NVIDIA NVSCI for L4T

Developer Guide

Document History

Doc_Number: SWE-C2C-002-PGRF

Version	Date	Description of Change
01	April 7, 2022	Initial release

NVSCI FOR L4T

Table of Contents

NvSci FOR L4T 3

NvSci for L4T Developer Guide 8

- Comparison with EGL 8
- NvStreams Libraries 9
- Buffer Allocation 9
  - Memory Buffer Basics 9
    - NvSciBuf Module 10
      - NvSciBufModule 10
    - Attribute Lists 11
  - Multi Datatype Attribute Lists 11
    - Limitations 12
  - Reconciliation 12
    - NvSciBufAttrLists 13
    - NvSciBuf Reconciliation 14
  - Buffer Management 14
    - Objects 14
      - NvSciBuf Object 15
  - Inter-Application 15
    - NvSciIpc Init 15
    - NvSciBufObj Permissions 16
    - Notations 16
    - Computation of Object Export Permissions 17
    - Computation of Imported NvSciBufObj Permissions 17
    - Export/Import NvSciBuf AttrLists 18
    - Export/Import NvSciBufObj 19
  - NvSciBuf API 20
  - UMD Access 20
    - CUDA 20
      - NvSciBufObj as a CUDA Pointer / Array 20
      - NvSciBuf-CUDA Interop 21
- Streaming 22
  - A Simple Stream 23
    - Setup 23
      - Simple Stream Setup 23
    - Streaming 26
      - Simple Stream Loop 26

More Complex Streams 27
  Multiple Buffers 28
  FIFO Mode 28
  Mailbox Mode 29
  Multiple Acquired Frames 30
  Multiple Consumers 31
  Limit Acquired Frames 34
  Cross-Application 34
  Forced Sychronization of Payloads 34
Data Packets 34
  Element Type 35
  Element Mode 36
Building Block Model 36
  Endpoints 37
  Producer 37
  Consumer 38
  Multicast 39
  IPC 40
    Memory Sharing IPC 40
    Memory Boundary IPC 41
  Pool 42
    Static Pool 42
    Dynamic Pool 43
  Queue 43
  Limiter 44
    ReturnSync 44
  Example 45
Stream Creation 46
  NvSciBuf and NvSciSync Initialization 47
  NvSciIpc Initialization 47
  NvSciEventService Initialization 47
  Block Creation 47
    Pool 48
    Producer 48
    Multicast 49
    Queues 49
    Consumer 50
    IPC 50
    Limiter 52
    PresentSync 52
    ReturnSync 52

Configure Block to Use NvSciEventService 53
User-defined Endpoint Information 53
Block Connection 54
Comparison with EGL 54
Simple Example 55
Sample Producer Creation 55
Sample Consumer Creation 56
Event Handling 56
Event Query 56
Event Notification 57
Connection and Disconnection Events 58
Error Events 58
Resource Creation 58
Synchronization Resources 59
Synchronization Requirements 59
Receiving Requirements 60
Synchronization Objects 60
Sending Objects 60
Receiving Objects 61
Comparison with EGL 62
Buffer Resources 62
Buffer Requirements 62
Specifying Requirements 62
Receiving Requirements 63
Reconciling Requirements 64
Buffer Exchange 65
Specifying Buffers 65
Receiving Buffers 66
Completing Buffer Setup 67
Comparison with EGL 68
Frame Production 68
Obtaining Packets 68
Writing Packets 69
Presenting Packets 69
Comparison with EGL 70
Frame Consumption 71
Acquiring Packets 71
Reading Packets 71
Releasing Packets 71
Comparison with EGL 72
Teardown 72

Packet Destruction ... 72
Block Destruction ... 73
Synchronization ... 73
Terminology ... 73
Synchronization Basics ... 74
NvSciSync Module ... 74
NvSciSyncModule ... 74
Inter-Application ... 75
NvSciIpc init ... 75
NvSciSync Attributes ... 75
NvSciSync Attributes List ... 75
NvSciSyncAttrList ... 75
Reconciliation ... 76
Inter-Application ... 77
Export/Import NvSciSyncattrList ... 77
Sync Management ... 78
NvSciSync Objects ... 78
NvSciSyncObj ... 78
Inter-Application ... 78
Export/Import NvSciSyncObj ... 78
Cpu Wait Contexts ... 79
NvSciSyncCpuWaitContext ... 79
NvSciSyncFence Operations ... 79
NvSciSyncFence Cpu operations ... 80
Inter-Application ... 80
Export/Import NvSciSyncFence ... 80
Timestamps ... 81
Waiter requires timestamp ... 81
Waiter gets timestamp ... 81
UMD Access ... 82
CUDA ... 82
CUDA APIs ... 82
Query NvSciSyncObj attributes (for waiting or signaling) from CUDA ... 82
NvSciSync object registration/unregistration with CUDA ... 82
Wait for an NvSciSyncFence. ... 82
Get an NvSciSyncFence. ... 83
CUDA-NvSciSync API Usage ... 83
Sample Application ... 85
CPU Signaler Usage ... 85
CPU Waiter Usage ... 88
Inter-Process Communication ... 89

Terminology 90
NvSciIpc Configuration Data 91
Example NvSciIpc DT node 91
Example NvSciIpc config file format 92
Adding a New Channel 93
Adding a New INTER_THREAD and INTER_PROCESS Channel 93
Adding a New INTER_VM Channel 94
NvSciIpc API Usage 95
Prepare an NvSciIpc Endpoint for Read/Write 95
Writing to the NvSciIpc Endpoint 96
Reading from the NvSciIpc Endpoint 98
Prepare an NvSciIpc Endpoint for read/write 99
Writing to the NvSciIpc Endpoint 99
Reading from the NvSciIpc Endpoint 100
Cleaning-up an NvSciIpc Endpoint 101
De-Initialize NvSciIpc Library 101
De-Initialize NvSciIpc Library 101
NvSciEventService API Usage 101
Preparing an Endpoint with NvSciEventService for Read/Write 102
Waiting for a Single Event for Read/Write 103
Waiting for Multiple Events for Read/Write 104
Creating a Local Event 105
Sending a Signal with a Local Event 106
Waiting for a Local Event 106
Cleaning Up Event Notifier and Local Event 107
De-Initializing NvSciEventService Library 107
NVSCI APIs on Jetson Linux 107
Differences for NvStreams and NvSciIpc on Jetson Linux 107
API Reference for Software Communication Interfaces 107

NvSci for L4T Developer Guide

The NVIDIA® SDK provides several different libraries for generating and processing various types of images and higher dimensional data. The interfaces used by these libraries were written independently to serve different needs, and each has its own means of representing memory and synchronization resources. There is no direct way to exchange data between them. Furthermore, most of the details of how and when resources used by these libraries are allocated is hidden from the application(s).

The NvStreams libraries serve two primary purposes:

- They allow resources to be allocated up front, with access restrictions defined by the application(s), and with details of their overall requirements well understood. This is vital for safety-critical systems that must ensure the availability of resources and proper encapsulation of information.
- They allow resources to be exchanged between libraries that otherwise do not have knowledge of each other.

This document describes three libraries. `NvSciBuf` allows applications to allocate and exchange buffers in memory. `NvSciSync` allows applications to manage synchronization objects that coordinate when sequence of operations begin and end. `NvSciStream` layers on `NvSciBuf` and `NvSciSync` to provide utilities for streaming sequences of data packets between multiple application modules to support a wide variety of use cases. These libraries also make use of NvSciIpc for inter-process/partition/system communication. This library is described in a separate chapter.

Comparison with EGL

Developers familiar with NVIDIA's non-safety SDK may have experience with EGL, which also provides objects (EGLImage, EGLSync, and EGLStream) for sharing resources between libraries. NVIDIA continues to support EGL, but EGL is not suitable for the rigorous requirements of a safety-certified system. Some reasons:

- Resources are allocated using one library. That library has no knowledge that resources will be shared with another library. There is therefore no guarantee that resources will be allocated in a way that meets the requirements of the other library. Mapping them may not be possible.
- EGL understands only two-dimensional image data. It cannot handle tensors or other non-image sensor data.
- The EGL interfaces were not designed with safety in mind and have failure modes not allowed in a safety-certified system.

Porting EGL applications to NvStreams requires more than a simple one-to-one replacement of functions. NvStreams requires the application to be more directly involved than EGL in determining resource requirements, allocating resources, and exchanging resources.

NvStreams Libraries

For more information about NvStreams libraries, see:

- Buffer Allocation
- Synchronization

Buffer Allocation

Every hardware engine inside NVIDIA hardware can have a different buffer constraint depending on how the buffer is interpreted by the engine. Hence, sharing a buffer across various engines requires that the allocated buffer satisfy the constraints of all engines that will access that buffer. The existing allocation APIs provided by existing libraries only consider the constraints of the engines managed by them.

NvSciBuf is a buffer allocation module that can enable applications to allocate a buffer shareable across various hardware engines that are managed by different engine APIs.

Memory Buffer Basics

The buffer allocation model of NvSciBuf is summarized as follows:

If two or more hardware engines want to access a common buffer (e.g., one engine is writing data into the buffer and the other engine is reading from the buffer), then:

Allocation Model

1. Applications create an attribute list for each accessor.
2. Set the attributes to define the properties of the buffer they intend to create in the respective attribute list.
   - For CUDA, applications must set all the required attributes.

Note:	**Applications must ensure they set the `NvSciBufGeneralAttrKey_GpuId` attribute on the CUDA side to specify the IDs of all the GPUs that access the buffer.**

3. Reconcile these multiple attribute lists. The process of reconciliation guarantees that a common buffer is allocated that satisfies the constraints of all the accessors.
4. Allocate the buffer using the reconciled attribute list. The reconciled attribute list used for allocating the object is associated with the object until the lifetime of the object.
5. Share the buffer with all the accessors.

Types of Buffers

The hardware engine constraints depend on the type of buffer allocated. The different types of buffers supported by NvSciBuf (applications can choose to allocate one of the following types):

- RawBuffer: Raw memory that is used by an application for storing data.
- Image: Memory used to store image data.
- ImagePyramid: Memory used to store ImagePyramid, a group of images arranged in multiple levels, with each level of image scaled to a specific scaling factor.
- NvSciBufArray: Memory used to store a group of units, where each unit represents data of various basic types such as int, float etc.
- Tensor: Memory used to store Tensor data.

Memory Domain Allocation

Applications can choose to allocate the memory from the following domains:

- System memory
- Vidmem of a specific dGPU (on the standard build)

Types of Buffer Attributes

The NvSciBuf attribute can be categorized into the following types:

Datatype attributes: Attributes that are specific to one of the buffer types mentioned in the previous section. If the buffer type in the attribute list is one type, then setting the attributes of another type returns an error.

General attributes: Attributes that are not specific to any buffer type and describes the general properties of the buffer. Some of the examples:

- `NvSciBufGeneralAttrKey_Types`: Defines the type of the buffer.
- `NvSciBufGeneralAttrKey_NeedCpuAccess`: Defines whether the buffer is accessed by the CPU.
- `NvSciBufGeneralAttrKey_RequiredPerm`: Defines the access permissions expected by this buffer accessor.

NvSciBuf Module

You must open an `NvSciBufModule` before invoking other NvSciBuf API. The NvSciBuf module is the library's instance created for that application. All NvSciBuf resources created within an application are associated with the `NvSciBufModule` of the application.

NvSciBufModule

```
    NvSciBufModule module = NULL;
    NvSciError err;
    err = NvSciBufModuleOpen(&module);
    if (err != NvSciError_Success) {
        goto fail;
    }
    /* ... */
    NvSciBufModuleClose(module);
```

Attribute Lists

NvSciBuf attribute lists are categorized into the following types:

- Unreconciled Attribute List
  - An application can create an unreconciled attribute list and perform get/set operations for each attribute in an unreconciled attribute list.

    Note: The 'set' operation is allowed only once per attribute.

  - Both the `NvSciBufAttrListCreate` and `NvSciBufAttrListIpcImportUnreconciled` APIs return an unreconciled attribute list.
  - Applications cannot use unreconciled attribute lists to allocate an NvSciBuf object.
- Reconciled Attribute List
  - A reconciled attribute list is the outcome of reconciliation (i.e., merging and validation) of various unreconciled lists that define final layout/allocation properties of the buffer. The application can use this list to allocate an NvSciBuf object. Refer to the Error! Reference source not found. section below for more details about the reconciliation process.
  - A successful call to `NvSciBufAttrListReconcile`, `NvSciBufAttrListIpcImportReconciled`, or `NvSciBufObjGetAttrList` APIs returns a reconciled attribute list.
  - Applications are not allowed to perform set operations on a reconciled attribute list.
- Conflict Attribute List
  - An attribute list returned by an unsuccessful reconciliation process of `NvSciBufAttrListReconcile` API is a conflict attribute list.
  - Applications are not allowed to perform get/set operations on a conflict attribute list.
  - Applications can only use conflict attribute lists to dump its content using the `NvSciBufAttrListDump` API.

Multi Datatype Attribute Lists

To support the use cases where applications can perceive a buffer with distinct datatypes, NvSciBuf supports either creating attribute lists with multiple datatypes or reconciling attribute lists of distinct datatypes. For example, if an application wants to create a buffer that is perceived as image by one engine and perceived as tensor by another engine, then the application can create an attribute list with NvSciBufGeneralAttrKey_Types attribute set to both image and tensor, reconcile the list and allocate the object. Alternatively, the application can create two different attribute lists, one with image attributes and the other with tensor attributes, reconcile both and allocate the object.

Limitations

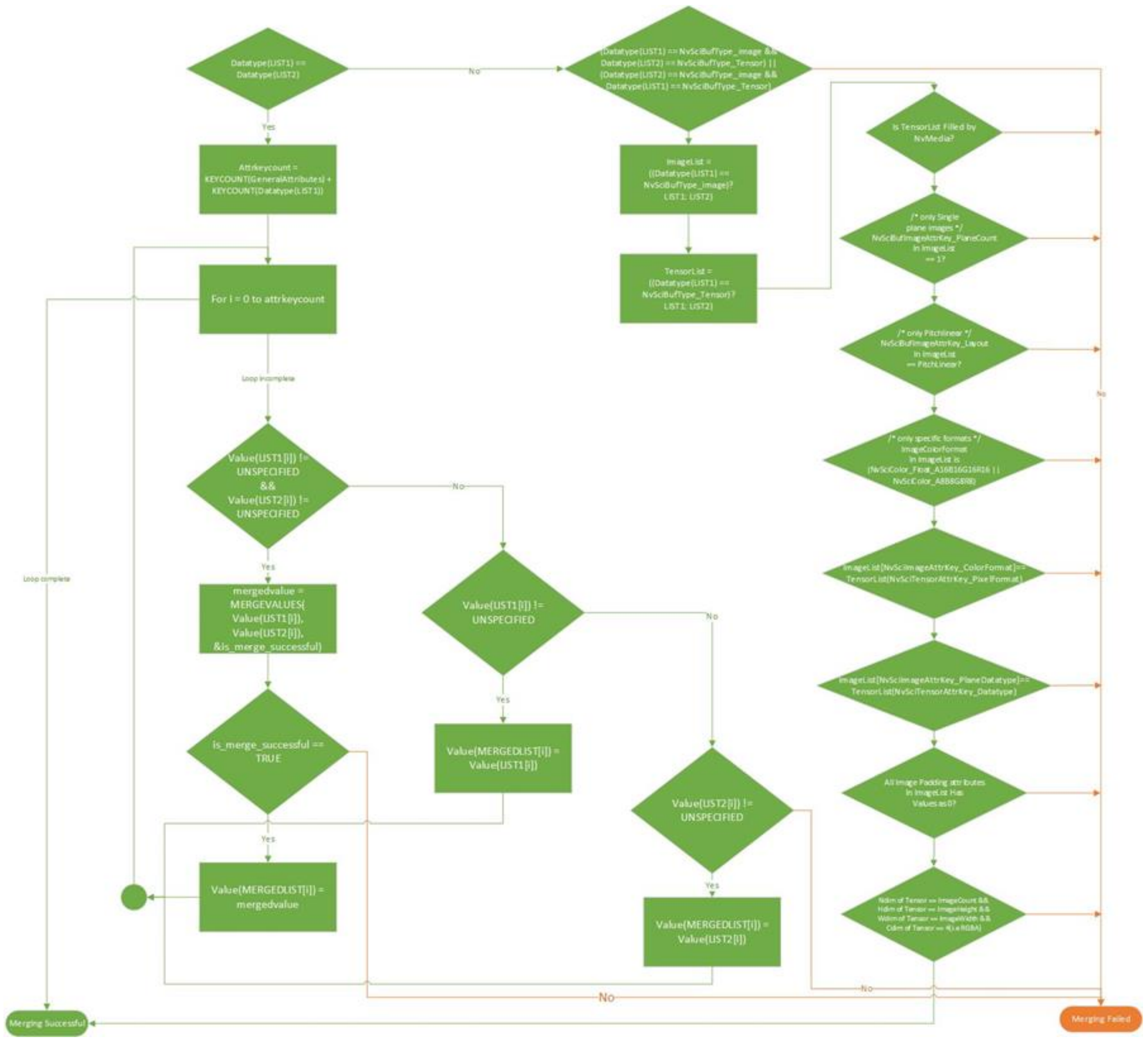
- NvSciBuf supports attribute lists to be either created or reconciled only with image and tensor attributes. Rest of the NvSciBuf datatypes doesn't support co-existence with other datatypes.

Reconciliation

An application can initiate the process of reconciliation on one or more attribute lists by invoking `NvSciBufAttrListReconcile` API. The process of NvSciBuf attribute list reconciliation:

1. **Merging**: Values from multiple attribute lists are merged. The process of merging is explained in the flow-chart below.

- `Datatype(List)`: Datatype of the attribute list `list`.
- `List[i]`: Attribute key named `i` of the attribute list `list`.
- `Value(List[i])`: Value corresponding to attribute key `i` in the attribute list `list`.
- `UNSPECIFIED`: Value of an attribute key is ignored and hence unspecified.
- `KEYCOUNT(datatype)`: Number of attribute keys for the given datatype.
- `MERGEVALUES(value1, value2)`: This function merges value1 and value2.
  - For most attributes, merging is successful only if value1 equals value2.
  - For attributes like alignment, if value1 is not equal to value 2, then the maximum of both is used as the merged value, provided both of them are a power of 2.



Datatype(LIST1) == Datatype(LIST2)
No
Yes
Attrkeycount = KEYCOUNT(GeneralAttributes) + KEYCOUNT(Datatype(LIST1))
For i = 0 to attrkeycount
Loop incomplete
Loop complete
Value(LIST1[i]) != UNSPECIFIED && Value(LIST2[i]) != UNSPECIFIED
mergedvalue = MERGEVALUES(Value(LIST1[i]), Value(LIST2[i]), &is_merge_successful)
is_merge_successful == TRUE
Value(MERGEDLIST[i]) = mergedvalue
Value(LIST1[i]) != UNSPECIFIED
Value(MERGEDLIST[i]) = Value(LIST1[i])
Value(LIST2[i]) != UNSPECIFIED
Value(MERGEDLIST[i]) = Value(LIST2[i])
(Datatype(LIST1) == NvSciBufType_image && Datatype(LIST2) == NvSciBufType_Tensor) || (Datatype(LIST2) == NvSciBufType_image && Datatype(LIST1) == NvSciBufType_Tensor)
ImageList = ((Datatype(LIST1) == NvSciBufType_image)? LIST1: LIST2)
TensorList = ((Datatype(LIST1) == NvSciBufType_Tensor)? LIST1: LIST2)
Is TensorList Filled by NvMedia?
/* only Single plane images */ NvSciBufImageAttrKey_PlaneCount in ImageList == 1?
/* only Pitchlinear */ NvSciBufImageAttrKey_Layout in ImageList == PitchLinear?
/* only specific formats */ ImageColorformat in ImageList is (NvSciColor_Float_A16B16G16R16 || NvSciColor_A8B8G8R8)
ImageList[NvSciImageAttrKey_ColorFormat]== TensorList[NvSciTensorAttrKey_PixelFormat]
ImageList[NvSciImageAttrKey_PlaneDatatype]== TensorList[NvSciTensorAttrKey_Datatype]
All Image Padding attributes in ImageList Has Values as 0?
Ndim of Tensor == ImageCount && Hdim of Tensor == ImageHeight && Wdim of Tensor == ImageWidth && Cdim of Tensor == 4(i.e RGBA)
No
Merging Successful
Merging Failed

2. **Validation**: After merging attributes from multiple lists successfully, validation of merged attributes occurs on the reconciled attribute list, which validates whether all the required attributes are set and the values of all the attributes are valid. For example, after the merging of attributes, if `plane-count` is set to 2 but the reconciled list contains more values for plane color-format, then validation is unsuccessful.
3. **Output Attributes Computation**: Post validation, output attributes like size, alignment, etc. are computed in the reconciled attribute list.

NvSciBufAttrLists

```
    NvSciBufType bufType = NvSciBufType_RawBuffer;
    uint64_t rawsize = (128 * 1024);  // Allocate 128K Raw-buffer
    uint64_t align = (4 * 1024);  //Buffer Alignment of 4K
    bool cpuaccess_flag = false;
    NvSciBufAttrKeyValuePair rawbuffattrs[] = {
        { NvSciBufGeneralAttrKey_Types, &bufType, sizeof(bufType) },
        { NvSciBufRawBufferAttrKey_Size, &rawsize, sizeof(rawsize) },
        { NvSciBufRawBufferAttrKey_Align, &align, sizeof(align) },
        { NvSciBufGeneralAttrKey_NeedCpuAccess, &cpuaccess_flag,
                sizeof(cpuaccess_flag) },
    };
    /* Created attrlist1 will be associated with bufmodule */
    err = NvSciBufAttrListCreate(bufmodule, &attrlist1);
     if (err != NvSciError_Success) {
    goto fail;
    }
    err = NvSciBufAttrListSetAttrs(umd1attrlist, rawbuffattrs,
            sizeof(rawbuffattrs)/sizeof(NvSciBufAttrKeyValuePair));
    /*......*/
    NvSciBufAttrListFree(attrlist1);
```

NvSciBuf Reconciliation

```
    NvSciBufAttrList unreconciledList[2] = {NULL};
    NvSciBufAttrList reconciledList = NULL;
    NvSciBufAttrList ConflictList = NULL;

    unreconciledList[0] = AttrList1;
    unreconciledList[1] = AttrList2;

    /* Reconciliation will be successful if and only all the
    * unreconciledLists belong to same NvSciBufModule and the
    * outputs of this API(i.e either reconciled attribute list
    * or conflict list will also be associated with the same
    * module with which input unreconciled lists belong to.
    */
    err = NvSciBufAttrListReconcile(
       unreconciledList,       /* array of unreconciled lists */
       2,                      /* size of this array */
       &reconciledList,        /* output reconciled list */
       &ConflictList);         /* conflict description filled
                               in case of reconciliation failure */

    if (err != NvSciError_Success) {
        goto fail;
    }
    /* ... */
    NvSciBufAttrListFree(AttrList1);
    NvSciBufAttrListFree(AttrList2);
    NvSciBufAttrListFree(reconciledList); //
          In case of successful reconciliation.
    NvSciBufAttrListFree(ConflictList); //
          In case of failed reconciliation.
```

Buffer Management

Objects

Applications can use the reconciled attribute list to create any number of NvSciBufObj objects. Each NvSciBufObj represents a buffer and the reconciled attribute list is associated with each object until the object is freed. Applications can create datatypes out of the allocated buffer using APIs and can operate on the buffers by submitting to the hardware engine using appropriate APIs. Applications wanting to access the buffer from the CPU can set the `NvSciBufGeneralAttrKey_NeedCpuAccess` attribute to `true` and get the CPU address using either `NvSciBufObjGetCpuPtr` or `NvSciBufObjGetConstCpuPtr` API.

Note: **Invoking NvSciBufObjGetCpuPtr on a read-only buffer or a buffer that does not have CPU access returns an error.**

NvSciBuf Object

```
    /* Allocate a Buffer using reconciled attribute list and the
    * created NvSciBufObj will be associated with the module to
    * which reconciledAttrlist belongs to.
    */
    err = NvSciBufAttrListObjAlloc(reconciledAttrlist,
        &nvscibufobj);
    if (err != NvSciError_Success) {
            goto fail;
    }
    /* ..... */
    /* Get the associated reconciled attrlist of the object. */
    err = NvSciBufObjGetAttrList(nvscibufobj,
        &objReconciledAttrList);
    if (err != NvSciError_Success) {
            goto fail;
    }
    /* ..... */
    err = NvSciBufObjGetCpuPtr(nvscibufobj, &va_ptr);
    if (err != NvSciError_Success) {
            goto fail;
    }
    /* ..... */
    NvSciBufAttrListFree(objReconciledAttrList);
    NvSciBufObjFree(nvscibufobj);
```

Inter-Application

If applications involve multiple processes, the exchange of NvSciBuf structures must only go through NvSciIpc channels. Each application must open its own NvSciIpc endpoint.

NvSciIpc Init

```
    NvSciIpcEndpoint ipcEndpoint = 0;
    err = NvSciIpcInit();
    if (err != NvSciError_Success) {
        goto fail;
    }
    err = NvSciIpcOpenEndpoint("ipc_endpoint", &ipcEndpoint);
    if (err != NvSciError_Success) {
        goto fail;
    }
    /* ... */
    NvSciIpcCloseEndpoint(ipcEndpoint);
    NvSciIpcDeinit();
```

Applications connected through an NvSciIpcEndpoint can exchange NvSciBuf structures (`NvSciBufAttrList` or `NvSciBufObj`) using the export/import APIs provided by NvSciBuf.

- NvSciBuf Export APIs return an appropriate export descriptor for the specified NvSciIpcEndpoint.
- Applications are responsible for transporting the export descriptor returned by NvSciBuf using the same NvSciIpcEndpoint.
- NvSciBuf Import APIs return the respective NvSciBuf structure for the specified export descriptor.
- NvSciBuf provides different APIs for transporting reconciled and unreconciled attribute lists. For reconciled attribute lists, applications can optionally validate the reconciled *list against one or* more unreconciled attribute lists to ensure that the reconciled attribute list satisfies the parameters of the importing *process' unreconciled* lists. This can be done by either passing unreconciled lists to `NvSciBufAttrListIpcImportReconciled` API while importing or by invoking `NvSciBufAttrListValidateReconciled` API after importing.

NvSciBufObj Permissions

- Applications have two options to specify their intended permissions on the NvSciBufObj:
  - Option 1: Set the intended permissions to NvSciBufGeneralAttrKey_RequiredPerm attribute in the unreconciled attribute-list. If no permissions are specified by the application, NvSciBufAccessPerm_Readonly becomes default value.
  - Option 2: Set the intended permissions in the object export/import APIs.
- The final permissions of the NvSciBufObj for an application is computed by NvSciBuf based on multiple factors explained below:
  - Applications that allocates the NvSciBufObj using NvSciBufObjAlloc or NvSciBufAttrListReconcileAndObjAlloc APIs will always get Read/Write permissions.
  - For applications importing the NvSciBufObj, the permissions are computed based the following:

Notations

- ReconList.Perm – The value of NvSciBufGeneralAttrKey_ActualPerm key in the reconciled attribute list. In simple cases, this value is same as the value of NvSciBufGeneralAttrKey_RequiredPerm key set by the application in its unreconciled attribute list. In cases where an application has multiple unreconciled lists, the NvSciBufGeneralAttrKey_ActualPerm value is the maximum value of the NvSciBufGeneralAttrKey_RequiredPerm key of all the unreconciled lists.
  - ExportAPI.Perm – The value of export permissions argument used by application with object export APIs such as NvSciBufIpcExportAttrListAndObj or NvSciBufObjIpcExport API.

- ImportAPI.Perm – The value of import permissions argument used by application with object import APIs such as NvSciBufIpcImportAttrListAndObj or NvSciBufObjIpcImport API.
- ObjExport.Perm – The value of final computed permissions with which NvSciBufObj is exported from the export application.
- ObjImport.Perm – The value of final permissions with which NvSciBufObj is imported by the import application.

Computation of Object Export Permissions

- Case-1: ExportAPI.Perm = NvSciBufAccessPerm_Auto. Export Application invoked the Export API with NvSciBufAccessPerm_Auto option. In this case:

```
ObjExport.Perm =  ReconList.Perm
```

- Case-2: ExportAPI.Perm = **explicit permissions**. Export Application invoked the Export API by explicitly specifying the permissions. In this case, ObjExport.Perm is computed as shown below:

INPUT		OUTPUT
ReconList.Perm	ExportAPI.Perm	ObjExport.Perm
RO	RO	RO
RO	RW	RW
RW	RO	RW
RW	RW	RW

Computation of Imported NvSciBufObj Permissions

- Case-1: ImportAPI.Perm = NvSciBufAccessPerm_Auto. Import Application invoked the Import API with NvSciBufAccessPerm_Auto option. In this case:

```
ObjImport.Perm =  ObjExport.Perm
```

- Case-2: ImportAPI.Perm = **explicit permissions.** Import Application invoked the Import API by explicitly specifying the permissions. In this case,ObjImport.Perm is computed as shown below:

INPUT		OUTPUT
ObjExport.Perm	ImportAPI.Perm	ObjImport.Perm
RO	RO	RO
RO	RW	Import Fail
RW	RO	RW
RW	RW	RW

Note: **While computing the imported NvSciBufObj permissions, the permissions in the reconciled list are ignored. In fact, the value of `NvSciBufGeneralAttrKey_ActualPerm` in the reconciled list is updated to the value of `ObjImport.Perm`.**

Note: **`NvSciBufObj` Import APIs return a failure if the export descriptors are imported using a wrong ipcEndpoint or inappropriate permissions.**

Export/Import NvSciBuf AttrLists

```
/* --------------------------App Process1 ----------------------------------*/
    NvSciBufAttrList AttrList1 = NULL;
    void* ListDesc = NULL;
    size_t ListDescSize = 0U;
    /* creation of the attribute list, receiving other lists from other listeners */
    err = NvSciBufAttrListIpcExportUnreconciled(
        &AttrList1,                    /* array of unreconciled lists to be exported */
        1,                             /* size of the array */
        ipcEndpoint,                   /* valid and opened NvSciIpcEndpoint intended to
send the descriptor through */
        &ListDesc,                     /* The descriptor buffer to be allocated and fill
ed in */
        &ListDescSize );               /* size of the newly created buffer */
    if (err != NvSciError_Success) {
        goto fail;
    }
    /* send the descriptor to the process2 */
    /* wait for process 1 to reconcile and export reconciled list */
    err = NvSciBufAttrListIpcImportReconciled(
        module,                        /* NvSciBuf module using which this attrlist to b
e
imported */
        ipcEndpoint,                   /* valid and opened NvSciIpcEndpoint on which the

descriptor is received */
        ListDesc,                      /* The descriptor buffer to be imported */
        ListDescSize,                  /* size of the descriptor buffer */
        &AttrList1,                    /* array of unreconciled lists to be used for
validating the reconciled list */
        1,                             /* Number or unreconciled lists */
        &reconciledAttrList,           /* Imported reconciled list */
    if (err != NvSciError_Success) {
        goto fail;
    }
    /* --------------------------App Process2 ----------------------------------*/
    void* ListDesc = NULL;
    size_t ListDescSize = 0U;
    NvSciBufAttrList unreconciledList[2] = {NULL};
    NvSciBufAttrList reconciledList = NULL;
    NvSciBufAttrList newConflictList = NULL;
    NvSciBufAttrList AttrList2 = NULL;
    NvSciBufAttrList importedUnreconciledAttrList = NULL;
    /* create the local AttrList */
    /* receive the descriptor from the other process */
    err = NvSciBufAttrListIpcImportUnreconciled(module, ipcEndpoint,
        ListDesc, ListDescSize,
        &importedUnreconciledAttrList);
    if (err != NvSciError_Success) {
        goto fail;
    }
    /* gather all the lists into an array and reconcile */
    unreconciledList[0] = AttrList2;
    unreconciledList[1] = importedUnreconciledAttrList;
    err = NvSciBufAttrListReconcile(unreconciledList, 2, &reconciledList,
            &newConflictList);
    if (err != NvSciError_Success) {
        goto fail;
    }
    err = NvSciBufAttrListIpcExportReconciled(
        &AttrList1,                  /* array of unreconciled lists to be
exported */
        ipcEndpoint,                 /* valid and opened NvSciIpcEndpoint
intended to send the descriptor through */
        &ListDesc,                   /* The descriptor buffer to be
allocated and filled in */
        &ListDescSize );             /* size of the newly created
buffer */
    if (err != NvSciError_Success) {
        goto fail;
    }
```

Export/Import NvSciBufObj

```
    /* process1 */
    void* objAndList;
    size_t objAndListSize;
    err = NvSciBufIpcExportAttrListAndObj(
        bufObj,                          /* bufObj to be exported
(the reconciled list is inside it) */
        NvSciBufAccessPerm_ReadOnly,     /* permissions we want the
receiver to have */
        ipcEndpoint,                     /* IpcEndpoint via which the
object is to be exported */
        &objAndList,                     /* descriptor of the object
and list to be communicated */
        &objAndListSize);                /* size of the descriptor */
    /* send via Ipc */
    /* process2 */
    void* objAndList;
    size_t objAndListSize;
    err = NvSciBufIpcImportAttrListAndObj(
        module,                          /* NvSciBufModule use to
create original unreconciled lists in the waiter */
        ipcEndpoint,                     /* ipcEndpoint from which the
descriptor was received */
        objAndList,                      /* the desciptor of the buf obj
and associated reconciled attribute list received from the signaler */
        objAndListSize,                  /* size of the descriptor */
        &AttrList1,                      /* the array of original
unreconciled lists prepared in this process */
```

```
        1,                               /* size of the array */
        NvSciBufAccessPerm_ReadOnly,    /* permissions expected by
this process */
        10000U,                          /* timeout in microseconds.
Some primitives might require time to transport all needed resources */
        &bufObj);                        /* buf object generated from
the descriptor */
    /* use the buf object */
    NvSciBufObjFree(bufObj);
```

NvSciBuf API

For information about NvSciBuf API, see Buffer Allocation APIs in the DRIVE OS Linux SDK.

UMD Access

CUDA

CUDA supports the import of an NvSciBufObj into CUDA as CUDA external memory of type NvSciBuf. Once imported, use CUDA API to get a CUDA pointer/array from the imported memory object, which can be passed to CUDA kernels. Applications must query NvSciBufObj for the attributes required to fill descriptors, which are passed as parameters to the import/map APIs.

If the NvSciBuf object imported into CUDA is also mapped by other drivers, then the application must use CUDA external semaphore APIs described here as appropriate barriers to maintain coherence between CUDA and the other drivers.

NvSciBufObj as a CUDA Pointer / Array

The following section describes the steps required to use NvSciBufObj as a CUDA pointer/array.

1. Allocate NvSciBufObj.
   - The application creates NvSciBufAttrList and sets the `NvSciBufGeneralAttrKey_GpuId` *attribute* to specify the ID of the GPU that shares the buffer, along with other attributes.
   - If the same object is used by other UMDs, the corresponding attribute lists must be created and reconciled. The reconciled list must be used to allocate the NvSciBufObj.

   Note: **The attribute list and NvSciBuf objects must be maintained by the application.**

2. Query NvSciBufObj attributes to fill CUDA descriptors.
   - The application must query the allocated NvSciBufObj for required attributes to fill the CUDA external memory descriptor.
3. NvSciBuf object registration with CUDA.

- `cudaImportExternalMemory()` must be used to register the allocated NvSciBuf object with CUDA by filling up cudaExternalMemoryHandleDesc for type cudaExternalMemoryHandleTypeNvSciBuf.
- `cudaDestroyExternalMemory()` API must be used to free the CUDA external memory. CUDA mappings created from external memory must be freed before invoking this API.

4. Getting CUDA pointer/array from imported external memory.
    - `cudaExternalMemoryGetMappedBuffer()` maps a buffer onto an imported memory object and returns a CUDA device pointer. The properties of the buffer must be described in the CUDA ExternalMemory buffer description by querying attributes from NvSciBufObj. The returned pointer device pointer must be freed using cudaFree.
    - `cudaExternalMemoryGetMappedMipmappedArray()` maps a CUDA mipmapped array onto an external object and returns a handle to it. The properties of the buffer must be described in the CUDA ExternalMemory MipmappedArray desc by querying attributes from NvSciBufObj. The returned CUDA mipmapped array must be freed using cudaFreeMipmappedArray.

Note: **All the APIs mentioned in the sections above are CUDA-runtime APIs. Each of them has an equivalent driver API. The syntax and usage of both versions are the same.**

NvSciBuf-CUDA Interop

```
    /*********** Allocate NvSciBuf object ************/
    // Raw Buffer Attributes for CUDA
    NvSciBufType bufType = NvSciBufType_RawBuffer;
    uint64_t rawsize = SIZE;
    uint64_t align = 0;
    bool cpuaccess_flag = true;
    NvSciBufAttrValAccessPerm perm = NvSciBufAccessPerm_ReadWrite;
    uint64_t gpuId[] = {};
    cuDeviceGetUuid(&uuid, dev));
    gpuid[0] = (uint64_t)uuid.bytes;
    // Fill in values
    NvSciBufAttrKeyValuePair rawbuffattrs[] = {
         { NvSciBufGeneralAttrKey_Types, &bufType, sizeof(bufType) },
         { NvSciBufRawBufferAttrKey_Size, &rawsize, sizeof(rawsize) },
         { NvSciBufRawBufferAttrKey_Align, &align, sizeof(align) },
         { NvSciBufGeneralAttrKey_NeedCpuAccess, &cpuaccess_flag,
                sizeof(cpuaccess_flag) },
         { NvSciBufGeneralAttrKey_RequiredPerm, &perm, sizeof(perm) },
         { NvSciBufGeneralAttrKey_GpuId, &gpuid, sizeof(gpuId) },

    };
    // Create list by setting attributes
    err = NvSciBufAttrListSetAttrs(attrListBuffer, rawbuffattrs,
            sizeof(rawbuffattrs)/sizeof(NvSciBufAttrKeyValuePair));
    NvSciBufAttrListCreate(NvSciBufModule, &attrListBuffer);
    // Reconcile And Allocate
    NvSciBufAttrListReconcile(&attrListBuffer, 1, &attrListReconciledBuffer,
&attrListConflictBuffer)
    NvSciBufObjAlloc(attrListReconciledBuffer, &bufferObjRaw);
    /*************** Query NvSciBuf Object **************/
     NvSciBufAttrKeyValuePair bufattrs[] = {
                {NvSciBufRawBufferAttrKey_Size, NULL, 0},
    };
    NvSciBufAttrListGetAttrs(retList, bufattrs, sizeof(bufattrs)/
sizeof(NvSciBufAttrKeyValuePair)));
    ret_size = *(static_cast<const uint64_t*>(bufattrs[0].value));
    /*************** NvSciBuf Registration With CUDA **************/
    // Fill up CUDA_EXTERNAL_MEMORY_HANDLE_DESC
    cudaExternalMemoryHandleDesc memHandleDesc;
    memset(&memHandleDesc, 0, sizeof(memHandleDesc));
    memHandleDesc.type = cudaExternalMemoryHandleTypeNvSciBuf;
    memHandleDesc.handle.nvSciBufObject = bufferObjRaw;
    memHandleDesc.size = ret_size;
    cudaImportExternalMemory(&extMemBuffer, &memHandleDesc);
    /************** Mapping to CUDA ******************************/
    cudaExternalMemoryBufferDesc bufferDesc;
    memset(&bufferDesc, 0, sizeof(bufferDesc));
    bufferDesc.offset = offset = 0;
    bufferDesc.size = ret_size;
    cudaExternalMemoryGetMappedBuffer(&dptr, extMemBuffer, &bufferDesc);
    /************** CUDA Kernel ***********************************/
    // Run CUDA Kernel on dptr
    /*************** Free CUDA mappings *****************************/
    cudaFree();
    cudaDestroyExternalMemory(extMemBuffer);
    /***************** Free NvSciBuf **********************************/
    NvSciBufAttrListFree(attrListBuffer);
    NvSciBufAttrListFree(attrListReconciledBuffer)
    NvSciBufAttrListFree(attrListConflictBuffer);
    NvSciBufObjFree(bufferObjRaw);
```

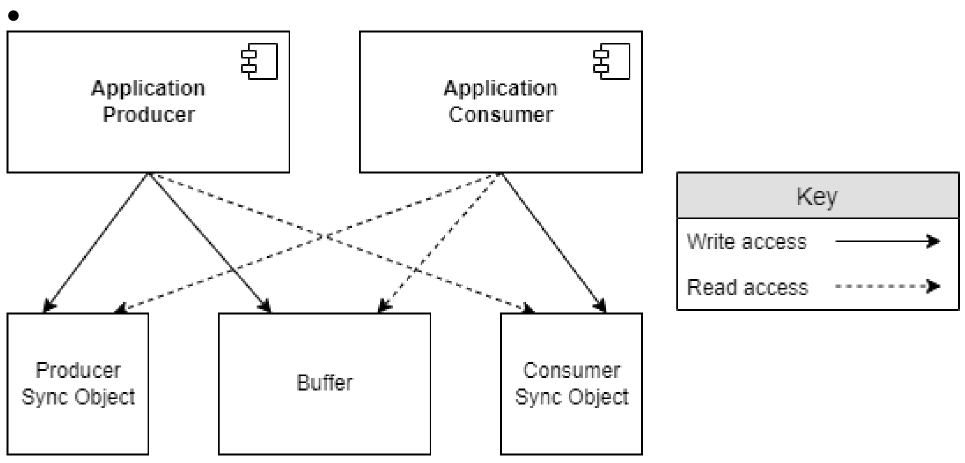
Streaming

Combining the buffer and synchronization functions from the previous chapters allows you to develop applications that stream sequences of data from one rendering component to another, building an efficient processing pipeline. For developers who wish to have complete control over the process, no additional functionality is required.

However, use cases for streaming can become quite complex, making the details difficult to manage. This is particularly true when portions of the pipeline are provided by independent developers who must coordinate the stream management. NVIDIA therefore provides an additional NvSciStream library layered on NvSciBuf and NvSciSync with utilities for constructing streaming application suites.

A Simple Stream

This section illustrates how streaming works with an example application that uses NvSciBuf and NvSciSync to send the images from CUDA producer to CUDA consumer for processing. This application is so simple that there is no need to involve the added NvSciStream layer. It uses a single buffer that CUDA producer can write to and CUDA consumer can read from, and a pair of sync objects. One sync object is for CUDA producer to write and CUDA consumer to read, and the other is for CUDA consumer to write and CUDA producer to read.



Setup

The initial setup for the application is shown below. Much of this should already be familiar from the earlier chapters on buffers and synchronization objects. (For brevity, we assume everything succeeds and omit error checking.)

Simple Stream Setup

```
/* Initialize CUDA (not all steps are shown) */
CUdevice cudaDevice;
cuDeviceGet(&cudaDevice, IGPU);
CUcontext cudaContext;
cuCtxCreate(&cudaContext, CU_CTX_MAP_HOST, dev);
cuCtxPushCurrent(&cudaContext);
CUstream cudaStream;
cuStreamCreate(&cudaStream, CU_STREAM_DEFAULT);

/* Initialize NvSci buffer and sync modules */
NvSciBufModule bufModule;
NvSciBufModuleOpen(&bufModule);
NvSciSyncModule syncModule;
NvSciSyncModuleOpen(&syncModule);

/* Obtain producer buffer requirements */
NvSciBufAttrList producerBufAttrs;
```

```
NvSciBufAttrListCreate(bufModule, &producerBufAttrs);
<Fill in with CUDA raw buffer attributes>

/*
 * Set more buffer attributes using NvSciBufAttrListSetAttrs.
 * Detail skipped.
 */

/* Obtain producer sync requirements */
NvSciSyncAttrList producerWriteSyncAttrs,
                  producerReadSyncAttrs;
NvSciSyncAttrListCreate(syncModule,
                        &producerWriteSyncAttrs);

cuDeviceGetNvSciSyncAttributes(producerWriteSyncAttrs,
                               cudaDevice,
                               CUDA_NVSCISYNC_ATTR_SIGNAL);

NvSciSyncAttrListCreate(syncModule,
                        &producerReadSyncAttrs);

cuDeviceGetNvSciSyncAttributes(producerReadSyncAttrs,
                               cudaDevice,
                               CUDA_NVSCISYNC_ATTR_WAIT);


/* Obtain consumer buffer requirements */
NvSciBufAttrList consumerBufAttrs;
NvSciBufAttrListCreate(bufModule, &consumerBufAttrs);
<Fill in with CUDA raw buffer attributes>

/* Obtain CUDA sync requirements */
NvSciSyncAttrList consumerWriteSyncAttrs,
                  consumerReadSyncAttrs;
NvSciSyncAttrListCreate(syncModule,
                        &consumerWriteSyncAttrs);
cuDeviceGetNvSciSyncAttributes(consumerWriteSyncAttrs,
                               cudaDevice,
                               CUDA_NVSCISYNC_ATTR_SIGNAL);
NvSciSyncAttrListCreate(syncModule,
                        & consumerReadSyncAttrs);
cuDeviceGetNvSciSyncAttributes(consumerReadSyncAttrs,
                               cudaDevice,
                               CUDA_NVSCISYNC_ATTR_WAIT);

/* Combine buffer requirements and allocate buffer */
NvSciBufAttrList allBufAttrs[2], conflictBufAttrs;
NvSciBufAttrList combinedBufAttrs;
allBufAttrs[0] = producerBufAttrs;
allBufAttrs[1] = consumerBufAttrs;
NvSciBufAttrListReconcile(allBufAttrs, 2,
                          &combinedBufAttrs, &conflictBufAttrs);
NvSciBufObj buffer;
NvSciBufObjAlloc(combinedBufAttrs, &buffer);
```

```
/* Combine sync requirements and allocate
   producer to consumer sync object */
NvSciSyncAttrList allSyncAttrs[2], conflictSyncAttrs;
allSyncAttrs[0] = producerWriteSyncAttrs;
allSyncAttrs[1] = consumerReadSyncAttrs;
NvSciSyncAttrList producerToConsumerSyncAttrs;
NvSciSyncAttrListReconcile(allSyncAttrs, 2,
                            &producerToConsumerSyncAttrs,
                            &confictSyncAttrs);
NvSciSyncObj producerToConsumerSync;
NvSciSyncObjAlloc(producerToConsumerSyncAttrs,
                  &producerToConsumerSync);

/* Combine sync requirements and allocate
   consumer to producer sync object */
allSyncAttrs[0] = consumerWriteSyncAttrs;
allSyncAttrs[1] = producerReadSyncAttrs;
NvSciSyncAttrList consumerToProducerSyncAttrs;
NvSciSyncAttrListReconcile(allSyncAttrs, 2,
                            &consumerToProducerSyncAttrs,
                            &confictSyncAttrs);
NvSciSyncObj consumerToProducerSync;
NvSciSyncObjAlloc(consumerToProducerSyncAttrs,
                  &consumerToProducerSync);


/* Map objects on producer side */
cudaExternalMemoryHandleDesc cudaMemHandleDesc;
memset(&cudaMemHandleDesc, 0, sizeof(cudaMemHandleDesc));
cudaMemHandleDesc.type = cudaExternalMemoryHandleTypeNvSciBuf;
cudaMemHandleDesc.handle.nvSciBufObject = buffer;
cudaMemHandleDesc.size = <allocated size>;
cudaImportExternalMemory(&cudaBuffer, &cudaMemHandleDesc);

CUDA_EXTERNAL_SEMAPHORE_HANDLE_DESC cudaSemDec;
CUexternalSemaphore producerToConsumerSem,
                    consumerToProducerSem;
cudaSemDesc.type = CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC;
cudaSemDesc.handle.nvSciSyncObj = (void*)producerToConsumerSync;
cuImportExternalSemaphore(&producerToConsumerSem,  &cudaSemDesc);

cudaSemDesc.type = CU_EXTERNAL_SEMAPHORE_HANDLE_TYPE_NVSCISYNC;
cudaSemDesc.handle.nvSciSyncObj = (void*)consumerToProducerSync;
cuImportExternalSemaphore(&consumerToProducerSem, &cudaSemDesc);

/* Map objects on consumer side similarly*/
```

First, the buffer and sync object requirements are queried from the producer of the stream, and from the consumer. These requirements are combined and used to allocate the objects, which are then mapped into CUDA so that they can be used for processing.

Two sync objects are required instead of one because synchronization is required in both directions. It is important that the consumer does not begin reading from the buffer until the producer is done writing to it. It is equally as important that the producer does not begin writing a new image to the buffer until the consumer is done reading the previous image. Otherwise, it overwrites data that is still in use.

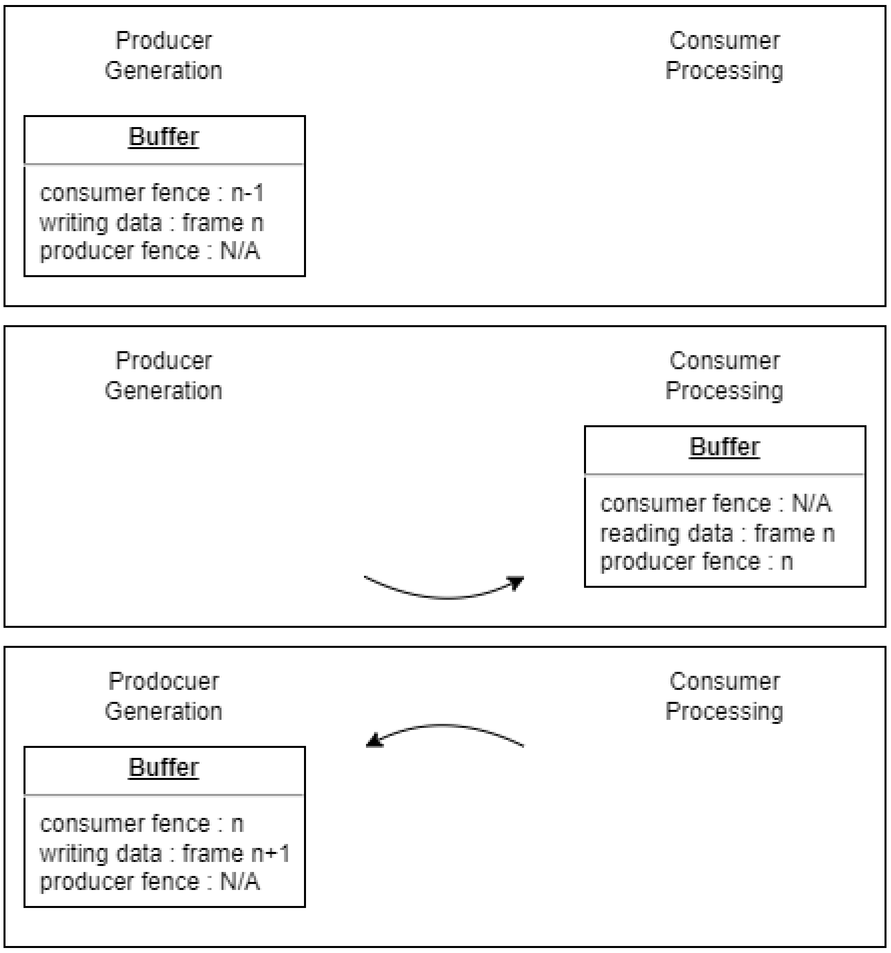
Streaming

Once initialized, the streaming loop appears as follows:

Simple Stream Loop

```
/* Initialize empty fences for each direction*/
NvSciSyncFence producerToConsumerFence = NV_SCI_SYNC_FENCE_INITIALIZER;
NvSciSyncFence consumerToProducerFence = NV_SCI_SYNC_FENCE_INITIALIZER;

/* Main rendering loop */
while (!done) {

    /* Instruct producer pipeline to wait for the fence from consumer */
    CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS cudaWaitParams;
    cudaWaitParams.params.nvSciSync.fence = (void*)&consumerToProducerFence;
    cudaWaitParams.flags = 0;
    cudaWaitExternalSemaphoresAsync(&consumerToProducerSem,
                                    &cudaWaitParams, 1, cudaStream);

    /* Producer Generates the image */
    cudaSomeProcessingOperation(..., cudaBuffer, ...);

    /* Generate a fence when processing finishes */
    CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS signalParams;
    signalParams.params.nvSciSync.fence = (void*)&producerToConsumerFence;
    signalParams.flags = 0;
    cudaSignalExternalSemaphoresAsync(&producerToConsumerSem,
                                      &signalParams, 1, cudaStream);


    /* Instruct consumer pipeline to wait for fence from producer */
    CUDA_EXTERNAL_SEMAPHORE_WAIT_PARAMS cudaWaitParams;
    cudaWaitParams.params.nvSciSync.fence = void*)& producerToConsumerFence;
    cudaWaitParams.flags = 0;
    cudaWaitExternalSemaphoresAsync(&producerToConsumerSem,
                                    &cudaWaitParams, 1, cudaStream);

    /* Process the frame in CUDA */
    cudaSomeProcessingOperation(..., cudaBuffer, ...);

    /* Generate a fence when processing finishes */
    CUDA_EXTERNAL_SEMAPHORE_SIGNAL_PARAMS cudaSignalParams;
    cudaSignalParams.params.nvSciSync.fence = void*)& consumerToProducerFence;
    cudaSignalParams.flags = 0;
    cudaSignalExternalSemaphoresAsync(&consumerToProducerSem,
                                      &cudaSignalParams, 1, cudaStream);
}
```

For each frame of data, the application instructs the producer to write the image to the buffer, and then issues a fence that indicates when writing finishes. The consumer is instructed to wait for that fence before it proceeds with any subsequent operations, then the commands to process the frame are issued to the consumer. Lastly, the consumer is told to generate a fence of its own, which indicates when all its operations finish. This fence is fed back to the producer, which waits for it before starting to write the next image to the buffer. "Ownership" of the buffer cycles back and forth between producer and consumer.

- 

-

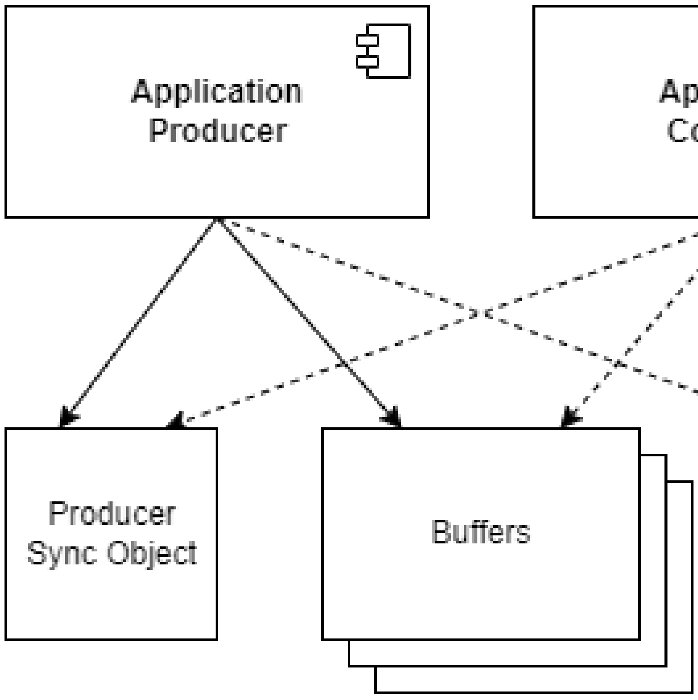
More Complex Streams

All the basic steps in the previous example to set up buffers and sync objects, and to generate and wait for fences must be performed for all streaming applications, regardless of whether they use NvSciStream. For a simple situation like that, there is no need for anything more.

What NvSciStream provides is the ability to manage more complex cases, such as cycling between multiple buffers, streaming to multiple consumers at once, and streaming between applications. It also provides a uniform set of interfaces for setting up streams for which independent developers can design modular producers and consumers without needing to directly coordinate and then plug their products together. This section describes some of those use cases Adding NvSciStreams for these use cases relieves the developer of many burdensome details.

Multiple Buffers

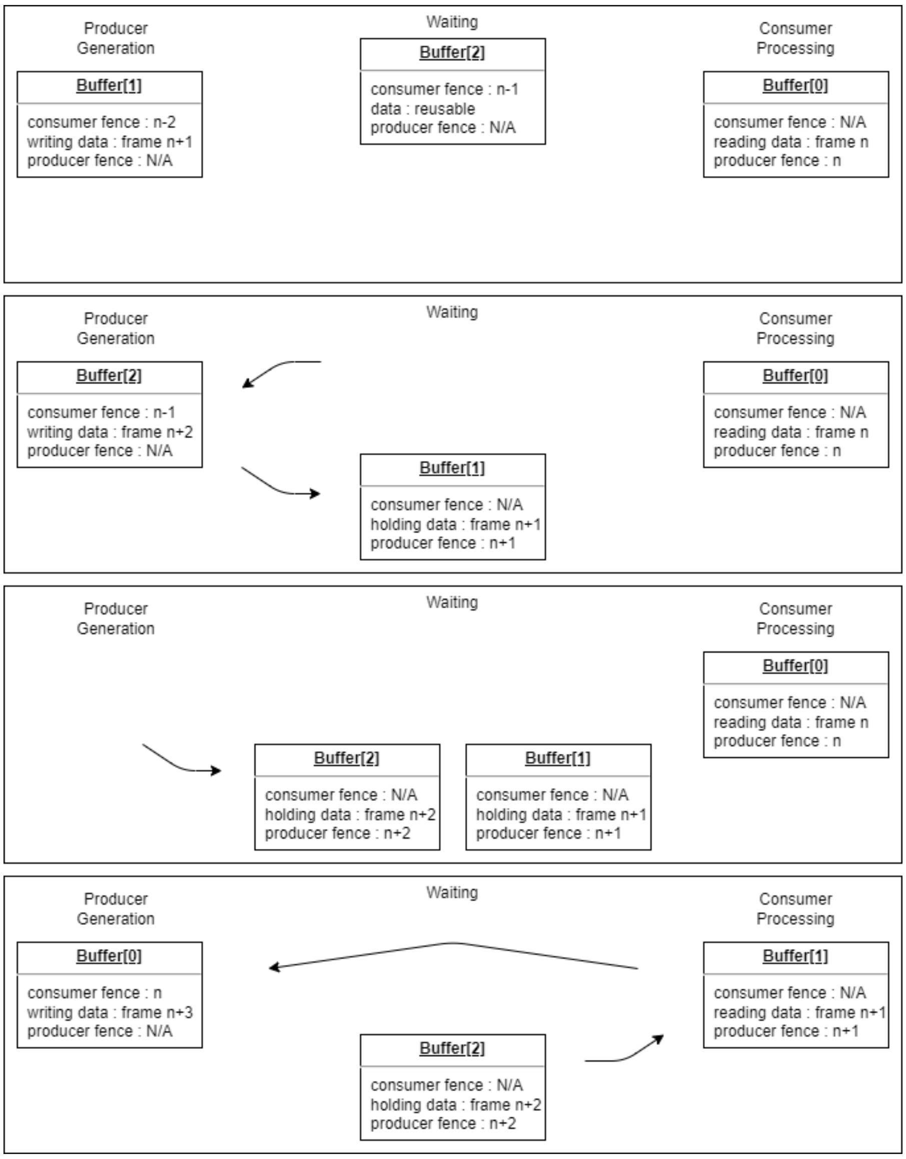
If the producer and the consumer use different portions of the NVIDIA hardware. The producer hardware is idle during the consumer hardware processing, and the consumer hardware is idle during the producer hardware processing. The pipeline can be made more efficient by allocating one (1) or more additional buffers and cycling between them, creating a queue of frames. This way, the consumer can process one image while the producer prepares the next one. No additional sync objects are needed, but it is necessary to generate a fence for each frame, and you must keep track of which fence is associated with the current contents of each buffer. Once a stream has multiple buffers available, there are several different possible modes of operation to consider.



FIFO Mode

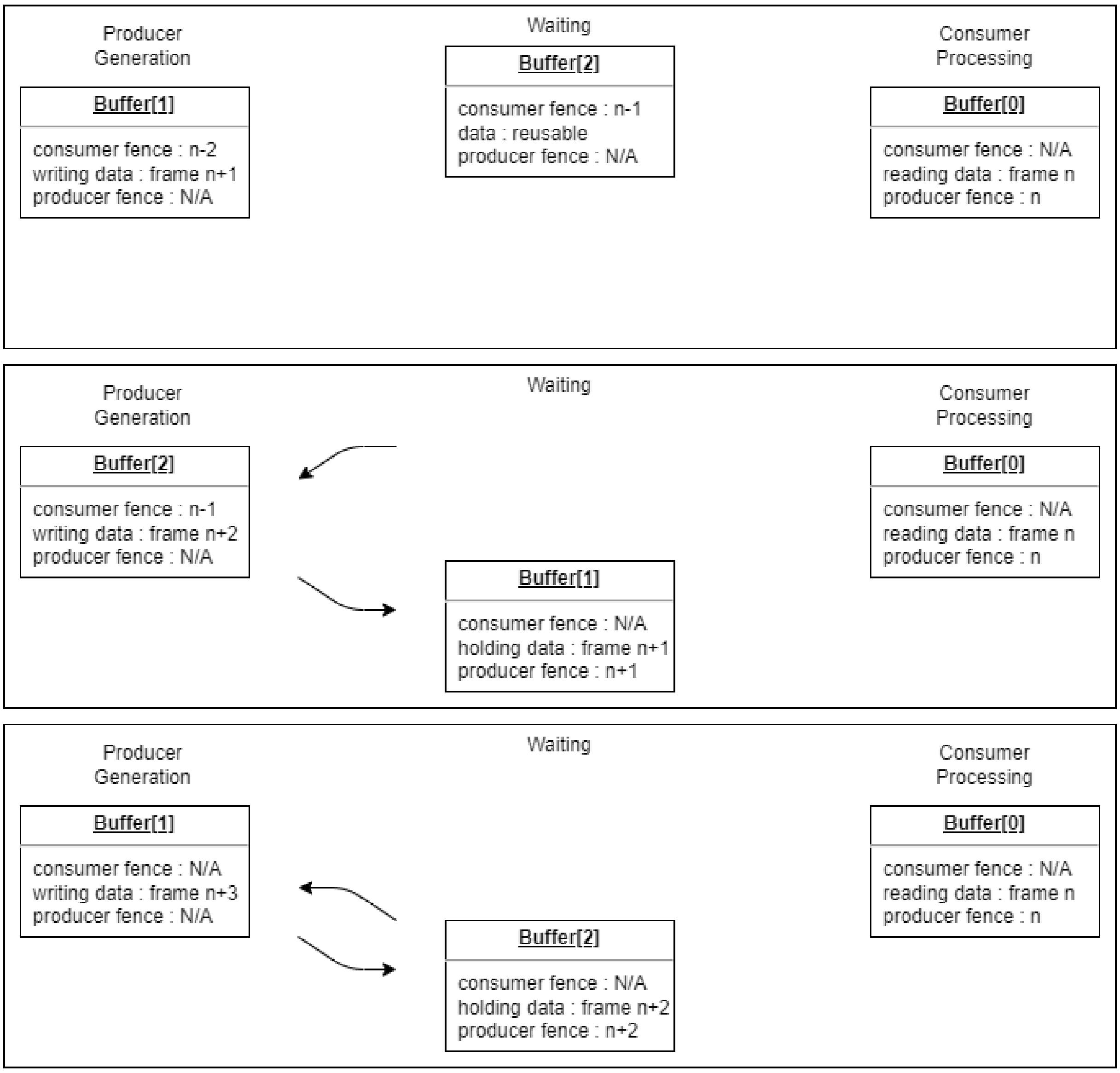
If the use case requires that all data in the sequence be processed, the stream application operates in FIFO mode. When the producer fills a buffer with new data, it must wait for the consumer to process it. If the consumer requires more time than the producer, the producer is slowed to the consumer's speed to wait for buffers to become available for reuse.

-

- 

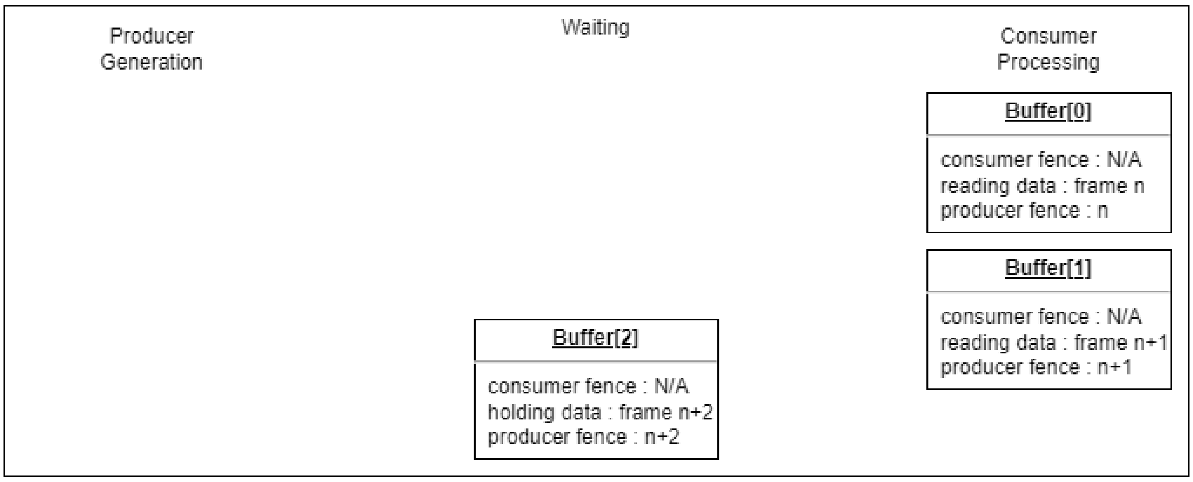
Mailbox Mode

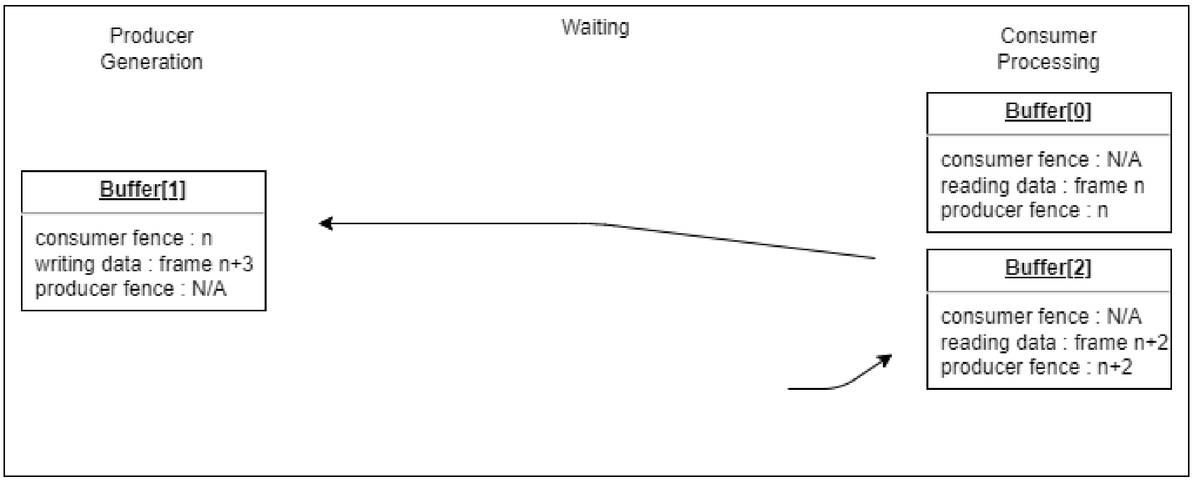
In other use cases, it is more important that the consumer always have the most recent input available. In this case, the stream application operates in mailbox mode. If the consumer has not started processing a previous frame when a new one becomes available, it is skipped, and its buffer immediately returned to the producer for reuse. This means that the order in which the producer and consumer cycle through buffers is subject to change at any time, and the streaming system must manage this.



Multiple Acquired Frames

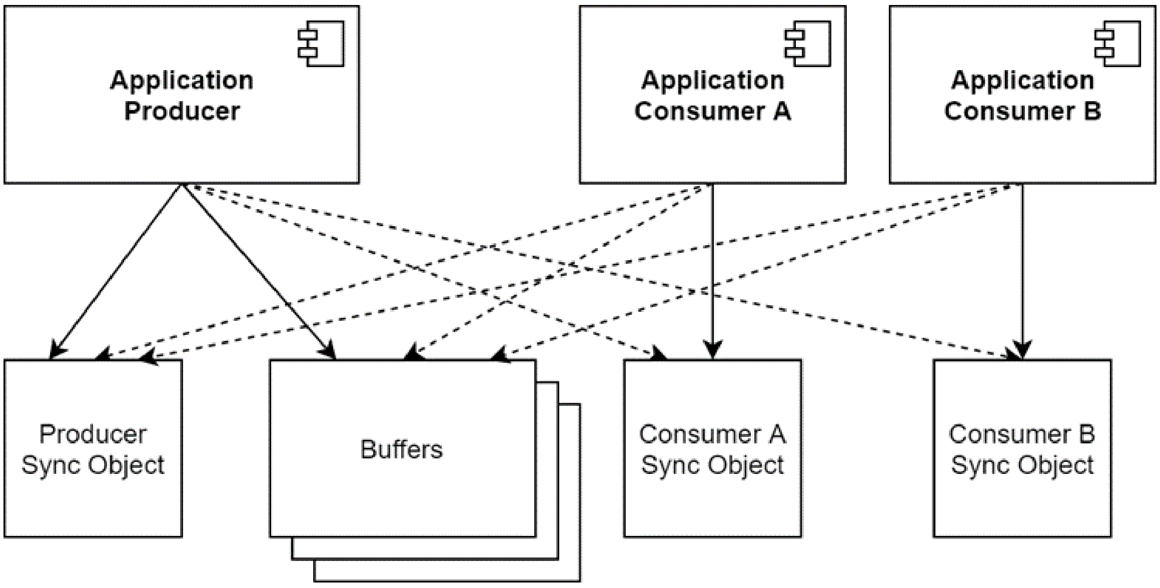
Some processing algorithms must compare data from multiple frames in a sequence. In this case, the consumer may hold multiple buffers at once. These buffers may be released for reuse in an order other than they arrive. The streaming system must manage this.





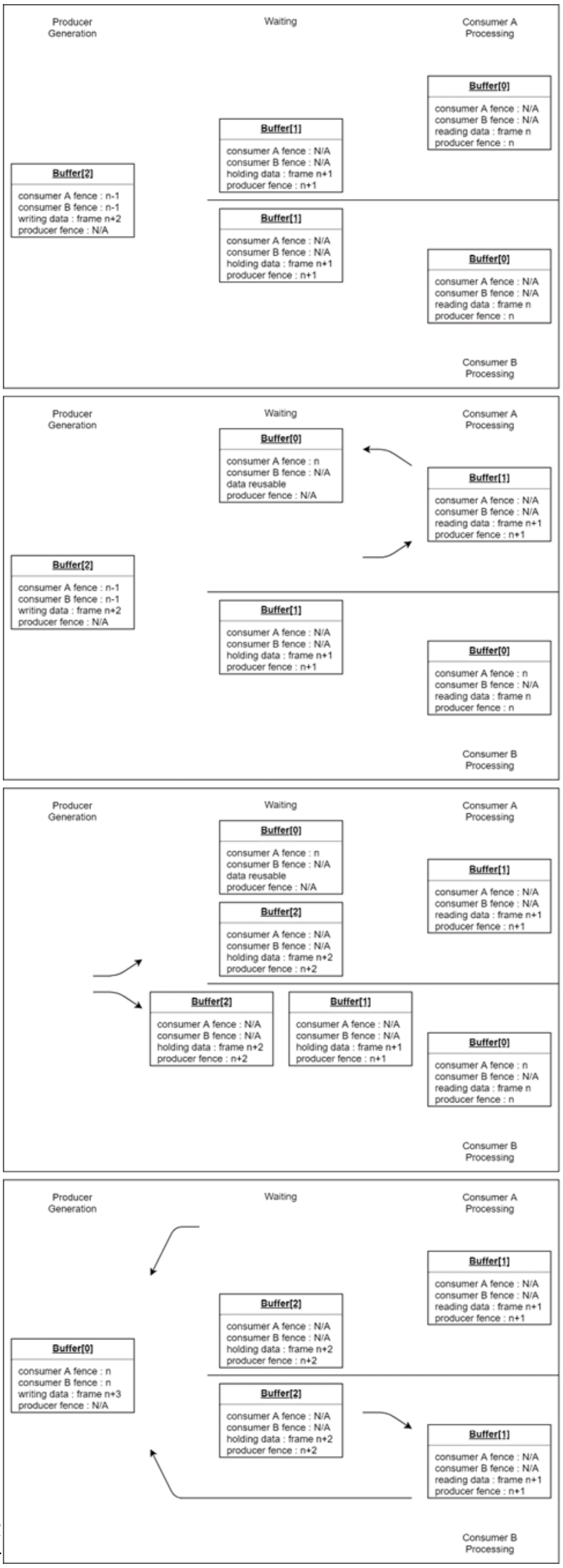
Multiple Consumers

In some cases, the output of a single producer must be sent to multiple consumers. This means that during initialization, the requirements from all of the consumers must be gathered, and sync objects from all of the consumers must be mapped into the producer.



•

During streaming, the system must track each buffer's state with regards to all the consumers and wait for fences from all of them to complete before the producer can reuse it. This can be further complicated if one consumer requires FIFO behavior and another requires mailbox. They might return buffers in different orders.


Producer Generation
Waiting
Consumer A Processing
Buffer[0]
consumer A fence : N/A
consumer B fence : N/A
reading data : frame n
producer fence : n
Buffer[1]
consumer A fence : N/A
consumer B fence : N/A
holding data : frame n+1
producer fence : n+1
Buffer[2]
consumer A fence : n-1
consumer B fence : n-1
writing data : frame n+2
producer fence : N/A
Buffer[1]
consumer A fence : N/A
consumer B fence : N/A
holding data : frame n+1
producer fence : n+1
Buffer[0]
consumer A fence : N/A
consumer B fence : N/A
reading data : frame n
producer fence : n
Consumer B Processing
Producer Generation
Waiting
Consumer A Processing
Buffer[0]
consumer A fence : n
consumer B fence : N/A
data reusable
producer fence : N/A
Buffer[1]
consumer A fence : N/A
consumer B fence : N/A
reading data : frame n+1
producer fence : n+1
Buffer[2]
consumer A fence : n-1
consumer B fence : n-1
writing data : frame n+2
producer fence : N/A
Buffer[1]
consumer A fence : N/A
consumer B fence : N/A
holding data : frame n+1
producer fence : n+1
Buffer[0]
consumer A fence : n
consumer B fence : N/A
reading data : frame n
producer fence : n
Consumer B Processing
Producer Generation
Waiting
Consumer A Processing
Buffer[0]
consumer A fence : n
consumer B fence : N/A
data reusable
producer fence : N/A
Buffer[1]
consumer A fence : N/A
consumer B fence : N/A
reading data : frame n+1
producer fence : n+1
Buffer[2]
consumer A fence : N/A
consumer B fence : N/A
holding data : frame n+2
producer fence : n+2
Buffer[2]
consumer A fence : N/A
consumer B fence : N/A
holding data : frame n+2
producer fence : n+2
Buffer[1]
consumer A fence : N/A
consumer B fence : N/A
holding data : frame n+1
producer fence : n+1
Buffer[0]
consumer A fence : n
consumer B fence : N/A
reading data : frame n
producer fence : n
Consumer B Processing
Producer Generation
Waiting
Consumer A Processing
Buffer[1]
consumer A fence : N/A
consumer B fence : N/A
reading data : frame n+1
producer fence : n+1
Buffer[2]
consumer A fence : N/A
consumer B fence : N/A
holding data : frame n+2
producer fence : n+2
Buffer[0]
consumer A fence : n
consumer B fence : n
writing data : frame n+3
producer fence : N/A
Buffer[2]
consumer A fence : N/A
consumer B fence : N/A
holding data : frame n+2
producer fence : n+2
Buffer[1]
consumer A fence : N/A
consumer B fence : N/A
reading data : frame n+1
producer fence : n+1
Consumer B Processing

Limit Acquired Frames

In the multiple consumers case, there is a security and robustness concern that if one of the consumer applications either maliciously or through some bug stops returning packets for reuse, it will bring the producer and all other consumers to a halt because they will run out of space for the new data payloads. One way to mitigate this risk is to place an optional cap on the number of packets that will be allowed to be sent downstream to the unreliable application.

During streaming, if a new packet is available but an untrusted application is already at its cap, the packet will immediately be returned upstream without the consumer ever seeing it.

Cross-Application

Having producers and consumers in separate applications requires inter-process communication during every step of the process. At setup, all the requirements must be transmitted between the endpoints, and then the allocated objects must be shared between the processes. During streaming, every time the producer generates a frame, a message letting the consumer know it is ready, which includes the fence, must be sent to the other side. When the consumer is done reading from the frame, a similar message must be sent back in the other direction. Each application must be prepared to read and process these messages so streaming can occur in a timely fashion.

Forced Sychronization of Payloads

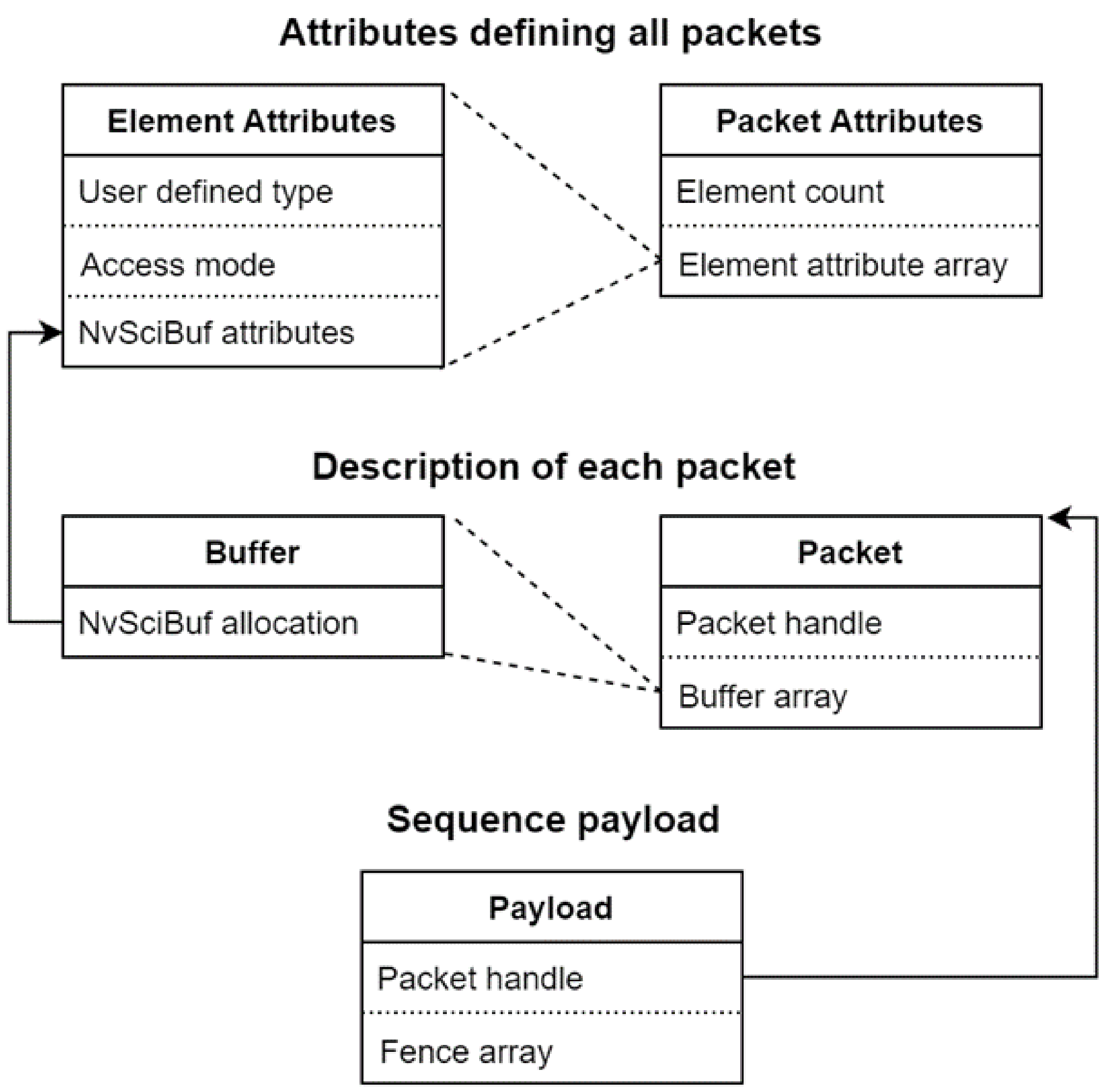
Certain sections in a stream may require the data passing through be synchronous, meaning when the payloads arrive in those sections all producers or consumers have done writing to or reading from the buffer data in the payloads. Before reaching those sections in the stream, payloads that are accompanied with fences have to be blocked until the fences expire.

Data Packets

In the previous examples, only 2D buffers containing single frames of image data are considered. For many use cases, that is all that is required. However, NvSciStream supports more arbitrary data. In the sections below, "buffers" are generalized to "packets", and "frames" to "payloads".

A packet is a set of indexed buffers that contains images, tensors, metadata, or other information. These buffers are referred to as the "elements" of a packet. Each stream may use one or more packets. For a given stream, all packets have the same number of elements, and the ith element of all packets is allocated with the same NvSciBuf attributes. As a result, all packets have uniform memory requirements and signatures. Producers and consumers send and receive entire packets at a time, along with associated fences to indicate when the payloads they contain are ready.

In addition to the NvSciBuf attributes used to allocate them, the elements of each packet also have a type and NvSciSync attributes associated with them, which help applications coordinate how the elements are used. These are described in the next sections.



Element Type

In a complex modular application suite, a given producer may know how to produce many different types of data, and the various consumers may only require a subset of them. NvSciStream provides mechanisms to help applications negotiate the data that is required for a given stream. The top-level system integrator must define a set of integer values to associate with each type of data that the application suite supports. Any non-zero value may be used for these types, and developers are advised to plan for future types and backwards compatibility.

As an example, the following table is a subset of the types that are available in an automotive system. These may not all be supplied by a single producer. For instance, one producer might be responsible for front-facing sensors, and another for rear-facing sensors. Different consumers make use of different sets of sensors to generate their output. The types listed in the table below all correspond to raw inputs, but a pipe-lined application may have a series of producers and consumers with additional types to represent intermediate processed data. The following table describes the sample data packet types:

Data Type	Assigned Enum
Front left camera	0x1101
Front-center camera	0x1102
Front-right camera	0x1103
Rear-left camera	0x1201
Rear-center camera	0x1202
Rear-right camera	0x1203
Front radar	0x2110
Front lidar	0x2120
Rear radar	0x2210
Rear lidar	0x2220
GPS	0x3010
IMU	0x3020

At initialization, a producer declares all the types of data it can provide. Consumers indicate the types of data they are interested in. The part of the application responsible for allocating the buffers takes this information and collates it to determine the packet layout and passes the packet attributes back to the producer and consumers. The producer checks this layout to see what elements it needs to provide. The consumers check it to determine where in the packets to find the elements they care about. Any element that a given consumer does not require can be ignored.

In simple applications that only deal with a single type of data such as images, producers and consumers can simply specify a value of 1 for the type. These type exchange mechanisms are only intended to aid integration of larger suites.

Element Mode

Payload data generated and processed by NVIDIA hardware is usually written and read asynchronously and requires waiting for a fence before it can be accessed. But in some use cases, auxiliary data may be generated synchronously by the CPU and must be read before the commands to process the rest of the data are issued. An example is a camera producer that generates images asynchronously, but also includes a synchronously generated metadata field that contains the camera's focal length, exposure time, and other settings. A consumer synchronously reads in the metadata first and uses the values it contains when issuing the commands to asynchronously process the image.

The endpoint that requires an element to be synchronous must provide the `NULL waiter sync` attribute to the opposing endpoint for that element.

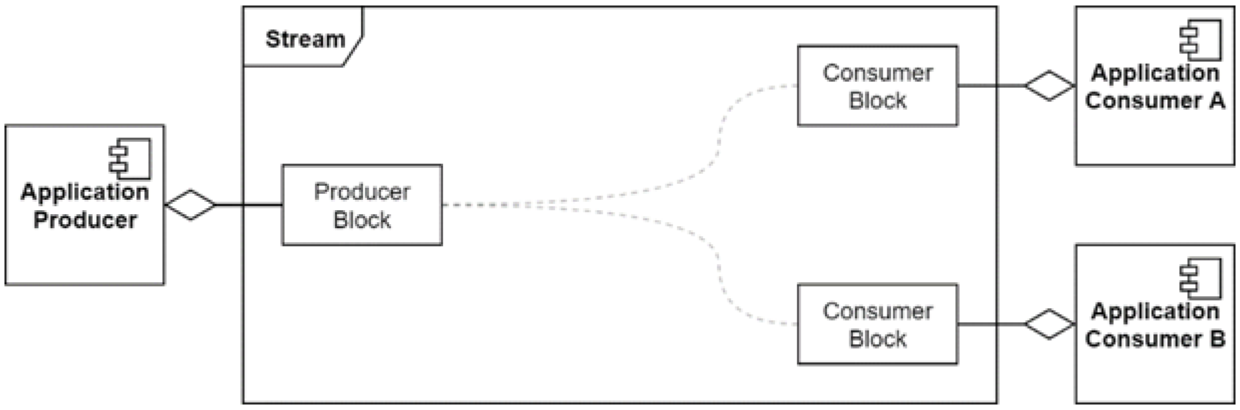
Building Block Model

NvSciStream is designed to make implementing complex stream use cases easier by providing a uniform set of interfaces for exchanging requirements and sending and receiving frames. At the top level, an application suite determines the overall structure of the stream. At the lower application component level, each producer and consumer performs its own function, without needing to know the details of the full stream. They can be developed independently in a modular fashion and plugged together as needed.

In order to support this wide variety of use cases, NvSciStream itself takes a modular approach. It defines a set of building blocks, each with a specific purpose, which applications create and connect together to suit their needs. This section describes the available building blocks and their behavior.

Endpoints

Each stream begins with a producer block and ends with one or more consumer blocks. These are referred to as the "endpoints" of the stream, with the producer being the "upstream" end and the consumers being the "downstream" end. For each stream endpoint, there is an application component responsible for interacting with it. The endpoints and their corresponding application components may all reside in a single process or be distributed across multiple processes, partitions, or systems.



Producer

A producer block provides the following interactions with the application producer component:

- During setup phase:
  - Synchronization:
    - Accepts producer synchronization requirements.
    - Reports consumer synchronization requirements.
    - Accepts producer allocated sync objects.
    - Reports consumer allocated sync objects.
  - Buffers
    - Accepts list of element attributes producer can generate.
    - Reports consolidated packet attribute list.
    - Reports all allocated packets.
- During streaming phase:
  - Signals availability of packets for reuse.
  - Retrieves next packet to reuse with fence to wait for before writing.
  - Accepts filled packet with fence to wait for before consumer should read.

- In non-safety builds, during streaming phase:
  - Accepts limited changes to requested element attributes (e.g., to modify the image size).
  - Reports updated packet attributes.
  - Reports removal of buffers allocated for old attributes and addition of new ones.

Note:	**Reallocation of buffers in non-safety builds are not supported in this release.**

Consumer

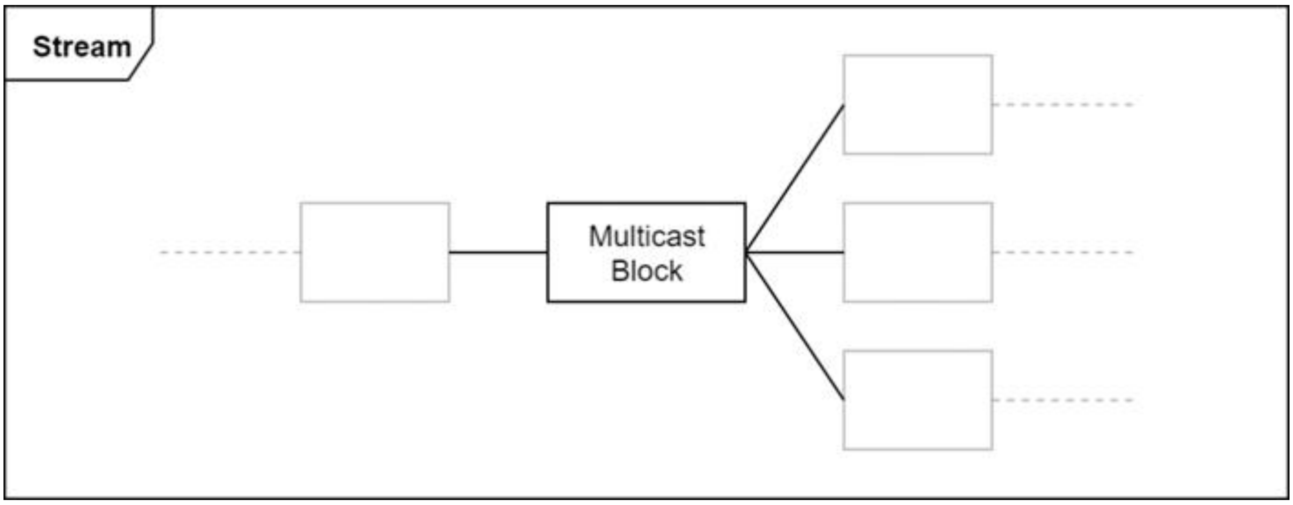
A consumer block provides the following interactions with its corresponding application consumer component:

- During setup phase:
  - Synchronization:
    - Accepts consumer synchronization requirements.
    - Reports producer synchronization requirements.
    - Accepts consumer allocated sync objects.
    - Reports producer allocated sync objects.
  - Buffers:
    - Accepts list of element attributes consumer requires.
    - Reports consolidated packet attribute list.
    - Reports all allocated packets.
- During streaming phase:
  - Signals availability of packets for reading.
  - Retrieves next payload with fence to wait for before reading.
  - Accepts packet, which can be reused with fence to wait for before the produce writes new data.
- In non-safety builds, during streaming phase:
  - Reports updated packet attributes.
  - Reports removal of buffers allocated for old attributes and addition of new ones.

Note:	**Reallocation of buffers in non-safety builds are not supported in this release.**

Multicast

When a stream has more than one consumer, a multicast block is used to connect the separate pipelines. This block distributes the producer's resources and actions to the consumers, who are unaware that each other exist. It combines the consumers' resources and actions, making them appear to the producer as a single virtual consumer. The block does not directly provide any mechanisms to safeguard any of its consumers against faulty or malicious behavior in its other consumers. Additional blocks can be used to isolate individual consumer pipelines from others when there are safety or security concerns.



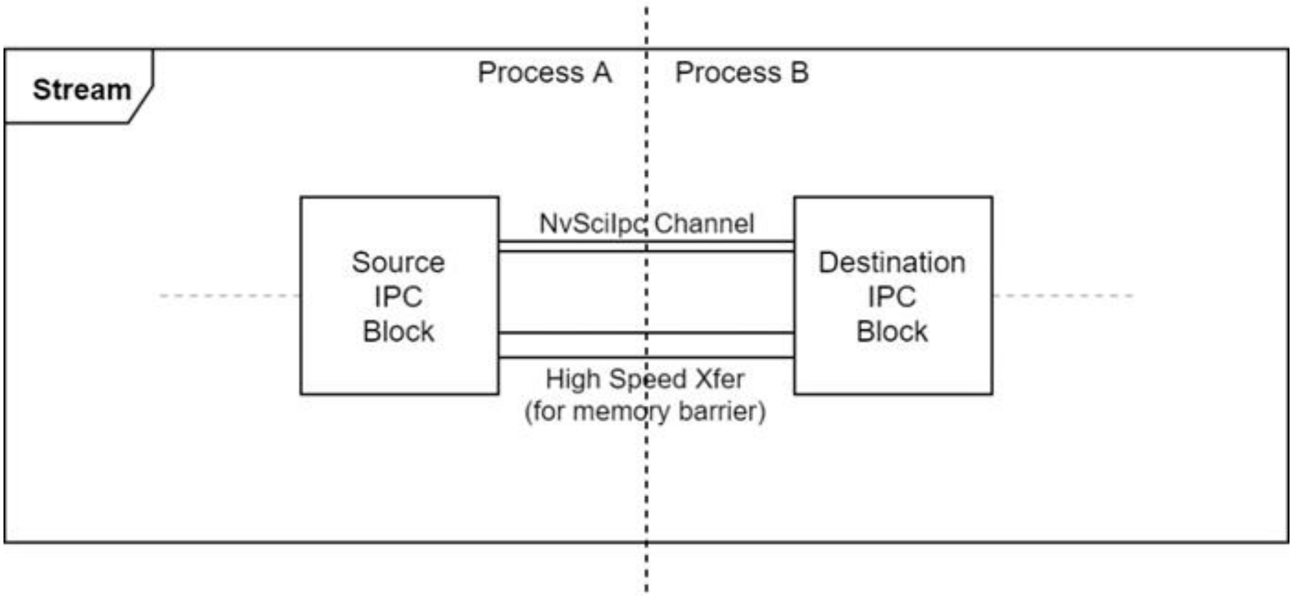
Once created and connected, no direct interaction by the application with a multicast block is required. It automatically performs the following operations as events arrive from up and downstream:

- During setup phase:
  - Synchronization:
    - Passes producer synchronization requirements to all consumers.
    - Combines consumer synchronization requirements and passes to producer.
    - Passes producer allocated sync objects to all consumers.
    - Passes sync objects allocated by all consumers to producer.
  - Buffers:
    - Combines lists of element attributes from all consumers into a single list and passes upstream.
    - Passes consolidated packet attribute list to all consumers.
    - Passes allocated packets to all consumers.
- During streaming phase:
  - Passes available payloads to all consumers.
  - Tracks packets returned for reuse by the consumers and returns them to the producer when all consumers are done with them. All fences are combined into one list.
- In non-safety builds, during streaming phase:
  - Distributes changes in packet attributes and buffers to all consumers.

Note: **Reallocation of buffers in non-safety builds are not supported in this release.**

IPC

When the endpoints of a stream reside in separate processes, IPC blocks are used to bridge the gaps. They are created in pairs, with a source IPC block in the upstream process and a destination IPC block in the downstream process. The communication must first be established by the application using NvSciIpc, and then the channel endpoints are passed to the two IPC blocks. They take ownership of the channel and use it to coordinate the exchange of requirements and resources and signal the availability of packets.



There are two types of IPC pairs available, depending on whether the upstream and downstream portions share memory.

Memory Sharing IPC

When the two halves of the stream access the same physical memory, memory sharing IPC blocks can be used. These coordinate the sharing of resources between the two ends but do not need to access the payload data. They perform the following actions when events arrive from up and downstream:

- During setup phase:
  - Synchronization:
    - Exports synchronization requirements from the producer and consumer(s) and imports them on the other side.
    - Exports sync objects from the producer and consumer(s) and imports them on the other side.

- Buffers:
  - Exports consumer element attributes from downstream and imports them upstream.
  - Exports consolidated packet attribute list and packets from upstream and imports them downstream.

- During streaming phase:
  - Source IPC block signals availability of new payloads to destination block, passing the fence.
  - Destination IPC block signals availability of packets for reuse to source block, passing the fence.
- In non-safety builds, during streaming phase:
  - Signals changes in packet attributes and buffers from source to destination.

Note:	**Reallocation of buffers in non-safety builds are not supported in this release.**

Memory Boundary IPC

When the two processes do not share memory, memory boundary IPC blocks must be used. Each half of the stream must provide a separate set of packets. When new payloads arrive, the source block transmits all the data to the destination block, where it is copied into a new packet. Auxiliary communication channels may be set up for this purpose. Once transmission is done, the original packet is returned upstream for reuse, without waiting for the consumer to finish reading the data, since it accesses a different set of buffers.

These IPC blocks can also be used to create virtual memory boundaries between portions of a stream. If one consumer operates at a lower level of safety and/or security than the rest of the stream, then even if it can share memory with the rest of the stream, it may not be desirable to do so. Requiring this consumer to use its own set of buffers ensures that if it fails, it will not prevent the rest of the stream from continuing, and if it falls prey to a security issue, it won't be able to modify the buffer data seen by the other consumers.

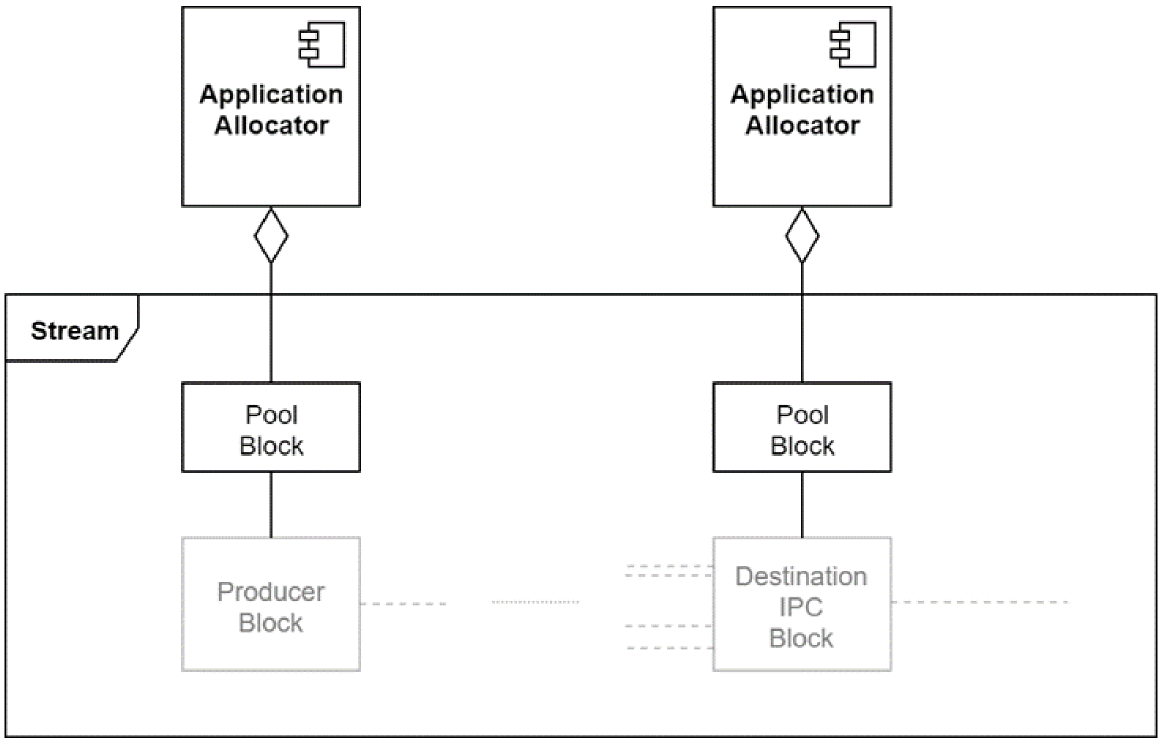
- During setup phase:
  - Synchronization:
    - Synchronization objects are not exchanged across the memory boundary.
    - Source and destination blocks provide their synchronization requirements for the producer and consumer, respectively.
    - If necessary, creates sync objects to be used to coordinate the data copy, and passes to the local endpoint.
    - Accesses sync objects from the local endpoint to coordinate data copy.
  - Buffers:
    - Exports a subset of the consumer element attributes from downstream and imports them upstream, replacing attributes related to memory access with those needed for the data transfer mechanism to access the memory.

- Exports a subset of the consolidated packet attribute list from upstream and imports them downstream, again replacing attributes related to memory access with those needed for the copy engine.
- On destination side, receives and maps buffers used for copy from downstream.

- During streaming phase:
  - Source IPC block transmits payload data to the destination block.
  - Destination IPC block reads the payload data into an available packet and passes it downstream.
- In non-safety builds, during streaming phase:
  - Signals changes in packet attributes and buffers from source to destination.

Pool

Pool blocks are used to introduce packets to the stream and track packets available for reuse. All streams must have at least one pool attached to the producer block. If a stream contains memory boundaries, then additional pools are needed for each section of the stream that uses its own set of packets, attached to the IPC block. For each pool block, there is an application component responsible for deciding the packet layout based on the requirements and allocating the buffers.



NvSciStream supports two kinds of pools: static and dynamic.

Static Pool

Static pools provide a set of packets that remain fixed for the life of the stream. The number of buffers must be specified at creation time, and the buffers are added during the setup phase. A static pool provides the following interactions with the application component that manages it:

- During setup phase:
  - Buffers:
    - Reports list of element attributes producer can generate.
    - Reports list of element attributes consumers require.
    - Accepts consolidated packet attribute list, and sends to producer and consumers.
      - Accepts allocated packets and sends to producer and consumers.
- During streaming phase:
  - Receives and queues packets returned to the producer for reuse.
  - Provides an available packet to the producer or IPC block it supports when requested.

Dynamic Pool

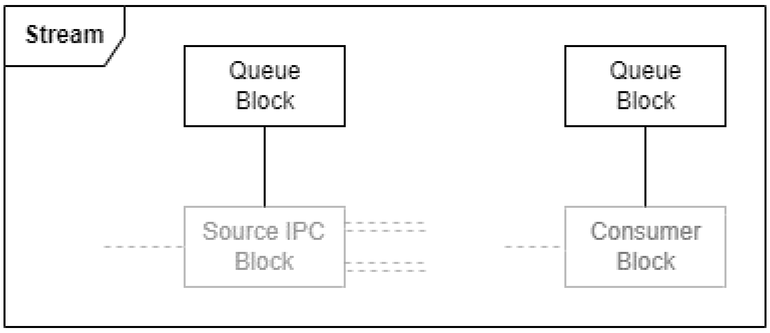
Dynamic pools are only available in non-safety builds. They allow buffers to be added and removed at any time. This supports use cases where the producer may need to change some of the buffer attributes, such as the size or pixel format of the data. Video playback is a typical example where such changes may occur. A dynamic pool provides all the interactions of a static pool, plus the following:

- In non-safety builds, during streaming phase:
  - Reports requested element attribute changes from the producer.
  - Accepts updated element attributes and sends it to the producer and consumers.
  - Accepts instructions to remove packets from the stream and sends it to the producer and consumers.
  - Accepts new allocated packets and sends it to the producer and consumers.

Note: **Dynamic blocks are not supported in this release.**

Queue

Queue blocks attached to consumer blocks keep track of payloads waiting to be acquired by the consumer. Each consumer block must have an attached queue block to manage available packets. Queue blocks attached to memory boundary Source IPC blocks keep track of payloads waiting to be copied across memory boundary. Each memory boundary Source IPC block must have an attached queue block to manage available packets.



NvSciStream supports two types of queue blocks: FIFOs and mailboxes. A FIFO block is used when all data must be processed. Payloads always wait in FIFO until the consumer acquires them. Mailboxes are used when the consumer always acts on the most recent data. If a new payload arrives in the mailbox when one is already waiting, the previous one is skipped and immediately returned to the producer for reuse.

Limiter

A limiter block places a cap on the number of packets allowed to be sent downstream at any given time. It is usually inserted between a Multicast block and a Consumer block (and in general before any IPC blocks transmitting to the consumer).

If a new packet arrives and the current number of downstream packets is at the capacity, the packet is returned upstream immediately without reaching the consumer. This is similar to frame-skipping that can be invoked by a Mailbox queue block attached to the consumer. However, in this case the skipping can occur in the producer application rather than in the consumer. When a consumer at capacity returns a packet for reuse it can then receive a new packet.

A Limiter block does not interact with any other Limiter blocks used with other consumers. Each independently imposes its own limit on what is downstream and not affected by the operations of any other limiter.

In addition, when using a mailbox queue, the consumer application must be capable of dealing with skipped frames. Typical consumers may not be aware of skipped frames and operate on the frame they receive without regard to previous frames. Where consumers need to know that a frame is missed the application suite must include a sequence number, timestamp, or other identifying information in the packet data.

Since faulty consumers may take some packets out of service, the application may need to allocate additional total packets to ensure that there are enough left for the producer and remaining consumers to use if one consumer reaches its cap and never returns any packets again.

ReturnSync

A ReturnSync block addresses security and robustness with multicast streams. A ReturnSync block can be inserted anywhere in the stream between the Producer and Consumer blocks, but its primary intent is to be inserted between a Multicast block and a Consumer block, and downstream of a Limiter block for that consumer. That is, if a producer application was concerned about one or more consumer applications generating unreliable fences, they can choose to add ReturnSync blocks to safeguard against them. This will isolate any packets with bad fences in the branch for an untrusted consumer, preventing the producer from getting stuck waiting for them.

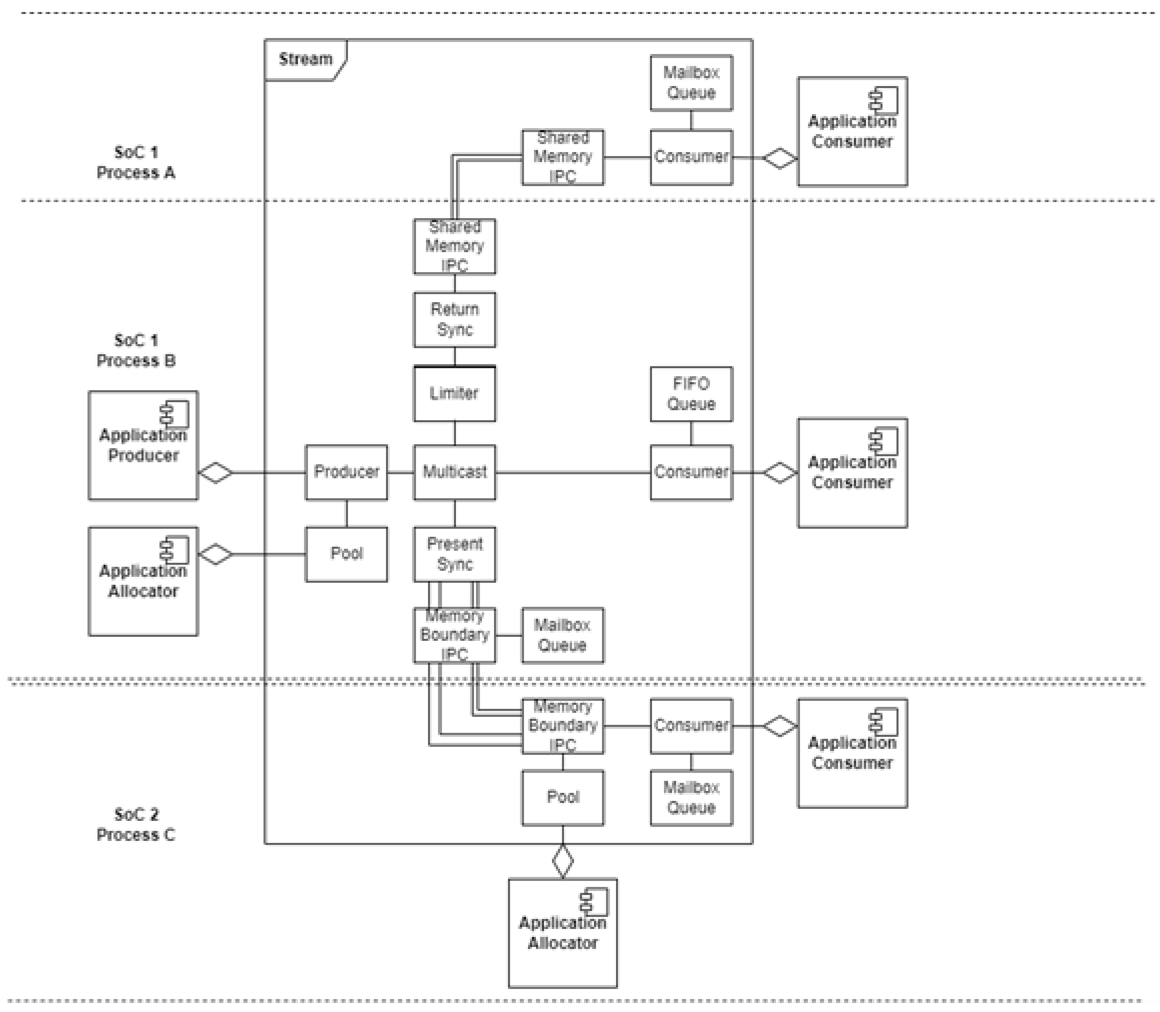
A ReturnSync block waits for fences for each received packet from downstream before sending it upstream. If fences for the received packet are already expired or empty, then it returns the packet upstream immediately. If not, it places the packet in the queue then trigger a thread spawned by this block to manage the fence waiting before sending the packet upstream. This type of block is useful if there are blocks upstream that require synchronous access of packet data.PresentSync

For symmetry, a PresentSync block is added, which does the waiting for fences for each received packet from upstream before sending it downstream. The block spawns a thread which waits for fences in the order of the packets received and send them downstream once fences are reached.This type of block is useful if there are blocks downstream that require synchronous access of packet data.

For example, a PresentSync block can be inserted between the Producer and the memory boundary IPC block, to keep packets from being queued for sending until their contents are ready, primarily when the memory boundary IPC block uses a mailbox queue.

Example

The following diagram shows a complete stream. It has three (3) consumers. One resides in the same process as the producer, the second resides in another process on the same system and shares memory with the producer process, and the third resides on another system and uses its own set of packets. The first uses a FIFO queue and the other two use mailbox queues. The producer application is concerned about the consumer in another process generating unreliable fences, it adds a return sync block to safeguard against it and places a cap on the number of packets held by the consumer in another processthis consumer using the limiter block. It also chooses to add a present sync block before the memory boundary source IPC block, which uses a mailbox queue, to keep packets from being queued for sending until their contents are ready.



Stream Creation

The modular nature of NvSciStream allows producer and consumer components, or entire applications, to be developed independently by different providers, but some planning is required to ensure proper inter-operation. System integrators must make several decisions before assembling a stream.

- Determine which consumers reside in the same process as the producer and which are separated in other processes, partitions, or systems. Factors that influence this decision include modularity of development, management of computational resources, and freedom from interference.
- Determine which consumers require physical or virtual memory boundaries to safeguard critical consumers from less robust or secure consumers.
- Determine which consumers must process every payload and which only require the most recent data.
- Determine which consumers cannot be fully trusted and should have their packet access limited.
- Determine how communication between the processes is established.

- Provide a uniform set of packet element types with associated data layout definitions for designing all endpoints.
- Based on the complexity of the stream, determine how many packets are required to keep the pipeline operating at desired efficiency.

Once these decisions are made, use NvSciStream to assemble the desired stream(s).

NvSciBuf and NvSciSync Initialization

All NvSci buffer and sync object handles are associated with a particular NvSciBufModule and NvSciSyncModule, respectively. Any streaming application must begin by creating one of each of these modules. In general, an application may create more than one of each type of module, but there is rarely any reason to do so. However, all buffer and sync objects used with a given stream within a single process must be associated with the same modules. Different streams may use different modules.

NvSciIpc Initialization

For cross-process streams, the applications must first establish communication with each other using NvSciIpc. Refer to Inter-Process Communicationfor more information on how to set up the connections.

Once an NvSciIpc channel is created, the applications may use it to do any required initial validation and coordination. Once that is complete, ensure no unprocessed messages remain in the channel, reset the channel, and pass the channel endpoint off to NvSciStream. NvSciStream then takes the ownership of the channel and uses it to coordinate stream operations between the two processes. From this point on, the application must not directly operate on the channel. Doing so leads to unpredictable behavior and will cause the stream to fail. After streaming is complete, the application should close the channel endpoint after deleting the NvSciStream blocks and freeing all the NvSciBuf/NvSciSync resources obtained from NvSciStream.

NvSciEventService Initialization

NvSciStream can optionally use NvSciEventService for event notification. For each NvSciStream block, applications can request an NvSciEventNotifier object on which applications can wait for new events in the block.

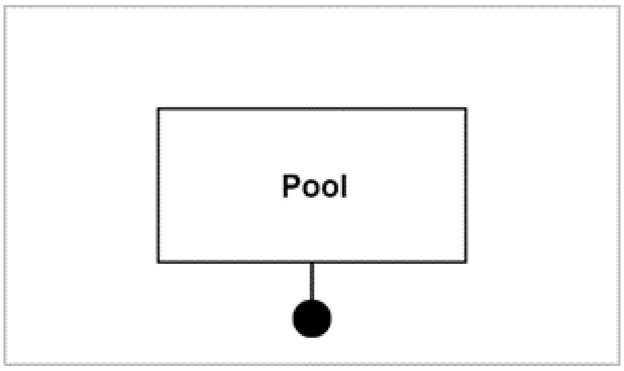
Refer to the NvSciEventService API Usage section on how to set up NvSciEventService and wait on an NvSciEventHandler object.

Block Creation

Each application is responsible for creating the NvSciStream blocks that reside in that process. The block creation functions all take a pointer parameter in. When successful, a block handle of type NvSciStreamBlock is returned. This handle is used for all further operations on the block. Some require additional parameters. For those that require a NvSciBufModule and/or NvSciSyncModule, the same module must be provided as those used to access the buffers and sync objects. Blocks may be created in any order.

Pool

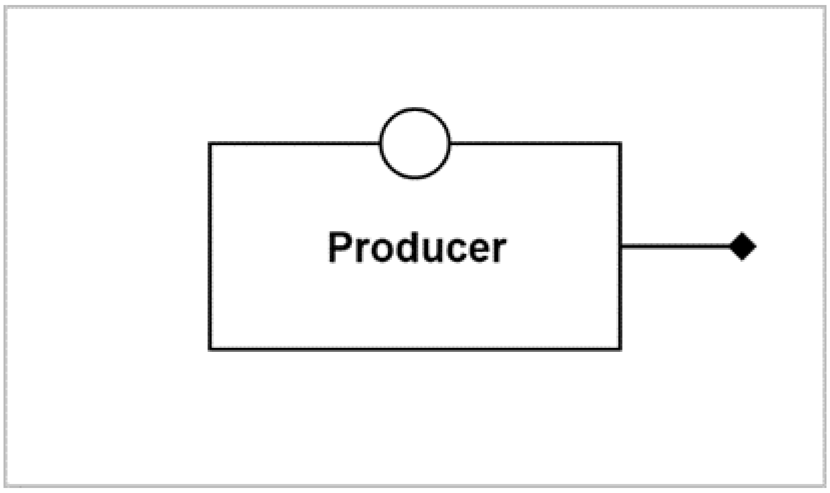
```
NvSciError
NvSciStreamStaticPoolCreate(
    uint32_t const           numPackets,
    NvSciStreamBlock *const pool
)
```



- Pool blocks have no inputs and outputs used with the connection function.
- Pools are attached directly to a producer or Memory Boundary IPC block during that block's creation.
- For static pools, the number of packets the pool provides must be specified at creation.

Producer

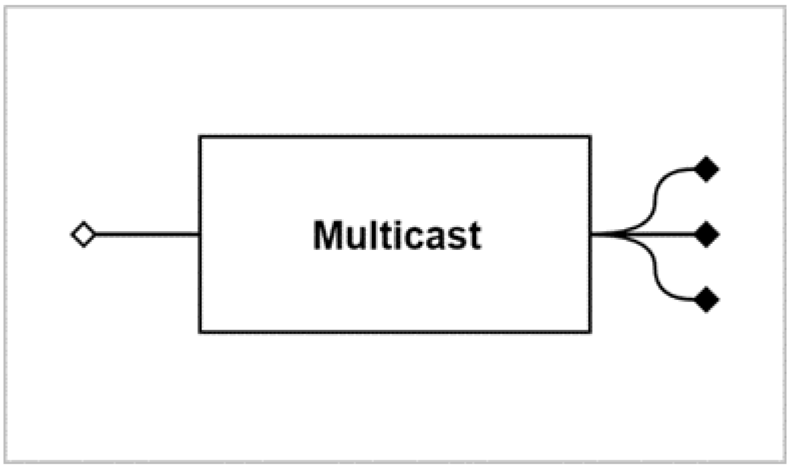
```
NvSciError
NvSciStreamProducerCreate(
    NvSciStreamBlock const  pool,
    NvSciStreamBlock *const producer
)
```



- A pool block must be provided for each producer block at creation.
- Producer blocks have a single output connection and no input connections.

Multicast

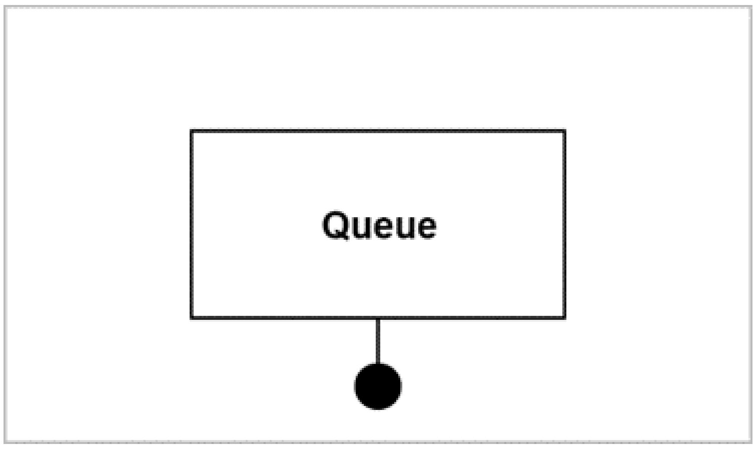
```
NvSciError
NvSciStreamMulticastCreate(
    uint32_t const         outputCount,
    NvSciStreamBlock *const multicast
)
```



- Multicast blocks have a single input connection and a fixed number of output connections specified at creation.
- During the connection process described in the following section, the order in which outputs are connected doesn't matter.

Queues

```
NvSciError
NvSciStreamFifoQueueCreate(
    NvSciStreamBlock *const queue
)
NvSciError
NvSciStreamMailboxQueueCreate(
    NvSciStreamBlock *const queue
)
```

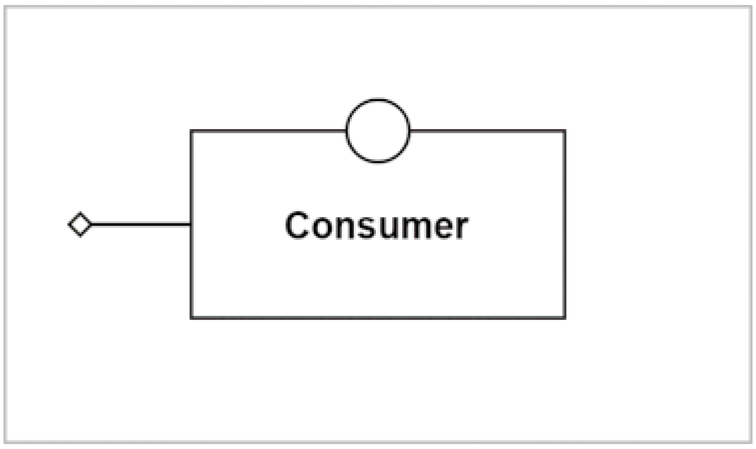


- Queue blocks have no inputs and outputs used with the connection function.

- Queues are attached directly to a consumer block or a memory boundary source IPC block during that block's creation.

Consumer

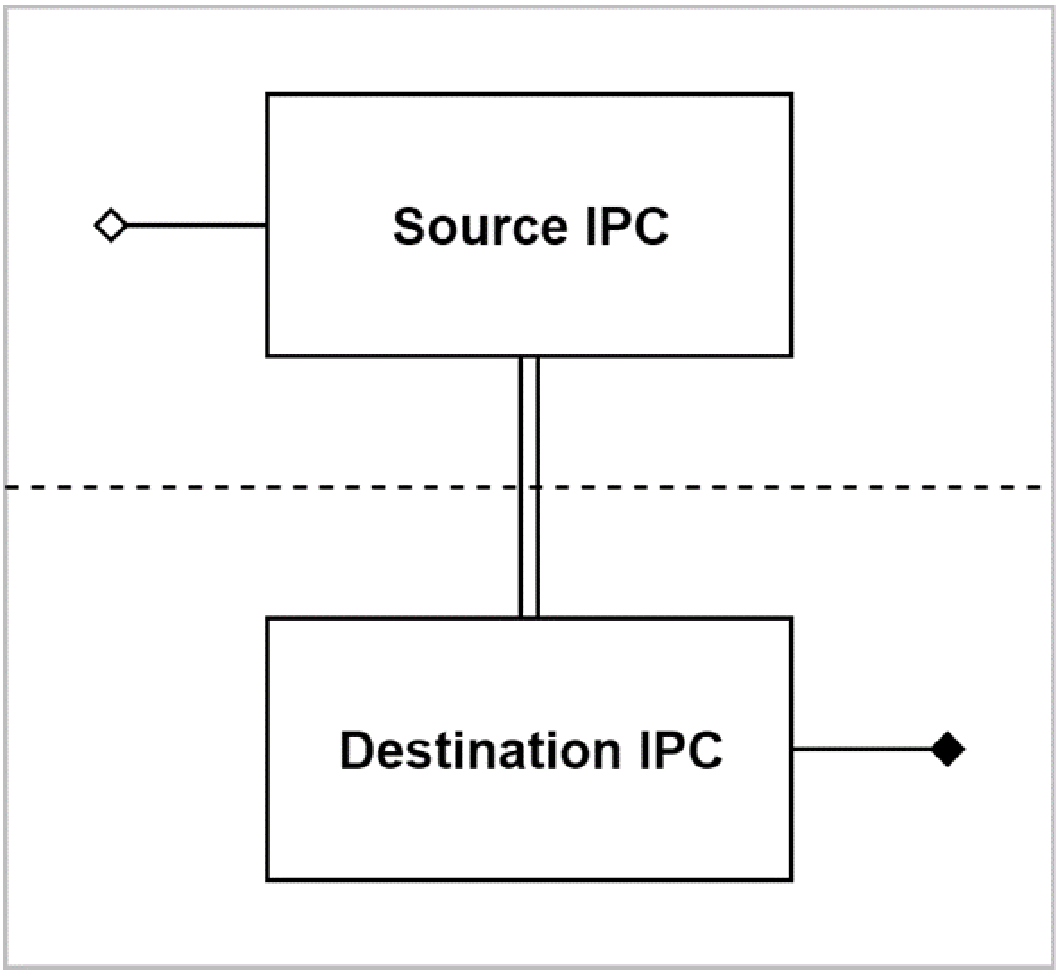
```
NvSciError
NvSciStreamConsumerCreate(
    NvSciStreamBlock const  queue,
    NvSciStreamBlock *const consumer
)
```



- A queue block must be provided for each consumer block at creation.
- Consumer blocks have a single input connection and no output connections.

IPC

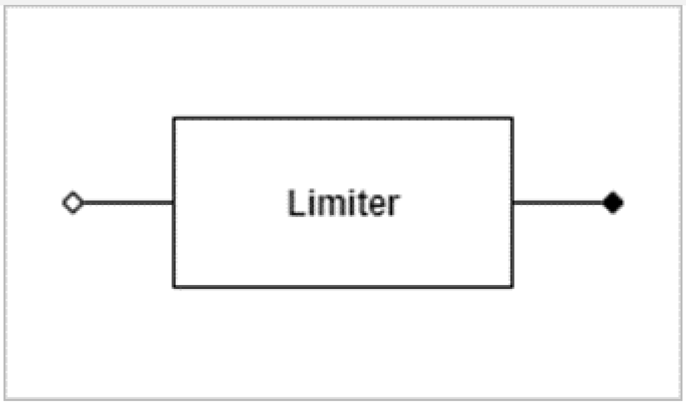
```

NvSciError
NvSciStreamIpcSrcCreate(
    NvSciIpcEndpoint const  ipcEndpoint,
    NvSciSyncModule const   syncModule,
    NvSciBufModule const    bufModule,
    NvSciStreamBlock *const ipc
)

NvSciError
NvSciStreamIpcSrcCreate2(
    NvSciIpcEndpoint const  ipcEndpoint,
    NvSciSyncModule const   syncModule,
NvSciBufModule const    bufModule,
NvSciStreamBlock const  queue,
    NvSciStreamBlock *const ipc
)

NvSciError
NvSciStreamIpcDstCreate(
    NvSciIpcEndpoint const  ipcEndpoint,
    NvSciSyncModule const   syncModule,
    NvSciBufModule const    bufModule,
    NvSciStreamBlock *const ipc
)

NvSciError
NvSciStreamIpcDstCreate2(
    NvSciIpcEndpoint const  ipcEndpoint,
    NvSciSyncModule const   syncModule,
NvSciBufModule const    bufModule,
NvSciStreamBlock const  pool,
    NvSciStreamBlock *const ipc
)
```



- IPC blocks are created in pairs, each using one end of an NvSciIpc channel.
- The caller must complete any of its own communication over the channel before passing it to NvSciStream and must not subsequently read from or write to it.
- The NvSciBuf and NvSciSync modules must be those used for all buffer and sync object associated with the stream in the calling process. The block uses them when importing objects from the other endpoint.
- The source (upstream) block has one input connection, and the destination (downstream) block has one output connection. Together with the channel between them, they are viewed as a single virtual block with one input and one output, spanning the two processes.

Limiter

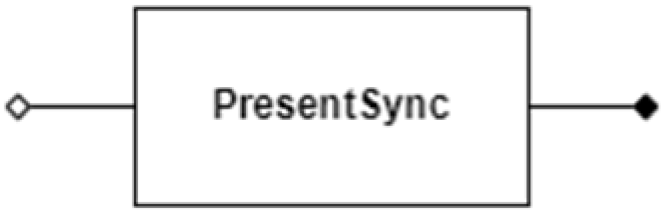
```
NvSciError
NvSciStreamLimiterCreate(
    uint32_t const          maxPackets,
NvSciStreamBlock *const limiter
)
```



- Limiter blocks have one input and one output.
- The number of packets allowed to be sent downstream to a consumer block is specified at creation.

PresentSync

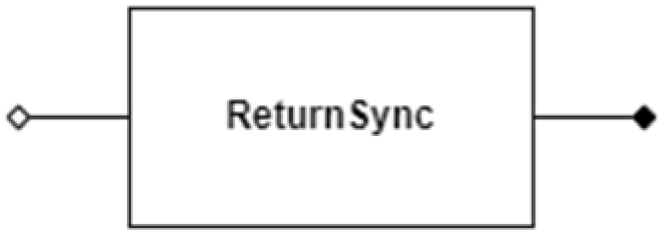
```

NvSciError
NvSciStreamPresentSyncCreate(
    NvSciSyncModule const syncModule,
    NvSciStreamBlock *const presentSync
)

```



- PresentSync block has one input and one output.
- The NvSciSyncModule used to create the block must be the module endpoints use to allocate NvSciSyncObj.

ReturnSync

```

NvSciError
NvSciStreamReturnSyncCreate(
    NvSciSyncModule const syncModule,
    NvSciStreamBlock *const returnSync
)
```



- ReturnSync block has one input and one output.
- The NvSciSyncModule used to create the block must be the module endpoints use to allocate NvSciSyncObj.

Configure Block to Use NvSciEventService

As an optional setup, applications can configure a block to use `NvSciEventService` for event notification.

```
NvSciError
NvSciStreamBlockEventServiceSetup(
    NvSciStreamBlock const block,
    NvSciEventService  *const eventService,
    NvSciEventNotifier **const eventNotifier
)
```

The function returns a `NvSciEventNotifier` object that applications can use to wait for new events on the block. It is the responsibility of the application to delete the `NvSciEventNotifier` objects when they are no longer needed. Event-notification behavior of a block is undefined after its associated `NvSciEventNotifier` object is deleted.

If any NvSciStream API, including block connection, is called on a block before this function, the block will automatically be configured to use the default event-notification method, described in the following Event Handling section. The event-notification method of a block cannot be changed once it is determined.

User-defined Endpoint Information

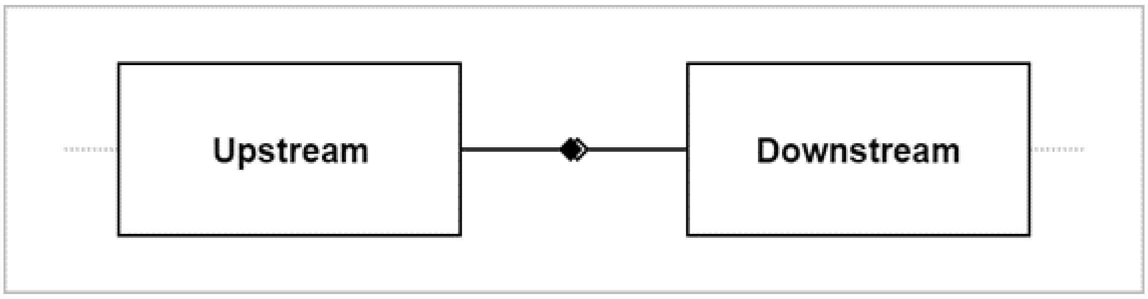
As an optional setup, before the blocks are connected, applications can supply user-defined data to the endpoint blocks (Producer and Consumer) with `NvSciStreamBlockUserInfoSet()`. After the blocks are connected, applications can query the data from every block in the connected stream with `NvSciStreamBlockUserInfoGet()`. All applications that operate the blocks should understand the `userType` value. Endpoint applications can use the information from opposed endpoints to make choices in resource creation or streaming.

```
NvSciError
NvSciStreamBlockUserInfoSet(
    NvSciStreamBlock const block,
    uint32_t const userType,
    uint32_t const dataSize,
    void const* const data
)

NvSciError
NvSciStreamBlockUserInfoGet(
    NvSciStreamBlock const block,
    NvSciSreamBlockType const queryBlockType,
    uint32_t const queryBlockIndex,
    uint32_t const userType,
    uint32_t* const dataSize,
    void* const data
)
```

Block Connection

Once blocks are created, they can be connected in pairs. The connection can be done in any order and can be intermingled with creation of the blocks.

```
NvSciError
NvSciStreamBlockConnect(
    NvSciStreamBlock const upstream,
    NvSciStreamBlock const downstream
)
```



An available output of the upstream block is connected to an available input of the downstream block. If there is no available output or input, the function fails. Each input and output block can be connected only once. A new block cannot be connected in place of an old one, even if the old one is destroyed.

Comparison with EGL

Those familiar with EGL will notice that this process is more involved than the creation of an EGLStream. For an EGLStream, all the desired features are encoded into an array of attributes, and then a single stream object (or two in the case of cross-process) is created. The details of setting up all the stream management are left to the EGL implementation. But a simple attribute array cannot convey all the possible stream feature permutations and use cases that an application may desire. In fact, multicasting to more than one consumer cannot be handled by EGLStream at all without additional extensions defining new objects.

The modular approach used by NvSciStream requires applications to perform separate creation calls for each feature of the desired stream, and then connect them together. This is more effort, but affords greater control over the stream's behavior, and allows for features not originally anticipated or readily supported by EGLStreams, such as multicasting. It allows the development of new block types to support new features in the future, which are harder to add to a monolithic stream object.

Simple Example

The following example assembles a simple cross-process stream with a producer and pool in one process, and a FIFO and consumer in another process. The cross-process communication channel is assumed to be established before this code executes.

Sample Producer Creation

```
// We'll use triple buffering for this stream
const uint32_t numPackets = 3;
// Stream variables
NvSciIpcEndpoint    srcIpc;
NvSciBufModule      bufModule     = 0;
NvSciSyncModule     syncModule    = 0;
NvSciStreamBlock    producerBlock = 0;
NvSciStreamBlock    poolBlock     = 0;
NvSciStreamBlock    srcIpcBlock   = 0;
NvSciError          err;
// Setting up communication is outside the scope of this guide
srcIpc = <something>;
// Set up buffer and sync modules
// (If using multiple streams, or doing other non-stream NvSci
//  operations, these might be passed in from some global setup.)
err = NvSciBufModuleOpen(&bufModule);
if (NvSciError_Success != err) {
    <handle failure>
}
err = NvSciSyncModuleOpen(&syncModule);
if (NvSciError_Success != err) {
    <handle failure>
}
// Create all the stream blocks
err = NvSciStreamStaticPoolCreate(numPackets, &poolBlock);
if (NvSciError_Success != err) {
    <handle failure>
}
err = NvSciStreamProducerCreate(poolBlock, &producerBlock);
if (NvSciError_Success != err) {
    <handle failure>
}
err = NvSciStreamIpcSrcCreate(srcIpc, syncModule, bufModule, &srcIpcBlock);
if (NvSciError_Success != err) {
    <handle failure>
}
// Connect the blocks
err = NvSciStreamBlockConnect(producerBlock, srcIpcBlock);
if (NvSciError_Success != err) {
    <handle failure>
}
```

Sample Consumer Creation

```
// Stream variables
NvSciIpcEndpoint    dstIpc;
NvSciBufModule      bufModule     = 0;
NvSciSyncModule     syncModule    = 0;
NvSciStreamBlock    consumerBlock = 0;
NvSciStreamBlock    fifoBlock     = 0;
NvSciStreamBlock    dstIpcBlock   = 0;
NvSciError          err;
// Setting up communication is outside the scope of this guide
dstIpc = <something>;
// Set up buffer and sync modules
// (If using multiple streams, or doing other non-stream NvSci
//  operations, these might be passed in from some global setup.)
err = NvSciBufModuleOpen(&bufModule);
if (NvSciError_Success != err) {
    <handle failure>
}
err = NvSciSyncModuleOpen(&syncModule);
if (NvSciError_Success != err) {
    <handle failure>
}
// Create all the stream blocks
err = NvSciStreamFifoQueueCreate(&fifoBlock);
if (NvSciError_Success != err) {
    <handle failure>
}
err = NvSciStreamConsumerCreate(fifoBlock, &consumerBlock);
if (NvSciError_Success != err) {
    <handle failure>
}
err = NvSciStreamIpcDstCreate(dstIpc, syncModule, bufModule, &dstIpcBlock);
if (NvSciError_Success != err) {
    <handle failure>
}
// Connect the blocks
err = NvSciStreamBlockConnect(dstIpcBlock, consumerBlock);
if (NvSciError_Success != err) {
    <handle failure>
}
```

Event Handling

NvSciStream is designed so that, once a stream is created and connected, applications can follow an event-driven model. Operations on one block in a stream trigger events in other blocks. These events are dequeued by the applications, which act in response, performing new block operations that trigger new events, and so on. Every block supports an event queue, although some types of blocks may only generate events during the initial setup phase.

Event Query

Events can be queried from each block:

```
NvSciError
NvSciStreamBlockEventQuery(
    NvSciStreamBlock const      block,
    int64_t const               timeoutUsec,
    NvSciStreamEventType *const eventType
)
```

- If the block is not configured to use NvSciEventService:
  - If no events are currently pending in the block's queue, a non-zero timeout causes it to wait the specified number of microseconds for one to arrive. If a negative value is used, the call waits forever.
- If the block is configured to use `NvSciEventService`:
  - Non-zero timeout causes it to return `NvSciError_BadParameter` error.
- The `eventType` parameter must point to an event structure the function fills in. Any previous contents of the structure are ignored and may be overwritten.

On success, a pending event is removed from the block's queue, the output parameter is filled with the type of the event, and `NvSciError_Success` is returned. If no event is available before the timeout period ends, an `NvSciError_Timeout` error is returned.

After an event is queried from a block, certain NvSciStream APIs become operational on the block. Refer to the API Reference for `NvSciStreamEventType` for additionaly information about the functions that become operational for each event type queried.

Event Notification

NvSciStream provides two event-notification methods. This is a per-block configuration, and it is not required that all blocks in a stream be configured to use the same event-notification method.

- Without `NvSciEventService`

Applications wait for events by calling `NvSciStreamBlockEventQuery()` with non-zero timeout value.

- With `NvSciEventService`

Applications wait for events by waiting on the `NvSciEventNotifier` objects bound to the blocks, using the event-waiting API of `NvSciEventService`. Applications should query all the events pending in the block after waking up from waiting on the associated `NvSciEventNotifier` object. It is possible that `NvSciEventNotifier` objects be signaled by spurious events, in which case the event query will return `NvSciError_Timeout` error.

The event-waiting API of `NvSciEventService` waits on `NvSciEventNotifier` objects, regardless of the UMD they are associated. It enables applications to simultaneously wait for `NvSciStream` events and events from other UMDs.

Refer to the Inter-Process Communication section for how to wait on a single or multiple `NvSciEventNotifier` objects.

Connection and Disconnection Events

Assuming no failures occur during setup, the first event type each block receives is `NvSciStreamEventType_Connected`. This indicates that the block has a complete connection path to the producer and consumers, respectively. After connecting, no operations are allowed on any block until this event is received. Applications must wait for it before proceeding with the resource setup described in the next section.

If a block is destroyed or, in the cross-process case, communication with a process is lost, an `NvSciStreamEventType_Disconnected` event is sent to connected blocks. If other events are pending, such as available payloads, they are processed first. Once this event is received, no further events arrive, and operations to send events fail.

Note: **In safety-certified systems, failures and teardown are never supposed to occur. Although disconnect events are currently supported in the current safety release, they will be removed in later safety releases, and only supported for non-safety platforms.**

Error Events

If the event queried is of type `NvSciStreamEventType_Error`, an error not directly triggered by user action occurred in the block. Applications can retrieve the error code with `NvSciStreamBlockErrorGet()`. If multiple errors occur before this retrieval, only the first error code is available. All subsequent errors are ignored until the error code is retrieved, with the assumption that they are related to the first error.

```
NvSciError
NvSciStreamBlockErrorGet(
NvSciStreamBlock  const block,
NvSciError* const status
)
```

Resource Creation

Once the stream blocks are fully created, the next step is to create synchronization and buffer resources. The process of determining resource requirements and allocating them is much like that described in the simple single-buffer example at the beginning of this chapter. But all coordination between the producer and consumers is done through the stream, which automatically deals with any translations required to share the resources between processes, partitions, or systems.

Creation of synchronization and buffer resources can be done in either order or can be intermingled. The two are similar, but the process for synchronizing objects is a little simpler.

Progression of Resource Creation

Various setup operations are divided into key groups. Data for each group is gathered and sent together at once when the application indicates that it is done with the setup. This completion is signaled with a call to `NvSciStreamBlockSetupStatusSet()`. The `completed` parameter to this function is for future support of dynamically modifying streams (for non-safety usecases) and currently must always be `true`.

```
NvSciError
NvSciStreamBlockSetupStatusSet(
NvSciStreamBlock  const block,
NvSciStreamSetup  const setupType,
bool const completed
)
```

Synchronization Resources

Before reading this section, be sure to read the full chapter on Synchronization to understand how synchronization requirements are specified and synchronization objects are created. This section assumes familiarity with the commands used there and does not explain them in detail. The following sections cover how to coordinate synchronization objects through a stream.

Synchronization Requirements

Setting up synchronization resources begins by determining the requirements of each endpoint. The producer and consumers must query the NVIDIA drivers they are going to use with the appropriate APIs to obtain separate NvSciSyncAttrList handles representing the requirements for signaling (writing to) and waiting for (reading from) synchronization objects.

If an endpoint directly writes to or reads from the stream packets with the CPU instead of using an NVIDIA API, it does not generate any fences, and therefore does not need synchronization signaling requirements. It needs to perform CPU waits for fences from the other endpoint, and therefore must create a waiting requirement attribute list with the NeedCpuAccess flag set.

In some use cases, an application endpoint may require that packets it receives be available immediately, without waiting for any fence. In this case, it does not need to provide any synchronization waiting requirement attributes. Instead, it indicates to the other endpoint that it must do a CPU wait before sending the packets. This is not common but is provided to support these cases when they arise.

Once an endpoint has determined its signal and wait requirements for synchronization objects, it stores the signal requirements locally, and passes the wait requirements, one `NvSciSyncAttrList` per packet element, to the other endpoint through the stream. It passes an `NvSciSyncAttrList` with the wait requirement attribute if fences are supported into the endpoint block by calling `NvSciStreamBlockElementWaiterAttrSet()`. If the waiter for the element referenced by elemIndex does not support fences, `NULL` pointer is passed. The function call is the same for both producer and consumer endpoints. Ownership of the sync attributes' handle remains with the caller. The stream creates a duplicate before the function returns.

After indicating their waiter requirements for all the elements, the producer and consumer applications call the `NvSciStreamBlockSetupStatusSet()` function with a value of `NvSciStreamSetup_WaiterAttrExport` to inform NvSciStream they are done specifying waiter requirements.

```
NvSciError
NvSciStreamBlockElementWaiterAttrSet(
NvSciStreamBlock  const block,
Uint32_t          const elemIndex,
NvSciSyncAttrList const waitSyncAttrList
)
```

Receiving Requirements

When the producer and consumer specify their synchronization requirements, the other endpoints receive a `NvSciStreamEventType_WaiterAttr` event. `NvSciStreamBlockElementWaiterAttrGet()` is operational on the endpoints receiving the event.

```
NvSciError
NvSciStreamBlockElementWaiterAttrGet(
NvSciStreamBlock   const block,
Uint32_t           const elemIndex,
NvSciSyncAttrList* const waitSyncAttrList
)
```

If `NvSciStreamBlockElementWaiterAttrGet()` sets the output parameter to NULL, the element referenced by elemIndex synchronization objects cannot be used to coordinate with the other endpoint. The application must not allocate synchronization objects for the element. This case is rare, but producer and consumer application components designed to be fully modular must recognize and handle this situation.

Otherwise, the output parameter `waitSyncAttrList` is set to a synchronization attribute list handle. Ownership of this handle belongs to the application calling the function, and it must free the handle when it is no longer needed. The attributes in this list may not exactly match those specified by the originating endpoint. For more than one consumer, the stream combines their requirements into a single attribute list for the element. The stream itself may also perform transformations on the attributes to handle cross-process or cross-system cases.

The application must merge these wait requirements with its own signal requirements to form the final reconciled synchronization attribute list for the element. It can then use the final list to allocate a synchronization object from NvSciSync. A synchronization object is allocated per asynchronous element. The application must map these synchronization objects into the drivers and use them to generate fences passed into the stream.

After receiving all the sync attribute lists they care about, the endpoint applications call the `NvSciStreamBlockSetupStatusSet` function with a value of `NvSciStreamSetup_WaiterAttrImport`.

Synchronization Objects

Sending Objects

After allocating the synchronization objects, the application must inform the stream.

For each asynchronous element, it calls `NvSciStreamBlockElementSignalObjSet()`, which is operational after event `NvSciStreamEventType_WaiterAttr` is queried from the block. Ownership of the synchronization object handle remains with the caller. The stream creates a duplicate before the function returns.

After specifying all the synchronization objects, the endpoint applications call the `NvSciStreamBlockSetupStatusSet()` function with a value of `NvSciStreamSetup_SignalObjExport.NvSciError`

```
NvSciStreamBlockElementSignalObjSet(
NvSciStreamBlock const block,
Uint32_t         const elemIndex,
NvSciSyncObj     const SignalSyncObj
)
```

Receiving Objects

On the other endpoint receves the `NvSciStreamEventType_SignalObj`.

`NvSciStreamBlockElementSignalObjGet()` is operational on the endpoint receiving the event.

```
NvSciError
NvSciStreamBlockElementSignalObjGet(
NvSciStreamBlock const block,
Uint32_t         const queryBlockIndex,
Uint32_t         const elemIndex,
NvSciSyncObj*    const SignalSyncObj
)
```

Ownership of the retrieved `NvSciSyncObj` handle belongs to the application calling the function , and it must free the sync object when it is no longer required. The application must map these into the drivers and use them to interpret fences received from the stream.

As with the requirements, the synchronization objects received may not match those sent. If there is more than one consumer, the stream combines their synchronization objects into a single list per element before passing it to the producer. The stream may also replace the synchronization objects with its own if it must perform intermediate copy operations to pass the data from one endpoint to the other.

Synchronization specified for each element requires care. If a consumer requires synchronous access (Producer receives `NULL NvSciSyncObj handle` by `NvSciStreamBlockElementSignalObjGet`()) but the producer normally generates this data asynchronously, then an extra burden is placed on the producer application. During streaming the producer application waits for that element to finish being generated before it inserts the payload into the stream.

After receiving all the synchronization objects they care about, the endpoint applications call the `NvSciStreamBlockSetupStatusSet()` function with a value of `NvSciStreamSetup_SignalObjImport`.

Comparison with EGL

When using EGLStreams, there is never any need to explicitly deal with synchronization objects. They are present in the NVIDIA EGL implementation, but require no action on the application's part. This is possible because the producer and consumer rendering libraries connect directly to an EGLStream and are able to coordinate synchronization setup themselves through the stream without the user being aware of it. In the NvSciStream model, the rendering libraries do not access the stream. Resources must be transferred between them and the stream. Therefore, these additional steps are required to initialize synchronization.

Buffer Resources

To allocate buffers, the producer and consumer blocks communicate their requirements to a third block, the pool. The application component that owns this pool block is responsible for performing the allocations. For many use cases, this is the same application that manages the producer. But keeping this functionality in a separate block from the producer serves several purposes.

It allows NvSciStream to provide different types of pools for different use cases while sharing a common set of interfaces. For instance, you may choose a static pool with a fixed set of buffers for safety-certified builds, a dynamic pool that allows buffers to be added and removed in cases where the producers may change the data layout over time, or a remote pool managed by a central server process that doles out buffers to all streams in the application suite.

Furthermore, a stream may require additional pools beyond the one which feeds buffers to the producer. For instance, in cross-system use cases where memory is not shared between the producer and consumer, the IPC block on the consumer side also requires a pool to provide buffers into which data is copied. Keeping the pool as a separate block allows generic application components to be written that do not need to know whether the pool is used with a producer or another block.

Before reading this section, refer to Buffer Management to understand how buffer requirements are specified and how buffers are created. This section assumes familiarity with the commands used there and does not explain them in detail. This section covers how to coordinate buffers through a stream.

Buffer Requirements

Specifying Requirements

With synchronization objects, fences are generated at both endpoints and flow in both directions, so it was necessary for producer and consumer(s) to determine and exchange both write and read requirements. Buffer data, on the other hand, flows in only one direction, from the producer to the consumer(s). So, the producer must query the NVIDIA drivers they use for buffer attribute lists that provide write capability, while consumers must query for buffer attributes that provide read capability. If the buffer memory is written or read directly with the CPU, attribute lists can be manually created, requesting CPU access.

A packet may consist of multiple buffer elements containing different types of data. The producer must obtain a buffer attribute list for each type of data it can generate, and consumers must obtain attribute lists for each type of data they require.

Once a producer knows all the elements it can provide, or a consumer knows all the elements it requires, it can inform the stream by calling `NvSciStreamBlockElementAttrSet()` (operational after `NvSciStreamEventType_Connected` is queried) for each element. The `userType` parameter is an application-defined value to identify the element, understood by both the producer and consumer(s). The `userType` should be unique for each element. The `bufAttrList` field contains a handle for the element's attribute list. Ownership of this handle remains with the caller, and it deletes it when it is no longer needed after the function returns. NvSciStream creates a duplicate.

After specifying all of the elements, the application will indicate it is done with the element setup by calling NvSciStreamBlockSetupStatusSet() with a value of NvSciStreamSetup_ElementExport. The indices and count will be determined automatically by NvSciStream.

Asynchronous elements data is not available to the consumer until the fences complete. This generally indicates data written using NVIDIA hardware. Synchronous elements data is available to be read as soon as the packet is acquired by the consumer, without waiting for any fence. This generally indicates data written directly with the CPU. Whether an element is asynchronous or synchronous is determined by sync requirements setup steps.

A typical use case that a packet contains both asynchronous and synchronous elements is a packet containing a large primary data element accompanied by a smaller metadata element containing information required to program the hardware for processing the primary element. Upon acquiring the packet, the metadata must be read immediately by the CPU, so that instructions for the NVIDIA drivers can be issued, but the primary data is read later once the fence has been reached.

```
NvSciError
NvSciStreamBlockElementAttrSet(
    NvSciStreamBlock const block,
uint32_t         const userType,
NvSciBufAttrList const bufAttrList
)
```

Receiving Requirements

Pool, producer, and consumer each receive a single `NvSciStreamEvent_Elements` event. The pool receives the event after the producer and all consumers finish specifying their element support. After the pool finishes specifying the final packet element layout, the producer and consumers receive this event. After this event is queried from pool, the following functions become operational on the pool block:

```
    NvSciStreamBlockElementCountGet()
    NvSciStreamBlockElementAttrGet()
    NvSciStreamBlockElementAttrSet()
```

After the producer and consumers query the event, the following functions become operational on the producer and consumer blocks:

```
    NvSciStreamBlockElementCountGet()
NvSciStreamBlockElementAttrGet()
NvSciStreamBlockElementUsageSet() (consumer only)
    NvSciStreamBlockElementWaiterAttrSet()
```

As with synchronization requirements, the attributes received by the pool may not exactly match those sent by the producer and consumer endpoints. For the multicast case, the pool receives a combined list with one element for each type of attribute the consumers requested. Elements with the same type arrive as a single event with their attribute lists merged. The stream may also transform attributes to handle cross-process or cross-system cases. When querying the elements, consumers can choose not to use all of them. A consumer can inform NvSciStream that an element will not be used by calling NvSciStreamBlockElementUsageSet() with <used> set to false. This will allow NvSciStream to optimize by not sharing the relevant buffers with the consumer. This function can be called with <used> set to true, but this is the default, and the call is not necessary. Therefore, most existing applications will not need to add this call.

After querying element information and (for the consumer) indicating which elements they will support, the producer and consumer(s) must call `NvSciStreamBlockSetupStatusSet()` function with a value of `NvSciStreamSetup_ElementImport`. This allows them to begin receiving packets from the pool.

Any secondary pools connected to IPC blocks also receive events for the producer and consumer elements. However, the producer events are delayed until the primary pool determines the final packet layout (discussed below). The producer element attributes the secondary pools receive are those sent by the primary pool.

```
NvSciError
NvSciStreamBlockElementCountGet(
    NvSciStreamBlock const block,
NvSciStreamBlockType const queryBlockType,
uint32_t* const numElements
)

NvSciError
NvSciStreamBlockElementAttrGet(
NvSciStreamBlock const block,
NvSciStreamBlockType const queryBlockType,
uint32_t          const elemIndex,
uint32_t*         const userType,
NvSciBufAttrList* const bufAttrList
)

NvSciError
NvSciStreamBlockElementUsageSet(
NvSciStreamBlock const block,
uint32_t const elemIndex,
bool const used
)

```

Reconciling Requirements

For the pool attached to the producer:

The application managing the pool must take the capabilities provided by the producer and the requirements provided by the consumer(s) and determine the final packet layout. For each element type provided by the producer and required by a consumer, it should merge their attribute lists to obtain the attribute list to use in allocating the buffer. If there is an element type provided by the producer but not required by any of the consumers, omit it from the packets. If there is a type required by a consumer that the producer cannot provide, the application can trigger an error. However, there may be cases understood by the application suite where the consumer requirements represent several options, and only one of the requested element types is needed. If the producer can provide one of them, streaming can proceed. It is to support possible complex situations like this that the reconciliation process is left to applications, rather than being done automatically by the stream.

As in producer and consumer(s) applications, pool application calls the `NvSciStreamBlockSetupStatusSet()` function with a value of `NvSciStreamSetup_ElementImport` to inform NvSciStream it has finished element import.

Once the application has determined the final layout, it calls `NvSciStreamBlockElementAttrSet()` to inform NvSciStream the reconciled element attributes. It also must call `NvSciStreamBlockSetupStatusSet()` function with a value of `NvSciStreamSetup_ElementExport` to inform NvSciStream it finished exporting element attributes. Producer and consumer(s) application(s) query event `NvSciStreamEventType_Elements` and get the element attributes by the function described above. Applications use this information to interpret the data layout and prepare to receive the packets.

For the pool attached to the memory-boundary destination IPC:

The application retrieves the element attributes reconciled by the producer's pool after querying event `NvSciStreamEventType_Elements`. The application calls the `NvSciStreamBlockSetupStatusSet()` function with a value of `NvSciStreamSetup_ElementImport` to inform NvSciStream it finished element import.

Buffer Exchange

Specifying Buffers

Once the element attributes are specified, the application can allocate buffers from NvSciBuf using the attribute lists. It creates packets using the pool object, allocates a buffer for each element of each packet, and then assigns them to their proper places in the packets.

To create a new packet, the application calls `NvSciStreamPoolPacketCreate()`. The cookie is any non-zero value that the application chooses to look up data structures for the packet. Typically, the cookie is either a 1-based index into an array of structures or a pointer directly to the structure. Any events received on the pool object related to the packet references this cookie. On success, the function returns a new handle that the application stores in its data structure for the packet and uses whenever it needs to tell the stream to operate on the packet.

After creating a packet, the application assigns a buffer to each element by calling `NvSciStreamPoolPacketInsertBuffer()`. The packet handle is returned at packet creation, and the index of the element is assigned the buffer handle, which NvSciStream duplicates. Ownership of the original remains with the caller, and once the function returns, it may safely free the buffer.

The application calls NvSciStreamPacketComplete() to inform NvSciStream finished assigning buffers for a packet.

When the application finished creating all packets and assigned all buffers, it calls the NvSciStreamBlockSetupStatusSet() function with a value of NvSciStreamSetup_PacketExport to indicate completion of packet creation.

For static pools, the number of packets is specified when the pool is created. Streaming won't start until this number of packets is created. If the application tries to create more than this number of packets, an error occurs.

```
NvSciError
NvSciStreamPoolPacketCreate(
    NvSciStreamBlock const   pool,
    NvSciStreamCookie const  cookie,
    NvSciStreamPacket *const handle
)

NvSciError
NvSciStreamPoolPacketInsertBuffer(
    NvSciStreamBlock const  pool,
    NvSciStreamPacket const handle,
    uint32_t const          index,
    NvSciBufObj const       bufObj
)

NvSciError
NvSciStreamPoolPacketComplete(
    NvSciStreamBlock const  pool,
    NvSciStreamPacket const handle
)
```

Receiving Buffers

When a pool completes a new packet, any producer or consumers that pool serves is notified with a NvSciStreamEventType_PacketCreate event.

Upon receiving this event, the endpoint calls NvSciStreamBlockPacketNewHandleGet() to dequeue the handle of the new packet and then NvSciStreamBlockPacketBufferGet() to get the buffers for the elements in the packet. After checking whether it can map in all the buffers for a given packet, the endpoint signals status back to the pool by

NvSciStreamBlockPacketStatusSet(). The status parameter indicates whether the application was successful in setting up the new packet. If so, the value is NvSciError_Success. Otherwise, it can be any value the application chooses. NvSciStream does not interpret the value except to check for success and passes it back to the pool. If successful, the cookie parameter provides the endpoint's cookie for the packet. Each endpoint can provide its own cookie for each packet, which is used in subsequent events. The producer and consumers .

```
NvSciError
NvSciStreamBlockPacketNewHandleGet(
    NvSciStreamBlock const  block,
    NvSciStreamPacket* const handle
```

```
)

NvSciError
NvSciStreamBlockPacketBufferGet(
    NvSciStreamBlock const  block,
NvSciStreamPacket const handle,
uint32_t const elemIndex,
NvSciBufObj* const bufObj
)

NvSciError
NvSciStreamBlockPacketStatusSet(
    NvSciStreamBlock const  block,
NvSciStreamPacket const handle,
NvSciStreamCookie const cookie,
NvSciError const status

```

After the pool finishes exporting the packets, the endpoints receive an `NvSciStreamEventType_PacketsComplete` event. They can complete setup related to packet resources and call the `NvSciStreamBlockSetupStatusSet()` function with a value of `NvSciStreamSetup_PacketImport` to indicate they finished importing the packets.

If a pool deletes a packet, when the producer or consumer receives the `NvSciStreamEventType_PacketDelete` event it can determine the identity of the deleted packet by calling `NvSciStreamBlockPacketOldCookieGet()`, which retrieves the cookie of a packet pending deletion. The handle of the returned packet becomes invalid for subsequent function calls.

```
NvSciError
NvSciStreamBlockPacketOldCookieGet(
    NvSciStreamBlock const  block,
NvSciStreamCookie* const cookie
)

```

Completing Buffer Setup

When the producer and consumers accept a packet, a the pool receives the `NvSciStreamEventType_PacketStatus` event. The pool application calls `NvSciStreamPoolPacketStatusAcceptGet()` for the acceptance status of a packet. If the packet is not accepted by producer or consumers, the pool application can call `NvSciStreamPoolPacketStatusValueGet()` to get the error code.

After receiving the acceptance status of all packets, the pool application calls the `NvSciStreamBlockSetupStatusSet()` function with a value of `NvSciStreamSetup_PacketImport` to complete the buffer setup.

```
NvSciError
NvSciStreamPoolPacketStatusAcceptGet(
    NvSciStreamBlock const pool,
NvSciStreamPacket const handle,
bool* const acccepted
)

```

```
NvSciError
NvSciStreamPoolPacketStatusValueGet(
    NvSciStreamBlock const pool,
NvSciStreamPacket const handle,
NvSciStreamBlockType const queryBlockType,
uint32_t const queryBlockIndex,
NvSciError* const status
)
```

Comparison with EGL

When using EGLStreams, back-and-forth coordination of buffer attributes between the endpoints is not required. Buffers are created through the producer rendering interface and are communicated to the consumer when inserted in the stream. The downsideis that, in most cases, the producer has no awareness of the consumer for which the buffers are intended. There is no way to ensure that the buffers the producer allocates are compatible with the consumer at the time they are created. If they are not, then either streaming fails when the consumer receives the buffers, or a costly conversion process must occur for every frame. The NvSciStream model puts more burden on the application when establishing the buffers but ensures optimal allocation settings for compatibility between producer and consumers.

Additionally, EGLStreams are far more restricted in the kinds of buffers they support. They allow two-dimensional image buffers rendered by the GPU and video, along with a limited set of metadata buffers generated by the CPU. NvSciStream allows multiple buffers of data rendered by either source. These buffers can contain images, arrays, tensors, or anything else the user requires.

Frame Production

Once setup is complete, the producer application enters a cycle of receiving empty packets for reuse, writing to them, and inserting them back into the stream to be sent to the consumer(s).

When the pool indicates it exported all the packets, and the endpoints indicate they finished importing the packets and importing and exporting the sync objects, all blocks will receive a `NvSciStreamEventType_SetupComplete` event in the stream. This indicates that all necessary setup steps are complete. Applications that divide their event handling into separate initialization and runtime phases can use this to trigger the transition.

Note:	**Resizing or otherwise replacing buffers during streaming is supported for non-safety builds and is described in a later section when it becomes available. It is not supported in the current release.**

Obtaining Packets

When setup completes, the pool begins releasing the packets to the producer block for rendering. Subsequently, as payloads are returned from downstream because they were skipped or the consumer no longer needs them, they become available to the producer again. The order in which packets are received by the producer depends on various stream settings and is not deterministic, but the pool tries to optimize so that buffers with the least wait time until it is safe to write to them are available first.

When a packet becomes available for writing, a `NvSciStreamEventType_PacketReady` event is received by the producer block. The packet info is retrieved by a subsequent call. If multiple packets are available, the producer receives separate events for each of them.

After a packet becomes available, a producer application may call `NvSciStreamProducerPacketGet()` to obtain it. The cookie field is filled in with the cookie the producer assigned to the packet. With the packet handle, the producer application calls `NvSciStreamBlockPacketFenceGet()` to obtain the prefences for the packet elements. The prefence field points to the location that will be filled with the fence value indicating when the consumer indexed by `queryBlockIndex` is no longer using the data in the packet element's buffer.

```
NvSciError
NvSciStreamProducerPacketGet(
    NvSciStreamBlock const   producer,
    NvSciStreamCookie *const cookie)

NvSciError
NvSciStreamBlockPacketFenceGet(
NvSciStreamBlock const block,
NvSciStreamPacket const handle,
uint32_t const queryBlockIndex,
uint32_t const elemIndex,
NvSciSyncFence* const prefence
)
```

Writing Packets

Upon obtaining a packet, the producer application must ensure that the consumers are done reading from it before modifying their contents. If it is writing to the buffers using NVIDIA API, it must use the appropriate API-specific operation to insert a wait for each of the fences into the hardware command sequence. Then it looks up the API-specific buffer handle(s) for the packet and makes those buffers the current rendering targets. It may then proceed to issue rendering commands.

If instead the application writes directly to the buffer memory, it performs a CPU wait for the fences of the element buffers it wants to write to. Then it can begin writing the new data.

A producer application may retrieve multiple packets at once from the pool. It may operate on them at the same time and may insert the completed packets into the stream in any order, not necessarily in which they were retrieved.

Presenting Packets

When it finishes issuing its rendering instructions, the producer application must provide synchronization for their completion For synchronous elements the producer must perform a CPU wait for rendering to finish on those synchronous elements before inserting the packet in the stream. For asynchronous elements, it instructs the APIs it used to generate fences to trigger when rendering finishes. The application sets the fences for the elements of the packet that is going to be presented by calling `NvSciStreamBlockPacketFenceSet()`.

The application can now insert the packet back into the stream along with the fences by calling `NvSciStreamProducerPacketPresent()` with the handle of the packet. The stream makes the packet available to consumers, performing any necessary copy or translations steps along the way to make the data and fences accessible. Once a packet is presented, the application must not attempt to modify its contents until it is returned for reuse.

```
NvSciError
NvSciStreamBlockPacketFenceSet(
NvSciStreamBlock const block,
NvSciStreamPacket const handle,
uint32_t const elemIndex,
NvSciSyncFence const* const postfence
)

NvSciError
NvSciStreamProducerPacketPresent(
    NvSciStreamBlock const      producer,
    NvSciStreamPacket const     handle
)
```

Comparison with EGL

With EGLStreams, the process for rendering and presenting frames varies depending on the rendering API chosen. For EGLSurface producers, the application never directly interacts with the individual image buffers. It simply issues rendering instructions for the surface and a swap command when it is done with each frame. The NvSciStream process therefore requires more hands-on interaction.

By contrast, use of CUDA producers for EGLStreams follows a very similar pattern to NvSciStream. After the first time a buffer is used, it is obtained from the stream when the consumer returns it. The application renders to the buffer, and then once again inserts it into the stream. The only real difference in buffer access is that with EGLStreams, the API's buffer handles are returned directly, whereas with NvSciStream the application receives a cookie and must look up the corresponding API handle.

For all EGLStream producers, a key difference in NvSciStream is the need to manage fences. In EGLStreams, the act of presenting a frame triggers automatic fence generation based on internal tracking of the last instructions issued for the buffer. Similarly, when buffers are returned, a fence from the consumer is internally associated with the buffer and the API waits for it the next time that buffer is used. In NvSciStream, the application must take control of requesting fences from the APIs to be inserted in the stream and waiting for fences received from the stream. This is necessary to support the more general variety of usage models that NvSciStream can handle that EGLStream cannot.

Frame Consumption

The consumer side cycle mirrors that of the producer, receiving packets full of data, reading from them, and returning them to the producer to be reused.

Acquiring Packets

Just as the producer block receives a NvSciStreamEventType_PacketReady event when a packet is available for reuse, the consumer block receives one when a packet containing new data arrives. Again, none of the extra data fields in the event structure are used for this type of event, and the packet info is retrieved by a subsequent call. If multiple packets are available, the consumer receives separate events for each of them.

After a packet becomes available, a consumer application may call `NvSciStreamConsumerPacketAcquire()` to obtain it. The cookie field is filled in with the cookie the consumer assigned to the packet. The prefences field points to an array with space for one fence for each of the sync objects provided by the producer. If the producer provides zero sync objects, this field can be NULL. The function fills in the array with the fences that indicate when the producer is finished writing data to the buffers.

Consumer packets are always received in the order that the producer sends them, but depending on the stream settings (e.g., if a mailbox queue is used) some packets may be skipped. The consumer may acquire and hold multiple packets at once.

```
NvSciError
NvSciStreamConsumerPacketAcquire(
    NvSciStreamBlock const   consumer,
    NvSciStreamCookie *const cookie,
    NvSciSyncFence *const    prefences
)
```

Reading Packets

Upon obtaining a packet, the consumer must wait until the contents are ready before reading from it. Any elements that are set up with immediate mode can be accessed right away. For all other elements, it must wait for the fences provided.

If it reads from the buffers using NVIDIA API, it must use the appropriate API-specific operation to insert a wait for each of the fences into the hardware command sequence. Then it can look up the API-specific buffer handle(s) for the packet and make those buffers the current read sources. It can then proceed to issue commands that use the data.

If instead the application reads directly from the buffer memory, it must perform a CPU wait for all of the fences in the array. Then it can begin reading the new data.

Releasing Packets

When it finishes issuing instructions to read from a packet, the consumer must return the packet to the producer for reuse. Packets may be returned in any order, regardless of the order they were acquired.

Before sending the frame back, the consumer provides synchronization to indicate when its operations complete. If the producer set the `synchronousOnly` flag during setup of the synchronization objects, then the consumer must perform a CPU wait for reading to finish before inserting the packet in the stream. Otherwise, it must instruct the APIs it used to generate fences, which triggers when rendering finishes. It fills in an array of fences, one for each of the sync objects the consumer created. If some of the sync objects aren't relevant for how a given packet was used, the application can clear their entries in the fence array to empty but must not leave the array contents undefined.

The application can now insert the packet back into the stream along with the fences by calling `NvSciStreamConsumerPacketRelease()` with the handle of the packet and the array of post-fences. The stream returns the packet to the producer, performing any necessary copy or translations steps along the way to make the fences accessible. Once a packet is released, the application must not attempt to read its contents until a new payload using the packet arrives.

```
NvSciError
NvSciStreamConsumerPacketRelease(
    NvSciStreamBlock const      consumer,
    NvSciStreamPacket const     handle,
    NvSciSyncFence const *const postfences
)
```

Comparison with EGL

As with the producer, consumer similarities between EGLStreams and NvSciStream depend on the rendering API chosen. For GL texture consumers, acquired buffers are bound directly to a selected texture, without the application knowing their details. CUDA consumers are closer to NvSciStream. The application receives individual buffers to read from and returns those to the stream when they are no longer required.

Again, the key difference in NvSciStream is the need to manage fences. Just as with the producer, EGLStreams handles all synchronization on behalf of the application, while with NvSciStream, the application is responsible for generating and waiting for fences.

Teardown

In safety-certified builds, destruction of NvSciStream blocks and packets is not supported. Once created, all objects persist until the application shuts down. In non-safety builds, streams can be dynamically torn down at any time, and new streams can be created.

Packet Destruction

Packets can be destroyed one by one with the function:

```
NvSciError
NvSciStreamPoolPacketDelete(
    NvSciStreamBlock const  pool,
    NvSciStreamPacket const handle
)
```

This API schedules an existing packet to be removed from the pool. If the packet is currently in the pool, it is removed right away. Otherwise, this is deferred until the packet returns to the pool. When the specified packet is returned to the pool, the pool releases resources associated with it and sends a NvSciStreamEventType_PacketDelete event to the producer and consumer. Once deleted, the packet may no longer be used for pool operations.

Block Destruction

Regardless of type, all blocks are destroyed with the same function:

```
NvSciError
NvSciStreamBlockDelete(
    NvSciStreamBlock const block
)
```

When called, the block handle immediately becomes invalid, making it an error to use in any function call unless it is reused for a new block. Any blocks connected to this one is informed that the stream has disconnected, if they have not already, so that they may do an orderly cleanup. (It is not possible to connect a new block in place of a destroyed one.) Any resources associated with the block are scheduled for deletion, although this may not happen immediately if they are still in use by some part of the pipeline. If there is an NvSciEventNotifier bound to the block, it is unbound.

Synchronization

This chapter describes how to set up synchronization objects required by an application or a set of applications, and how to use them to control the order in which operations are performed by NVIDIA hardware. The basic setup process is similar to the process used for allocating buffers described in the previous chapter:

- Specify the restrictions imposed by the hardware components that signal the sync objects.
- Specify the restrictions imposed by the hardware components that wait on the sync fences.
- Gather the information for each set of sync objects into one application, which allocates them.
- Share the allocated sync objects with other applications.
- Map the sync objects into UMD specific interfaces.

The streaming process issues commands to update the sync objects, and commands to waits for those updates, so that different sets of operations remain in sync with each other.

Terminology

Agent: An entity in the system that executes instructions. An agent may be a CPU thread and also a hardware engine. An agent is the entity that interacts with actual hardware primitives abstracted by NvSciSync. The goal of NvSciSync is to synchronize agents.

Synchronization Basics

In an NVIDIA hardware system, there are typically multiple execution agents running simultaneously. For example: CPU, GPU, and other engines. Task interfaces to engines behave asynchronously. The CPU prepares a job for it and queues it in the interface. There might be multiple jobs pending on an engine and they are scheduled automatically in the hardware. CPU and applications don't know upfront the order in which jobs are executed.

This creates a need for job synchronization with dependencies between them, like a pipeline of several engines working on the same data, which preferably should not involve expensive CPU intervention.

NVIDIA hardware supports multiple synchronization mechanisms that solve this problem. They are context sensitive and not every engine understands all the mechanisms.

NvSciSync provides an abstraction layer that hides details of synchronization primitives used in a concrete situation. One of the most basic concepts of NvSciSync is a sync object. It abstracts a single instance of a specific synchronization primitive. A sync object has a current state and can be signaled. Signaling a sync object moves it to the next state. Normally, an application developer associates a sync object with a chain of events that must occur in the same order. For example, a video input engine may always signal the same sync object after producing a camera frame. This way, the sync object can be inspected at any time to check which frames were already written. A sync object must only be signaled by a single agent. An agent that signals (at the request of an application) is called a signaler.

Another basic concept of NvSciSync is a sync fence. A sync fence is associated with a specific sync object and contains a snapshot of that object's state. A fence is considered expired if its snapshot is behind or equal to the current state of the object. A fence whose state has not yet been reached by the object is said to be pending. Usually, multiple fences are associated with a single sync object and might correspond to different states of that object. A sync fence is generated by the signaler application and shared with others. An application can make an agent wait on a fence. An agent waiting on a sync fence is called a waiter in the context of the given sync object.

NvSciSync Module

To use NvSciSync you must first open an NvSciSyncModule. This module represents the library's instance created for that application and acts as a container for other NvSciSync resources. Typically, there is only a single NvSciSyncModule in an application but having all resources contained in NvSciSyncModule allows multiple threads or other libraries to use NvSciSync in an isolated manner. All other NvSciSync resources are associated with an NvSciSyncModule on creation.

NvSciSyncModule

```
NvSciSyncModule module = NULL;
NvSciError err;
err = NvSciSyncModuleOpen(&module);
if (err != NvSciError_Success) {
    goto fail;
}
```

```
/* ... */
NvSciSyncModuleClose(module);
```

Inter-Application

If there are multiple processes involved, all communication of NvSciSync structures should go via NvSciIpc channels. Each application needs to open its own Ipc endpoints.

NvSciIpc init

```
NvSciIpcEndpoint ipcEndpoint = 0;
NvSciError err;
err = NvSciIpcInit();
if (err != NvSciError_Success) {
    goto fail;
}
err = NvSciIpcOpenEndpoint("ipc_endpoint", &ipcEndpoint);
if (err != NvSciError_Success) {
    goto fail;
}
/* ... */
NvSciIpcCloseEndpoint(ipcEndpoint);
NvSciIpcDeinit();
```

NvSciSync Attributes

NvSciSync clients must supply the properties and constraints of an NvSciSync object to NvSciSync before allocating the object. This is expressed with attributes. An attribute is a key - value pair. You can view all supported keys in the header files together with value types that can be used with them.

Each application wanting to use a sync object indicates its needs in the form of various attributes before the sync object is created. Those attributes are then communicated to the signaler, who gathers all applications' attributes and has NvSciSync reconcile them. Successful reconciliation creates a new attribute list satisfying all applications constraints. The signaler then allocates a sync object using resources described by those attributes. This sync object, together with reconciled attributes list, is then shared with all waiters that need access to this sync object.

NvSciSync Attributes List

Attributes coming from a single source are kept in an attribute list structure.

NvSciSyncAttrList

```
NvSciSyncAttrList attrList = NULL;
NvSciError err;
/* create a new, empty attribute list */
err = NvSciSyncAttrListCreate(module, &signalerAttrList);
if (err != NvSciError_Success) {
    goto fail;
}
/*
 * fill the list - this example corresponds to a CPU signaler
 * that only needs to signal but not wait
*/
NvSciSyncAttrKeyValuePair keyValue[2] = {0};
bool cpuSignaler = true;
keyValue[0].attrKey = NvSciSyncAttrKey_NeedCpuAccess;
keyValue[0].value = (void*) &cpuSignaler;
keyValue[0].len = sizeof(cpuSignaler);
NvSciSyncAccessPerm cpuPerm = NvSciSyncAccessPerm_SignalOnly;
keyValue[1].attrKey = NvSciSyncAttrKey_RequiredPerm;
keyValue[1].value = (void*) &cpuPerm;
keyValue[1].len = sizeof(cpuPerm);
err = NvSciSyncAttrListSetAttrs(list, keyValue, 2);
if (err != NvSciError_Success) {
    goto fail;
}
/* ... */
NvSciSyncAttrListFree(signalerAttrList);
```

Reconciliation

After gathering all attribute lists, the signaler application must reconcile them. Successful reconciliation results in a new, reconciled attribute list that satisfies all applications' requirements. If NvSciSync cannot create such a list because attributes contradict, it instead creates an attribute list that describes the conflicts in more detail. In the example below, assume that waiterAttrList1 and waiterAttrList2 were created in the same process, so the variables are visible.

```
NvSciSyncAttrList unreconciledList[3] = {NULL};
NvSciSyncAttrList reconciledList = NULL;
NvSciSyncAttrList newConflictList = NULL;
NvSciError err;
unreconciledList[0] = signalerAttrList;
unreconciledList[1] = waiterAttrList1;
unreconciledList[2] = waiterAttrList2;
err = NvSciSyncAttrListReconcile(
        unreconciledList,        /* array of unreconciled lists */
        3,                       /* size of this array */
        &reconciledList,         /* output reconciled list */
        &newConflictList);       /* conflict description filled in case of reconcili
ation failure */
if (err != NvSciError_Success) {
    goto fail;
}
/* ... */
NvSciSyncAttrListFree(reconciledList);
NvSciSyncAttrListFree(newConflictList);
```

NvSciSync recognizes which attribute lists are reconciled and which are not. Some NvSciSync APIs that take NvSciSyncAttrList expect the list to be reconciled.

Inter-Application

The above example assumes that waiterAttrList1 and waiterAttrList2 were received from the waiter applications. Hence, no cross-process semantics were required. If a waiter lives in another process, then the attributeList must be exported to a descriptor first, communicated via NvSciIpc, and then imported on the receiver's end. In some cases, there are multiple lists travelling through multiple NvSciIpc channels.

Export/Import NvSciSyncattrList

```
/* waiter */
NvSciSyncAttrList waiterAttrList = NULL;
void* waiterListDesc = NULL;
size_t waiterListDescSize = 0U;
NvSciError err;
/* creation of the attribute list, receiving other lists from other waiters */
err = NvSciSyncAttrListIpcExportUnreconciled(
    &waiterAttrList,                /* array of unreconciled lists to be
exported */
    1,                              /* size of the array */
    ipcEndpoint,                    /* valid and opened NvSciIpcEndpoint
intended to send the descriptor through */
    &waiterListDesc,                /* The descriptor buffer to be allocated
and filled in */
    &waiterListDescSize );          /* size of the newly created buffer */
if (err != NvSciError_Success) {
    goto fail;
}
/* send the descriptor to the signaler */
NvSciSyncAttrListFreeDesc(waiterListDesc);
/* signaler */
void* waiterListDesc = NULL;
size_t waiterListDescSize = 0U;
NvSciSyncAttrList unreconciledList[2] = {NULL};
NvSciSyncAttrList reconciledList = NULL;
NvSciSyncAttrList newConflictList = NULL;
NvSciSyncAttrList signalerAttrList = NULL;
NvSciSyncAttrList importedUnreconciledAttrList = NULL;
/* create the local signalerAttrList */
/* receive the descriptor from the waiter */
err = NvSciSyncAttrListIpcImportUnreconciled(module, ipcEndpoint,
    waiterListDesc, waiterListDescSize,
    &importedUnreconciledAttrList);
if (err != NvSciError_Success) {
    goto fail;
}
/* gather all the lists into an array and reconcile */
unreconciledList[0] = signalerAttrList;
unreconciledList[1] = importedUnreconciledAttrList;
err = NvSciSyncAttrListReconcile(unreconciledList, 2, &reconciledList,
        &newConflictList);
if (err != NvSciError_Success) {
    goto fail;
}
```

```
/* ... */
NvSciSyncAttrListFree(importedUnreconciledAttrList);
NvSciSyncAttrListFree(reconciledList);
```

Sync Management

NvSciSync Objects

The NvSciSyncObj structure represents a sync object. It can be allocated after successful reconciliation. The reconciled attribute list contains all information about resources needed for the allocation.

NvSciSyncObj

```
/* Create NvSciSync object and get the syncObj */
NvSciError err;
err = NvSciSyncObjAlloc(reconciledList, &syncObj);
if (err != NvSciError_Success) {
    goto fail;
}
/* using the object, sharing it with others */
NvSciSyncAttrListFree(reconciledList);
NvSciSyncObjFree(syncObj);
```

Inter-Application

In the cross-process case, the reconciled list and object must be shared with all the waiters.

Export/Import NvSciSyncObj

```
/* signaler */
void* objAndList;
size_t objAndListSize;
NvSciError err;
err = NvSciSyncIpcExportAttrListAndObj(
    syncObj,                        /* syncObj to be exported
(the reconciled list is inside it) */
    NvSciSyncAccessPerm_WaitOnly,   /* permissions we want the
receiver to have. Setting this to NvSciSyncAccessPerm_Auto allows
NvSciSync to automatically determine necessary permissions */
    ipcEndpoint,                    /* IpcEndpoint via which the
object is to be exported */
    &objAndList,                    /* descriptor of the object
and list to be communicated */
    &objAndListSize);               /* size of the descriptor */
/* send via Ipc */
NvSciSyncAttrListAndObjFreeDesc(objAndList);
/* waiter */
void* objAndList;
size_t objAndListSize;
err = NvSciSyncIpcImportAttrListAndObj(
    module,                         /* NvSciSyncModule use to create
original unreconciled lists in the waiter */
    ipcEndpoint,                    /* ipcEndpoint from which the
descriptor was received */
    objAndList,                     /* the desciptor of the sync obj and
associated reconciled attribute list received from the signaler */
    objAndListSize,                 /* size of the descriptor */
    &waiterAttrList,                /* the array of original unreconciled
lists prepared in the waiter */
    1,                              /* size of the array */
    NvSciSyncAccessPerm_WaitOnly,   /* permissions expected by the waiter.
Setting this to NvSciSyncAccessPerm_Auto allows NvSciSync to automatically
determine necessary permissions */
    10000U,                         /* Recommended timeout in microseconds.
Some primitives might require time to transport all needed resources. */
    &syncObj);                      /* sync object generated from the
descriptor on the waiter's side */
/* use the sync object, perhaps export it to more peers... */
NvSciSyncObjFree(syncObj);
```

Cpu Wait Contexts

NvSciSync can be used to wait on a fence from the CPU but it might require some additional resources to perform this wait. Allocating those resources is controlled by the application and encapsulated in the NvSciSyncCpuWaitContext structure. In the initialization phase a waiter allocates this structure. It can then be used to wait on any number of sync fences but it cannot be used from multiple threads at the same time:

NvSciSyncCpuWaitContext

```
/* waiter */
NvSciSyncCpuWaitContext waitContext = NULL;
NvSciError err;
/* initialize module */
err = NvSciSyncCpuWaitContextAlloc(module, &waitContext);
if (err != NvSciError_Success) {
    goto fail;
}
/* more initialization, using the context for fence waiting */
NvSciSyncCpuWaitContextFree(waitContext);
```

NvSciSyncFence Operations

After successfully allocating an object and exporting it to all the waiters, the application can proceed to the runtime phase. Typically, it is a loop where the signaler prepares its job, associates its completion with a sync fence, and shares the fence with a waiter. The waiter constructs its job in such a way that it only starts after the fence expires. Both waiter and signaler than enqueue their jobs and the use of fences establishes ordering: the waiter's job only starts when the signaler's job is complete.

NvSciSyncFence Cpu operations

```
/* signaler*/
NvSciSyncFence sharedFence = NvSciSyncFenceInitializer;
NvSciSyncFence localFence = NvSciSyncFenceInitializer; /* always
initialize with the Initializer */
NvSciError err;
err = NvSciSyncObjGenerateFence(syncObj, &localFence);
if (err != NvSciError_Success) {
    goto fail;
}
/* duplicate fence before sharing */
err = NvSciSyncFenceDup(&localFence, sharedFence);
if (err != NvSciError_Success) {
    goto fail;
}
/* create more duplicates if necessary */
/* communicate the fence to the waiter. */
/* local copy no longer necessary, so dispose of it */
NvSciSyncFenceClear(&localFence); /* this call cleans references to
sync object and is needed for proper freeing */
/* do something else, like some CPU job */
err = NvSciSyncObjSignal(syncObj);
if (err != NvSciError_Success) {
    goto fail;
}
/* waiter */
/* receive the sharedFence from the signaler */
err = NvSciSyncFenceWait(sharedFence,
        waitContext, NV_WAIT_INFINITE);
if (err != NvSciError_Success) {
    return err;
}
NvSciSyncFenceClear(sharedFence);
```

Inter-Application

The above examples assume an inter thread case but in an inter process case the fence must be exported and imported, similar to how the attribute lists were packaged.

Export/Import NvSciSyncFence

```
/* signaler*/
NvSciSyncFenceIpcExportDescriptor fenceDesc;
NvSciError err;
/* generate sharedFence */
err = NvSciSyncIpcExportFence(
    &sharedFence,      /* fence to be exported */
    ipcEndpoint,       /* should be the same ipcEndpoint used for
communicating the attribute lists */
    &fenceDesc);       /* fence descriptor has a fixed size and
is only filled in this call */
if (err != NvSciError_Success) {
    return err;
}
NvSciSyncFenceClear(&sharedFence);
/* send the descriptor via Ipc */
/* waiter */
/* receive the descriptor fenceDesc */
err = NvSciSyncIpcImportFence(syncObj,
                              fenceDesc,
                              &syncFence);
if (err != NvSciError_Success) {
    return err;
}
```

Fences are designed to be small, fixed sized objects, and interactions with them do not involve any runtime allocation. All fence structures and fence descriptors are allocated once at initialization. During runtime, NvSciSync only updates the fence and related structures, as needed.

Timestamps

NvSciSync supports timestamps in fences. They represent the time of a fence's expiration. This can help profiling the timing of streaming and debugging performance issues. This feature can be enabled during the initialization of the sync object. Then the waiter can call NvSciSyncFenceGetTimestamp to obtain the timestamp data in the streaming phase.

To enable this feature, the waiter should set NvSciSyncAttrKey_WaiterRequireTimestamps to true in its attribute list.

Waiter requires timestamp

```
bool requireTimestamps = true;
NvSciSyncAttrKeyValuePair keyValue[] = {
    ...  // other attributes
    {   .attrKey = NvSciSyncAttrKey_WaiterRequireTimestamps,
        .value = (void*) &requireTimestamps,
        .len = sizeof(bool),
    },
};
```

During reconciliation, if the signaler supports timestamp, this feature is enabled. If the waiter doesn't require timestamp, then this feature is disabled. If the waiter requires a timestamp but the signaler doesn't support it, then the reconciliation fails.

During the streaming phase, the waiter can obtain the timestamp value by calling NvSciSyncFenceGetTimestamp on an expired fence.

Waiter gets timestamp

```
uint64_t timestamp;
err = NvSciSyncFenceWait(&fence, waitContext, NV_WAIT_INFINITE);
if (err != NvSciError_Success) {
    return err;
}
err = NvSciSyncFenceGetTimestamp(&fence, &timestamp);
```

```
if (err != NvSciError_Success) {
    return err;
}
```

UMD Access

CUDA

CUDA supports `NvSciSync` by enabling applications to signal and wait for them on a CUDA stream. (Signaling an NvSciSync is similar to cudaEventRecord and waiting for an NvSciSync is similar to issuing `cudaStreamWaitEvent`). CUDA treats `NvSciSync` as an external semaphore object of type `cudaExternalSemaphoreHandleType`, which can be imported into the CUDA address space. The application can use existing cudaExternalSemaphore API to build dependencies between an `NvSciSync` object and CUDA streams, and vice-versa. Since cudaExternalSemaphore APIs are treated as regular stream operations, CUDA-NvSciSync interop follows regular stream semantics.

CUDA APIs

Query NvSciSyncObj attributes (for waiting or signaling) from CUDA

Use `cudaDeviceGetNvSciSyncAttributes` API to query the `NvSciSync` attributes from CUDA for a given CUDA device. `NvSciSyncAttrLists` passed to this API must be allocated and managed by the application.

NvSciSync object registration/unregistration with CUDA

Use `cudaImportExternalSemaphore` API to register/import an `NvSciSync` Objects into the CUDA address space. This API accepts a valid `NvSciSyncObject` as a parameter to semHandleDesc. On completion, the extSem_out returned internally holds a reference to the `NvSciSyncObject` passed earlier and must be sent to the APIs listed below.

Use `cudaDestroyExternalSemaphore` (for runtime) to unregister/destroy an already registered/imported `NvSciSyncObject` from the CUDA address space.

Wait for an NvSciSyncFence.

Use `cudaWaitExternalSemaphoresAsync` to make all operations enqueued on the CUDA stream (passed as a parameter to this API) wait until the NvSciSyncFence (sent as a parameter to this API via paramsArray) is signaled by the relevant signaler. Such a wait happens asynchronously on the GPU (i.e., the calling thread returns immediately). Applications can also optionally set flag `CUDA_EXTERNAL_SEMAPHORE_WAIT_SKIP_NVSCIBUF_MEMSYNC` to indicate that memory synchronization operations are disabled over all CUDA-NvSciBufs imported into CUDA (in that process), which are normally performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects. Use this flag when CUDA-NvSciSync is used to build only control-dependencies (i.e., no data sharing between the signaler and waiter).

Get an NvSciSyncFence.

`cudaSignalExternalSemaphoresAsync` takes a valid NvSciSyncFence as input. Upon return, the fence tracks the completion of all work submitted to the same CUDA stream on which the API was invoked. Waiting on a fence is equivalent to waiting for the completion of all the work on the stream. This API ensures that when the dependent work (in the stream) completes, the NvSciSyncFence is signaled, and any potential waiters waiting on the NvSciSyncFence are unblocked. The signal happens asynchronously in the GPU (i.e., the calling thread returns immediately). Applications can also optionally set flag `CUDA_EXTERNAL_SEMAPHORE_SIGNAL_SKIP_NVSCIBUF_MEMSYNC` to indicate that memory synchronization operations are disabled over all CUDA-NvSciBufs imported into CUDA (in that process), which are normally performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects. Use this flag when CUDA-NvSciSync is used to build only control-dependencies (i.e., no data sharing between the signaler and waiter).

`cudaWait|SignalExternalSemaphoresAsync` API takes an array of `cudaExternalSemaphore_t` and `cudaExternalSemaphoresWait|SignalParams`. This allows the application to enqueue one or more external semaphore objects, each being one of the cudaExternalSemaphoreHandleType types. This option is an efficient way to describe a dependency between a CUDA stream and more than one NvSciSyncFence as a single operation.

`cudaSignalExternalSemaphoresAsync` overwrites the previous contents of NvSciSyncFence passed to it.

CUDA-NvSciSync API Usage

```
NvSciSyncFence *signalerFence = NULL;
NvSciSyncFence *waiterFence = NULL;
NvSciIpcEndpoint signalerIpcEndpoint = 0;
NvSciIpcEndpoint waiterIpcEndpoint = 0;
NvSciSyncAttrList unreconciledList[2] = {NULL};
NvSciSyncAttrList reconciledList = NULL;
NvSciSyncAttrList newConflictList = NULL;
NvSciSyncAttrList signalerAttrList = NULL;
NvSciSyncAttrList waiterAttrList = NULL;
NvSciSyncAttrList importedWaiterAttrList = NULL;
NvSciSyncObjIpcExportDescriptor objDesc;
NvSciSyncFenceIpcExportDescriptor fenceDesc;
NvSciSyncObj signalObj;
NvSciSyncObj waitObj;
NvSciSyncModule module = NULL;
void* objAndList;
size_t objAndListSize = 0;
void* waiterListDesc;
size_t waiterAttrListSize = 0;
CUcontext signalerCtx = 0;
CUcontext waiterCtx = 0;
int iGPU = 0;
int dGPU = 1;
cudaStream_t signalerCudaStream;
cudaStream_t waiterCudaStream;
cudaExternalSemaphore_t signalerSema, waiterSema;
cudaExternalSemaphoreHandleDesc semaDesc;
cudaExternalSemaphoreSignalParams sigParams;
```

```
cudaExternalSemaphoreWaitParams waitParams;
/*****************INIT PHASE*************************/
err = NvSciSyncModuleOpen(&module);
err = NvSciIpcInit();
err = NvSciIpcOpenEndpoint("ipc_test", &signalerIpcEndpoint);
err = NvSciIpcOpenEndpoint("ipc_test", &waiterIpcEndpoint);
err = NvSciSyncAttrListCreate(module, &signalerAttrList);
err = NvSciSyncAttrListCreate(module, &waiterAttrList);
signalerFence = (NvSciSyncFence *)calloc(1, sizeof(*signalerFence));
waiterFence = (NvSciSyncFence *)calloc(1, sizeof(*waiterFence));
cudaFree(0);
cudaSetDevice(iGPU);// Signaler will be on Device-1/iGPU
cuCtxCreate(&signalerCtx, CU_CTX_MAP_HOST, iGPU);
cudaSetDevice(dGPU);// Waiter will be on Device-0/dGPU
cuCtxCreate(&waiterCtx, CU_CTX_MAP_HOST, dGPU);
cuCtxPushCurrent(signalerCtx);
cudaStreamCreate(&signalerCudaStream);
cuCtxPopCurrent(&signalerCtx);
cuCtxPushCurrent(waiterCtx);
cudaStreamCreate(&waiterCudaStream);
cuCtxPopCurrent(&waiterCtx);
cuCtxPushCurrent(waiterCtx);
cudaDeviceGetNvSciSyncAttributes(waiterAttrList, dGPU, cudaNvSciSyncAttrWait);
err = NvSciSyncAttrListIpcExportUnreconciled(&waiterAttrList, 1, waiterIpcEndpoint,
&waiterListDesc, &waiterAttrListSize);
// Allocate cuda memory for the signaler, if needed
cuCtxPopCurrent(&waiterCtx);
cuCtxPushCurrent(signalerCtx);
cudaDeviceGetNvSciSyncAttributes(signalerAttrList, iGPU, cudaNvSciSyncAttrSignal);
// Allocate cuda memory for the waiter, if needed
err = NvSciSyncAttrListIpcImportUnreconciled(module, signalerIpcEndpoint, waiterList
Desc, waiterAttrListSize, &importedWaiterAttrList);
cuCtxPopCurrent(&signalerCtx);
unreconciledList[0] = signalerAttrList;
unreconciledList[1] = importedWaiterAttrList;
err = NvSciSyncAttrListReconcile(unreconciledList, 2, &reconciledList, &newConflictL
ist);
err = NvSciSyncObjAlloc(reconciledList, &signalObj);
// Export Created NvSciSyncObj and attribute list to waiter
err = NvSciSyncIpcExportAttrListAndObj(signalObj, NvSciSyncAccessPerm_WaitOnly, sign
alerIpcEndpoint, &objAndList, &objAndListSize);
// Import already created NvSciSyncObj into a new NvSciSyncObj
err = NvSciSyncIpcImportAttrListAndObj(module, waiterIpcEndpoint, objAndList, objAnd
ListSize, &waiterAttrList, 1, NvSciSyncAccessPerm_WaitOnly, 1000000, &waitObj);
cuCtxPushCurrent(signalerCtx);
semaDesc.type = cudaExternalSemaphoreHandleTypeNvSciSync;
semaDesc.handle.nvSciSyncObj = (void*)signalObj;
cudaImportExternalSemaphore(&signalerSema, &semaDesc);
cuCtxPopCurrent(&signalerCtx);
cuCtxPushCurrent(waiterCtx);
semaDesc.type = cudaExternalSemaphoreHandleTypeNvSciSync;
semaDesc.handle.nvSciSyncObj = (void*)waitObj;
cudaImportExternalSemaphore(&waiterSema, &semaDesc);
cuCtxPopCurrent(&waiterCtx);
/****************************************************/
```

```
/*****************STREAMING PHASE**************************/
cuCtxPushCurrent(signalerCtx);
sigParams.params.nvSciSync.fence = (void*)signalerFence;
sigParams.flags = 0; //Set flags = cudaExternalSemaphoreSignalSkipNvSciBufMemSync if
 needed
// LAUNCH CUDA WORK ON signalerCudaStream
cudaSignalExternalSemaphoresAsync(&signalerSema, &sigParams, 1, signalerCudaStream);
err = NvSciSyncIpcExportFence(signalerFence, signalerIpcEndpoint, &fenceDesc);
NvSciSyncFenceClear(signalerFence);
cuCtxPopCurrent(&signalerCtx);
cuCtxPushCurrent(waiterCtx);
err = NvSciSyncIpcImportFence(waitObj, &fenceDesc, waiterFence);
waitParams.params.nvSciSync.fence = (void*)waiterFence;
waitParams.flags = 0; //Set flags = cudaExternalSemaphoreWaitSkipNvSciBufMemSync if
needed
cudaWaitExternalSemaphoresAsync(&waiterSema, &waitParams, 1, waiterCudaStream);
// LAUNCH CUDA WORK ON waiterCudaStream
cudaStreamSynchronize(waiterCudaStream);
cuCtxPopCurrent(&waiterCtx);
/***********************************************************/
/*****************TEAR-DOWN PHASE*************************/
NvSciSyncObjFree(signalObj);
NvSciSyncObjFree(waitObj);
NvSciSyncAttrListFree(reconciledList);
NvSciSyncAttrListFree(newConflictList);
NvSciSyncAttrListFree(signalerAttrList);
NvSciSyncAttrListFree(waiterAttrList);
NvSciSyncAttrListFree(importedWaiterAttrList);
NvSciSyncModuleClose(module);
NvSciIpcCloseEndpoint(signalerIpcEndpoint);
NvSciIpcCloseEndpoint(waiterIpcEndpoint);
cudaStreamSynchronize(signalerCudaStream);
cudaStreamSynchronize(waiterCudaStream);
cudaStreamDestroy(waiterCudaStream);
cudaStreamDestroy(signalerCudaStream);
cudaDestroyExternalSemaphore(signalerSema);
cudaDestroyExternalSemaphore(waiterSema);
cuCtxDestroy(signalerCtx);
cuCtxDestroy(waiterCtx);
free(signalerFence);
free(waiterFence);
/***********************************************************/
```

Sample Application

CPU Signaler Usage

```
NvSciSyncAttrList unreconciledList[2] = {NULL};
NvSciSyncAttrList reconciledList = NULL;
NvSciSyncAttrList newConflictList = NULL;
NvSciSyncAttrList signalerAttrList = NULL;
NvSciSyncModule module = NULL;
NvSciSyncObj syncObj = NULL;
```

```
NvSciSyncAttrList importedUnreconciledAttrList = NULL;
NvSciSyncFence syncFence = NvSciSyncFenceInitializer;
NvSciIpcEndpoint ipcEndpoint = 0;
NvSciSyncFenceIpcExportDescriptor fenceDesc;
void* waiterAttrListDesc;
size_t waiterAttrListSize;
void* objAndListDesc;
size_t objAndListSize;
NvSciSyncAttrKeyValuePair keyValue[2] = {0};
bool cpuSignaler = true;
NvSciSyncAccessPerm cpuPerm;
/* Initialize NvSciIpc */
err = NvSciIpcInit();
if (err != NvSciError_Success) {
    goto fail;
}
err = NvSciIpcOpenEndpoint("example", &ipcEndpoint);
if (err != NvSciError_Success) {
    goto fail;
}
/* Signaler Setup/Init phase */
/* Initialize the NvSciSync module */
err = NvSciSyncModuleOpen(&module);
if (err != NvSciError_Success) {
    goto fail;
}
/* create local attribute list */
err = NvSciSyncAttrListCreate(module, &signalerAttrList);
if (err != NvSciError_Success) {
    goto fail;
}
err = largs->fillSignalerAttrList(signalerAttrList);
if (err != NvSciError_Success) {
    goto fail;
}
cpuSignaler = true;
keyValue[0].attrKey = NvSciSyncAttrKey_NeedCpuAccess;
keyValue[0].value = (void*) &cpuSignaler;
keyValue[0].len = sizeof(cpuSignaler);
cpuPerm = NvSciSyncAccessPerm_SignalOnly;
keyValue[1].attrKey = NvSciSyncAttrKey_RequiredPerm;
keyValue[1].value = (void*) &cpuPerm;
keyValue[1].len = sizeof(cpuPerm);
err = NvSciSyncAttrListSetAttrs(list, keyValue, 2);
if (err != NvSciError_Success) {
    goto fail;
}
/* receive waiterAttrListSize; */
/* receive waiterAttrListDesc */
err = NvSciSyncAttrListIpcImportUnreconciled(
    module, ipcEndpoint,
    waiterAttrListDesc, waiterAttrListSize,
    &importedUnreconciledAttrList);
if (err != NvSciError_Success) {
    goto fail;
}
unreconciledList[0] = signalerAttrList;
unreconciledList[1] = importedUnreconciledAttrList;
/* Reconcile Signaler and Waiter NvSciSyncAttrList */
err = NvSciSyncAttrListReconcile(
    unreconciledList, 2, &reconciledList,
    &newConflictList);
if (err != NvSciError_Success) {
    goto fail;
}
/* Create NvSciSync object and get the syncObj */
err = NvSciSyncObjAlloc(reconciledList, &syncObj);
if (err != NvSciError_Success) {
    goto fail;
}
/* Export attr list and obj and signal waiter*/
err = NvSciSyncIpcExportAttrListAndObj(
    syncObj,
    NvSciSyncAccessPerm_WaitOnly, ipcEndpoint,
    &objAndListDesc, &objAndListSize);
/* send objAndListSize */
/* send objAndListDesc */
/* signaler's streaming phase */
err = NvSciSyncObjGenerateFence(syncObj, &syncFence);
if (err != NvSciError_Success) {
    return err;
}
err = NvSciSyncIpcExportFence(&syncFence, ipcEndpoint, &fenceDesc);
if (err != NvSciError_Success) {
    goto fail;
}
NvSciSyncFenceClear(&syncFence);
/* do job that the waiter is supposed to wait on */
NvSciSyncObjSignal(syncObj);
/* cleanup */
fail:
/* Free descriptors */
free(objAndListDesc);
free(waiterAttrListDesc);
/* Free NvSciSyncObj */
NvSciSyncObjFree(syncObj);
/* Free Attribute list objects */
NvSciSyncAttrListFree(reconciledList);
NvSciSyncAttrListFree(newConflictList);
NvSciSyncAttrListFree(signalerAttrList);
NvSciSyncAttrListFree(importedUnreconciledAttrList);
/* Deinitialize the NvSciSync module */
NvSciSyncModuleClose(module);
/* Deinitialize NvSciIpc */
NvSciIpcCloseEndpoint(ipcEndpoint);
NvSciIpcDeinit();
```

CPU Waiter Usage

```
NvSciSyncAttrKeyValuePair keyValue[2] = {0};
NvSciSyncModule module = NULL;
NvSciSyncAttrList waiterAttrList = NULL;
void* waiterAttrListDesc;
size_t waiterAttrListSize;
NvSciSyncObj syncObj = NULL;
void* objAndListDesc = NULL;
size_t objAndListSize = 0U;
NvSciSyncCpuWaitContext waitContext = NULL;
NvSciSyncFenceIpcExportDescriptor fenceDesc;
NvSciIpcEndpoint ipcEndpoint = 0;
bool cpuWaiter = true;
NvSciSyncAttrKeyValuePair keyValue[2] = {0};
NvSciSyncAccessPerm cpuPerm = NvSciSyncAccessPerm_WaitOnly;
err = NvSciIpcInit();
if (err != NvSciError_Success) {
    goto fail;
}
err = NvSciIpcOpenEndpoint("example", &ipcEndpoint);
if (err != NvSciError_Success) {
    goto fail;
}
/* Waiter Setup/Init phase */
/* Initialize the NvSciSync module */
err = NvSciSyncModuleOpen(&module);
if (err != NvSciError_Success) {
    goto fail;
}
err = NvSciSyncCpuWaitContextAlloc(module, &waitContext);
if (err != NvSciError_Success) {
    goto fail;
}
/* Get waiter's NvSciSyncAttrList from NvSciSync for CPU waiter */
err = NvSciSyncAttrListCreate(module, &waiterAttrList);
if (err != NvSciError_Success) {
    goto fail;
}
cpuWaiter = true;
keyValue[0].attrKey = NvSciSyncAttrKey_NeedCpuAccess;
keyValue[0].value = (void*) &cpuWaiter;
keyValue[0].len = sizeof(cpuWaiter);
cpuPerm = NvSciSyncAccessPerm_WaitOnly;
keyValue[1].attrKey = NvSciSyncAttrKey_RequiredPerm;
keyValue[1].value = (void*) &cpuPerm;
keyValue[1].len = sizeof(cpuPerm);
err = NvSciSyncAttrListSetAttrs(list, keyValue, 2);
if (err != NvSciError_Success) {
    goto fail;
}
/* Export waiter's NvSciSyncAttrList */
err = NvSciSyncAttrListIpcExportUnreconciled(
    &waiterAttrList, 1,
    ipcEndpoint,
    &waiterAttrListDesc, &waiterAttrListSize);
if (err != NvSciError_Success) {
    goto fail;
}
/* send waiterAttrListSize */
/* send waiterAttrListDesc */
/* receive objAndListDesc */
err = NvSciSyncIpcImportAttrListAndObj(
    module, ipcEndpoint,
    objAndListDesc, objAndListSize,
    &waiterAttrList, 1,
    NvSciSyncAccessPerm_WaitOnly, 10000U, &syncObj);
if (err != NvSciError_Success) {
    goto fail;
}
/* Waiter streaming phase */
/* receive fenceDesc */
err = NvSciSyncIpcImportFence(
    syncObj,
    &fenceDesc,
    &syncFence);
if (err != NvSciError_Success) {
    goto fail;
}
err = NvSciSyncFenceWait(
    &syncFence,
    waitContext, 30000U);
if (err != NvSciError_Success) {
    goto fail;
}
NvSciSyncFenceClear(&syncFence);
/* cleanup */
fail:
free(waiterAttrListDesc);
free(objAndListDesc);
NvSciSyncAttrListFree(waiterAttrList);
NvSciSyncObjFree(syncObj);
NvSciSyncCpuWaitContextFree(waitContext);
/* Deinitialize the NvSciSync module */
NvSciSyncModuleClose(module);
/* Deinitialize NvSciIpc */
NvSciIpcCloseEndpoint(ipcEndpoint);
NvSciIpcDeinit();
```

Inter-Process Communication

The NvSciIpc library provides API for any two (2) entities in a system to communicate with each other irrespective of where they are placed. Entities can be:

- In different threads in the same process.
- In the same process.
- In different processes in the same VM.

- In different VMs on the same SoC.
- In different SoCs.

Each of these different boundaries are abstracted by a library that provides unified communication (read/write) API to entities.

Terminology

Channel: An NvSciIpc channel connection allows bidirectional exchange of fixed-length messages between exactly two NvSciIpc endpoints.

Endpoint: A software entity that uses NvSciIpc channel to communicate with another software entity.

Frame: A message that consists of a sequence of bytes that is sent along an NvSciIpc channel from one of the two `NvSciIpc` endpoints of the NvSciIpc channel to the other `NvSciIpc` endpoint.

Frame Size: The size, in bytes, of every message exchanged along an NvSciIpc channel. Each NvSciIpc channel may have a distinct frame size.

Frame Count: The maximum number of NvSciIpc frames that may simultaneously be queued for transfer in each direction along an NvSciIpc channel.

Backend: An NvSciIpc backend implements NvSciIpc functionality for a particular class of NvSciIpc channels. For example, for NvSciIpc communication confined to SoCs, there are five different classes of NvSciIpc channels depending on the maximum level of separation between NvSciIpc endpoints that is supported by the NvSciIpc channel.

- INTER_THREAD: Handles communication between entities that may be in different threads in the same process.
- INTER_PROCESS: Handles communication between entities that may be in different processes in the same VM.
- INTER_VM: Handles communication between entities that may be in different VMs in the same SoC.
- INTER_CHIP: Handles communication between entities that may be in different SoCs.

Endpoint Handle: Abstract data type that is passed to all NvSciIpc channel communication.

Channel Reset: Defines the abrupt end of communication by one of the NvSciIpc endpoints. In case of reset, no communication is allowed over NvSciIpc channel until both endpoints reset their internal states and are ready for communication. NvSciIpc relies on its backend to implement the channel reset mechanism. It may require cooperation from both endpoints where two endpoints have to wait for confirmation that the other has reset its local state.

Notification: An asynchronous signal along an NvSciIpc channel through which one of the two endpoints of the NvSciIpc channel indicates to the other NvSciIpc endpoint that there may be an event for the latter NvSciIpc endpoint to process.

NvSciIpc Configuration Data

NvSciIpc configuration data is used to define all the channels present for a given Guest VM. Currently these details are provided via the device tree (DT), where each line contains details about a single NvSciIpc channel in the system.

Each channel entry is added to the DT property in string list form.

For INTER_THREAD and INTER_PROCESS backend, the format is :

`<Backend-Name>, <Endpoint-Name>, <Endpoint-Name>, <Backend-Specific-Data>,`

For INTER_VM and INTER_CHIP backend, the format is :

`<Backend-Name>, <Endpoint-Name>, <Backend-Specific-Data>,`

`<Endpoint-Name>` is unique string that is used to tag/identify a single NvSciIpc endpoint in a system. Ideally it should describe the purpose for which the NvSciIpc endpoint is created.

For INTER_THREAD and INTER_PROCESS backends, two endpoint names must be defined.

`<Backend-Name>` must be one of the following:

- INTER_THREAD
- INTER_PROCESS
- INTER_VM
- INTER_CHIP

`<Backend-Specific-Data>` may span multiple fields.

The INTER_THREAD and INTER_PROCESS backend contains two (2) integer fields that describe `<No-of-Frames Frame-Size>` tuple. `<Frame-Size>` must be multiples of 64 bytes.

INTER_VM contains a single integer field that denotes the IVC queue ID.

INTER_CHIP contains a single integer field that denotes the inter-chip device number.

Note: **For Linux VMs, the configuration data above is passed as a plain text file in the root file system. It is located at: `/etc/nvsciipc.cfg`.**

Example NvSciIpc DT node

```
{
    chosen {
        nvsciipc {
            /* NvSciIpc configuration format : string array
             *
             * INTER_THREAD/PROCESS backend case:
             * "BACKEND_TYPE" "ENDPOINT1_NAME" "ENDPOINT2_NAME"
"BACKEND_INFO1" "BACKEND_INFO2",
             *
             * INTER_VM/CHIP backend case:
             * "BACKEND_TYPE" "ENDPOINT_NAME"  "BACKEND_INFO1"
"BACKEND_INFO2",
             *
             * BACKEND_TYPE : INTER_THREAD, INTER_PROCESS, INTER_VM,
INTER_CHIP
             * For INTER_THREAD and INTER_PROCESS, two endpoints
name should be different.
             * you can use different suffix with basename for them.
             */
            compatible = "nvsciipc,channel-db";
            /* Below IPC channels are defined only for testing
and debugging purpose.
             * They SHOULD be removed in production.
             *     itc_test, ipc_test, ipc_test_a, ipc_test_b,
ipc_test_c
             *     ivc_test
             */
            nvsciipc,channel-db =
                "INTER_THREAD",  "itc_test_0",     "itc_test_1",
"64",   "1536",    /* itc_test */
                "INTER_PROCESS", "ipc_test_0",     "ipc_test_1",
"64",   "1536",    /* ipc_test */
                "INTER_PROCESS", "ipc_test_a_0",   "ipc_test_a_1",
"64",   "1536",    /* ipc_test_a */
                "INTER_PROCESS", "ipc_test_b_0",   "ipc_test_b_1",
"64",   "1536",    /* ipc_test_b */
                "INTER_PROCESS", "ipc_test_c_0",   "ipc_test_c_1",
"64",   "1536",    /* ipc_test_c */
                "INTER_VM",      "ivc_test",       "255", "0";
/* ivc_test */
            status = "okay";
        };
    };
};
```

Example NvSciIpc config file format

```
# <Backend_name>  <Endpoint-name1> <Endpoint-name2>
   <backend-specific-info>
INTER_PROCESS   ipc_test_0      ipc_test_1      64      1536
INTER_PROCESS   ipc_test_a_0    ipc_test_a_1    64      1536
INTER_PROCESS   ipc_test_b_0    ipc_test_b_1    64      1536
INTER_PROCESS   ipc_test_c_0    ipc_test_c_1    64      1536
INTER_THREAD    itc_test_0      itc_test_1      64      1536
INTER_VM        ivm_test        255
INTER_VM        loopback_tx     256
INTER_VM        loopback_rx     257
```

Adding a New Channel

Adding a New INTER_THREAD and INTER_PROCESS Channel

INTER_THREAD and INTER_PROCESS channels are implemented using POSIX shared memory, and POSIX mqueue for notifications. You must add a new line in the `/etc/nvsciipc.cfg` file describing the new channel.

Reboot the Linux VM if the added endpoint is for secure memory usecase; otherwise, there is no need to boot the Linux VM - just restart the NvSciIpc application once `nvsciipc.cfg` is updated.

INTER_THREAD and INTER_PROCESS channels are implemented using POSIX shared memory, and PULSE for notifications. You must add a new entry in the DT file describing the new channel.

The DT files for the Guest partition are:

- Guest VM 0: `drive-qnx/bsp/device-tree/qnx-device-tree/platform/t23x/common/qnx/tegra234-nvsciipc-gos0.dtsi`
- Guest VM 1(Dual-QNX): `drive-qnx/bsp/device-tree/qnx-device-tree/platform/t23x/common/qnx/tegra234-nvsciipc-gos1.dtsi`

The DT files for the Server partition are:

- `drive-foundation/platform-config/hardware/nvidia/platform/t23x/automotive/kernel-dts/server_dt/drive_av/t23x-vm-server.dts`
- `drive-foundation/platform-config/hardware/nvidia/platform/t23x/automotive/kernel-dts/server_dt/drive_av/t23x-update-service.dts`

To build the DTBs:

- `cd drive-qnx/bsp/device-tree/qnx-device-tree`
- `make BIND_ARCH_t23x=1`

NvSciIpc supports the following channel count per VM.

Backend	Maximum Channel Entry Count in DT	Description
INTER_THREAD	1000	Maximum channel count per VM
INTER_PROCESS	1000	Maximum channel count per VM
INTER_VM	512	Maximum IVC channel count for the entire DRIVE OS

In the default setting, QNX OS supports 1000 channels for Intra-VM backend (INTER_THREAD + INTER_PROCESS). To use 2000 Intra-VM channels, the system integrator should modify the QNX OS build file (such as `orin_gos_vm.build`) to increase the maximum number of file descriptors by adding `-F` options to procnto. The path to the build file is as follows:

```
drive-qnx/bsp/images/orin_gos_vm.build
```

Setting	procnto Option
default	`PATH=/proc/boot LD_LIBRARY_PATH=/proc/boot procnto-smp-instr-safety -~fl -m~xg -s -x2048`
update	`PATH=/proc/boot LD_LIBRARY_PATH=/proc/boot procnto-smp-instr-safety -~fl -m~xg -s -x2048 -F 2010`

Adding a New INTER_VM Channel

The INTER_VM channel relies on Hypervisor to set up the shared area details between two (2) VMs. At present, it is done via IVC queues that are described in the PCT. For any new INTER_VM channel:

1. Add a new IVC queue between two (2) VMs to the PCT file (`platform_config.h`) of the corresponding platform. The VM partition IDs are defined in the `server-partitions.mk` makefile. The `frame_size` value is in multiples of 64 bytes. The maximum IVC queue entries are 512 (the value imposed by the NVIDIA DRIVE® OS Hypervisor kernel). The location of the configuration file and makefile are as follows:
    - `drive-foundation/platform-config/hardware/nvidia/platform/t23x/automotive/pct/drive_av/platform_config.h`
    - `drive-foundation/virtualization/pct/make/t23x/server-partitions.mk`

    When INTER_VM IVC notification latency is critical between different PCPUs, the user can choose the MSI-based (Message Signaled Interrupt) IVC notification by adding `use_msi = 1` option in the IVC queue table. The user should contact NVIDIA to use the MSI-based IVC notification since the total MSI-based IVC channel count is limited in the NVIDIA DRIVE OS® system.
2. Update the NvSciIpc configuration data (DT) (cfg) file in both VMs. This requires adding a new entry that describes the channel information in both VMs.

    The DT files for the Guest partition are:
    - Guest VM 0:

        `drive-qnx/bsp/device-tree/qnx-device-tree/platform/t23x/common/qnx/tegra234-nvsciipc-gos0.dtsi`
    - Guest VM 1 (Dual-QNX):

        `drive-qnx/bsp/device-tree/qnx-device-tree/platform/t23x/common/qnx/tegra234-nvsciipc-gos1.dtsi`

    The DT files for the Server partition are:
    - `drive-foundation/platform-config/hardware/nvidia/platform/t23x/automotive/kernel-dts/server_dt/drive_av/t23x-vm-server.dts`
    - `drive-foundation/platform-config/hardware/nvidia/platform/t23x/automotive/kernel-dts/server_dt/drive_av/t23x-update-service.dts`

    To build the DTBs:
    - `cd drive-qnx/bsp/device-tree/qnx-device-tree`
    - `make BIND_ARCH_t23x=1`

3. If the INTER_VM channel is defined in the configuration data of DT the cfg file but its IVC queue ID is NOT available in the PCT, that channel is ignored.

Example: IVC queue table format of PCT

```
.ivc = {
.queue = {
    ... skipped ...
    [queue id] = { .peers = {VM1_ID, VM2_ID}, .nframes = ##, .frame_size = ##, .use_
msi = # },
    ... skipped ...
},
... skipped ...
}

/* example */
[255] = { .peers = {GID_GUEST_VM, GID_UPDATE}, .nframes = 64, .frame_size = 1536 },

or

[255] = { .peers = {GID_GUEST_VM, GID_UPDATE}, .nframes = 64, .frame_size = 1536, .u
se_msi = 1 },
```

NvSciIpc API Usage

Each application first has to call `NvSciIpcInit()` before using any of the other NvSciIpc APIs. This initializes the NvSciIpc library instance for that application.

`NvSciIpcInit()` must be called by the application only once at startup..

Initializing the NvSciIpc Library

```
NvSciError err;
err = NvSciIpcInit();
if (err != NvSciError_Success) {
        return err;
}
```

Prepare an NvSciIpc Endpoint for Read/Write

To enable read/write on an endpoint, the following steps must be completed.

1. The application must first create a channel ID and connection ID to get notifications from the peer endpoint.
    - If you want to receive notification events from a single peer endpoint, then create a single channel ID and attach one connection ID to that channel ID.
    - If you want to receive notification events from multiple peer endpoints through a single channel ID, then create a single channel ID and attach multiple connection IDs to that channel ID.
2. Open the endpoint.
3. Set the pulse message parameter to get event notifications from the peer endpoint.

- The application must provide the connection ID, pulse message priority, code, and data. Both code and data can be defined by the application.

4. Get the endpoint information, such as the number of frames and each frame size. This is important as only single frames can be read/written at a given time.
5. Reset the endpoint. This is important as it ensures that the endpoint is not reading/writing any stale data (for example, from the previous start or instance).

Preparing an NvSciIpc Endpoint for read/write

```
#define APP_PULSE_CODE 10
NvSciIpcEndpoint ipcEndpoint;
struct NvSciIpcEndpointInfo info;
int32_t chid;
int32_t coid;
int16_t priority = SIGEV_PULSE_PRIO_INHERIT;
int16_t code = APP_PULSE_CODE; /* application-defined code value */
int32_t *data = NULL; /* cookie data for application use or NULL */
NvSciError err;
chid = ChannelCreate_r(_NTO_CHF_UNBLOCK);
if (chid < 0) {
    goto fail;
}
coid = ConnectAttach_r(0, 0, chid, _NTO_SIDE_CHANNEL, 0);
if (coid < 0) {
    goto fail;
}
err = NvSciIpcOpenEndpoint("ipc_endpoint", &ipcEndpoint);
if (err != NvSciError_Success) {
    goto fail;
}
err = NvSciIpcSetQnxPulseParam(ipcEndpoint, coid, priority, code, (void *)data);
if (err != NvSciError_Success) {
    goto fail;
}
err = NvSciIpcGetEndpointInfo(ipcEndpoint, &info);
if (err != NvSciError_Success) {
    goto fail;
}
printf("Endpointinfo: nframes = %d, frame_size = %d\n", info.nframes, info.frame_siz
e);
NvSciIpcResetEndpoint(ipcEndpoint);
```

Writing to the NvSciIpc Endpoint

- Before writing data, connection must be established between two endpoint processes. This can be done by just calling NvSciIpcGetEvent(). There is no way to check availability of the remote endpoint process except for connection establishment. That is, NvSciIpcGetEvent() must be called first after NvSciIpcResetEndpoint().
- If the desired event type (i.e., READ, WRITE, CONN_EST) is not available from NvSciIpcGetEvent(), event blocking call (MsgReceivePulse) is used to wait for new events from a remote endpoint.
- For this reason, NvSciGetEvent() is used with MsgReceivePulse_r() in pair all the time.

- Connection events (CONN_EST, CONN_RESET) are edge-triggered. Hence, they are cleared after the reading event. On the contrary, READ and WRITE events are level triggered. As long as there is data in the receive channel buffer or there is free buffer in the transmit channel buffer, buffer status events are returned from NvSciIpcGetEvent(). READ and WRITE events are generated only when a connection is established. Therefore, READ and WRITE events can be recognized as connection establishment as well.
- Connection status transition is detected in the NvSciIpcGetEvent() context and read/write channel buffer status transition is detected in read/write NvSciIpc APIs context. These status transitions are sent to the remote endpoint process as event notification. MsgReceivePulse_r() is used to receive that kind of notification. Once this notification is received, the detail event type is identified by NvSciIpcGetEvent().
- Event blocking calls (MsgReceivePulse) with same channel ID must be called in a single thread only.
- That is, do not call MsgReceivePulse blocking calls with the same channel ID from multiple threads.
- If you want to process read events in a loop, you can add the `READ` event check routine and `NvSciIpcRead()`.

Writing to an NvSciIpc channel

```
NvSciIpcEndpoint ipcEndpoint;
struct NvSciIpcEndpointInfo info;
uint32_t event = 0;
void *buf;
int32_t buf_size, bytes;
int32_t retval;
NvSciError err;
buf = malloc(info.frame_size);
if (buf == NULL) {
    goto fail;
}
buf_size = info.frame_size;
while (1) {
       err = NvSciIpcGetEvent(ipcEndpoint, &event);
       if (err != NvSciError_Success) {
                goto fail;
       }
       if (event & NV_SCI_IPC_EVENT_WRITE) {
               /* Assuming buf contains the pointer to data to be written.
                * buf_size contains the size of data. It should be less than
                * Endpoint frame size.
                */
               err = NvSciIpcWrite(ipcEndpoint, buf, buf_size, &bytes);
               if(err != NvSciError_Success) {
                      printf("error in writing endpoint\n");
                      goto fail;
               }
       } else {
               retval = MsgReceivePulse_r(chid, &pulse, sizeof(pulse), NULL);
               if (retval < 0) {
                      goto fail;
               }
```

```
                if (pulse.code != APP_PULSE_CODE) {
                       printf("invalid pulse event code\n");
                       goto fail;
                }
         }
}
```

Reading from the NvSciIpc Endpoint

- Before reading data, connection must be established between two endpoint processes. This can be done by calling `NvSciIpcGetEvent()`.
- Basic event handling mechanism is already described in the **Error! Reference source not found.** section. Reading has the same mechanism.
- Event blocking calls (MsgReceivePulse) with the same channel ID must be called in a single thread only.
- Do not call MsgReceivePulse blocking calls with the same channel ID from multiple threads.
- If you want to process write events in s loop, you can add the `WRITE` event check routine and `NvSciIpcWrite()`.

Reading from an NvSciIpc channel

```
NvSciIpcEndpoint ipcEndpoint;
struct NvSciIpcEndpointInfo info;
uint32_t event = 0;
void *buf;
int32_t buf_size, bytes;
int retval;
NvSciError err;
buf = malloc(info.frame_size);
if (buf == NULL) {
    goto fail;
}
buf_size = info.frame_size;
while (1) {
       err = NvSciIpcGetEvent(ipcEndpoint, &event);
       if (err != NvSciError_Success) {
                goto fail;
       }
       if (event & NV_SCI_IPC_EVENT_READ) {
               /* Assuming buf contains pointer to area where frame is read.
                * buf_size contains the size of data. It should be less than
                * Endpoint frame size.
                */
               err = NvSciIpcRead(ipcEndpoint, buf, buf_size, &bytes);
               if(err != NvSciError_Success) {
                      printf("error in reading endpoint\n");
                      goto fail;
               }
       } else {
               retval = MsgReceivePulse_r(chid, &pulse, sizeof(pulse), NULL);
               if (retval < 0) {
                      goto fail;
               }
```

```
                if (pulse.code != APP_PULSE_CODE) {
                        printf("invalid pulse event code\n");
                        goto fail;
                }
        }
}
```

Prepare an NvSciIpc Endpoint for read/write

To enable read/write on an endpoint, the following steps must be completed.

1. The application must open the endpoint.
2. Get the endpoint information, such as the number of frames and each frame size. This is important as only single frames can be read/written at a given time.
3. Get the FD associated with the endpoint. This is required to handle event notifications.
4. Reset the endpoint. This is important as it ensures that the endpoint is not reading/writing any stale data (for example, from the previous start or instance).

Prepare an NvSciIpc Endpoint for read/write

```
NvSciIpcEndpoint ipcEndpoint;
struct NvSciIpcEndpointInfo info;
int32_t fd;
NvSciError err;
err = NvSciIpcOpenEndpoint("ipc_endpoint", &ipcEndpoint);
if (err != NvSciError_Success) {
    goto fail;
}
err = NvSciIpcGetLinuxEventFd(ipcEndpoint, &fd);
if (err != NvSciError_Success) {
    goto fail;
}
err = NvSciIpcGetEndpointInfo(ipcEndpoint, &info);
if (err != NvSciError_Success) {
    goto fail;
}
printf("Endpointinfo: nframes = %d, frame_size = %d\n", info.nframes, info.frame_siz
e);
NvSciIpcResetEndpoint(ipcEndpoint);
```

Writing to the NvSciIpc Endpoint

The follow example shows how to write to the NvSciIpc endpoint.

Write to NvSciIpc channel

```
NvSciIpcEndpoint ipcEndpoint;
struct NvSciIpcEndpointInfo info;
int32_t fd;
fd_set rfds;
uint32_t event = 0;
void *buf;
int32_t buf_size, bytes;
int retval;
```

```
NvSciError err;
FD_ZERO(&rfds);
FD_SET(fd, &rfds);
buf = malloc(info.frame_size);
if (buf == NULL) {
    goto fail;
}
while (1) {
       err = NvSciIpcGetEvent(ipcEndpoint, &event);
       if (err != NvSciError_Success) {
                goto fail;
       }
       if (event & NV_SCI_IPC_EVENT_WRITE) {
               /* Assuming buf contains the pointer to data to be written.
                * buf_size contains the size of data. It should be less than
                * Endpoint frame size.
                */
               err = NvSciIpcWrite(ipcEndpoint, buf, buf_size, &bytes);
               if(err != NvSciError_Success) {
                      printf("error in writing endpoint\n");
                      goto fail;
               }
       } else {
               retval = select(fd + 1, &rfds, NULL, NULL, NULL);
               if ((retval < 0) & (retval != EINTR)) {
                       exit(-1);
               }
        }
}
```

Reading from the NvSciIpc Endpoint

Read from NvSciIpc channel

```
NvSciIpcEndpoint ipcEndpoint;
struct NvSciIpcEndpointInfo info;
int32_t fd;
fd_set rfds;
uint32_t event = 0;
void *buf;
int32_t buf_size, bytes;
int retval;
NvSciError err;
FD_ZERO(&rfds);
FD_SET(fd, &rfds);
buf = malloc(info.frame_size);
if (buf == NULL) {
    goto fail;
}
while (1) {
       err = NvSciIpcGetEvent(ipcEndpoint, &event);
       if (err != NvSciError_Success) {
                goto fail;
       }
       if (event & NV_SCI_IPC_EVENT_READ) {
```

```
                /* Assuming buf contains pointer to area where frame is read.
                 * buf_size contains the size of data. It should be less than
                 * Endpoint frame size.
                 */
                err = NvSciIpcRead(ipcEndpoint, buf, buf_size, &bytes);
                if(err != NvSciError_Success) {
                       printf("error in reading endpoint\n");
                       goto fail;
                }
        } else {
                retval = select(fd + 1, &rfds, NULL, NULL, NULL);
                if ((retval < 0) & (retval != EINTR)) {
                        exit(-1);
                }
         }
}
```

Cleaning-up an NvSciIpc Endpoint

Once read/write is completed, you must free and clean the resources that were allocated by NvSciIpc endpoints.

Clean up the Endpoint

```
NvSciIpcEndpoint ipcEndpoint;
NvSciIpcCloseEndpoint(ipcEndpoint);
```

De-Initialize NvSciIpc Library

```
(void)ConnectDetach(coid);
(void)ChannelDestroy(chid);
NvSciIpcDeinit();
```

De-Initialize NvSciIpc Library

```
NvSciIpcDeinit();
```

NvSciEventService API Usage

NvSciEventService is an event-driven framework that provides OS-agnostic APIs to send events and wait for events. The framework enables you to build portable event-driven applications and simplifies the steps required to prepare endpoint connections.

Initializing the NvSciEventService Library

Each application must call `NvSciEventLoopServiceCreate()` before using any of the other NvSciEventService and NvSciIpc APIs. This call initializes the NvSciEventService library instance for the application.

`NvSciEventLoopServiceCreate()` must be called by the application only once at startup. Only single loop service is currently supported.

```
NvSciEventLoopService *eventLoopService;
NvSciError err;
err = NvSciEventLoopServiceCreate(1, &eventLoopService);
if (err != NvSciError_Success) {
        goto fail;
}
err = NvSciIpcInit();
if (err != NvSciError_Success) {
        return err;
```

Preparing an Endpoint with NvSciEventService for Read/Write

To enable read/write on an endpoint, the following steps must be completed.

1. Open the endpoint with an event service that is previously instantiated.
2. Get the event notifier associated with the endpoint that was created in Step 1.
3. Get the endpoint information, such as the number of frames and each frame size. This is important as only single frames can be read/written at a given time.
4. Reset the endpoint. This is important as it ensures that the endpoint is not reading/writing any stale data (for example, from the previous start or instance).

Note: **The event associated with an endpoint is called a native event. It is created internally when `NvSciIpcGetEventNotifier()` is called and is not visible to the application.**

Prepare an endpoint and getting an event notifier

```
NvSciEventLoopService *eventLoopService;
NvSciIpcEndpoint ipcEndpoint;
NvSciEventNotifier *eventNotifier;
struct NvSciIpcEndpointInfo info;
NvSciError err;
err = NvSciIpcOpenEndpointWithEventService("ipc_endpoint", &ipcEndpoint, &eventLoopS
ervice->EventService);
if (err != NvSciError_Success) {
        goto fail;
}
err = NvSciIpcGetEventNotifier(ipcEndpoint, &eventNotifier);
if (err != NvSciError_Success) {
        goto fail;
}
err = NvSciIpcGetEndpointInfo(ipcEndpoint, &info);
if (err != NvSciError_Success) {
    goto fail;
}
printf("Endpointinfo: nframes = %d, frame_size = %d\n", info.nframes, info.frame_siz
e);
NvSciIpcResetEndpoint(ipcEndpoint);
```

Waiting for a Single Event for Read/Write

Before reading data, a connection must be established between two endpoint processes. This can be done by calling `NvSciIpcGetEvent()`.

Basic event handling mechanism is already described in **Error! Reference source not found.** section. (The only difference is calling `WaitForEvent()` instead of `select()MsgReceivePulse_r()`).

Note:	**WaitForEvent() is an event-blocking call. It must be called from a single thread only.**

To process the write event in a loop, add the `WRITE` event check routine and `NvSciIpcWrite()`.

Wait for a single event for read

```
NvSciEventLoopService *eventLoopService;
NvSciIpcEndpoint ipcEndpoint;
NvSciEventNotifier *eventNotifier;
struct NvSciIpcEndpointInfo info;
int64_t timeout;
uint32_t event = 0;
void *buf;
int32_t buf_size, bytes;
int retval;
NvSciError err;
timeout = NV_SCI_EVENT_INFINITE_WAIT;
buf = malloc(info.frame_size);
if (buf == NULL) {
    goto fail;
}
buf_size = info.frame_size;
while (1) {
       err = NvSciIpcGetEvent(ipcEndpoint, &event);
       if (err != NvSciError_Success) {
                goto fail;
       }
       if (event & NV_SCI_IPC_EVENT_READ) {
               /* Assuming buf contains pointer to area where frame is read.
                * buf_size contains the size of data. It should be less than
                * Endpoint frame size.
                */
               err = NvSciIpcRead(ipcEndpoint, buf, buf_size, &bytes);
               if(err != NvSciError_Success) {
                      printf("error in reading endpoint\n");
                      goto fail;
               }
       } else {
               err = eventLoopService->WaitForEvent(eventNotifier, timeout);
               if(err != NvSciError_Success) {
                      printf("error in waiting event\n");
                      goto fail;
```

```
                    }
            }
}
```

Waiting for Multiple Events for Read/Write

In this scenario, multiple endpoints are opened, and multiple event notifiers are created

A connection must be established between two endpoint processes before reading the data. This can be done by calling `NvSciIpcGetEvent()`.

The event handling mechanism is similar to waiting for a single event, but it can wait for multiple events.

Check the `newEventArray` boolean array returned by `WaitForMultipleEvents()` to determine which event is notified.

Note: WaitForEvent() is an event-blocking call. It must be called from a single thread only.

To process the write event in a loop, add WRITE event check routine and `NvSciIpcWrite()`.

Here is a list of supported events:

- Multiple native events
- Multiple local events
- Multiple native and local events

Wait for multiple events for read/write

```
#define NUM_OF_EVENTNOTIFIER 2
NvSciEventLoopService *eventLoopService;
NvSciIpcEndpoint ipcEndpointArray[NUM_OF_EVENTNOTIFIER];
NvSciEventNotifier *eventNotifierArray[NUM_OF_EVENTNOTIFIER];
bool newEventArray[NUM_OF_EVENTNOTIFIER];
struct NvSciIpcEndpointInfo infoArray; /* Assuming two endpoints have the same info
*/
int64_t timeout;
uint32_t event = 0;
void *buf;
int32_t buf_size, bytes, inx;
int retval;
NvSciError err;
bool gotEvent;
timeout = NV_SCI_EVENT_INFINITE_WAIT;
buf = malloc(info.frame_size);
if (buf == NULL) {
    goto fail;
}
buf_size = info.frame_size;
for (inx = 0; inx < NUM_OF_EVENTNOTIFIER; inx++) {
    newEventArray[inx] = true;
}
while (1) {
       gotEvent = false;
       for (inx = 0; inx < NUM_OF_EVENTNOTIFIER; inx++) {
           if (newEventArray[inx]) {
               err = NvSciIpcGetEvent(ipcEndpointArray[inx], &event);
               if (err != NvSciError_Success) {
                   goto fail;
               }
               if (event & NV_SCI_IPC_EVENT_READ)) {
                   /* Assuming buf contains pointer to area where frame is read.
                    * buf_size contains the size of data. It should be less than
                    * Endpoint frame size.
                    */
                   err = NvSciIpcRead(ipcEndpointArray[inx], buf, buf_size, &bytes);
                   if(err != NvSciError_Success) {
                       printf("error in reading endpoint\n");
                       goto fail;
                   }
                   gotEvent = true;
               }
           }
       }
       if (gotEvent) {
           continue;
       }
       err = eventLoopService->WaitForMultipleEvents(eventNotifierArray,
               NUM_OF_EVENTNOTIFIER, timeout, newEventArray);
       if(err != NvSciError_Success) {
               printf("error in waiting event\n");
                      goto fail;
       }
}
```

Creating a Local Event

A local event does not require an associated endpoint. It uses two threads of a process. When creating a local event, use `NvSciEventLoopServiceCreate()` instead of `NvSciIpcInit`.

One thread called a sender is for sending a signal and the other called receiver is for waiting for the signal.

The application must first create a local event by calling `EventService.CreateLocalEvent()`. This call also creates an event notifier and associates the notifier with the local event.

Create local event

```
NvSciEventLoopService *eventLoopService;
NvSciLocalEvent *localEvent;
NvSciError err;
err = eventLoopService->EventService.CreateLocalEvent(
                    &eventLoopService->EventService,
                    &localEvent);
if (err != NvSciError_Success) {
        goto fail;
}
```

Sending a Signal with a Local Event

A sender in the same or different thread can send a signal to a receiver by calling `Signal()`.

Send a signal with local event

```
NvSciLocalEvent *localEvent;
NvSciError err;
err = localEvent->Signal(localEvent);
if (err != NvSciError_Success) {
        goto fail;
}
```

Waiting for a Local Event

A sender can send a signal to a receiver in the same or different thread.

A receiver in the same or a different thread can be notified of the signal being sent by sender.

The receiver uses `WaitForEvent` for a single signal or `WaitForMultipleEvents` for multiple signals or mixed events associated with an endpoint.

Note: **WaitForEvent() is an event-blocking call. It must be called from a single thread only.**

Here is a list of supported events:

- Single native event
- Single local events

Wait for a local event

```
NvSciEventLoopService *eventLoopService;
NvSciLocalEvent *localEvent;
NvSciError err;
int64_t timeout;
timeout = NV_SCI_EVENT_INFINITE_WAIT;
while (1) {
    err = eventLoopService->WaitForEvent(localEvent->eventNotifier, timeout);
        if(err != NvSciError_Success) {
            printf("error in waiting event\n");
            goto fail;
        }
    }
    /* Do something with the local event notified */
}
```

Cleaning Up Event Notifier and Local Event

Event notifiers for local events and for native events are deleted when they are no longer used. A local event is deleted explicitly by an application while a native event is deleted implicitly by an event notifier.

When deleting events, the application must delete the event notifier before the local event.

Clean up an event notifier and a local event

```
NvSciEventNotifier *nativeEventNotifier; /* Assume
this is event notifier for native event */
NvSciLocalEvent *localEvent;
nativeEventNotifier->Delete(nativeEventNotifier);
local->eventNotifier->Delete(local->eventNotifier);
local->Delete(local);
```

De-Initializing NvSciEventService Library

The application must call `EventService.Delete()` after de-initializing the NvSciIpc library.

De-initialize NvSciEventService library

```
NvSciIpcEndpoint ipcEndpoint;
NvSciEventLoopService *eventLoopService;

NvSciIpcCloseEndpoint(ipcEndpoint);
NvSciIpcDeinit();

eventLoopService->EventService.Delete
(&eventLoopService->EventService);
```

NVSCI APIs on Jetson Linux

Release Properties

- NvStreams, NvSciIpc, and NvSciEvent libraries are included in the Jetson Linux BSP.
- NvStreams, NvSciIpc, and NvSciEvent headers and NvSciIpc configuration file are packaged in `nvsci_headers.tbz2`. `nvsci_headers.tbz2` is packaged in `public_sources.tbz2`.

Differences for NvStreams and NvSciIpc on Jetson Linux

`NvSciIpc` and `NvSciEvent`

The IVM back end is not supported in Jetson Linux.

API Reference for Software Communication Interfaces

Refer to the NVIDIA DRIVE OS Linux SDK API Reference for information regarding Software Communication Interfaces.