordinate.
vx_uint32	y	The Y coordinate.

struct vx_coordinates3d_t

The 3D Coordinates structure.

Definition at line 1491 of file vx_types.h.

Data Fields

vx_uint32	x	The X coordinate.
vx_uint32	y	The Y coordinate.
vx_uint32	z	The Z coordinate.

struct vx_keypoint_t

The keypoint data structure.

Definition at line 1460 of file vx_types.h.

Data Fields

vx_int32	x	The x coordinate.
vx_int32	y	The y coordinate.
vx_float32	strength	The strength of the keypoint. Its definition is specific to the corner detector.
vx_float32	scale	Initialized to 0 by corner detectors.
vx_float32	orientation	Initialized to 0 by corner detectors.
vx_int32	tracking_status	A zero indicates a lost point. Initialized to 1 by corner detectors.
vx_float32	error	A tracking method specific error. Initialized to 0 by corner detectors.

struct vx_rectangle_t

The rectangle data structure that is shared with the users. The area of the rectangle can be computed as (end_x-start_x)∗(end_y-start_y).

Definition at line 1473 of file vx_types.h.

Data Fields

vx_uint32	start_x	The Start X coordinate.
vx_uint32	start_y	The Start Y coordinate.
vx_uint32	end_x	The End X coordinate.
vx_uint32	end_y	The End Y coordinate.

3.45.3 Macro Definition Documentation

#define VX_TYPE_MASK (0x000FFF00)

A type mask removes the scalar/object type from the attribute. It is 3 nibbles in size and is contained between the third and second byte.

See Also

vx_type_e

Definition at line 456 of file vx_types.h.

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

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

Note

Use a `vx_df_image` variable to hold the value.

Definition at line 509 of file vx_types.h.

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

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

From any enumerated value (with exceptions), the vendor, and enumeration type should be extractable. Those types that are exceptions are `vx_vendor_id_e`, `vx_type_e`, `vx_enum_e`, `vx_df_image_e`, and `vx_bool`.

Definition at line 533 of file vx_types.h.

3.45.4 Typedef Documentation

typedef int32_t vx_enum

Sets the standard enumeration type size to be a fixed quantity.

All enumerable fields must use this type as the container to enforce enumeration ranges and sizeof() operations.

Definition at line 160 of file vx_types.h.

typedef vx_enum vx_status

A formal status type with known fixed size.

See Also

vx_status_e

Definition at line 428 of file vx_types.h.

3.45.5 Enumeration Type Documentation

enum vx_bool

A Boolean value. This allows 0 to be FALSE, as it is in C, and any non-zero to be TRUE.

```
vx_bool ret = vx_true_e;
if (ret) printf("true!\n");
ret = vx_false_e;
if (!ret) printf("false!\n");
```

This would print both strings.

Enumerator

vx_false_e The "false" value.

vx_true_e The "true" value.

Definition at line 301 of file vx_types.h.

enum vx_type_e

The type enumeration lists all the known types in OpenVX.

Enumerator

VX_TYPE_INVALID An invalid type value. When passed an error must be returned.

VX_TYPE_CHAR A `vx_char`.

VX_TYPE_INT8 A `vx_int8`.

VX_TYPE_UINT8 A `vx_uint8`.

VX_TYPE_INT16 A `vx_int16`.

VX_TYPE_UINT16 A `vx_uint16`.

VX_TYPE_INT32 A `vx_int32`.

VX_TYPE_UINT32 A `vx_uint32`.

VX_TYPE_INT64 A `vx_int64`.

VX_TYPE_UINT64 A `vx_uint64`.

VX_TYPE_FLOAT32 A `vx_float32`.

VX_TYPE_FLOAT64 A `vx_float64`.

VX_TYPE_ENUM A `vx_enum`. Equivalent in size to a `vx_int32`.

VX_TYPE_SIZE A `vx_size`.

VX_TYPE_DF_IMAGE A `vx_df_image`.

VX_TYPE_BOOL A `vx_bool`.

VX_TYPE_SCALAR_MAX A floating value for comparison between OpenVX scalars and OpenVX structs.

VX_TYPE_RECTANGLE A `vx_rectangle_t`.

VX_TYPE_KEYPOINT A `vx_keypoint_t`.

VX_TYPE_COORDINATES2D A `vx_coordinates2d_t`.

VX_TYPE_COORDINATES3D A `vx_coordinates3d_t`.

VX_TYPE_USER_STRUCT_START A user-defined struct base index.

VX_TYPE_VENDOR_STRUCT_START A vendor-defined struct base index.

VX_TYPE_KHRONOS_OBJECT_START A Khronos defined object base index.

VX_TYPE_VENDOR_OBJECT_START A vendor defined object base index.

VX_TYPE_KHRONOS_STRUCT_MAX A value for comparison between Khronos defined structs and user structs.

VX_TYPE_USER_STRUCT_END A value for comparison between user structs and vendor structs.

VX_TYPE_VENDOR_STRUCT_END A value for comparison between vendor structs and Khronos defined objects.

VX_TYPE_KHRONOS_OBJECT_END A value for comparison between Khronos defined objects and vendor structs.

VX_TYPE_VENDOR_OBJECT_END A value used for bound checking of vendor objects.

VX_TYPE_REFERENCE A `vx_reference`.

VX_TYPE_CONTEXT A `vx_context`.

VX_TYPE_GRAPH A `vx_graph`.

VX_TYPE_NODE A `vx_node`.

VX_TYPE_KERNEL A `vx_kernel`.

VX_TYPE_PARAMETER A `vx_parameter`.

VX_TYPE_DELAY A `vx_delay`.

VX_TYPE_LUT A `vx_lut`.

VX_TYPE_DISTRIBUTION A `vx_distribution`.

VX_TYPE_PYRAMID A `vx_pyramid`.

VX_TYPE_THRESHOLD A `vx_threshold`.

VX_TYPE_MATRIX A `vx_matrix`.

VX_TYPE_CONVOLUTION A `vx_convolution`.

VX_TYPE_SCALAR A `vx_scalar`. when needed to be completely generic for kernel validation.

VX_TYPE_ARRAY A `vx_array`.

VX_TYPE_IMAGE A `vx_image`.

VX_TYPE_REMAP A `vx_remap`.

VX_TYPE_ERROR An error object which has no type.

VX_TYPE_META_FORMAT A `vx_meta_format`.

VX_TYPE_OBJECT_ARRAY A `vx_object_array`.

Definition at line 322 of file vx_types.h.

enum vx_status_e

The enumeration of all status codes.

See Also

vx_status.

Enumerator

VX_STATUS_MIN Indicates the lower bound of status codes in VX. Used for bounds checks only.

VX_ERROR_REFERENCE_NONZERO Indicates that an operation did not complete due to a reference count being non-zero.

VX_ERROR_MULTIPLE_WRITERS Indicates that the graph has more than one node outputting to the same data object. This is an invalid graph structure.

VX_ERROR_GRAPH_ABANDONED Indicates that the graph is stopped due to an error or a callback that abandoned execution.

VX_ERROR_GRAPH_SCHEDULED Indicates that the supplied graph already has been scheduled and may be currently executing.

VX_ERROR_INVALID_SCOPE Indicates that the supplied parameter is from another scope and cannot be used in the current scope.

VX_ERROR_INVALID_NODE Indicates that the supplied node could not be created.

VX_ERROR_INVALID_GRAPH Indicates that the supplied graph has invalid connections (cycles).

VX_ERROR_INVALID_TYPE Indicates that the supplied type parameter is incorrect.

VX_ERROR_INVALID_VALUE Indicates that the supplied parameter has an incorrect value.

VX_ERROR_INVALID_DIMENSION Indicates that the supplied parameter is too big or too small in dimension.

VX_ERROR_INVALID_FORMAT Indicates that the supplied parameter is in an invalid format.

VX_ERROR_INVALID_LINK Indicates that the link is not possible as specified. The parameters are incompatible.

VX_ERROR_INVALID_REFERENCE Indicates that the reference provided is not valid.

VX_ERROR_INVALID_MODULE This is returned from `vxLoadKernels` when the module does not contain the entry point.

VX_ERROR_INVALID_PARAMETERS Indicates that the supplied parameter information does not match the kernel contract.

VX_ERROR_OPTIMIZED_AWAY Indicates that the object refered to has been optimized out of existence.

VX_ERROR_NO_MEMORY Indicates that an internal or implicit allocation failed. Typically catastrophic. After detection, deconstruct the context.

See Also

vxVerifyGraph.

VX_ERROR_NO_RESOURCES Indicates that an internal or implicit resource can not be acquired (not memory). This is typically catastrophic. After detection, deconstruct the context.

See Also

vxVerifyGraph.

VX_ERROR_NOT_COMPATIBLE Indicates that the attempt to link two parameters together failed due to type incompatibilty.

VX_ERROR_NOT_ALLOCATED Indicates to the system that the parameter must be allocated by the system.

VX_ERROR_NOT_SUFFICIENT Indicates that the given graph has failed verification due to an insufficient number of required parameters, which cannot be automatically created. Typically this indicates required atomic parameters.

See Also

vxVerifyGraph.

VX_ERROR_NOT_SUPPORTED Indicates that the requested set of parameters produce a configuration that cannot be supported. Refer to the supplied documentation on the configured kernels.

See Also

vx_kernel_e. This is also returned if a function to set an attribute is called on a Read-only attribute.

VX_ERROR_NOT_IMPLEMENTED Indicates that the requested kernel is missing.

See Also

vx_kernel_e vxGetKernelByName.

VX_FAILURE Indicates a generic error code, used when no other describes the error.

VX_SUCCESS No error.

Definition at line 394 of file vx_types.h.

enum vx_enum_e

The set of supported enumerations in OpenVX.

These can be extracted from enumerated values using `VX_ENUM_TYPE`.

Enumerator

VX_ENUM_DIRECTION Parameter Direction.

VX_ENUM_ACTION Action Codes.

VX_ENUM_HINT Hint Values.

VX_ENUM_DIRECTIVE Directive Values.

VX_ENUM_INTERPOLATION Interpolation Types.

VX_ENUM_OVERFLOW Overflow Policies.

VX_ENUM_COLOR_SPACE Color Space.

VX_ENUM_COLOR_RANGE Color Space Range.

VX_ENUM_PARAMETER_STATE Parameter State.

VX_ENUM_CHANNEL Channel Name.

VX_ENUM_CONVERT_POLICY Convert Policy.

VX_ENUM_THRESHOLD_TYPE Threshold Type List.

VX_ENUM_BORDER Border Mode List.

VX_ENUM_COMPARISON Comparison Values.

VX_ENUM_MEMORY_TYPE The memory type enumeration.

VX_ENUM_TERM_CRITERIA A termination criteria.

VX_ENUM_NORM_TYPE A norm type.

VX_ENUM_ACCESSOR An accessor flag type.

VX_ENUM_ROUND_POLICY Rounding Policy.

VX_ENUM_TARGET Target.

VX_ENUM_BORDER_POLICY Unsupported Border Mode Policy List.

VX_ENUM_GRAPH_STATE Graph attribute states.

VX_ENUM_NONLINEAR Non-linear function list.

VX_ENUM_PATTERN Matrix pattern enumeration.

Definition at line 539 of file vx_types.h.

enum vx_convert_policy_e

The Conversion Policy Enumeration.

Enumerator

VX_CONVERT_POLICY_WRAP Results are the least significant bits of the output operand, as if stored in two's complement binary format in the size of its bit-depth.

VX_CONVERT_POLICY_SATURATE Results are saturated to the bit depth of the output operand.

Definition at line 663 of file vx_types.h.

enum vx_df_image_e

Based on the VX_DF_IMAGE definition.

Note

Use `vx_df_image` to contain these values.

Enumerator

VX_DF_IMAGE_VIRT A virtual image of no defined type.

VX_DF_IMAGE_RGB A single plane of 24-bit pixel as 3 interleaved 8-bit units of R then G then B data. This uses the BT709 full range by default.

VX_DF_IMAGE_RGBX A single plane of 32-bit pixel as 4 interleaved 8-bit units of R then G then B data, then a *don't care* byte. This uses the BT709 full range by default.

VX_DF_IMAGE_NV12 A 2-plane YUV format of Luma (Y) and interleaved UV data at 4:2:0 sampling. This uses the BT709 full range by default.

VX_DF_IMAGE_NV21 A 2-plane YUV format of Luma (Y) and interleaved VU data at 4:2:0 sampling. This uses the BT709 full range by default.

VX_DF_IMAGE_UYVY A single plane of 32-bit macro pixel of U0, Y0, V0, Y1 bytes. This uses the BT709 full range by default.

VX_DF_IMAGE_YUYV A single plane of 32-bit macro pixel of Y0, U0, Y1, V0 bytes. This uses the BT709 full range by default.

VX_DF_IMAGE_IYUV A 3 plane of 8-bit 4:2:0 sampled Y, U, V planes. This uses the BT709 full range by default.

VX_DF_IMAGE_YUV4 A 3 plane of 8 bit 4:4:4 sampled Y, U, V planes. This uses the BT709 full range by default.

VX_DF_IMAGE_U8 A single plane of unsigned 8-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

VX_DF_IMAGE_U16 A single plane of unsigned 16-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

VX_DF_IMAGE_S16 A single plane of signed 16-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

VX_DF_IMAGE_U32 A single plane of unsigned 32-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

VX_DF_IMAGE_S32 A single plane of unsigned 32-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

Definition at line 676 of file vx_types.h.

enum vx_target_e

The Target Enumeration.

Enumerator

VX_TARGET_ANY Any available target. An OpenVX implementation must support at least one target associated with this value.

VX_TARGET_STRING Target, explicitly specified by its (case-insensitive) name string.

VX_TARGET_VENDOR_BEGIN Start of Vendor specific target enumerates.

Definition at line 742 of file vx_types.h.

enum vx_channel_e

The channel enumerations for channel extractions.

See Also

vxChannelExtractNode
vxuChannelExtract
VX_KERNEL_CHANNEL_EXTRACT

Enumerator

VX_CHANNEL_0 Used by formats with unknown channel types.

VX_CHANNEL_1 Used by formats with unknown channel types.

VX_CHANNEL_2 Used by formats with unknown channel types.

VX_CHANNEL_3 Used by formats with unknown channel types.

VX_CHANNEL_R Use to extract the RED channel, no matter the byte or packing order.

VX_CHANNEL_G Use to extract the GREEN channel, no matter the byte or packing order.

VX_CHANNEL_B Use to extract the BLUE channel, no matter the byte or packing order.

VX_CHANNEL_A Use to extract the ALPHA channel, no matter the byte or packing order.

VX_CHANNEL_Y Use to extract the LUMA channel, no matter the byte or packing order.

VX_CHANNEL_U Use to extract the Cb/U channel, no matter the byte or packing order.

VX_CHANNEL_V Use to extract the Cr/V/Value channel, no matter the byte or packing order.

Definition at line 1110 of file vx_types.h.

enum vx_interpolation_type_e

The image reconstruction filters supported by image resampling operations.

The edge of a pixel is interpreted as being aligned to the edge of the image. The value for an output pixel is evaluated at the center of that pixel.

This means, for example, that an even enlargement of a factor of two in nearest-neighbor interpolation will replicate every source pixel into a 2x2 quad in the destination, and that an even shrink by a factor of two in bilinear interpolation will create each destination pixel by average a 2x2 quad of source pixels.

Samples that cross the boundary of the source image have values determined by the border mode - see `vx_border_e` and `VX_NODE_BORDER`.

See Also

vxuScaleImage
vxScaleImageNode
VX_KERNEL_SCALE_IMAGE
vxuWarpAffine
vxWarpAffineNode
VX_KERNEL_WARP_AFFINE
vxuWarpPerspective
vxWarpPerspectiveNode
VX_KERNEL_WARP_PERSPECTIVE

Enumerator

VX_INTERPOLATION_NEAREST_NEIGHBOR Output values are defined to match the source pixel whose center is nearest to the sample position.

VX_INTERPOLATION_BILINEAR Output values are defined by bilinear interpolation between the pixels whose centers are closest to the sample position, weighted linearly by the distance of the sample from the pixel centers.

VX_INTERPOLATION_AREA Output values are determined by averaging the source pixels whose areas fall under the area of the destination pixel, projected onto the source image.

Definition at line 1170 of file vx_types.h.

enum vx_non_linear_filter_e

An enumeration of non-linear filter functions.

Enumerator

VX_NONLINEAR_FILTER_MEDIAN Nonlinear median filter.

VX_NONLINEAR_FILTER_MIN Nonlinear Erode.

VX_NONLINEAR_FILTER_MAX Nonlinear Dilate.

Definition at line 1184 of file vx_types.h.

enum vx_pattern_e

An enumeration of matrix patterns. See `vxCreateMatrixFromPattern`

Enumerator

VX_PATTERN_BOX Box pattern matrix.

VX_PATTERN_CROSS Cross pattern matrix.

VX_PATTERN_DISK A square matrix (rows = columns = size)

VX_PATTERN_OTHER Matrix with any pattern othern than above.

Definition at line 1196 of file vx_types.h.

enum vx_vendor_id_e

The Vendor ID of the Implementation. As new vendors submit their implementations, this enumeration will grow.

Enumerator

VX_ID_KHRONOS The Khronos Group.

VX_ID_TI Texas Instruments, Inc.

VX_ID_QUALCOMM Qualcomm, Inc.

VX_ID_NVIDIA NVIDIA Corporation.

VX_ID_ARM ARM Ltd.

VX_ID_BDTI Berkley Design Technology, Inc.

VX_ID_RENESAS Renasas Electronics.

VX_ID_VIVANTE Vivante Corporation.

VX_ID_XILINX Xilinx Inc.

VX_ID_AXIS Axis Communications.

VX_ID_MOVIDIUS Movidius Ltd.

VX_ID_SAMSUNG Samsung Electronics.

VX_ID_FREESCALE Freescale Semiconductor.

VX_ID_AMD Advanced Micro Devices.

VX_ID_BROADCOM Broadcom Corporation.

VX_ID_INTEL Intel Corporation.

VX_ID_MARVELL Marvell Technology Group Ltd.

VX_ID_MEDIATEK MediaTek, Inc.

VX_ID_ST STMicroelectronics.

VX_ID_CEVA CEVA DSP.

VX_ID_ITSEEZ Itseez, Inc.

VX_ID_IMAGINATION Imagination Technologies.

VX_ID_NXP NXP Semiconductors.

VX_ID_VIDEANTIS Videantis.

VX_ID_SYNOPSYS Synopsys.

VX_ID_CADENCE Cadence.

VX_ID_HUAWEI Huawei.

VX_ID_USER For use by vxAllocateUserKernelId and vxAllocateUserKernelLibraryId.

VX_ID_DEFAULT For use by all Kernel authors until they can obtain an assigned ID.

Definition at line 36 of file vx_vendors.h.

3.45.6 Function Documentation

vx_status VX_API_CALL vxGetStatus (vx_reference *reference*)

Provides a generic API to return status values from Object constructors if they fail.

Note

Users do not need to strictly check every object creator as the errors should properly propagate and be detected during verification time or run-time.

```
vx_image img = vxCreateImage(context, 639, 480,
      VX_DF_IMAGE_UYVY);
vx_status status = vxGetStatus((vx_reference)img);
// status == VX_ERROR_INVALID_DIMENSIONS
vxReleaseImage(&img);
```

Precondition

Appropriate Object Creator function.

Postcondition

Appropriate Object Release function.

Parameters

in	*reference*	The reference to check for construction errors.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No error.
*	Some error occurred, please check enumeration list and constructor.

3.46 Objects

3.46.1 Detailed Description

Defines the basic objects within OpenVX. All objects in OpenVX derive from a `vx_reference` and contain a reference to the `vx_context` from which they were made, except the `vx_context` itself.

Modules

- Object: Reference

  Defines the Reference Object interface.
- Object: Context

  Defines the Context Object Interface.
- Object: Graph

  Defines the Graph Object interface.
- Object: Node

  Defines the Node Object interface.
- Object: Array

  Defines the Array Object Interface.
- Object: Convolution

  Defines the Image Convolution Object interface.
- Object: Distribution

  Defines the Distribution Object Interface.
- Object: Image

  Defines the Image Object interface.
- Object: LUT

  Defines the Look-Up Table Interface.
- Object: Matrix

  Defines the Matrix Object Interface.
- Object: Pyramid

  Defines the Image Pyramid Object Interface.
- Object: Remap

  Defines the Remap Object Interface.
- Object: Scalar

  Defines the Scalar Object interface.
- Object: Threshold

  Defines the Threshold Object Interface.
- Object: ObjectArray

  An opaque array object that could be an array of any data-object (not data-type) of OpenVX except Delay and ObjectArray objects.

3.47 Object: Reference

3.47.1 Detailed Description

Defines the Reference Object interface. All objects in OpenVX are derived (in the object-oriented sense) from `vx_reference`. All objects shall be able to be cast back to this type safely.

Macros

- #define VX_MAX_REFERENCE_NAME (64)

  Defines the length of the reference name string, including the trailing zero.

Typedefs

- typedef struct _vx_reference ∗ vx_reference

  A generic opaque reference to any object within OpenVX.

Enumerations

- enum vx_reference_attribute_e {
  VX_REF_ATTRIBUTE_COUNT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REFERENCE << 8)) + 0x0,
  VX_REF_ATTRIBUTE_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REFERENCE << 8)) + 0x1,
  VX_REF_ATTRIBUTE_NAME = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REFERENCE << 8)) + 0x2
  }

  The reference attributes list.

Functions

- vx_status VX_API_CALL vxQueryReference (vx_reference ref, vx_enum attribute, void ∗ptr, vx_size size)

  Queries any reference type for some basic information like count or type.
- vx_status VX_API_CALL vxReleaseReference (vx_reference ∗ref_ptr)

  Releases a reference. The reference may potentially refer to multiple OpenVX objects of different types. This function can be used instead of calling a specific release function for each individual object type (e.g. vxRelease<object>). The object will not be destroyed until its total reference count is zero.
- vx_status VX_API_CALL vxRetainReference (vx_reference ref)

  Increments the reference counter of an object This function is used to express the fact that the OpenVX object is referenced multiple times by an application. Each time this function is called for an object, the application will need to release the object one additional time before it can be destructed.
- vx_status VX_API_CALL vxSetReferenceName (vx_reference ref, const vx_char ∗name)

  Name a reference
  This function is used to associate a name to a referenced object. This name can be used by the OpenVX implementation in log messages and any other reporting mechanisms.

3.47.2 Macro Definition Documentation

#define VX_MAX_REFERENCE_NAME (64)

Defines the length of the reference name string, including the trailing zero.

See Also

vxSetReferenceName

Definition at line 56 of file vx.h.

3.47.3 Typedef Documentation

typedef struct _vx_reference∗ vx_reference

A generic opaque reference to any object within OpenVX.

A user of OpenVX should not assume that this can be cast directly to anything; however, any object in OpenVX can be cast back to this for the purposes of querying attributes of the object or for passing the object as a parameter to functions that take a `vx_reference` type. If the API does not take that specific type but may take others, an error may be returned from the API.

Definition at line 153 of file vx_types.h.

3.47.4 Enumeration Type Documentation

enum vx_reference_attribute_e

The reference attributes list.

Enumerator

VX_REF_ATTRIBUTE_COUNT Returns the reference count of the object. Read-only. Use a `vx_uint32` parameter.

VX_REF_ATTRIBUTE_TYPE Returns the `vx_type_e` of the reference. Read-only. Use a `vx_enum` parameter.

VX_REF_ATTRIBUTE_NAME Used to query the reference for its name. Read-write. Use a ∗ `vx_char` parameter.

Definition at line 754 of file vx_types.h.

3.47.5 Function Documentation

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

Queries any reference type for some basic information like count or type.

Parameters

`in`	*ref*	The reference to query.
`in`	*attribute*	The value for which to query. Use `vx_reference_attribute_e`.
`out`	*ptr*	The location at which to store the resulting value.
`in`	*size*	The size in bytes of the container to which ptr points.

Returns

A `vx_status_e` enumeration.

vx_status VX_API_CALL vxReleaseReference (vx_reference ∗ *ref_ptr*)

Releases a reference. The reference may potentially refer to multiple OpenVX objects of different types. This function can be used instead of calling a specific release function for each individual object type (e.g. vxRelease<object>). The object will not be destroyed until its total reference count is zero.

Note

After returning from this function the reference is zeroed.

Parameters

in	*ref_ptr*	The pointer to the reference of the object to release.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the reference is not valid.

vx_status VX_API_CALL vxRetainReference (vx_reference *ref*)

Increments the reference counter of an object This function is used to express the fact that the OpenVX object is referenced multiple times by an application. Each time this function is called for an object, the application will need to release the object one additional time before it can be destructed.

Parameters

in	*ref*	The reference to retain.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	if reference is not valid.

vx_status VX_API_CALL vxSetReferenceName (vx_reference *ref*, const vx_char ∗ *name*)

Name a reference

This function is used to associate a name to a referenced object. This name can be used by the OpenVX implementation in log messages and any other reporting mechanisms.

The OpenVX implementation will not check if the name is unique in the reference scope (context or graph). Several references can then have the same name.

Parameters

in	*ref*	The reference to the object to be named.
in	*name*	Pointer to the '\0' terminated string that identifies the referenced object. The string is copied by the function so that it stays the property of the caller. NULL means that the reference is not named. The length of the string shall be lower than VX_MAX_REFERENCE_NAME bytes.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If reference is not valid.

3.48 Object: Context

3.48.1 Detailed Description

Defines the Context Object Interface. The OpenVX context is the object domain for all OpenVX objects. All data objects *live* in the context as well as all framework objects. The OpenVX context keeps reference counts on all objects and must do garbage collection during its deconstruction to free lost references. While multiple clients may connect to the OpenVX context, all data are private in that the references referring to data objects are given only to the creating party.

Macros

- #define VX_MAX_IMPLEMENTATION_NAME (64)

  Defines the length of the implementation name string, including the trailing zero.

Typedefs

- typedef struct _vx_context ∗ vx_context

  An opaque reference to the implementation context.

Enumerations

- enum vx_accessor_e {
  VX_READ_ONLY = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACCESSOR << 12)) + 0x1,
  VX_WRITE_ONLY = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACCESSOR << 12)) + 0x2,
  VX_READ_AND_WRITE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACCESSOR << 12)) + 0x3 }

  *The memory accessor hint flags. These enumeration values are used to indicate desired system behavior, not the **User** intent. For example: these can be interpretted as hints to the system about cache operations or marshalling operations.*
- enum vx_context_attribute_e {
  VX_CONTEXT_VENDOR_ID = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x0,
  VX_CONTEXT_VERSION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x1,
  VX_CONTEXT_UNIQUE_KERNELS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x2,
  VX_CONTEXT_MODULES = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x3,
  VX_CONTEXT_REFERENCES = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x4,
  VX_CONTEXT_IMPLEMENTATION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x5,
  VX_CONTEXT_EXTENSIONS_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x6,
  VX_CONTEXT_EXTENSIONS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x7,
  VX_CONTEXT_CONVOLUTION_MAX_DIMENSION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x8,
  VX_CONTEXT_OPTICAL_FLOW_MAX_WINDOW_DIMENSION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x9,
  VX_CONTEXT_IMMEDIATE_BORDER = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0xA,
  VX_CONTEXT_UNIQUE_KERNEL_TABLE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0xB,
  VX_CONTEXT_IMMEDIATE_BORDER_POLICY = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0xC,
  VX_CONTEXT_NONLINEAR_MAX_DIMENSION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0xd }

  A list of context attributes.
- enum vx_memory_type_e {
  VX_MEMORY_TYPE_NONE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_MEMORY_TYPE << 12)) +

0x0,
VX_MEMORY_TYPE_HOST = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_MEMORY_TYPE << 12)) + 0x1 }

An enumeration of memory import types.

- enum vx_round_policy_e {
VX_ROUND_POLICY_TO_ZERO = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ROUND_POLICY << 12)) + 0x1,
VX_ROUND_POLICY_TO_NEAREST_EVEN = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ROUND_POLICY << 12)) + 0x2 }

  The Round Policy Enumeration.
- enum vx_termination_criteria_e {
VX_TERM_CRITERIA_ITERATIONS = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_TERM_CRITERIA << 12)) + 0x0,
VX_TERM_CRITERIA_EPSILON = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_TERM_CRITERIA << 12)) + 0x1,
VX_TERM_CRITERIA_BOTH = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_TERM_CRITERIA << 12)) + 0x2 }

  The termination criteria list.

Functions

- vx_context VX_API_CALL vxCreateContext ()

  Creates a `vx_context`.
- vx_context VX_API_CALL vxGetContext (vx_reference reference)

  Retrieves the context from any reference from within a context.
- vx_status VX_API_CALL vxQueryContext (vx_context context, vx_enum attribute, void *ptr, vx_size size)

  Queries the context for some specific information.
- vx_status VX_API_CALL vxReleaseContext (vx_context *context)

  Releases the OpenVX object context.
- vx_status VX_API_CALL vxSetContextAttribute (vx_context context, vx_enum attribute, const void *ptr, vx_size size)

  Sets an attribute on the context.
- vx_status VX_API_CALL vxSetImmediateModeTarget (vx_context context, vx_enum target_enum, const char *target_string)

  Sets the default target of the immediate mode. Upon successful execution of this function any future execution of immediate mode function is attempted on the new default target of the context.

3.48.2 Typedef Documentation

typedef struct _vx_context* vx_context

An opaque reference to the implementation context.

See Also

vxCreateContext

Definition at line 226 of file vx_types.h.

3.48.3 Enumeration Type Documentation

enum vx_context_attribute_e

A list of context attributes.

Enumerator

VX_CONTEXT_VENDOR_ID Queries the unique vendor ID. Read-only. Use a `vx_uint16`.

VX_CONTEXT_VERSION Queries the OpenVX Version Number. Read-only. Use a `vx_uint16`

VX_CONTEXT_UNIQUE_KERNELS Queries the context for the number of *unique* kernels. Read-only. Use a `vx_uint32` parameter.

VX_CONTEXT_MODULES Queries the context for the number of active modules. Read-only. Use a `vx_uint32` parameter.

VX_CONTEXT_REFERENCES Queries the context for the number of active references. Read-only. Use a `vx_uint32` parameter.

VX_CONTEXT_IMPLEMENTATION Queries the context for it's implementation name. Read-only. Use a `vx_char`[`VX_MAX_IMPLEMENTATION_NAME`] array.

VX_CONTEXT_EXTENSIONS_SIZE Queries the number of bytes in the extensions string. Read-only. Use a `vx_size` parameter.

VX_CONTEXT_EXTENSIONS Retrieves the extensions string. Read-only. This is a space-separated string of extension names. Each OpenVX official extension has a unique identifier, comprised of capital letters, numbers and the underscore character, prefixed with "KHR_", for example "KHR_NEW_FEATURE". Use a `vx_char` pointer allocated to the size returned from `VX_CONTEXT_EXTENSIONS_SIZE`.

VX_CONTEXT_CONVOLUTION_MAX_DIMENSION The maximum width or height of a convolution matrix. Read-only. Use a `vx_size` parameter. Each vendor must support centered kernels of size w X h, where both w and h are odd numbers, 3 <= w <= n and 3 <= h <= n, where n is the value of the `VX_CONTEXT_CONVOLUTION_MAX_DIMENSION` attribute. n is an odd number that should not be smaller than 9. w and h may or may not be equal to each other. All combinations of w and h meeting the conditions above must be supported. The behavior of `vxCreateConvolution` is undefined for values larger than the value returned by this attribute.

VX_CONTEXT_OPTICAL_FLOW_MAX_WINDOW_DIMENSION The maximum window dimension of the OpticalFlowPyrLK kernel. The value of this attribute shall be equal to or greater than '9'.

See Also

`VX_KERNEL_OPTICAL_FLOW_PYR_LK`. Read-only. Use a `vx_size` parameter.

VX_CONTEXT_IMMEDIATE_BORDER The border mode for immediate mode functions. Graph mode functions are unaffected by this attribute. Read-write. Use a pointer to a `vx_border_t` structure as parameter.

Note

The assumed default value for immediate mode functions is `VX_BORDER_UNDEFINED`.

VX_CONTEXT_UNIQUE_KERNEL_TABLE Returns the table of all unique the kernels that exist in the context. Read-only. Use a `vx_kernel_info_t` array.

Precondition

You must call `vxQueryContext` with `VX_CONTEXT_UNIQUE_KERNELS` to compute the necessary size of the array.

VX_CONTEXT_IMMEDIATE_BORDER_POLICY The unsupported border mode policy for immediate mode functions. Read-only. Graph mode functions are unaffected by this attribute. Use a `vx_enum` as parameter.

Note

The assumed default value for immediate mode functions is `VX_BORDER_POLICY_DEFAULT_TO_UNDEFINED`.

VX_CONTEXT_NONLINEAR_MAX_DIMENSION The dimension of the largest nonlinear filter supported. See `vxNonLinearFilterNode`. The implementation must support all dimensions (height or width, not necessarily the same) up to the value of this attribute. The lowest value that must be supported for this attribute is 9. Read-only. Use a `vx_size` parameter.

Definition at line 766 of file vx_types.h.

enum vx_memory_type_e

An enumeration of memory import types.

Enumerator

VX_MEMORY_TYPE_NONE For memory allocated through OpenVX, this is the import type.

VX_MEMORY_TYPE_HOST The default memory type to import from the Host.

Definition at line 1139 of file vx_types.h.

enum vx_termination_criteria_e

The termination criteria list.

See Also

Optical Flow Pyramid (LK)

Enumerator

VX_TERM_CRITERIA_ITERATIONS Indicates a termination after a set number of iterations.

VX_TERM_CRITERIA_EPSILON Indicates a termination after matching against the value of eplison provided to the function.

VX_TERM_CRITERIA_BOTH Indicates that both an iterations and eplison method are employed. Whichever one matches first causes the termination.

Definition at line 1278 of file vx_types.h.

enum vx_accessor_e

The memory accessor hint flags. These enumeration values are used to indicate desired *system* behavior, not the **User** intent. For example: these can be interpretted as hints to the system about cache operations or marshalling operations.

Enumerator

VX_READ_ONLY The memory shall be treated by the system as if it were read-only. If the User writes to this memory, the results are implementation defined.

VX_WRITE_ONLY The memory shall be treated by the system as if it were write-only. If the User reads from this memory, the results are implementation defined.

VX_READ_AND_WRITE The memory shall be treated by the system as if it were readable and writeable.

Definition at line 1316 of file vx_types.h.

enum vx_round_policy_e

The Round Policy Enumeration.

Enumerator

VX_ROUND_POLICY_TO_ZERO When scaling, this truncates the least significant values that are lost in operations.

VX_ROUND_POLICY_TO_NEAREST_EVEN When scaling, this rounds to nearest even output value.

Definition at line 1333 of file vx_types.h.

3.48.4 Function Documentation

vx_context VX_API_CALL vxCreateContext ( )

Creates a vx_context.

This creates a top-level object context for OpenVX.

Note

This is required to do anything else.

Returns

The reference to the implementation context vx_context. Any possible errors preventing a successful creation should be checked using vxGetStatus.

Postcondition

vxReleaseContext

vx_status VX_API_CALL vxReleaseContext (vx_context * *context*)

Releases the OpenVX object context.

All reference counted objects are garbage-collected by the return of this call. No calls are possible using the parameter context after the context has been released until a new reference from vxCreateContext is returned. All outstanding references to OpenVX objects from this context are invalid after this call.

Parameters

in	*context*	The pointer to the reference to the context.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If context is not a vx_context.

Precondition

vxCreateContext

vx_context VX_API_CALL vxGetContext (vx_reference *reference*)

Retrieves the context from any reference from within a context.

Parameters

in	*reference*	The reference from which to extract the context.

Returns

The overall context that created the particular reference. Any possible errors preventing a successful creation should be checked using vxGetStatus.

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

Queries the context for some specific information.

Parameters

in	*context*	The reference to the context.
in	*attribute*	The attribute to query. Use a vx_context_attribute_e.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the context is not a vx_context.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.
VX_ERROR_NOT_SUPPORTED	If the attribute is not supported on this implementation.

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

Sets an attribute on the context.

Parameters

in	*context*	The handle to the overall context.
in	*attribute*	The attribute to set from vx_context_attribute_e.
in	*ptr*	The pointer to the data to which to set the attribute.
in	*size*	The size in bytes of the data to which *ptr* points.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the context is not a vx_context.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.
VX_ERROR_NOT_SUPPORTED	If the attribute is not settable.

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

Sets the default target of the immediate mode. Upon successful execution of this function any future execution of immediate mode function is attempted on the new default target of the context.

Parameters

in	*context*	The reference to the implementation context.
in	*target_enum*	The default immediate mode target enum to be set to the vx_context object. Use a vx_target_e.

in	*target_string*	The target name ASCII string. This contains a valid value when target_enum is set to `VX_TARGET_STRING`, otherwise it is ignored.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Default target set.
VX_ERROR_INVALID_REFERENCE	If the context is not a `vx_context`.
VX_ERROR_NOT_SUPPORTED	If the specified target is not supported in this context.

3.49 Object: Graph

3.49.1 Detailed Description

Defines the Graph Object interface. A set of nodes connected in a directed (only goes one-way) acyclic (does not loop back) fashion. A Graph may have sets of Nodes that are unconnected to other sets of Nodes within the same Graph. See Graph Formalisms. Figure below shows the Graph state transition diagram. Also see `vx_graph_state_e`.

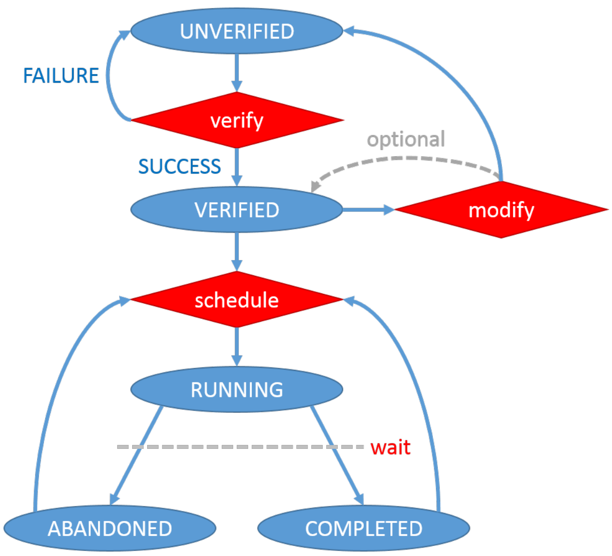


Figure 3.1: Graph State Transition

Typedefs

- typedef struct _vx_graph ∗ vx_graph

  An opaque reference to a graph.

Enumerations

- enum vx_graph_attribute_e {
  VX_GRAPH_NUMNODES = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_GRAPH << 8)) + 0x0,
  VX_GRAPH_PERFORMANCE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_GRAPH << 8)) + 0x2,
  VX_GRAPH_NUMPARAMETERS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_GRAPH << 8)) + 0x3,
  VX_GRAPH_STATE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_GRAPH << 8)) + 0x4 }

  The graph attributes list.
- enum vx_graph_state_e {
  VX_GRAPH_STATE_UNVERIFIED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_GRAPH_STATE << 12)) + 0x0,
  VX_GRAPH_STATE_VERIFIED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_GRAPH_STATE << 12)) + 0x1,
  VX_GRAPH_STATE_RUNNING = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_GRAPH_STATE << 12)) + 0x2,
  VX_GRAPH_STATE_ABANDONED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_GRAPH_STATE << 12)) + 0x3,
  VX_GRAPH_STATE_COMPLETED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_GRAPH_STATE << 12)) + 0x4 }

  The Graph State Enumeration.

Functions

- vx_graph VX_API_CALL vxCreateGraph (vx_context context)

    Creates an empty graph.
- vx_bool VX_API_CALL vxIsGraphVerified (vx_graph graph)

    Returns a Boolean to indicate the state of graph verification.
- vx_status VX_API_CALL vxProcessGraph (vx_graph graph)

    This function causes the synchronous processing of a graph. If the graph has not been verified, then the implementation verifies the graph immediately. If verification fails this function returns a status identical to what `vxVerifyGraph` would return. After the graph verfies successfully then processing occurs. If the graph was previously verified via `vxVerifyGraph` or `vxProcessGraph` then the graph is processed. This function blocks until the graph is completed.
- vx_status VX_API_CALL vxQueryGraph (vx_graph graph, vx_enum attribute, void *ptr, vx_size size)

    Allows the user to query attributes of the Graph.
- vx_status VX_API_CALL vxRegisterAutoAging (vx_graph graph, vx_delay delay)

    Register a delay for auto-aging.
- vx_status VX_API_CALL vxReleaseGraph (vx_graph *graph)

    Releases a reference to a graph. The object may not be garbage collected until its total reference count is zero. Once the reference count is zero, all node references in the graph are automatically released as well. Data referenced by those nodes may not be released as the user may have external references to the data.
- vx_status VX_API_CALL vxScheduleGraph (vx_graph graph)

    Schedules a graph for future execution.
- vx_status VX_API_CALL vxSetGraphAttribute (vx_graph graph, vx_enum attribute, const void *ptr, vx_size size)

    Allows the attributes of the Graph to be set to the provided value.
- vx_status VX_API_CALL vxVerifyGraph (vx_graph graph)

    Verifies the state of the graph before it is executed. This is useful to catch programmer errors and contract errors. If not verified, the graph verifies before being processed.
- vx_status VX_API_CALL vxWaitGraph (vx_graph graph)

    Waits for a specific graph to complete. If the graph has been scheduled multiple times since the last call to vxWaitGraph, then vxWaitGraph returns only when the last scheduled execution completes.

3.49.2 Typedef Documentation

typedef struct _vx_graph* vx_graph

An opaque reference to a graph.

See Also

vxCreateGraph

Definition at line 219 of file vx_types.h.

3.49.3 Enumeration Type Documentation

enum vx_graph_state_e

The Graph State Enumeration.

Enumerator

VX_GRAPH_STATE_UNVERIFIED The graph should be verified before execution.

VX_GRAPH_STATE_VERIFIED The graph has been verified and has not been executed or scheduled for execution yet.

VX_GRAPH_STATE_RUNNING The graph either has been scheduled and not completed, or is being executed.

VX_GRAPH_STATE_ABANDONED The graph execution was abandoned.

VX_GRAPH_STATE_COMPLETED The graph execution is completed and the graph is not scheduled for execution.

Definition at line 630 of file vx_types.h.

enum vx_graph_attribute_e

The graph attributes list.

Enumerator

VX_GRAPH_NUMNODES Returns the number of nodes in a graph. Read-only. Use a `vx_uint32` parameter.

VX_GRAPH_PERFORMANCE Returns the overall performance of the graph. Read-only. Use a `vx_perf_t` parameter. The accuracy of timing information is platform dependent.

Note

Performance tracking must have been enabled. See `vx_directive_e`

VX_GRAPH_NUMPARAMETERS Returns the number of explicitly declared parameters on the graph. Read-only. Use a `vx_uint32` parameter.

VX_GRAPH_STATE Returns the state of the graph. See `vx_graph_state_e` enum.

Definition at line 646 of file vx_types.h.

3.49.4 Function Documentation

vx_graph VX_API_CALL vxCreateGraph (vx_context *context*)

Creates an empty graph.

Parameters

in	*context*	The reference to the implementation context.

Returns

A graph reference `vx_graph`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

vx_status VX_API_CALL vxReleaseGraph (vx_graph * *graph*)

Releases a reference to a graph. The object may not be garbage collected until its total reference count is zero. Once the reference count is zero, all node references in the graph are automatically released as well. Data referenced by those nodes may not be released as the user may have external references to the data.

Parameters

in	*graph*	The pointer to the graph to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a `vx_graph`.

vx_status VX_API_CALL vxVerifyGraph (vx_graph *graph*)

Verifies the state of the graph before it is executed. This is useful to catch programmer errors and contract errors. If not verified, the graph verifies before being processed.

Precondition

Memory for data objects is not guarenteed to exist before this call.

Postcondition

After this call data objects exist unless the implementation optimized them out.

Parameters

in	*graph*	The reference to the graph to verify.

Returns

A status code for graphs with more than one error; it is undefined which error will be returned. Register a log callback using `vxRegisterLogCallback` to receive each specific error in the graph.
A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a `vx_graph`.
VX_ERROR_MULTIPLE_WRITERS	If the graph contains more than one writer to any data object.
VX_ERROR_INVALID_NODE	If a node in the graph is invalid or failed be created.
VX_ERROR_INVALID_GRAPH	If the graph contains cycles or some other invalid topology.
VX_ERROR_INVALID_TYPE	If any parameter on a node is given the wrong type.
VX_ERROR_INVALID_VALUE	If any value of any parameter is out of bounds of specification.
VX_ERROR_INVALID_FORMAT	If the image format is not compatible.

See Also

vxProcessGraph

vx_status VX_API_CALL vxProcessGraph (vx_graph *graph*)

This function causes the synchronous processing of a graph. If the graph has not been verified, then the implementation verifies the graph immediately. If verification fails this function returns a status identical to what `vxVerifyGraph` would return. After the graph verfies successfully then processing occurs. If the graph was previously verified via `vxVerifyGraph` or `vxProcessGraph` then the graph is processed. This function blocks until the graph is completed.

Parameters

in	*graph*	The graph to execute.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Graph has been processed.
VX_FAILURE	A catastrophic error occurred during processing.
*	See `vxVerifyGraph`.

Precondition

`vxVerifyGraph` must return `VX_SUCCESS` before this function will pass.

See Also

vxVerifyGraph

vx_status VX_API_CALL vxScheduleGraph (vx_graph *graph*)

Schedules a graph for future execution.

Parameters

`in`	*graph*	The graph to schedule.

Returns

A `vx_status_e` enumeration.

Return values

VX_ERROR_NO_RESOURCES	The graph cannot be scheduled now.
VX_ERROR_NOT_SUFFICIENT	The graph is not verified and has failed forced verification.
VX_SUCCESS	The graph has been scheduled.

Precondition

`vxVerifyGraph` must return `VX_SUCCESS` before this function will pass.

vx_status VX_API_CALL vxWaitGraph (vx_graph *graph*)

Waits for a specific graph to complete. If the graph has been scheduled multiple times since the last call to vxWaitGraph, then vxWaitGraph returns only when the last scheduled execution completes.

Parameters

`in`	*graph*	The graph to wait on.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	The graph has successfully completed execution and its outputs are the valid results of the most recent execution.
VX_FAILURE	An error occurred or the graph was never scheduled. Output data of the graph is undefined.

Precondition

`vxScheduleGraph`

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

Allows the user to query attributes of the Graph.

Parameters

in	*graph*	The reference to the created graph.
in	*attribute*	The vx_graph_attribute_e type needed.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

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

Allows the attributes of the Graph to be set to the provided value.

Parameters

in	*graph*	The reference to the graph.
in	*attribute*	The vx_graph_attribute_e type needed.
in	*ptr*	The location from which to read the value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_bool VX_API_CALL vxIsGraphVerified (vx_graph *graph*)

Returns a Boolean to indicate the state of graph verification.

Parameters

in	*graph*	The reference to the graph to check.

Returns

A vx_bool value.

Return values

vx_true_e	The graph is verified.
vx_false_e	The graph is not verified. It must be verified before execution either through vxVerifyGraph or automatically through vxProcessGraph or vxScheduleGraph.

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

Register a delay for auto-aging.

This function registers a delay object to be auto-aged by the graph. This delay object will be automatically aged after each successful completion of this graph. Aging of a delay object cannot be called during graph execution. A graph abandoned due to a node callback will trigger an auto-aging.

If a delay is registered for auto-aging multiple times in a same graph, the delay will be only aged a single time at each graph completion. If a delay is registered for auto-aging in multiple graphs, this delay will aged automatically after each successful completion of any of these graphs.

Parameters

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

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the `graph` or `delay` is not a valid reference

3.50 Object: Node

3.50.1 Detailed Description

Defines the Node Object interface. A node is an instance of a kernel that will be paired with a specific set of references (the parameters). Nodes are created from and associated with a single graph only. When a `vx_parameter` is extracted from a Node, an additional attribute can be accessed:

- *Reference* - The `vx_reference` assigned to this parameter index from the Node creation function (e.g., `vxSobel3x3Node`).

Typedefs

- typedef struct _vx_node ∗ vx_node

    An opaque reference to a kernel node.

Enumerations

- enum vx_node_attribute_e {
VX_NODE_STATUS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x0,
VX_NODE_PERFORMANCE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x1,
VX_NODE_BORDER = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x2,
VX_NODE_LOCAL_DATA_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x3,
VX_NODE_LOCAL_DATA_PTR = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x4,
VX_NODE_PARAMETERS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x5,
VX_NODE_IS_REPLICATED = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x6,
VX_NODE_REPLICATE_FLAGS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x7,
VX_NODE_VALID_RECT_RESET = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x8 }

    The node attributes list.

Functions

- vx_status VX_API_CALL vxQueryNode (vx_node node, vx_enum attribute, void ∗ptr, vx_size size)

    Allows a user to query information out of a node.
- vx_status VX_API_CALL vxReleaseNode (vx_node ∗node)

    Releases a reference to a Node object. The object may not be garbage collected until its total reference count is zero.
- vx_status VX_API_CALL vxRemoveNode (vx_node ∗node)

    Removes a Node from its parent Graph and releases it.
- vx_status VX_API_CALL vxReplicateNode (vx_graph graph, vx_node first_node, vx_bool replicate[ ], vx_uint32 number_of_parameters)

    Creates replicas of the same node first_node to process a set of objects stored in `vx_pyramid` or `vx_object_array`. first_node needs to have as parameter levels 0 of a `vx_pyramid` or the index 0 of a `vx_object_array`. Replica nodes are not accessible by the application through any means. An application request for removal of first_node from the graph will result in removal of all replicas. Any change of parameter or attribute of first_node will be propagated to the replicas. `vxVerifyGraph` shall enforce consistency of parameters and attributes in the replicas.
- vx_status VX_API_CALL vxSetNodeAttribute (vx_node node, vx_enum attribute, const void ∗ptr, vx_size size)

    Allows a user to set attribute of a node before Graph Validation.
- vx_status VX_API_CALL vxSetNodeTarget (vx_node node, vx_enum target_enum, const char ∗target_string)

    Sets the node target to the provided value. A success invalidates the graph that the node belongs to (`vxVerifyGraph` must be called before the next execution)

3.50.2 Typedef Documentation

typedef struct _vx_node∗ vx_node

An opaque reference to a kernel node.

See Also

vxCreateGenericNode

Definition at line 212 of file vx_types.h.

3.50.3 Enumeration Type Documentation

enum vx_node_attribute_e

The node attributes list.

Enumerator

VX_NODE_STATUS Queries the status of node execution. Read-only. Use a `vx_status` parameter.

VX_NODE_PERFORMANCE Queries the performance of the node execution. The accuracy of timing information is platform dependent and also depends on the graph optimizations. Read-only.

Note

Performance tracking must have been enabled. See `vx_directive_e`.

VX_NODE_BORDER Gets or sets the border mode of the node. Read-write. Use a `vx_border_t` structure with a default value of VX_BORDER_UNDEFINED.

VX_NODE_LOCAL_DATA_SIZE Indicates the size of the kernel local memory area. Read-only. Can be written only at user-node (de)initialization if VX_KERNEL_LOCAL_DATA_SIZE==0. Use a `vx_size` parameter.

VX_NODE_LOCAL_DATA_PTR Indicates the pointer kernel local memory area. Read-Write. Can be written only at user-node (de)initialization if VX_KERNEL_LOCAL_DATA_SIZE==0. Use a void ∗ parameter.

VX_NODE_PARAMETERS Indicates the number of node parameters, including optional parameters that are not passed. Read-only. Use a `vx_uint32` parameter.

VX_NODE_IS_REPLICATED Indicates whether the node is replicated. Read-only. Use a `vx_bool` parameter.

VX_NODE_REPLICATE_FLAGS Indicates the replicated parameters. Read-only. Use a `vx_bool`∗ parameter.

VX_NODE_VALID_RECT_RESET Indicates the behavior with respect to the valid rectangle. Read-only. Use a `vx_bool` parameter.

Definition at line 854 of file vx_types.h.

3.50.4 Function Documentation

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

Allows a user to query information out of a node.

Parameters

`in`	*node*	The reference to the node to query.
`in`	*attribute*	Use `vx_node_attribute_e` value to query for information.
`out`	*ptr*	The location at which to store the resulting value.
`in`	*size*	The size in bytesin bytes of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Successful
VX_ERROR_INVALID_PARAMETERS	The type or size is incorrect.

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

Allows a user to set attribute of a node before Graph Validation.

Parameters

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

Note

Some attributes are inherited from the `vx_kernel`, which was used to create the node. Some of these can be overridden using this API, notably VX_NODE_LOCAL_DATA_SIZE and VX_NODE_LOCAL_DATA_PTR.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	The attribute was set.
VX_ERROR_INVALID_REFERENCE	node is not a vx_node.
VX_ERROR_INVALID_PARAMETER	size is not correct for the type needed.

vx_status VX_API_CALL vxReleaseNode (vx_node * *node*)

Releases a reference to a Node object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*node*	The pointer to the reference of the node to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If node is not a `vx_node`.

vx_status VX_API_CALL vxRemoveNode (vx_node * *node*)

Removes a Node from its parent Graph and releases it.

Parameters

in	*node*	The pointer to the node to remove and release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If node is not a `vx_node`.

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

Sets the node target to the provided value. A success invalidates the graph that the node belongs to (`vxVerifyGraph` must be called before the next execution)

Parameters

in	*node*	The reference to the `vx_node` object.
in	*target_enum*	The target enum to be set to the `vx_node` object. Use a `vx_target_e`.
in	*target_string*	The target name ASCII string. This contains a valid value when target_enum is set to `VX_TARGET_STRING`, otherwise it is ignored.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Node target set.
VX_ERROR_INVALID_REFERENCE	If node is not a `vx_node`.
VX_ERROR_NOT_SUPPORTED	If the node kernel is not supported by the specified target.

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

Creates replicas of the same node first_node to process a set of objects stored in `vx_pyramid` or `vx_object_array`. first_node needs to have as parameter levels 0 of a `vx_pyramid` or the index 0 of a `vx_object_array`. Replica nodes are not accessible by the application through any means. An application request for removal of first_node from the graph will result in removal of all replicas. Any change of parameter or attribute of first_node will be propagated to the replicas. `vxVerifyGraph` shall enforce consistency of parameters and attributes in the replicas.

Parameters

in	*graph*	The reference to the graph.
in	*first_node*	The reference to the node in the graph that will be replicated.

in	*replicate*	an array of size equal to the number of node parameters, vx_true_e for the parameters that should be iterated over (should be a reference to a vx_pyramid or a vx_object_array), vx_false_e for the parameters that should be the same across replicated nodes and for optional parameters that are not used. Should be vx_true_e for all output and bidirectional parameters.
in	*number_of_-parameters*	number of elements in the replicate array

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the first_node is not a `vx_node`, or it is not the first child of a vx_pyramid.
VX_ERROR_NOT_COMPATIBLE	At least one of replicated parameters is not of level 0 of a pyramid or at index 0 of an object array.
VX_FAILURE	If the node does not belong to the graph, or the number of objects in the parent objects of inputs and output are not the same.

3.51 Object: Array

3.51.1 Detailed Description

Defines the Array Object Interface. Array is a strongly-typed container, which provides random access by index to its elements in constant time. It uses value semantics for its own elements and holds copies of data. This is an example `for` loop over an Array:

```
vx_size i, stride = sizeof(vx_size);
void *base = NULL;
vx_map_id map_id;
/* access entire array at once */
vxMapArrayRange(array, 0, num_items, &map_id, &stride, &base,
VX_READ_AND_WRITE, VX_MEMORY_TYPE_HOST, 0);
for (i = 0; i < num_items; i++)
{
    vxArrayItem(mystruct, base, i, stride).some_uint += i;
    vxArrayItem(mystruct, base, i, stride).some_double = 3.14f;
}
vxUnmapArrayRange(array, map_id);
```

Macros

- #define vxArrayItem(type, ptr, index, stride) (∗(type ∗)(vxFormatArrayPointer((ptr), (index), (stride))))

  Allows access to an array item as a typecast pointer deference.
- #define vxFormatArrayPointer(ptr, index, stride) (&(((vx_uint8∗)(ptr))[(index) ∗ (stride)]))

  Accesses a specific indexed element in an array.

Typedefs

- typedef struct _vx_array ∗ vx_array

  The Array Object. Array is a strongly-typed container for other data structures.

Enumerations

- enum vx_array_attribute_e {
  VX_ARRAY_ITEMTYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_ARRAY << 8)) + 0x0,
  VX_ARRAY_NUMITEMS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_ARRAY << 8)) + 0x1,
  VX_ARRAY_CAPACITY = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_ARRAY << 8)) + 0x2,
  VX_ARRAY_ITEMSIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_ARRAY << 8)) + 0x3 }

  The array object attributes.

Functions

- vx_status VX_API_CALL vxAddArrayItems (vx_array arr, vx_size count, const void ∗ptr, vx_size stride)

  Adds items to the Array.
- vx_status VX_API_CALL 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)

  Allows the application to copy a range from/into an array object.
- vx_array VX_API_CALL vxCreateArray (vx_context context, vx_enum item_type, vx_size capacity)

  Creates a reference to an Array object.
- vx_array VX_API_CALL vxCreateVirtualArray (vx_graph graph, vx_enum item_type, vx_size capacity)

  Creates an opaque reference to a virtual Array with no direct user access.
- vx_status VX_API_CALL 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)

  Allows the application to get direct access to a range of an array object.
- vx_status VX_API_CALL vxQueryArray (vx_array arr, vx_enum attribute, void ∗ptr, vx_size size)

  Queries the Array for some specific information.
- vx_status VX_API_CALL vxReleaseArray (vx_array ∗arr)

Releases a reference of an Array object. The object may not be garbage collected until its total reference count is zero. After returning from this function the reference is zeroed.

- vx_status VX_API_CALL vxTruncateArray (vx_array arr, vx_size new_num_items)

  Truncates an Array (remove items from the end).

- vx_status VX_API_CALL vxUnmapArrayRange (vx_array array, vx_map_id map_id)

  Unmap and commit potential changes to an array object range that was previously mapped. Unmapping an array range invalidates the memory location from which the range could be accessed by the application. Accessing this memory location after the unmap function completes has an undefined behavior.

3.51.2 Macro Definition Documentation

#define vxFormatArrayPointer(*ptr, index, stride*) (&(((vx_uint8*)(ptr))[(index) * (stride)]))

Accesses a specific indexed element in an array.

Parameters

in	*ptr*	The base pointer for the array range.
in	*index*	The index of the element, not byte, to access.
in	*stride*	The 'number of bytes' between the beginning of two consecutive elements.

Definition at line 2358 of file vx_api.h.

#define vxArrayItem(*type, ptr, index, stride*) (*(type *)(vxFormatArrayPointer((ptr), (index), (stride))))

Allows access to an array item as a typecast pointer deference.

Parameters

in	*type*	The type of the item to access.
in	*ptr*	The base pointer for the array range.
in	*index*	The index of the element, not byte, to access.
in	*stride*	The 'number of bytes' between the beginning of two consecutive elements.

Definition at line 2369 of file vx_api.h.

3.51.3 Enumeration Type Documentation

enum vx_array_attribute_e

The array object attributes.

Enumerator

VX_ARRAY_ITEMTYPE The type of the Array items. Read-only. Use a `vx_enum` parameter.

VX_ARRAY_NUMITEMS The number of items in the Array. Read-only. Use a `vx_size` parameter.

VX_ARRAY_CAPACITY The maximal number of items that the Array can hold. Read-only. Use a `vx_size` parameter.

VX_ARRAY_ITEMSIZE Queries an array item size. Read-only. Use a `vx_size` parameter.

Definition at line 1075 of file vx_types.h.

3.51.4 Function Documentation

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

Creates a reference to an Array object.

User must specify the Array capacity (i.e., the maximal number of items that the array can hold).

Parameters

in	*context*	The reference to the overall Context.
in	*item_type*	The type of objects to hold. Types allowed are: plain scalar types (i.e. type with enum below VX_TYPE_SCALAR_MAX), VX_TYPE_RECTANGLE, VX_TYPE_KEYPOINT, VX_TYPE_COORDINATES2D, VX_TYPE_COORDINATES3D and user registered structures. Use: • VX_TYPE_RECTANGLE for vx_rectangle_t. • VX_TYPE_KEYPOINT for vx_keypoint_t. • VX_TYPE_COORDINATES2D for vx_coordinates2d_t. • VX_TYPE_COORDINATES3D for vx_coordinates3d_t. • vx_enum returned from vxRegisterUserStruct.
in	*capacity*	The maximal number of items that the array can hold.

Returns

An array reference vx_array. Any possible errors preventing a successful creation should be checked using vxGetStatus.

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

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

Virtual Arrays are useful when item type or capacity are unknown ahead of time and the Array is used as internal graph edge. Virtual arrays are scoped within the parent graph only.

All of the following constructions are allowed.

```
vx_context context = vxCreateContext();
vx_graph graph = vxCreateGraph(context);
vx_array virt[] = {
    vxCreateVirtualArray(graph, 0, 0), // totally unspecified
    vxCreateVirtualArray(graph, VX_TYPE_KEYPOINT, 0), // unspecified
      capacity
    vxCreateVirtualArray(graph, VX_TYPE_KEYPOINT, 1000), // no access
};
```

Parameters

in	*graph*	The reference to the parent graph.
in	*item_type*	The type of objects to hold. Types allowed are: plain scalar types (i.e. type with enum below VX_TYPE_SCALAR_MAX), VX_TYPE_RECTANGLE, VX_TYPE_KEYPOINT, VX_TYPE_COORDINATES2D, VX_TYPE_COORDINATES3D and user registered structures. This may to set to zero to indicate an unspecified item type.
in	*capacity*	The maximal number of items that the array can hold. This may be to set to zero to indicate an unspecified capacity.

See Also

vxCreateArray for a type list.

Returns

A array reference vx_array. Any possible errors preventing a successful creation should be checked using vxGetStatus.

vx_status VX_API_CALL vxReleaseArray (vx_array $*$ *arr*)

Releases a reference of an Array object. The object may not be garbage collected until its total reference count is zero. After returning from this function the reference is zeroed.

Parameters

in	*arr*	The pointer to the Array to release.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If arr is not a `vx_array`.

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

Queries the Array for some specific information.

Parameters

in	*arr*	The reference to the Array.
in	*attribute*	The attribute to query. Use a `vx_array_attribute_e`.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the *arr* is not a `vx_array`.
VX_ERROR_NOT_SUPPORTED	If the *attribute* is not a value supported on this implementation.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

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

Adds items to the Array.

This function increases the container size.

By default, the function does not reallocate memory, so if the container is already full (number of elements is equal to capacity) or it doesn't have enough space, the function returns `VX_FAILURE` error code.

Parameters

in	*arr*	The reference to the Array.
in	*count*	The total number of elements to insert.
in	*ptr*	The location from which to read the input values.
in	*stride*	The number of bytes between the beginning of two consecutive elements.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the *arr* is not a `vx_array`.
VX_FAILURE	If the Array is full.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

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

Truncates an Array (remove items from the end).

Parameters

`in,out`	*arr*	The reference to the Array.
`in`	*new_num_items*	The new number of items for the Array.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the *arr* is not a `vx_array`.
VX_ERROR_INVALID_PARAMETERS	The *new_size* is greater than the current size.

vx_status VX_API_CALL 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*)

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

Parameters

`in`	*array*	The reference to the array object that is the source or the destination of the copy.
`in`	*range_start*	The index of the first item of the array object to copy.
`in`	*range_end*	The index of the item following the last item of the array object to copy. (range_end range_start) items are copied from index range_start included. The range must be within the bounds of the array: 0 <= range_start < range_end <= number of items in the array.
`in`	*user_stride*	The number of bytes between the beginning of two consecutive items in the user memory pointed by user_ptr. The layout of the user memory must follow an item major order: user_stride >= element size in bytes.
`in`	*user_ptr*	The address of the memory location where to store the requested data if the copy was requested in read mode, or from where to get the data to store into the array object if the copy was requested in write mode. The accessible memory must be large enough to contain the specified range with the specified stride: accessible memory in bytes >= (range_end range_start) $*$ user_stride.
`in`	*usage*	This declares the effect of the copy with regard to the array object using the `vx_accessor_e` enumeration. Only `VX_READ_ONLY` and `VX_WRITE_ONLY` are supported: • `VX_READ_ONLY` means that data are copied from the array object into the user memory. • `VX_WRITE_ONLY` means that data are copied into the array object from the user memory.
`in`	*user_mem_type*	A `vx_memory_type_e` enumeration that specifies the memory type of the memory referenced by the user_addr.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_OPTIMIZED_AWAY	This is a reference to a virtual array that cannot be accessed by the application.
VX_ERROR_INVALID_REFERENCE	The array reference is not actually an array reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

vx_status VX_API_CALL 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*)

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

Parameters

`in`	*array*	The reference to the array object that contains the range to map.
`in`	*range_start*	The index of the first item of the array object to map.
`in`	*range_end*	The index of the item following the last item of the array object to map. (range_end range_start) items are mapped, starting from index range_start included. The range must be within the bounds of the array: Must be 0 <= range_start < range_end <= number of items.
`out`	*map_id*	The address of a `vx_map_id` variable where the function returns a map identifier. • (*map_id) must eventually be provided as the map_id parameter of a call to `vxUnmapArrayRange`.
`out`	*stride*	The address of a vx_size variable where the function returns the memory layout of the mapped array range. The function sets (*stride) to the number of bytes between the beginning of two consecutive items. The application must consult (*stride) to access the array items starting from address (*ptr). The layout of the mapped array follows an item major order: (*stride) >= item size in bytes.
`out`	*ptr*	The address of a pointer that the function sets to the address where the requested data can be accessed. The returned (*ptr) address is only valid between the call to the function and the corresponding call to `vxUnmapArrayRange`.
`in`	*usage*	This declares the access mode for the array range, using the `vx_accessor_e` enumeration. • `VX_READ_ONLY`: after the function call, the content of the memory location pointed by (*ptr) contains the array range data. Writing into this memory location is forbidden and its behavior is undefined. • `VX_READ_AND_WRITE`: after the function call, the content of the memory location pointed by (*ptr) contains the array range data; writing into this memory is allowed only for the location of items and will result in a modification of the affected items in the array object once the range is unmapped. Writing into a gap between items (when (*stride) > item size in bytes) is forbidden and its behavior is undefined. • `VX_WRITE_ONLY`: after the function call, the memory location pointed by (*ptr) contains undefined data; writing each item of the range is required prior to unmapping. Items not written by the application before unmap will become undefined after unmap, even if they were well defined before map. Like for VX_READ_AND_WRITE, writing into a gap between items is forbidden and its behavior is undefined.
`in`	*mem_type*	A `vx_memory_type_e` enumeration that specifies the type of the memory where the array range is requested to be mapped.
`in`	*flags*	An integer that allows passing options to the map operation. Use the `vx_map_flag_e` enumeration.

Returns

A `vx_status_e` enumeration.

Return values

VX_ERROR_OPTIMIZED_-AWAY	This is a reference to a virtual array that cannot be accessed by the application.
VX_ERROR_INVALID_RE-FERENCE	The array reference is not actually an array reference.
VX_ERROR_INVALID_PA-RAMETERS	An other parameter is incorrect.

Postcondition

`vxUnmapArrayRange` with same (*map_id) value.

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

Unmap and commit potential changes to an array object range that was previously mapped. Unmapping an array range invalidates the memory location from which the range could be accessed by the application. Accessing this memory location after the unmap function completes has an undefined behavior.

Parameters

`in`	*array*	The reference to the array object to unmap.
`out`	*map_id*	The unique map identifier that was returned when calling `vxMapArrayRange` .

Returns

A `vx_status_e` enumeration.

Return values

VX_ERROR_INVALID_RE-FERENCE	The array reference is not actually an array reference.
VX_ERROR_INVALID_PA-RAMETERS	An other parameter is incorrect.

Precondition

`vxMapArrayRange` returning the same map_id value

3.52 Object: Convolution

3.52.1 Detailed Description

Defines the Image Convolution Object interface.

Typedefs

- typedef struct _vx_convolution ∗ vx_convolution

  The Convolution Object. A user-defined convolution kernel of MxM elements.

Enumerations

- enum vx_convolution_attribute_e {
  VX_CONVOLUTION_ROWS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONVOLUTION << 8)) + 0x0,
  VX_CONVOLUTION_COLUMNS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONVOLUTION << 8)) + 0x1,
  VX_CONVOLUTION_SCALE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONVOLUTION << 8)) + 0x2,
  VX_CONVOLUTION_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONVOLUTION << 8)) + 0x3 }

  The convolution attributes.

Functions

- vx_status VX_API_CALL vxCopyConvolutionCoefficients (vx_convolution conv, void ∗user_ptr, vx_enum usage, vx_enum user_mem_type)

  Allows the application to copy coefficients from/into a convolution object.
- vx_convolution VX_API_CALL vxCreateConvolution (vx_context context, vx_size columns, vx_size rows)

  Creates a reference to a convolution matrix object.
- vx_status VX_API_CALL vxQueryConvolution (vx_convolution conv, vx_enum attribute, void ∗ptr, vx_size size)

  Queries an attribute on the convolution matrix object.
- vx_status VX_API_CALL vxReleaseConvolution (vx_convolution ∗conv)

  Releases the reference to a convolution matrix. The object may not be garbage collected until its total reference count is zero.
- vx_status VX_API_CALL vxSetConvolutionAttribute (vx_convolution conv, vx_enum attribute, const void ∗ptr, vx_size size)

  Sets attributes on the convolution object.

3.52.2 Enumeration Type Documentation

enum vx_convolution_attribute_e

The convolution attributes.

Enumerator

VX_CONVOLUTION_ROWS The number of rows of the convolution matrix. Read-only. Use a `vx_size` parameter.

VX_CONVOLUTION_COLUMNS The number of columns of the convolution matrix. Read-only. Use a `vx_size` parameter.

VX_CONVOLUTION_SCALE The scale of the convolution matrix. Read-write. Use a `vx_uint32` parameter.

Note

For 1.0, only powers of 2 are supported up to 2^31.

VX_CONVOLUTION_SIZE The total size of the convolution matrix in bytes. Read-only. Use a `vx_size` parameter.

Definition at line 1027 of file vx_types.h.

3.52.3 Function Documentation

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

Creates a reference to a convolution matrix object.

Parameters

in	*context*	The reference to the overall context.
in	*columns*	The columns dimension of the convolution. Must be odd and greater than or equal to 3 and less than the value returned from `VX_CONTEXT_CONVOLUTION_MAX_DIMENSION`.
in	*rows*	The rows dimension of the convolution. Must be odd and greater than or equal to 3 and less than the value returned from `VX_CONTEXT_CONVOLUTION_MAX_DIMENSION`.

Returns

A convolution reference `vx_convolution`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

vx_status VX_API_CALL vxReleaseConvolution (vx_convolution * *conv*)

Releases the reference to a convolution matrix. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*conv*	The pointer to the convolution matrix to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If conv is not a `vx_convolution`.

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

Queries an attribute on the convolution matrix object.

Parameters

in	*conv*	The convolution matrix object to set.
in	*attribute*	The attribute to query. Use a vx_convolution_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

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

Sets attributes on the convolution object.

Parameters

in	*conv*	The coordinates object to set.
in	*attribute*	The attribute to modify. Use a vx_convolution_attribute_e enumeration.
in	*ptr*	The pointer to the value to which to set the attribute.
in	*size*	The size in bytes of the data pointed to by *ptr*.

Returns

A vx_status_e enumeration.

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

Allows the application to copy coefficients from/into a convolution object.

Parameters

in	*conv*	The reference to the convolution object that is the source or the destination of the copy.
in	*user_ptr*	The address of the memory location where to store the requested coefficient data if the copy was requested in read mode, or from where to get the coefficient data to store into the convolution object if the copy was requested in write mode. In the user memory, the convolution coefficient data is structured as a row-major 2D array with elements of the type corresponding to VX_TYPE_CONVOLUTION, with a number of rows corresponding to VX_CONVOLUTION_ROWS and a number of columns corresponding to VX_CONVOLUTION_COLUMNS. The accessible memory must be large enough to contain this 2D array: accessible memory in bytes >= sizeof(data_element) ∗ rows ∗ columns.

in	*usage*	This declares the effect of the copy with regard to the convolution object using the vx_accessor_e enumeration. Only VX_READ_ONLY and VX_WRITE_ONLY are supported: • VX_READ_ONLY means that data are copied from the convolution object into the user memory. • VX_WRITE_ONLY means that data are copied into the convolution object from the user memory.
in	*user_mem_type*	A vx_memory_type_e enumeration that specifies the memory type of the memory referenced by the user_addr.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	The convolution reference is not actually a convolution reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

3.53 Object: Distribution

3.53.1 Detailed Description

Defines the Distribution Object Interface.

Typedefs

- typedef struct _vx_distribution ∗ vx_distribution

    The Distribution object. This has a user-defined number of bins over a user-defined range (within a uint32_t range).

Enumerations

- enum vx_distribution_attribute_e {
VX_DISTRIBUTION_DIMENSIONS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x0,
VX_DISTRIBUTION_OFFSET = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x1,
VX_DISTRIBUTION_RANGE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x2,
VX_DISTRIBUTION_BINS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x3,
VX_DISTRIBUTION_WINDOW = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x4,
VX_DISTRIBUTION_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x5 }

    The distribution attribute list.

Functions

- vx_status VX_API_CALL vxCopyDistribution (vx_distribution distribution, void ∗user_ptr, vx_enum usage, vx_enum user_mem_type)

    Allows the application to copy from/into a distribution object.
- vx_distribution VX_API_CALL vxCreateDistribution (vx_context context, vx_size numBins, vx_int32 offset, vx_uint32 range)

    Creates a reference to a 1D Distribution of a consecutive interval [offset, offset + range - 1] defined by a start offset and valid range, divided equally into numBins parts.
- vx_status VX_API_CALL vxMapDistribution (vx_distribution distribution, vx_map_id ∗map_id, void ∗∗ptr, vx_enum usage, vx_enum mem_type, vx_bitfield flags)

    Allows the application to get direct access to distribution object.
- vx_status VX_API_CALL vxQueryDistribution (vx_distribution distribution, vx_enum attribute, void ∗ptr, vx_size size)

    Queries a Distribution object.
- vx_status VX_API_CALL vxReleaseDistribution (vx_distribution ∗distribution)

    Releases a reference to a distribution object. The object may not be garbage collected until its total reference count is zero.
- vx_status VX_API_CALL vxUnmapDistribution (vx_distribution distribution, vx_map_id map_id)

    Unmap and commit potential changes to distribution object that was previously mapped. Unmapping a distribution invalidates the memory location from which the distribution data could be accessed by the application. Accessing this memory location after the unmap function completes has an undefined behavior.

3.53.2 Enumeration Type Documentation

enum vx_distribution_attribute_e

The distribution attribute list.

Enumerator

VX_DISTRIBUTION_DIMENSIONS Indicates the number of dimensions in the distribution. Read-only. Use a `vx_size` parameter.

VX_DISTRIBUTION_OFFSET Indicates the start of the values to use (inclusive). Read-only. Use a vx_int32 parameter.

VX_DISTRIBUTION_RANGE Indicates the total number of the consecutive values of the distribution interval.

VX_DISTRIBUTION_BINS Indicates the number of bins. Read-only. Use a vx_size parameter.

VX_DISTRIBUTION_WINDOW Indicates the width of a bin. Equal to the range divided by the number of bins. If the range is not a multiple of the number of bins, it is not valid. Read-only. Use a vx_uint32 parameter.

VX_DISTRIBUTION_SIZE Indicates the total size of the distribution in bytes. Read-only. Use a vx_size parameter.

Definition at line 959 of file vx_types.h.

3.53.3 Function Documentation

vx_distribution VX_API_CALL vxCreateDistribution (vx_context *context*, vx_size *numBins*, vx_int32 *offset*, vx_uint32 *range*)

Creates a reference to a 1D Distribution of a consecutive interval [offset, offset + range - 1] defined by a start offset and valid range, divided equally into numBins parts.

Parameters

in	*context*	The reference to the overall context.
in	*numBins*	The number of bins in the distribution.
in	*offset*	The start offset into the range value that marks the begining of the 1D Distribution.
in	*range*	The total number of the consecutive values of the distribution interval.

Returns

A distribution reference vx_distribution. Any possible errors preventing a successful creation should be checked using vxGetStatus.

vx_status VX_API_CALL vxReleaseDistribution (vx_distribution * *distribution*)

Releases a reference to a distribution object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*distribution*	The reference to the distribution to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If distribution is not a vx_distribution.

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

Queries a Distribution object.

Parameters

in	*distribution*	The reference to the distribution to query.
in	*attribute*	The attribute to query. Use a vx_distribution_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

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

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

Parameters

in	*distribution*	The reference to the distribution object that is the source or the destination of the copy.
in	*user_ptr*	The address of the memory location where to store the requested data if the copy was requested in read mode, or from where to get the data to store into the distribution object if the copy was requested in write mode. In the user memory, the distribution is represented as a vx_uint32 array with a number of elements equal to the value returned via VX_DISTRIBUTION_BINS. The accessible memory must be large enough to contain this vx_uint32 array: accessible memory in bytes $>=$ sizeof(vx_uint32) $*$ num_bins.
in	*usage*	This declares the effect of the copy with regard to the distribution object using the vx_accessor_e enumeration. Only VX_READ_ONLY and VX_WRITE_ONLY are supported: • VX_READ_ONLY means that data are copied from the distribution object into the user memory. • VX_WRITE_ONLY means that data are copied into the distribution object from the user memory.
in	*user_mem_type*	A vx_memory_type_e enumeration that specifies the memory type of the memory referenced by the user_addr.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	The distribution reference is not actually a distribution reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

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

Allows the application to get direct access to distribution object.

Parameters

in	*distribution*	The reference to the distribution object to map.
out	*map_id*	The address of a vx_map_id variable where the function returns a map identifier. • (∗map_id) must eventually be provided as the map_id parameter of a call to vxUnmapDistribution.
out	*ptr*	The address of a pointer that the function sets to the address where the requested data can be accessed. In the mapped memory area, data are structured as a vx_uint32 array with a number of elements equal to the value returned via VX_DISTRIBUTION_BINS. Each element of this array corresponds to a bin of the distribution, with a range-major ordering. Accessing the memory out of the bound of this array is forbidden and has an undefined behavior. The returned (∗ptr) address is only valid between the call to the function and the corresponding call to vxUnmapDistribution.
in	*usage*	This declares the access mode for the distribution, using the vx_accessor_e enumeration. • VX_READ_ONLY: after the function call, the content of the memory location pointed by (∗ptr) contains the distribution data. Writing into this memory location is forbidden and its behavior is undefined. • VX_READ_AND_WRITE: after the function call, the content of the memory location pointed by (∗ptr) contains the distribution data; writing into this memory is allowed only for the location of bins and will result in a modification of the affected bins in the distribution object once the distribution is unmapped. • VX_WRITE_ONLY: after the function call, the memory location pointed by (∗ptr) contains undefined data; writing each bin of distribution is required prior to unmapping. Bins not written by the application before unmap will become undefined after unmap, even if they were well defined before map.

in	*mem_type*	A vx_memory_type_e enumeration that specifies the type of the memory where the distribution is requested to be mapped.
in	*flags*	An integer that allows passing options to the map operation. Use 0 for this option.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	The distribution reference is not actually a distribution reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

Postcondition

vxUnmapDistribution with same (∗map_id) value.

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

Unmap and commit potential changes to distribution object that was previously mapped. Unmapping a distribution invalidates the memory location from which the distribution data could be accessed by the application. Accessing this memory location after the unmap function completes has an undefined behavior.

Parameters

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

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	The distribution reference is not actually a distribution reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

Precondition

vxMapDistribution returning the same map_id value

3.54 Object: Image

3.54.1 Detailed Description

Defines the Image Object interface.

Data Structures

- struct vx_imagepatch_addressing_t

  The addressing image patch structure is used by the Host only to address pixels in an image patch. The fields of the structure are defined as: More...
- union vx_pixel_value_t

  Union that describes the value of a pixel for any image format. Use the field corresponding to the image format. More...

Macros

- #define VX_IMAGEPATCH_ADDR_INIT {0u, 0u, 0, 0, 0u, 0u, 0u, 0u}

  Use to initialize a `vx_imagepatch_addressing_t` structure on the stack.

Typedefs

- typedef struct _vx_image ∗ vx_image

  An opaque reference to an image.
- typedef uintptr_t vx_map_id

  Holds the address of a variable where the map/unmap functions return a map identifier.

Enumerations

- enum vx_channel_range_e {
  VX_CHANNEL_RANGE_FULL = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_RANGE << 12)) + 0x0,
  VX_CHANNEL_RANGE_RESTRICTED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_RANGE << 12)) + 0x1 }

  The image channel range list used by the `VX_IMAGE_RANGE` attribute of a `vx_image`.
- enum vx_color_space_e {
  VX_COLOR_SPACE_NONE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_SPACE << 12)) + 0x0,
  VX_COLOR_SPACE_BT601_525 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_SPACE << 12)) + 0x1,
  VX_COLOR_SPACE_BT601_625 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_SPACE << 12)) + 0x2,
  VX_COLOR_SPACE_BT709 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_SPACE << 12)) + 0x3,
  VX_COLOR_SPACE_DEFAULT = VX_COLOR_SPACE_BT709 }

  The image color space list used by the `VX_IMAGE_SPACE` attribute of a `vx_image`.
- enum vx_image_attribute_e {
  VX_IMAGE_WIDTH = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x0,
  VX_IMAGE_HEIGHT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x1,
  VX_IMAGE_FORMAT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x2,
  VX_IMAGE_PLANES = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x3,
  VX_IMAGE_SPACE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x4,
  VX_IMAGE_RANGE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x5,
  VX_IMAGE_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x6,
  VX_IMAGE_MEMORY_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x7 }

  The image attributes list.

- enum vx_map_flag_e { VX_NOGAP_X = 1 }

  The Map/Unmap operation enumeration.

Functions

- vx_size VX_API_CALL vxComputeImagePatchSize (vx_image image, const vx_rectangle_t ∗rect, vx_uint32 plane_index)

  This computes the size needed to retrieve an image patch from an image.
- vx_status VX_API_CALL 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)

  Allows the application to copy a rectangular patch from/into an image object plane.
- vx_image VX_API_CALL vxCreateImage (vx_context context, vx_uint32 width, vx_uint32 height, vx_df_image color)

  Creates an opaque reference to an image buffer.
- vx_image VX_API_CALL vxCreateImageFromChannel (vx_image img, vx_enum channel)

  Create a sub-image from a single plane channel of another image.
- vx_image VX_API_CALL vxCreateImageFromHandle (vx_context context, vx_df_image color, const vx_imagepatch_addressing_t addrs[ ], void ∗const ptrs[ ], vx_enum memory_type)

  Creates a reference to an image object that was externally allocated.
- vx_image VX_API_CALL vxCreateImageFromROI (vx_image img, const vx_rectangle_t ∗rect)

  Creates an image from another image given a rectangle. This second reference refers to the data in the original image. Updates to this image updates the parent image. The rectangle must be defined within the pixel space of the parent image.
- vx_image VX_API_CALL vxCreateUniformImage (vx_context context, vx_uint32 width, vx_uint32 height, vx_df_image color, const vx_pixel_value_t ∗value)

  Creates a reference to an image object that has a singular, uniform value in all pixels. The uniform image created is read-only.
- vx_image VX_API_CALL vxCreateVirtualImage (vx_graph graph, vx_uint32 width, vx_uint32 height, vx_df_image color)

  Creates an opaque reference to an image buffer with no direct user access. This function allows setting the image width, height, or format.
- void ∗VX_API_CALL vxFormatImagePatchAddress1d (void ∗ptr, vx_uint32 index, const vx_imagepatch_addressing_t ∗addr)

  Accesses a specific indexed pixel in an image patch.
- void ∗VX_API_CALL vxFormatImagePatchAddress2d (void ∗ptr, vx_uint32 x, vx_uint32 y, const vx_imagepatch_addressing_t ∗addr)

  Accesses a specific pixel at a 2d coordinate in an image patch.
- vx_status VX_API_CALL vxGetValidRegionImage (vx_image image, vx_rectangle_t ∗rect)

  Retrieves the valid region of the image as a rectangle.
- vx_status VX_API_CALL 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)

  Allows the application to get direct access to a rectangular patch of an image object plane.
- vx_status VX_API_CALL vxQueryImage (vx_image image, vx_enum attribute, void ∗ptr, vx_size size)

  Retrieves various attributes of an image.
- vx_status VX_API_CALL vxReleaseImage (vx_image ∗image)

  Releases a reference to an image object. The object may not be garbage collected until its total reference count is zero.
- vx_status VX_API_CALL vxSetImageAttribute (vx_image image, vx_enum attribute, const void ∗ptr, vx_size size)

  Allows setting attributes on the image.
- vx_status VX_API_CALL vxSetImageValidRectangle (vx_image image, const vx_rectangle_t ∗rect)

  Sets the valid rectangle for an image according to a supplied rectangle.

- vx_status VX_API_CALL vxSwapImageHandle (vx_image image, void ∗const new_ptrs[ ], void ∗prev_ptrs[ ], vx_size num_planes)

  Swaps the image handle of an image previously created from handle.
- vx_status VX_API_CALL vxUnmapImagePatch (vx_image image, vx_map_id map_id)

  Unmap and commit potential changes to a image object patch that were previously mapped. Unmapping an image patch invalidates the memory location from which the patch could be accessed by the application. Accessing this memory location after the unmap function completes has an undefined behavior.

3.54.2 Data Structure Documentation

struct vx_imagepatch_addressing_t

The addressing image patch structure is used by the Host only to address pixels in an image patch. The fields of the structure are defined as:

- dim - The dimensions of the image in logical pixel units in the x & y direction.
- stride - The physical byte distance from a logical pixel to the next logically adjacent pixel in the positive x or y direction.
- scale - The relationship of scaling from the primary plane (typically the zero indexed plane) to this plane. An integer down-scaling factor of f shall be set to a value equal to $scale = \frac{unity}{f}$ and an integer up-scaling factor of f shall be set to a value of $scale = unity * f$. $unity$ is defined as `VX_SCALE_UNITY`.
- step - The step is the number of logical pixel units to skip to arrive at the next physically unique pixel. For example, on a plane that is half-scaled in a dimension, the step in that dimension is 2 to indicate that every other pixel in that dimension is an alias. This is useful in situations where iteration over unique pixels is required, such as in serializing or de-serializing the image patch information.

  See Also

  `vxMapImagePatch`

Definition at line 1394 of file vx_types.h.

Data Fields

vx_uint32	dim_x	Width of patch in X dimension in pixels.
vx_uint32	dim_y	Height of patch in Y dimension in pixels.
vx_int32	stride_x	Stride in X dimension in bytes.
vx_int32	stride_y	Stride in Y dimension in bytes.
vx_uint32	scale_x	Scale of X dimension. For sub-sampled planes this is the scaling factor of the dimension of the plane in relation to the zero plane. Use `VX_SCALE_UNITY` in the numerator.
vx_uint32	scale_y	Scale of Y dimension. For sub-sampled planes this is the scaling factor of the dimension of the plane in relation to the zero plane. Use `VX_SCALE_UNITY` in the numerator.
vx_uint32	step_x	Step of X dimension in pixels.
vx_uint32	step_y	Step of Y dimension in pixels.

union vx_pixel_value_t

Union that describes the value of a pixel for any image format. Use the field corresponding to the image format.

Definition at line 1501 of file vx_types.h.

Data Fields

vx_uint8	RGB[3]	`VX_DF_IMAGE_RGB` format in the R,G,B order

vx_uint8	RGBX[4]	VX_DF_IMAGE_RGBX format in the R,G,B,X order
vx_uint8	YUV[3]	All YUV formats in the Y,U,V order.
vx_uint8	U8	VX_DF_IMAGE_U8
vx_uint16	U16	VX_DF_IMAGE_U16
vx_int16	S16	VX_DF_IMAGE_S16
vx_uint32	U32	VX_DF_IMAGE_U32
vx_int32	S32	VX_DF_IMAGE_S32
vx_uint8	reserved[16]	

3.54.3 Typedef Documentation

typedef struct _vx_image* vx_image

An opaque reference to an image.

See Also

vxCreateImage

Definition at line 190 of file vx_types.h.

3.54.4 Enumeration Type Documentation

enum vx_image_attribute_e

The image attributes list.

Enumerator

VX_IMAGE_WIDTH Queries an image for its width. Read-only. Use a `vx_uint32` parameter.

VX_IMAGE_HEIGHT Queries an image for its height. Read-only. Use a `vx_uint32` parameter.

VX_IMAGE_FORMAT Queries an image for its format. Read-only. Use a `vx_df_image` parameter.

VX_IMAGE_PLANES Queries an image for its number of planes. Read-only. Use a `vx_size` parameter.

VX_IMAGE_SPACE Queries an image for its color space (see `vx_color_space_e`). Read-write. Use a `vx_enum` parameter.

VX_IMAGE_RANGE Queries an image for its channel range (see `vx_channel_range_e`). Read-only. Use a `vx_enum` parameter.

VX_IMAGE_SIZE Queries an image for its total number of bytes. Read-only. Use a `vx_size` parameter.

VX_IMAGE_MEMORY_TYPE Queries memory type if created using vxCreateImageFromHandle. If vx_image was not created using vxCreateImageFromHandle, VX_MEMORY_TYPE_NONE is returned. Use a `vx_memory_type_e` parameter.

Definition at line 914 of file vx_types.h.

enum vx_color_space_e

The image color space list used by the `VX_IMAGE_SPACE` attribute of a `vx_image`.

Enumerator

VX_COLOR_SPACE_NONE Use to indicate that no color space is used.

VX_COLOR_SPACE_BT601_525 Use to indicate that the BT.601 coefficients and SMPTE C primaries are used for conversions.

VX_COLOR_SPACE_BT601_625 Use to indicate that the BT.601 coefficients and BTU primaries are used for conversions.

VX_COLOR_SPACE_BT709 Use to indicate that the BT.709 coefficients are used for conversions.

VX_COLOR_SPACE_DEFAULT All images in VX are by default BT.709.

Definition at line 1210 of file vx_types.h.

enum vx_channel_range_e

The image channel range list used by the `VX_IMAGE_RANGE` attribute of a `vx_image`.

Enumerator

VX_CHANNEL_RANGE_FULL Full range of the unit of the channel.

VX_CHANNEL_RANGE_RESTRICTED Restricted range of the unit of the channel based on the space given.

Definition at line 1227 of file vx_types.h.

enum vx_map_flag_e

The Map/Unmap operation enumeration.

Enumerator

VX_NOGAP_X No Gap.

Definition at line 1635 of file vx_types.h.

3.54.5 Function Documentation

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

Creates an opaque reference to an image buffer.
Not guaranteed to exist until the `vx_graph` containing it has been verified.

Parameters

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

Returns

An image reference `vx_image`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

See Also

vxMapImagePatch to obtain direct memory access to the image data.

vx_image VX_API_CALL vxCreateImageFromROI (vx_image *img,* const vx_rectangle_t * *rect*)

Creates an image from another image given a rectangle. This second reference refers to the data in the original image. Updates to this image updates the parent image. The rectangle must be defined within the pixel space of the parent image.

Parameters

in	*img*	The reference to the parent image.
in	*rect*	The region of interest rectangle. Must contain points within the parent image pixel space.

Returns

An image reference `vx_image` to the sub-image. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

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

Creates a reference to an image object that has a singular, uniform value in all pixels. The uniform image created is read-only.

Parameters

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

Returns

An image reference vx_image. Any possible errors preventing a successful creation should be checked using vxGetStatus.

See Also

vxMapImagePatch to obtain direct memory access to the image data.

Note

vxMapImagePatch and vxUnmapImagePatch may be called with a uniform image reference.

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

Creates an opaque reference to an image buffer with no direct user access. This function allows setting the image width, height, or format.

Virtual data objects allow users to connect various nodes within a graph via data references without access to that data, but they also permit the implementation to take maximum advantage of possible optimizations. Use this API to create a data reference to link two or more nodes together when the intermediate data are not required to be accessed by outside entities. This API in particular allows the user to define the image format of the data without requiring the exact dimensions. Virtual objects are scoped within the graph they are declared a part of, and can't be shared outside of this scope. All of the following constructions of virtual images are valid.

```
vx_context context = vxCreateContext();
vx_graph graph = vxCreateGraph(context);
vx_image virt[] = {
    vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_U8), // no specified
       dimension
    vxCreateVirtualImage(graph, 320, 240, VX_DF_IMAGE_VIRT), // no
       specified format
    vxCreateVirtualImage(graph, 640, 480, VX_DF_IMAGE_U8), // no user
       access
};
```

Parameters

in	*graph*	The reference to the parent graph.
in	*width*	The width of the image in pixels. A value of zero informs the interface that the value is unspecified.
in	*height*	The height of the image in pixels. A value of zero informs the interface that the value is unspecified.
in	*color*	The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space. A value of VX_DF_IMAGE_VIRT informs the interface that the format is unspecified.

Returns

An image reference vx_image. Any possible errors preventing a successful creation should be checked using vxGetStatus.

Note

Passing this reference to vxMapImagePatch will return an error.

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

Creates a reference to an image object that was externally allocated.

Parameters

in	*context*	The reference to the implementation context.
in	*color*	See the vx_df_image_e codes. This mandates the number of planes needed to be valid in the *addrs* and *ptrs* arrays based on the format given.
in	*addrs[]*	The array of image patch addressing structures that define the dimension and stride of the array of pointers. See note below.
in	*ptrs[]*	The array of platform-defined references to each plane. See note below.
in	*memory_type*	vx_memory_type_e. When giving VX_MEMORY_TYPE_HOST the *ptrs* array is assumed to be HOST accessible pointers to memory.

Returns

An image reference vx_image. Any possible errors preventing a successful creation should be checked using vxGetStatus.

Note

The user must call vxMapImagePatch prior to accessing the pixels of an image, even if the image was created via vxCreateImageFromHandle. Reads or writes to memory referenced by ptrs[ ] after calling vxCreateImageFromHandle without first calling vxMapImagePatch will result in undefined behavior. The property of addr[] and ptrs[] arrays is kept by the caller (It means that the implementation will make an internal copy of the provided information. *addr* and *ptrs* can then simply be application's local variables). Only *dim_x*, *dim_y*, *stride_x* and *stride_y* fields of the vx_imagepatch_addressing_t need to be provided by the application. Other fields (*step_x*, *step_y*, *scale_x* & *scale_y*) are ignored by this function. The layout of the imported memory must follow a row-major order. In other words, *stride_x* should be sufficiently large so that there is no overlap between data elements corresponding to different pixels, and $stride\_y >= stride\_x * dim\_x$.

In order to release the image back to the application we should use vxSwapImageHandle.

Import type of the created image is available via the image attribute vx_image_attribute_e parameter.

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

Swaps the image handle of an image previously created from handle.

This function sets the new image handle (i.e. pointer to all image planes) and returns the previous one.

Once this function call has completed, the application gets back the ownership of the memory referenced by the previous handle. This memory contains up-to-date pixel data, and the application can safely reuse or release it.

The memory referenced by the new handle must have been allocated consistently with the image properties since the import type, memory layout and dimensions are unchanged (see addrs, color, and memory_type in vxCreateImageFromHandle).

All images created from ROI with this image as parent or ancestor will automatically use the memory referenced by the new handle.

The behavior of vxSwapImageHandle when called from a user node is undefined.

Parameters

in	*image*	The reference to an image created from handle
in	*new_ptrs[]*	pointer to a caller owned array that contains the new image handle (image plane pointers) • new_ptrs is non NULL. new_ptrs[i] must be non NULL for each i such as $0 < i <$ nbPlanes, otherwise, this is an error. The address of the storage memory for image plane i is set to new_ptrs[i] • new_ptrs is NULL: the previous image storage memory is reclaimed by the caller, while no new handle is provided.
out	*prev_ptrs[]*	pointer to a caller owned array in which the application returns the previous image handle • prev_ptrs is non NULL. prev_ptrs must have at least as many elements as the number of image planes. For each i such as $0 < i <$ nbPlanes , prev_ptrs[i] is set to the address of the previous storage memory for plane i. • prev_ptrs NULL: the previous handle is not returned.
in	*num_planes*	Number of planes in the image. This must be set equal to the number of planes of the input image. The number of elements in new_ptrs and prev_ptrs arrays must be equal to or greater than num_planes. If either array has more than num_planes elements, the extra elements are ignored. If either array is smaller than num_planes, the results are undefined.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	image is not a valid image reference.
VX_ERROR_INVALID_PARAMETERS	The image was not created from handle or the content of new_ptrs is not valid.
VX_FAILURE	The image was already being accessed.

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

Retrieves various attributes of an image.

Parameters

in	*image*	The reference to the image to query.
in	*attribute*	The attribute to query. Use a vx_image_attribute_e.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the image is not a vx_image.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.
VX_ERROR_NOT_SUPPORTED	If the attribute is not supported on this implementation.

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

Allows setting attributes on the image.

Parameters

in	*image*	The reference to the image on which to set the attribute.
in	*attribute*	The attribute to set. Use a vx_image_attribute_e enumeration.
in	*ptr*	The pointer to the location from which to read the value.
in	*size*	The size in bytes of the object pointed to by *ptr*.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the image is not a vx_image.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

vx_status VX_API_CALL vxReleaseImage (vx_image * *image*)

Releases a reference to an image object. The object may not be garbage collected until its total reference count is zero.

An implementation may defer the actual object destruction after its total reference count is zero (potentially until context destruction). Thus, releasing an image created from handle (see vxCreateImageFromHandle) and all others objects that may reference it (nodes, ROI for instance) are not sufficient to get back the ownership of the memory referenced by the current image handle. The only way for this is to call vxSwapImageHandle) before releasing the image.

Parameters

in	*image*	The pointer to the image to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If image is not a `vx_image`.

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

This computes the size needed to retrieve an image patch from an image.

Parameters

in	*image*	The reference to the image from which to extract the patch.
in	*rect*	The coordinates. Must be 0 <= start < end <= dimension where dimension is width for x and height for y.
in	*plane_index*	The plane index from which to get the data.

Returns

vx_size

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

Accesses a specific indexed pixel in an image patch.

Parameters

in	*ptr*	The base pointer of the patch as returned from `vxMapImagePatch`.
in	*index*	The 0 based index of the pixel count in the patch. Indexes increase horizontally by 1 then wrap around to the next row.
in	*addr*	The pointer to the addressing mode information returned from `vxMapImagePatch`.

Returns

void * Returns the pointer to the specified pixel.

Precondition

`vxMapImagePatch`

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

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

Parameters

in	*ptr*	The base pointer of the patch as returned from `vxMapImagePatch`.
in	*x*	The x dimension within the patch.
in	*y*	The y dimension within the patch.
in	*addr*	The pointer to the addressing mode information returned from `vxMapImagePatch`.

Returns

void * Returns the pointer to the specified pixel.

Precondition

`vxMapImagePatch`

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

Retrieves the valid region of the image as a rectangle.

Parameters

in	*image*	The image from which to retrieve the valid region.
out	*rect*	The destination rectangle.

Returns

vx_status

Return values

VX_ERROR_INVALID_REFERENCE	Invalid image.
VX_ERROR_INVALID_PARAMETERS	Invalid rect.
VX_SUCCESS	Valid image.

Note

This rectangle can be passed directly to `vxMapImagePatch` to get the full valid region of the image.

vx_status VX_API_CALL 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*)

Allows the application to copy a rectangular patch from/into an image object plane.

Parameters

in	*image*	The reference to the image object that is the source or the destination of the copy.
in	*image_rect*	The coordinates of the image patch. The patch must be within the bounds of the image. (start_x, start_y) gives the coordinates of the topleft pixel inside the patch, while (end_x, end_y) gives the coordinates of the bottomright element out of the patch. Must be 0 <= start < end <= number of pixels in the image dimension.
in	*image_plane_index*	The plane index of the image object that is the source or the destination of the patch copy.
in	*user_addr*	The address of a structure describing the layout of the user memory location pointed by user_ptr. In the structure, only dim_x, dim_y, stride_x and stride_y fields must be provided, other fields are ignored by the function. The layout of the user memory must follow a row major order: stride_x >= pixel size in bytes, and stride_y >= stride_x ∗ dim_x.
in	*user_ptr*	The address of the memory location where to store the requested data if the copy was requested in read mode, or from where to get the data to store into the image object if the copy was requested in write mode. The accessible memory must be large enough to contain the specified patch with the specified layout: accessible memory in bytes >= (end_y - start_y) ∗ stride_y.
in	*usage*	This declares the effect of the copy with regard to the image object using the `vx_accessor_e` enumeration. For uniform images, only VX_READ_ONLY is supported. For other images, Only `VX_READ_ONLY` and `VX_WRITE_ONLY` are supported: • `VX_READ_ONLY` means that data is copied from the image object into the application memory • `VX_WRITE_ONLY` means that data is copied into the image object from the application memory
in	*user_mem_type*	A `vx_memory_type_e` enumeration that specifies the memory type of the memory referenced by the user_addr.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_OPTIMIZED_-AWAY	This is a reference to a virtual image that cannot be accessed by the application.
VX_ERROR_INVALID_REFERENCE	The image reference is not actually an image reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

Note

The application may ask for data outside the bounds of the valid region, but such data has an undefined value.

vx_status VX_API_CALL 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*)

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

Parameters

in	*image*	The reference to the image object that contains the patch to map.
in	*rect*	The coordinates of image patch. The patch must be within the bounds of the image. (start_x, start_y) gives the coordinate of the topleft element inside the patch, while (end_x, end_y) give the coordinate of the bottomright element out of the patch. Must be 0 <= start < end.
in	*plane_index*	The plane index of the image object to be accessed.
out	*map_id*	The address of a vx_map_id variable where the function returns a map identifier. • (*map_id) must eventually be provided as the map_id parameter of a call to vxUnmapImagePatch.
out	*addr*	The address of a structure describing the memory layout of the image patch to access. The function fills the structure pointed by addr with the layout information that the application must consult to access the pixel data at address (*ptr). The layout of the mapped memory follows a row-major order: stride_x>0, stride_y>0 and stride_y >= stride_x * dim_x. If the image object being accessed was created via vxCreateImageFromHandle, then the returned memory layout will be the identical to that of the addressing structure provided when vxCreateImageFromHandle was called.
out	*ptr*	The address of a pointer that the function sets to the address where the requested data can be accessed. This returned (*ptr) address is only valid between the call to this function and the corresponding call to vxUnmapImagePatch. If image was created via vxCreateImageFromHandle then the returned address (*ptr) will be the address of the patch in the original pixel buffer provided when image was created.

in	*usage*	This declares the access mode for the image patch, using the vx_accessor_e enumeration. For uniform images, only VX_READ_ONLY is supported. • VX_READ_ONLY: after the function call, the content of the memory location pointed by (*ptr) contains the image patch data. Writing into this memory location is forbidden and its behavior is undefined. • VX_READ_AND_WRITE: after the function call, the content of the memory location pointed by (*ptr) contains the image patch data; writing into this memory is allowed only for the location of pixels only and will result in a modification of the written pixels in the image object once the patch is unmapped. Writing into a gap between pixels (when addr->stride_x > pixel size in bytes or addr->stride_y > addr->stride_x*addr->dim_x) is forbidden and its behavior is undefined. • VX_WRITE_ONLY: after the function call, the memory location pointed by (*ptr) contains undefined data; writing each pixel of the patch is required prior to unmapping. Pixels not written by the application before unmap will become undefined after unmap, even if they were well defined before map. Like for VX_READ_AND_WRITE, writing into a gap between pixels is forbidden and its behavior is undefined.
in	*mem_type*	A vx_memory_type_e enumeration that specifies the type of the memory where the image patch is requested to be mapped.
in	*flags*	An integer that allows passing options to the map operation. Use the vx_map_flag_e enumeration.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_OPTIMIZED_AWAY	This is a reference to a virtual image that cannot be accessed by the application.
VX_ERROR_INVALID_REFERENCE	The image reference is not actually an image reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

Note

The user may ask for data outside the bounds of the valid region, but such data has an undefined value.

Postcondition

vxUnmapImagePatch with same (*map_id) value.

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

Unmap and commit potential changes to a image object patch that were previously mapped. Unmapping an image patch invalidates the memory location from which the patch could be accessed by the application. Accessing this memory location after the unmap function completes has an undefined behavior.

Parameters

in	*image*	The reference to the image object to unmap.
out	*map_id*	The unique map identifier that was returned by vxMapImagePatch .

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	The image reference is not actually an image reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

Precondition

vxMapImagePatch with same map_id value

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

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

The sub-image refers to the data in the original image. Updates to this image update the parent image and reversely.

The function supports only channels that occupy an entire plane of a multi-planar images, as listed below. Other cases are not supported. VX_CHANNEL_Y from YUV4, IYUV, NV12, NV21 VX_CHANNEL_U from YUV4, IYUV VX_CHANNEL_V from YUV4, IYUV

Parameters

in	*img*	The reference to the parent image.
in	*channel*	The vx_channel_e channel to use.

Returns

An image reference vx_image to the sub-image. Any possible errors preventing a successful creation should be checked using vxGetStatus.

vx_status VX_API_CALL vxSetImageValidRectangle (vx_image *image,* const vx_rectangle_t ∗ *rect*)

Sets the valid rectangle for an image according to a supplied rectangle.

Note

Setting or changing the valid region from within a user node by means other than the call-back, for example by calling vxSetImageValidRectangle, might result in an incorrect valid region calculation by the framework.

Parameters

in	*image*	The reference to the image.
in	*rect*	The value to be set to the image valid rectangle. A NULL indicates that the valid region is the entire image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	The image is not a `vx_image`.
VX_ERROR_INVALID_PARAMETERS	The rect does not define a proper valid rectangle.

3.55 Object: LUT

3.55.1 Detailed Description

Defines the Look-Up Table Interface. A lookup table is an array that simplifies run-time computation by replacing computation with a simpler array indexing operation.

Typedefs

- typedef struct _vx_lut ∗ vx_lut

  The Look-Up Table (LUT) Object.

Enumerations

- enum vx_lut_attribute_e {
  VX_LUT_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_LUT << 8)) + 0x0,
  VX_LUT_COUNT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_LUT << 8)) + 0x1,
  VX_LUT_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_LUT << 8)) + 0x2,
  VX_LUT_OFFSET = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_LUT << 8)) + 0x3 }

  The Look-Up Table (LUT) attribute list.

Functions

- vx_status VX_API_CALL vxCopyLUT (vx_lut lut, void ∗user_ptr, vx_enum usage, vx_enum user_mem_type)

  Allows the application to copy from/into a LUT object.
- vx_lut VX_API_CALL vxCreateLUT (vx_context context, vx_enum data_type, vx_size count)

  Creates LUT object of a given type. The value of `VX_LUT_OFFSET` is equal to 0 for data_type = `VX_TYPE_UINT8`, and (vx_uint32)(count/2) for `VX_TYPE_INT16`.
- vx_status VX_API_CALL vxMapLUT (vx_lut lut, vx_map_id ∗map_id, void ∗∗ptr, vx_enum usage, vx_enum mem_type, vx_bitfield flags)

  Allows the application to get direct access to LUT object.
- vx_status VX_API_CALL vxQueryLUT (vx_lut lut, vx_enum attribute, void ∗ptr, vx_size size)

  Queries attributes from a LUT.
- vx_status VX_API_CALL vxReleaseLUT (vx_lut ∗lut)

  Releases a reference to a LUT object. The object may not be garbage collected until its total reference count is zero.
- vx_status VX_API_CALL vxUnmapLUT (vx_lut lut, vx_map_id map_id)

  Unmap and commit potential changes to LUT object that was previously mapped. Unmapping a LUT invalidates the memory location from which the LUT data could be accessed by the application. Accessing this memory location after the unmap function completes has an undefined behavior.

3.55.2 Enumeration Type Documentation

enum vx_lut_attribute_e

The Look-Up Table (LUT) attribute list.

Enumerator

VX_LUT_TYPE Indicates the value type of the LUT. Read-only. Use a `vx_enum`.

VX_LUT_COUNT Indicates the number of elements in the LUT. Read-only. Use a `vx_size`.

VX_LUT_SIZE Indicates the total size of the LUT in bytes. Read-only. Uses a `vx_size`.

VX_LUT_OFFSET Indicates the index of the input value = 0. Read-only. Uses a `vx_uint32`.

Definition at line 945 of file vx_types.h.

3.55.3 Function Documentation

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

Creates LUT object of a given type. The value of `VX_LUT_OFFSET` is equal to 0 for data_type = `VX_TYPE_UINT8`, and (vx_uint32)(count/2) for `VX_TYPE_INT16`.

Parameters

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

Note

data_type can only be VX_TYPE_UINT8 or VX_TYPE_INT16. If data_type is VX_TYPE_UINT8, count should be not greater than 256. If data_type is VX_TYPE_INT16, count should not be greater than 65536.

Returns

An LUT reference `vx_lut`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

vx_status VX_API_CALL vxReleaseLUT (vx_lut ∗ *lut*)

Releases a reference to a LUT object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*lut*	The pointer to the LUT to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If lut is not a `vx_lut`.

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

Queries attributes from a LUT.

Parameters

in	*lut*	The LUT to query.
in	*attribute*	The attribute to query. Use a `vx_lut_attribute_e` enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

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

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

Parameters

in	*lut*	The reference to the LUT object that is the source or the destination of the copy.
in	*user_ptr*	The address of the memory location where to store the requested data if the copy was requested in read mode, or from where to get the data to store into the LUT object if the copy was requested in write mode. In the user memory, the LUT is represented as a array with elements of the type corresponding to VX_LUT_TYPE, and with a number of elements equal to the value returned via VX_LUT_COUNT. The accessible memory must be large enough to contain this array: accessible memory in bytes $>=$ sizeof(data_element) $*$ count.
in	*usage*	This declares the effect of the copy with regard to the LUT object using the vx_accessor_e enumeration. Only VX_READ_ONLY and VX_WRITE_ONLY are supported: • VX_READ_ONLY means that data are copied from the LUT object into the user memory. • VX_WRITE_ONLY means that data are copied into the LUT object from the user memory.
in	*user_mem_type*	A vx_memory_type_e enumeration that specifies the memory type of the memory referenced by the user_addr.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	The LUT reference is not actually a LUT reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

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

Allows the application to get direct access to LUT object.

Parameters

in	*lut*	The reference to the LUT object to map.
out	*map_id*	The address of a vx_map_id variable where the function returns a map identifier. • ($*$map_id) must eventually be provided as the map_id parameter of a call to vxUnmapLUT.
out	*ptr*	The address of a pointer that the function sets to the address where the requested data can be accessed. In the mapped memory area, the LUT data are structured as an array with elements of the type corresponding to VX_LUT_TYPE, with a number of elements equal to the value returned via VX_LUT_COUNT. Accessing the memory out of the bound of this array is forbidden and has an undefined behavior. The returned ($*$ptr) address is only valid between the call to the function and the corresponding call to vxUnmapLUT.

in	*usage*	This declares the access mode for the LUT, using the vx_accessor_e enumeration. • VX_READ_ONLY: after the function call, the content of the memory location pointed by (*ptr) contains the LUT data. Writing into this memory location is forbidden and its behavior is undefined. • VX_READ_AND_WRITE: after the function call, the content of the memory location pointed by (*ptr) contains the LUT data; writing into this memory is allowed only for the location of entries and will result in a modification of the affected entries in the LUT object once the LUT is unmapped. • VX_WRITE_ONLY: after the function call, the memory location pointed by(*ptr) contains undefined data; writing each entry of LUT is required prior to unmapping. Entries not written by the application before unmap will become undefined after unmap, even if they were well defined before map.
in	*mem_type*	A vx_memory_type_e enumeration that specifies the type of the memory where the LUT is requested to be mapped.
in	*flags*	An integer that allows passing options to the map operation. Use 0 for this option.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	The LUT reference is not actually a LUT reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

Postcondition

vxUnmapLUT with same (*map_id) value.

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

Unmap and commit potential changes to LUT object that was previously mapped. Unmapping a LUT invalidates the memory location from which the LUT data could be accessed by the application. Accessing this memory location after the unmap function completes has an undefined behavior.

Parameters

in	*lut*	The reference to the LUT object to unmap.
out	*map_id*	The unique map identifier that was returned when calling vxMapLUT .

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	The LUT reference is not actually a LUT reference.

VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

Precondition

`vxMapLUT` returning the same map_id value

3.56 Object: Matrix

3.56.1 Detailed Description

Defines the Matrix Object Interface.

Typedefs

- typedef struct _vx_matrix * vx_matrix

    The Matrix Object. An MxN matrix of some unit type.

Enumerations

- enum vx_matrix_attribute_e {
  VX_MATRIX_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x0,
  VX_MATRIX_ROWS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x1,
  VX_MATRIX_COLUMNS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x2,
  VX_MATRIX_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x3,
  VX_MATRIX_ORIGIN = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x4,
  VX_MATRIX_PATTERN = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x5 }

    The matrix attributes.

Functions

- vx_status VX_API_CALL vxCopyMatrix (vx_matrix matrix, void *user_ptr, vx_enum usage, vx_enum user_-mem_type)

    Allows the application to copy from/into a matrix object.
- vx_matrix VX_API_CALL vxCreateMatrix (vx_context c, vx_enum data_type, vx_size columns, vx_size rows)

    Creates a reference to a matrix object.
- vx_matrix VX_API_CALL vxCreateMatrixFromPattern (vx_context context, vx_enum pattern, vx_size columns, vx_size rows)

    Creates a reference to a matrix object from a boolean pattern.
- vx_status VX_API_CALL vxQueryMatrix (vx_matrix mat, vx_enum attribute, void *ptr, vx_size size)

    Queries an attribute on the matrix object.
- vx_status VX_API_CALL vxReleaseMatrix (vx_matrix *mat)

    Releases a reference to a matrix object. The object may not be garbage collected until its total reference count is zero.

3.56.2 Enumeration Type Documentation

enum vx_matrix_attribute_e

The matrix attributes.

Enumerator

VX_MATRIX_TYPE The value type of the matrix. Read-only. Use a `vx_enum` parameter.

VX_MATRIX_ROWS The M dimension of the matrix. Read-only. Use a `vx_size` parameter.

VX_MATRIX_COLUMNS The N dimension of the matrix. Read-only. Use a `vx_size` parameter.

VX_MATRIX_SIZE The total size of the matrix in bytes. Read-only. Use a `vx_size` parameter.

VX_MATRIX_ORIGIN The origin of the matrix with a default value of [floor(VX_MATRIX_COLUMNS/2), floor(VX_MATRIX_ROWS/2)]. Read-only. Use a `vx_coordinates2d_t` parameter.

VX_MATRIX_PATTERN The pattern of the matrix. See `vx_pattern_e` . Read-only. Use a `vx_enum` parameter.

Definition at line 1008 of file vx_types.h.

3.56.3 Function Documentation

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

Creates a reference to a matrix object.

Parameters

in	*c*	The reference to the overall context.
in	*data_type*	The unit format of the matrix. VX_TYPE_UINT8 or VX_TYPE_INT32 or VX_TYPE_FLOAT32.
in	*columns*	The first dimensionality.
in	*rows*	The second dimensionality.

Returns

An matrix reference vx_matrix. Any possible errors preventing a successful creation should be checked using vxGetStatus.

vx_status VX_API_CALL vxReleaseMatrix (vx_matrix * *mat*)

Releases a reference to a matrix object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*mat*	The matrix reference to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If mat is not a vx_matrix.

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

Queries an attribute on the matrix object.

Parameters

in	*mat*	The matrix object to set.
in	*attribute*	The attribute to query. Use a vx_matrix_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

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

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

Parameters

in	*matrix*	The reference to the matrix object that is the source or the destination of the copy.
in	*user_ptr*	The address of the memory location where to store the requested data if the copy was requested in read mode, or from where to get the data to store into the matrix object if the copy was requested in write mode. In the user memory, the matrix is structured as a row-major 2D array with elements of the type corresponding to `VX_MATRIX_TYPE`, with a number of rows corresponding to `VX_MATRIX_ROWS` and a number of columns corresponding to `VX_MATRIX_COLUMNS`. The accessible memory must be large enough to contain this 2D array: accessible memory in bytes >= sizeof(data_element) * rows * columns.
in	*usage*	This declares the effect of the copy with regard to the matrix object using the `vx_accessor_e` enumeration. Only `VX_READ_ONLY` and `VX_WRITE_ONLY` are supported: • `VX_READ_ONLY` means that data are copied from the matrix object into the user memory. • `VX_WRITE_ONLY` means that data are copied into the matrix object from the user memory.
in	*user_mem_type*	A `vx_memory_type_e` enumeration that specifies the memory type of the memory referenced by the user_addr.

Returns

A `vx_status_e` enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	The matrix reference is not actually a matrix reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

vx_matrix VX_API_CALL vxCreateMatrixFromPattern (vx_context *context*, vx_enum *pattern*, vx_size *columns*, vx_size *rows*)

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

The matrix created by this function is of type `vx_uint8`, with the value 0 representing False, and the value 255 representing True. It supports patterns described below. See `vx_pattern_e`.

- VX_PATTERN_BOX is a matrix with dimensions equal to the given number of rows and columns, and all cells equal to 255. Dimensions of 3x3 and 5x5 must be supported.
- VX_PATTERN_CROSS is a matrix with dimensions equal to the given number of rows and columns, which both must be odd numbers. All cells in the center row and center column are equal to 255, and the rest are equal to zero. Dimensions of 3x3 and 5x5 must be supported.
- VX_PATTERN_DISK is an RxC matrix, where R and C are odd and cell (c, r) is 255 if:

  (r-R/2 + 0.5)^2 / (R/2)^2 + (c-C/2 + 0.5)^2/(C/2)^2 is less than or equal to 1,

  and 0 otherwise.
- VX_PATTERN_OTHER is any other pattern than the above (matrix created is still binary, with a value of 0 or 255).

If the matrix was created via `vxCreateMatrixFromPattern`, this attribute must be set to the appropriate pattern enum. Otherwise the attribute must be set to VX_PATTERN_OTHER. The vx_matrix objects returned by this function are read-only. The behavior when attempting to modify such a matrix is undefined.

Parameters

in	*context*	The reference to the overall context.
in	*pattern*	The pattern of the matrix. See VX_MATRIX_PATTERN.
in	*columns*	The first dimensionality.
in	*rows*	The second dimensionality.

Returns

An matrix reference vx_matrix of type vx_uint8. Any possible errors preventing a successful creation should be checked using vxGetStatus.

3.57 Object: Pyramid

3.57.1 Detailed Description

Defines the Image Pyramid Object Interface. A Pyramid object in OpenVX represents a collection of related images. Typically, these images are created by either downscaling or upscaling a *base image*, contained in level zero of the pyramid. Successive levels of the pyramid increase or decrease in size by a factor given by the `VX_PYRAMID_SCALE` attribute. For instance, in a pyramid with 3 levels and `VX_SCALE_PYRAMID_HALF`, the level one image is one-half the width and one-half the height of the level zero image, and the level two image is one-quarter the width and one quarter the height of the level zero image. When downscaling or upscaling results in a non-integral number of pixels at any level, fractional pixels always get rounded up to the nearest integer. (E.g., a 3-level image pyramid beginning with level zero having a width of 9 and a scaling of `VX_SCALE_PYRAMID_HALF` results in the level one image with a width of $5 = \mathbf{ceil}(9 * 0.5)$ and a level two image with a width of $3 = \mathbf{ceil}(5 * 0.5)$. Position (r_N, c_N) at level N corresponds to position $(r_{N-1}/\mathbf{scale}, c_{N-1}/\mathbf{scale})$ at level $N-1$.

Macros

- #define VX_SCALE_PYRAMID_HALF (0.5f)

  Use to indicate a half-scale pyramid.
- #define VX_SCALE_PYRAMID_ORB ((vx_float32)0.8408964f)

  Use to indicate a ORB scaled pyramid whose scaling factor is $\frac{1}{\sqrt[4]{2}}$.

Typedefs

- typedef struct _vx_pyramid $*$ vx_pyramid

  The Image Pyramid object. A set of scaled images.

Enumerations

- enum vx_pyramid_attribute_e {
  VX_PYRAMID_LEVELS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x0,
  VX_PYRAMID_SCALE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x1,
  VX_PYRAMID_WIDTH = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x2,
  VX_PYRAMID_HEIGHT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x3,
  VX_PYRAMID_FORMAT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x4 }

  The pyramid object attributes.

Functions

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

  Creates a reference to a pyramid object of the supplied number of levels.
- vx_pyramid VX_API_CALL vxCreateVirtualPyramid (vx_graph graph, vx_size levels, vx_float32 scale, vx_uint32 width, vx_uint32 height, vx_df_image format)

  Creates a reference to a virtual pyramid object of the supplied number of levels.
- vx_image VX_API_CALL vxGetPyramidLevel (vx_pyramid pyr, vx_uint32 index)

  Retrieves a level of the pyramid as a `vx_image`, which can be used elsewhere in OpenVX. A call to vxReleaseImage is necessary to release an image for each call of vxGetPyramidLevel.
- vx_status VX_API_CALL vxQueryPyramid (vx_pyramid pyr, vx_enum attribute, void $*$ptr, vx_size size)

  Queries an attribute from an image pyramid.
- vx_status VX_API_CALL vxReleasePyramid (vx_pyramid $*$pyr)

  Releases a reference to a pyramid object. The object may not be garbage collected until its total reference count is zero.

3.57.2 Enumeration Type Documentation

enum vx_pyramid_attribute_e

The pyramid object attributes.

Enumerator

VX_PYRAMID_LEVELS The number of levels of the pyramid. Read-only. Use a vx_size parameter.

VX_PYRAMID_SCALE The scale factor between each level of the pyramid. Read-only. Use a vx_float32 parameter.

VX_PYRAMID_WIDTH The width of the 0th image in pixels. Read-only. Use a vx_uint32 parameter.

VX_PYRAMID_HEIGHT The height of the 0th image in pixels. Read-only. Use a vx_uint32 parameter.

VX_PYRAMID_FORMAT The vx_df_image_e format of the image. Read-only. Use a vx_df_image parameter.

Definition at line 1045 of file vx_types.h.

3.57.3 Function Documentation

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

Creates a reference to a pyramid object of the supplied number of levels.

Parameters

in	*context*	The reference to the overall context.
in	*levels*	The number of levels desired. This is required to be a non-zero value.
in	*scale*	Used to indicate the scale between pyramid levels. This is required to be a non-zero positive value. Only permissible values are VX_SCALE_PYRAMID_HALF or VX_SCALE_PYRAMID_ORB.
in	*width*	The width of the 0th level image in pixels.
in	*height*	The height of the 0th level image in pixels.
in	*format*	The format of all images in the pyramid. NV12, NV21, IYUV, UYVY and YUYV formats are not supported.

Returns

A pyramid reference vx_pyramid containing the sub-images. Any possible errors preventing a successful creation should be checked using vxGetStatus.

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

Creates a reference to a virtual pyramid object of the supplied number of levels.

Virtual Pyramids can be used to connect Nodes together when the contents of the pyramids will not be accessed by the user of the API. All of the following constructions are valid:

```
vx_context context = vxCreateContext();
vx_graph graph = vxCreateGraph(context);
vx_pyramid virt[] = {
    vxCreateVirtualPyramid(graph, 4, VX_SCALE_PYRAMID_HALF, 0, 0
      , VX_DF_IMAGE_VIRT), // no dimension and format specified for level 0
    vxCreateVirtualPyramid(graph, 4, VX_SCALE_PYRAMID_HALF, 640,
       480, VX_DF_IMAGE_VIRT), // no format specified.
    vxCreateVirtualPyramid(graph, 4, VX_SCALE_PYRAMID_HALF, 640,
       480, VX_DF_IMAGE_U8), // no access
};
```

Parameters

in	*graph*	The reference to the parent graph.
in	*levels*	The number of levels desired. This is required to be a non-zero value.
in	*scale*	Used to indicate the scale between pyramid levels. This is required to be a non-zero positive value. Only permissible values are `VX_SCALE_PYRAMID_HALF` or `VX_SCALE_PYRAMID_ORB`.
in	*width*	The width of the 0th level image in pixels. This may be set to zero to indicate to the interface that the value is unspecified.
in	*height*	The height of the 0th level image in pixels. This may be set to zero to indicate to the interface that the value is unspecified.
in	*format*	The format of all images in the pyramid. This may be set to `VX_DF_IMAGE_VIRT` to indicate that the format is unspecified.

Returns

A pyramid reference `vx_pyramid`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

Note

Images extracted with `vxGetPyramidLevel` behave as Virtual Images and cause `vxMapImagePatch` to return errors.

vx_status VX_API_CALL vxReleasePyramid (vx_pyramid * *pyr*)

Releases a reference to a pyramid object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*pyr*	The pointer to the pyramid to release.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If pyr is not a `vx_pyramid`.

Postcondition

After returning from this function the reference is zeroed.

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

Queries an attribute from an image pyramid.

Parameters

in	*pyr*	The pyramid to query.
in	*attribute*	The attribute for which to query. Use a `vx_pyramid_attribute_e` enumeration.

out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

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

Retrieves a level of the pyramid as a vx_image, which can be used elsewhere in OpenVX. A call to vxRelease-Image is necessary to release an image for each call of vxGetPyramidLevel.

Parameters

in	*pyr*	The pyramid object.
in	*index*	The index of the level, such that index is less than levels.

Returns

A vx_image reference. Any possible errors preventing a successful creation should be checked using vx-GetStatus.

Return values

0	Indicates that the index or the object is invalid.

3.58 Object: Remap

3.58.1 Detailed Description

Defines the Remap Object Interface.

Typedefs

- typedef struct _vx_remap ∗ vx_remap

  The remap table Object. A remap table contains per-pixel mapping of output pixels to input pixels.

Enumerations

- enum vx_remap_attribute_e {
  VX_REMAP_SOURCE_WIDTH = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REMAP << 8)) + 0x0,
  VX_REMAP_SOURCE_HEIGHT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REMAP << 8)) + 0x1,
  VX_REMAP_DESTINATION_WIDTH = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REMAP << 8)) + 0x2,
  VX_REMAP_DESTINATION_HEIGHT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REMAP << 8)) + 0x3 }

  The remap object attributes.

Functions

- vx_remap VX_API_CALL vxCreateRemap (vx_context context, vx_uint32 src_width, vx_uint32 src_height, vx_uint32 dst_width, vx_uint32 dst_height)

  Creates a remap table object.
- vx_status VX_API_CALL vxGetRemapPoint (vx_remap table, vx_uint32 dst_x, vx_uint32 dst_y, vx_float32 ∗src_x, vx_float32 ∗src_y)

  Retrieves the source pixel point from a destination pixel.
- vx_status VX_API_CALL vxQueryRemap (vx_remap r, vx_enum attribute, void ∗ptr, vx_size size)

  Queries attributes from a Remap table.
- vx_status VX_API_CALL vxReleaseRemap (vx_remap ∗table)

  Releases a reference to a remap table object. The object may not be garbage collected until its total reference count is zero.
- vx_status VX_API_CALL vxSetRemapPoint (vx_remap table, vx_uint32 dst_x, vx_uint32 dst_y, vx_float32 src_x, vx_float32 src_y)

  Assigns a destination pixel mapping to the source pixel.

3.58.2 Enumeration Type Documentation

enum vx_remap_attribute_e

The remap object attributes.

Enumerator

VX_REMAP_SOURCE_WIDTH The source width. Read-only. Use a `vx_uint32` parameter.

VX_REMAP_SOURCE_HEIGHT The source height. Read-only. Use a `vx_uint32` parameter.

VX_REMAP_DESTINATION_WIDTH The destination width. Read-only. Use a `vx_uint32` parameter.

VX_REMAP_DESTINATION_HEIGHT The destination height. Read-only. Use a `vx_uint32` parameter.

Definition at line 1061 of file vx_types.h.

3.58.3 Function Documentation

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

Creates a remap table object.

Parameters

in	*context*	The reference to the overall context.
in	*src_width*	Width of the source image in pixel.
in	*src_height*	Height of the source image in pixels.
in	*dst_width*	Width of the destination image in pixels.
in	*dst_height*	Height of the destination image in pixels.

Returns

A remap reference `vx_remap`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

vx_status VX_API_CALL vxReleaseRemap (vx_remap ∗ *table*)

Releases a reference to a remap table object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*table*	The pointer to the remap table to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If table is not a `vx_remap`.

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

Assigns a destination pixel mapping to the source pixel.

Parameters

in	*table*	The remap table reference.
in	*dst_x*	The destination x coordinate.
in	*dst_y*	The destination y coordinate.
in	*src_x*	The source x coordinate in float representation to allow interpolation.
in	*src_y*	The source y coordinate in float representation to allow interpolation.

Returns

A `vx_status_e` enumeration.

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

Retrieves the source pixel point from a destination pixel.

Parameters

`in`	*table*	The remap table reference.
`in`	*dst_x*	The destination x coordinate.
`in`	*dst_y*	The destination y coordinate.
`out`	*src_x*	The pointer to the location to store the source x coordinate in float representation to allow interpolation.
`out`	*src_y*	The pointer to the location to store the source y coordinate in float representation to allow interpolation.

Returns

A `vx_status_e` enumeration.

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

Queries attributes from a Remap table.

Parameters

`in`	*r*	The remap to query.
`in`	*attribute*	The attribute to query. Use a `vx_remap_attribute_e` enumeration.
`out`	*ptr*	The location at which to store the resulting value.
`in`	*size*	The size in bytes of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

3.59 Object: Scalar

3.59.1 Detailed Description

Defines the Scalar Object interface.

Typedefs

- typedef struct _vx_scalar ∗ vx_scalar

    An opaque reference to a scalar.

Enumerations

- enum vx_scalar_attribute_e { VX_SCALAR_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_SCALAR << 8)) + 0x0 }

    The scalar attributes list.

Functions

- vx_status VX_API_CALL vxCopyScalar (vx_scalar scalar, void ∗user_ptr, vx_enum usage, vx_enum user_mem_type)

    Allows the application to copy from/into a scalar object.
- vx_scalar VX_API_CALL vxCreateScalar (vx_context context, vx_enum data_type, const void ∗ptr)

    Creates a reference to a scalar object. Also see Node Parameters.
- vx_status VX_API_CALL vxQueryScalar (vx_scalar scalar, vx_enum attribute, void ∗ptr, vx_size size)

    Queries attributes from a scalar.
- vx_status VX_API_CALL vxReleaseScalar (vx_scalar ∗scalar)

    Releases a reference to a scalar object. The object may not be garbage collected until its total reference count is zero.

3.59.2 Typedef Documentation

typedef struct _vx_scalar∗ vx_scalar

An opaque reference to a scalar.
A scalar can be up to 64 bits wide.

See Also

vxCreateScalar

Definition at line 183 of file vx_types.h.

3.59.3 Enumeration Type Documentation

enum vx_scalar_attribute_e

The scalar attributes list.

Enumerator

VX_SCALAR_TYPE Queries the type of atomic that is contained in the scalar. Read-only. Use a `vx_enum` parameter.

Definition at line 937 of file vx_types.h.

3.59.4 Function Documentation

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

Creates a reference to a scalar object. Also see Node Parameters.

Parameters

`in`	*context*	The reference to the system context.
`in`	*data_type*	The `vx_type_e` of the scalar. Must be greater than `VX_TYPE_INVALID` and less than `VX_TYPE_SCALAR_MAX`.
`in`	*ptr*	The pointer to the initial value of the scalar.

Returns

A scalar reference `vx_scalar`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

vx_status VX_API_CALL vxReleaseScalar (vx_scalar ∗ *scalar*)

Releases a reference to a scalar object. The object may not be garbage collected until its total reference count is zero.

Parameters

`in`	*scalar*	The pointer to the scalar to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If scalar is not a `vx_scalar`.

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

Queries attributes from a scalar.

Parameters

`in`	*scalar*	The scalar object.
`in`	*attribute*	The enumeration to query. Use a `vx_scalar_attribute_e` enumeration.
`out`	*ptr*	The location at which to store the resulting value.
`in`	*size*	The size of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

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

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

Parameters

in	*scalar*	The reference to the scalar object that is the source or the destination of the copy.
in	*user_ptr*	The address of the memory location where to store the requested data if the copy was requested in read mode, or from where to get the data to store into the scalar object if the copy was requested in write mode. In the user memory, the scalar is a variable of the type corresponding to `VX_SCALAR_TYPE`. The accessible memory must be large enough to contain this variable.
in	*usage*	This declares the effect of the copy with regard to the scalar object using the `vx_accessor_e` enumeration. Only `VX_READ_ONLY` and `VX_WRITE_ONLY` are supported: • `VX_READ_ONLY` means that data are copied from the scalar object into the user memory. • `VX_WRITE_ONLY` means that data are copied into the scalar object from the user memory.
in	*user_mem_type*	A `vx_memory_type_e` enumeration that specifies the memory type of the memory referenced by the user_addr.

Returns

A `vx_status_e` enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	The scalar reference is not actually a scalar reference.
VX_ERROR_INVALID_PARAMETERS	An other parameter is incorrect.

3.60 Object: Threshold

3.60.1 Detailed Description

Defines the Threshold Object Interface.

Typedefs

- typedef struct _vx_threshold ∗ vx_threshold

  The Threshold Object. A thresholding object contains the types and limit values of the thresholding required.

Enumerations

- enum vx_threshold_attribute_e {
  VX_THRESHOLD_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x0,
  VX_THRESHOLD_THRESHOLD_VALUE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x1,
  VX_THRESHOLD_THRESHOLD_LOWER = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x2,
  VX_THRESHOLD_THRESHOLD_UPPER = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x3,
  VX_THRESHOLD_TRUE_VALUE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x4,
  VX_THRESHOLD_FALSE_VALUE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x5,
  VX_THRESHOLD_DATA_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x6 }

  The threshold attributes.
- enum vx_threshold_type_e {
  VX_THRESHOLD_TYPE_BINARY = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_THRESHOLD_TYPE << 12)) + 0x0,
  VX_THRESHOLD_TYPE_RANGE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_THRESHOLD_TYPE << 12)) + 0x1 }

  The Threshold types.

Functions

- vx_threshold VX_API_CALL vxCreateThreshold (vx_context c, vx_enum thresh_type, vx_enum data_type)

  Creates a reference to a threshold object of a given type.
- vx_status VX_API_CALL vxQueryThreshold (vx_threshold thresh, vx_enum attribute, void ∗ptr, vx_size size)

  Queries an attribute on the threshold object.
- vx_status VX_API_CALL vxReleaseThreshold (vx_threshold ∗thresh)

  Releases a reference to a threshold object. The object may not be garbage collected until its total reference count is zero.
- vx_status VX_API_CALL vxSetThresholdAttribute (vx_threshold thresh, vx_enum attribute, const void ∗ptr, vx_size size)

  Sets attributes on the threshold object.

3.60.2 Enumeration Type Documentation

enum vx_threshold_type_e

The Threshold types.

Enumerator

VX_THRESHOLD_TYPE_BINARY A threshold with only 1 value.

VX_THRESHOLD_TYPE_RANGE A threshold with 2 values (upper/lower). Use with Canny Edge Detection.

Definition at line 978 of file vx_types.h.

enum vx_threshold_attribute_e

The threshold attributes.

Enumerator

VX_THRESHOLD_TYPE The value type of the threshold. Read-only. Use a `vx_enum` parameter. Will contain a `vx_threshold_type_e`.

VX_THRESHOLD_THRESHOLD_VALUE The value of the single threshold. Read-write. Use a `vx_int32` parameter.

VX_THRESHOLD_THRESHOLD_LOWER The value of the lower threshold. Read-write. Use a `vx_int32` parameter.

VX_THRESHOLD_THRESHOLD_UPPER The value of the higher threshold. Read-write. Use a `vx_int32` parameter.

VX_THRESHOLD_TRUE_VALUE The value of the TRUE threshold (default value is 255). Read-write. Use a `vx_int32` parameter.

VX_THRESHOLD_FALSE_VALUE The value of the FALSE threshold (default value is 0). Read-write. Use a `vx_int32` parameter.

VX_THRESHOLD_DATA_TYPE The data type of the threshold's value. Read-only. Use a `vx_enum` parameter. Will contain a `vx_type_e`.

Definition at line 988 of file vx_types.h.

3.60.3 Function Documentation

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

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

Parameters

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

Returns

An threshold reference `vx_threshold`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

vx_status VX_API_CALL vxReleaseThreshold (vx_threshold ∗ *thresh*)

Releases a reference to a threshold object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*thresh*	The pointer to the threshold to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If thresh is not a `vx_threshold`.

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

Sets attributes on the threshold object.

Parameters

`in`	*thresh*	The threshold object to set.
`in`	*attribute*	The attribute to modify. Use a `vx_threshold_attribute_e` enumeration.
`in`	*ptr*	The pointer to the value to which to set the attribute.
`in`	*size*	The size of the data pointed to by *ptr*.

Returns

A `vx_status_e` enumeration.

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

Queries an attribute on the threshold object.

Parameters

`in`	*thresh*	The threshold object to set.
`in`	*attribute*	The attribute to query. Use a `vx_threshold_attribute_e` enumeration.
`out`	*ptr*	The location at which to store the resulting value.
`in`	*size*	The size of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

3.61 Object: ObjectArray

3.61.1 Detailed Description

An opaque array object that could be an array of any data-object (not data-type) of OpenVX except Delay and ObjectArray objects. ObjectArray is a strongly-typed container of OpenVX data-objects. ObjectArray refers to the collection of similar data-objects as a single entity that can be created or assigned as inputs/outputs and as a single entity. In addition, a single object from the collection can be accessed individually by getting its reference. The single object remains as part of the ObjectArray through its entire life cycle.

Typedefs

- typedef struct _vx_object_array ∗ vx_object_array

    The ObjectArray Object. ObjectArray is a strongly-typed container of OpenVX data-objects.

Enumerations

- enum vx_object_array_attribute_e {
  VX_OBJECT_ARRAY_ITEMTYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_OBJECT_ARRAY << 8)) + 0x0,
  VX_OBJECT_ARRAY_NUMITEMS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_OBJECT_ARRAY << 8)) + 0x1 }

    The ObjectArray object attributes.

Functions

- vx_object_array VX_API_CALL vxCreateObjectArray (vx_context context, vx_reference exemplar, vx_size count)

    Creates a reference to an ObjectArray of count objects.
- vx_object_array VX_API_CALL vxCreateVirtualObjectArray (vx_graph graph, vx_reference exemplar, vx_size count)

    Creates an opaque reference to a virtual ObjectArray with no direct user access.
- vx_reference VX_API_CALL vxGetObjectArrayItem (vx_object_array arr, vx_uint32 index)

    Retrieves the reference to the OpenVX Object in location index of the ObjectArray.
- vx_status VX_API_CALL vxQueryObjectArray (vx_object_array arr, vx_enum attribute, void ∗ptr, vx_size size)

    Queries an atribute from the ObjectArray.
- vx_status VX_API_CALL vxReleaseObjectArray (vx_object_array ∗arr)

    Releases a reference of an ObjectArray object.

3.61.2 Enumeration Type Documentation

enum vx_object_array_attribute_e

The ObjectArray object attributes.

Enumerator

VX_OBJECT_ARRAY_ITEMTYPE The type of the ObjectArray items. Read-only. Use a `vx_enum` parameter.

VX_OBJECT_ARRAY_NUMITEMS The number of items in the ObjectArray. Read-only. Use a `vx_enum` parameter.

Definition at line 1089 of file vx_types.h.

3.61.3 Function Documentation

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

Creates a reference to an ObjectArray of count objects.

It uses the metadata of the exemplar to determine the object attributes, ignoring the object data. It does not alter the exemplar or keep or release the reference to the exemplar. For the definition of supported attributes see `vxSetMetaFormatAttribute`. In case the exemplar is a virtual object it must be of immutable metadata, thus it is not allowed to be dimensionless or formatless.

Parameters

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

Returns

An ObjectArray reference `vx_object_array`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`. Data objects are not initialized by this function.

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

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

This function creates an ObjectArray of count objects with similar behavior as `vxCreateObjectArray`. The only difference is that the objects that are created are virtual in the given graph.

Parameters

`in`	*graph*	Reference to the graph where to create the virtual ObjectArray.
`in`	*exemplar*	The exemplar object that defines the type of object in the ObjectArray. Only exemplar type of `vx_image`, `vx_array` and `vx_pyramid` are allowed.
`in`	*count*	Number of Objects to create in the ObjectArray.

Returns

A ObjectArray reference `vx_object_array`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

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

Retrieves the reference to the OpenVX Object in location index of the ObjectArray.

This is a vx_reference, which can be used elsewhere in OpenVX. A call to vxRelease<Object> or `vxReleaseReference` is necessary to release the Object for each call to this function.

Parameters

`in`	*arr*	The ObjectArray.
`in`	*index*	The index of the object in the ObjectArray.

Returns

A reference to an OpenVX data object.

vx_status VX_API_CALL vxReleaseObjectArray (vx_object_array $*$ *arr*)

Releases a reference of an ObjectArray object.

The object may not be garbage collected until its total reference and its contained objects count is zero. After returning from this function the reference is zeroed/cleared.

Parameters

`in`	*arr*	The pointer to the ObjectArray to release.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If arr is not a `vx_object_array`.

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

Queries an atribute from the ObjectArray.

Parameters

`in`	*arr*	The reference to the ObjectArray.
`in`	*attribute*	The attribute to query. Use a `vx_object_array_attribute_e`.
`out`	*ptr*	The location at which to store the resulting value.
`in`	*size*	The size in bytes of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the *arr* is not a `vx_object_array`.
VX_ERROR_NOT_SUPPORTED	If the *attribute* is not a value supported on this implementation.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

3.62 Administrative Features

3.62.1 Detailed Description

Defines the Administrative Features of OpenVX. These features are administrative in nature and require more understanding and are more complex to use.

Modules

- Advanced Objects

  Defines the Advanced Objects of OpenVX.
- Advanced Framework API

  Describes components that are considered to be advanced.

3.63 Advanced Objects

3.63.1 Detailed Description

Defines the Advanced Objects of OpenVX.

Modules

- Object: Array (Advanced)

    Defines the advanced features of the Array Interface.
- Object: Node (Advanced)

    Defines the advanced features of the Node Interface.
- Object: Delay

    Defines the Delay Object interface.
- Object: Kernel

    Defines the Kernel Object and Interface.
- Object: Parameter

    Defines the Parameter Object interface.

3.64 Object: Array (Advanced)

3.64.1 Detailed Description

Defines the advanced features of the Array Interface.

Functions

- vx_enum VX_API_CALL vxRegisterUserStruct (vx_context context, vx_size size)

  Registers user-defined structures to the context.

3.64.2 Function Documentation

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

Registers user-defined structures to the context.

Parameters

in	*context*	The reference to the implementation context.
in	*size*	The size of user struct in bytes.

Returns

A `vx_enum` value that is a type given to the User to refer to their custom structure when declaring a `vx_array` of that structure.

Return values

VX_TYPE_INVALID	If the namespace of types has been exhausted.

Note

This call should only be used once within the lifetime of a context for a specific structure.

3.65 Object: Node (Advanced)

3.65.1 Detailed Description

Defines the advanced features of the Node Interface.

Modules

- Node: Border Modes

  Defines the border mode behaviors.

Functions

- vx_node VX_API_CALL vxCreateGenericNode (vx_graph graph, vx_kernel kernel)

  Creates a reference to a node object for a given kernel.

3.65.2 Function Documentation

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

Creates a reference to a node object for a given kernel.

This node has no references assigned as parameters after completion. The client is then required to set these parameters manually by `vxSetParameterByIndex`. When clients supply their own node creation functions (for use with User Kernels), this is the API to use along with the parameter setting API.

Parameters

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

Returns

A node reference `vx_node`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

Note

A call to this API sets all parameters to NULL.

Postcondition

Call `vxSetParameterByIndex` for as many parameters as needed to be set.

3.66 Node: Border Modes

3.66.1 Detailed Description

Defines the border mode behaviors. Border Mode behavior is set as an attribute of the node, not as a direct parameter to the kernel. This allows clients to *set-and-forget* the modes of any particular node that supports border modes. All nodes shall support `VX_BORDER_UNDEFINED`.

Data Structures

- struct vx_border_t

  Use with the enumeration `VX_NODE_BORDER` to set the border mode behavior of a node that supports borders. *More...*

Enumerations

- enum vx_border_e {
  VX_BORDER_UNDEFINED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_BORDER << 12)) + 0x0,
  VX_BORDER_CONSTANT = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_BORDER << 12)) + 0x1,
  VX_BORDER_REPLICATE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_BORDER << 12)) + 0x2 }

  The border mode list.
- enum vx_border_policy_e {
  VX_BORDER_POLICY_DEFAULT_TO_UNDEFINED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_BORDER_POLICY << 12)) + 0x0,
  VX_BORDER_POLICY_RETURN_ERROR = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_BORDER_POLICY << 12)) + 0x1 }

  The unsupported border mode policy list.

3.66.2 Data Structure Documentation

struct vx_border_t

Use with the enumeration `VX_NODE_BORDER` to set the border mode behavior of a node that supports borders.

If the indicated border mode is not supported, an error `VX_ERROR_NOT_SUPPORTED` will be reported either at the time the `VX_NODE_BORDER` is set or at the time of graph verification.

Definition at line 1520 of file vx_types.h.

Data Fields

vx_enum	mode	See `vx_border_e`.
vx_pixel_value_t	constant_value	For the mode `VX_BORDER_CONSTANT`, this union contains the value of out-of-bound pixels.

3.66.3 Enumeration Type Documentation

enum vx_border_e

The border mode list.

Enumerator

VX_BORDER_UNDEFINED No defined border mode behavior is given.

VX_BORDER_CONSTANT For nodes that support this behavior, a constant value is *filled-in* when accessing out-of-bounds pixels.

VX_BORDER_REPLICATE For nodes that support this behavior, a replication of the nearest edge pixels value is given for out-of-bounds pixels.

Definition at line 1251 of file vx_types.h.

enum vx_border_policy_e

The unsupported border mode policy list.

Enumerator

VX_BORDER_POLICY_DEFAULT_TO_UNDEFINED Use VX_BORDER_UNDEFINED instead of unsupported border modes.

VX_BORDER_POLICY_RETURN_ERROR Return VX_ERROR_NOT_SUPPORTED for unsupported border modes.

Definition at line 1267 of file vx_types.h.

3.67 Object: Delay

3.67.1 Detailed Description

Defines the Delay Object interface. A Delay is an opaque object that contains a manually-controlled, temporally-delayed list of objects. A Delay cannot be an output of a kernel. Also, aging of a Delay (see vxAgeDelay) cannot be performed during graph execution. Supported delay object types include:

- `VX_TYPE_ARRAY`,
- `VX_TYPE_IMAGE`,
- `VX_TYPE_PYRAMID`,
- `VX_TYPE_MATRIX`,
- `VX_TYPE_CONVOLUTION`,
- `VX_TYPE_DISTRIBUTION`,
- `VX_TYPE_REMAP`,
- `VX_TYPE_LUT`,
- `VX_TYPE_THRESHOLD`,
- `VX_TYPE_SCALAR`

Typedefs

- typedef struct _vx_delay ∗ vx_delay

  The delay object. This is like a ring buffer of objects that is maintained by the OpenVX implementation.

Enumerations

- enum vx_delay_attribute_e {
  VX_DELAY_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DELAY << 8)) + 0x0,
  VX_DELAY_SLOTS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DELAY << 8)) + 0x1 }

  The delay attribute list.

Functions

- vx_status VX_API_CALL vxAgeDelay (vx_delay delay)

  Ages the internal delay ring by one. This means that once this API is called the reference from index 0 will go to index -1 and so forth until $-count+1$ *is reached. This last object will become 0. Once the delay has been aged, it updates the reference in any associated nodes. Here* $count$ *is the number of slots in delay ring.*
- vx_delay VX_API_CALL vxCreateDelay (vx_context context, vx_reference exemplar, vx_size slots)

  Creates a Delay object.
- vx_reference VX_API_CALL vxGetReferenceFromDelay (vx_delay delay, vx_int32 index)

  Retrieves a reference from a delay object.
- vx_status VX_API_CALL vxQueryDelay (vx_delay delay, vx_enum attribute, void ∗ptr, vx_size size)

  Queries a `vx_delay` object attribute.
- vx_status VX_API_CALL vxReleaseDelay (vx_delay ∗delay)

  Releases a reference to a delay object. The object may not be garbage collected until its total reference count is zero.

3.67.2 Typedef Documentation

typedef struct _vx_delay∗ vx_delay

The delay object. This is like a ring buffer of objects that is maintained by the OpenVX implementation.

See Also

vxCreateDelay

Definition at line 234 of file vx_types.h.

3.67.3 Enumeration Type Documentation

enum vx_delay_attribute_e

The delay attribute list.

Enumerator

VX_DELAY_TYPE The type of reference contained in the delay. Read-only. Use a `vx_enum` parameter.

VX_DELAY_SLOTS The number of items in the delay. Read-only. Use a `vx_size` parameter.

Definition at line 1303 of file vx_types.h.

3.67.4 Function Documentation

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

Queries a `vx_delay` object attribute.

Parameters

in	*delay*	A pointer to a delay object.
in	*attribute*	The attribute to query. Use a `vx_delay_attribute_e` enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

vx_status VX_API_CALL vxReleaseDelay (vx_delay ∗ *delay*)

Releases a reference to a delay object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*delay*	The pointer to the delay to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If delay is not a `vx_delay`.

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

Creates a Delay object.

This function uses a subset of the attributes defining the metadata of the exemplar, ignoring the object. It does not alter the exemplar or keep or release the reference to the exemplar. For the definition of supported attributes see vxSetMetaFormatAttribute.

Parameters

`in`	*context*	The reference to the system context.
`in`	*exemplar*	The exemplar object.Supported delay object types include: • VX_TYPE_ARRAY • VX_TYPE_IMAGE • VX_TYPE_PYRAMID • VX_TYPE_MATRIX • VX_TYPE_CONVOLUTION • VX_TYPE_DISTRIBUTION • VX_TYPE_REMAP • VX_TYPE_LUT • VX_TYPE_THRESHOLD • VX_TYPE_SCALAR

in	*slots*	The number of reference in the delay.

Returns

A delay reference `vx_delay`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

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

Retrieves a reference from a delay object.

Parameters

in	*delay*	The reference to the delay object.
in	*index*	An index into the delay from which to extract the reference.

Returns

`vx_reference`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

Note

The delay index is in the range $[-count+1,0]$. 0 is always the *current* object.
A reference from a delay object must not be given to its associated release API (e.g. `vxReleaseImage`) unless `vxRetainReference` is used.

vx_status VX_API_CALL vxAgeDelay (vx_delay *delay*)

Ages the internal delay ring by one. This means that once this API is called the reference from index 0 will go to index -1 and so forth until $-count+1$ is reached. This last object will become 0. Once the delay has been aged, it updates the reference in any associated nodes. Here $count$ is the number of slots in delay ring.

Parameters

in	*delay*	

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Delay was aged.
VX_ERROR_INVALID_REFERENCE	The value passed as delay was not a `vx_delay`.

3.68 Object: Kernel

3.68.1 Detailed Description

Defines the Kernel Object and Interface. A Kernel in OpenVX is the abstract representation of an computer vision function, such as a "Sobel Gradient" or "Lucas Kanade Feature Tracking". A vision function may implement many similar or identical features from other functions, but it is still considered a single unique kernel as long as it is named by the same string and enumeration and conforms to the results specified by OpenVX. Kernels are similar to function signatures in this regard.

In each of the cases, a client of OpenVX could request the kernels in nearly the same manner. There are two main approaches, which depend on the method a client calls to get the kernel reference. The first uses enumerations.

```
  vx_kernel kernel = vxGetKernelByEnum(context,
VX_KERNEL_SOBEL_3x3);
  vx_node node = vxCreateGenericNode(graph, kernel);
```

The second method depends on using strings to get the kernel reference.

```
  vx_kernel kernel = vxGetKernelByName(context, "
org.khronos.openvx.sobel_3x3");
  vx_node node = vxCreateGenericNode(graph, kernel);
```

Data Structures

- struct vx_kernel_info_t

  The Kernel Information Structure. This is returned by the Context to indicate which kernels are available in the OpenVX implementation. More...

Macros

- #define VX_MAX_KERNEL_NAME (256)

  Defines the length of a kernel name string to be added to OpenVX, including the trailing zero.

Typedefs

- typedef struct _vx_kernel ∗ vx_kernel

  An opaque reference to the descriptor of a kernel.

Enumerations

- enum vx_kernel_attribute_e {
  VX_KERNEL_PARAMETERS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x0,
  VX_KERNEL_NAME = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x1,
  VX_KERNEL_ENUM = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x2,
  VX_KERNEL_LOCAL_DATA_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x3
  }

  The kernel attributes list.
- enum vx_kernel_e {
  VX_KERNEL_COLOR_CONVERT = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1,
  VX_KERNEL_CHANNEL_EXTRACT = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x2,
  VX_KERNEL_CHANNEL_COMBINE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x3,
  VX_KERNEL_SOBEL_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x4,
  VX_KERNEL_MAGNITUDE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x5,
  VX_KERNEL_PHASE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x6,
  VX_KERNEL_SCALE_IMAGE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x7,
  VX_KERNEL_TABLE_LOOKUP = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) +

0x8,
VX_KERNEL_HISTOGRAM = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x9,
VX_KERNEL_EQUALIZE_HISTOGRAM = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xA,
VX_KERNEL_ABSDIFF = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xB,
VX_KERNEL_MEAN_STDDEV = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xC,
VX_KERNEL_THRESHOLD = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xD,
VX_KERNEL_INTEGRAL_IMAGE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xE,
VX_KERNEL_DILATE_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xF,
VX_KERNEL_ERODE_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x10,
VX_KERNEL_MEDIAN_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x11,
VX_KERNEL_BOX_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x12,
VX_KERNEL_GAUSSIAN_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x13,
VX_KERNEL_CUSTOM_CONVOLUTION = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x14,
VX_KERNEL_GAUSSIAN_PYRAMID = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x15,
VX_KERNEL_ACCUMULATE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x16,
VX_KERNEL_ACCUMULATE_WEIGHTED = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x17,
VX_KERNEL_ACCUMULATE_SQUARE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x18,
VX_KERNEL_MINMAXLOC = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x19,
VX_KERNEL_CONVERTDEPTH = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1A,
VX_KERNEL_CANNY_EDGE_DETECTOR = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1B,
VX_KERNEL_AND = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1C,
VX_KERNEL_OR = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1D,
VX_KERNEL_XOR = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1E,
VX_KERNEL_NOT = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1F,
VX_KERNEL_MULTIPLY = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x20,
VX_KERNEL_ADD = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x21,
VX_KERNEL_SUBTRACT = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x22,
VX_KERNEL_WARP_AFFINE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x23,
VX_KERNEL_WARP_PERSPECTIVE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x24,
VX_KERNEL_HARRIS_CORNERS = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x25,
VX_KERNEL_FAST_CORNERS = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x26,
VX_KERNEL_OPTICAL_FLOW_PYR_LK = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x27,
VX_KERNEL_REMAP = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x28,
VX_KERNEL_HALFSCALE_GAUSSIAN = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x29,
VX_KERNEL_LAPLACIAN_PYRAMID = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x2A,
VX_KERNEL_LAPLACIAN_RECONSTRUCT = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x2B,
VX_KERNEL_NON_LINEAR_FILTER = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x2C,
VX_KERNEL_MAX_1_0 }

The standard list of available vision kernels.

- enum vx_library_e { VX_LIBRARY_KHR_BASE = 0x0 }

  The standard list of available libraries.

Functions

- vx_kernel VX_API_CALL vxGetKernelByEnum (vx_context context, vx_enum kernel)

  Obtains a reference to the kernel using the `vx_kernel_e` enumeration.
- vx_kernel VX_API_CALL vxGetKernelByName (vx_context context, const vx_char ∗name)

  Obtains a reference to a kernel using a string to specify the name.
- vx_status VX_API_CALL vxQueryKernel (vx_kernel kernel, vx_enum attribute, void ∗ptr, vx_size size)

  This allows the client to query the kernel to get information about the number of parameters, enum values, etc.
- vx_status VX_API_CALL vxReleaseKernel (vx_kernel ∗kernel)

  Release the reference to the kernel. The object may not be garbage collected until its total reference count is zero.

3.68.2 Data Structure Documentation

struct vx_kernel_info_t

The Kernel Information Structure. This is returned by the Context to indicate which kernels are available in the OpenVX implementation.

Definition at line 1433 of file vx_types.h.

Data Fields

vx_enum	enumeration	The kernel enumeration value from `vx_kernel_e` (or an extension thereof). See Also vxGetKernelByEnum
vx_char	name[VX_MAX_KERNEL_NAME]	The kernel name in dotted hierarchical format. e.g. "org.khronos.openvx.sobel_3x3". See Also vxGetKernelByName

3.68.3 Typedef Documentation

typedef struct _vx_kernel∗ vx_kernel

An opaque reference to the descriptor of a kernel.

See Also

vxGetKernelByName
vxGetKernelByEnum

Definition at line 198 of file vx_types.h.

3.68.4 Enumeration Type Documentation

enum vx_library_e

The standard list of available libraries.

Enumerator

VX_LIBRARY_KHR_BASE The base set of kernels as defined by Khronos.

Definition at line 45 of file vx_kernels.h.

enum vx_kernel_e

The standard list of available vision kernels.

Each kernel listed here can be used with the `vxGetKernelByEnum` call. When programming the parameters, use

- `VX_INPUT` for [in]
- `VX_OUTPUT` for [out]
- `VX_BIDIRECTIONAL` for [in,out]

When programming the parameters, use

- `VX_TYPE_IMAGE` for a `vx_image` in the size field of `vxGetParameterByIndex` or `vxSetParameterByIndex` *
- `VX_TYPE_ARRAY` for a `vx_array` in the size field of `vxGetParameterByIndex` or `vxSetParameterByIndex` *
- or other appropriate types in vx_type_e.

Enumerator

VX_KERNEL_COLOR_CONVERT The Color Space conversion kernel. The conversions are based on the `vx_df_image_e` code in the images.

See Also

Color Convert

VX_KERNEL_CHANNEL_EXTRACT The Generic Channel Extraction Kernel. This kernel can remove individual color channels from an interleaved or semi-planar, planar, sub-sampled planar image. A client could extract a red channel from an interleaved RGB image or do a Luma extract from a YUV format.

See Also

Channel Extract

VX_KERNEL_CHANNEL_COMBINE The Generic Channel Combine Kernel. This kernel combine multiple individual planes into a single multiplanar image of the type specified in the output image.

See Also

Channel Combine

VX_KERNEL_SOBEL_3x3 The Sobel 3x3 Filter Kernel.

See Also

Sobel 3x3

VX_KERNEL_MAGNITUDE The Magnitude Kernel. This kernel produces a magnitude plane from two input gradients.

See Also

Magnitude

VX_KERNEL_PHASE The Phase Kernel. This kernel produces a phase plane from two input gradients.

See Also

Phase

VX_KERNEL_SCALE_IMAGE The Scale Image Kernel. This kernel provides resizing of an input image to an output image. The scaling factor is determined but the relative sizes of the input and output.

See Also

Scale Image

VX_KERNEL_TABLE_LOOKUP The Table Lookup kernel.

See Also

TableLookup

VX_KERNEL_HISTOGRAM The Histogram Kernel.

See Also

Histogram

VX_KERNEL_EQUALIZE_HISTOGRAM The Histogram Equalization Kernel.

See Also

Equalize Histogram

VX_KERNEL_ABSDIFF The Absolute Difference Kernel.

See Also

Absolute Difference

VX_KERNEL_MEAN_STDDEV The Mean and Standard Deviation Kernel.

See Also

Mean and Standard Deviation

VX_KERNEL_THRESHOLD The Threshold Kernel.

See Also

Thresholding

VX_KERNEL_INTEGRAL_IMAGE The Integral Image Kernel.

See Also

Integral Image

VX_KERNEL_DILATE_3x3 The dilate kernel.

See Also

Dilate Image

VX_KERNEL_ERODE_3x3 The erode kernel.

See Also

Erode Image

VX_KERNEL_MEDIAN_3x3 The median image filter.

See Also

Median Filter

VX_KERNEL_BOX_3x3 The box filter kernel.

See Also

Box Filter

VX_KERNEL_GAUSSIAN_3x3 The gaussian filter kernel.

See Also

Gaussian Filter

VX_KERNEL_CUSTOM_CONVOLUTION The custom convolution kernel.

See Also

Custom Convolution

VX_KERNEL_GAUSSIAN_PYRAMID The gaussian image pyramid kernel.

See Also

Gaussian Image Pyramid

VX_KERNEL_ACCUMULATE The accumulation kernel.

See Also

Accumulate

VX_KERNEL_ACCUMULATE_WEIGHTED The weigthed accumulation kernel.

See Also

Accumulate Weighted

VX_KERNEL_ACCUMULATE_SQUARE The squared accumulation kernel.

See Also

Accumulate Squared

VX_KERNEL_MINMAXLOC The min and max location kernel.

See Also

Min, Max Location

VX_KERNEL_CONVERTDEPTH The bit-depth conversion kernel.

See Also

Convert Bit depth

VX_KERNEL_CANNY_EDGE_DETECTOR The Canny Edge Detector.

See Also

Canny Edge Detector

VX_KERNEL_AND The Bitwise And Kernel.

See Also

Bitwise AND

VX_KERNEL_OR The Bitwise Inclusive Or Kernel.

See Also

Bitwise INCLUSIVE OR

VX_KERNEL_XOR The Bitwise Exclusive Or Kernel.

See Also

Bitwise EXCLUSIVE OR

VX_KERNEL_NOT The Bitwise Not Kernel.

See Also

Bitwise NOT

VX_KERNEL_MULTIPLY The Pixelwise Multiplication Kernel.

See Also

Pixel-wise Multiplication

VX_KERNEL_ADD The Addition Kernel.

See Also

Arithmetic Addition

VX_KERNEL_SUBTRACT The Subtraction Kernel.

See Also

Arithmetic Subtraction

VX_KERNEL_WARP_AFFINE The Warp Affine Kernel.

See Also

Warp Affine

VX_KERNEL_WARP_PERSPECTIVE The Warp Perspective Kernel.

See Also

Warp Perspective

VX_KERNEL_HARRIS_CORNERS The Harris Corners Kernel.

See Also

Harris Corners

VX_KERNEL_FAST_CORNERS The FAST Corners Kernel.

See Also

Fast Corners

VX_KERNEL_OPTICAL_FLOW_PYR_LK The Optical Flow Pyramid (LK) Kernel.

See Also

Optical Flow Pyramid (LK)

VX_KERNEL_REMAP The Remap Kernel.

See Also

Remap

VX_KERNEL_HALFSCALE_GAUSSIAN The Half Scale Gaussian Kernel.

See Also

Scale Image

VX_KERNEL_LAPLACIAN_PYRAMID The Laplacian Image Pyramid Kernel.

See Also

Laplacian Image Pyramid

VX_KERNEL_LAPLACIAN_RECONSTRUCT The Laplacian Pyramid Reconstruct Kernel.

See Also

Laplacian Image Pyramid

VX_KERNEL_NON_LINEAR_FILTER The Non Linear Filter Kernel.

See Also

Non Linear Filter

Definition at line 63 of file vx_kernels.h.

enum vx_kernel_attribute_e

The kernel attributes list.

Enumerator

VX_KERNEL_PARAMETERS Queries a kernel for the number of parameters the kernel supports. Read-only. Use a `vx_uint32` parameter.

VX_KERNEL_NAME Queries the name of the kernel. Not settable. Read-only. Use a `vx_char[VX_MAX_KERNEL_NAME]` array (not a `vx_array`).

VX_KERNEL_ENUM Queries the enum of the kernel. Not settable. Read-only. Use a `vx_enum` parameter.

VX_KERNEL_LOCAL_DATA_SIZE The local data area allocated with each kernel when it becomes a node. Read-write. Can be written only before user-kernel finalization. Use a `vx_size` parameter.

Note

If not set it will default to zero.

Definition at line 830 of file vx_types.h.

3.68.5 Function Documentation

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

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

User Kernels follow a "dotted" heirarchical syntax. For example: "com.company.example.xyz". The following are strings specifying the kernel names:

org.khronos.openvx.color_convert
org.khronos.openvx.channel_extract
org.khronos.openvx.channel_combine
org.khronos.openvx.sobel_3x3
org.khronos.openvx.magnitude
org.khronos.openvx.phase
org.khronos.openvx.scale_image
org.khronos.openvx.table_lookup
org.khronos.openvx.histogram
org.khronos.openvx.equalize_histogram
org.khronos.openvx.absdiff
org.khronos.openvx.mean_stddev
org.khronos.openvx.threshold
org.khronos.openvx.integral_image
org.khronos.openvx.dilate_3x3
org.khronos.openvx.erode_3x3
org.khronos.openvx.median_3x3
org.khronos.openvx.box_3x3
org.khronos.openvx.gaussian_3x3
org.khronos.openvx.custom_convolution
org.khronos.openvx.gaussian_pyramid
org.khronos.openvx.accumulate
org.khronos.openvx.accumulate_weighted
org.khronos.openvx.accumulate_square
org.khronos.openvx.minmaxloc
org.khronos.openvx.convertdepth
org.khronos.openvx.canny_edge_detector
org.khronos.openvx.and
org.khronos.openvx.or
org.khronos.openvx.xor
org.khronos.openvx.not
org.khronos.openvx.multiply
org.khronos.openvx.add
org.khronos.openvx.subtract
org.khronos.openvx.warp_affine
org.khronos.openvx.warp_perspective
org.khronos.openvx.harris_corners
org.khronos.openvx.fast_corners
org.khronos.openvx.optical_flow_pyr_lk
org.khronos.openvx.remap
org.khronos.openvx.halfscale_gaussian
org.khronos.openvx.laplacian_pyramid
org.khronos.openvx.laplacian_reconstruct
org.khronos.openvx.non_linear_filter

Parameters

`in`	*context*	The reference to the implementation context.
`in`	*name*	The string of the name of the kernel to get.

Returns

A kernel reference or zero if an error occurred. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

Return values

0	The kernel name is not found in the context.

Precondition

`vxLoadKernels` if the kernel is not provided by the OpenVX implementation.

Note

User Kernels should follow a "dotted" heirarchical syntax. For example: "com.company.example.xyz".

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

Obtains a reference to the kernel using the `vx_kernel_e` enumeration.

Enum values above the standard set are assumed to apply to loaded libraries.

Parameters

`in`	*context*	The reference to the implementation context.
`in`	*kernel*	A value from `vx_kernel_e` or a vendor or client-defined value.

Returns

A `vx_kernel`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

Return values

0	The kernel enumeration is not found in the context.

Precondition

`vxLoadKernels` if the kernel is not provided by the OpenVX implementation.

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

This allows the client to query the kernel to get information about the number of parameters, enum values, etc.

Parameters

`in`	*kernel*	The kernel reference to query.
`in`	*attribute*	The attribute to query. Use a `vx_kernel_attribute_e`.
`out`	*ptr*	The pointer to the location at which to store the resulting value.
`in`	*size*	The size of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the kernel is not a `vx_kernel`.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

VX_ERROR_NOT_SUPPORTED	If the attribute value is not supported in this implementation.

vx_status VX_API_CALL vxReleaseKernel (vx_kernel * *kernel*)

Release the reference to the kernel. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*kernel*	The pointer to the kernel reference to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If kernel is not a vx_kernel.

3.69 Object: Parameter

3.69.1 Detailed Description

Defines the Parameter Object interface. An abstract input, output, or bidirectional data object passed to a computer vision function. This object contains the signature of that parameter's usage from the kernel description. This information includes:

- *Signature Index* - The numbered index of the parameter in the signature.
- *Object Type* - e.g., `VX_TYPE_IMAGE` or `VX_TYPE_ARRAY` or some other object type from `vx_type_e`.
- *Usage Model* - e.g., `VX_INPUT`, `VX_OUTPUT`, or `VX_BIDIRECTIONAL`.
- *Presence State* - e.g., `VX_PARAMETER_STATE_REQUIRED` or `VX_PARAMETER_STATE_OPTIONAL`.

Typedefs

- typedef struct _vx_parameter ∗ vx_parameter

  An opaque reference to a single parameter.

Enumerations

- enum vx_direction_e {
  VX_INPUT = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTION << 12)) + 0x0,
  VX_OUTPUT = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTION << 12)) + 0x1,
  VX_BIDIRECTIONAL = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTION << 12)) + 0x2 }

  An indication of how a kernel will treat the given parameter.
- enum vx_parameter_attribute_e {
  VX_PARAMETER_INDEX = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x0,
  VX_PARAMETER_DIRECTION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x1,
  VX_PARAMETER_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x2,
  VX_PARAMETER_STATE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x3,
  VX_PARAMETER_REF = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x4 }

  The parameter attributes list.
- enum vx_parameter_state_e {
  VX_PARAMETER_STATE_REQUIRED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_PARAMETER_STATE << 12)) + 0x0,
  VX_PARAMETER_STATE_OPTIONAL = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_PARAMETER_STATE << 12)) + 0x1 }

  The parameter state type.

Functions

- vx_parameter VX_API_CALL vxGetKernelParameterByIndex (vx_kernel kernel, vx_uint32 index)

  Retrieves a `vx_parameter` from a `vx_kernel`.
- vx_parameter VX_API_CALL vxGetParameterByIndex (vx_node node, vx_uint32 index)

  Retrieves a `vx_parameter` from a `vx_node`.
- vx_status VX_API_CALL vxQueryParameter (vx_parameter param, vx_enum attribute, void ∗ptr, vx_size size)

  Allows the client to query a parameter to determine its meta-information.
- vx_status VX_API_CALL vxReleaseParameter (vx_parameter ∗param)

  Releases a reference to a parameter object. The object may not be garbage collected until its total reference count is zero.
- vx_status VX_API_CALL vxSetParameterByIndex (vx_node node, vx_uint32 index, vx_reference value)

  Sets the specified parameter data for a kernel on the node.
- vx_status VX_API_CALL vxSetParameterByReference (vx_parameter parameter, vx_reference value)

  Associates a parameter reference and a data reference with a kernel on a node.

3.69.2 Typedef Documentation

typedef struct _vx_parameter* vx_parameter

An opaque reference to a single parameter.

See Also

vxGetParameterByIndex

Definition at line 205 of file vx_types.h.

3.69.3 Enumeration Type Documentation

enum vx_direction_e

An indication of how a kernel will treat the given parameter.

Enumerator

VX_INPUT The parameter is an input only.

VX_OUTPUT The parameter is an output only.

VX_BIDIRECTIONAL The parameter is both an input and output.

Definition at line 580 of file vx_types.h.

enum vx_parameter_attribute_e

The parameter attributes list.

Enumerator

VX_PARAMETER_INDEX Queries a parameter for its index value on the kernel with which it is associated. Read-only. Use a `vx_uint32` parameter.

VX_PARAMETER_DIRECTION Queries a parameter for its direction value on the kernel with which it is associated. Read-only. Use a `vx_enum` parameter.

VX_PARAMETER_TYPE Queries a parameter for its type, vx_type_e is returned. Read-only. The size of the parameter is implied for plain data objects. For opaque data objects like images and arrays a query to their attributes has to be called to determine the size.

VX_PARAMETER_STATE Queries a parameter for its state. A value in `vx_parameter_state_e` is returned. Read-only. Use a `vx_enum` parameter.

VX_PARAMETER_REF Use to extract the reference contained in the parameter. Read-only. Use a `vx_reference` parameter.

Definition at line 898 of file vx_types.h.

enum vx_parameter_state_e

The parameter state type.

Enumerator

VX_PARAMETER_STATE_REQUIRED Default. The parameter must be supplied. If not set, during Verify, an error is returned.

VX_PARAMETER_STATE_OPTIONAL The parameter may be unspecified. The kernel takes care not to deference optional parameters until it is certain they are valid.

Definition at line 1237 of file vx_types.h.

3.69.4 Function Documentation

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

Retrieves a `vx_parameter` from a `vx_kernel`.

Parameters

in	*kernel*	The reference to the kernel.
in	*index*	The index of the parameter.

Returns

A `vx_parameter`.Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

Return values

0	Either the kernel or index is invalid.
*	The parameter reference.

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

Retrieves a `vx_parameter` from a `vx_node`.

Parameters

in	*node*	The node from which to extract the parameter.
in	*index*	The index of the parameter to which to get a reference.

Returns

`vx_parameter`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

vx_status VX_API_CALL vxReleaseParameter (vx_parameter * *param*)

Releases a reference to a parameter object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*param*	The pointer to the parameter to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If param is not a `vx_parameter`.

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

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

Parameters

in	*node*	The node that contains the kernel.

in	*index*	The index of the parameter desired.
in	*value*	The desired value of the parameter.

Note

A user may not provide a NULL value for a mandatory parameter of this API.

Returns

A `vx_status_e` enumeration.

See Also

vxSetParameterByReference

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

Associates a parameter reference and a data reference with a kernel on a node.

Parameters

in	*parameter*	The reference to the kernel parameter.
in	*value*	The value to associate with the kernel parameter.

Note

A user may not provide a NULL value for a mandatory parameter of this API.

Returns

A `vx_status_e` enumeration.

See Also

vxGetParameterByIndex

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

Allows the client to query a parameter to determine its meta-information.

Parameters

in	*param*	The reference to the parameter.
in	*attribute*	The attribute to query. Use a `vx_parameter_attribute_e`.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size in bytes of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

3.70 Advanced Framework API

3.70.1 Detailed Description

Describes components that are considered to be advanced. Advanced topics include: extensions through User Kernels; Reflection and Introspection; Performance Tweaking through Hinting and Directives; and Debugging Callbacks.

Modules

- Framework: Node Callbacks

  Allows Clients to receive a callback after a specific node has completed execution.
- Framework: Performance Measurement

  Defines Performance measurement and reporting interfaces.
- Framework: Log

  Defines the debug logging interface.
- Framework: Hints

  Defines the Hints Interface.
- Framework: Directives

  Defines the Directives Interface.
- Framework: User Kernels

  Defines the User Kernels, which are a method to extend OpenVX with new vision functions.
- Framework: Graph Parameters

  Defines the Graph Parameter API.

3.71 Framework: Node Callbacks

3.71.1 Detailed Description

Allows Clients to receive a callback after a specific node has completed execution. Callbacks are not guaranteed to be called *immediately* after the Node completes. Callbacks are intended to be used to create simple *early exit* conditions for Vision graphs using `vx_action_e` return values. An example of setting up a callback can be seen below:

```
vx_graph graph = vxCreateGraph(context);
status = vxGetStatus((vx_reference)graph);
if (status == VX_SUCCESS) {
    vx_uint8 lmin = 0, lmax = 0;
    vx_uint32 minCount = 0, maxCount = 0;
    vx_scalar scalars[] = {
        vxCreateScalar(context, VX_TYPE_UINT8, &lmin),
        vxCreateScalar(context, VX_TYPE_UINT8, &lmax),
        vxCreateScalar(context, VX_TYPE_UINT32, &minCount),
        vxCreateScalar(context, VX_TYPE_UINT32, &maxCount),
    };
    vx_array arrays[] = {
        vxCreateArray(context, VX_TYPE_COORDINATES2D, 1),
        vxCreateArray(context, VX_TYPE_COORDINATES2D, 1)
    };
    vx_node nodes[] = {
        vxMinMaxLocNode(graph, input, scalars[0], scalars[1], arrays[0], arrays[1],
  scalars[2], scalars[3]),
    };
    status = vxAssignNodeCallback(nodes[0], &analyze_brightness);
    // do other
}
```

Once the graph has been initialized and the callback has been installed then the callback itself will be called during graph execution.

```
#define MY_DESIRED_THRESHOLD (10)
vx_action VX_CALLBACK analyze_brightness(vx_node node) {
    // extract the max value
    vx_action action = VX_ACTION_ABANDON;
    vx_parameter pmax = vxGetParameterByIndex(node, 2); // Max Value
    if (pmax) {
        vx_scalar smax = 0;
        vxQueryParameter(pmax, VX_PARAMETER_REF, &smax, sizeof(smax));
        if (smax) {
            vx_uint8 value = 0u;
            vxCopyScalar(smax, &value, VX_READ_ONLY,
  VX_MEMORY_TYPE_HOST);
            if (value >= MY_DESIRED_THRESHOLD) {
                action = VX_ACTION_CONTINUE;
            }
            vxReleaseScalar(&smax);
        }
        vxReleaseParameter(&pmax);
    }
    return action;
}
```

Warning

> This should be used with ***extreme*** caution as it can *ruin* optimizations in the power/performance efficiency of a graph.

The callback must return a `vx_action` code indicating how the graph processing should proceed.

- If `VX_ACTION_CONTINUE` is returned, the graph will continue execution with no changes.
- If `VX_ACTION_ABANDON` is returned, execution is unspecified for all nodes for which this node is a dominator. Nodes that are dominators of this node will have executed. Execution of any other node is unspecified.

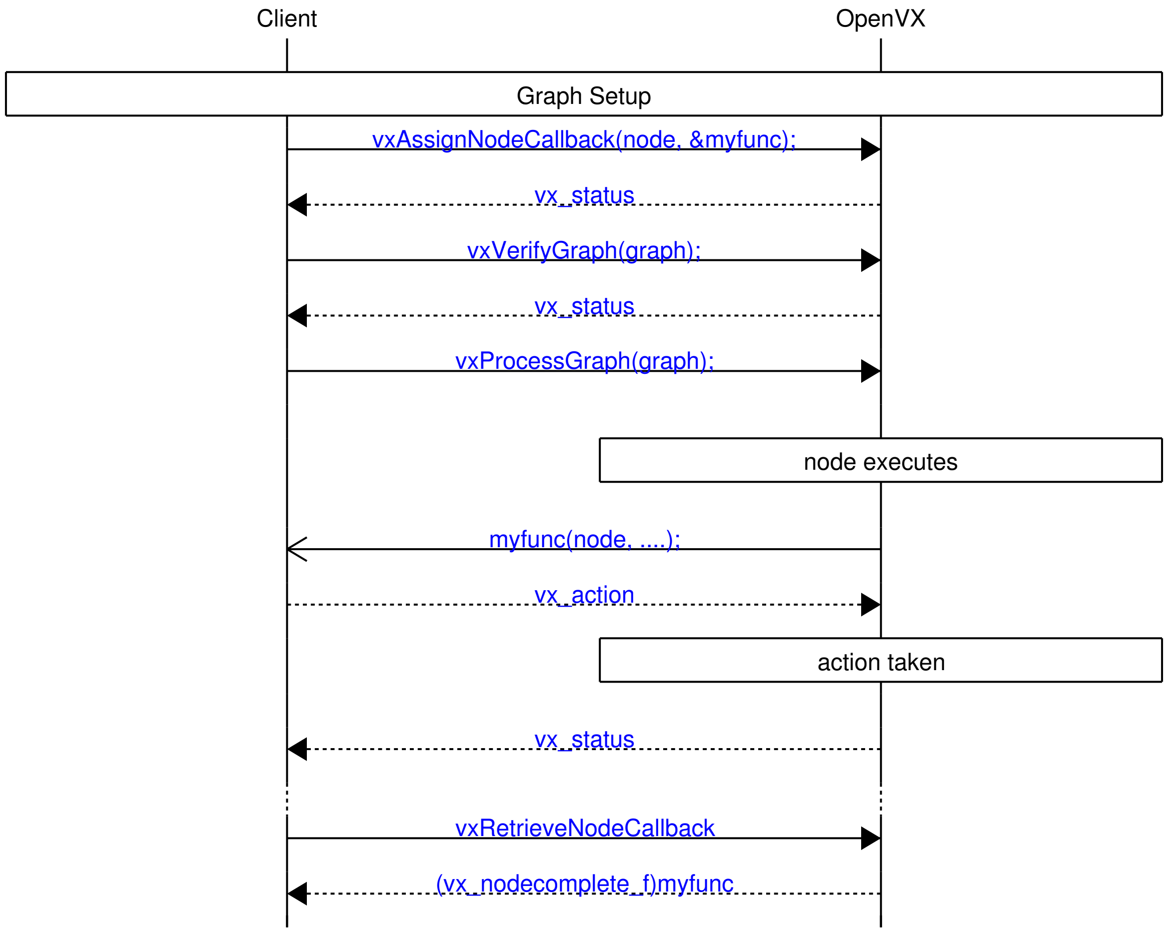


Figure 3.2: Node Callback Sequence

Typedefs

- typedef vx_enum vx_action

  The formal typedef of the response from the callback.
- typedef vx_action(∗ vx_nodecomplete_f)(vx_node node)

  A callback to the client after a particular node has completed.

Enumerations

- enum vx_action_e {
  VX_ACTION_CONTINUE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACTION << 12)) + 0x0,
  VX_ACTION_ABANDON = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACTION << 12)) + 0x1 }

  A return code enumeration from a `vx_nodecomplete_f` during execution.

Functions

- vx_status VX_API_CALL vxAssignNodeCallback (vx_node node, vx_nodecomplete_f callback)

  Assigns a callback to a node. If a callback already exists in this node, this function must return an error and the user may clear the callback by passing a NULL pointer as the callback.
- vx_nodecomplete_f VX_API_CALL vxRetrieveNodeCallback (vx_node node)

  Retrieves the current node callback function pointer set on the node.

3.71.2 Typedef Documentation

typedef vx_enum vx_action

The formal typedef of the response from the callback.

See Also

vx_action_e

Definition at line 434 of file vx_types.h.

typedef vx_action(* vx_nodecomplete_f)(vx_node node)

A callback to the client after a particular node has completed.

See Also

vx_action
vxAssignNodeCallback

Parameters

in	*node*	The node to which the callback was attached.

Returns

An action code from `vx_action_e`.

Definition at line 443 of file vx_types.h.

3.71.3 Enumeration Type Documentation

enum vx_action_e

A return code enumeration from a `vx_nodecomplete_f` during execution.

See Also

`vxAssignNodeCallback`

Enumerator

VX_ACTION_CONTINUE Continue executing the graph with no changes.

VX_ACTION_ABANDON Stop executing the graph.

Definition at line 570 of file vx_types.h.

3.71.4 Function Documentation

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

Assigns a callback to a node. If a callback already exists in this node, this function must return an error and the user may clear the callback by passing a NULL pointer as the callback.

Parameters

in	*node*	The reference to the node.
in	*callback*	The callback to associate with completion of this specific node.

Warning

This must be used with ***extreme*** caution as it can *ruin* optimizations in the power/performance efficiency of a graph.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Callback assigned.
VX_ERROR_INVALID_REFERENCE	The value passed as node was not a vx_node.

vx_nodecomplete_f VX_API_CALL vxRetrieveNodeCallback (vx_node *node*)

Retrieves the current node callback function pointer set on the node.

Parameters

in	*node*	The reference to the vx_node object.

Returns

vx_nodecomplete_f The pointer to the callback function.

Return values

NULL	No callback is set.
*	The node callback function.

3.72 Framework: Performance Measurement

3.72.1 Detailed Description

Defines Performance measurement and reporting interfaces. In OpenVX, both `vx_graph` objects and `vx_node` objects track performance information. A client can query either object type using their respective `vxQuery<Object>` function with their attribute enumeration `VX_<OBJECT>_PERFORMANCE` along with a `vx_perf_t` structure to obtain the performance information.

```
vx_perf_t perf;
vxQueryNode(node, VX_NODE_PERFORMANCE, &perf, sizeof(perf));
```

Data Structures

- struct vx_perf_t

  The performance measurement structure. The time or durations are in units of nano seconds. More...

3.72.2 Data Structure Documentation

struct vx_perf_t

The performance measurement structure. The time or durations are in units of nano seconds.

Definition at line 1413 of file vx_types.h.

Data Fields

vx_uint64	tmp	Holds the last measurement.
vx_uint64	beg	Holds the first measurement in a set.
vx_uint64	end	Holds the last measurement in a set.
vx_uint64	sum	Holds the summation of durations.
vx_uint64	avg	Holds the average of the durations.
vx_uint64	min	Holds the minimum of the durations.
vx_uint64	num	Holds the number of measurements.
vx_uint64	max	Holds the maximum of the durations.

3.73 Framework: Log

3.73.1 Detailed Description

Defines the debug logging interface. The functions of the debugging interface allow clients to receive important debugging information about OpenVX.

See Also

vx_status_e for the list of possible errors.

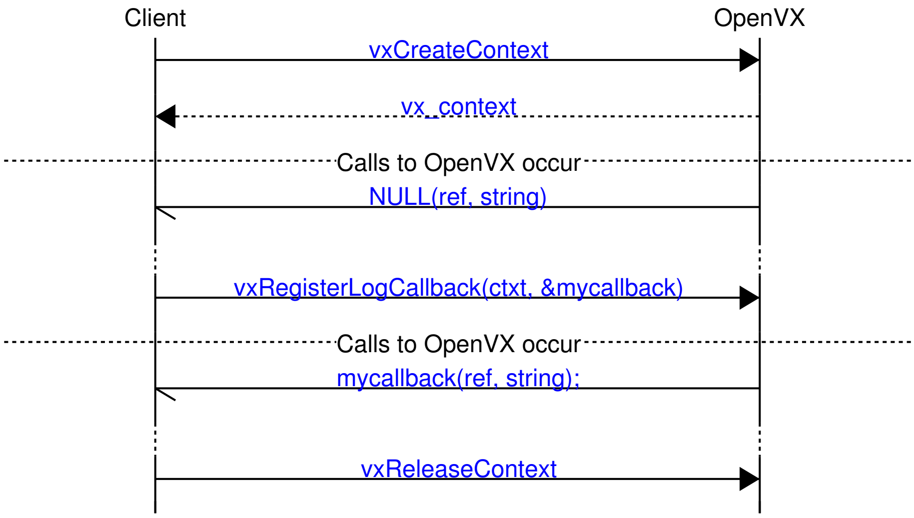


Figure 3.3: Log messages only can be received after the callback is installed.

Typedefs

- typedef void(∗ vx_log_callback_f)(vx_context context, vx_reference ref, vx_status status, const vx_char string[])

  The log callback function.

Functions

- void VX_API_CALL vxAddLogEntry (vx_reference ref, vx_status status, const char ∗message,...)

  Adds a line to the log.
- void VX_API_CALL vxRegisterLogCallback (vx_context context, vx_log_callback_f callback, vx_bool reentrant)

  Registers a callback facility to the OpenVX implementation to receive error logs.

3.73.2 Function Documentation

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

Adds a line to the log.

Parameters

in	*ref*	The reference to add the log entry against. Some valid value must be provided.

in	*status*	The status code. VX_SUCCESS status entries are ignored and not added.
in	*message*	The human readable message to add to the log.
in	*...*	a list of variable arguments to the message.

Note

Messages may not exceed VX_MAX_LOG_MESSAGE_LEN bytes and will be truncated in the log if they exceed this limit.

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

Registers a callback facility to the OpenVX implementation to receive error logs.

Parameters

in	*context*	The overall context to OpenVX.
in	*callback*	The callback function. If NULL, the previous callback is removed.
in	*reentrant*	If reentrancy flag is vx_true_e, then the callback may be entered from multiple simultaneous tasks or threads (if the host OS supports this).

3.74 Framework: Hints

3.74.1 Detailed Description

Defines the Hints Interface. *Hints* are messages given to the OpenVX implementation that it may support. (These are optional.)

Enumerations

- enum vx_hint_e {
  VX_HINT_PERFORMANCE_DEFAULT = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_HINT << 12)) + 0x1,
  VX_HINT_PERFORMANCE_LOW_POWER = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_HINT << 12)) + 0x2,
  VX_HINT_PERFORMANCE_HIGH_SPEED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_HINT << 12)) + 0x3 }

  These enumerations are given to the `vxHint` API to enable/disable platform optimizations and/or features. Hints are optional and usually are vendor-specific.

Functions

- vx_status VX_API_CALL vxHint (vx_reference reference, vx_enum hint, const void ∗data, vx_size data_size)

  Provides a generic API to give platform-specific hints to the implementation.

3.74.2 Enumeration Type Documentation

enum vx_hint_e

These enumerations are given to the `vxHint` API to enable/disable platform optimizations and/or features. Hints are optional and usually are vendor-specific.

See Also

`vxHint`

Enumerator

VX_HINT_PERFORMANCE_DEFAULT Indicates to the implementation that user do not apply any specific requirements for performance.

VX_HINT_PERFORMANCE_LOW_POWER Indicates the user preference is low power consumption versus highest performance.

VX_HINT_PERFORMANCE_HIGH_SPEED Indicates the user preference for highest performance over low power consumption.

Definition at line 594 of file vx_types.h.

3.74.3 Function Documentation

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

Provides a generic API to give platform-specific hints to the implementation.

Parameters

in	*reference*	The reference to the object to hint at. This could be `vx_context`, `vx_graph`, `vx_node`, `vx_image`, `vx_array`, or any other reference.

in	*hint*	A `vx_hint_e` *hint* to give to a vx_context. This is a platform-specific optimization or implementation mechanism.
in	*data*	Optional vendor specific data.
in	*data_size*	Size of the data structure `data`.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No error.
VX_ERROR_INVALID_REFERENCE	If context or reference is invalid.
VX_ERROR_NOT_SUPPORTED	If the hint is not supported.

3.75 Framework: Directives

3.75.1 Detailed Description

Defines the Directives Interface. *Directives* are messages given the OpenVX implementation that it must support. (These are required, i.e., non-optional.)

Enumerations

- enum vx_directive_e {
  VX_DIRECTIVE_DISABLE_LOGGING = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTIVE << 12)) + 0x0,
  VX_DIRECTIVE_ENABLE_LOGGING = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTIVE << 12)) + 0x1,
  VX_DIRECTIVE_DISABLE_PERFORMANCE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTIVE << 12)) + 0x2,
  VX_DIRECTIVE_ENABLE_PERFORMANCE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTIVE << 12)) + 0x3 }

  These enumerations are given to the `vxDirective` API to enable/disable platform optimizations and/or features. Directives are not optional and usually are vendor-specific, by defining a vendor range of directives and starting their enumeration from there.

Functions

- vx_status VX_API_CALL vxDirective (vx_reference reference, vx_enum directive)

  Provides a generic API to give platform-specific directives to the implementations.

3.75.2 Enumeration Type Documentation

enum vx_directive_e

These enumerations are given to the `vxDirective` API to enable/disable platform optimizations and/or features. Directives are not optional and usually are vendor-specific, by defining a vendor range of directives and starting their enumeration from there.

See Also

`vxDirective`

Enumerator

VX_DIRECTIVE_DISABLE_LOGGING Disables recording information for graph debugging.

VX_DIRECTIVE_ENABLE_LOGGING Enables recording information for graph debugging.

VX_DIRECTIVE_DISABLE_PERFORMANCE Disables performance counters for the context. By default performance counters are disabled.

VX_DIRECTIVE_ENABLE_PERFORMANCE Enables performance counters for the context.

Definition at line 616 of file vx_types.h.

3.75.3 Function Documentation

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

Provides a generic API to give platform-specific directives to the implementations.

Parameters

in	*reference*	The reference to the object to set the directive on. This could be vx_-context, vx_graph, vx_node, vx_image, vx_array, or any other reference.
in	*directive*	The directive to set. See vx_directive_e.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No error.
VX_ERROR_INVALID_RE-FERENCE	If context or reference is invalid.
VX_ERROR_NOT_SUPPO-RTED	If the directive is not supported.

Note

The performance counter directives are only available for the reference vx_context. Error VX_ERROR_NOT-_SUPPORTED is returned when used with any other reference.

3.76 Framework: User Kernels

3.76.1 Detailed Description

Defines the User Kernels, which are a method to extend OpenVX with new vision functions. User Kernels can be loaded by OpenVX and included as nodes in the graph or as immediate functions (if the Client supplies the interface). User Kernels will typically be loaded and executed on High Level Operating System/CPU compatible targets, not on remote processors or other accelerators. This specification does not mandate what constitutes compatible platforms.

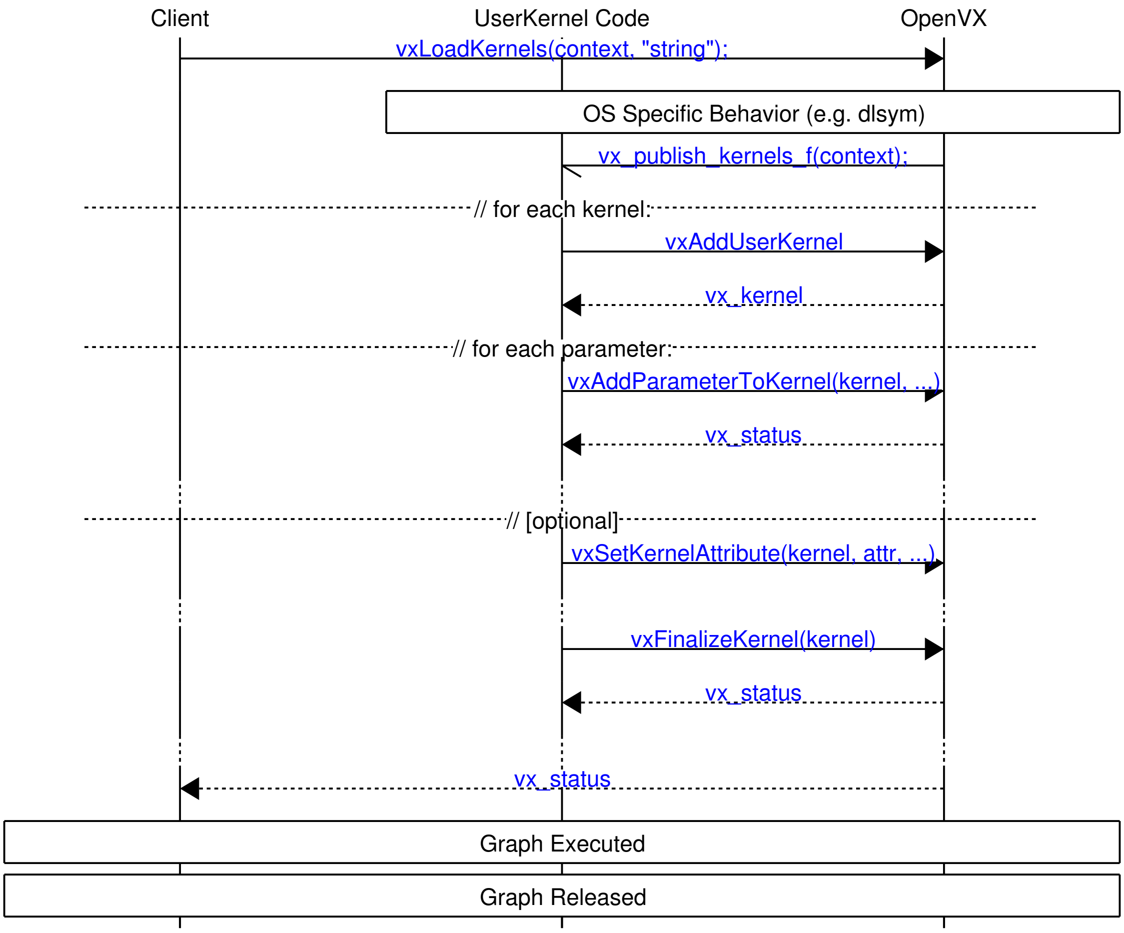


Figure 3.4: Call sequence of User Kernels Installation

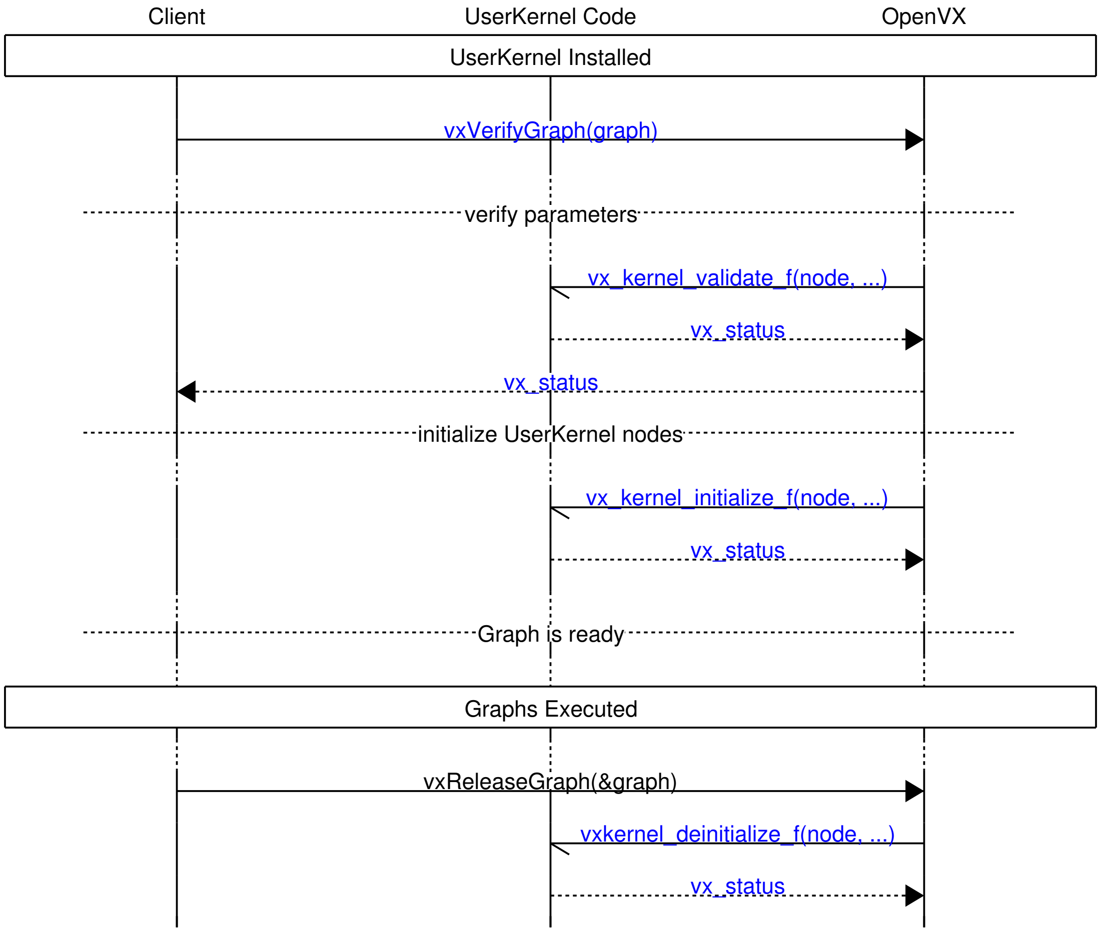


Figure 3.5: Call sequence of a Graph Verify and Release with User Kernels.

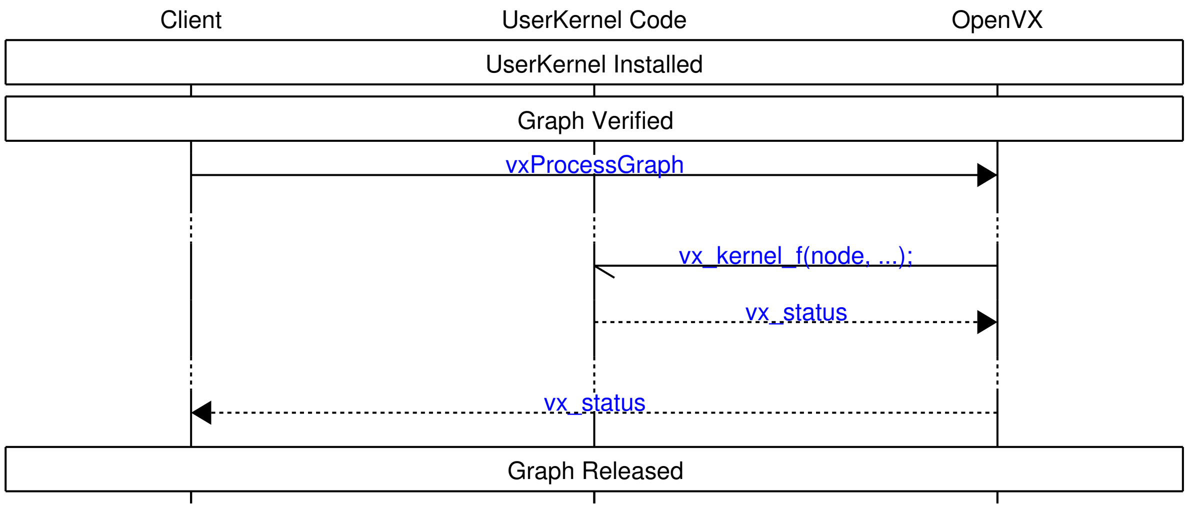


Figure 3.6: Call sequence of a Graph Execution with User Kernels

During the first graph verification, the implementation will perform the following action sequence:

1. Initialize local data node attributes
   - If VX_KERNEL_LOCAL_DATA_SIZE == 0, then set VX_NODE_LOCAL_DATA_SIZE to 0 and set VX-_NODE_LOCAL_DATA_PTR to NULL.

  - If VX_KERNEL_LOCAL_DATA_SIZE != 0, set VX_NODE_LOCAL_DATA_SIZE to VX_KERNEL_LOCAL_DATA_SIZE and set VX_NODE_LOCAL_DATA_PTR to the address of a buffer of VX_KERNEL_LOCAL_DATA_SIZE bytes.

2. Call the vx_kernel_validate_f callback.
3. Call the vx_kernel_initialize_f callback (if not NULL):
   - If VX_KERNEL_LOCAL_DATA_SIZE == 0, the callback is allowed to set VX_NODE_LOCAL_DATA_SIZE and VX_NODE_LOCAL_DATA_PTR.
   - If VX_KERNEL_LOCAL_DATA_SIZE != 0, then any attempt by the callback to set VX_NODE_LOCAL_DATA_SIZE or VX_NODE_LOCAL_DATA_PTR attributes will generate an error.
4. Provide the buffer optionally requested by the application
   - If VX_KERNEL_LOCAL_DATA_SIZE == 0 and VX_NODE_LOCAL_DATA_SIZE != 0, and VX_NODE_LOCAL_DATA_PTR == NULL, then the implementation will set VX_NODE_LOCAL_DATA_PTR to the address of a buffer of VX_NODE_LOCAL_DATA_SIZE bytes.

At node destruction time, the implementation will perform the following action sequence:

1. Call vx_kernel_deinitialize_f callback (if not NULL): If the VX_NODE_LOCAL_DATA_PTR was set earlier by the implementation, then any attempt by the callback to set the VX_NODE_LOCAL_DATA_PTR attributes will generate an error.
2. If the VX_NODE_LOCAL_DATA_PTR was set earlier by the implementation, then the pointed memory must not be used anymore by the application after the vx_kernel_deinitialize_f callback completes.

A user node requires re-verification, if any changes below occurred after the last node verification:

1. The VX_NODE_BORDER node attribute was modified.
2. At least one of the node parameters was replaced by a data object with different meta-data, or was replaced by the 0 reference for optional parameters, or was set to a data object if previously not set because optional.

The node re-verification can by triggered explicitly by the application by calling vxVerifyGraph that will perform a complete graph verification. Otherwise, it will be triggered automatically at the next graph execution.

During user node re-verification, the following action sequence will occur:

1. Call the vx_kernel_deinitialize_f callback (if not NULL): If the VX_NODE_LOCAL_DATA_PTR was set earlier by the OpenVX implementation, then any attempt by the callback to set the VX_NODE_LOCAL_DATA_PTR attributes will generate an error.
2. Reinitialize local data node attributes if needed If VX_KERNEL_LOCAL_DATA_SIZE == 0:
   - set VX_NODE_LOCAL_DATA_PTR to NULL.
   - set VX_NODE_LOCAL_DATA_SIZE to 0.
3. Call the vx_kernel_validate_f callback.
4. Call the vx_kernel_initialize_f callback (if not NULL):
   - If VX_KERNEL_LOCAL_DATA_SIZE == 0, the callback is allowed to set VX_NODE_LOCAL_DATA_SIZE and VX_NODE_LOCAL_DATA_PTR.
   - If VX_KERNEL_LOCAL_DATA_SIZE is != 0, then any attempt by the callback to set VX_NODE_LOCAL_DATA_SIZE or VX_NODE_LOCAL_DATA_PTR attributes will generate an error.
5. Provide the buffer optionally requested by the application
   - If VX_KERNEL_LOCAL_DATA_SIZE == 0 and VX_NODE_LOCAL_DATA_SIZE != 0, and VX_NODE_LOCAL_DATA_PTR == NULL, then the OpenVX implementation will set VX_NODE_LOCAL_DATA_PTR to the address of a buffer of VX_NODE_LOCAL_DATA_SIZE bytes.

When an OpenVX implementation sets the VX_NODE_LOCAL_DATA_PTR, the data inside the buffer will not be persistent between kernel executions.

Typedefs

- typedef vx_status(∗ vx_kernel_deinitialize_f)(vx_node node, const vx_reference ∗parameters, vx_uint32 num)

    The pointer to the kernel deinitializer. If the host code requires a call to deinitialize data during a node garbage collection, this function is called if not NULL.
- typedef vx_status(∗ vx_kernel_f)(vx_node node, const vx_reference ∗parameters, vx_uint32 num)

    The pointer to the Host side kernel.
- typedef vx_status(∗ vx_kernel_image_valid_rectangle_f)(vx_node node, vx_uint32 index, const vx_rectangle_t ∗const input_valid[ ], vx_rectangle_t ∗const output_valid[ ])

    A user-defined callback function to set the valid rectangle of an output image.
- typedef vx_status(∗ vx_kernel_initialize_f)(vx_node node, const vx_reference ∗parameters, vx_uint32 num)

    The pointer to the kernel initializer. If the host code requires a call to initialize data once all the parameters have been validated, this function is called if not NULL.
- typedef vx_status(∗ vx_kernel_validate_f)(vx_node node, const vx_reference parameters[ ], vx_uint32 num, vx_meta_format metas[ ])

    The user-defined kernel node parameters validation function. The function only needs to fill in the meta data structure(s).
- typedef struct _vx_meta_format ∗ vx_meta_format

    This object is used by output validation functions to specify the meta data of the expected output data object.
- typedef vx_status(∗ vx_publish_kernels_f)(vx_context context)

    The type of the `vxPublishKernels` entry function of modules loaded by `vxLoadKernels` and unloaded by `vxUnloadKernels`.

Enumerations

- enum vx_meta_valid_rect_attribute_e { VX_VALID_RECT_CALLBACK = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_META_FORMAT << 8)) + 0x1 }

    The meta valid rectangle attributes.

Functions

- vx_status VX_API_CALL vxAddParameterToKernel (vx_kernel kernel, vx_uint32 index, vx_enum dir, vx_enum data_type, vx_enum state)

    Allows users to set the signatures of the custom kernel.
- vx_kernel VX_API_CALL 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)

    Allows users to add custom kernels to a context at run-time.
- vx_status VX_API_CALL vxAllocateUserKernelId (vx_context context, vx_enum ∗pKernelEnumId)

    Allocates and registers user-defined kernel enumeration to a context. The allocated enumeration is from available pool of 4096 enumerations reserved for dynamic allocation from VX_KERNEL_BASE(VX_ID_USER,0).
- vx_status VX_API_CALL vxAllocateUserKernelLibraryId (vx_context context, vx_enum ∗pLibraryId)

    Allocates and registers user-defined kernel library ID to a context.
- vx_status VX_API_CALL vxFinalizeKernel (vx_kernel kernel)

    This API is called after all parameters have been added to the kernel and the kernel is ready to be used. Notice that the reference to the kernel created by vxAddUserKernel is still valid after the call to vxFinalizeKernel.
- vx_status VX_API_CALL vxLoadKernels (vx_context context, const vx_char ∗module)

    Loads a library of kernels, called module, into a context.
- vx_status VX_API_CALL vxRemoveKernel (vx_kernel kernel)

    Removes a custom kernel from its context and releases it.
- vx_status VX_API_CALL vxSetKernelAttribute (vx_kernel kernel, vx_enum attribute, const void ∗ptr, vx_size size)

    Sets kernel attributes.

- vx_status VX_API_CALL vxSetMetaFormatAttribute (vx_meta_format meta, vx_enum attribute, const void ∗ptr, vx_size size)

    This function allows a user to set the attributes of a `vx_meta_format` object in a kernel output validator.
- vx_status VX_API_CALL vxSetMetaFormatFromReference (vx_meta_format meta, vx_reference exemplar)

    Set a meta format object from an exemplar data object reference.
- vx_status VX_API_CALL vxUnloadKernels (vx_context context, const vx_char ∗module)

    Unloads all kernels from the OpenVX context that had been loaded from the module using the vxLoadKernels function.

3.76.2 Typedef Documentation

typedef struct _vx_meta_format∗ vx_meta_format

This object is used by output validation functions to specify the meta data of the expected output data object.

Note

When the actual output object of the user node is virtual, the information given through the vx_meta_format object allows the OpenVX framework to automatically create the data object when meta data were not specified by the application at object creation time.

Definition at line 317 of file vx_types.h.

typedef vx_status(∗ vx_publish_kernels_f)(vx_context context)

The type of the `vxPublishKernels` entry function of modules loaded by `vxLoadKernels` and unloaded by `vxUnloadKernels`.

Parameters

`in`	*context*	The reference to the context kernels must be added to.

Definition at line 1535 of file vx_types.h.

typedef vx_status(∗ vx_kernel_f)(vx_node node, const vx_reference ∗parameters, vx_uint32 num)

The pointer to the Host side kernel.

Parameters

`in`	*node*	The handle to the node that contains this kernel.
`in`	*parameters*	The array of parameter references.
`in`	*num*	The number of parameters.

Definition at line 1551 of file vx_types.h.

typedef vx_status(∗ vx_kernel_initialize_f)(vx_node node, const vx_reference ∗parameters, vx_uint32 num)

The pointer to the kernel initializer. If the host code requires a call to initialize data once all the parameters have been validated, this function is called if not NULL.

Parameters

`in`	*node*	The handle to the node that contains this kernel.
`in`	*parameters*	The array of parameter references.
`in`	*num*	The number of parameters.

Definition at line 1562 of file vx_types.h.

typedef vx_status(∗ vx_kernel_deinitialize_f)(vx_node node, const vx_reference ∗parameters, vx_uint32 num)

The pointer to the kernel deinitializer. If the host code requires a call to deinitialize data during a node garbage collection, this function is called if not NULL.

Parameters

in	*node*	The handle to the node that contains this kernel.
in	*parameters*	The array of parameter references.
in	*num*	The number of parameters.

Definition at line 1573 of file vx_types.h.

typedef vx_status(∗ vx_kernel_validate_f)(vx_node node, const vx_reference parameters[ ], vx_uint32 num, vx_meta_format metas[ ])

The user-defined kernel node parameters validation function. The function only needs to fill in the meta data structure(s).

Note

This function is called once for whole set of parameters.

Parameters

in	*node*	The handle to the node that is being validated.
in	*parameters*	The array of parameters to be validated.
in	*num*	Number of parameters to be validated.
in	*metas*	A pointer to a pre-allocated array of structure references that the system holds. The system pre-allocates a number of vx_meta_format structures for the output parameters only, indexed by the same indices as parameters[]. The validation function fills in the correct type, format, and dimensionality for the system to use either to create memory or to check against existing memory.

Returns

An error code describing the validation status on parameters.

Definition at line 1589 of file vx_types.h.

typedef vx_status(∗ vx_kernel_image_valid_rectangle_f)(vx_node node, vx_uint32 index, const vx_rectangle_t ∗const input_valid[ ], vx_rectangle_t ∗const output_valid[ ])

A user-defined callback function to set the valid rectangle of an output image.

The `VX_VALID_RECT_CALLBACK` attribute in the `vx_meta_format` object should be set to the desired callback during user node's output validator. The callback must not call `vxGetValidRegionImage` or `vxSetImageValidRectangle`. Instead, an array of the valid rectangles of all the input images is supplied to the callback to calculate the output valid rectangle. The output of the user node may be a pyramid, or just an image. If it is just an image, the 'Out' array associated with that output only has one element. If the output is a pyramid, the array size is equal to the number of pyramid levels. Notice that the array memory allocation passed to the callback is managed by the framework, the application must not allocate or deallocate those pointers.

The behavior of the callback function vx_kernel_image_valid_rectangle_f is undefined if one of the following is true:

- One of the input arguments of a user node is a pyramid or an array of images.
- Either input or output argument of a user node is an array of pyramids.

Parameters

in,out	*node*	The handle to the node that is being validated.
in	*index*	The index of the output parameter for which a valid region should be set.
in	*input_valid*	A pointer to an array of valid regions of input images or images contained in image container (e.g. pyramids). They are provided in same order as the parameter list of the kernel's declaration.
out	*output_valid*	An array of valid regions that should be set for the output images or image containers (e.g. pyramid) after graph processing. The length of the array should be equal to the size of the image container (e.g. number of levels in the pyramid). For a simple output image the array size is always one. Each rectangle supplies the valid region for one image. The array memory allocation is managed by the framework.

Returns

An error code describing the validation status on parameters.

Definition at line 1622 of file vx_types.h.

3.76.3 Enumeration Type Documentation

enum vx_meta_valid_rect_attribute_e

The meta valid rectangle attributes.

Enumerator

VX_VALID_RECT_CALLBACK Valid rectangle callback during output parameter validation. Write-only.

Definition at line 1099 of file vx_types.h.

3.76.4 Function Documentation

vx_status VX_API_CALL vxAllocateUserKernelId (vx_context *context,* vx_enum ∗ *pKernelEnumId*)

Allocates and registers user-defined kernel enumeration to a context. The allocated enumeration is from available pool of 4096 enumerations reserved for dynamic allocation from VX_KERNEL_BASE(VX_ID_USER,0).

Parameters

in	*context*	The reference to the implementation context.
out	*pKernelEnumId*	pointer to return `vx_enum` for user-defined kernel.

Return values

VX_SUCCESS	No errors.
VX_ERROR_NO_RESOURCES	The enumerations has been exhausted.

vx_status VX_API_CALL vxAllocateUserKernelLibraryId (vx_context *context,* vx_enum ∗ *pLibraryId*)

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

The allocated library ID is from available pool of library IDs (1..255) reserved for dynamic allocation. The returned libraryId can be used by user-kernel library developer to specify individual kernel enum IDs in a header file, shown below:

```
#define MY_KERNEL_ID1(libraryId) (VX_KERNEL_BASE(VX_ID_USER,libraryId) + 0);
#define MY_KERNEL_ID2(libraryId) (VX_KERNEL_BASE(VX_ID_USER,libraryId) + 1);
#define MY_KERNEL_ID3(libraryId) (VX_KERNEL_BASE(VX_ID_USER,libraryId) + 2);
```

Parameters

in	*context*	The reference to the implementation context.
out	*pLibraryId*	pointer to `vx_enum` for user-kernel libraryId.

Return values

VX_SUCCESS	No errors.
VX_ERROR_NO_RESOURCES	The enumerations has been exhausted.

vx_status VX_API_CALL vxLoadKernels (vx_context *context,* const vx_char ∗ *module*)

Loads a library of kernels, called module, into a context.

The module must be a dynamic library with by convention, two exported functions named `vxPublishKernels` and `vxUnpublishKernels`.

`vxPublishKernels` must have type `vx_publish_kernels_f`, and must add kernels to the context by calling `vxAddUserKernel` for each new kernel. `vxPublishKernels` is called by `vxLoadKernels`.

`vxUnpublishKernels` must have type `vx_unpublish_kernels_f`, and must remove kernels from the context by calling `vxRemoveKernel` for each kernel the `vxPublishKernels` has added. `vxUnpublishKernels` is called by `vxUnloadKernels`.

Note

When all references to loaded kernels are released, the module may be automatically unloaded.

Parameters

`in`	*context*	The reference to the context the kernels must be added to.
`in`	*module*	The short name of the module to load. On systems where there are specific naming conventions for modules, the name passed should ignore such conventions. For example: `libxyz.so` should be passed as just `xyz` and the implementation will *do the right thing* that the platform requires.

Note

This API uses the system pre-defined paths for modules.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the context is not a `vx_context`.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

See Also

vxGetKernelByName

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

Unloads all kernels from the OpenVX context that had been loaded from the module using the vxLoadKernels function.

The kernel unloading is performed by calling the `vxUnpublishKernels` exported function of the module.

Note

`vxUnpublishKernels` is defined in the description of `vxLoadKernels`.

Parameters

`in`	*context*	The reference to the context the kernels must be removed from.
`in`	*module*	The short name of the module to unload. On systems where there are specific naming conventions for modules, the name passed should ignore such conventions. For example: `libxyz.so` should be passed as just `xyz` and the implementation will *do the right thing* that the platform requires.

Note

This API uses the system pre-defined paths for modules.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the context is not a `vx_context`.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

See Also

vxLoadKernels

vx_kernel VX_API_CALL 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*)

Allows users to add custom kernels to a context at run-time.

Parameters

`in`	*context*	The reference to the context the kernel must be added to.
`in`	*name*	The string to use to match the kernel.
`in`	*enumeration*	The enumerated value of the kernel to be used by clients.
`in`	*func_ptr*	The process-local function pointer to be invoked.
`in`	*numParams*	The number of parameters for this kernel.
`in`	*validate*	The pointer to `vx_kernel_validate_f`, which validates parameters to this kernel.
`in`	*init*	The kernel initialization function.
`in`	*deinit*	The kernel de-initialization function.

Returns

`vx_kernel`. Any possible errors preventing a successful creation should be checked using `vxGetStatus`.

Return values

0	Indicates that an error occurred when adding the kernel.
$*$	Kernel added to OpenVX.

vx_status VX_API_CALL vxFinalizeKernel (vx_kernel *kernel*)

This API is called after all parameters have been added to the kernel and the kernel is *ready* to be used. Notice that the reference to the kernel created by vxAddUserKernel is still valid after the call to vxFinalizeKernel.

Parameters

`in`	*kernel*	The reference to the loaded kernel from `vxAddUserKernel`.

Returns

A `vx_status_e` enumeration. If an error occurs, the kernel is not available for usage by the clients of OpenVX. Typically this is due to a mismatch between the number of parameters requested and given.

Precondition

`vxAddUserKernel` and `vxAddParameterToKernel`

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

Allows users to set the signatures of the custom kernel.

Parameters

in	*kernel*	The reference to the kernel added with vxAddUserKernel.
in	*index*	The index of the parameter to add.
in	*dir*	The direction of the parameter. This must be either VX_INPUT or VX_OUTPUT. VX_BIDIRECTIONAL is not supported for this function.
in	*data_type*	The type of parameter. This must be a value from vx_type_e.
in	*state*	The state of the parameter (required or not). This must be a value from vx_parameter_state_e.

Returns

A vx_status_e enumerated value.

Return values

VX_SUCCESS	Parameter is successfully set on kernel.
VX_ERROR_INVALID_REFERENCE	The value passed as kernel was not a vx_kernel.

Precondition

vxAddUserKernel

vx_status VX_API_CALL vxRemoveKernel (vx_kernel *kernel*)

Removes a custom kernel from its context and releases it.

Parameters

in	*kernel*	The reference to the kernel to remove. Returned from vxAddUserKernel.

Note

Any kernel enumerated in the base standard cannot be removed; only kernels added through vxAddUserKernel can be removed.

Returns

A vx_status_e enumeration. The function returns to the application full control over the memory resources provided at the kernel creation time.

Return values

VX_ERROR_INVALID_REFERENCE	If an invalid kernel is passed in.
VX_ERROR_INVALID_PARAMETER	If a base kernel is passed in.
VX_FAILURE	If the application has not released all references to the kernel object OR if the application has not released all references to a node that is using this kernel OR if the application has not released all references to a graph which has nodes that is using this kernel.

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

Sets kernel attributes.

Parameters

in	*kernel*	The reference to the kernel.
in	*attribute*	The enumeration of the attributes. See vx_kernel_attribute_e.
in	*ptr*	The pointer to the location from which to read the attribute.
in	*size*	The size in bytes of the data area indicated by *ptr* in bytes.

Note

After a kernel has been passed to vxFinalizeKernel, no attributes can be altered.

Returns

A vx_status_e enumeration.

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

This function allows a user to set the attributes of a vx_meta_format object in a kernel output validator.

The vx_meta_format object contains two types of information: data object meta data and some specific information that defines how the valid region of an image changes

The meta data attributes that can be set are identified by this list:

- vx_image : VX_IMAGE_FORMAT, VX_IMAGE_HEIGHT, VX_IMAGE_WIDTH
- vx_array : VX_ARRAY_CAPACITY, VX_ARRAY_ITEMTYPE
- vx_pyramid : VX_PYRAMID_FORMAT, VX_PYRAMID_HEIGHT, VX_PYRAMID_WIDTH, VX_PYRAMID_LEVELS, VX_PYRAMID_SCALE
- vx_scalar : VX_SCALAR_TYPE
- vx_matrix : VX_MATRIX_TYPE, VX_MATRIX_ROWS, VX_MATRIX_COLUMNS
- vx_distribution : VX_DISTRIBUTION_BINS, VX_DISTRIBUTION_OFFSET, VX_DISTRIBUTION_RANGE
- vx_remap : VX_REMAP_SOURCE_WIDTH, VX_REMAP_SOURCE_HEIGHT, VX_REMAP_DESTINATION_WIDTH, VX_REMAP_DESTINATION_HEIGHT
- vx_lut : VX_LUT_TYPE, VX_LUT_COUNT
- vx_threshold : VX_THRESHOLD_TYPE
- VX_VALID_RECT_CALLBACK

Note

For vx_image, a specific attribute can be used to specify the valid region evolution. This information is not a meta data.

Parameters

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

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	The attribute was set.
VX_ERROR_INVALID_REFERENCE	meta was not a `vx_meta_format`.
VX_ERROR_INVALID_PARAMETER	size was not correct for the type needed.
VX_ERROR_NOT_SUPPORTED	the object attribute was not supported on the meta format object.
VX_ERROR_INVALID_TYPE	attribute type did not match known meta format type.

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

Set a meta format object from an exemplar data object reference.

This function sets a vx_meta_format object from the meta data of the exemplar

Parameters

`in`	*meta*	The meta format object to set
`in`	*exemplar*	The exemplar data object.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	The meta format was correctly set.
VX_ERROR_INVALID_REFERENCE	the reference was not a reference to a data object

3.77 Framework: Graph Parameters

3.77.1 Detailed Description

Defines the Graph Parameter API. Graph parameters allow Clients to create graphs with Client settable parameters. Clients can then create Graph creation methods (a.k.a. *Graph Factories*). When creating these factories, the client will typically not be able to use the standard Node creator functions such as `vxSobel3x3Node` but instead will use the *manual* method via `vxCreateGenericNode`.

```
vx_graph vxCornersGraphFactory(vx_context context)
{
    vx_status  status = VX_SUCCESS;
    vx_uint32  i;
    vx_float32 strength_thresh = 10000.0f;
    vx_float32 r = 1.5f;
    vx_float32 sensitivity = 0.14f;
    vx_int32 window_size = 3;
    vx_int32 block_size = 3;
    vx_enum channel = VX_CHANNEL_Y;
    vx_graph graph = vxCreateGraph(context);
    if (vxGetStatus((vx_reference)graph) == VX_SUCCESS)
    {
        vx_image virts[] = {
            vxCreateVirtualImage(graph, 0, 0,
      VX_DF_IMAGE_VIRT),
            vxCreateVirtualImage(graph, 0, 0,
      VX_DF_IMAGE_VIRT),
        };
        vx_kernel kernels[] = {
            vxGetKernelByEnum(context,
      VX_KERNEL_CHANNEL_EXTRACT),
            vxGetKernelByEnum(context, VX_KERNEL_MEDIAN_3x3),
            vxGetKernelByEnum(context, VX_KERNEL_HARRIS_CORNERS),
        };
        vx_node nodes[dimof(kernels)] = {
            vxCreateGenericNode(graph, kernels[0]),
            vxCreateGenericNode(graph, kernels[1]),
            vxCreateGenericNode(graph, kernels[2]),
        };
        vx_scalar scalars[] = {
            vxCreateScalar(context, VX_TYPE_ENUM, &channel),
            vxCreateScalar(context, VX_TYPE_FLOAT32, &strength_thresh),
            vxCreateScalar(context, VX_TYPE_FLOAT32, &r),
            vxCreateScalar(context, VX_TYPE_FLOAT32, &sensitivity),
            vxCreateScalar(context, VX_TYPE_INT32, &window_size),
            vxCreateScalar(context, VX_TYPE_INT32, &block_size),
        };
        vx_parameter parameters[] = {
            vxGetParameterByIndex(nodes[0], 0),
            vxGetParameterByIndex(nodes[2], 6)
        };
        // Channel Extract
        status |= vxAddParameterToGraph(graph, parameters[0]);
        status |= vxSetParameterByIndex(nodes[0], 1, (
      vx_reference)scalars[0]);
        status |= vxSetParameterByIndex(nodes[0], 2, (
      vx_reference)virts[0]);
        // Median Filter
        status |= vxSetParameterByIndex(nodes[1], 0, (
      vx_reference)virts[0]);
        status |= vxSetParameterByIndex(nodes[1], 1, (
      vx_reference)virts[1]);
        // Harris Corners
        status |= vxSetParameterByIndex(nodes[2], 0, (
      vx_reference)virts[1]);
        status |= vxSetParameterByIndex(nodes[2], 1, (
      vx_reference)scalars[1]);
        status |= vxSetParameterByIndex(nodes[2], 2, (
      vx_reference)scalars[2]);
        status |= vxSetParameterByIndex(nodes[2], 3, (
      vx_reference)scalars[3]);
        status |= vxSetParameterByIndex(nodes[2], 4, (
      vx_reference)scalars[4]);
        status |= vxSetParameterByIndex(nodes[2], 5, (
      vx_reference)scalars[5]);
        status |= vxAddParameterToGraph(graph, parameters[1]);

        for (i = 0; i < dimof(scalars); i++)
        {
            vxReleaseScalar(&scalars[i]);
        }
        for (i = 0; i < dimof(virts); i++)
        {
```

```
            vxReleaseImage(&virts[i]);
        }
        for (i = 0; i < dimof(kernels); i++)
        {
            vxReleaseKernel(&kernels[i]);
        }
        for (i = 0; i < dimof(nodes);i++)
        {
            vxReleaseNode(&nodes[i]);
        }
        for (i = 0; i < dimof(parameters); i++)
        {
            vxReleaseParameter(&parameters[i]);
        }
    }
    return graph;
}
```

Some data are contained in these Graphs and do not become exposed to Clients of the factory. This allows ISVs or Vendors to create custom IP or IP-sensitive factories that Clients can use but may not be able to determine what is inside the factory. As the graph contains internal references to the data, the objects will not be freed until the graph itself is released.

Functions

- vx_status VX_API_CALL vxAddParameterToGraph (vx_graph graph, vx_parameter parameter)

  Adds the given parameter extracted from a `vx_node` to the graph.
- vx_parameter VX_API_CALL vxGetGraphParameterByIndex (vx_graph graph, vx_uint32 index)

  Retrieves a `vx_parameter` from a `vx_graph`.
- vx_status VX_API_CALL vxSetGraphParameterByIndex (vx_graph graph, vx_uint32 index, vx_reference value)

  Sets a reference to the parameter on the graph. The implementation must set this parameter on the originating node as well.

3.77.2 Function Documentation

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

Adds the given parameter extracted from a `vx_node` to the graph.

Parameters

in	*graph*	The graph reference that contains the node.
in	*parameter*	The parameter reference to add to the graph from the node.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Parameter added to Graph.
VX_ERROR_INVALID_REFERENCE	The parameter is not a valid `vx_parameter`.
VX_ERROR_INVALID_PARAMETER	The parameter is of a node not in this graph.

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

Sets a reference to the parameter on the graph. The implementation must set this parameter on the originating node as well.

Parameters

in	*graph*	The graph reference.
in	*index*	The parameter index.
in	*value*	The reference to set to the parameter.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Parameter set to Graph.
VX_ERROR_INVALID_REFERENCE	The value is not a valid vx_reference.
VX_ERROR_INVALID_PARAMETER	The parameter index is out of bounds or the dir parameter is incorrect.

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

Retrieves a vx_parameter from a vx_graph.

Parameters

in	*graph*	The graph.
in	*index*	The index of the parameter.

Returns

vx_parameter reference. Any possible errors preventing a successful creation should be checked using vxGetStatus.

Return values

0	if the index is out of bounds.
*	The parameter reference.

Chapter 4

Data Structure Documentation

4.1 vx_delta_rectangle_t Struct Reference

Data Fields

- vx_int32 delta_end_x

  The change in the end x.
- vx_int32 delta_end_y

  The change in the end y.
- vx_int32 delta_start_x

  The change in the start x.
- vx_int32 delta_start_y

  The change in the start y.

4.1.1 Detailed Description

Definition at line 160 of file vx_compatibility.h.

4.1.2 Field Documentation

vx_int32 vx_delta_rectangle_t::delta_start_x

The change in the start x.
Definition at line 161 of file vx_compatibility.h.

vx_int32 vx_delta_rectangle_t::delta_start_y

The change in the start y.
Definition at line 162 of file vx_compatibility.h.

vx_int32 vx_delta_rectangle_t::delta_end_x

The change in the end x.
Definition at line 163 of file vx_compatibility.h.

vx_int32 vx_delta_rectangle_t::delta_end_y

The change in the end y.
Definition at line 164 of file vx_compatibility.h.

Bibliography

[1] Jean-Yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker description of the algorithm, 2000. 96

[2] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, June 1986. 50

[3] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European Conference on Computer Vision*, volume 1, pages 430–443, May 2006. 30, 70

[4] Edward Rosten, Reid Porter, and Tom Drummond. Faster and better: A machine learning approach to corner detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 32:105–119, October 2010. 30, 70

Index