The OpenVX™ Safe Casts Extension

The Khronos® OpenVX Working Group, Contributor: Steve Ramm

Version 1.3.1, Wed, 20 Nov 2024 17:10:13 +0000: Git branch information not available

Table of Contents

Chapter 1. Purpose

OpenVX was designed as a C API with an inheritance model.The specifications state that any OpenVX Object may be cast to a reference, and certain reference attributes are always available.

However, in C we do not have inheritance as such, and casting between pointers of different types is not a safe operation as defined by for example MISRA and a common static code analysis rule is to detect casts between pointers to different types.

Since all the OpenVX objects are defined as pointers to incomplete and unrelated types, there is no difference between casting a vx_image to a vx_reference, or the reverse operation, since C has no concept of "down-casts" or "up-casts".

We could reasonably argue a waiver with a safety auditor for a cast from vx_image to vx_reference, since here we logically have inheritance, but the reverse is not true. If we cast from vx_reference to vx_image, and this is often necessary, for example when retrieving an item from an object array, then we have to show due care and diligence and make sure that we check the run-time type information by querying the reference for its type before casting it.

This is all extra boiler plate code, potential for error, a lot of waivers and a lot of conversations with the auditor.

There are situations where we would like to create a new reference to the objects, with the correct type, incrementing the reference count, and there are situations where we would like just to assert the type of the reference "on the fly", for example when getting an item out of an object array - here we would not want to increment the reference count.

Finally, some of the APIs take as their parameter a pointer to a vx_reference, or to an array of vx_references, and for these APIs it would be useful to be able to cast safely from (vx_some_type *) to (vx_reference *), or to (const vx_reference *). There does not seem to be a need for the reverse cast, nor one that increments the reference count.

The following table summarises the known functions in the main specification and various extensions that require the conversion between `vx_reference` and other object types. Please note that this may not be an exhaustive list, and the functions may be used in many other ways than the examples given:

Function or function type	**Notes**	**Example use of cast function**
vxQueryReference, vxSetReferenceName, vxGetStatus, vxRetainReference, vxGetContext, vxAddLogEntry, vxHint, vxDirective	Downcast required for all objects	status = vxGetStatus(vxCastRefFromImage(image))
vxReleaseReference	Downcast required for pointer to vx_meta_format object	vxReleaseReference(vxCastRefFromMetaFormatP(&meta_format_variable))
vxSelectNode, vxCopyNode, vxuCopy, vxMoveNode, vxSwapNode, vxuSwap, vxSetParameterByIndex, vxSetParameterByReference, vxSetGraphParameterByIndex, vxSetMetaFormatFromReference	Downcasts required for all data objects	node = vxCopyNode(vxCastRefFromTensor(tensor_in), vxCastRefFromTensor(tensor_out))

Function or function type	Notes	Example use of cast function
vxCreateObjectArray, vxCreateVirtualObjectArray	Downcasts required for all data objects except delay and object array	array = vxCreateObjectArray(vxCastRefFromMatrix(matrix));
vxCreateDelay	Downcasts required for all data objects except delay	lut_delay = vxCreateDelay(vxCastRefFromLUT(lut))
vxGetObjectArrayItem	Upcasts required for all data objects except delay and object array	image = vxCastRefAsImage(vxGetObjectArrayItem(object_array, 2), &status)
vxGetReferenceFromDelay	Upcasts required for all data objects except delay	tensor = vxCastRefAsTensor(vxGetReferenceFromDelay(delay, 1), &status)
vxQueryParameter	Downcasts potentially required for pointer to all data objects that could be retrieved via VX_PARAMETER_REF	status = vxQueryParameter(param, VX_PARAMETER_REF, vxCastRefFromImageP(&image), sizeof(vx_reference))
vx_kernel_f, vx_kernel_deinitialize_f, vx_kernel_initialize_f, vx_kernel_validate_f	Upcasts required for all data objects	vx_image input_image = vxCastRefAsImage(parameters[0], &status)
vx_graph_parameter_queue_params_t, vxGraphParameterEnqueueReadyRef, vxGraphParameterDequeueDoneRef	Downcasts for pointers to and upcasts required for all data objects	status = vxGraphParameterEnqueueReadyRef(graph, graph_parameter_index, vxCastRefFromImageP(&image), 1)
vxRegisterEvent	Downcasts required for vx_graph, vx_parameter, vx_node	vxRegisterEvent(vxCastRefFromNode(node1), VX_EVENT_NODE_COMPLETED, 0, 0)
vxExportObjectsToMemory	Downcasts required for const pointers to all exportable objects of the IX extension (not vx_context, vx_import, vx_node, vx_kernel, vx_parameter or vx_meta_format)	vxExportObjectsToMemory(context, 1, vxCastRefFromGraphConstP(&graph), &uses, &ptr, &size)
vxGetImportReferenceByName	Upcasts required for all importable objects of the IX extension (not vx_context, vx_import, vx_node, vx_kernel, vx_parameter or vx_meta_format)	image = vxCastRefAsImage(vxGetImportReferenceByName(import, "input_image_0"), & status)
vxImportObjectsFromMemory	Downcasts for pointers and upcasts required for all importable objects of the IX extension	vxImportObjectsFromMemory(…); vx_image image_55 = vxGetRefAsImage(&refs[55], &status);
vxGetImportReferenceByIndex, vxGetImportReferenceByName	Upcasts required for all objects except vx_context, vx_import and vx_meta_format (XML extension)	image = vxCastRefAsImage(vxGetImportReferenceByIndex(import, 12), &status)

This extension, with a full set of functions to suit every eventuality, seeks to remove all the SCA warnings into one place. Better still, the warnings are all in the implementation, so the user of the OpenVX library does not need to use any casts at all.

As can be seen from the table above, upcasts are not required for vx_context, vx_import or vx_meta_format. There are also some types where downcasts of pointers or const pointers are not strictly required, however since the implementation cost is low these are kept.

The style of the six functions to be implemented for each type is as follows:

```
vx_some_type vxGetRefAsSomeType(const vx_reference *ref, vx_status *status);
vx_some_type vxCastRefAsSomeType(vx_reference ref, vx_status *status);
vx_reference vxGetRefFromSomeType(const vx_some_type *some_type);
vx_reference vxCastRefFromSomeType(vx_some_type some_type);
vx_reference *vxCastRefFromSomeTypeP(vx_some_type *p_some_type);
const vx_reference *vxCastRefFromSomeTypeConstP(const vx_some_type *p_some_type);
```

The "vxGet" form of the functions will increment the reference count on success and are intended to be used with variables as parameters (hence the pointers), whereas the "vxCast" form of the functions will not increment the reference count and so are intended to be used directly in the place of casts.

All the functions will return an error object in the event of failure, and none of the functions will update the status variable if it is already non-zero.

Chapter 2. Requirements

2.1. The vxGetRefAsSomeType functions.

For example:

```
vx_object_array vxGetRefAsObjectArray(const vx_reference *ref, vx_status *status);
```

2.1.1. Parameters

[REQ-SCF01]

These functions take a pointer to a vx_reference as their first parameter, and deliver a vx_some_type as their result. They all have a second parameter, which is a pointer to a vx_status variable, which shall be ignored if NULL.

2.1.2. Successful operation

[REQ-SCF02]

The functions check that their reference is valid, and that the type of the reference matches the type required. If all is good, then the reference count of the object is incremented, and the pointer returned as a pointer to the correct type. Note that the vx_status variable will not be updated at all, allowing any error to propagated from the previous operation.

2.1.3. Unsuccessful operation

[REQ-SCF03]

If the checks fail, then the vx_status variable will be updated with an error code only if the pointer is not NULL and the value was initially zero, in other words, a previous error will not be over-written. An error object will be returned.

2.2. The vxGetRefFromSomeType functions.

For example:

```
vx_reference vxGetRefFromImage(const vx_image *image);
```

2.2.1. Operation

[REQ-SCF04]

These functions take a pointer to a vx_some_type as their parameter, and deliver a vx_reference as a result. The reference count is incremented. As this is a simple down-cast, no checks are done and if an invalid object was passed an invalid object will be returned.

2.3. The vxCastRefAsSomeType functions.

For example:

```
vx_object_array vxCastRefAsObjectArray(vx_reference ref, vx_status *status);
```

2.3.1. Parameters

[REQ-SCF05]

These functions take a vx_reference as their first parameter, and deliver a vx_some_type as their result. They all have a second parameter, which is a pointer to a vx_status variable that shall ignored if NULL.

2.3.2. Successful operation

[REQ-SCF06]

The functions check that their reference is valid, and that the type of the reference matches the type required. If all is good, then the pointer is returned as a pointer to the correct type. The references count is not incremented. Note that the vx_status variable will not be updated at all, allowing any error to propagated from the previous operation.

2.3.3. Unsuccessful operation

[REQ-SCF07]

If the checks fail, then the vx_status variable will be updated with an error code only if the pointer is not NULL and the value was initially zero, in other words, a previous error will not be over-written. An error object will be returned.

2.4. The vxCastRefFromSomeType functions.

For example:

```
vx_reference vxCastRefFromImage(vx_image image);
```

2.4.1. Operation

[REQ-SCF08]

These functions take a vx_some_type as their parameter, and deliver a vx_reference as a result. The reference count is not incremented. As this is a simple down-cast, no checks are done and if an invalid object was passed an invalid object will be returned.

2.5. The vxCastRefFromSomeTypeP functions.

For example:

```
vx_reference *vxCastRefFromTensorP(vx_tensor *p_tensor);
```

2.5.1. Operation

[REQ-SCF09]

These functions take a pointer to vx_some_type as their parameter, and deliver a pointer to vx_reference as a result. As this is a simple down-cast, no checks are done and if a pointer to an invalid reference was passed then a pointer to an invalid reference will be returned.

2.6. The vxCastRefFromSomeTypeConstP functions.

For example:

```
const vx_reference *vxCastRefFromTensorConstP(const vx_tensor *p_tensor);
```

2.6.1. Operation

[REQ-SCF10]

These functions take a pointer to const vx_some_type as their parameter, and deliver a pointer to const vx_reference as a result. As this is a simple down-cast, no checks are done and if a pointer to an invalid reference was passed then a pointer to an invalid reference will be returned.

2.7. Scope: The types to be covered, requirement numbering

Requirements are identified by two numbers, one from the list above (SCFxx) and one from the list below (SCTxx)

Object type to be supported	Requirement number	Note
array	[REQ-SCT01]	
context	[REQ-SCT03]	Upcast not required
convolution	[REQ-SCT02]	
delay	[REQ-SCT04]	
distribution	[REQ-SCT05]	
graph	[REQ-SCT06]	Upcast required only if XML or IX extension supported
image	[REQ-SCT07]	
import	[REQ-SCT08]	Upcast not required, downcast required only if XML or IX supported
kernel	[REQ-SCT09]	Upcast required only if XML extension supported
lut	[REQ-SCT10]	
matrix	[REQ-SCT11]	
meta_format	[REQ-SCT12]	Upcast not required
node	[REQ-SCT13]	Upcast required only if XML extension supported

Object type to be supported	Requirement number	Note
object_array	[REQ-SCT14]	
parameter	[REQ-SCT15]	Upcast required only if XML extension supported
pyramid	[REQ-SCT16]	
remap	[REQ-SCT17]	
scalar	[REQ-SCT18]	
tensor	[REQ-SCT19]	
threshold	[REQ-SCT20]	
user_data_object	[REQ-SCT21]	If vx_khr_user_data_object supported

Chapter 3. Usage

Hints and cautions for usage.

3.1. Reference counting

Where a variable is being copied, use the "vxGet" form of conversion (passing a pointer to that variable), but where a value is being modified inside an expression, i.e. the reference will be thrown away by the compiler, use the "vxCast" form of conversion, for example:

```
vx_status status = VX_SUCCESS;
vx_image image = vxCreateImage(context, 1200, 600, VX_DF_IMAGE_RGB);
/* New variable, reference count will be incremented */
vx_reference ref = vxGetRefFromImage(&image, &status);
...
vx_object_array obj_array = vxCreateObjectArray(image, 10);
/* Don't increment the ref count of a transient object */
vx_image item5 = vxCastRefAsImage(vxGetObjectArrayItem(obj_array, 5), &status);
vxSetReferenceName(vxCastRefFromImage(item5), "my_image");
...
vxReleaseImage(&item5);
vxReleaseReference(&ref);
vxReleaseImage(&image);
vxReleaseObjectArray(&obj_array);
```

3.2. Status values

Non-zero status values are always propagated.

```
vx_status status = VX_SUCCESS;
vx_image good_image = vxCreateImage(context, 1200, 600, VX_DF_IMAGE_U8);
vx_image bad_image = vxCreateImage(context, 1200, 600, -1);
status = vxGetStatus(vxCastRefFromImage(bad_image));

/* New variable, reference count will be incremented */
vx_reference ref1 = vxGetRefFromImage(&bad_image, &status);
vx_reference ref2 = vxGetRefFromImage(&good_image, &status);

if (VX_ERROR_INVALID_FORMAT == status)
{
  /* When vxGetReferenceFromImage was called, the status was not updated, neither for the invalid nor the
valid
  image, because it was already non-zero. So the original bad image creation error is still there... */
}
```