# Argus 0.96 API Specification

September 29, 2016

# 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 L4T on NVIDIA Tegra TX1-based 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 with EGLStreams, which are directly supported by other system components such as OpenGL and Cuda, and which require no buffer copies during delivery to the consumer.
- 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 are unsupported in the current release, and will be available on only some platforms.)
- 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.)
- Libraries that consume EGLStream outputs in different ways; for example, jpeg encoding or direct application access to the images.
- Metadata delivery via both Argus events and 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.

Functionality *not* provided by Argus:

- Auto-focus. *(Will be added in a later release.)*
- Reprocessing of YUV images (such as that required by Androids Zero Shutter Lag feature.)
- Reprocessing of Bayer (raw) images. *(Will be added in a later release.)*
- Output of Bayer (raw) images. *(Will be added in a later release.)*

# 2 Fundamentals

## 2.1 Types

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

### 2.1.1 Enumerations

**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.

Types.h also declares a variety of enumerations used for settings and metadata; see individual include files for definitions and specific uses.

### 2.1.2 Structs

**UUID** A 128-bit unique identifier used to define interfaces and pixel formats in Argus. There is also a **NamedUUID** subclass (which includes a string value along with the identifier) and a number of further subclasses (such as **PixelFormat**) that provide type safety.

**PixelFormat** Defines the pixel format used in an output stream.

**Range** Specifies an operational range. When passed to Argus components the components will try to keep their values between min and max.

**Rectangle** Specifies a rectangle in pixel dimensions.

**Size** Defines a width and height in integer units.

**ClipRect** Specifies a bounding box in normalized coordinates where top left is (0.0, 0.0) and bottom right is (1.0, 1.0).

**AcRegion** Specifies an autocontrol region of interest, defined by the left and top coordinates with a width and height (all defined in pixel space.) Additionally a scalar weighting factor can be defined for each region.

**BayerTuple<T>** Specifies a set of values of type **T**, one for each Bayer channel (red, green-even, green-odd, and blue).

### 2.1.3 Support Classes

**NonCopyable** A utility class in Argus that many objects inherit; this class overrides the standard copy operator and disables the inheriting class from being copied.

## 2.2 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;
}
```

## 2.3 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.

## 2.4 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** have lifetimes defined by the API. The rules for each such object are in the object descriptions below.

## 2.5 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. While extensions will often introduce new objects or interfaces, it is not required. An extension may simply relax previous restrictions on the API and allow behavior that was previously disallowed. 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.

## 2.6 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.

### 2.6.1 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::createEGLStreamProducer()**. 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 enough there is 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 will be included with all events and other output from this capture. The capture id will be incremented by one from each capture to the next. If the call fails for any reason, the return value will be zero.

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.

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

### 2.6.2 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.

## 2.7 EGLStreams

All Argus image input and output is done exclusively by connecting Argus to an EGLStream as either a producer or consumer endpoint. 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/]

### 2.7.1 EGLStream Producer

Image output from Argus is performed by connecting a **Stream** object to the producer endpoint of an EGLStream using **ICaptureSession::createEGLStreamProducer()**. This **Stream** object is then enabled as an output stream in a **Request** to have the capture result written to a new frame in the EGLStream upon completion. 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.

### 2.7.2 EGLStream Consumer

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 classnameFrame 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 frames 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. |

### 2.7.3 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 consumer, but this negotiation (whether it occurs and how it works) is an implementation details."

There is currently no automatic format negotiation between Argus and any consumers, and it's up to 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.

### 2.8 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 events 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.

### 2.9 Multiple Clients and Multiple Threads

In the current L4T release, Argus can be used by multiple processes simultaneously. 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.

In the current Android release, Argus can be used by only one process at a time. Future Android releases will allow multiple simultaneous Argus applications.

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 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::getAllSensorModes()**. Every **SensorMode** object, whether basic or extended, exposes the **ISensorMode** interface, which provides the following information:

Table 1: 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. |
| SensorModeType | SensorModeType | Describes the type of the sensor (Bayer, YUV, etc.) (Not all sensor mode types are supported in the current release.) |

Supported Interfaces:

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

### 3.3 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()**. **CameraDevice** objects can be used to query the capabilities of that particular device, and are used to create **CaptureSessions** which will issue captures using the device. A **CameraDevice** can be used by only one **CaptureSession** at any point in time, and 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 2: ICameraProperties

| Name | Type | Description |
|---|---|---|
| MaxAeRegions | uint32_t | Maximum number of AeRegions supported |
| MaxAwbRegions | uint32_t | Maximum number of AwbRegions supported |
| SensorModes | vector<SensorMode> | Sensor modes supported by this device |
| FocusPositionRange | Range<int32_t> | Range of valid focuser positions |
| LensApertureRange | Range<float> | Range of supported lens apertures |

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.

### 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. Heres 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 its 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 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 images 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**.

Heres 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.

**IVideoStabilizationSettings**
Provides controls for video stabilization.

## 3.6 Settings

### 3.6.1 Source Settings

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

Table 3: 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()**.) |
| 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. |
| 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.

### 3.6.2 Autocontrol Settings

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

Table 4: 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. |
| 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. |
| WbGains | BayerTuple<float> | Manual white balance gains. |
| ColorCorrectionMatrixSize | Size | Dimensions of the ColorCorrectionMatrix (read-only). |
| 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). |

**Table 4: IAutocontrolSettings – continued from previous page**

| Name | Type | Description |
|---|---|---|
| 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. |

### 3.6.3 Stream Settings

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

Table 5: 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 includes denoising and video stabilization, and possibly other operations. |

### 3.6.4 Denoise Settings

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

Table 6: 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. |

### 3.6.5 Edge Enhance Settings

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

**Table 7: IEdgeEnhanceSettings – continued from previous page**

| Name | Type | Description |
|---|---|---|

Table 7: 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. |

### 3.6.6 Video Stabilization Settings

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

Table 8: IVideoStabilizationSettings

| Name | Type | Description |
|---|---|---|
| VideoStabilizationMode | VideoStabilizationMode | Video stabilization mode (disabled or enabled). |

### 3.6.7 Output Stream Settings

These settings are available through the **IOutputStreamSettings** interface, which is exposed on the settings object passed to **ICaptureSession::createOutputStream()**:

Table 9: 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. |
| CameraDevice | CameraDevice | Camera source for this stream. |
| EGLDisplay | EGLDisplay | EGL display that the created stream must belong to. |
| Mode | StreamMode | Selects 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.) |

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

### 3.7 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.

**IVideoStabilizationMetadata**
: Provides read-only access to metadata related to video stabilization.

Table 10: 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. |
| AeState | AeState | State of AE at the time of the capture (inactive, searching, converged, or locked). |
| 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. |
| 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. |
| 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. |

**Table 10: ICaptureMetadata – continued from previous page**

| Name | Type | Description |
|---|---|---|
| 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. |
| 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 11: IDenoiseMetadata

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

Table 12: IEdgeEnhanceMetadata

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

Table 13: IVideoStabilizationMetadata

| Name | Type | Description |
|---|---|---|
| VideoStabilizationMode | VideoStabilizationMode | Video stabilization mode used. |

### 3.8 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 events associated frame, time, and type.

An event can have one of the following types:

### 3.8.1 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.

### 3.8.2 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 SoCs time domain.

### 3.8.3 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.

## 3.9 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.

### 3.10 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 objects 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.