Argus 0.99.3 API Specification

March 16, 2023

Contents

List of Tables

1 Introduction

Argus is an API for acquiring images and associated metadata from cameras. The fundamental operation is a *capture*: acquiring an image from a sensor and processing it into a final output image.

Currently, Argus is supported on Android and all Jetson Linux platforms.

Argus is designed to address a number of fundamental requirements:

- Support for a wide variety of use cases (traditional photography, computational photography, video, computer vision, and other application areas.) To this end, Argus is a *frame-based* API; every capture is triggered by an explicit request that specifies exactly how the capture is to be performed.
- Support for multiple platforms, including L4T and Android.
- Efficient and simple integration into applications and larger frameworks. In support of this, Argus delivers images in one of two ways:
  - EGLStreams, which are directly supported by other system components such as OpenGL, Cuda, and GStreamer. EGLStreams manage the allocation and lifespan of all buffers, and they are passed between Argus and the consumer directly such that no buffer copies are required during delivery to the consumer.
  - Buffer OutputStreams, which are created and managed by the client in order to wrap native buffer resources as Buffer objects that are used as the destination for capture requests.
- Expansive metadata along with each output image.
- Support for multiple sensors, including both separate control over independent sensors and access to synchronized multi-sensor configurations. (The latter is unsupported in the current release.)
- Support for multiple simultaneous clients and system-wide camera access and permission control via a system-level daemon (i.e., nvargus-daemon).
- Version stability and extensibility, which are provided by unchanging virtual interfaces and the ability for vendors to add specialized extension interfaces.

Argus provides functionality in a number of different areas:

- Captures with a wide variety of settings.
- Optional autocontrol (such as auto-exposure and auto-white-balance.)
- Multiple options for metadata delivery via events, output streams, or EGLStream metadata.
- Image post-processing such as noise reduction and edge sharpening.
- Notification of errors, image acquisition start, and other events via synchronous event queues.
- Wide range of support for consuming output images via zero-copy EGLStream consumers or direct access to application-allocated and managed native buffer resources.

- Support for offline processing of Bayer raw image frames along with capture metadata to produce ISP processed images that binary match with live sensor capture.

Functionality *not* provided by Argus:

- Auto-focus. *(Will be added in a later release.)*
- Reprocessing of YUV images (such as that required by Android's Zero Shutter Lag feature.)

2 Fundamentals

2.1 Types

Argus/Types.h defines fundamental enumerations, data types, and classes used in the API.

2.1.1 Enumerations

Standard integer enumerant values can be unsuitable for an extensible API like Argus due to the possibility for conflicts or overlap when new values are added by multiple vendors. Therefore, the use of enums in Argus is limited to the **Status** return code as well as for constant types that are guaranteed not to be extended (for example, the X/Y coordinates of a 2D point).

Status
: Reports the result of an Argus operation. `STATUS_OK` signifies a successfully completed operation. All other return values are errors and should be handled appropriately.

BayerChannel
: Identifies the color channels of a Bayer image.

RGBChannel
: Identifies the color channels of an RGB image.

Coordinate
: Identifies the coordinates of a 2D or 3D point.

2.1.2 UUIDs

Besides the standard integer enumerants listed above, 128-bit Universally Unique Identifiers (UUIDs) are used to identify all other settings and interfaces in Argus. Since UUIDs are virtually guaranteed not to conflict, this allows for new settings and enumerant values to be added by multiple vendors or extensions without the potential for conflicts.

All constant UUIDs that are used to identify object types or setting values extend the **NamedUUID** subclass, which includes a string value along with the UUID value itself. These named UUID types ensure type safety, and the value string is helpful for debugging purposes.

Examples of key **NamedUUID** types include:

ExtensionName
Identifies an extension to the Argus API; used with **supportsExtension()** to determine runtime extension support.

InterfaceID
Identifies an Interface; used with **getInterface()**.

PixelFormat
Defines a pixel format that may be used for an OutputStream.

SensorModeType
Defines the type of image data that is output by the imaging sensor before any sort of image processing.

SensorPlacement
Defines the placement of the sensor on the module.

BayerPhase
Defines the phase of input raw bayer stream.

PixelFormatType
Defines the output format type of CVOutput. CVOutput is another output from argus apart from ISP processed main output which can be used in computer vision based applications. Supported CVOutput format is RGB Linear (no tonecurve and no gamma applied). CVOutput can be enabled separately or in conjunction with main ISP processed output.

CVOutput
Defines if CVOutput is linear or non-linear, only linear is supported.

Other **NamedUUID** types are used to control capture settings, such as **DenoiseMode**, and are documented by their settings interfaces.

2.1.3 Data Types

Most of the data types that Argus provides extend from the **Tuple** template class, which provides a finite ordered list of typed elements and provides type safety and named data accessors using template specializations:

BayerTuple
Provides the 4 components of a Bayer quad (identified by the **BayerChannel** channels).

RGBTuple
Provides the 3 components of an RGB pixel (identified by the **RGBChannel** channels).

Point2D
Provides the X and Y coordinates of a 2-dimensional point (identifed by **Coordinates**).

Point3D
Provides the X, Y and Z coordinates of a 3-dimensional point (identifed by **Coordinates**).

Size2D
Provides the width and height of a 2-dimensional size.

Rectangle
Provides the bounding box of a rectangle (left, top, right, and bottom coordinates).

Range
Provides the minimum and maximum values allowed for a range.

AcRegion
Provides an autocontrol region of interest by extending **Rectangle** and adding a floating-point weight value.

Additional non-Tuple data types defined by Argus include:

Array2D
Provides data storage and member access methods for a 2-dimensional array.

2.1.4 Base Classes

Argus objects and interfaces depend on and extend a few key base classes:

NonCopyable
A class that overrides the standard copy operator and disables the inheriting class from being copied. All Argus objects and interfaces are non copyable.

Interface
The base class inherited by all Interfaces, and the base type returned by **getInterface()**.

InterfaceProvider
The base class for any Argus object which provides Interfaces (i.e., implements **getInterface()**).

Destructable
The base class for any Argus object which must be destroyed by the client (i.e., implements **destroy()**).

2.2 Timestamps

Unless otherwise stated, all timestamp values returned by Argus will be uint64_t values representing the number of nanoseconds since the start of the system's monotonic clock (i.e., **CLOCK_MONOTONIC**).

Similarly, all timeout parameters passed to Argus by the client must be provided in nanoseconds. The constant **TIMEOUT_INFINITE** may be used to indicate an infinite timeout.

2.3 Objects and Interfaces

Argus makes a distinction between *objects* and *interfaces*:

- An object is an independent API entity with a well-defined lifetime but no methods specific to its type.
- An interface is a pure virtual class that the client acquires from an object, and uses to perform specific operations on the object that provided it.

Each interface has an associated UUID – a 128-bit unique identifier that is assigned to that interface. These UUIDs will never change. These UUIDs are defined with the **InterfaceID** type, a subclass of **NamedUUID**. They are used to acquire interfaces from objects (see **getInterface()** below). As a convenience, each interface also defines a static **id()** method that returns the UUID for that interface.

Every interface inherits from the **Interface** base class, which defines no public methods but ensures that subclasses are not copyable or assignable.

Every object inherits from the **InterfaceProvider** virtual base class, which defines the **getInterface()** method:

```
virtual Interface* getInterface(const InterfaceID& id) = 0;
```

The client acquires interfaces using this method. If the object supports the requested interface, this method will return a pointer to an instance of that interface; otherwise it returns `NULL`. An example using the **Event** object and its **IEvent** interface:

```
uint64_t getEventTime(Event* evt) {
    Interface* if = evt->getInterface(IEvent::id());
    IEvent* ievt = static_cast<IEvent*>(if);
    return ievt ? ievt->getTime() : 0ULL;
}
```

The convenience function **interface_cast<>()** calls **getInterface()** on the object provided and returns a pointer to the interface type specified as the template argument (or `NULL` if the object is `NULL`, or the interface cannot be acquired). This example illustrates its use:

```
uint64_t getEventTime(Event* evt) {
    IEvent* ievt = interface_cast<IEvent>(evt);
    return ievt ? ievt->getTime() : 0ULL;
}
```

2.4 Versioning

Argus version compatibility is managed with one simple rule: Once an interface has been released, it will never change. It may eventually become deprecated, and no longer be available at runtime, but the signatures in the interface will not change.

Important note: Interface immutability is not guaranteed for beta versions of Argus. Beta version numbers begin with zero; for example, Release 0.91.

2.5 Object Lifetimes

The lifetime of any Argus interface is the same as the lifetime of the object providing that interface. The client takes no explicit action to release or destroy an interface.

The lifetime of an Argus object depends on whether or not the object inherits from the **Destructable** base class, which declares the **destroy()** method:

```
virtual void destroy() = 0;
```

If an Argus object inherits from **Destructable**, the client must call **destroy()** when it is finished using the object. After that call, the object (and all interfaces acquired from it) are no longer valid. The implementation is free to immediately destroy the object, or to defer destruction. For some objects (in particular, **CaptureSession**), **destroy()** may block until associated operations are complete.

Argus objects that do not inherit from **Destructable** are generally provided by and share the lifespan of a parent Argus object, and therefore they are often referred to as *child objects*. One example of a child object is the **CaptureMetadata** object that may be attached to an **Event**: this metadata object will remain valid throughout the life of the **Event** that returned it. The use and lifespan of all child objects will be documented by the API methods that provide them.

2.5.1 UniqueObj Smart Pointer

While clients are free to explicitly call **destroy()** on every **Destructable** object that they create, the recommended way to manage **Destructable** objects is through the use of the **UniqueObj** smart pointer template. This class mimicks the use of **std::unique_ptr**, offering a movable smart pointer which calls **destroy()** on the **Destructable** object being referenced by the pointer once it leaves scope.

The **interface_cast** method is overloaded to accept **UniqueObj** pointers, allowing clients to do the following:

```
{
    UniqueObj<Request> request(iSession->createRequest());
    IRequest *iRequest = interface_cast<IRequest>(request);
    // request is destroy()ed when it leaves scope.
}
```

2.6 Extensions

Due to the exclusive use of Interfaces, Argus is inherently extensible by nature: new interfaces can be defined and exposed by an Argus implementation as needed while maintaining backwards compatibility with applications that do not use the extension. Extensions will often introduce new objects or interfaces to the API, but they are not required to do so; an extension may simply relax previous restrictions on the API and allow behavior that was previously disallowed. For example, an extension may just add a new **PixelFormat** UUID that can be used with any method accepting **PixelFormat** values. All extensions added to Argus must include an **ExtensionName** identifier

which is used by the **ICameraProvider::supportsExtension()** method to query the existence of an extension in an Argus implementation. This allows a client to check for required extension support before creating any **CaptureSessions**. Note that support for an extension does not imply that the hardware or resources used by the extension are available; standard interface checking and other extension-specific runtime checks, as described by the extension documentation, should always be performed before any extension is used.

2.7 Capture Sessions

Argus is a capture-based API, meaning that a client must make explicit capture requests to receive output from the sensor(s). The client uses **CaptureSession** objects to make these requests. A capture session is bound to one or more sensors, and each sensor can be bound to only one capture session.

2.7.1 Capture Methods

The **ICaptureSession** interface defines four capture methods:

capture()	A standard single capture call; it produces one output on each of the streams enabled in the request.
captureBurst()	Like **capture()**, this will create a single output for the streams enabled in each request, but multiple independent requests can be specified as a vector to this call. (See further explanation below.)
repeat()	Equivalent to calling **capture()** repeatedly until **stopRepeat()** is called.
repeatBurst()	Equivalent to calling **captureBurst()** repeatedly until **stopRepeat()** is called.

All four of these calls require one or more capture requests, which must be configured prior to issuing the captures. The client creates a **Request** by calling **ICaptureSession::createRequest()**, and configures it via the available interfaces. (The client can change or delete a **Request** without affecting any earlier captures that used it.)

Each **Request** object exports an **IRequest** interface which is used to set capture settings: output streams, stream settings, autocontrol settings, and source settings. Likewise the client must create an output **Stream** object(s) with **ICaptureSession::createOutputStream()**. More than one output stream can be created and used with the Session.

The signature for **capture()** looks like this:

```
uint32_t capture(const Request* request,
                 uint64_t timeout = TIMEOUT_INFINITE,
                 Status* status = NULL);
```

A single call to **capture()** will capture a single frame, using the settings and output stream(s) specified in the request. If too many captures are pending, Argus will block until there is enough space for the new capture, or set ***status** to `STATUS_TIMEOUT` if the timeout period is exceeded. The

return value from **capture()** is a *capture id*, unique within the session, which will be included with all events and other outputs from this capture. The capture id starts at 1 and will be incremented by one from each successfully submitted capture request to the next. If this counter ever exceeds the uint32_t limits, it will wrap back to 1. This counter is shared between all other capture request methods, and so the capture IDs will be unique and incremental across all submission methods described below. If the call fails for any reason, the return value will be zero and the error status code will be written to **status**.

Burst captures take a list of **Requests** instead of just one. Each call to **captureBurst()** will result in N captures, where N is the number of **Requests** in the **requests** parameter:

```
uint32_t captureBurst(const vector<const Request*>& requests,
                      uint64_t timeout = TIMEOUT_INFINITE,
                      Status* status = NULL);
```

The first capture will be performed using the first item in **requests**, the second capture using the second item in **requests**, and so on. The return value from **captureBurst()** is the capture id of the first capture. The second capture will be assigned a capture id of (return value + 1), and so on. In effect, a successful burst capture request will increment the counter by N. If the call fails for any reason, such as exceeding the maximum number of burst requests, the return value will be zero and the error status code will be written to **status**.

The number of requests in a burst can be no more than the value returned by **ICapture Session::maxBurstRequests()**.

The requests used in a burst can have any properties, but best results may be achieved by following a few guidelines:

- Use the same sensor mode for every request. Otherwise, there may be large performance delays every time the sensor mode changes, resulting in dropped frames and lower overall frame rate.
- Use the same high-level autocontrol settings (for example, whether auto-exposure is enabled) for all requests.
- Use requests that were created with the same **CaptureIntent** if possible.

Calls to **repeat()** and **repeatBurst()** return the capture id of the first capture submitted. Both of these methods will continually capture frames until **stopRepeat()** is called. Even after **stop Repeat()** has been called, captures that have already been submitted will continue to be processed, so the application should still expect events and output frames to be delivered from those captures. Repeating capture is automatically stopped when a critical error event is received.

2.7.2 Capture Timing and Interactions

All of the capture methods are blocking calls, returning as soon as the capture request is accepted by the underlying driver. Call durations will vary based on the number of captures in the system and camera pipeline state.

When a repeating capture is in effect, the client may still call another capture method. If the new method is one of the repeating capture methods, it will replace the current repeating capture method, but captures already being processed from the earlier method will still be completed. If the new method is not a repeating capture method, it will be inserted into the stream of repeating captures. The timing of that inserted capture is not guaranteed; for example, other captures from the repeating sequence may be submitted before the new call completes. However, capture bursts will never be interrupted by other capture calls. Once a burst request begins processing, all the requests in that burst will be handled before any other captures occur.

2.8 Output Streams

All capture requests submitted to Argus must write their capture output to one or more **Output Stream** objects created by the **CaptureSession**. There are two types of **OutputStream** objects:

2.8.1 Buffer Streams

Identified with the **STREAM_TYPE_BUFFER** StreamType, a Buffer Stream is an **Output Stream** that wraps a set of client-allocated and managed native buffer resources as **Buffer** objects within the stream. These **Buffer** objects are acquired from the **OutputStream** when they have been written by a capture request and made available to the client, and then released back to the stream when the Buffer may be reused for another capture.

Buffer objects point directly to the underlying data store of the native buffer being wrapped, and so capture results will be written directly to the buffer allocation without requiring another copy post-capture.

Since the application is responsible for allocating and populating the set of **Buffers** to use in a stream, it is also the client's responsibility to ensure that there are enough Buffers allocated to maintain a capture pipeline depth that is deep enough to prevent frame drops. This buffer count is often dependent on many variables, from image size to overall system load, and generally requires experimentation to find the optimal number of buffers.

To further help with efficient hardware pipelining, **Buffer** objects may optionally support sync primitives in order to allow buffers to be passed between Argus and the client with sync information to block on any pending buffer reads and/or writes before proceeding. For example, a **Buffer** object may be acquired from Argus before the image data has been fully written to the buffer, but an acquire sync will be made available to the client to wait on before it may access the data. Similarly, the client can provide a release sync when the **Buffer** is released to block the Argus capture writes on a pending client operation.

2.8.1.1 Buffer Types

Buffer objects that are created in a buffer stream wrap native buffer resources and are configured using an **OutputStreamSettings** object, so an Interface to **OutputStreamSettings** must exist for each native resource type that can be wrapped by a **Buffer**. Due to the platform-specific nature

of native buffer resources, however, the only native buffer resources that can be used with the core Argus API are EGLImages. Any other native buffer types must be supported through the use of platform-specific extensions and interfaces (of which there are currently none).

BUFFER_TYPE_EGL_IMAGE is currently the only supported **BufferType**. When a buffer output stream is using the **BUFFER_TYPE_EGL_IMAGE** type, the **BufferSettings** objects that the stream returns from **createBufferSettings()** will expose the **IEGLImageBufferSettings** interface; this interface is used to set the handle of the EGLImage that will be wrapped by the new **Buffer** object that is created when the settings are then passed to **createBuffer()**.

Buffer objects that wrap EGLImage resources act as EGLImage siblings, and Argus captures that write to these **Buffers** will write directly to the EGLImage data store. Note, however, that other EGLImage siblings may need to be invalidated once the capture is complete in order to reference the new image data. For example, an EGLImage may need to be rebound to a GL texture object to invalidate any previous cached texture data. Details on the use of EGLImages outside of the Argus API are outside of the scope of this document; documentation may be found in the Khronos EGL registry: https://www.khronos.org/registry/egl/.

Note that there is currently no mechanism to determine which EGLImage image formats will be compatible with Argus until the Buffer has actually been created from the EGLImage. Once the Buffer has been created, it can be passed to **supportsOutputStreamFormat()** to check that the buffer is compatible before continuing to submit capture requests.

2.8.1.2 Sync Types

Sync types generally have the same platform-specific requirements as native buffer types, and so the only **SyncType** that is currently supported is **SYNC_TYPE_EGL_SYNC**. The use of this sync type means that **Buffer** objects acquired and released from the stream will have EGLSync objects attached which may be used to synchronize image reads/writes with any API that supports EGLSync objects. The EGLSync object may be (EGL_NO_SYNC) if no sync is required for that frame.

Note that sync support is optional, and the default **SyncType** is **SYNC_TYPE_NONE**. In this case, all buffer reads and writes must be complete before the **Buffer** is released or acquired, respectively.

2.8.2 EGLStreams

Argus supports image output to EGLStreams by assuming the role of an EGLStream producer, which is provided by the **STREAM_TYPE_EGL** stream type. EGLStreams facilitate simple and efficient transfer of image buffers between EGLStream-enabled APIs, and have been extended by numerous extensions to further enhance their utility beyond the basic stream features. The majority of the EGLStream API and documentation is maintained by Khronos registry and is outside the scope of this document [see https://www.khronos.org/registry/egl/]

Configuring an EGLStream output stream requires the use of the **IEGLOutputStreamSettings** interface to set the pixel format, resolution, and EGLDisplay for the stream. Once the **Output**

Stream object has been created, **IEGLOutputStream::getEGLStream()** will then return the newly created EGLStream handle which may be used to connect the EGLStream consumer.

Once an EGLStream is fully connected to an **OutputStream**, each capture request outputting to this stream will output a new frame to the EGLStream. These frames can then be acquired by one of the many EGLStream consumer endpoints that can be attached to the stream, including those outside the Argus namespace. These may include endpoint APIs such as OpenGL, CUDA, and GStreamer. How these consumers acquire and release frames from the stream are documented by their respective specifications.

Note that the size of the EGLStream buffer pool as well as all buffer data synchronization is handled internally by the EGLStream.

2.8.2.1 EGLStream Utility Namespace

In addition to the EGLStream consumer endpoints that already exist within the Khronos registry, Argus introduces an EGLStream namespace and **Consumer** class which provides various interfaces to acquire and read frames and image data directly from an EGLStream. **Consumer** objects are created using the static **Consumer::create()** method, which allows the EGLStream namespace and **Consumer** objects to be used without a **CameraProvider** . It is also not a requirement that the producer endpoint be connected to Argus; the **Consumer** object can be used in conjunction with any EGLStream producer. The core interfaces supported by a **Consumer** object are:

IStreamConnection
: Provides controls to connect/disconnect from the stream.

IFrameConsumer
: Provides methods to acquire/release frames and Frame objects.

The **IFrameConsumer::acquireFrame()** method returns **Frame** objects corresponding to frames acquired from the EGLStream. These **Frames** are valid until they are released, either explicitly or implicitly, and contain the frame metadata and image buffer. The interfaces exposed by a **Frame** include:

IFrame
: Exposes the core metadata (frame number and timestamp) and **Image** contained in the frame.

IFrameCaptureMetadata
: When connected to an Argus producer, classnameFrames may expose this interface to provide access to the frame's corresponding **CaptureMetadata**

Finally, the **IFrame::getImage()** method returns an **Image** object corresponding to the image buffer(s) included with an EGLStream frame. The format of the data contained in an **Image** is identified with a unique **ImageFormatID** and is described, accessed, and read using the following interfaces:

IImageBuffers Allows mapping of the image buffers for CPU read access.

IImage2D Provides the dimensions of a 2D image

IImageYUV Describes the plane/channel layout/size of YUV formats.

IImageRGBA Describes the channel layout/size of RGBA formats.

IImageBayer Describes the channel layout/size of Bayer formats.

IImageJPEG Encodes and writes the image to disk as a JPEG file.

2.8.2.2 EGLStream Buffer Formats

According to the EGL_KHR_stream specification,

> "It is the responsibility of the producer to convert the images to a form that the consumer can consume. The producer may negotiate with the consumer as to what formats and sizes the consumer is able to consume, but this negotiation (whether it occurs and how it works) is an implementation detail."

There is currently no automatic format negotiation between Argus and any consumers, and it is the responsibility of the application to select an Argus pixel format that is compatible with the consumer. At this point in time, however, there is also no mechanism within Argus for an application to query format compatibility with a Consumer before connection. This will be added sometime before Argus leaves the Beta state, but until then please refer to the implementation release notes for details on buffer format compatibility.

2.9 Input Streams

Argus supports reading Bayer raw image frames from disk or user provided raw buffer instead of a physical camera sensor and using the read frames as input to the Image Signal Processor. To do this, the client must explicitly enable reprocessing in the capture requests that it issues. One or more **InputStream** objects created by the **CaptureSession** are used as the source streams for such capture requests. The reprocessed frames are written to one or more **OutputStream** objects created by the **CaptureSession**. An **InputStream** is based on an EGLStream which has Argus as its consumer and for such streams, the stream type must be specified as **STREAM_TYPE_EGL** when creating the corresponding **InputStreamSettings** object through the **CaptureSession**. An **InputStream** uses a **CameraDevice** object as the capture source for the stream, which can be set through the corresponding **InputStreamSettings** object. The **IReprocessInfo** interface supported by the **CameraDevice** object should be used to set and access different parameters that are used for reprocessing, such as the color format , pixel bit depth and resolution.

Argus provides the **FrameProducer** and **FrameBuf** classes as part of the EGLStream Utility Namespace to facilitate insertion of frames to the underlying EGLStream of an **InputStream**. These are briefly described below:

FrameProducer
: Acts as producer point to an **InputStream** or EGLStream provided during creation. The **FrameProducer** is responsible for allocating and managing **FrameBuf** objects that are used in the stream. Supports the **IFrameProducer** interface which provides methods to present/get frames to/from the EGLStream.

FrameBuf
: Wraps raw frame data and associated metadata corresponding to a stream frame. Supports the **IFrameBuf** interface.

The **IFrameBuf::loadInputImageFromFile(cons char * fileName)** method reads frame data from the file specified by the filename provided as the argument, to the underlying buffer.

2.10 Events

Argus uses events as the mechanism to notify applications of driver state changes. Applications use events by creating and waiting on Argus event queues.

Events are available from any object that exposes an **IEventProvider** interface. This interface provides methods to:

- List the supported events with **getAvailableEventTypes()**.
- Create event queues with **createEventQueue()**.
- Wait for the events with **waitForEvents()**.

In this release of Argus, capture sessions are the only event providers.

When creating event queues, applications can request a specific event, or they can provide a list of events that the queue will contain. Multiple queues can be created for a single object. For example, an application can create one event queue that will wait for error messages and another that will wait for capture complete events.

In order to wait for an event, the application calls **waitForEvents()**. This can be done with one or more event queues. **waitForEvents()** will block until at least one event is available; once available, the corresponding queue will have the event(s) copied to it and **waitForEvents()** will return. If there are any outstanding events at the time the application calls **waitForEvents()**, they will be copied immediately and the method will return. In the case of multiple queues registered for the same event, the queue with the lowest index will receive the event.

Once **waitForEvents()** returns, the application can look at the event objects via **getEvent()**. The event objects and any data they possess are valid until the event queue is destroyed or the event queue is again passed to **waitForEvents()**, at which point the queue is cleared and the objects invalidated. Event objects expose the **IEvent** interface, which allows the application to:

- Get the event type with **getEventType()**.
- Get the time of the event in nanoseconds with **getTime()**.
- Get the frame id this event is associated with **getFrame()**.

To get data for a specific event, the client should query the Event object for that event's type interface using **getInterface()**. This will allow the application to get event-specific data such as the metadata in a capture complete event, or the specific error status in an error event.

2.11 Multiple Clients and Multiple Threads

Argus supports multiple simultaneous client processes through the use of a system-level service (i.e., nvargus-daemon) that manages camera resources and provides connections to the Argus API for each client that wishes to connect. Each sensor can be bound to a session in only one process at a time; that is, no sensor can be simultaneously used by more than one process.

Within an Argus application, all captures on a single session must be performed by a single thread, and **waitForIdle()** calls must also be made on this thread. (See the **CaptureSession** section below for more information on **waitForIdle()**.) Captures on different sessions can be performed by separate threads, and other threads can be used for non-capture operations such as querying event queues and setting up new **Request** objects.

Additional application threads are expected to service events. Since **IEventProvider::waitForEvents()** calls are blocking, an application should usually provide one or more additional threads to wait for events. It is recommended that these event threads only handle event/metadata logic and that, as mentioned above, a single thread be used to control captures.

3 Argus Objects and Interfaces

3.1 CameraProvider

The **CameraProvider** object is the core Argus object which provides access to the cameras in the system along with capture session creation methods. It is the first Argus object that should be created. The Argus entry point is thus the static method **CameraProvider::create()** , which creates and returns the single instance of the **CameraProvider** object. (A second call to **create()** will fail.)

Supported Interfaces:

ICameraProvider
: Provides methods to query the **CameraDevices** available in the system and methods to create **CaptureSession** objects utilizing these **CameraDevices**.

3.2 CameraDevice

The **CameraDevice** object represents a camera in the system. A **CameraDevice** object can be a one-to-one mapping to a physical camera device, or it can be a virtual device that maps to multiple physical devices that produce a single image.

All of the **CameraDevice** objects present in the system are enumerated using **ICameraProvider::getCameraDevices()**. A **CameraDevice** object can be used to query the capabilities of the camera sensor (or other imaging device) it represents, and is provided to **createCaptureSession()** to create a **CaptureSession** that may issue capture requests to that device. A **CameraDevice** can be used by only one **CaptureSession** at any point in time; attempting to create a new session using a device that is already in use by another session will

fail. Every **CameraDevice** object supports the **ICameraProperties** interface, which describes the capabilities of that device:

Table 1: ICameraProperties

Name	Type	Description
SensorPlacement	SensorPlacement	Camera sensor placement position on the module
MaxAeRegions	uint32_t	Maximum number of AeRegions supported
MinAeRegionSize	Size	Minimum size of resultant AeRegion supported
MaxAwbRegions	uint32_t	Maximum number of AwbRegions supported
MaxAfRegions	uint32_t	Maximum number of AfRegions supported
BasicSensorModes	vector<modes >	Basic Sensor modes supported by this device that do not have extensions
AllSensorModes	vector<moes >	All Sensor modes supported by this device including the ones that have extensions
AperturePositions	vector<positions>	Returns all the recommended aperture positions
AvailableApertureFNumbers	vector<fnumbers >	Returns all the available aperture f-numbers
FocusPositionRange	Range<int32_t>	Range of valid focuser positions
AperturePositionRange	Range<float>	Returns the range of supported apertures positions
ApertureMotorSpeedRange	Range<float>	Returns the valid aperture speed range in step positions per second units
IspDigitalGainRange	Range<float>	Returns the supported range of ISP digital gain
ExposureCompensationRange	Range<float>	Returns the supported range of Exposure Compensation
ModelName	const std::string&	Model name of the camera device
ModuleString	const std::string&	Module string of the camera device, unique per each device on a system

Supported Interfaces:

ICameraProperties

Provides methods to query the properties and capabilities of a **CameraDevice**. These include but are not limited to: available sensor modes, focal range, aperture range, and autocontrol region limits.

IReprocessInfo

Provides methods to set and access reprocessing information to run camera using a user provided raw buffer instead of a physical camera sensor. The reprocessing information includes a flag tracking whether the camera device operates in reprocessing mode or not. Apart from this, the information includes parameters defining the reprocessing sensor mode, namely; resolution, scaling, crop rectangle, frame rate, color format and pixel bit depth.

3.3 SensorMode

The **SensorMode** object provides information about a single mode supported by the sensor. There are two types of valid **SensorModes**. The first type called basic **SensorMode** is a **SensorMode** that does not have an associated extension. Basic **SensorMode** types include Depth, RGB, YUV and Bayer types. The list of valid basic **SensorModes** is available from **ICameraProperties::getBasicSensorModes()**. The second type called extended **SensorMode** is a **SensorMode** that has extensions associated with it. The extended **SensorMode** supports some form of Wide Dynamic Range (WDR) technology. The extensions provided by this type of **SensorMode** give access to the features of the concerned WDR technology. The full list of valid **SensorModes**, both basic and extended, is available from **ICameraProperties::getAllSensor Modes()**. Every **SensorMode** object, whether basic or extended, exposes the **ISensorMode** interface, which provides the following information:

Table 2: ISensorMode

Name	Type	Description
Resolution	Size	Width and height of sensor mode
ExposureTimeRange	Range<uint64_t>	Valid range for exposure time in this mode (in nanoseconds)
FrameDurationRange	Range<uint64_t>	Valid range for frame duration in this mode (in nanoseconds)
AnalogGainRange	Range<float>	Valid range for analog gain in this mode
InputBitDepth	uint32_t	The bit depth of the image captured by the image sensor in the current mode. For example, a wide dynamic range image sensor capturing 16 bits per pixel would have an input bit depth of 16.
OutputBitDepth	uint32_t	The bit depth of the image returned from the image sensor in the current mode. For example, a wide dynamic range image sensor capturing 16 bits per pixel might be connected through a Camera Serial Interface (CSI-3) which is limited to 12 bits per pixel. The sensor would have to compress the image internally and would have an output bit depth not exceeding 12.

Table 2: ISensorMode – continued from previous page

Name	Type	Description
SensorModeType	SensorModeType	Describes the type of the sensor (Bayer, YUV, etc.) (Not all sensor mode types are supported in the current release.)
isBufferFormatSupported	bool	Is the provided buffer supported by camera device

Supported Interfaces:

ISensorMode
Provides methods to query the properties of a particular sensor mode.

3.4 CaptureSession

The **CaptureSession** object maintains an active connection to one or more **CameraDevices**, and controls the entire capture pipeline from input capture requests to output image streams. Here is an example of using a **CameraDevice** to create a **CaptureSession**:

```
// Create a capture session from the first reported device
CaptureSession* createSession(ICameraProvider* provider) {
    vector<CameraDevice*> devices;
    provider->getCameraDevices(&devices);
    if (devices.size() == 0)  // (only if no sensors present)
        return NULL;
    CameraDevice* dev = devices[0];

    Status status;
    CaptureSession* result =
        provider->createCaptureSession(dev, &status);
    if (status == STATUS_UNAVAILABLE)
        printf("First device not available\n");
    return result;
}
```

All capture requests start as a **Request** object and terminate at one or more **Stream** objects. All of these objects are created by a **CaptureSession**, and these objects can only be used by the session that creates them.

As an **IEventProvider**, a CaptureSession can also create **EventQueues** and generate various state, pipeline, or capture-related **Events**.

Capture requests may be submitted to a **CaptureSession** by the client on an individual, on-demand basis. Alternatively, a repeat capture request can be issued such that the **CaptureSession** will automatically repeat one or more requests automatically until the client cancels the request. Repeat captures are controlled by an internal **CaptureSession** thread, and may result in more than one capture being active in the pipeline at any point in time – this allows the implementation to maximize parallelization and resource utilization to increase framerate and overall throughput.

The **waitForIdle()** call will block until all captures have been completed. Once this call returns, no more events will be reported in this **CaptureSession** until new capture requests have been issued. The caller can specify an optional timeout value for the call; if the underlying camera pipeline does not become idle within that period of time, the call will return `STATUS_TIMEOUT`.

Before a **CaptureSession** can be fully destroyed – and the **CameraDevice(s)** resources it holds be released – the capture pipeline must be idle. Thus, **CaptureSession::destroy()** is a synchronous call that will block until the capture pipeline is idle, and returning from this call implies that the **CameraDevice(s)** the session had used are now available for use by another **CaptureSession**. Any overhead related to shutting down the camera device hardware will also be realized before returning from this call, so it is not guaranteed that **destroy()** will return quickly, even if the pipeline is idle.

Supported Interfaces:

ICaptureSession
: Provides methods to create **Requests**, output **Streams**, and submit capture **Requests** to the pipeline.

IEventProvider
: See **Event Provider**

3.5 OutputStreamSettings

An **OutputStreamSetting** object must be created in order to configure and then create an **OutputStream**. When an **OutputStreamSettings** object is created (with **createOutputStreamSettings()**), a **StreamType** must be provided, which will dictate which interfaces will be supported by the returned settings object. These settings must then be configured based on the **StreamType** that was used:

3.5.1 STREAM_TYPE_BUFFER

These settings are available through the **IBufferOutputStreamSettings** interface:

Table 3: IBufferOutputStreamSettings

Name	Type	Description
BufferType	BufferType	The native buffer type to use with the stream.
SyncType	SyncType	The sync type to use with the stream (or SYNC_TYPE_NONE for no sync).
MetadataEnable	bool	Whether or not **CaptureMetadata** should be attached to **Buffer** objects that are output by the stream.

3.5.2 STREAM_TYPE_EGL

These settings are available through the **IEGLOutputStreamSettings** interface:

Table 4: IOutputStreamSettings

Name	Type	Description
PixelFormat	PixelFormat	Pixel format for the buffers in this stream.
Resolution	Size	Width and height of the buffers in this stream.
EGLDisplay	EGLDisplay	EGL display that the created stream must belong to.
Mode	StreamMode	Choice of Mailbox or FIFO mode for this stream. In Mailbox mode, the most recently-acquired frame is always returned. In FIFO mode, frames are placed into a queue, and the head of the queue is returned.
fifoLength	uint32_t	Length of the frame fifo for this stream. (Ignored for streams created in Mailbox mode.)
metadataEnable	bool	Whether or not metadata should be attached to frames that are output to EGLStream.

For more information on Mailbox and FIFO modes, please refer to the EGL_KHR_stream specification.

3.6 OutputStream

OutputStream objects are created with **ICaptureSession::createOutputStream()**, and manage either a buffer- or EGLStream-based output stream that may be used for outputs from a capture request. The interfaces that are exposed by an **OutputStream** are dictated by the **StreamType** that was provided when the **OutputStreamSettings** object was first created.

3.6.1 STREAM_TYPE_BUFFER

Buffer output streams manage a set of **Buffer** objects that are created by the client to wrap native buffer resources. **Buffer** objects that are created by the **OutputStream** are **Destructable** objects and are used to manage and pass the buffer resources between Argus and the client. This is accomplished with two main actions:

Acquire
: When a **Request** is submitted that outputs to a buffer output stream, the buffer that is written to as a result of that request will be moved to an "acquire" queue within the stream to be acquired by the client using **IBufferOutputStream::acquire()**. These completed buffers must be acquired in the order they were written to the stream. If sync or metadata is enabled on the stream, it will be attached to the **Buffer** when it is acquired by the client.

Release
When a client has finished with a **Buffer** and is ready to reuse it again for another capture request, the client must "release" the **Buffer** back to the stream. Doing so will make the buffer available for use by Argus for a capture request. Note that there is no guarantee that the first-released Buffer be used first; Argus may select any available buffer for a capture request.

Supported Interfaces:

IBufferOutputStream
Provides controls for creating, acquiring, and releasing **Buffers** in the stream.

3.6.2 STREAM_TYPE_EGL

EGLStream output streams maintain a producer connection to an EGLStream, and output capture request results directly as EGLStream frames. These frames are then acquired by the EGLStream consumer using the EGLStream API available to the consumer.

Supported Interfaces:

IEGLOutputStream
Provides methods for synchronizing the EGLStream with the consumer connection, such as **waitUntilConnected()** and **disconnect()**.

3.7 BufferSettings

BufferSettings objects are created by **OutputStream** objects having the **STREAM_TYPE_BUFFER** type, and the interfaces that the buffer settings expose depend on the **BufferType** that is being used by the stream. They are used to create **Buffer** objects.

The only supported buffer type is currently **BUFFER_TYPE_EGL_IMAGE**, which is configured using the **IEGLImageBufferSettings** interface:

Table 5: IEGLImageBufferSettings

Name	Type	Description
EGLDisplay	EGLDisplay	EGLDisplay that created the provided EGLImage.
EGLImage	EGLImage	The EGLImage that will be wrapped as a sibling by the created Buffer.

3.8 Buffer

A **Buffer** object is created and owned by an **OutputStream** with the **STREAM_TYPE_BUFFER** type. It wraps a native buffer resource, and the interfaces that the buffer exposes depends on the **BufferType** and the **SyncType** of the Buffer.

All **Buffers** support the **IBuffer** interface, which provides the **getBufferType()** and **getSyncType()**.

3.8.1 BUFFER_TYPE_EGL_IMAGE

The only supported buffer type is currently **BUFFER_TYPE_EGL_IMAGE**. Buffers of this type expose the IEGLImageBuffer interface, which provides methods to get the EGLImage and EGLDisplay wrapped by the **Buffer**.

3.8.2 SYNC_TYPE_EGL_SYNC

The only supported sync type is currently **SYNC_TYPE_EGL_SYNC**. Buffers using this sync type expose the IEGLSync interface, which provides methods for getting the acquire sync and pushing the release sync.

3.9 InputStreamSettings

An **InputStreamSettings** object must be created in order to configure and then create an **InputStream**. An **InputStreamSettings** object must be created with the **ICaptureSession::createInputStreamSettings()** method, with **STREAM_TYPE_EGL** as the **StreamType** argument.

3.10 InputStream

An **InputStream** object is created with the **ICaptureSession::createInputStream()** method and represents an input stream required for reprocessing stream using libargus. **InputStream** objects are used as source streams for a capture request. The operation of a stream, the consumer of its buffers, and the interfaces it supports is based on **STREAM_TYPE_EGL**.

3.11 Request

A **Request** specifies exactly how a capture should be performed. The major controls are:

- *Enabled output streams.* Each enabled output stream will receive an output image from the capture. This image will be scaled as needed to fill the output buffer (which may include changing the image's aspect ratio.)
- *Per-stream settings.* Each output stream has its own settings, including post-processing controls (such as amount of noise reduction) and a source clip rectangle.
- *Source settings.* These control the sensor(s) being used for the capture, and include the sensor mode (primarily the resolution) as well as min/max limits for exposure time, gain, and frame duration.

- *Autocontrol settings.* These control the Argus autocontrol algorithms – primarily auto-exposure (AE) and auto-white-balance (AWB) – and related settings such as the color correction matrix.

Details of these settings are in the Settings section below.

Requests are created via **ICaptureSession::createRequest()**. The client can create any number of **Requests**, and use any of them with each call to **ICaptureSession::capture()** or any of the other capture methods. Although **Request** implements the **Destructable** interface, and therefore has an independent lifespan, each **Request** can be used only with the **CaptureSession** that created it.

When creating a **Request**, the client can specify one parameter: the *intent* of captures to be done with this request (of type **CaptureIntent**) – for example, preview, still, or video. This intent may change the initial values in the **Request**.

Here is a simple example of creating and configuring a **Request**:

```
// Create request with one stream and +1 EV
Request* createRequest(ICaptureSession* sess, Stream* stream) {
    const CaptureIntent intent = CAPTURE_INTENT_PREVIEW;
    Request* result = sess->createRequest(intent);
    IRequest* ireq = interface_cast<IRequest>(result);
    if (!ireq)
        return NULL;
    ireq->enableOutputStream(stream);
    IAutocontrolSettings* as = ireq->getAutocontrolSettings();
    as->setExposureCompensation(1.0f);
    return result;
}
```

Every time a **Request** is used in a capture, Argus effectively makes a copy of it, so that the client can change or delete a **Request** at any time without affecting any previous captures that used it.

Supported Interfaces:

IRequest
: Provides controls for output streams and methods to obtain the remaining interfaces in this list. Also allows the client to set a uint32_t **ClientData** value. This value will be found in the **ClientData** field of all metadata results for captures made with this **Request**, and is intended to help clients keep track of the request used for each block of metadata reported (especially when using burst captures).

ISourceSettings
: (obtained via **IRequest::getSourceSettings()** Allows the client to read and write sensor settings, including the sensor resolution and exposure time limits.

IAutocontrolSettings
: (obtained via **IRequest::getAutocontrolSettings()**) Provides access to settings for auto-exposure and auto-white-balance.

IDenoiseSettings
: Provides controls for denoise algorithms.

IEdgeEnhanceSettings
Provides controls for edge enhancement algorithms.

3.11.1 Request Settings

A capture **Request** object contains settings that are provided through a number of different interfaces:

3.11.1.1 Source Settings

These settings are available through the **ISourceSettings** interface and configure the source (i.e., camera device) for the capture:

Table 6: ISourceSettings

Name	Type	Description
ExposureTimeRange	Range<uint64_t>	Minimum and maximum exposure time (in nanoseconds.) The AE algorithm will strive to keep exposure time within this range.
FocusPosition	int32_t	Focuser position in focuser units. (Min and max values are provided by **ICameraProperties:: getFocuserPositionRange()**.)
AperturePosition	int32_t	Aperture position. (Min and max values are provided by **ICameraProperties:: getAperturePositionRange()** and all the recommended aperture positions are provided by **ICameraProperties:: getAperturePositions(()**.)
ApertureMotorSpeed	float_t	Aperture motor speed in motor steps per second units. (Min and max values are provided by **ICameraProperties:: getApertureMotorSpeedRange(()**.)
ApertureFNumber	float_t	Aperture f-number. All the available aperture f-numbers are provided by **ICameraProperties:: getAvailableApertureFNumbers()**.)
FrameDurationRange	Range<uint64_t>	Minimum and maximum frame duration (in nanoseconds.) Note that the current SensorMode may make this range impossible to obey.
GainRange	Range<float>	Minimum and maximum gain values to be used by the AE algorithm.

Table 6: ISourceSettings – continued from previous page

Name	Type	Description
SensorMode	SensorMode	Sensor mode used for the current capture(s). This must match one of the modes reported by **ICameraProperties::getAllSensorModes()**. Note that changing this from one capture to the next may incur a significant delay before the second capture completes.

The minimum and maximum legal values for the ranges above can be found in the **SensorMode** object that is being used for this capture. Values outside of the legal ranges will be clamped.

3.11.1.2 Autocontrol Settings

These settings are available through the **IAutocontrolSettings** interface:

Table 7: IAutocontrolSettings

Name	Type	Description
AeAntibandingMode	AeAntibandingMode	Adjustment of exposure duration to avoid banding caused by flickering of fluorescent light source (off, 50 Hz, 60 Hz, or auto).
AeLock	bool	If true, AE will maintain the current exposure value.
AeRegions	vector<AcRegion>	Image regions considered by the AE algorithm. An empty list (the default) means to consider the entire image.
BayerHistogramRegion	Rectangle<uint32_t>	Rectangle of the bayer histogram region of interest
AwbLock	bool	If true, AWB will maintain the current white balance gains.
AwbMode	AwbMode	Auto white balance mode (disabled, automatic, or any of a number of preset lighting modes).
AwbRegions	vector<AcRegion>	Image regions considered by the AWB algorithm. An empty list (the default) means to consider the entire image.
AfRegions	vector<AcRegion>	Image regions considered by the AF algorithm. An empty list (the default) means to consider the entire image.
WbGains	BayerTuple<float>	Manual white balance gains.
ColorCorrectionMatrixSize	Size	Dimensions of the ColorCorrection Matrix (read-only).

Table 7: IAutocontrolSettings – continued from previous page

Name	Type	Description
ColorCorrectionMatrix	vector<float>	Matrix that maps sensor RGB to linear sRGB. The matrix is stored in row-major order, and must have the size **width** * **height**, where **width** and **height** are the members of **ColorCorrectionMatrixSize**.
ColorCorrectionMatrixEnable	bool	If true, the **ColorCorrectionMatrix** will be used.
ColorSaturation	float	User-specified absolute color saturation, in the range [0.0, 2.0]. Will be ignored if **ColorSaturationEnable** is false.
ColorSaturationEnable	bool	If true, **ColorSaturation** will be used.
ColorSaturationBias	float	A multiplier for color saturation, in the range [0.0, 2.0], that will be applied to either the automatically-generated value, or the user-specified value in **ColorSaturation**.
ExposureCompensation	float	Exposure compensation, in (EV) stops.
ToneMapCurveSize	uint32_t	Number of elements in the **ToneMapCurve** (read-only).
ToneMapCurve	vector<float>	Tone map curve for one channel (R, G, or B). Must have size **ToneMapCurveSize**.
ToneMapCurveEnable	bool	If true, use the client-supplied ToneMapCurve.
IspDigitalGainRange	Range<float>	User-specified Isp Digital gain range

3.11.1.3 Stream Settings

These settings are available through the **IStreamSettings** interface and are provided on a per-stream basis to configure the output stream(s):

Table 8: IStreamSettings

Name	Type	Description
SourceClipRect	ClipRect	Rectangular portion of the sensor image (in normalized coordinates) that should appear in this output stream. Contents of this region will be scaled as needed to fill the output buffer, which may change the aspect ratio.
PostProcessingEnable	bool	If true, enable post-processing for this stream. Post-processing currently includes (but is not limited to) denoise.

3.11.1.4 Denoise Settings

These settings are available through the **IDenoiseSettings** interface:

Table 9: IDenoiseSettings

Name	Type	Description
DenoiseMode	DenoiseMode	Noise reduction mode (none, fast, or high quality).
DenoiseStrength	float	Amount of denoising to be performed; 0.0 is none, 1.0 is maximum.

3.11.1.5 Edge Enhance Settings

These settings are available through the **IEdgeEnhanceSettings** interface:

Table 10: IEdgeEnhanceSettings

Name	Type	Description
EdgeEnhanceMode	EdgeEnhanceMode	Edge enhancement mode (none, fast, or high quality).
EdgeEnhanceStrength	float	Amount of edge enhancement to be performed; 0.0 is none, 1.0 is maximum.

3.12 CaptureMetadata

A **CaptureMetadata** object contains the metadata associated with a single capture. This metadata includes:

- Many of the settings that were used to generate this capture.
- Current states associated with autocontrol algorithms; for example, AE convergence state.
- Information about the conditions of the capture; for example, total capture time, scene brightness, and gain values.

The client retrieves this information primarily through the **ICaptureMetadata** interface, which consists entirely of methods that return individual pieces of metadata. **CaptureMetadata** objects are delivered via `CAPTURE_COMPLETE` events (see the **Event** section). When a `CAPTURE_COMPLETE` event arrives, the client can obtain the **IEventCaptureComplete** interface from it, and retrieve the metadata via **getMetadata()**, as in this example:

```
float getSceneLux(IEvent* ievt) {
    IEventCaptureComplete* ccevt =
        interface_cast<IEventCaptureComplete>(ievt);
    if (!ccevt)
        return 0;    // not a CaptureComplete event!
    CaptureMetadata* cm = ccevt->getMetadata();
```

```
    ICaptureMetadata* icm = interface_cast<ICaptureMetadata>(cm);
    return icm->getSceneLux();
}
```

The metadata object will be valid as long as the `CAPTURE_COMPLETE` event is valid. See the **EventQueue** section for details on the lifespan of an **Event** object.

Metadata can also be acquired from the **IArgusCaptureMetadata** interface, which can be exposed from both **EGLStream::Frame** objects and **MetadataContainer** objects created directly from the metadata embedded in an EGLStream frame.

Supported interfaces:

ICaptureMetadata
Provides read-only access to general metadata items.

IDenoiseMetadata
Provides read-only access to metadata related to denoising.

IEdgeEnhanceMetadata
Provides read-only access to metadata related to edge enhancement.

Table 11: ICaptureMetadata

Name	Type	Description
CaptureId	uint32_t	Unique id for this capture (return value from **capture()** methods).
ClientData	uint32_t	ClientData field from the **Request** that was used for this capture.
StreamMetadata	InterfaceProvider	The per-stream metadata provider for the specified stream.
BayerHistogram	InterfaceProvider	Histogram of pixel values coming off the sensor. This object supports the **IBayerHistogram** interface.
RGBHistogram	InterfaceProvider	Histogram of pixel values after conversion to RGB space. This object supports the **IRGBHistogram** interface.
AeLocked	bool	Was the AE algorithm locked?
AeRegions	vector<AcRegion>	Regions of interest used by the AE algorithm.
BayerHistogramRegion	Rectangle<uint32_t>	Rectangle of the bayer histogram region of interest
AeState	AeState	State of AE at the time of the capture (inactive, searching, converged, or timeout).
FlickerState	AeFlickerState	Flicker state of AE at the time of capture

Table 11: ICaptureMetadata – continued from previous page

Name	Type	Description
FocuserPosition	int32_t	The position of the focuser in focuser units.
AwbCct	uint32_t	Correlated color temperature (in degrees Kelvin) calculated by the AWB algorithm.
AwbGains	BayerTuple<float>	Per-color-channel gains calculated by the AWB algorithm as per AWB mode used.
AwbMode	AwbMode	AWB mode used (disabled, automatic, or any of a number of preset lighting modes).
AwbRegions	vector<AcRegion>	Regions of interest used by the AWB algorithm.
AfRegions	vector<AcRegion>	Regions of interest used by the AF algorithm.
SharpnessScore	vector<float>	Sharpness score values calculated for corresponding AF regions.
AwbState	AwbState	State of AWB at the time of the capture (inactive, searching, converged, or locked).
AwbWbEstimate	vector<float>	The camera neutral color point estimate in native sensor color space.
ColorCorrectionMatrixEnable	bool	Was the client-supplied ColorCorrectionMatrix used?
ColorCorrectionMatrix	vector<float>	The 3x3 color correction matrix.
ColorSaturation	float	Color saturation used (including biasing).
FrameDuration	uint64_t	Time from start of frame exposure to start of the next frame exposure (in nanoseconds).
IspDigitalGain	float	Digital gain used.
FrameReadoutTime	uint64_t	Sensor frame readout time, in nanoseconds – the time between the beginning of the first line and the beginning of the last line.
SceneLux	float	The estimated brightness of the target scene (in lux).
SensorAnalogGain	float	Sensor analog gain used.
SensorExposureTime	uint64_t	Total sensor exposure time (in nanoseconds)
SensorSensitivity	uint32_t	ISO value used.

Table 11: ICaptureMetadata – continued from previous page

Name	Type	Description
SensorTimestamp	uint64_t	Start timestamp for the sensor capture, in nanoseconds.
ToneMapCurveEnabled	bool	Was client-supplied ToneMapCurve used?
ToneMapCurve	vector<float>	Tone map curve for one channel (R, G, or B).

Table 12: IDenoiseMetadata

Name	Type	Description
DenoiseMode	DenoiseMode	Denoise mode used.
DenoiseStrength	float	Denoise strength used.

Table 13: IEdgeEnhanceMetadata

Name	Type	Description
EdgeEnhanceMode	EdgeEnhanceMode	Edge enhancement mode used.
EdgeEnhanceStrength	float	Edge enhancement strength used.

3.13 Event

Each event implements an **IEvent** interface but, depending on type, it may also expose additional interfaces in order to provide additional data.

Supported interfaces:

IEvent
: Provides access to the event's associated frame, time, and type.

An event can have one of the following types:

3.13.1 EVENT_TYPE_ERROR

Event to report errors encountered during the processing of a capture request. Some errors may cause the capture to fail to produce valid output; in those cases, a `CAPTURE_COMPLETE` event will still be generated, but the status reported through **IEventCaptureComplete** will indicate an error.

Supported interfaces:

IEventError
: Provides access to the error status.

3.13.2 EVENT_TYPE_CAPTURE_STARTED

Event signifying the start of capture of a given frame. This event exposes only the **IEvent** interface.

For this event, **IEvent::getTime()** returns the frame timestamp in nanoseconds. This timestamp specifies the arrival of the start of the frame into the SoC in the SoC's time domain.

3.13.3 EVENT_TYPE_CAPTURE_COMPLETE

Event signalling the completion of a frame capture including any post-processing. When this event is generated, all outputs from the capture have already been pushed to their respective consumers.

Supported interfaces:

IEventCaptureComplete
Provides access to the **CaptureMetadata** and ths status of the capture.

3.14 EventQueue

An event queue is a container for Argus events. A queue is created via **IEventProvider::createEventQueue()**. The queue is populated with pending events with a call to **IEventProvider::waitForEvents()**. The user can then access the events with **IEventQueue::getNextEvent()**, and **IEventQueue::getEvent()**. An event object is valid until the next time **IEventProvider::waitForEvents()** is called on the **EventQueue** that provided the event, or until the **EventQueue** is destroyed, whichever comes first.

Every event queue has an upper bound on the number of events it can contain. That limit is currently 1,024. Once the event queue contains that number of events, new events will be dropped until the queue has been emptied. Applications should avoid this by regularly pulling events from all queues with **waitForEvents()**.

Supported interfaces:

IEventQueue
Provides access to the individual events.

3.15 EventProvider

An event provider is any object that generates events. (There is no dedicated **EventProvider** class.) Currently, the only event provider is a **CaptureSession**.

Event providers support the **IEventProvider** interface, which is used to create and populate **EventQueues**. Any object that implements this interface has to provide at least one event. The object's available event types can be seen by calling **getAvailableEventTypes()**. Once the event types are known, a **EventQueue** can be created to listen for those event types by calling **createEventQueue()**. **EventQueues** can be created to listen for any subset of the available event types. Note that each **EventQueue** belongs to a single event provider and contains events from only that provider.

To wait for events, the client calls **waitForEvents()**. If an **EventQueue** contains events and it is passed to **waitForEvents()**, those events are no longer valid and will be cleared from the **EventQueue**. Likewise any **Events** read from that queue will not be valid after the call to **waitForEvents()**.

Supported interfaces:

IEventProvider

Allows creation of **EventQueues** and waiting on/retreiving those queues for new events.