arena_api package

Submodules

arena_api.arena_api_config module

Used by autodoc_mock_imports.

- arena_api is built on the C API (ArenaC) for Arena SDK. arena_api loads the ArenaC binary and its dependencies automatically from the Arena SDK's default installation directory. To load the ArenaC binary from a custom location add the full path to ArenaC binary as a value to a valid key in the dictionary. The dictionary name must be `ARENAC_CUSTOM_PATHS` and must have all of the following keys:

  'python32_win' :
  - Used to point Python 32 on Windows to load the 32-bit ArenaC binary.
  - If this key has a value of empty string, arena_api loads '<Installation Dir>\Win32Release\ArenaC_v140.dll'.

  'python64_win' :
  - Used to point Python 64 on Windows to load the 64-bit ArenaC binary.
  - If this key has a value of empty string, arena_api loads '<Installation Dir>\x64Release\ArenaC_v140.dll'.

  'python32_lin' :
  - Used to point Python 32 on Linux to load the 32-bit ArenaC binary.
  - If this key has a value of empty string, arena_api uses the paths in '/etc/ld.so.conf.d/Arena_SDK.conf' to find the ArenaC shared object.

  'python64_lin' :
  - Used to point Python 64 on Linux to load the 64-bit ArenaC binary.
  - If this key has a value of empty string, arena_api uses the paths in '/etc/ld.so.conf.d/Arena_SDK.conf' to find the ArenaC shared object.

Note:

- If the library path assigned to any of the keys does not exist, a FileNotFoundError exception will be thrown.
- Linux keys have been tested on Ubuntu 16.04 LTS.
- To use the installed arena, give the key a value of empty string ‘’.
- If ArenaSDK is not installed in the default location, ‘C:Program FilesLucid Vision LabsArena SDK’, by the installer , it is not necessary to add the non-default installation path to the custom paths.

arena_api.buffer module

class **arena_api.buffer.BufferFactory**

Bases: `object`

A static class responsible for the copying, conversion, and destruction of buffers with image data –Image buffers.

The factory allocates and deallocates memory for its images. Memory is allocated when an image is copied `BufferFactory.Copy()``or converted ``BufferFactory.Convert()`. To clean up memory, all images created by `BufferFactory` must be destroyed via `BufferFactory.Destroy()`.

Images from this factory are treated noticeably differently than those from a `Device` instance. Retrieving an image from a device grabs a buffer that had its memory preallocated when the device started streaming; retrieving and requeuing does not allocate or deallocate memory, but simply moves buffers around the acquisition engine. Copying, and converting an image with this factory allocates and deallocates memory as needed. This is why images from a device must be requeued with `device.requeue_buffer()` while images from the image factory must be destroyed via `BufferFactory.Destroy()`.

———————————————————— ————————————————————-

static **convert**(*new_pixel_format*, *bayer_algorithm=None*)

Converts an image buffer to the selected pixel format. In doing so, it creates a completely new image, similar to a deep copy but with a different pixel format.

Args:

- buffer:
    - A `_Buffer` instance to convert.
- new_pixel_format:
    - `enums.PixelFormat` to convert to.
- bayer_algorithm:
    - `enums.BayerAlgorithm` type. This is the Bayer conversion algorithm to use. Only applicable when converting from bayer.
    - `None` for no conversion algorithm.

Raises:

- `TypeError`:
    - `buffer` is not of `_Buffer` type.
    - `new_pixel_format` type is not a `str`, `int`, nor `enum.PixelFormat`.
    - `bayer_algorithm` type is not a `str`, `int`, nor `enum.BayerAlgorithm`.

Returns:

- `_Buffer` instance with new pixel format. This is a new instance.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

```
>>> image = device.get_buffer()
>>> image_BGRa8 = None
>>> if image.pixel_format != enums.PixelFormat.BGRa8:
>>>     image_BGRa8 = BufferFactory.covert(image, enums.PixelFormat.BGRa8)
>>> else:
>>>     image_BGRa8 = BufferFactory.copy(image)
>>> device.requeue_buffer(image)
>>> # process image_BRGa8, and then destroy image from factory
>>> BufferFactory.destroy(image_BGRa8)
```

The list of supported pixel formats can be found in `enums.PixelFormat`. The list of supported conversion pixel formats differs from a device's pixel formats `PixelFormat` node. In order for the conversion to succeed, both the source and destination pixel formats must be supported. Bayer formats are supported as source formats only.

Warning:

- Images from the image factory must be destroyed.
- Images from a device should be requeued.

- Cannot convert to Bayer formats.
- Bayer conversion algorithm is only necessary when converting from Bayer formats.

---

static **copy**()

Creates a deep copy of an image buffer from another image buffer. **Args**:

buffer:

- A `_Buffer` instance to copy.

Raises:

- `TypeError`:
  - `buffer` is not of type `_Buffer`.

Returns:

- `_Buffer` instance.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

When copying an image, the `BufferFactory` allocates memory for the new image. As such, images created by copying an image with the image factory must be destroyed; otherwise, memory will leak.

```
>>> image = device.get_buffer()
>>> image_copy = BufferFactory.copy(image)
>>> # must use requeue_buffer for image from device
>>> device.requeue_buffer(image)
>>> # process image_copy, and then destroy image from factory
>>> BufferFactory.destroy(image_copy)
```

Warning:

- Images from the image factory must be destroyed.
- Images from a device should be requeued.
- Instantiates all lazy properties of the original image.

---

static **copy_compressed_image**()

Creates a deep copy of a compressed image buffer from another compressed image buffer. **Args**:

buffer:

- A `_Buffer` instance to copy.

Raises:

- `TypeError`:
  - `buffer` is not of type `_Buffer`.
  - `buffer` is not of compressed image type.

Returns:

- `_Buffer` instance.

Compressed images from this method factory must be destroyed via `BufferFactory.Destroy()`.

Warning:

- Images from the image factory must be destroyed.
- Images from a device should be requeued.
- Instantiates all lazy properties of the original image.

---

static **create**(*data_size*, *width*, *height*, *pixel_format*)

static **create_compressed_image**(*data_size*, *pixel_format*)

static **create_empty**(*width*, *height*, *pixel_format*)

static **decompress_image**()

Decompresses compressed image data. **Args**:

buffer:

- A `_Buffer` instance to copy.

Raises:

- `TypeError`:
    - `buffer` is not of type `_Buffer`.
    - `buffer` is not of compressed image type.

Returns:

- `_Buffer` instance.

Decompressed images from this method factory must be destroyed via `BufferFactory.Destroy()`.

Warning:

- Images from the image factory must be destroyed.

---

static `deinterleave_channels`()

Separates interleaved channels into a planar image.

Args:

buffer:

- A `_Buffer` instance to separate. It must be created by `BufferFactory`.

Returns:

- `_Buffer` instance with a planar image. This is a new instance.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

---

static `deinterleave_channels_len`()

Return length of buffer that needed for a planar image.

Args:

buffer:

- A `_Buffer` instance to calculate a separated image. It must be created by `BufferFactory`.

Returns:

- Length (size_t) of buffer that needed for a planar image.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

---

static **`deinterleave_channels_shallow`**(*pdata*, *len*)

Separates interleaved channels into a planar image.

Args:

buffer:

- A `_Buffer` instance to separate. It must be created by `BufferFactory`.

pdata:

- Data of the image in a form of a pointer to ctypes.c_uint8 or ctypes.c_ubyte.

len:

- Length of buffer that needed for a planar image.

Returns:

- `_Buffer` instance with a planar image. This is a new instance.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

---

static **`destroy`**()

Destroyes an image buffer.

Args:

buffer:

- A `_Buffer` instance to destroy. it must be created by `BufferFactory`.

Raises:

- `TypeError`:
    - `buffer` is not of type `_Buffer`.

Returns:

- `None`.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

Cleans up an image buffer and deallocates its memory. It must be called on any image created by the buffer factory function `BufferFactory.copy()` `BufferFactory.convert()`.

> Images from the buffer factory must be destroyed via `BufferFactory.Destroy()`.

All images from the buffer factory, whether copied or converted, must be destroyed to deallocate their memory; otherwise, memory will leak. It is important that this method only be called on image buffers from the image factory, and not on those retrieved from a device.

> **Warning:**

- Images from the image factory must be destroyed.
- Images from a device should be requeued.

---

static **destroy_compressed_image**()

Destroyes a compressed image buffer.

Args:

> buffer:

- A `_Buffer` instance to destroy. it must be created by `BufferFactory`.

Raises:

- `TypeError`:
  - `buffer` is not of type `_Buffer`.
  - `buffer` is not of compressed image type.

Returns:

- `None`.

> **Warning:**

- Images from the image factory must be destroyed.
- Images from a device should be requeued.

---

static **process_software_lut**(*plut*, *len*)

Runs an image through a lookup table, allowing for a redefinition of values.

Args:

buffer:

- A `_Buffer` instance to destroy. It must be created by `BufferFactory`.

plut:

- Pointer to the beginning of the lookup table

len:

- Length of buffer

Raises:

- `TypeError`:
  - `buffer` is not of type `_Buffer`.

Returns:

- `_Buffer` instance processed with lookup table. This is a new instance.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

---

static **select_bits**(*num_bits*, *offset*)

Creates a copy of an image buffer from another image buffer while selecting bits.

Args:

buffer:

- A `_Buffer` instance to destroy. It must be created by `BufferFactory`.

numBits:

- Number of bits to scale to. It must be <8>

offset:

- Offset to scale to.

Raises:

- `TypeError`:
    - `buffer` is not of type `_Buffer`.

Returns:

- `_Buffer` instance with new scale. This is a new instance.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

———————————————————— ————————————————————

static **`select_bits_and_scale`**(*num_bits*, *offset*)

Creates a scaled copy of an image buffer from another image buffer.

Args:

buffer:

- A `_Buffer` instance to destroy. It must be created by `BufferFactory`.

numBits:

- Number of bits to scale to. It must be <8>

offset:

- Offset to scale to.

Raises:

- `TypeError`:
    - `buffer` is not of type `_Buffer`.

Returns:

- `_Buffer` instance with new scale. This is a new instance.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

———————————————————— ————————————————————

static **`shallow`**(*data_size*, *width*, *height*, *pixel_format*)

static **split_channels**()

Takes an interleaved image and separates the channels into multiple images.

Args:

buffer:

- A `_Buffer` instance to split. It must be created by `BufferFactory`.

Returns:

- `_Buffer` instance in a vector with split images. This is a new instance.

Images from this method factory must be destroyed via `BufferFactory.Destroy()`.

———————————————————— ————————————————————

class **arena_api.buffer._Buffer**(*hxbuffer*)

Bases: `object`

Buffers are the most generic form of acquisition engine data retrieved from a device. They are acquired and requeued via `Device` instances. Buffers with image data are referred to as image, image data, image buffer.

```
>>> # retrieving a buffer after starting the stream
>>> # requeuing it before stopping the stream
>>> device.start_stream()
>>> buffer = device.get_buffer()
>>> device.requeue_buffer(buffer)
>>> device.stop_stream()
```

Buffers can hold image data, chunk data or both. buffer class provides:

- Buffer and payload information like payload `buffer.data`, payload and buffer size `buffer.size_filled`, `buffer.buffer_size`, and frame ID `buffer.frame_id`.
- Type information like payload type `buffer.payload_type` and whether the payload has image and/or chunk data `buffer.has_imagedata`, `buffer.has_chunkdata`.
- Error information `buffer.is_incomplete` and `buffer.is_data_larger_than_buffer`.
- Size information `buffer.width`, `buffer.height`, `buffer.offset_x`, and `buffer.offset_y`.
- Padding `buffer.padding_x` and `buffer.padding_y`.
- Pixel information `buffer.pixel_format`, `buffer.pixel_endianness`, and `buffer.timestamp_ns`.

images: Image buffers are the most common form of data retrieved in `_Buffer`. Image data can be copied, and converted via the `BufferFactory`. It is important to note that images retrieved from the camera must be requeued `buffer.requeue_buffer()` whereas images

created using the image factory must be destroyed `BufferFactory.destroy()`.

chunk: The concept of chunk data is a method of adding extra data (such as CRC, width, height, etc.) to an image. A nuance of this concept is whether the additional information is appended to the back of the image or the image is treated as part of the chunk data. This is important for parsing the data. LUCID devices create chunk data by appending it to the payload. In order to receive chunk data with an image, chunk data must be enabled and configured on node map `device.nodemap`. Chunk data must first be activated via `ChunkModeActive`. Each specific chunk must then be selected and enabled via `ChunkSelector` and `ChunkEnable`.

```
>>> # enabling pixel format chunk data
>>> device.nodemap.get_node('ChunkModeActive').value = True
>>> device.nodemap.get_node('ChunkSelector').value = 'PixelFormat'
>>> # another syntax for get_node is []
>>> device.nodemap['ChunkEnable'].value = True
```

Chunk data objects provide the ability to get chunks `buffer.get_chunk()`. Otherwise a exception will be raised.

Warning:

- Should be requeued; otherwise, acquisition engine may starve.
- Properties lazily instantiated if buffer retrieved from device.
- Chunk buffers:
    - Should be requeued; same as other buffers `buffer.requeue_buffer()`.
- Image buffers:
    - Should be requeued if retrieved from the device.
    - Must be destroyed if created by the image factory.
    - Properties of images from the image factory may be unavailable.

`bits_per_pixel`

Bits per pixel of the image.

Getter: Gets the number of bits per pixel of the image buffer from the integer value of the `buffer.pixel_format` (PfncFormat).

Type: `int`.

Unit: pixels

Gets the number of bits per pixel of the image buffer from the integer value of the `buffer.pixel_format` (PfncFormat). Internally, a public helper function is called `get_bits_per_pixel()`. Pixel format values are determined by the PFNC (Pixel Format Naming Convention) specification. The PFNC assigns a name and number to each pixel format, helping to standardize pixel formats. The number of bits per pixel can be found in each integer at bytes 5 and 6 (mask 0x00FF0000).

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`buffer_size`

Retrieves the size of a buffer.

Getter: Retrieves the size of a buffer.

Type: `int`.

Unit: bytes

The payload size is calculated at the beginning of the stream `device.start_stream()` and cannot be recalculated until the stream has stopped `device.stop_stream()`. Because of this, features that can affect payload size (`Width`, `Height`, `PixelFormat`) become unwritable when the stream has started.

buffer_size vs size_filled

The size filled is often same as the size of the buffer (`buffer.buffer_size`), but not because they are one and the same. `buffer.size_filled` returns the number of bytes received whereas `buffer.buffer_size` returns the size of the buffer, which can either be allocated by the user or calculated by Arena.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`compressed_image_data`

---

compressed_image_pbytes

———————————————————— ————————————————————-

compressed_image_pdata

———————————————————— ————————————————————-

compressed_image_pixel_format

The compressed buffer's pixel format as an enum, as defined by the PFNC specification.

Getter: The compressed buffer's pixel format.

Type: enums.PixelFormat

Gets the pixel format (PfncFormat) of the image, as defined by the PFNC (Pixel Format Naming Convention). Images are self-describing, so the device does not need to be queried to get this information.

Pixel format value are determined by the PFNC (Pixel Format Naming Convention) specification. The PFNC assigns a name and number to each pixel format helping to standardize pixel formats. The number of bits per pixel can be found in each integer at bytes 5 and 6 (mask 0x00FF0000). The pixel format can be determined by the integer using the GetPixelFormatName function provided by the PFNC.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

———————————————————— ————————————————————-

compressed_image_timestamp_ns

Timestamp of the compressed image buffer in nanoseconds.

Getter: Gets the timestamp of the compressed image in nanoseconds.

Type: `int`.

Unit: nanoseconds/

Gets the timestamp of the compressed image in nanoseconds. Images are self-describing, so the device does not need to be queried to get this information.

The `BufferFactory` can create an compressed image buffer from another compressed image buffer or from a minimal set of parameters `buffer.data`, `buffer.width`, `buffer.height`, `buffer.pixel_format`. If the image buffer is created from parameters, the

timetamp will be set to `0`, no matter its original value.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

———————————————————— ————————————————————-

`data`

———————————————————— ————————————————————-

`frame_id`

Gets the frame ID, a sequential identifier for buffers.

Getter: Gets the frame ID.

Type: `int`.

Frame IDs start at `1` and continue until `2^64-1` (64-bit), at which point they roll over back to `1`. The frame ID should never be `0`.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

———————————————————— ————————————————————-

`get_chunk`(*chunk_names*)

Gets a specified chunk or multiple chunks, returning it as a node(s) in order to preserve metadata related to the chunk.

Args:

chunk_names: it can be:

- A `str`.
- A `list` of `str`.
- A `tuple` of `str`.

- The str's value represents the name of the chunk to retrieve as a node. the name is prefixed with 'Chunk'. for example the name of the CRC chunk is 'ChunkCRC'

Raises:

- `ValueError` :
  - `chunk_names` is a `list` or `tuple` but has an element that is not a `str`.
  - `chunk_names` value does not match a valid chunk name.
- `TypeError` :
  - `chunk_names` type is not `list`, `tuple` nor `str`.

Returns:

- A `dict`, that has chunk name as a key and the node is the value, if `chunk_names` is a `list`.
- A `node` instance when `chunk_names` is a str.

Internally, chunk data objects have an internal node map and a chunk adapter. These allow chunk information to be processed and read as `Node` instances.

There is a chance that incomplete images have garbage data in place of expected chunk data. If this is the case, it is still possible to attempt chunk retrieval. Invalid chunks raise a `ValueError`.

```
>>> # enabling timestamp chunk data
>>> device.nodemap.get_node('ChunkModeActive').value = True
>>> device.nodemap.get_node('ChunkSelector').value = 'CRC'
>>> device.nodemap.get_node('ChunkEnable').value = True
>>> device.start_stream()
>>> buffer_with_chunk_data = device.get_buffer()
>>> if buffer_with_chunk_data.is_incomplete:
>>>     print('Chunks might contain garbage values')
>>> else:
>>>     chunk_crc_node = buffer_with_chunk_data.get_chunk('ChunkCRC')
>>>     print(chunk_crc_node.value)
>>> device.stop_stream()
```

Chunk data must meet three criteria to provide relevant data. Chunk mode must be activated `ChunkModeActive`, the chunk must be enabled `ChunkSelector` value is `ChunkEnable`, and the node must exist:

- If chunk mode is inactive, the buffer will not contain chunk data. - If chunk does not exist, returns null. - If chunk is not enabled, returned node will be unavailable.

  Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

---

has_chunkdata

Returns whether or not a buffer's payload has data that may be interpreted as chunk data. `True` if the payload has chunk data otherwise, `False`.

Getter: Returns whether or not a buffer's payload has data that may be interpreted as chunk data.

Type: `bool`.

Returns `True` if the payload type is:

- `enums.PayloadType.CHUNKDATA`.
- `enums.PayloadType.IMAGE_EXTENDED_CHUNK`.
- `enums.PayloadType.COMPRESSED_IMAGE_EXTENDED_CHUNK`.

Returns `False` if the payload type is:

- `enums.PayloadType.IMAGE`.
- `enums.PayloadType.COMPRESSED_IMAGE`.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

---

has_imagedata

Returns whether or not a buffer's payload has data that may be interpreted as image data. `True` if the payload has image data otherwise, `False`.

Getter: Returns whether or not a buffer's payload has data that may be interpreted as image data.

Type: `bool`.

Returns `True` if the payload type is:

- `enums.PayloadType.IMAGE`.
- `enums.PayloadType.IMAGE_EXTENDED_CHUNK`.

Returns `False` if the payload type is:

- `enums.PayloadType.CHUNKDATA`.
- `enums.PayloadType.COMPRESSED_IMAGE`.

- `enums.PayloadType.COMPRESSED_IMAGE_EXTENDED_CHUNK`.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`height`

Height of the image.

Getter: Gets the height of the image buffer.

Type: `int`.

Unit: pixels

Gets the height of the image buffer in pixels. Images are self-describing, so the device does not need to be queried to get this information.

Image buffers are either retrieved from a `Device` instance or created by the factory `BufferFactory`. If the image was retrieved from a device, the height is populated by the acquisition engine payload leader. The device itself is not queried as this data is present in the image data. If the image was created by the `BufferFactory`, the height is populated by the arguments.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`is_compressedimage`

Returns whether or not a buffer's payload has data that may be interpreted as compressed image data. `True` if the payload has compressed image data otherwise, `False`.

Getter: Returns whether or not a buffer's payload has data that may be interpreted as compressed image data.

Type: `bool`.

Returns `True` if the payload type is:

- `enums.PayloadType.COMPRESSED_IMAGE`.

- `enums.PayloadType.COMPRESSED_IMAGE_EXTENDED_CHUNK`.

Returns `False` if the payload type is:

- `enums.PayloadType.CHUNKDATA`.
- `enums.PayloadType.IMAGE`.
- `enums.PayloadType.IMAGE_EXTENDED_CHUNK`.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`is_data_larger_than_buffer`

Returns whether or not a buffer's payload data is larger than the buffer.

Getter: Returns whether or not a buffer's payload data is too large for the buffer.

Type: `bool`.

A buffer may be missing data if the buffer to hold the data is too small. This happens when the size of the buffer `buffer.buffer_size` does not match the expected data size `PayloadSize`. This will also return `True` when checking whether the data is larger than the buffer `buffer.is_data_larger_than_buffer`.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`is_incomplete`

Returns whether or not the payload is complete.

Getter: Returns whether or not the payload is complete.

Type: `bool`.

Error handling may be required in the case that the data is incomplete. An incomplete image signifies that the data size `buffer.size_filled` does not match the expected data size `PayloadSize`. This is either due to missed packets or a small buffer.

The number of missed packets may be discovered through the stream node map `device.tl_stream_nodemap`. The missed packet count feature `StreamMissedPacketCount` is a cumulative count of all missed packets, and does not necessarily reflect the number of missed packets for any given buffer.

A buffer may be missing data if the buffer to hold the data is too small. This happens when the size of the buffer `buffer.buffer_size` does not match the expected data size `PayloadSize`. This will also return `True` when checking whether the data is larger than the buffer `buffer.is_data_larger_than_buffer`.

> **Warning:**
>
> - Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
> - Properties lazily instantiated if buffer retrieved from device.

`is_valid_crc`

Returns whether or not a buffer data has a matching crc value as that which was received from the device.

> **Getter:** Returns whether or not a buffer data has a matching

crc value as that which was received from the device.

> **Type:** `bool`.

Returns `True` if the calculated CRC value equals the one sent from the device, otherwise, `False`.

Calculates the CRC of a buffer's data and verifies it against the CRC value sent from the device. This helps verify that no data has been changed or missed during a transmission. This calls a global helper function to calculate the CRC. A CRC is performed by running a set of calculations on a dataset both before and after a transmission. The two calculated values are then compared for equality. If the values are the same, then the transmission is deemed successful; if different, then something in the transmission went wrong.

A device can be set to send a CRC value by enabling its chunk data setting.

```
>>> nodemap.get_node('ChunkModeActive').value = True
```

```
>>> nodemap.get_node('ChunkSelector').value = 'CRC'
```

```
>>> nodemap.get_node('ChunkEnable').value = True
```

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.
- If CRC chunk is not enabled, calling `buffer.is_valid_crc` will raise an exception.

`offset_x`

Offset X of the image buffer.

Getter: Gets the offset X of the image buffer.

Type: `int`.

Unit: pixels

Gets the offset of the image along the X-axis. Images are self-describing, so the device does not need to be queried to get this information.

Image buffers are either retrieved from a `Device` instance or created by the factory `BufferFactory`. If the image was retrieved from a device, the height is populated by the acquisition engine payload leader. The device itself is not queried as this data is present in the image data. If the image was created by the `BufferFactory`, the height is populated by the arguments.

The `BufferFactory` can create an image buffer from another image buffer or from a minimal set of parameters (`buffer.data`, `buffer.width`, `buffer.height`, `buffer.pixel_format`). If the image buffer is created from parameters, the offset X will be set to `0`, no matter its original value.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`offset_y`

Offset Y of the image buffer.

Getter: Gets the offset Y of the image buffer.

Type: `int`.

Unit: pixels

Gets the offset of the image along the Y-axis. Images are self-describing, so the device does not need to be queried to get this information.

Image buffers are either retrieved from a `Device` instance or created by the factory `BufferFactory`. If the image was retrieved from a device, the height is populated by the acquisition engine payload leader. The device itself is not queried as this data is present in the image data. If the image buffer was created by the `BufferFactory`, the height is populated by the arguments.

The `BufferFactory` can create an image buffer from another image buffer or from a minimal set of parameters (`buffer.data`, `buffer.width`, `buffer.height`, `buffer.pixel_format`). If the image buffer is created from parameters, the offset Y will be set to `0`, no matter its original value.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`padding_x`

Padding X of the image.

Getter: Gets the padding of the image along the X-axis.

Type: `int`.

Unit: pixels

Gets the padding of the image along the X-axis. Images are self-describing, so the device does not need to be queried to get this information.

Image buffers are either retrieved from a `Device` instance or created by the factory `BufferFactory`. If the image was retrieved from a device, the height is populated by the acquisition engine payload leader. The device itself is not queried as this data is present in the image data. If the image buffer was created by the `BufferFactory`, the height is populated by the arguments.

The `BufferFactory` can create an image buffer from another image buffer or from a minimal set of parameters (`buffer.data`, `buffer.width`, `buffer.height`, `buffer.pixel_format`). If the image buffer is created from parameters, the padding X will be set to `0`, no matter its original value.

Padding X specifically refers to the number of bytes padding the end of each line. This number will affect the pitch/stride/step of an image.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`padding_y`

Padding Y of the image.

Getter: Gets the padding of the image along the Y-axis.

Type: `int`.

Unit: pixels

Gets the padding of the image along the Y-axis. Images are self-describing, so the device does not need to be queried to get this information.

Image buffers are either retrieved from a `Device` instance or created by the factory `BufferFactory`. If the image was retrieved from a device, the height is populated by the acquisition engine payload leader. The device itself is not queried as this data is present in the image data. If the image buffer was created by the `BufferFactory`, the height is populated by the arguments.

The `BufferFactory` can create an image buffer from another image buffer or from a minimal set of parameters (`buffer.data`, `buffer.width`, `buffer.height`, `buffer.pixel_format`). If the image buffer is created from parameters, the padding Y will be set to `0`, no matter its original value.

Padding Y specifically refers to the number of bytes padding the end of each line. This number will affect the pitch/stride/step of an image.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

---

payload_size

Size of the intended payload.

Getter: Size of the intended payload.

Type: `int`.

Unit: bytes.

payload_size vs size_filled

Retrieves the intended size of the payload. This is similar to the retrieved payload size `buffer.size_filled` but different in that missed data is included. This returns the same as the SFNC feature by the same name ('PayloadSize').

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

---

payload_type

The buffer's payload type as an enum, as defined by the GigE Vision specification.

Getter: The buffer's payload.

Type: enums.PayloadType

The payload type indicates how to interpret the data stored in the buffer `buffer.data` or `buffer.pdata`. Lucid devices may provide three ways to interpret the data:

- As an image `enums.PayloadType.IMAGE`.
- As an image with chunk data appended to the end `enums.PayloadType.IMAGE_EXTENDED_CHUNK`.
- As chunk data, which may or may not include image data as a chunk `enums.PayloadType.CHUNKDATA`.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

---

pbytes

pdata

---

pixel_endianness

Endianness of the pixels of the image.

Getter: Gets the pixel endianness of the image data.

Type: enums.PixelEndianness

Gets the pixel endianness of the image data. Images are self-describing, so the device does not need to be queried to get this information.

Image buffers are either retrieved from a `Device` instance or created by the factory `BufferFactory`. If the image was retrieved from a device, the height is populated by the acquisition engine payload leader. The device itself is not queried as this data is present in the image data. If the image buffer was created by the `BufferFactory`, the height is populated by the arguments.

The `BufferFactory` can create an image buffer from another image buffer or from a minimal set of parameters `buffer.data`, `buffer.width`, `buffer.height`, `buffer.pixel_format`. If the image buffer is created from parameters, the pixel endianness will be set to `0` which is enums.PixelEndianness.UNKNOWN , no matter its original value.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

---

pixel_format

The buffer's pixel format as an enum, as defined by the PFNC specification.

Getter: The buffer's pixel format.

Type: enums.PixelFormat

Gets the pixel format (PfncFormat) of the image, as defined by the PFNC (Pixel Format Naming Convention). Images are self-describing, so the device does not need to be queried to get this information.

Image buffers are either retrieved from a `Device` instance or created by the factory `BufferFactory`. If the image was retrieved from a device, the height is populated by the acquisition engine payload leader. The device itself is not queried as this data is present in

the image data. If the image buffer was created by the `BufferFactory`, the height is populated by the arguments.

Pixel format value are determined by the PFNC (Pixel Format Naming Convention) specification. The PFNC assigns a name and number to each pixel format helping to standardize pixel formats. The number of bits per pixel can be found in each integer at bytes 5 and 6 (mask 0x00FF0000). The pixel format can be determined by the integer using the GetPixelFormatName function provided by the PFNC.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`size_filled`

Size of the received payload.

Getter: Size of the received payload.

Type: `int`.

Unit: bytes

Retrieves the size of the payload data, excluding transport layer protocol leaders. The payload data may include image data, chunk data, or both.

size_filled vs buffer_size

The size filled is often same as the size of the buffer `buffer.buffer_size`, but not because they are one and the same. `buffer.size_filled` returns the number of bytes received whereas `buffer.buffer_size` returns the size of the buffer, which can either be allocated by the user or calculated by Arena `nodemap.get_node('PayloadSize')`.

size_filled vs payload_size

Retrieves the intended size of the payload. This is similar to the retrieved payload size `buffer.size_filled` but different in that missed data is included. This returns the same as the SFNC feature by the same name ('PayloadSize').

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`timestamp_ns`

Timestamp of the image buffer in nanoseconds.

Getter: Gets the timestamp of the image in nanoseconds.

Type: `int`.

Unit: nanoseconds/

Gets the timestamp of the image in nanoseconds. Images are self-describing, so the device does not need to be queried to get this information.

Image buffers are either retrieved from a `Device` instance or created by the factory `BufferFactory`. If the image was retrieved from a device, the height is populated by the acquisition engine payload leader. The device itself is not queried as this data is present in the image data. If the image buffer was created by the `BufferFactory`, the height is populated by the arguments.

The `BufferFactory` can create an image buffer from another image buffer or from a minimal set of parameters `buffer.data`, `buffer.width`, `buffer.height`, `buffer.pixel_format`. If the image buffer is created from parameters, the timetamp will be set to `0` which is enums.PixelEndianness.UNKNOWN , no matter its original value.

This is the same as the nanosecond timestamp property `buffer.timestamp` (deprecated in 2.0.0).

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

`width`

Width of the image buffer.

Getter: Gets the width of the image buffer.

Type: `int`.

Unit: pixels.

Gets the width of the image buffer in pixels. Images are self-describing, so the device does not need to be queried to get this information.

Image buffers are either retrieved from a `Device` instance or created by the factory `BufferFactory`. If the image was retrieved from a device, the width is populated by the acquisition engine payload leader. The device itself is not queried as this data is present in the image data. If the image was created by the `BufferFactory`, the width is populated by the arguments.

Warning:

- Causes undefined behavior if buffer requeued `device.requeue_buffer()`.
- Properties lazily instantiated if buffer retrieved from device.

---

arena_api.callback module

Callback module

This module provides callback registration and management for devices, nodes, and systems in Arena API.

- Use `arena_api.callback._Callback` via the singleton instance `arena_api.callback.callback`.
- Use `arena_api.callback._CallbackFunction` via the instance `arena_api.callback.callback_function` to decorate Python functions for events.

Examples

```
>>> from arena_api.callback import callback, callback_function
>>> @callback_function.node.on_update
... def cb(node, *args, **kwargs):
...     print(node.name, node.value)
```

arena_api.callback.callback *= <arena_api.callback._Callback object>*

Singleton instance of `_Callback` for registering/deregistering callbacks.

arena_api.callback.callback_function *= <arena_api.callback._CallbackFunction object>*

Singleton instance of `_CallbackFunction` providing decorators.

arena_api.enums module

class **arena_api.enums.AccessMode**

Bases: `enum.IntEnum`

An enumeration.

NA = *1*

NI = *0*

RO = *3*

RW = *4*

UNDEFINED = *5*

WO = *2*

---

class **arena_api.enums.BayerAlgorithm**

Bases: `enum.IntEnum`

An enumeration.

ADAPTIVE_HOMOGENEITY_DIRECTED = *2*

DIRECTIONAL_INTERPOLATION = *1*

UNKNOWN = *0*

---

class **arena_api.enums.CachingMode**

Bases: `enum.IntEnum`

An enumeration.

NO_CACHE = *0*

UNDEFINED = *3*

WRITE_AROUND = *2*

WRITE_THROUGH = *1*

class arena_api.enums.DisplayNotation

Bases: `enum.IntEnum`

An enumeration.

AUTOMATIC = *0*

FIXED = *1*

SCIENTIFIC = *2*

UNDEFINED = *3*

class arena_api.enums.IncMode

Bases: `enum.IntEnum`

An enumeration.

FIXED = *1*

LIST = *2*

NONE = *0*

class arena_api.enums.InterfaceType

Bases: `enum.IntEnum`

An enumeration.

BOOLEAN = *3*

CATEGORY = *8*

COMMAND = *4*

ENUMENTRY = *10*

ENUMERATION = *9*

FLOAT = *5*

INTEGER = *2*

REGISTER = *7*

STRING = *6*

---

class **arena_api.enums.Namespace**

Bases: `enum.IntEnum`

An enumeration.

CUSTOM = *0*

STANDARD = *1*

UNDEFINED = *2*

---

class **arena_api.enums.PayloadType**

Bases: `enum.IntEnum`

An enumeration.

CHUNKDATA = *4*

COMPRESSED_IMAGE = *1001*

COMPRESSED_IMAGE_EXTENDED_CHUNK = *1002*

IMAGE = *1*

IMAGE_EXTENDED_CHUNK = *16385*

class **arena_api.enums.PixelEndianness**

Bases: `enum.IntEnum`

An enumeration.

BIG = *2*

LITTLE = *1*

UNKNOWN = *0*

class **arena_api.enums.PixelFormat**

Bases: `enum.IntEnum`

An enumeration.

B10 = *17432786*

B12 = *17563859*

B16 = *17826004*

B8 = *17301713*

BGR10 = *36700185*

BGR10p = *35520584*

BGR12 = *36700187*

BGR12p = *35913801*

BGR14 = *36700234*

BGR16 = *36700235*

BGR24 *= 2185757958*

BGR565p *= 34603062*

BGR8 *= 35127317*

BGRY8 *= 2183135234*

BGRa10 *= 37748812*

BGRa10p *= 36175949*

BGRa12 *= 37748814*

BGRa12p *= 36700239*

BGRa14 *= 37748816*

BGRa16 *= 37748817*

BGRa8 *= 35651607*

BayerBG10 *= 17825807*

BayerBG10Packed *= 17563689*

BayerBG10p *= 17432658*

BayerBG12 *= 17825811*

BayerBG12Packed *= 17563693*

BayerBG12p *= 17563731*

BayerBG16 = *17825841*

BayerBG24 = *2165835012*

BayerBG8 = *17301515*

BayerGB10 = *17825806*

BayerGB10Packed = *17563688*

BayerGB10p = *17432660*

BayerGB12 = *17825810*

BayerGB12Packed = *17563692*

BayerGB12p = *17563733*

BayerGB16 = *17825840*

BayerGB24 = *2165835011*

BayerGB8 = *17301514*

BayerGR10 = *17825804*

BayerGR10Packed = *17563686*

BayerGR10p = *17432662*

BayerGR12 = *17825808*

BayerGR12Packed = *17563690*

BayerGR12p = *17563735*

BayerGR16 = *17825838*

BayerGR24 = *2165835009*

BayerGR8 = *17301512*

BayerRG10 = *17825805*

BayerRG10Packed = *17563687*

BayerRG10p = *17432664*

BayerRG12 = *17825809*

BayerRG12Packed = *17563691*

BayerRG12p = *17563737*

BayerRG16 = *17825839*

BayerRG24 = *2165835010*

BayerRG8 = *17301513*

BiColorBGRG10 = *35651753*

BiColorBGRG10p = *34865322*

BiColorBGRG12 = *35651757*

BiColorBGRG12p = *35127470*

BiColorBGRG8 *= 34603174*

BiColorRGBG10 *= 35651751*

BiColorRGBG10p *= 34865320*

BiColorRGBG12 *= 35651755*

BiColorRGBG12p *= 35127468*

BiColorRGBG8 *= 34603173*

CFA1by2_BR10 *= 2165309467*

CFA1by2_BR10Packed *= 2165047349*

CFA1by2_BR10p *= 2164916320*

CFA1by2_BR12 *= 2165309511*

CFA1by2_BR12Packed *= 2165047415*

CFA1by2_BR12p *= 2165047393*

CFA1by2_BR16 *= 2165309497*

CFA1by2_BR8 *= 2164785171*

CFA1by2_RB10 *= 2165309466*

CFA1by2_RB10Packed *= 2165047348*

CFA1by2_RB10p *= 2164916319*

CFA1by2_RB12 *= 2165309510*

CFA1by2_RB12Packed *= 2165047414*

CFA1by2_RB12p *= 2165047392*

CFA1by2_RB16 *= 2165309496*

CFA1by2_RB8 *= 2164785170*

CFA2by1_BW10 *= 2165309469*

CFA2by1_BW10Packed *= 2165047351*

CFA2by1_BW10p *= 2164916322*

CFA2by1_BW12 *= 2165309513*

CFA2by1_BW12Packed *= 2165047417*

CFA2by1_BW12p *= 2165047395*

CFA2by1_BW16 *= 2165309499*

CFA2by1_BW8 *= 2164785173*

CFA2by1_WB10 *= 2165309468*

CFA2by1_WB10Packed *= 2165047350*

CFA2by1_WB10p *= 2164916321*

CFA2by1_WB12 *= 2165309512*

CFA2by1_WB12Packed *= 2165047416*

CFA2by1_WB12p *= 2165047394*

CFA2by1_WB16 *= 2165309498*

CFA2by1_WB8 *= 2164785172*

CFA4by1_RGBW10 *= 2165309465*

CFA4by1_RGBW10Packed *= 2165047347*

CFA4by1_RGBW10p *= 2164916318*

CFA4by1_RGBW12 *= 2165309509*

CFA4by1_RGBW12Packed *= 2165047413*

CFA4by1_RGBW12p *= 2165047391*

CFA4by1_RGBW16 *= 2165309495*

CFA4by1_RGBW8 *= 2164785169*

CFA4by1_WBGR10 *= 2165309464*

CFA4by1_WBGR10Packed *= 2165047346*

CFA4by1_WBGR10p *= 2164916317*

CFA4by1_WBGR12 *= 2165309508*

CFA4by1_WBGR12Packed *= 2165047412*

CFA4by1_WBGR12p *= 2165047390*

CFA4by1_WBGR16 *= 2165309494*

CFA4by1_WBGR8 *= 2164785168*

CFA_BRGG10 *= 2165309461*

CFA_BRGG10Packed *= 2165047343*

CFA_BRGG10p *= 2164916314*

CFA_BRGG12 *= 2165309505*

CFA_BRGG12Packed *= 2165047409*

CFA_BRGG12p *= 2165047387*

CFA_BRGG16 *= 2165309491*

CFA_BRGG8 *= 2164785165*

CFA_GGBR10 *= 2165309463*

CFA_GGBR10Packed *= 2165047345*

CFA_GGBR10p *= 2164916316*

CFA_GGBR12 *= 2165309507*

CFA_GGBR12Packed *= 2165047411*

CFA_GGBR12p *= 2165047389*

CFA_GGBR16= *2165309493*

CFA_GGBR8= *2164785167*

CFA_GGRB10= *2165309462*

CFA_GGRB10Packed= *2165047344*

CFA_GGRB10p= *2164916315*

CFA_GGRB12= *2165309506*

CFA_GGRB12Packed= *2165047410*

CFA_GGRB12p= *2165047388*

CFA_GGRB16= *2165309492*

CFA_GGRB8= *2164785166*

CFA_RBGG10= *2165309460*

CFA_RBGG10Packed= *2165047342*

CFA_RBGG10p= *2164916313*

CFA_RBGG12= *2165309504*

CFA_RBGG12Packed= *2165047408*

CFA_RBGG12p= *2165047386*

CFA_RBGG16= *2165309490*

CFA_RBGG8 = *2164785164*

Confidence1 = *17301700*

Confidence16 = *17825991*

Confidence1p = *16842949*

Confidence32f = *18874568*

Confidence8 = *17301702*

Coord3D_A10p = *17432789*

Coord3D_A12p = *17563864*

Coord3D_A16 = *17825974*

Coord3D_A32f = *18874557*

Coord3D_A8 = *17301679*

Coord3D_ABC10p = *35520731*

Coord3D_ABC10p_Planar = *35520732*

Coord3D_ABC12p = *35913950*

Coord3D_ABC12p_Planar = *35913951*

Coord3D_ABC16 = *36700345*

Coord3D_ABC16_Planar = *36700346*

Coord3D_ABC16s = *2184184833*

Coord3D_ABC32f = *39846080*

Coord3D_ABC32f_Planar = *39846081*

Coord3D_ABC8 = *35127474*

Coord3D_ABC8_Planar = *35127475*

Coord3D_ABCY16 = *2185233411*

Coord3D_ABCY16s = *2185233408*

Coord3D_ABCY8 = *2183136259*

Coord3D_AC10p = *34865392*

Coord3D_AC10p_Planar = *34865393*

Coord3D_AC12p = *35127538*

Coord3D_AC12p_Planar = *35127539*

Coord3D_AC16 = *35651771*

Coord3D_AC16_Planar = *35651772*

Coord3D_AC32f = *37748930*

Coord3D_AC32f_Planar = *37748931*

Coord3D_AC8 = *34603188*

Coord3D_AC8_Planar *= 34603189*

Coord3D_B10p *= 17432790*

Coord3D_B12p *= 17563865*

Coord3D_B16 *= 17825975*

Coord3D_B32f *= 18874558*

Coord3D_B8 *= 17301680*

Coord3D_C10p *= 17432791*

Coord3D_C12p *= 17563866*

Coord3D_C16 *= 17825976*

Coord3D_C16Y8 *= 2182611972*

Coord3D_C16s *= 2165310466*

Coord3D_C32f *= 18874559*

Coord3D_C8 *= 17301681*

Coord3D_CY16 *= 2183136261*

Coord3D_CY8 *= 2182087684*

Coord3D_Y16 *= 2182087685*

DualBayerBG10 *= 2165313551*

DualBayerBG10Packed *= 2165051433*

DualBayerBG10_2ch *= 2166362127*

DualBayerBG10p *= 2164920402*

DualBayerBG12 *= 2165313555*

DualBayerBG12Packed *= 2165051437*

DualBayerBG12_2ch *= 2166362131*

DualBayerBG12p *= 2165051475*

DualBayerBG16 *= 2165313585*

DualBayerBG16_2ch *= 2166362161*

DualBayerBG8 *= 2164789259*

DualBayerBG8_2ch *= 2165313547*

DualBayerGB10 *= 2165313550*

DualBayerGB10Packed *= 2165051432*

DualBayerGB10_2ch *= 2166362126*

DualBayerGB10p *= 2164920404*

DualBayerGB12 *= 2165313554*

DualBayerGB12Packed *= 2165051436*

DualBayerGB12_2ch *= 2166362130*

DualBayerGB12p *= 2165051477*

DualBayerGB16 *= 2165313584*

DualBayerGB16_2ch *= 2166362160*

DualBayerGB8 *= 2164789258*

DualBayerGB8_2ch *= 2165313546*

DualBayerGR10 *= 2165313548*

DualBayerGR10Packed *= 2165051430*

DualBayerGR10_2ch *= 2166362124*

DualBayerGR10p *= 2164920406*

DualBayerGR12 *= 2165313552*

DualBayerGR12Packed *= 2165051434*

DualBayerGR12_2ch *= 2166362128*

DualBayerGR12p *= 2165051479*

DualBayerGR16 *= 2165313582*

DualBayerGR16_2ch *= 2166362158*

DualBayerGR8 *= 2164789256*

DualBayerGR8_2ch = *2165313544*

DualBayerRG10 = *2165313549*

DualBayerRG10Packed = *2165051431*

DualBayerRG10_2ch = *2166362125*

DualBayerRG10p = *2164920408*

DualBayerRG12 = *2165313553*

DualBayerRG12Packed = *2165051435*

DualBayerRG12_2ch = *2166362129*

DualBayerRG12p = *2165051481*

DualBayerRG16 = *2165313583*

DualBayerRG16_2ch = *2166362159*

DualBayerRG8 = *2164789257*

DualBayerRG8_2ch = *2165313545*

DualMono10 = *2165313539*

DualMono10Packed = *2165051396*

DualMono10_2ch = *2166362115*

DualMono10p = *2164920390*

DualMono12 = *2165313541*

DualMono12Packed = *2165051398*

DualMono12_2ch = *2166362117*

DualMono12p = *2165051463*

DualMono16 = *2165313543*

DualMono16_2ch = *2166362119*

DualMono8 = *2164789249*

DualMono8_2ch = *2165313537*

G10 = *17432782*

G12 = *17563855*

G16 = *17826000*

G8 = *17301709*

InvalidPixelFormat = *0*

LucidXYTP128f = *2189444438*

Mono10 = *17825795*

Mono10Packed = *17563652*

Mono10p = *17432646*

Mono12 = *17825797*

Mono12Packed = *17563654*

Mono12p = *17563719*

Mono14 = *17825829*

Mono16 = *17825799*

Mono1p = *16842807*

Mono24 = *2165835008*

Mono2p = *16908344*

Mono4p = *17039417*

Mono8 = *17301505*

Mono8s = *17301506*

PolarizeMono12 = *2165309448*

PolarizeMono12Packed = *2165047302*

PolarizeMono12p = *2165047367*

PolarizeMono16 = *2165309447*

PolarizeMono8 = *2164785153*

PolarizedAngles_0d_45d_90d_135d_BayerRG12 = *2185232927*

PolarizedAngles_0d_45d_90d_135d_BayerRG12p *= 2184184351*

PolarizedAngles_0d_45d_90d_135d_BayerRG8 *= 2183135759*

PolarizedAngles_0d_45d_90d_135d_Mono12 *= 2185232415*

PolarizedAngles_0d_45d_90d_135d_Mono12p *= 2184183839*

PolarizedAngles_0d_45d_90d_135d_Mono16 *= 2185232431*

PolarizedAngles_0d_45d_90d_135d_Mono8 *= 2183135247*

PolarizedAolp_BayerRG12 *= 2165310191*

PolarizedAolp_BayerRG12p *= 2165048047*

PolarizedAolp_BayerRG8 *= 2164785887*

PolarizedAolp_Mono12 *= 2165309679*

PolarizedAolp_Mono12p *= 2165047535*

PolarizedAolp_Mono8 *= 2164785375*

PolarizedDolpAngle_BayerRG12p *= 2182611727*

PolarizedDolpAngle_BayerRG16 *= 2183136031*

PolarizedDolpAngle_BayerRG8 *= 2182087423*

PolarizedDolpAngle_Mono12p *= 2182611215*

PolarizedDolpAngle_Mono16 *= 2183135519*

PolarizedDolpAngle_Mono8 = *2182086911*

PolarizedDolpAolp_BayerRG12 = *2183135919*

PolarizedDolpAolp_BayerRG12p = *2182611631*

PolarizedDolpAolp_BayerRG8 = *2182087327*

PolarizedDolpAolp_Mono12 = *2183135407*

PolarizedDolpAolp_Mono12p = *2182611119*

PolarizedDolpAolp_Mono8 = *2182086815*

PolarizedDolp_BayerRG12 = *2165310159*

PolarizedDolp_BayerRG12p = *2165048015*

PolarizedDolp_BayerRG16 = *3138436027*

PolarizedDolp_BayerRG8 = *2164785855*

PolarizedDolp_Mono12 = *2165309647*

PolarizedDolp_Mono12p = *2165047503*

PolarizedDolp_Mono16 = *2853218986*

PolarizedDolp_Mono8 = *2164785343*

PolarizedStokes_S0_S1_S2_BayerRG12p = *2183397967*

PolarizedStokes_S0_S1_S2_BayerRG8 = *2182611519*

PolarizedStokes_S0_S1_S2_Mono12p = *2183397455*

PolarizedStokes_S0_S1_S2_Mono16 = *2184183903*

PolarizedStokes_S0_S1_S2_Mono8 = *2182611007*

PolarizedStokes_S0_S1_S2_S3_BayerRG12 = *2185233023*

PolarizedStokes_S0_S1_S2_S3_BayerRG12p = *2184184447*

PolarizedStokes_S0_S1_S2_S3_BayerRG8 = *2183135855*

PolarizedStokes_S0_S1_S2_S3_Mono12 = *2185232511*

PolarizedStokes_S0_S1_S2_S3_Mono12p = *2184183935*

PolarizedStokes_S0_S1_S2_S3_Mono16 = *2185232527*

PolarizedStokes_S0_S1_S2_S3_Mono8 = *2183135343*

QOI_BGR8 = *2182613506*

QOI_BayerBG8 = *2164787971*

QOI_BayerGB8 = *2164787970*

QOI_BayerGR8 = *2164787968*

QOI_BayerRG8 = *2164787969*

QOI_Mono8 = *2164787712*

QOI_RGB8 = *2182613505*

QOI_YCbCr8 *= 2182613507*

QOI_YCbCr8_CbYCr *= 2182613508*

QuadBayerBG10 *= 2165321743*

QuadBayerBG10Packed *= 2165059625*

QuadBayerBG10_4ch *= 2168467471*

QuadBayerBG10p *= 2164928594*

QuadBayerBG12 *= 2165321747*

QuadBayerBG12Packed *= 2165059629*

QuadBayerBG12_4ch *= 2168467475*

QuadBayerBG12p *= 2165059667*

QuadBayerBG16 *= 2165321777*

QuadBayerBG16_4ch *= 2168467505*

QuadBayerBG8 *= 2164797451*

QuadBayerBG8_4ch *= 2166370315*

QuadBayerGB10 *= 2165321742*

QuadBayerGB10Packed *= 2165059624*

QuadBayerGB10_4ch *= 2168467470*

QuadBayerGB10p = *2164928596*

QuadBayerGB12 = *2165321746*

QuadBayerGB12Packed = *2165059628*

QuadBayerGB12_4ch = *2168467474*

QuadBayerGB12p = *2165059669*

QuadBayerGB16 = *2165321776*

QuadBayerGB16_4ch = *2168467504*

QuadBayerGB8 = *2164797450*

QuadBayerGB8_4ch = *2166370314*

QuadBayerGR10 = *2165321740*

QuadBayerGR10Packed = *2165059622*

QuadBayerGR10_4ch = *2168467468*

QuadBayerGR10p = *2164928598*

QuadBayerGR12 = *2165321744*

QuadBayerGR12Packed = *2165059626*

QuadBayerGR12_4ch = *2168467472*

QuadBayerGR12p = *2165059671*

QuadBayerGR16 *= 2165321774*

QuadBayerGR16_4ch *= 2168467502*

QuadBayerGR8 *= 2164797448*

QuadBayerGR8_4ch *= 2166370312*

QuadBayerRG10 *= 2165321741*

QuadBayerRG10Packed *= 2165059623*

QuadBayerRG10_4ch *= 2168467469*

QuadBayerRG10p *= 2164928600*

QuadBayerRG12 *= 2165321745*

QuadBayerRG12Packed *= 2165059627*

QuadBayerRG12_4ch *= 2168467473*

QuadBayerRG12p *= 2165059673*

QuadBayerRG16 *= 2165321775*

QuadBayerRG16_4ch *= 2168467503*

QuadBayerRG8 *= 2164797449*

QuadBayerRG8_4ch *= 2166370313*

QuadMono10 *= 2165321731*

QuadMono10Packed = *2165059588*

QuadMono10_4ch = *2168467459*

QuadMono10p = *2164928582*

QuadMono12 = *2165321733*

QuadMono12Packed = *2165059590*

QuadMono12_4ch = *2168467461*

QuadMono12p = *2165059655*

QuadMono16 = *2165321735*

QuadMono16_4ch = *2168467463*

QuadMono8 = *2164797441*

QuadMono8_4ch = *2166370305*

R10 = *17432778*

R12 = *17563851*

R16 = *17825996*

R8 = *17301705*

RGB10 = *36700184*

RGB10V1Packed = *35651612*

RGB10_Planar = *36700194*

RGB10p = *35520604*

RGB10p32 = *35651613*

RGB12 = *36700186*

RGB12V1Packed = *35913780*

RGB12_Planar = *36700195*

RGB12p = *35913821*

RGB14 = *36700254*

RGB16 = *36700211*

RGB16_Planar = *36700196*

RGB24 = *2185757957*

RGB565p = *34603061*

RGB8 = *35127316*

RGB8_Planar = *35127329*

RGBY8 = *2183135233*

RGBa10 = *37748831*

RGBa10p = *36175968*

RGBa12 = *37748833*

RGBa12p = *36700258*

RGBa14 = *37748835*

RGBa16 = *37748836*

RGBa8 = *35651606*

SCF1WBWG10 = *17825896*

SCF1WBWG10p = *17432681*

SCF1WBWG12 = *17825898*

SCF1WBWG12p = *17563755*

SCF1WBWG14 = *17825900*

SCF1WBWG16 = *17825901*

SCF1WBWG8 = *17301607*

SCF1WGWB10 = *17825903*

SCF1WGWB10p = *17432688*

SCF1WGWB12 = *17825905*

SCF1WGWB12p = *17563762*

SCF1WGWB14 = *17825907*

SCF1WGWB16 = *17825908*

SCF1WGWB8 = *17301614*

SCF1WGWR10 = *17825910*

SCF1WGWR10p = *17432695*

SCF1WGWR12 = *17825912*

SCF1WGWR12p = *17563769*

SCF1WGWR14 = *17825914*

SCF1WGWR16 = *17825915*

SCF1WGWR8 = *17301621*

SCF1WRWG10 = *17825917*

SCF1WRWG10p = *17432702*

SCF1WRWG12 = *17825919*

SCF1WRWG12p = *17563776*

SCF1WRWG14 = *17825921*

SCF1WRWG16 = *17825922*

SCF1WRWG8 = *17301628*

YCbCr10_CbYCr = *36700291*

YCbCr10p_CbYCr = *35520644*

YCbCr12_CbYCr = *36700293*

YCbCr12p_CbYCr = *35913862*

YCbCr16_CbYCr = *2184185095*

YCbCr2020_10_CbYCr = *36700405*

YCbCr2020_10p_CbYCr = *35520758*

YCbCr2020_12_CbYCr = *36700407*

YCbCr2020_12p_CbYCr = *35913976*

YCbCr2020_411_8_CbYYCrYY = *34341113*

YCbCr2020_422_10 = *35651836*

YCbCr2020_422_10_CbYCrY = *35651837*

YCbCr2020_422_10p = *34865406*

YCbCr2020_422_10p_CbYCrY = *34865407*

YCbCr2020_422_12 = *35651840*

YCbCr2020_422_12_CbYCrY = *35651841*

YCbCr2020_422_12p = *35127554*

YCbCr2020_422_12p_CbYCrY = *35127555*

YCbCr2020_422_8 = *34603258*

YCbCr2020_422_8_CbYCrY = *34603259*

YCbCr2020_8_CbYCr = *35127540*

YCbCr24_CbYCr = *2185757960*

YCbCr411_16_CbYYCrYY = *2182612233*

YCbCr411_24_CbYYCrYY = *2183398666*

YCbCr411_8 = *34340954*

YCbCr411_8_CbYYCrYY = *34340924*

YCbCr422_10 = *35651685*

YCbCr422_10_CbYCrY = *35651737*

YCbCr422_10p = *34865287*

YCbCr422_10p_CbYCrY = *34865306*

YCbCr422_12 = *35651686*

YCbCr422_12_CbYCrY = *35651739*

YCbCr422_12p = *35127432*

YCbCr422_12p_CbYCrY = *35127452*

YCbCr422_16_CbYCrY = *2183136519*

YCbCr422_24_CbYCrY= *2184185096*

YCbCr422_8= *34603067*

YCbCr422_8_CbYCrY= *34603075*

YCbCr601_10_CbYCr= *36700297*

YCbCr601_10p_CbYCr= *35520650*

YCbCr601_12_CbYCr= *36700299*

YCbCr601_12p_CbYCr= *35913868*

YCbCr601_411_8_CbYYCrYY= *34340927*

YCbCr601_422_10= *35651725*

YCbCr601_422_10_CbYCrY= *35651741*

YCbCr601_422_10p= *34865294*

YCbCr601_422_10p_CbYCrY= *34865310*

YCbCr601_422_12= *35651727*

YCbCr601_422_12_CbYCrY= *35651743*

YCbCr601_422_12p= *35127440*

YCbCr601_422_12p_CbYCrY= *35127456*

YCbCr601_422_8= *34603070*

YCbCr601_422_8_CbYCrY = *34603076*

YCbCr601_8_CbYCr = *35127357*

YCbCr709_10_CbYCr = *36700305*

YCbCr709_10p_CbYCr = *35520658*

YCbCr709_12_CbYCr = *36700307*

YCbCr709_12p_CbYCr = *35913876*

YCbCr709_411_8_CbYYCrYY = *34340930*

YCbCr709_422_10 = *35651733*

YCbCr709_422_10_CbYCrY = *35651745*

YCbCr709_422_10p = *34865302*

YCbCr709_422_10p_CbYCrY = *34865314*

YCbCr709_422_12 = *35651735*

YCbCr709_422_12_CbYCrY = *35651747*

YCbCr709_422_12p = *35127448*

YCbCr709_422_12p_CbYCrY = *35127460*

YCbCr709_422_8 = *34603073*

YCbCr709_422_8_CbYCrY = *34603077*

YCbCr709_8_CbYCr = *35127360*

YCbCr8 = *35127387*

YCbCr8_CbYCr = *35127354*

YUV411_8_UYYVYY = *34340894*

YUV422_8 = *34603058*

YUV422_8_UYVY = *34603039*

YUV8_UYV = *35127328*

---

class **arena_api.enums.Representation**

Bases: `enum.IntEnum`

An enumeration.

BOOLEAN = *2*

HEX_NUMBER = *4*

IPV4_ADDRESS = *5*

LINEAR = *0*

LOGARITHMIC = *1*

MAC_ADDRESS = *6*

PURE_NUMBER = *3*

UNDEFINED = *7*

class **arena_api.enums.SC_TIFF_COMPRESSION_LIST**

Bases: `enum.IntEnum`

An enumeration.

SC_NO_TIFF_COMPRESSION = *0*

SC_TIFF_COMPRESSION_ADOBE_DEFLATE = *3*

SC_TIFF_COMPRESSION_CCITTFAX3 = *4*

SC_TIFF_COMPRESSION_CCITTFAX4 = *5*

SC_TIFF_COMPRESSION_DEFLATE = *2*

SC_TIFF_COMPRESSION_JPEG_TIFF = *7*

SC_TIFF_COMPRESSION_LOGLUV = *8*

SC_TIFF_COMPRESSION_LZW = *6*

SC_TIFF_COMPRESSION_PACKBITS = *1*

class **arena_api.enums.ScJpegSubsamplingList**

Bases: `enum.IntEnum`

An enumeration.

SC_JPEG_SUBSAMPLING_411 = *1*

SC_JPEG_SUBSAMPLING_420 = *2*

SC_JPEG_SUBSAMPLING_422 = *3*

SC_NO_JPEG_SUBSAMPLING = *0*

class **arena_api.enums.Visibility**

Bases: `enum.IntEnum`

An enumeration.

BEGINNER = *0*

EXPERT = *1*

GURU = *2*

INVISIBLE = *3*

UNDEFINED = *99*

arena_api.system module

class **arena_api.system._System**

Bases: `object`

The System is the entry point to the `Arena SDK`. The class is a singleton and an instance gets created when `arena_api` module is imported. Use `from arena_api.system import system` to import the system instance.

The class manages devices `Device` and the Transport Layer System node map `system.tl_system_nodemap` by :

- Maintaing a list of enumerated devices `system.device_infos`,
- creating and destroying devices `system.create_device()` and `system.destroy_device()`,
- and providing access to its node map `tl_system_nodemap`

Warning:

- The instance of `System`, `system` is created when `arena_api.system` module is imported the first time. Every other import after that returns the same instance.
- You may not import `_System` nor create it directly. Instead, Import the system instance as follows `from arena_api.system import system`.

DEVICE_INFOS_TIMEOUT_MILLISEC

Time to wait for connected devices to respond. The default value is `100` millisec.

Getter: Returns the current timeout.

Setter: Sets the timeout. expects int or float.

Type: int

When `system.device_infos` is called, the system broadcasts a discovery packet to all interfaces, waiting until the end of the timeout for any responses from enumerated devices.

The GigE Vision spec requires devices respond to a broadcast discovery packet within one second unless set otherwise `device.get_node('DiscoveryAckDelay')`.

LUCID devices are set to respond within 100 ms. Therefore, 100 works as an appropriate timeout value in many use cases.This response time can be customized through the `DiscoveryAckDelay` feature, if supported. The timeout value should reflect any such changes.

Warning:

- Slightly affects bandwidth usage due to the broadcasting of discovery packets.
- Discovers devices on all subnets, even when unable to communicate with them due to IP configuration.

---

add_unicast_discovery_device(*ip*)

Registers an IP address for a device on a different subnet than the host. Registered devices will be enumerated using unicast discovery messages. The list of remote devices will persist until they are removed using RemoveUnicastDiscoveryDevice() or until the application terminates. Unicast discovery's will be sent when UpdateDevices() is called.

Args:

ip: a string value that represents the IP address in dot-

decimal notation.

---

create_device(*device_infos=None*)

Creates and initializes `arena_api._device.Device` instance(s) from device_infos argument. The device(s) must be destroyed using `arena_api.system.destroy_device()` when no longer needed.

Args:

device_infos : can be

- A list of device info dicts which is obtained from `system.device_infos`. Also, it can be a sliced list from the full list returned from `system.device_infos`.
- A device info dict is acceptable in case the user wants to create one device from `system.device_infos` list.
- `None`. This is the default value. The system calls `system.device_infos` and create all of the devices in the returned list. in other words, if these calls as the same:
  - `system.create_device(system.device_infos)`
  - `system.create_device()`

Raises:

- `ValueError` :
  - device_infos is an empty list.
- `TypeError` :
  - device_infos type is not a list of dicts, a dict, nor None.
  - device_infos is a dict with MAC Address of a device that is not on the network.
- **BaseException :**
  - device_info is a dict and `system.device_infos` was not called.

Returns:

- A list of `arena_api._device.Device` instances.

When called, prepares each device for user interaction, opening the control channel socket and initializing all node maps. The returned device(s) are ready to stream images, send events, and read or customize features.

A single process may only create a single device once, but a single device may be opened on multiple processes. The first process to create the device is given read-write access. Additional processes are given read-only access. With read-only access, processes can read features and receive images and events; they cannot, however, write values, start the image stream or initialize events.

Warning:

- This is the only way `arena_api._device.Device` instances are created.
- Provides read-write access only to initial process that creates the device; following processes given read-only access.
- Devices must be destroyed.

———————————————————— ————————————————————-

> **`destroy_device`**(*device=None*)

Destroys and cleans up the internal memory of a `Device` instance(s). Devices that have been created `system.create_device()` must be destroyed. If not called, the system will called it when the module unloads.

> **Args:**

device: can be

- List of `arena_api._device.Device` instances obtained from `system.create_device()`. Also, it can be a sliced list from the full list returned from `system.create_device()`.
- An `arena_api._device.Device` instance is acceptable in case the user wants to destroy one device.
- `None`. This is the default value. The system destroys all of the created devices. Any device reference can not be used after calling this function

> **Raises:**

- `ValueError` :
  - Device is an empty list.
  - A device is not found in the internal connected devices internal list.
  - This function is called with a device instance but the internal connected devices list is empty.
  - the device internal C pointer has changed from Python.
- `TypeError` :
  - Device type is not a list of `arena_api._device.Device`, an `arena_api._device.Device` instance, nor None.

> **Returns:**

- None

When called, it deletes all internal memory associated with a a device: if a stream has been left open, it is closed; all node maps and chunk data adapters are deallocated; events are unregistered and the message channel closed; finally, the control channel socket is closed, allowing the device to be opened in read-write mode again.

Destroying a device does not reset device settings, and will not return a camera to a stable state. To reset settings or return to a stable state, power-cycle a device (unplug and plug back in) or reset it using `DeviceReset` node.

Warning:

- Devices must be destroyed.
- Does not affect device settings.

———————————————————— ————————————————————-

`device_infos`

A list of dictionaries used to create devices. Each dictionary represents a discovered device on the network. A device info dictionary has the following keys: `model`, `vendor`, `serial`, `ip`, `subnetmask`, `defaultgateway`, `mac`, `name`, `dhcp`, `persistentip`, `lla`, and `version`. User changes the values of the dictionary will not reflect on the device.

model: A string value that represents the model name of the discovered device. Model names are used to differentiate between products. LUCID Vision model names: PHX050S-MC, PHX032S-CC, TRI032S-MC. The model name is the same as the one received in the GigE Vision discovery acknowledgement.

vendor: A string value that represents the vendor/manufacturer name of the discovered device. Vendor names differentiate between device vendors/manufacturers. LUCID devices return 'LUCID Vision Labs'. The vendor name is the same as the one received in the GigE Vision discovery acknowledgement.

serial: A string value that represents the serial number of the discovered device. A serial number differentiates between devices. Each LUCID device has a unique serial number. LUCID serial numbers are numeric, but the serial numbers of other vendors may be alphanumeric. The serial number is the same as the one received in the GigE Vision discovery acknowledgement.

warning:

- Serial numbers from different manufacturers may overlap.

ip: A string value that represets the discovered device IP address on the network in dot-decimal notation. This IP address is the same as the one received in the GigE Vision discovery acknowledgement, but as a dot-separated string. The GigE Vision specification

only allows for IPv4 IP addresses. A device may have its IP address, subnet mask, and default gateway assigned by LLA or DHCP, set as persistent, or temporarily forced. They can be checked through the main node map nodes: `GevCurrentIPAddress`, `GevCurrentSubnetMask`, `GevCurrentDefaultGateway`.

- DHCP `GevCurrentIPConfigurationDHCP` and IP persistence `GevCurrentIPConfigurationPersistentIP` can be enabled/disabled through the node map. If both are enabled, a device will default to its persistent IP settings. If neither, it will default to LLA `GevCurrentIPConfigurationLLA`, which cannot be disabled.
- In order to configure a device to use a persistent IP configuration, not only must IP persistence be enabled `GevCurrentIPConfigurationPersistentIP`, but the IP address `GevPersistentIPAddress`, subnet mask `GevPersistentSubnetMask`, and default gateway `GevPersistentDefaultGateway` must be set.
- Forcing an IP temporarily changes an IP address, subnet mask, and default gateway of a device. A forced IP configuration will reset on a device reboot `DeviceReset`.
- A persistent IP may be quicker to enumerate than DHCP, which should be faster than LLA

subnetmask: a string value that represets the discovered device subnet mask on the network in dot-decimal notation. This subnet mask is the same as the one received in the GigE Vision discovery acknowledgement, but as a dot-separated string. The GigE Vision specification only allows for IPv4 subnet masks. A device may have its IP address, subnet mask, and default gateway assigned by LLA or DHCP, set as persistent, or temporarily forced. They can be checked through the main node map `GevCurrentIPAddress`, `GevCurrentSubnetMask`, `GevCurrentDefaultGateway`.

- DHCP `GevCurrentIPConfigurationDHCP` and IP persistence `GevCurrentIPConfigurationPersistentIP` can be enabled/disabled through the node map. If both are enabled, a device will default to its persistent IP settings. If neither, it will default to LLA `GevCurrentIPConfigurationLLA`, which cannot be disabled.
- In order to configure a device to use a persistent IP configuration, not only must IP persistence be enabled `GevCurrentIPConfigurationPersistentIP`, but the IP address `GevPersistentIPAddress`, subnet mask `GevPersistentSubnetMask`, and default gateway `GevPersistentDefaultGateway` must be set.
- Forcing an IP temporarily changes an IP address, subnet mask, and default gateway of a device. A forced IP configuration will reset on a device reboot `DeviceReset`.
- A persistent IP may be quicker to enumerate than DHCP, which should be faster than LLA.

defaultgateway: A string value represents the default gateway of the discovered device.

name: A string value that represents User-defined name of a device. If supported, it is a customizable string with a maximum of 16 bytes that can be used to identify a device `DeviceUserID`.

> **warning:**
>
> - Not necessarily supported.

dhcp: A bool value that represents whether DHCP is enabled on the discovered device. True if DHCP enabled Otherwise, false.

persistentip: A bool value that represents whether persistent IP is enabled on the discovered device. True if persistent IP enabled otherwise, false.

lla: A bool value that represents whether LLA is enabled on the device. True if LLA enabled Otherwise, false.

version: A string value that represents the version of the device currently running on the device. For LUCID devices, this refers to firmware version.

> **Warning:**
>
> - This is not guaranteed to keep the order of device infos in the list even if there is no change in the number of devices.

———————————————————— ————————————————————-

> **`force_ip`**(*device_info*)

Forces the device that matches the MAC address to a temporary new ip address, subnet mask and default gateway. The function will send a ForceIP command out on all the interfaces. This call also updates the internal list of interfaces in case that has not been done yet. The ForceIP command will be a network-wide broadcast `255.255.255.255` and will request an acknowledgment to be broadcast back to the host. The information needed to force the new ip are: MAC address of the device, ip, subnet mask, and default gateway.

> **Args:**
>
> device_infos : can be
>
> - `dict` **that has the following keys:** `mac`, `ip`, `subnetmask`, `defaultgateway`.
>
> **mac**: A string value that represents MAC address of the device to change the ip for.
>
> **ip**: A string value that represets the new IP address in dot-decimal notation.

subnetmask: A string value that represets the new subnet mask in dot-decimal notation.

defaultgateway: A string value that represets the new default gateway in dot-decimal notation.

- `list` of the formentioned dicts.

`interface_infos`

A list of dictionaries that informs about interfaces. Each dictionary represents an interface on the host. An interface info dictionary has the following keys: `ip`, `subnetmask`, and `mac`. User changes to the values of the dictionary will not reflect on the device.

ip: A string value that represets the IP address, in dot-decimal notation, of the interface on the host. An interface has its IP address, subnet mask, and MAC address checked through the Transport Layer Interface node map: `GevInterfaceIPAddress`, `GevInterfaceSubnetMask`, `GevInterfaceMACAddress`.

subnetmask: A string value that represets the subnet mask, in dot-decimal notation, of the interface on the host. An interface has its IP address, subnet mask, and MAC address checked through the Transport Layer Interface node map: `GevInterfaceIPAddress`, `GevInterfaceSubnetMask`, `GevInterfaceMACAddress`.

mac: A string value that represents the MAC address of the interface on the host. An interface has its IP address, subnet mask, and MAC address checked through the Transport Layer Interface node map: `GevInterfaceIPAddress`, `GevInterfaceSubnetMask`, `GevInterfaceMACAddress`.

`remove_unicast_discovery_device`(*ip*)

Unregisters an IP address for a device on a different subnet than the host. To remove all registered devices, pass None for the IP address argument.

> **Args:**
>
> **ip**: A string value that represents the IP address in dot- decimal notation, or None.

`select_device`(*devices=None*)

Selects one `arena_api._device.Device` instance from devices list argument.

Args:

devices : can be

- A list of `arena_api._device.Device`

Raises:

- `ValueError` :
  - devices is an empty list.
- `TypeError` :
  - devices type is not a list of `arena_api._device.Device`.

Returns:

- A `arena_api._device.Device` instance.

The returned device is ready to stream images, send events, and read or customize features.

Warning:

- The `arena_api._device.Device` instance(s) in the devices list have to be created by calling `system.create_device()`.
- Devices must be destroyed.

———————————————————— ————————————————————-

`tl_system_nodemap`

Used to access system related node.

Getter: Returns `GenTL` node map for the system

Type: `arean_api.nodemap.Nodemap` instance.

Nodes in this node map include nodes related to:

- `Arena SDK` information
- `GenTL` and GEV versioning information
- The ability to update and select interfaces
- Interface discovery and IP configuration information

Retrieves this node map without doing anything to initialize, manage, or maintain it. This node map is initialized when `arena_api` package is imported and deinitialized when the package is unloads. All available nodes can be viewed in `ArenaView` software or run `py_nodemaps_exploration.py`.

`arena_api` package provides access to five different node maps that can be splitted into two groups:

- Software:

  The following node maps describe and provide access to information and settings through the software rather than the device:

  - `system.tl_system_nodemap`
  - `system.tl_interface_nodemap`
  - `device.tl_device_nodemap`
  - `device.tl_stream_nodemap`
  - `device.tl_interface_nodemap`

- Device:

  The following node maps describe and provide access to information and settings through the device:

  - `device.nodemap`

---

arena_api._device module

class **arena_api._device.Device**(*hxdevice*)

Bases: `object`

Devices constitute the core of the `Arena SDK`, providing the means to interacting with physical devices. They are created and destroyed via `system.create_device()` and `system.destroy_device()`.

A device manages its images and chunk data , events, and node maps by:

- Starting and stopping the stream (`device.start_stream()`, `device.stop_stream()`),
- retrieving and requeuing image buffers and chunk data buffers (`device.get_buffer()`, `device.requeue_buffer()`). `Buffer` instances could contain image data only or image data appended with chunkdata,
- handling events (`device.initialize_events()`, `device.deinitialize_events()`, `device.wait_on_event()`),

- and providing access to its node maps (`tl_device_nodemap` , `tl_stream_nodemap` , `tl_interface_nodemap`).

Warning:

- Must be destroyed; otherwise, memory will leak.
- You may not import `Device` nor create it directly. Instead, use `system.create_device()` to retreave `Device` instances.

`DEFAULT_NUM_BUFFERS`

Number of internal buffers to use in the acquisition engine. The default value is `10` , and the minimum accepted value is `1` .

Getter: Returns the current default number of buffers.

Setter: Sets the default number of buffers.

Type: int

The streaming underlying engine has an input and an output queue. The size of both queues is determined by `device.DEFAULT_NUM_BUFFERS` .

Warning:

- It is recommended to keep this value relatively small.

`GET_BUFFER_TIMEOUT_MILLISEC`

Maximum time to wait for a buffer. The default value is `math.inf` , and the minimum accepted value is `0` .

Getter: Returns the current default number of buffers.

Setter: Sets the default number of buffers.

Type: int , float infinity

The value zero will return a buffer(s) if there is a ready buffer(s) in the output queue, otherwise, a `TimeoutError` will be raised. `device.GET_BUFFER_TIMEOUT_MILLISEC` initial value is `math.inf` which is a float; however, `device.GET_BUFFER_TIMEOUT_MILLISEC` only accepts `int` or `math.inf` values.

`WAIT_ON_EVENT_TIMEOUT_MILLISEC`

Maximum time to wait for an event to occur. The default value is `math.inf`, and the minimum accepted value is `0`.

Getter: Returns the current default timeout to wait for an event.

Setter: Sets the default timeout to wait for an event.

Type: int , float infinity

The value zero will return an event if there is a ready event in the events output queue, otherwise, a `TimeoutError` will be raised. `device.WAIT_ON_EVENT_TIMEOUT_MILLISEC` initial value is `math.inf` which is a float; however `device.WAIT_ON_EVENT_TIMEOUT_MILLISEC` only accepts `int` or `math.inf` values. ————————————————————————
————————————————————————-

`deinitialize_events`()

Stops the underlying events engine from listening for messages, shutting it down and cleaning it up. It should be called only after the events infrastructure has been initialized with `device.initialize_events()` and after all events have been processed with `device.wait_on_event()`.

Args:

- `None`.

Returns:

- `None`.

Roughly speaking, `device.deinitialize_events()` takes all necessary steps to undoing and cleaning up the event infrastructure's initialization. It does the following:

- Stops the listening thread.
- Closes the message channel socket.
- Unregisters all event buffers and deallocates their memory.

Warning:

- Event infrastructure must be deinitialized.
- Stops events processing.
- Deallocates event data that has not yet been processed.

———————————————————— ————————————————————-

`get_buffer`(*number_of_buffers=1*, *timeout=None*)

Retrieves, from the device, a `Buffer` instance from the buffer output queue. The function must be called after the stream has started `device.start_stream()` and before the stream has stopped `device.stop_stream()`. Retrieved buffers must be requeued `device.requeue_buffer()`.

Args:

number_of_buffers:

An `int` value that represents the number of `Buffer` instances to retrieve. The default value is `1`. Zero or a negative integer will cause a `ValueError` to throw.

timeout: can be

- A positive `int` value that represents the maximum time, in millisec, to wait for a buffer. The value zero will return a buffer(s) if there is a ready buffer(s) in the output queue, otherwise, a `TimeoutError` will be raised.
- `None`. This is the parameter's default value. The function will use `device.GET_BUFFER_TIMEOUT_MILLISEC` value instead –which has a default value of `10000`.

Raises:

- `ValueError` :
  - `number_of_buffers` parameter is less than `1` or greater than the number of buffers with which the stream has started.
  - `timeout` is a negative integer.
- `TypeError` :
  - `number_of_buffers` type is not `int`.
  - `timeout` type is not `int`
- `TimeoutError`:
  - `ArenaSDK` is not able to get a buffer(s) before the timeout expiration.
- `BaseException` :
  - `device.get_buffer()` is called before starting the stream `device.start_stream()`.
  - if the returned buffer list size is < `number_of_buffers`

Returns:

- A `Buffer` instance to manage the next buffer in the output queue, if `number_of_buffers` is `1`.
- A list of `Buffer` instances, to manage the next buffers in the output queue, The list size is equal to `number_of_buffers`.

Retrieving multiple buffers by setting `number_of_buffers` to > `1`, is the same as calling `device.get_buffer()` in a for loop and getting one buffer in each iteration.

Retrieving multiple buffers will use the same timeout to wait for each buffer.

The data returned may represent different payload types:

- an image without chunk,
- an image with chunk, or
- just chunk data.

Note that a buffer of chunk data payload type may contain image data, but cannot be cast to an image because the image data is treated as a chunk.

The payload type can be retrieved via `Buffer.payload_type`, which returns an enum `enums.PayloadType`.

When called, `device.get_buffer()` checks the output queue for image/chunk data, grabbing the first buffer(s) in the queue. If nothing is in the output queue, the call will wait until something arrives. If nothing arrives before expiration of the timeout, a `TimeoutError` is thrown.

This method is a blocking call. If it is called with a timeout of 20 seconds and nothing arrives in the output queue, then its thread will be blocked for the full 20 seconds. However, as the timeout is a maximum, as soon as something arrives in the output queue, it will be returned, not waiting for the full timeout. A timeout value of `0` ensures the call will not block, throwing instead of waiting if nothing is in the output queue.

It is a best practice to requeue buffers with `device.requeue_buffer()` as soon as they are no longer needed. If image data is needed for longer (i.e. for processing), it is recommended to copy the data `BufferFactory.copy()` and requeue the buffer.

** ——————————————————-

> `Device.start_stream()` **number of buffers parameter**
>
> VS
>
> `Device.get_buffer()` number of buffers parameter

——————————————————- **

You can start stream with 30 buffers:

Buffer available without requeue = 30 buffers

Buffers taken out and needs to be requeued = 0 buffers

- If you call `Device.get_buffer()` with no arguments to get one buffer:

  Buffer available without requeue = 29 buffers

  Buffers taken out and needs to be requeued = 1 buffers

- If you call `Device.requeue_buffer(buffer)` passing one buffer:

  Buffer available without requeue = 30 buffers

  Buffers taken out and needs to be requeued = 0 buffers

- If you call `Device.get_buffer(5)` to get 5 buffers then:

  Buffer available without requeue = 25 buffers

  Buffers taken out and needs to be requeued = 5 buffers

- If you call `Device.requeue_buffer(list_of_4_buffers)` then:

  Buffer available without requeue = 29 buffers

  Buffers taken out and needs to be requeued = 1 buffers

- If you call``Device.get_buffer(29)`` to get 29 buffers then:

  Buffer available without requeue = 0 buffers

  Buffers taken out and needs to be requeued = 30 buffers

- If you call `Device.get_buffer(_)` with any number of buffers then:

  The call to to the function will wait forever for a buffer to be requeued :(

Warning:

- Does not guarantee valid data.
- `Buffer` instance(s) should be requeued `device.requeue_buffer()`.

`initialize_events`()

Causes the underlying events engine to start listening for events. It must be called before waiting on events `device.wait_on_event()`. The event infrastructure must be turned off `device.deinitialize_events()` when no longer needed.

Args:

- `None`

Returns:

- `None`.

The underlying events engine works very similarly to the acquisition engine, except that event data is processed instead of image data. It consists of 100 buffers, an input and an output queue, and event registration information. When an event fires, the events engine takes an event buffer from the input queue, stores all relevant data, and places it in the output queue. When `device.wait_on_event()` is called, the engine takes the buffer from the output queue, processes its data, and returns it to the input queue.

More specifically, `device.initialize_events()`:

- Allocates and registers 100 buffers for the events engine.
- Places all buffers into the input queue.
- Opens a message channel socket.
- Configures the IP and port, and sets the packet size.
- Fires a dummy packet to help with firewalls.
- Starts the worker thread listening for event packets.

Events are transmitted from the device through the `GigE Vision` message channel. Arena processes event data internally, which it attaches to `device.tl_device_nodemap` using a `GenApi::EventAdapter`. The appropriate nodes are then updated in the node map. It can be helpful to incorporate callbacks to be notified when these events occur.

Warning:

- Event infrastructure must be deinitialized via `device.deinitialize_events`

———————————————————— ————————————————————-

`is_connected`()

Returns true if a device has been opened and maintains a valid communication socket. The device is opened when system.create_device() is called. If the connection to the device is lost this will return false.

Returns: - `True` or *False* `

nodemap

Used to access a device's complete feature set of nodes.

Getter: Returns main node map for the device.

Type: `arena_api._nodemap.Nodemap` instance.

The node map is built from XMLs stored on the device itself. The XML is downloaded and parsed before constructing and initializing the node map. This node map describes and provides access to all device features, and may vary from device to device. LUCID products conform to the SFNC 2.3 specification. Note that both chunk data and event data are updated on this node map.

Retrieves this node map without doing anything to initialize, manage, or maintain it. This node map is initialized when the device is created with `system.create_device()` and deinitialized when the device is destroyed with `system.destroy_device()`. All available nodes can be viewed in `ArenaView` software or run `py_nodemaps_exploration.py`.

`arena_api` package provides access to five different node maps that can be split into two groups:

- Software:

  The following node maps describe and provide access to information and settings through the software rather than the device:

  - `system.tl_system_nodemap`
  - `system.tl_interface_nodemap`
  - `device.tl_device_nodemap`
  - `device.tl_stream_nodemap`
  - `device.tl_interface_nodemap`

- Device:

  The following node maps describe and provide access to information and settings through the device:

  - `device.nodemap`

device.tl_device_nodemap vs device.nodemap:

The most noticeable difference between the two device node maps is that the `GenTL` device node map `device.tl_device_nodemap` has only a small set of features compared to the main node map `device.nodemap`. There are a few features that overlap. For example, the difference between retrieving the serial number `DeviceSerialNumber` is that using the main node map queries the camera directly whereas the `GenTL` node map queries a set of information cached at device creation. The result, however, should be the same. Basically, the `GenTL` node map queries the software for information whereas the main node map queries the device.

Warning:

- Provides access to main node map `device.nodemap`, which is not to be confused with the `GenTL` device node map `device.tl_device_nodemap`.

`requeue_buffer`(*buffers*)

Relinquishes control of a buffer(s) back to Arena. It must be called after a buffer(s) has been retrieved `device.get_buffer()`.

Args:

buffers:

The buffer(s) to requeue. It can be:

- A list of `Buffer` instances
- A `Buffer` instance.

Raises:

- `ValueError` :
  - `buffers` is an empty list.
- `TypeError` :
  - `buffers` is not a list of `Buffer` nor a `Buffer` instance.
  - `buffers` is a list but one or more element is not of `Buffer` type.

Returns:

- `None`.

When called, `device.get_buffer()` deallocates any lazily instantiated memory and returns the internal buffer to the acquisition engine's input queue, where it can be filled with new data. If enough buffers have been removed from the acquisition engine (i.e. not requeued), it is possible to starve the acquisition engine. If this happens and depending on the buffer handling mode `StreamBufferHandlingMode` node, data may start being dropped or buffers may start being recycled.

Best practices recommends that buffers be requeued as soon as they are no longer needed. If image data is needed for longer (i.e. for processing), it is recommended to copy the data `BufferFactory.copy()` and requeue the buffer.

It is important to only call `device.requeue_buffer()` on buffers retrieved from a `Device` instance , and not on images created through `BufferFactory.copy()`.

Warning:

- Used only on buffers retrieved from a device, not on buffers created through the buffer factory `BufferFactory`.

———————————————————— ————————————————————-

reset_wait_for_next_leader()

Clears any pending flag for a received leader event.

Returns: - `None`.

start_stream(*number_of_buffers=None*)

Causes the device to begin streaming image/chunk data buffers. It must be called before image or chunk data buffers are retrieved via `device.get_buffer()` otherwise, a `BaseException` will be raised.

> **Args:**
>
> number_of_buffers :
>
> Number of internal buffers to use in the acquisition engine. The default value is `None`, and the minimum accepted value is `1`. It can be:
>
> - A positive integer. Relatively small numbers are recommended. Zero or a negative int values will raise `ValueError` exception.
> - `None`. This is the default value, which is equivalent to `device.start_stream(device.DEFAULT_NUM_BUFFERS)`.

> **Raises:**
>
> - `ValueError` :
>   - `number_of_buffers` is zero or a negative intger.
> - `TypeError` :
>   - `number_of_buffers` type is not int.

> **Returns:**
>
> - None

Basically, this method prepares and starts the underlying streaming engine. The streaming engine primarily consists of a number of buffers, an input and an output queue, and a worker thread to run off of the main thread. All buffers are first placed in the input queue. When a buffer reaches its turn, it is filled with data. Once complete, it is moved to the output queue. At this point a buffer might be retrieved by the user by calling `device.get_buffer()` and then returned to the input queue by calling `device.requeue_buffer()`. More specifically:

- allocates and announces a number of buffers according to the `number_of_buffers` parameter.
- pushes all buffers to the input queue.
- opens a stream channel socket.
- configures the destination IP and port on the device.
- fires a dummy packet to help with firewalls.
- requests a test packet to ensure configured packet size is appropriate.

- starts the worker thread and begins listening for packets related to the acquisition engine.
- has the device lock out certain features (e.g. 'Width', 'Height') that cannot be changed during the stream.
- executes the `AcquisitionStart` feature in order to have the device start sending packets.

All stream configurations must be completed before starting the stream. This includes the buffer handling mode `StreamBufferHandlingMode` node found on the stream node map `device.tl_stream_nodemap`. Setting the buffer handling mode configures what the streaming engine does with buffers as they are filled and moved between queues. There are three modes to choose from:

- `OldestFirst` node is the default buffer handling mode. As buffers are filled with data, they get pushed to the back of the output queue. When a buffer is requested `device.get_buffer()`, the buffer at the front of the queue is returned. If there are no input buffers available, the next incoming buffer is dropped and the lost frame count `StreamLostFrameCount` node value is incremented.
- `OldestFirstOverwrite` node is similar to `OldestFirst` except for what happens when there are no input buffers. Instead of dropping a buffer, the oldest buffer in the output queue gets returned to the input queue so that its data can be overwritten.
- `NewestOnly` node only ever has a single buffer in the output queue. If a second buffer gets placed into the output queue, the older buffer gets returned to the back of the input queue. If there are no input buffers available, the next image is dropped and the lost frame count `StreamLostFrameCount` node value is incremented.

There are three ways to start and stop stream:

- **As a regular function call:**

  With this way, the user has control over when to call `device.stop_stream()`. For example:

```
>>> device.start_stream()
>>> # do something like grab a buffer
>>> buffer = device.get_buffer()
>>> device.requeue_buffer(buffer)
>>> # do more stuff
>>> device.stop_stream()
```

- As a context manager:

  This will call `device.stop_stream()` automatically when the context manager exits. For example:

```
>>> with device.start_stream():
>>>     # do something like grab a buffer
>>>     buffer = device.get_buffer()
>>>     device.requeue_buffer(buffer)
>>>     # do more stuff
>>> # device.stop_stream() is already called at this point
```

- As a regular function call but without calling a stop on the stream:

  This will call `device.stop_stream()` automatically when `system.destroy_device()` is called. For example:

```
>>> from arena_api.system import system
>>> devices = system.create_device()
>>> my_device = devices[0]
>>> device.start_stream()
>>> # do something like grab a buffer
>>> buffer = device.get_buffer()
>>> device.requeue_buffer(buffer)
>>> system.destory_device(my_device)
>>> # device.stop_stream() is already called at this point
```

warning:

- Stream must already be configured prior to call.
- Updates write access to certain nodes.
- May only be called once per stream without stopping.
- Minimum number of buffers is `1`.

`stop_stream()`

Stops the device from streaming image/chunk data buffers and cleans up the stream. Reverses the set up of the stream:

- Stops the worker thread.
- Shuts down the stream channel socket.
- Executes the `AcquisitionStop` feature in order to stop the device from sending packets.
- Has the device unlock features that had been locked for streaming (e.g. `Width`, `Height`).
- Revokes all buffers and cleans up their allocated memory

Args:

- `None`

Returns:

- `None`

Buffers used internally are allocated when the stream has started `device.start_stream()` and deallocated when it has stopped `device.stop_stream()`. If an image has been retrieved `device.get_buffer()`, it can be copied `BufferFactory.copy()` or saved before stopping the stream. If image data were accessed after stopping the stream, the memory would be deallocated and the behavior undefined.

Warning:

- Is an optional to call. Check `device.start_stream()` documentation.
- Updates write access to certain nodes.
- Disallows retrieval of image/chunk data from device.
- Deallocates image/chunk data that has not been copied to memory or disk.

`tl_device_nodemap`

Used to access a subset of cached device related nodes.

Getter: Returns `GenTL` node map for the device

Type: `arean_api._nodemap.Nodemap` instance.

Nodes in this node map include nodes related to:

- Device discovery information.
- GigE Vision IP configuration information.
- The ability to select streams.

Retrieves this node map without doing anything to initialize, manage, or maintain it. This node map is initialized when the device is created with `system.create_device()` and deinitialized when the device is destroyed with `system.destroy_device()`. All available nodes can be viewed in `ArenaView` software or run `py_nodemaps_exploration.py`.

`arena_api` package provides access to five different node maps that can be split into two groups:

- Software:

  The following node maps describe and provide access to information and settings through the software rather than the device:

  - `system.tl_system_nodemap`
  - `system.tl_interface_nodemap`
  - `device.tl_device_nodemap`
  - `device.tl_stream_nodemap`
  - `device.tl_interface_nodemap`

- Device:

  The following node maps describe and provide access to information and settings through the device:

  - `device.nodemap`

device.tl_device_nodemap vs device.nodemap:

The most noticeable difference between the two device node maps is that the `GenTL` device node map `device.tl_device_nodemap` has only a small set of features compared to the main node map `device.nodemap`. There are a few features that overlap. For example, the difference between retrieving the serial number `DeviceSerialNumber` is that using the main node map queries the camera directly whereas the `GenTL` node map queries a set of information cached at device creation. The result, however, should be the same. Basically, the `GenTL` node map queries the software for information whereas the main node map queries the device.

Warning:

- **Provides access to the `GenTL` device node map, not to be confused with**

the main device node map `device.nodemap`.

`tl_interface_nodemap`

Used to access interface related nodes.

Getter: Returns `GenTL` node map for the interface

Type: `arena_api._nodemap.Nodemap` instance.

Nodes in this node map include nodes related to:

- Interface discovery information
- Interface IP configuration information
- Ability to update and select devices
- Device discovery and IP configuration information

Retrieves this node map without doing anything to initialize, manage, or maintain it. This node map is initialized when `arena_api` package is imported and deinitialized when the package is unloads. All available nodes can be viewed in `ArenaView` software or run `py_nodemaps_exploration.py`.

`arena_api` package provides access to five different node maps that can be split into two groups:

- Software:

  The following node maps describe and provide access to information and settings through the software rather than the device:

  - `system.tl_system_nodemap`
  - `system.tl_interface_nodemap`
  - `device.tl_device_nodemap`
  - `device.tl_stream_nodemap`
  - `device.tl_interface_nodemap`

- Device:

  The following node maps describe and provide access to information and settings through the device:

  - `device.nodemap`

---

tl_stream_nodemap

Used to access stream related nodes.

Getter: Returns `GenTL` node map for the stream

Type: `arena_api._nodemap.Nodemap` instance.

Nodes in this node map include nodes related to:

- Stream ID and type.
- Buffer handling mode.
- Stream information such as the payload size or whether the device is currently streaming.
- Stream statistics such as lost frames, announced buffers, or missed packets.

Retrieves this node map without doing anything to initialize, manage, or maintain it. This node map is initialized when the device is created `system.create_device()` and deinitialized when the device is destroyed `system.destroy_device()`. All available nodes can be viewed in `ArenaView` software or run `py_nodemaps_exploration.py`.

`arena_api` package provides access to five different node maps that can be split into two groups:

- Software:

  The following node maps describe and provide access to information and settings through the software rather than the device:

  - `system.tl_system_nodemap`
  - `system.tl_interface_nodemap`
  - `device.tl_device_nodemap`
  - `device.tl_stream_nodemap`
  - `device.tl_interface_nodemap`

- Device:

  The following node maps describe and provide access to information and settings through the device:

  - `device.nodemap`

---

wait_for_next_leader(*timeout_millisec=None*)

Will wait until the leader for the next image buffer has arrived. It must be called after the stream has started and before the stream has stopped. This function can be used to determine when the host has received the leader for the next image buffer. Note that if the time that the camera has finished the exposure for the next buffer is desired, it is recommended to use the GenICam ExposureEnd Event instead.

Args: - timeout_millisec: can be

- A positive `int` value that represents

the maximum time, in millisec, to wait for next buffer. The value zero will return a buffer if there is a ready buffer in the output queue, otherwise, a `TimeoutError` will be raised.

Raises: - `TimeoutError`:

- `ArenaSDK` is not able to get next leader before

the timeout expiration

Returns: - `None`.

`wait_on_event`(*timeout=None*)

Waits for an event to occur in order to process its data. It must be called after the event infrastructure has been initialized `device.initialize_events()` and before it is deinitialized `device.deinitialize_events()`

Args:

- timeout: can be
    - A positive `int` value that represents the maximum time, in millisec, to wait for an event. The value zero will return an event if there is a ready event in the output queue, otherwise, a `TimeoutError` will be raised.
    - `None`. This is the parameter's default value. The function will use `device.WAIT_ON_EVENT_TIMEOUT_MILLISEC` value instead –which has a default value of `10000`.

Raises:

- `TimeoutError`:
    - `ArenaSDK` is not able to get an event before the timeout expiration

Returns:

- `None`.

Event processing has been designed to largely abstract away its complexities. When an event occurs, the data is stored in an event buffer and placed on the output queue. This method causes the data to be processed, updating all relevant nodes appropriately. This is why `device.wait_on_event()` does not return any event data; when the data is processed, nodes are updated, which can then be queried for information through the node map. This is also why callbacks work so well with the events infrastructure; they provide a method of accessing nodes of interest as they change.

When called, `device.wait_on_event()` checks the output queue for event data to process, grabbing the first buffer from the queue. If nothing is in the output queue, the call will wait until an event arrives. If nothing arrives before expiration of the timeout, a GenICam::TimeoutException is thrown.

This method is a blocking call. If it is called with a timeout of 20 seconds and nothing arrives in the output queue, then its thread will be blocked for the full 20 seconds. However, as the timeout is a maximum, when an event arrives in the output queue, the event will process, not waiting for the full timeout. A timeout value of 0 ensures the call will not block, throwing instead of waiting if nothing is in the output queue.

Warning:

- Event data processed internally.

---

arena_api._node module

class **arena_api._node.Node**(*xhnode*)

Bases: `object`

access_mode

alias_node

caching_mode

cast_alias_node

description

device_name

display_name

docu_url

event_id

fully_qualified_name

interface_type

invalidate_node()

is_cachable

is_deprecated

is_feature

is_readable

is_writable

name

namespace

polling_time

properties

tool_tip

visibility

---

class **arena_api._node.NodeBoolean**(*hxnode*)

Bases: `arena_api._node.Node`

value

---

class **arena_api._node.NodeCategory**(*hxnode*)

Bases: `arena_api._node.Node`

features

---

class **arena_api._node.NodeCommand**(*hxnode*)

Bases: `arena_api._node.Node`

execute()

is_done

---

class **arena_api._node.NodeEnumentry**(*hxnode*)

Bases: `arena_api._node.Node`

int_value

is_self_clearing

name

---

class **arena_api._node.NodeEnumeration**(*hxnode*)

Bases: `arena_api._node.Node`

enumentry_names

enumentry_nodes

value

class **arena_api._node.NodeFloat**(*hxnode*)

Bases: `arena_api._node.Node`

display_notation

display_precision

inc

inc_mode

max

min

representation

unit

value

class **arena_api._node.NodeInteger**(*hxnode*)

Bases: `arena_api._node.Node`

inc

inc_mode

max

min

representation

unit

value

---

class **arena_api._node.NodeRegister**(*hxnode*)

Bases: `arena_api._node.Node`

get(*register_length*)

set(*hsrc_register*, *src_register_length: int*)

---

class **arena_api._node.NodeString**(*hxnode*)

Bases: `arena_api._node.Node`

value

arena_api._nodemap module

---

class **arena_api._nodemap.Nodemap**(*xhnodemap*)

Bases: `object`

Only `arena_api` instantiates this class. Use the print function to get a list of all feature nodes under a nodemap instance.

device_name

Name of the device which the node map is comming from.

Getter: Returns the device name

Type: `str`

---

feature_names

A `list` of feature nodes' names.

Getter: Returns the current default number of buffers.

Type: `list` of `str`.

Warning:

- Any node becomes unavailable would not show in the feature_names list.
- Expensive to call because `arena_api` acquires all nodes in the node map then check if `_node.is_feature` evaluates to true.

---

get_node(*nodes_names*)

Gets a node or multiple nodes from the node map.

> **There are two ways to retrieve a single node:**
>
> - `nodemap.get_node('node_name')` and
> - `nodemap['node_name']`

> **Args:**
>
> nodes_names : it can be:
>
> - A `str`.
> - A `list` of `str`.
> - A `tuple` of `str`.

> **Raises:**
>
> - `ValueError` :
>   - `nodes_names` is a `list` or `tuple` but has an element that is not a `str`
>   - `nodes_names` value does not match any node name in this node map
> - `TypeError` :
>   - `nodes_names` type is not `list`, `tuple` nor `str`. The exception will suggest similar node names.

> **Returns:**
>
> - A `dict` that has node name as a key and the node as the value, if `nodes_names` is a `list`.
> - A `node` instance when `nodes_names` is a `str`.

Examples:

- Single node:

```
>>> height_node = device.nodemap.get_node('Height')
>>> height_node.value = height_node.max
>>> print(f'height value is {height_node.value} pxls)
height value is 2500 pxls
```

- Multiple nodes:

```
>>> nodes = device.nodemap.get_node(['Width', Height'])
>>> # width
>>> nodes['Width'].value = nodes['Width'].max
>>> # height
>>> height_node = nodes['Height']
>>> height_node.value = height_node.max
>>> print(f'Image buffer size will be '
>>>       f'{nodes['Width'].value} by '
>>>       f'{height_node.value} pxls')
Image buffer size will be 3000 by 2500 pxls
```

---

invalidate_nodes()

---

lock()

---

poll(*elapsed_time_millisec=None*)

Args:

elapsed_time_millisec: can be

- A positive `int` value that represents the delta of time, in millisec, to poll. The value zero causes a `ValueError` to be raised.
- `None`. This is the parameter's default value. The function will use `nodemap.DEFAULT_POLL_TIME_MILLISEC` value instead, which has a default value of `1000`.

Raises:

- `ValueError`:
  - elapsed_time_millisec is `0`.

- `TypeError` :
    - elapsed_time_millisec type is not `int` , nor `None` .

> Returns:
>
> - `None` .

`read_streamable_node_values_from`(*file_name*)

Read streamable features from a file and write to the corresponding node map

Args:

> file_name
>
> - Pass relative path name of the file stored on machine as a string.

Raises:

- `TypeError` :
    - `file_name` is not a string

Returns:

- `None` .

`try_to_lock`()

`unlock`()

`write_streamable_node_values_to`(*file_name=None*)

Read streamable features from a nodemap and save to a file

Args:

> **file_name**
>
> - Pass relative path file name as a string.
> - If None passed a file will be generated
>
> <device serialnumber>_<device user id>_<nodemap_type_name>.txt
>
> ex. 190400015_PHX050S-P_MY_DEVICE_ID_device_nodemap.txt

Raises:

- `TypeError` :
  - `file_name` is not a string

Returns:

- `None`.

arena_api.version module

Used by autodoc_mock_imports.

- arena_api is a wrapper built on top of ArenaC library, so the package uses 'ArenaCd_v140.dll' or libarenac.so. The ArenaC binary has different versions for different platforms. Here is a way to know the minimum and maximum version of ArenaC supported by the current package. This could help in deciding whether to update arena_api or ArenaC.

```
>>> pprint(arena_api.version.supported_dll_versions)
```

- For the current platform, the key 'this_platform' key can be used.

```
>>> pprint(arena_api.version.supported_dll_versions['this_platform'])
```

- Print loaded ArenaC and SaveC binaries versions.

```
>>> pprint(arena_api.version.loaded_binary_versions)
```