

Migration Guide for the Acquisition Stacks

© 2022 Stemmer Imaging AG

1. Migration Guide for the Acquisition Stacks 3
1.1 Purpose of the 3rd Generation Stack ... 3
1.2 CVB++ ... 5
1.2.1 Discover and Open ... 5
1.2.2 Single Stream ... 6
1.2.3 Ringbuffer vs Flow Set Pool ... 10
1.2.4 Camera Configuration ... 14
1.2.5 Point Cloud ... 16
1.2.6 Multi Part Image ... 17
1.2.7 Multiple Streams ... 19
1.2.8 Cancel Wait Call ... 21
1.3 CVB C# ... 22
1.3.1 Discover and Open ... 22
1.3.2 Single Stream ... 23
1.3.3 Ringbuffer vs Flow Set Pool ... 26
1.3.4 Camera Configuration ... 28
1.3.5 Point Cloud ... 30
1.3.6 Multi Part Image ... 31
1.3.7 Multiple Streams ... 32
1.3.8 Cancel Wait Call ... 34
1.4 CVBpy ... 34
1.4.1 Discover and Open ... 34
1.4.2 Single Stream ... 35
1.4.3 RingBuffer vs Flow Set Pool ... 37
1.4.4 Camera Configuration ... 39
1.4.5 Point Cloud ... 41
1.4.6 Multi Part Image ... 42
1.4.7 Multiple Streams ... 43
1.4.8 Cancel Wait Call ... 44
1.5 Service and Troubleshooting ... 44
Index 46

© 2022 Stemmer Imaging AG

1 Migration Guide for the Acquisition Stacks

1.1 Purpose of the 3rd Generation Stack

With Common Vision Blox 13.3 new image and data acquisition interfaces have been introduced in 2020. After the original IGrab and IPingPong interface from 1997 and the IGrab2 interface from 2003 this is now the 3rd generation software interface for data acquisition in Common Vision Blox. Unlike its predecessors, the 3rd generation interface now also support acquisition of multiple streams from one device, acquisition of structured data streams that contain multiple logical parts (enabling the acquisition of 3D point clouds directly from devices that support this) and acquisition into user-specified destination buffers.

Preconditions

The sole precondition for using the 3rd generation acquisition stack is to work with a so-called GenTL (generic transport layer). If you have already been using GigE Vision or USB3 Vision devices you have already relied - unwittingly - on the services of the underlying GenTLs that were installed with Common Vision Blox. These days, many more devices (like e. g. frame grabbers) support this software standard and can be used with all the features of the 3rd generation acquisition stack. Whether your device is among them can be checked using the GenICam Browser: If you have Common Vision Blox and the software package provided by your hardware manufacturer installed, simply open the GenICam Browser, which will search and display all accessible transport layers on your computer.

The Acquisition Stack so far (2nd generation)

The 2nd generation acquisition stack (based on the IGrab2 interface) allows the acquisition of streams that provide regular image buffers (with the possible option to make use of chunk data) one after the other. The buffers handed back to the caller are always allocated and owned by the driver. There is no possibility to open multiple streams or structured buffers consisting of more than one image buffer or even buffers containing different data. The use of a caller-allocated buffer is not possible with the 2nd generation stack.

To summarize, the 2nd generation stack's functionality:

- Acquisition from devices with only one stream
- Acquisition from devices with a stream containing only image data as one part
- Acquisition into buffers, which are allocated and owned by the driver

Use cases for the new Acquisition Stack (3rd generation)

The GenICam standard has over time been extended to support more possibilities than implemented in the 2nd generation acquisition stack in order to keep up with evolving requirements of camera users and camera vendors. Recently introduced features are:

- Acquisition from devices that provide more than just one data stream
- Acquisition from devices that provide a structured buffer with multiple distinct parts (subsequently called "composites")

© 2022 Stemmer Imaging AG

- Acquisition from devices that do not send a “traditional” image buffer but data like 3D point clouds or HSI cubes
- Acquisition into buffers that have been allocated by the caller (as opposed to buffers that have been allocated by the driver itself)

Architecture of the 3rd generation Acquisition Stack

To accommodate these new use cases, the acquisition engine of CVB has been modified. Where before the distinction between device and image was not immediately visible on the C-API level and only modeled into the object oriented wrappers, the new acquisition stack introduces the the following set of entities in the CVB hardware abstraction layer already in the C-API:

Device

- The Device acts as the representative of the camera/sensor in the CVB driver stack.
- On a device, one or several streams can be accessed that facilitate the data transfer from the remote device.

Stream

- A stream provides a time series of the objects that its owning device can provide. When accessing a stream provided by a device, the developer has to be aware of the device capabilities and specify whether the stream delivers images, point clouds or composites. These three cases are further explained in Stream Types.
- Each stream owns a flow set pool which defines a list of buffers into which the data coming from the stream will be acquired. A flow set pool can be provided by the caller or allocated by the driver.
- For each device and each stream, an acquisition engine can be instantiated that controls the acquisition, specifically the passing of the received buffers to the caller.

Stream Types

Depending on the hardware used for streaming, the caller can choose between three different stream options that return the corresponding object type during acquisition:

Image Stream

This stream type matches the vast majority of applications, where a standard camera is used for acquiring mono or color images. The objects acquired can be interpreted as traditional image buffers (i. e. in terms of width, height, color format). When using Image Streams to acquire from devices that supply multi part images, the stream will just provide the first image encountered in the returned composite.

Point Cloud Stream

© 2022 Stemmer Imaging AG

As the name implies, this stream type delivers the data received from a 3D acquisition device. This stream type allows an indexed access to the different parts of the point cloud data (typically coordinates in x, y and z; potentially also additional data like confidence, color, or normal vectors).

Composite Stream

If none of the above stream types properly matches what the device provides, this generic stream type offers a dynamic payload interpretation based on the other interpretable object types. For example, this object type can combine buffers holding an image with buffers holding a point clouds at the cost of increased complexity when accessing the structured data.

Cancellation Tokens

The newly introduced cancellation tokens give the possibility to stop wait calls rather than stopping the entire acquisition. This makes sense because restarting the whole acquisition takes significantly longer than reinitiating a wait for the next data.

A typical use for this would be a scenario where an external stop signal is received by the application. The fastest way to stop the acquisition and restart it once the (externally defined) conditions are right would be to abort the wait function with a cancellation token. The wait call will then return with the wait status "abort".

1.2 CVB++

- Online Reference: https://help.commonvisionblox.com/Reference/html_cpp.html

1.2.1 Discover and Open

What has changed?

The function to open a device accepts an **access token** provided by the device discovery as well as the preferred **acquisition stack**. It is possible to still use the 2nd generation acquisition stack (Vin) or to move to the 3rd generation acquisition stack (GenTL), depending on whether or not the new features like multi stream, multi part or composite streams should be used.
The following **acquisition stack** settings can be selected:

- Vin: Use Vin acquisition stack or fail.
- PreferVin: Prefer to load the Vin acquisition stack. If the Vin stack cannot be loaded try opening the non-streamable device interface before failing.
- GenTL: Use GenTL acquisition stack or fail.
- PreferGenTL: Prefer the GenTL acquisition stack. If the GenTL stack cannot be loaded first try opening the Vin stack then try opening the non-streamable device interface before failing.

Important Note

The discover method returns a list of device properties on which the functions for discovery properties can be used for retrieving device information such as MAC address or device model

© 2022 Stemmer Imaging AG

name. The list also includes a unique access token that is used for instantiating the corresponding device. The **ignore vins** flag may be passed as discovery flag to **exclude CVB *.vin drivers other than the GenICam.vin**.

Code Examples

2nd generation stack

```
1 auto devices = DeviceFactory::Discover(Cvb::DiscoverFlags::IgnoreVins);
2 // or any other flag combination e.g.
3 auto devices = DeviceFactory::Discover(Cvb::DiscoverFlags::IgnoreVins
4   | DiscoverFlags::IgnoreGevSD); // using filter driver (ignore socket driver)
5 auto device = DeviceFactory::Open<VinDevice>(devices.at(0).AccessToken(),
6   AcquisitionStack::Vin);
```

(1)(3) Discover all devices ignoring all *.vin drivers accept CVB's GenICam.vin.

(6) The Vin flag is responsible for opening the 2nd generation stack. To open non-streamable devices as well, use PreferVin instead.

3rd generation stack

```
1 auto devices = DeviceFactory::Discover(Cvb::DiscoverFlags::IgnoreVins);
2 // or any other flag combination e.g.
3 auto devices = DeviceFactory::Discover(Cvb::DiscoverFlags::IgnoreVins
4   | DiscoverFlags::IgnoreGevSD); // using filter driver (ignore socket driver)
5 auto device = DeviceFactory::Open<GenICamDevice>(devices[0].AccessToken(),
6   AcquisitionStack::GenTL);
```

(5) The GenTL flag is responsible for opening the new stack. To open non-streamable devices as well, use PreferGenTL instead.

1.2.2 Single Stream

What has changed?

In the 2nd generation acquisition stack the stream from the device was assumed to always yield images. However, depending on the hardware, a stream can also contain point clouds or any other type of data. Therefore we provide the generic stream type **composite**. A composite can be an **image**, a **multi part image** or a **point cloud**.

From the device a data stream should be instantiated with the expected stream type or the generic stream type **composite**. The different stream types are described in the chapter Stream Types. The stream type composite allows for a dynamic payload interpretation as it is composed of the other interpretable object types. For example this object type can combine buffers holding both, an image and a point cloud in the same object.

Starting the stream and waiting for the result has not changed much. If this offers advantages (like synchronously starting all streams), the stream start may now be split into **starting the engine and the device separately**, but in most cases this won't be needed.

© 2022 Stemmer Imaging AG

The next difference will be in the **wait** function to return the streamed objects. The result consists of three components: The actual **composite**, a **status code**, indicating the success of the wait, and optionally a **node map enumerator**, which can hold node maps delivering information about the received object parts. It is recommended to check the status first. If the status is not "Ok" one must assume that something went wrong and the composite does not actually hold a valid handle to newly acquired data.

Please note that with the 3rd generation acquisition stack the DeviceImage supported by the 2nd generation stack will no longer be usable and will always be null.

Code Examples

2nd generation stack

```
1  auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
2  auto device = DeviceFactory::Open(devices[0].AccessToken(),
3    AcquisitionStack::Vin);
4  auto stream = device->Stream();
5  stream->Start();
6  for (int i = 0; i < 10; ++i)
7  {
8    auto result = stream->Wait();
9    if (result.Status == WaitStatus::Ok)
10   {
11     auto timestamp = result.Image->RawTimestamp();
12   }
13 }
14 stream->Abort();
```

(4) In the 2nd generation stack the stream is always an image stream. Each device can have only one stream.

(5) The acquisition stream is started.

(8) The wait function waits until the globally set timeout for the next acquired image and (unless a timeout occurred) returns it. The function returns a result structure that combines the wait status and the acquired image. The wait status informs about the validity of the acquired image and should be checked prior to working with the image.

(14) The abort operation immediately stops the ongoing streaming. Restarting the stream will take a significant amount of time, typically in the 100 to 200 ms range.

With the 3rd generation acquisition stack it is also possible to access a set of buffer node maps that provide extended diagnostic information:

3rd generation stack

```
1  auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
2  auto device = DeviceFactory::Open<GenICamDevice>(devices[0].AccessToken(),
3    AcquisitionStack::GenTL);
4  auto stream = device->Stream<CompositeStream>(); // default: index = 0
```


© 2022 Stemmer Imaging AG

```
5  stream->Start();
6  for (int i = 0; i < 10; ++i)
7  {
8    CompositePtr composite;
9    WaitStatus status;
10   NodeMapEnumerator nodeMaps;
11   std::tie(composite, status, nodeMaps) = stream->Wait(); // C++14
12   auto [composite, status, nodeMaps] = stream->Wait(); // C++17
13 }
14 stream->Abort();
```

(4) A data stream is instantiated with index zero (default; when working with devices with more than one stream, the stream index could be passed as parameter). The stream type is defined by the template parameter of the stream query function. The returned object is a shared pointer to the stream object.

(5) The stream acquisition is simplified to the combined start, which advises the driver to start the acquisition engine and stream control mechanism automatically. A user does not have to call these separate streaming components separately unless he wants to. How to call them separately is described in Multiple Streams. By default, infinite acquisition is started.

After starting the stream, the stream engine and control are running in the background until the stream is stopped, sending a continuous flow of images or composites.

(11)(12) Each composite will need to be proactively waited for by means of a call to wait on the stream. It is possible to pass a timeout to this function, which defines the maximum time to wait for the next piece of data. The returned triple consists of three components. The actual composite, a status code and a nodemap enumerator, which are described in the introduction of this chapter.

(14) The abort operation immediately stops the ongoing stream.

The following map helps printing the return code of the reported wait status as a readable string in case of errors:

```
1  static map<WaitStatus, const char*>
2  WAIT_ERROR_STATES
3  {
4    { WaitStatus::Timeout, "timeout" },
5    { WaitStatus::Abort, "abort" }
6  };
```

```
1  switch (waitStatus)
2  {
3    default:
4      std::cout << "wait status unknown.\n";
5    case WaitStatus::Abort:
6    case WaitStatus::Timeout:
7    {
8      std::cout << "wait status not ok: " << WAIT_ERROR_STATES[waitStatus] <<
9  "\n";
```


© 2022 Stemmer Imaging AG

```
    continue;
  }
  ...
}
```

If the wait status returns Ok, a new composite has been acquired and can be processed:

```
auto firstElement = composite->ItemAt(0);
if (!holds_alternative<ImagePtr>(firstElement))
{
  std::cout << "composite does not contain an image at the first element\n";
  continue;
}
auto image = get<ImagePtr>(firstElement);
auto linearAccess = image->Plane(0).LinearAccess();
std::cout << "acquired image: " << i << " at memory location: "
  << linearAccess.BasePtr() << "\n";
```

(1) As our simple application assumes a single composite element we extract this first element. In the simplest of cases, the newly acquired composite will simply point to an image object. For composites, which hold multiple elements, check out Multi Part Image.

(2) This assumption can (and should) be verified by the holds_alternative function.

(7) The next step is to check the correct interpretation of this first element as an image.

(8) The received image now provides convenient access to the image's properties, as in this case the buffer memory address.

```
auto purpose = composite->Purpose();
std::cout << "Composite purpose: ";
switch (purpose)
{
  case CompositePurpose::Custom: std::cout << "Custom\n"; break;
  case CompositePurpose::Image: std::cout << "Image\n"; break;
  case CompositePurpose::ImageList: std::cout << "Image List\n"; break;
  case CompositePurpose::MultiAoi: std::cout << "MultiAoi\n"; break;
  case CompositePurpose::RangeMap: std::cout << "RangeMap\n"; break;
  case CompositePurpose::PointCloud: std::cout << "PointCloud\n"; break;
  case CompositePurpose::ImageCube: std::cout << "ImageCube\n"; break;
  default: std::cout << "Unknown\n"; break;
}
```

(1) The composite's purpose can be determined out by the purpose function.

Summary

- A device can be identified through a discovery information object.
- Start data acquisition by the start function. This safely starts the acquisition engine, and the data stream module, in that order.
- The abort operation immediately terminates the ongoing data acquisition process while the stop

© 2022 Stemmer Imaging AG

operation waits until the ongoing data acquisition process is completed.

1.2.3 Ringbuffer vs Flow Set Pool

What has changed?

With the 3rd generation stack it is possible to use buffers (organized in a structure named "flow set pool") managed either by the driver or allocate and passed to the driver by the user/caller. The option to use user-allocated memory is useful if, for example, the image data buffers need to satisfy conditions like e.g. address alignment (if extensions like SSE or AVX are going to be used on the data) or a particular block of memory (if the image is to be used by a GPU).

If the objective is simply to change the size of the memory pool but it is irrelevant, where on the heap the flow sets are created, then the function to register a managed flow set pool provides an easy alternative to the fully user-allocated pool.

Code Examples

2nd generation stack

```
auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
auto device = DeviceFactory::Open(devices[0].AccessToken(),
  AcquisitionStack::Vin);
auto stream = device->Stream();
if (stream->RingBuffer())
{
  stream->RingBuffer()->ChangeCount(5, DeviceUpdateMode::UpdateDeviceImage);
  stream->RingBuffer()->SetLockMode(RingBufferLockMode::On);
}
stream->Start();
std::vector<RingBufferImagePtr> images;
for (int i = 0; i < 10; ++i)
{
  if (stream->RingBuffer())
  {
    auto waitResult = stream->WaitFor<RingBufferImage>(std::chrono::seconds(5));
    switch (waitResult.Status)
    {
      case WaitStatus::Ok:
        images.push_back(waitResult.Image);
        continue;
      case WaitStatus::Timeout:
        if (imgList.size() > 0)
            imgList.front()->Unlock();
        continue;
    }
  }
}
```


© 2022 Stemmer Imaging AG

```
29 stream->Abort();
```

(5) The presence or absence of ringbuffer capability can (and should) be verified by checking the return value of the ringbuffer access function.

(7) Changes the number of buffers in the stream's ring buffer. Calling the change count function in mode update device image will discard all buffers, with which the device was created with and free the memory before reallocating the new buffers.

(8) Activates the lock mode of the ring buffer. The buffer in this case is unlocked automatically when running out of scope.

(16) Like in the simple Single Stream case, the `Wait()` function returns with the wait result that combines the stream image with the wait status.

(24) Calling the `Unlock()` method for freeing the ring buffer element is necessary in the `RingBufferLockMode::On` so that the buffer may be re-used for newly acquired images.

3rd generation stack

```
1  auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
2  auto device = DeviceFactory::Open<GenICamDevice>(devices[0].AccessToken(),
3    AcquisitionStack::GenTL);
4  auto stream = device->Stream<ImageStream>();
5  stream->RegisterManagedFlowSetPool(100);
6  stream->Start();
7  for (int i = 0; i < 10; ++i)
8  {
9    auto [image, status, nodeMaps] = stream->Wait();
10 }
11 stream->Abort();
```

(5) With the function to register a managed flow set pool, it is possible to create an internal flow set pool with the specified size. A previously registered flow set pool will be detached from the acquisition engine after the new flow set pool was created.

Large Buffer Number Change

3rd generation stack

```
1  auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
2  auto device = DeviceFactory::Open<GenICamDevice>(discoveryInfo[0].AccessToken(),
3    AcquisitionStack::GenTL);
4  auto stream = device->Stream<ImageStream>();
5  stream->RegisterManagedFlowSetPool(100);
6  stream->DeregisterFlowSetPool();
7  stream->RegisterManagedFlowSetPool(200);
8  stream->Start();
9  for (int i = 0; i < 10; ++i)
10 {
11   auto [image, result, nodeMaps] = stream->Wait();
```


© 2022 Stemmer Imaging AG

```
12 }
13 stream->Abort();
```

(6) Calling the `DeregisterFlowSetPool()` function between flow set pool registrations helps keep the memory consumption of the software low (otherwise memory usage would spike for a short moment to the sum of the currently used pool and the new pool). Note that a running stream must be stopped prior to registering or deregistering a flow set pool.

User-Allocated Memory (External Flow Set Pool)

Buffer Layout

When passing user-allocated buffers to a stream to be used as the destination memory for image acquisition, these buffers are organized in flow set pools, one of which is to be set per stream by means of the registration function for external flow set pools.

Each **flow set pool** is effectively a list of **flow sets**. The required minimum number of flow sets per flow set pool can be queried with the function for **min required flow set count**. This minimum pool size must be observed when constructing a flow set pool. A maximum pool size is not explicitly defined and is normally up to the user and the amount of available suitable memory in a given system.

The flow sets in turn are lists of **flows**. Flows can simply be thought of as buffers. However it is to a certain extent up to the camera, how this buffer will be used and therefore the simple equation 1 flow = 1 image buffer is not necessarily true. The size of these flows is a device-specific information that needs to be queried with the function size on **flow set info**.

```
 1 class UserFlowSetPool final
 2   : public Driver::FlowSetPool
 3 {
 4 using UserFlowSetPoolPtr = std::shared_ptr<UserFlowSetPool>;
 5 public:
 6   UserFlowSetPool(const std::vector<FlowInfo>& flowInfo) noexcept
 7     : Cvb::FlowSetPool(flowInfo, FlowSetPool::ProtectedTag{})
 8   {
 9   }
10   virtual ~UserFlowSetPool()
11   {
12     for (auto& flowSet : *this)
13       for (auto& flow : flowSet)
14         Tutorial::aligned_free(flow.Buffer);
15   }
16   static UserFlowSetPoolPtr Create(const std::vector<FlowInfo>& flowInfos)
17   {
18     return std::make_shared<UserFlowSetPool>(flowInfos);
19   }
20 };
```


© 2022 Stemmer Imaging AG

This example is a helper class for flow sets and derived from FlowSetPool. At the core of the class is a vector with flow sets. That vector simply holds the buffers that have been allocated for the flow set pool.

```
1  auto flowSetInfo = stream->FlowSetInfo();
2  auto flowSetPoolPtr = Tutorial::UserFlowSetPool::Create(flowSetInfo);
3  std::generate_n(std::back_inserter(*flowSetPoolPtr), NUM_BUFFERS,
4    [&flowSetInfo]()
5    {
6      auto flows = std::vector<void*>(flowSetInfo.size());
7      std::transform(flowSetInfo.begin(), flowSetInfo.end(),
8       flows.begin(), [](Driver::FlowInfo info)
9        {
10         return Tutorial::aligned_malloc(info.Size, info.Alignment);
11       });
12     return flows;
13   });
14 stream->RegisterExternalFlowSetPool(std::move(flowSetPoolPtr));
15 // acquire images
16 stream->DeregisterFlowSetPool();
```

(1) The first step is to query the required layout of the flow sets from the current stream.

(2) Then create the custom flow set pool.

(3) The flow buffers are allocated with the size and alignment information of the flow set info and stored into the flow set pool. The buffers will later be released in the destructor of the user flow set pool (line 14 in previous snippet)

(14) Then the flow set pool to be passed to the stream. Note that this transfers ownership of the flow set pool from the local scope to the stream on which the pool is registered. Once the stream ceases to exist, it will no longer need the registered flow set pool and the destructor is called. The stream also takes care of flow sets still in use by the user. Thus the user flow set pool will be deleted only when the last multipart image, point cloud or composite is deleted.

(16) Deregister the user flow set pool to free the buffers.

Summary

Using user-allocated memory for data acquisition is possible, but properly generating the pool of memory, that a stream can draw upon, is somewhat complex as it will be necessary to...

- ... make sure the stream accepts user-allocated memory in the first place
- ... make sure that the flow set pool contains enough flow sets
- ... make sure that each flow set contains the right amount of flows
- ... make sure that each flow in a set is appropriately sized
- ... make sure that the memory is valid when passed to the stream
- ... make sure that the entire data structure is disposed of correctly under all circumstances

To that end, the creation of an object hierarchy, that takes care of these intricacies, is

© 2022 Stemmer Imaging AG

recommended.

If the objective is simply to change the size of the memory pool, but it is irrelevant where on the heap the flows are created, then the function of registering managed flow set pools provides an easy alternative to the fully user-allocated pool.

1.2.4 Camera Configuration

What has changed?

The procedure for configuring a camera has not changed with the presence of the new acquisition stack. However, as the memory buffer structure has been adapted to flow set pools, (re)allocating the buffers is different. This has to be considered when changing the camera configuration after streaming resp. after the buffers have been allocated.

Code Examples

Before Streaming

If camera settings that affect the buffer layout, are changed before streaming, the buffers are automatically allocated correctly. The procedure is the same for 2nd and 3rd generation stack.

2nd generation stack

```
auto devices = Factory::Discover(DiscoverFlags::IgnoreVins);
auto device = DeviceFactory::Open(devices[0].AccessToken(),
  AcquisitionStack::Vin);
auto stream = device->Stream();
auto nodes = device->NodeMap(CVB_LIT("Device"));
auto pixelFormat = nodes->Node<EnumerationNode>(CVB_LIT("PixelFormat"));
pixelFormat->SetValue(CVB_LIT("Mono10"));
stream->Start();
for (int i = 0; i < 10; ++i)
{
  auto result = stream->Wait();
  if (result.Status == WaitStatus::Ok)
  {}
}
stream->Abort();
```

(5) Get the required node map, on which the parameter needs to be changed.

(6) Get the node to be read or written.

(7) Set the new value. In this example the pixel format is changed and the buffers will be allocated with the correct size automatically.

3rd generation stack

```
auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
auto device = DeviceFactory::Open<GenICamDevice>(devices[0].AccessToken()
```


© 2022 Stemmer Imaging AG

```
  , AcquisitionStack::GenTL);;
auto stream = device->Stream<ImageStream>();
auto nodes = device->NodeMap(CVB_LIT("Device"));
auto pixelFormat = nodes->Node<EnumerationNode>(CVB_LIT("PixelFormat"));
pixelFormat->SetValue(CVB_LIT("Mono10"));
stream->Start();
for (int i = 0; i < 10; ++i)
{
  auto [image, result, nodeMaps] = stream->Wait();
}
stream->Abort();
```

After Streaming

If camera settings, that affect the buffers, are changed *after* streaming, the buffers have to be re-allocated with the new layout. As the memory structure has changed, the updating of the buffers to the new size is done differently in 3rd generation stack.

2nd generation stack

```
auto devices = Factory::Discover(DiscoverFlags::IgnoreVins);
auto device = DeviceFactory::Open(devices[0].AccessToken(),
  AcquisitionStack::Vin);
auto stream = device->Stream();
// do some acquisition here, i.e.
//   stream->Start();
//   stream->Wait(); n times
//   stream->Abort();
auto nodes = device->NodeMap(CVB_LIT("Device"));
auto pixelFormat = nodes->Node<EnumerationNode>(CVB_LIT("PixelFormat"));
pixelFormat->SetValue(CVB_LIT("Mono10"));
device->ImageRect()->Update(DeviceUpdateMode::UpdateDeviceImage);
stream->Start();
for (int i = 0; i < 10; ++i)
{
  auto result = WaitStatus::Ok;
  auto image = stream->Wait(result);
}
stream->Abort();
```

(11) The camera setting pixel format is changed, which means, that the buffers will need to be resized.

(12) This update function call resizes the buffers. The result of this operation is that the currently active device image is moved to the new device image internally. As a side effect, the current image buffer is lost and replaced by an empty buffer.

3rd generation stack

© 2022 Stemmer Imaging AG

```
auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
auto device =
DeviceFactory::Open<GenICamDevice>(discoveryInfo[0].AccessToken()
  , AcquisitionStack::GenTL);
auto stream = device->Stream<ImageStream>();
// do some acquisition here, i.e.
//   stream->Start();
//   stream->Wait(); n times
//   stream->Abort();
auto nodes = device->NodeMap(CVB_LIT("Device"));
auto pixelFormat = nodes->Node<EnumerationNode>(CVB_LIT("PixelFormat"));
pixelFormat->SetValue(CVB_LIT("Mono10"));
stream->DeregisterFlowSetPool();
stream->Start();
for (int i = 0; i < 10; ++i)
{
  auto [image, result, nodeMaps] = stream->Wait();
}
stream->Abort();
```

(11) The camera setting "PixelFormat" is changed, which means, that the buffers will need to be resized.

(12) In the 3rd generation stack the buffers are organized in a flow set pool. To reallocate the buffers, the existing flow set pool has to be removed from the acquisition engine. Find more information on flow set pools in Ringbuffer vs Flow Set Pool.

1.2.5 Point Cloud

This example is pretty much the same as in Single Stream, only the stream type is a point cloud stream in this case.

Code Example

3rd generation stack

```
auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
auto device = DeviceFactory::Open<GenICamDevice>(devices[0].AccessToken(),
  AcquisitionStack::GenTL););
auto stream = device->Stream<PointCloudStream>();
stream->Start();
for (int i = 0; i < 10; ++i)
{
  auto [cloud, status, nodeMaps] = stream->Wait();
}
stream->Abort();
```

(4) From the device a data stream is instantiated. In our assumed scenario, only one data stream is involved, so by default, the stream at index zero is returned. The stream type is defined by the

© 2022 Stemmer Imaging AG

template specialization of the stream query function. With the 3rd generation GenTL acquisition stack it is possible to get different stream types, which are described in Stream Types.

1.2.6 Multi Part Image

Multi part basically means that the stream returns a **composite** instead of an image which may itself consist of one or more image(s), planes of point cloud(s) etc. In other words: A composite is a way to iterate over and interpret the delivered data and it will in fact be necessary to do so in order to make sense of the acquired data.

The use of a composite element's data is represented by the **composite purpose**, which can be:

- Custom: an unspecified custom composite - consult the source of the image for an interpretation.
- Image: one image, potentially with extra data.
- ImageList: multiple images grouped together.
- MultiAoi: multiple images that represent multiple AOIs in one frame.
- RangeMap: a 2.5D image, potentially with extra data.
- PointCloud: 3D data, potentially with extra data.
- ImageCube: a (hyper spectral) image cube.

The different **part types** are:

- A standard image
- A single plane of data holding, e.g for point clouds confidence data
- A collection of grouped planes, e.g. R,G and B planes
- A generic raw buffer
- A raw buffer following Pixel Format Naming Convention (PFNC) for e.g. packed data, which is not representable as an image.

Code Examples

There are multiple ways to check for multi part images. One possibility is to check the incoming composite's elements for image data:

3rd generation stack

```
auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
auto device = DeviceFactory::Open<GenICamDevice>(devices[0].AccessToken(),
  AcquisitionStack::GenTL);
auto stream = device->Stream<CompositeStream>();
stream->Start();
for (int i = 0; i < 10; ++i)
{
  auto [composite, status, nodeMaps] = stream->Wait();
  auto numElements = composite->ItemCount();
  if (numElements < 2)
    return; // no multi part
  for (auto j = 0; j < numElements; j++)
```


© 2022 Stemmer Imaging AG

```
13    {
14      auto element = composite->ItemAt(j);
15      if (!holds_alternative<ImagePtr>(element))
16      {
17        auto image = get<ImagePtr>(element);
18      }
19      else
20        std::cout << "Element is not an image\n";
21    }
22  }
23  stream->Abort();
```

(9) Extract the number of elements.

(14) Get each element from this composite.

(15) Verify the element's type using the `holds_alternative` function.

An alternative approach would be to build a `MultiPartImage` object from the composite:

3rd generation stack

```
1  auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
2  auto device = DeviceFactory::Open<GenICamDevice>(devices[0].AccessToken(),
3    AcquisitionStack::GenTL);
4  auto stream = device->Stream<CompositeStream>();
5  stream->Start();
6  for (int i = 0; i < 10; ++i)
7  {
8    auto [composite, status, nodeMaps] = stream->Wait();
9    auto multiPartImage = MultiPartImage::FromComposite(composite);
10   auto numParts = multiPartImage->NumParts();
11   for (auto j = 0; j < numParts; j++)
12   {
13     auto element = multiPartImage->GetPartAt(j);
14     if (!holds_alternative<ImagePtr>(element))
15     {
16       auto image = get<ImagePtr>(element);
17     }
18     else
19       std::cout << "Element is not an image\n";
20   }
21 }
22 stream->Abort();
```

(9) Converting the composite to a `MultiPartImage` will make will make the image data accessible. Please note that the device needs to deliver at least one composite elements that may be interpreted as a CVB image for this approach to be viable.

(10) Retrieve the number of parts within this multi part image.

(13) Get the element from the multi part image.

© 2022 Stemmer Imaging AG

(14) The type of this element can be verified by the `holds_alternative` function.

```
if (holds_alternative<ImagePtr>(part))
  std::cout << part.index() + 1 << " is an image\n";
if (holds_alternative<PlanePtr>(part))
  std::cout << part.index() + 1 << " is a plane\n";
if (holds_alternative<PlaneEnumeratorPtr>(part))
  std::cout << part.index() + 1 << " is a plane enumerator\n";
if (holds_alternative<BufferPtr>(part))
  std::cout << part.index() + 1 << " is a buffer\n";
if (holds_alternative<PFNCBufferPtr>(part))
  std::cout << part.index() + 1 << " is a pfnc buffer\n";
```

Summary

As has been shown, acquiring multi part data from a camera is not actually any more complex than acquiring a composite - the actual effort with multi part data goes into parsing the content of the received composite. Usually this will be done with the properties of the hardware in mind, i.e. like in the example above, the code usually contains assumptions about the content of the composite (that should, of course, be verified) and therefore tends to be somewhat device-specific.

1.2.7 Multiple Streams

This example expands the use case of the single data stream example to **multi stream devices**, for example devices that potentially deliver more than just one data stream with potentially different data types per stream and potentially asynchronously delivered data. An example for such a device is a multispectral camera providing multiple streams transferring different spectral information.

Code Examples

3rd generation stack

```
auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
auto device = DeviceFactory::Open<GenICamDevice>(devices[0].AccessToken(),
  AcquisitionStack::GenTL);
std::vector<ImageStreamPtr> streams;
std::generate_n(std::back_inserter(streams), device->StreamCount(),
  [&device = *device, i = 0]() mutable
  {
    return device.Stream<ImageStream>(i++);
  });
for (auto& stream : streams)
  stream->Start();
for (int i = 0; i < 10; ++i)
{
  std::vector<Driver::WaitResultTuple<MultiPartImage>> images;
```


© 2022 Stemmer Imaging AG

```
15    for (auto& stream : streams)
16      images.push_back(stream->Wait());
17  }
18  for (auto& stream : streams)
19    stream->Abort();
```

(5) Where previously only one data stream was accessed, now all available streams on the device are accessed. Therefore, the index based stream fetching is used. The number of available streams gets queried with the stream count function on the device - the queried streams are collected as image stream type. This enables parallel streaming over all streams.

(10) The approach to starting the acquisition and acquiring the data basically remains the same in the multi stream case – the only difference being that it is now necessary to start the acquisition in a loop for all streams.

(16) The multi stream wait also requires a separate processing of the streams. For simplicity, in this code snippet the `Wait()` function is called sequentially for all streams - a more reasonable real-life implementation would of course be to work with multiple threads to facilitate true asynchronous processing of the `Wait()` calls.

(18) Again, the only difference to the single stream example is that the stop/abort function needs to be called on each of the streams. Following the reverse order compared to starting the stream, now the stream control is stopped before the acquisition engines for each stream.

When working in a multi stream scenario, it might make sense to split the stream start into its two separate steps (engine start and device start). This is because the sum of both steps is significantly longer than the `DeviceStart` step alone - which can lead to notable latencies between the streams: When working with `Stream::Start()` the first stream might already have acquired some three to five images before the next one is up and running. Coding it as follows will drastically reduce this effect:

3rd generation stack

```
1  auto devices = DeviceFactory::Discover(DiscoverFlags::IgnoreVins);
2  auto device = DeviceFactory::Open<GenICamDevice>(devices[0].AccessToken(),
3    AcquisitionStack::GenTL);;
4  auto stream = device->Stream<ImageStream>();
5  std::vector<ImageStreamPtr> streams;
6  std::generate_n(std::back_inserter(streams), device->StreamCount(),
7    [&device = *device, i = 0]() mutable
8    {
9      return device.Stream<ImageStream>(i++);
10   });
11 for (auto& stream : streams)
12   stream->EngineStart();
13 for (auto& stream : streams)
14   stream->DeviceStart();
15 // your acquisition here
16 for (auto& stream : streams)
17   stream->DeviceAbort();
```


© 2022 Stemmer Imaging AG

```
18 for (auto& stream : streams)
19   stream->EngineAbort();
```

(12) The acquisition engine on all streams is started.

(14) The device acquisition is started in all streams.

(17) The device acquisition has to be either stopped or aborted. Following the reverse order compared to starting the stream: now the stream control needs to be stopped before the acquisition engine for each stream.

(19) Finally the acquisition engine is stopped / aborted on all streams.

Summary

When using multi stream devices, the sequence of actions necessary on a single stream device simply needs to be extended from 1 stream to N streams. This means that start, stop and wait for on the stream need to be called in a loop over all the required streams. When working with asynchronous streams it should be considered to put the `Wait()` calls into dedicated threads to make sure that the streams don't stall each other.

1.2.8 Cancel Wait Call

The **cancellation token** gives the possibility to cancel individual wait calls instead of the whole acquisition. Restarting the whole acquisition takes much longer, than to simply call wait functions again. A use case for example would be if an external stop signal is received by the application, the fastest reaction to stop the acquisition and to restart it, would be to abort the wait function with a cancellation token. The wait call returns with the wait status abort in this case. The token itself can be checked if it has been canceled.

Code Example

3rd generation stack

```
1 CancellationTokenSource source;
2 auto token = source.Token();
3 CompositePtr composite;
4 WaitStatus waitStatus;
5 NodeMapEnumerator enumerator;
6 std::tie(composite, waitStatus, enumerator) =
7   stream->WaitFor(std::chrono::milliseconds(10), *token);
8 source.Cancel();
9 token->IsCanceled();
```

(1) Create a cancellation token source object, which provides tokens and the signal to cancel.

(2) Get a cancellation token.

(6) Pass the cancellation token to the wait function.

(8) Cancel the wait function by using the cancellation token.

(9) Returns, whether or not the cancellation token has been canceled.

© 2022 Stemmer Imaging AG

1.3 CVB C#

- Online Reference: https://help.commonvisionblox.com/Reference/html_dotnet.html

1.3.1 Discover and Open

What has changed?

The function to open a device accepts an **access token** provided by the device discovery as well as the preferred **acquisition stack**. It is possible to still use the 2nd generation acquisition stack (Vin) or to move to the 3rd generation acquisition stack (GenTL), depending on whether or not the new features like multi stream, multi part or composite streams should be used.
The following **acquisition stack** settings can be selected:

- Vin: Use Vin acquisition stack or fail.
- PreferVin: Prefer to load the Vin acquisition stack. If the Vin stack cannot be loaded try opening the non-streamable device interface before failing.
- GenTL: Use GenTL acquisition stack or fail.
- PreferGenTL: Prefer the GenTL acquisition stack. If the GenTL stack cannot be loaded first try opening the Vin stack then try opening the non-streamable device interface before failing.

Important Note

The discover method returns a list of device properties on which the functions for discovery properties can be used for retrieving device information such as MAC address or device model name. The list also includes a unique access token that is used for instantiating the corresponding device. The **ignore vins** flag may be passed as discovery flag to **exclude CVB *.vin drivers other than the GenICam.vin**.

Code Examples

2nd generation stack

```
1  using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
2  // or any other flag combination e.g.
3  using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins
4    | DiscoverFlags.IgnoreGevSD))
5  {
6    using (var device = DeviceFactory.Open(devices[0], AcquisitionStack.Vin))
7    {
8      // work with the opened device
9    }
10 }?
```

(1)(3) Discover all devices ignoring all *.vin drivers accept CVB's GenICam.vin.

© 2022 Stemmer Imaging AG

(6) The Vin flag is responsible for opening the 2nd generation stack. To open non-streamable devices as well, use PreferVin instead.

3rd generation stack

```
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
// or any other flag combination e.g.
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins
  | DiscoverFlags.IgnoreGevSD))
{
  using (var device = DeviceFactory.Open(devices[0], AcquisitionStack.GenTL))
  {
    // work with the opened device
  }
}
```

(6) The GenTL flag is responsible for opening the new stack. To open non-streamable devices as well, use PreferGenTL instead.

1.3.2 Single Stream

What has changed?

In the 2nd generation acquisition stack the stream from the device was assumed to always yield images. However, depending on the hardware, a stream can also contain point clouds or any other type of data. Therefore we provide the generic stream type **composite**. A composite can be an **image**, a **multi part image** or a **point cloud**.

From the device a data stream should be instantiated with the expected stream type or the generic stream type **composite**. The different stream types are described in the chapter Stream Types. The stream type composite allows for a dynamic payload interpretation as it is composed of the other interpretable object types. For example this object type can combine buffers holding both, an image and a point cloud in the same object.

Starting the stream and waiting for the result has not changed much. If this offers advantages (like synchronously starting all streams), the stream start may now be split into **starting the engine and the device separately**, but in most cases this won't be needed.

The next difference will be in the **wait** function to return the streamed objects. The result consists of three components: The actual **composite**, a **status code**, indicating the success of the wait, and optionally a **node map enumerator**, which can hold node maps delivering information about the received object parts. It is recommended to check the status first. If the status is not "Ok" one must assume that something went wrong and the composite does not actually hold a valid handle to newly acquired data.

Please note that with the 3rd generation acquisition stack the DeviceImage supported by the 2nd generation stack will no longer be usable and will always be null.

Code Examples

2nd generation stack

© 2022 Stemmer Imaging AG

```
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
{
  using (var device = DeviceFactory.Open(devices[0], AcquisitionStack.Vin))
  {
    var stream = device.Stream;
    stream.Start();
    for (int i = 0; i < 10; i++)
    {
      using (var image = stream.Wait(out WaitStatus status))
      {
        // work with the acquired image
      }
    }
    stream.Abort();
  }
}
```

(5) In the 2nd generation stack the stream is always an image stream. Each device can have only one stream.

(6) The acquisition stream is started.

(9) The wait function waits until the globally set timeout for the next acquired image and (unless a timeout occurred) returns it. The function returns a result structure that combines the wait status and the acquired image. The wait status informs about the validity of the acquired image and should be checked prior to working with the image.

(13) The abort operation immediately stops the ongoing streaming. Restarting the stream will take a significant amount of time, typically in the 100 to 200 ms range.

With the 3rd generation acquisition stack it is also possible to access a set of buffer node maps that provide extended diagnostic information:

3rd generation stack

```
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
{
  using (var device = DeviceFactory.Open(devices[0], AcquisitionStack.GenTL)
as GenICamDevice)
  {
    var stream = device.GetStream<CompositeStream>(0);
    stream.Start();
    for (int i = 0; i < 10; i++)
    {
      using (var composite = stream.Wait(out WaitStatus status))
      {
        using (var nodeMaps = NodeMapDictionary.FromBuffer(composite))
        {
          // do something with the composite and the node map
        }
```


© 2022 Stemmer Imaging AG

```
      }
    }
    stream.Abort();
  }
}
```

(5) A data stream is instantiated with index zero (default; when working with devices with more than one stream, the stream index could be passed as parameter). The stream type is defined by the generic parameter of the stream query function. The returned object is a shared pointer to the stream object.

(6) The stream acquisition is simplified to the combined start, which advises the driver to start the acquisition engine and stream control mechanism automatically. A user does not have to call these separate streaming components separately. How to call them separately is described in Multiple Streams. By default, infinite acquisition is triggered.

After starting the stream, the stream engine and control are running in the background until the stream is stopped, sending a continuous flow of images or composites.

(9) Each composite will need to be proactively waited for by means of a call to wait on the stream. It is possible to pass a timeout to this function, which defines the maximum time to wait for the next piece of data. The returned value is a composite, as well as the status code and a nodemap enumerator, which are described in the introduction of this chapter.

(12) The node map dictionary needs to be deleted manually. The composite is freed automatically with the using statement, bu the node map dictionary will keep the buffer locked if not deleted.

(15) The abort operation immediately stops the ongoing streaming. Restarting the stream will take a significant amount of time, typically in the 100 to 200 ms range.

```
if (composite.Purpose == CompositePurpose.Image)
Console.WriteLine("is image");
else if (composite.Purpose == CompositePurpose.ImageList)
Console.WriteLine("is image list");
else if (composite.Purpose == CompositePurpose.MultiAoi)
Console.WriteLine("is multi aoi");
else if (composite.Purpose == CompositePurpose.RangeMap)
Console.WriteLine("is range map");
else if (composite.Purpose == CompositePurpose.PointCloud)
Console.WriteLine("is point cloud");
else if (composite.Purpose == CompositePurpose.ImageCube)
Console.WriteLine("is image cube");
else if (composite.Purpose == CompositePurpose.Custom)
Console.WriteLine("is custom");
else
Console.WriteLine("is something else");
```

(1) The composite's purpose can be found out by the purpose function.

Summary

© 2022 Stemmer Imaging AG

- A device can be identified through a discovery information object.
- Start data acquisition by the start function. This safely starts the acquisition engine, and the data stream module, in that order.
- The abort operation immediately terminates the ongoing data acquisition process while the stop operation waits until the ongoing data acquisition process is completed.

1.3.3 Ringbuffer vs Flow Set Pool

What has changed?

With the 3rd generation stack the ring buffer is replaced by the managed flow set pool.

Code Examples

2nd generation stack

```
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
{
    using (var device = DeviceFactory.Open(devices[0], AcquisitionStack.Vin))
    {
        var stream = device.Stream;
        stream.RingBuffer.ChangeCount(5, DeviceUpdateMode.UpdateDeviceImage);
        stream.RingBuffer.LockMode = RingBufferLockMode.On;
        stream.Start();
        List<StreamImage> images = new List<StreamImage>();
        for (int i = 0; i < 10; i++)
        {
            var image = stream.WaitFor(UsTimeSpan.FromSeconds(5),
              out WaitStatus status);
            if (status == WaitStatus.Ok)
            {
                images.Add(image);
            }
            else if (status == WaitStatus.Timeout && images.Count > 0)
            {
                (images[0] as RingBufferImage).Unlock();
                images.RemoveAt(0);
            }
        }
        images.Clear();
        stream.Abort();
    }
}
```

(6) Changes the number of buffers in the stream's ring buffer. Calling the `ChangeCount` function in mode update device image will discard all buffers, with which the device was created with and free the memory before reallocating the new buff. Note that the `RingBuffer` property may be `null` if the device does not support ring buffers - check before accessing if you are uncertain.

© 2022 Stemmer Imaging AG

(7) Activates the lock mode of the ring buffer. The buffer is unlocked automatically when running out of scope and the image is not stored.

(18) Wait returns with the wait result, also containing the stream image, when the ring buffer interface is available on the device.

(20) This unlocks the ringbuffer if the lock mode is on, so the buffer is returned into the acquisition queue.

3rd generation stack

```
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
{
    using (var device = DeviceFactory.Open(devices[0], AcquisitionStack.GenTL))
    {
        var stream = ((GenICamDevice)device).GetStream<ImageStream>(0);
        stream.RegisterManagedFlowSetPool(100);
        stream.Start();
        for (int i = 0; i < 10; i++)
        {
            using (var image = stream.Wait(out WaitStatus status))
            {
                using (var nodeMaps = NodeMapDictionary.FromBuffer(image))
                {
                    // do something with the composite and the node map
                }
            }
        }
        stream.Abort();
    }
}
```

(6) With the function to register a managed flow set pool, you can create an internally managed flow set pool with the given size. Any previously registered flow set pool will be removed from the acquisition engine after the new flow set pool was created.

Large Buffer Number Change

3rd generation stack

```
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
{
    using (var device = DeviceFactory.Open(devices[0], AcquisitionStack.GenTL))
    {
        var stream = device.GetStream<ImageStream>(0);
        stream.RegisterManagedFlowSetPool(100);
        stream.DeregisterFlowSetPool();
        stream.RegisterManagedFlowSetPool(200);
        stream.Start();
```


© 2022 Stemmer Imaging AG

```
        for (int i = 0; i < 10; i++)
        {
            using (var image = stream.Wait(out WaitStatus status,
              out NodeMapDictionary nodeMaps))
            {
                nodeMaps.Dispose();
            }
        }
        stream.Abort();
    }
}
```

(7) Calling the DeregisterFlowSetPool() function between flow set pool registrations helps keep the memory consumption of the software low (otherwise memory usage would spike for a short moment to the sum of the currently used pool and the new pool). Note that a running stream must be stopped prior to registering or deregistering a flow set pool.

User-Allocated Memory (External Flow Set Pool)

The external flow set pool is only available in C++: Ringbuffer vs Flow Set Pool

1.3.4 Camera Configuration

What has changed?

The procedure for configuring a camera has not changed with the presence of the new acquisition stack. However, as the memory buffer structure has been adapted to flow set pools, (re)allocating the buffers is different. This has to be considered when changing the camera configuration after streaming resp. after the buffers have been allocated.

Code Examples

Before Streaming

If camera settings that affect the buffers, are changed before streaming, the buffers are allocated correctly automatically be the acquisition stack. The procedure is the same for 2nd and 3rd generation stack.

3rd generation stack

```
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
{
    using (var device = DeviceFactory.Open(devices[0],
AcquisitionStack.GenTL))
    {
        var stream = device.Stream;
        var nodes = device.NodeMaps[NodeMapNames.Device];
        var pixelFormat = nodes["PixelFormat"] as EnumerationNode;
        pixelFormat.Value = "Mono10";
```


© 2022 Stemmer Imaging AG

```
        stream.Start();
        for (int i = 0; i < 10; i++)
        {
            using (var composite = stream.Wait(out WaitStatus status))
            {
                using (var nodeMaps = NodeMapDictionary.FromBuffer(composite))

                {
                    // do something with the composite and the node map
                }
            }
        }
        stream.Abort();
    }
}
```

(5) Get the required node map, on which the parameter needs to be changed.

(6) Get the node to be read or written.

(7) Set the new value. In this example the pixel format is changed and the buffers will be allocated with the new correct size automatically.

After Streaming

If camera settings, that affect the buffers, are changed after streaming, the buffers will need to be allocated with the new layout. As the memory structure has changed, the updating of the buffers to the new size is done differently in 3rd generation stack.

3rd generation stack

```
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
{
    using (var device = DeviceFactory.Open(devices[0],
AcquisitionStack.GenTL))
    {
        using (var stream = ((GenICamDevice)device).GetStream<ImageStream>(0))
        {
            // do a first round of image acquisition here, i.e.
            //   stream.Start()
            //   stream.Wait() n times
            //   stream.Abort()
            var nodes = device.NodeMaps[NodeMapNames.Device];
            var pixelFormat = nodes["PixelFormat"] as EnumerationNode;
            pixelFormat.Value = "Mono10";
            // deregister old flow set pool to force an update of the buffers
            stream.DeregisterFlowSetPool();
            stream.Start();
            for (int i = 0; i < 10; i++)
            {
```


© 2022 Stemmer Imaging AG

```
                using (var composite = stream.Wait(out WaitStatus status))
                {
                    using (var nodeMaps =
    NodeMapDictionary.FromBuffer(composite))
                    {
                        // do something with the composite and the node map
                    }
                }
            }
            stream.Abort();
        }
    }
}
```

(9) The camera setting "PixelFormat" is changed, which means, that the buffers will need to be resized.

(11) In the 3rd generation stack the buffers are organized in a flow set pool. To reallocate the buffers, an existing flow set pool has to be removed from the acquisition engine. Removing a flow set pool can be useful in between subsequent registrations to reduce memory consumption. Find more information on flow set pools in Ringbuffer vs Flow Set Pool.

1.3.5 Point Cloud

This example is pretty much the same as in Single Stream, only the stream type is a point cloud stream in this case.

Code Example

3rd generation stack

```
using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
{
  using (var device = DeviceFactory.Open(devices[0], AcquisitionStack.GenTL))
  {
    var stream = ((GenICamDevice)device).GetStream<PointCloudStream>(0);
    stream.Start();
    for (int i = 0; i < 10; i++)
    {
      using (var cloud = stream.Wait(out WaitStatus status,
        out NodeMapDictionary nodeMaps))
      {
        nodeMaps.Dispose();
      }
    }
    stream.Abort();
  }
}
```


© 2022 Stemmer Imaging AG

(5) From the device a data stream is instantiated. In our assumed scenario, only one data stream is involved, so by default, the stream at index zero is returned. The stream type is defined by the template specialization of the stream query function. With the 3rd generation GenTL acquisition stack it is possible to get different stream types, which are described in Stream Types.

1.3.6 Multi Part Image

Multi part basically means that the stream returns a **composite** instead of an image which may itself consist of one or more image(s), planes of point cloud(s) etc. In other words: A composite is a way to iterate over and interpret the delivered data and it will in fact be necessary to do so in order to make sense of the acquired data.

The use of a composite element's data is represented by the **composite purpose**, which can be:

- Custom: an unspecified custom composite - consult the source of the image for an interpretation.
- Image: one image, potentially with extra data.
- ImageList: multiple images grouped together.
- MultiAoi: multiple images that represent multiple AOIs in one frame.
- RangeMap: a 2.5D image, potentially with extra data.
- PointCloud: 3D data, potentially with extra data.
- ImageCube: a (hyper spectral) image cube.

The different **part types** are:

- A standard image
- A single plane of data holding, e.g for point clouds confidence data
- A collection of grouped planes, e.g. R,G and B planes
- A generic raw buffer
- A raw buffer following Pixel Format Naming Convention (PFNC) for e.g. packed data, which is not representable as an image.

Code Example

3rd generation stack

```
1  using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
2  {
3      using (var device = DeviceFactory.Open(devices[0],
4  AcquisitionStack.GenTL))
5      {
6          var stream = ((GenICamDevice)device).GetStream<ImageStream>(0);
7          stream.Start();
8          for (int i = 0; i < 10; i++)
9          {
10             using (var composite = stream.Wait(out WaitStatus status))
11             {
12
```


© 2022 Stemmer Imaging AG

```
13                  using (var nodeMaps = NodeMapDictionary.FromBuffer(composite))
14
15                  {
16                      var img = MultiPartImage.FromComposite(composite);
17                      foreach (var part in image.Parts)
18                      {
19                          switch (part)
20                          {
21                              case Image partImage:;
22                              break;
23                              case PfncBuffer buffer:
24                              break;
25                              default:
26                              break; // and more
27                          }
28                      }
29                  }
30              }
31          }
            stream.Abort();
        }
    }
```

(9) With the wait function we await a multi part image.

(14) Get the parts from the multi part image.

(18) The type of this element can be verified and handled differently.

Summary

As has been shown, acquiring multi part data from a camera is not actually any more complex than acquiring a composite - the actual effort with multi part data goes into parsing the content of the received composite. Usually this will be done with the properties of the hardware in mind, i.e. like in the example above, the code usually contains assumptions about the content of the composite (that should, of course, be verified) and therefore tends to be somewhat device-specific.

1.3.7 Multiple Streams

This example expands the use case of the single data stream example to **multi stream devices**, for example devices that potentially deliver more than just one data stream with potentially different data types per stream and potentially asynchronously delivered data. An example for such a device is a multispectral camera providing multiple streams transferring different spectral information.

Code Example

When working in a multi stream scenario, it often makes sense to split the stream start into its two separate steps (engine start and device start). This is because the sum of both steps is significantly longer than the DeviceStart step alone - which can lead to notable latencies between the streams:

© 2022 Stemmer Imaging AG

When working with Stream::Start() the first stream might already have acquired some three to five images before the next one is up and running. Coding it as follows will drastically reduce this effect:

3rd generation stack

```
1  using (var devices = DeviceFactory.Discover(DiscoverFlags.IgnoreVins))
2  {
3      using (var device = DeviceFactory.Open(devices[0], AcquisitionStack.GenTL)
4  as GenICamDevice)
5      {
6          var streams = Enumerable.Range(0, device.StreamCount).Select(i =>
7            device.GetStream<ImageStream>(i)).ToArray();
8          foreach (var stream in streams)
9              stream.EngineStart();
10         foreach (var stream in streams)
11             stream.DeviceStart();
12         // your acquisition here
13         foreach (var stream in streams)
14             stream.DeviceAbort();
15         foreach (var stream in streams)
16             stream.EngineAbort();
17     }
   }
```

(5) Where previously only one data stream was accessed, now all available streams on the device are accessed. Therefore, the index based stream fetching is used. The number of available streams gets queried with the stream count function on the device - the queried streams are collected as image stream type. This enables parallel streaming over all streams.

(8) The acquisition engine on all streams is started.

(10) The device acquisition is started in all streams.

(13) The device acquisition has to be either stopped or aborted. Following the reverse order compared to starting the stream, now the stream control is stopped before the acquisition engines for each stream.

(15) Finally the acquisition engine is stopped / aborted on all streams.

Summary

When using multi stream devices, the sequence of actions necessary on a single stream device simply needs to be extended from 1 stream to N streams. This means that start, stop and wait for on the stream need to be called in a loop over all the required streams. When working with asynchronous streams it should be considered to put the `Wait()` calls into dedicated threads to make sure that the streams don't stall each other.

© 2022 Stemmer Imaging AG

1.3.8 Cancel Wait Call

The **cancellation token** gives the possibility to cancel individual wait calls instead of the whole acquisition. Restarting the whole acquisition takes much longer, than to simply call wait functions again. A use case for example would be if an external stop signal is received by the application, the fastest reaction to stop the acquisition and to restart it, would be to abort the wait function with a cancellation token. The wait call returns with the wait status abort in this case. The token itself can be checked if it has been canceled.

Code Example

3rd generation stack

```
1 var source = new System.Threading.CancellationTokenSource();
2 var token = source.Token;
3 var image = stream.WaitFor(UsTimeSpan.FromSeconds(5), token);
4 source.Cancel();
```

(1) Create a cancellation token source object, which provides tokens and the signal to cancel.

(2) Get a cancellation token.

(3) Pass the cancellation token to the wait function.

(4) Cancel the wait function by using the cancellation token.

1.4 CVBpy

- Online Reference: https://help.commonvisionblox.com/Reference/html_pycvb.html

1.4.1 Discover and Open

What has changed?

The function to open a device accepts an **access token** provided by the device discovery as well as the preferred **acquisition stack**. It is possible to still use the 2nd generation acquisition stack (Vin) or to move to the 3rd generation acquisition stack (GenTL), depending on whether or not the new features like multi stream, multi part or composite streams should be used.
The following **acquisition stack** settings can be selected:

- Vin: Use Vin acquisition stack or fail.
- PreferVin: Prefer to load the Vin acquisition stack. If the Vin stack cannot be loaded try opening the non-streamable device interface before failing.
- GenTL: Use GenTL acquisition stack or fail.
- PreferGenTL: Prefer the GenTL acquisition stack. If the GenTL stack cannot be loaded first try opening the Vin stack then try opening the non-streamable device interface before failing.

© 2022 Stemmer Imaging AG

Important Note

The discover method returns a list of device properties on which the functions for discovery properties can be used for retrieving device information such as MAC address or device model name. The list also includes a unique access token that is used for instantiating the corresponding device. The **ignore vins** flag may be passed as discovery flag to **exclude CVB *.vin drivers other than the GenICam.vin**.

Code Examples

2nd generation stack

```
1 devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
2 # or any other flag combination e.g.
3 devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins |
4     DiscoverFlags.IgnoreGevSD)
5 with DeviceFactory.open(devices[0].access_token, AcquisitionStack.Vin) as
  device:
```

(1)(3) Discover all devices ignoring all *.vin drivers accept CVB's GenICam.vin.

(5) The Vin flag is responsible for opening the 2nd generation stack. To open non-streamable devices as well, use PreferVin instead.

3rd generation stack

```
1 devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
2 # or any other flag combination e.g.
3 devices = DeviceFactory.discover_from_root(cvb.DiscoverFlags.IgnoreVins |
4     DiscoverFlags.IgnoreGevSD)
5 with DeviceFactory.open(devices[0].access_token, AcquisitionStack.GenTL) as
  device:
```

(5) The GenTL flag is responsible for opening the new stack. To open non-streamable devices as well, use PreferGenTL instead.

1.4.2 Single Stream

What has changed?

In the 2nd generation acquisition stack the stream from the device was assumed to always yield images. However, depending on the hardware, a stream can also contain point clouds or any other type of data. Therefore we provide the generic stream type **composite**. A composite can be an **image**, a **multi part image** or a **point cloud**.

From the device a data stream should be instantiated with the expected stream type or the generic stream type **composite**. The different stream types are described in the chapter Stream Types. The stream type composite allows for a dynamic payload interpretation as it is composed of the other interpretable object types. For example this object type can combine buffers holding both, an

© 2022 Stemmer Imaging AG

image and a point cloud in the same object.

Starting the stream and waiting for the result has not changed much. If this offers advantages (like synchronously starting all streams), the stream start may now be split into **starting the engine and the device separately**, but in most cases this won't be needed.

The next difference will be in the **wait** function to return the streamed objects. The result consists of three components: The actual **composite**, a **status code**, indicating the success of the wait, and optionally a **node map enumerator**, which can hold node maps delivering information about the received object parts. It is recommended to check the status first. If the status is not "Ok" one must assume that something went wrong and the composite does not actually hold a valid handle to newly acquired data.

Please note that with the 3rd generation acquisition stack the DeviceImage supported by the 2nd generation stack will no longer be usable and will always be null.

Code Examples

2nd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.Vin) as
device:
    stream = device.stream()
    stream.start()
    for _ in range(10):
        image, status = stream.wait()
        with image:
            pass
    stream.abort()
```

(3) In 2nd generation stack the stream is always an image stream. A device can have only one stream.

(4) The acquisition stream is started.

(6) The wait function waits until the globally set timeout for the next acquired image and (unless a timeout occurred) returns it. The function returns a result tuple that combines the wait status and the acquired image. The wait status informs about the validity of the acquired image and can be checked prior to working with the image.

(9) The abort operation immediately stops the ongoing streaming.

3rd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.GenTL) as
device:
    stream = device.stream(CompositeStream)  // default: index = 0
    stream.start()
    for _ in range(10):
```


© 2022 Stemmer Imaging AG

```
        composite, status, node_maps = stream.wait()
        with composite:
            pass
    stream.abort()
```

(3) A data stream is instantiated with index zero (default). The type is defined by the parameter passed to the stream query function. The returned object is a stream object.

(4) The stream acquisition is simplified to the combined start, which advises the driver to start the acquisition engine and stream control mechanism automatically. A user does not have to call these separate streaming components separately. How to call them separately is described in Multiple Streams. By default, infinite acquisition is triggered.

After starting the stream, the stream engine and control are running in the background until the stream is stopped, sending a continuous flow of images or composites.

(6) Each composite will need to be proactively waited for by means of a call to `wait()` on the stream. It is possible to pass a timeout to this function, which defines the maximum time to wait for the next piece of data. The returned triple consists of three components. The actual composite, a status code and a nodemap enumerator, which are described in the introduction of this chapter.

(9) The abort operation immediately stops the ongoing streaming.

Summary

- A device can be identified through a discovery information object.
- Start data acquisition by the start function. This safely starts the acquisition engine, and the data stream module, in that order.
- The abort operation immediately terminates the ongoing data acquisition process while the stop operation waits until the ongoing data acquisition process is completed.

1.4.3 RingBuffer vs Flow Set Pool

What has changed?

In 3rd generation stack the ring buffer is replaced by the managed flow set pool.

Code Examples

2nd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.Vin) as device:
    stream = device.stream()
    stream.ring_buffer.change_count(5, DeviceUpdateMode.UpdateDeviceImage)
    stream.ring_buffer.lock_mode = RingBufferLockMode.On
    stream.start()
    images = []
    for _ in range(10):
        image, status = stream.wait_for(5000)
```


© 2022 Stemmer Imaging AG

```
        if status == WaitStatus.Ok:
            images.append(image)
        elif (status == WaitStatus.Timeout and len(images) > 0):
            first = images.pop(0)
            first.unlock()
    stream.abort()
```

(4) Change the number of buffers in this ring buffer. Calling the change count function in mode update device image will discard all buffers, with which the device was created with and free the memory. Note that if the device does not support ring buffers the value of `ring_buffer` is `None` - when in doubt it's a good idea to check before accessing it.

(5) Activates the lock mode of the ring buffer. The buffer is unlocked automatically when running out of scope and the image is not stored.

(9) Wait returns with the wait result, also containing the stream image, when the ring buffer interface is available on the device.

(14) This unlocks the ringbuffer if the lock mode is on, so the buffer is returned into the acquisition queue.

3rd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.GenTL) as
device:
    stream = device.stream(ImageStream)
    stream.register_managed_flow_set_pool(100)
    stream.start()
    for _ in range(10):
        image, status, node_maps = stream.wait()
        with image:
            pass
    stream.abort()
```

(4) With the function to register a managed flow set pool, it is possible to create an internal flow set pool with the given size. Any previously registered flow set pool will be removed from the acquisition engine after the new flow set pool was created.

User-Allocated Memory (External Flow Set Pool)

The external flow set pool is only available in C++: Ringbuffer vs Flow Set Pool

Large Buffer Number Change

3rd generation stack

```
import cvb
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
access_token = devices[0].access_token
```


© 2022 Stemmer Imaging AG

```
with DeviceFactory.open(access_token, AcquisitionStack.GenTL) as device:
    stream = device.stream(ImageStream)
    stream.register_managed_flow_set_pool(100)
    stream.deregister_flow_set_pool()
    stream.register_managed_flow_set_pool(200)
    stream.start()
    for _ in range(10):
        image, status, node_maps = stream.wait()
        with image:
            pass
    stream.abort()
```

(7) Calling the deregister flow set pull function between consecutive registrations reduces the memory consumption. The stream must be stopped.

1.4.4 Camera Configuration

What has changed?

The procedure for configuring a camera has not changed with the presence of the new acquisition stack. However, as the memory buffer structure has been adapted to flow set pools, (re)allocating the buffers is different. This has to be considered when changing the camera configuration after streaming resp. after the buffers have been allocated.

Code Examples

Before Streaming

If camera settings that affect the buffers, are changed before streaming, the buffers are automatically allocated correctly. The procedure is the same for 2nd and 3rd generation stack.

2nd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.Vin) as
device:
    stream = device.stream()
    nodes = device.node_maps["Device"]
    pixel_format = nodes["Std::PixelFormat"]
    pixel_format.value = "Mono10"
    stream.start()
    for _ in range(10):
        image, status = stream.wait()
        if status == WaitStatus.Ok:
            pass
    stream.abort
```

(4) Get the required node map, on which the parameter needs to be changed.

(5) Get the node to be read or written.

© 2022 Stemmer Imaging AG

(6) Set the new value. In this example the pixel format is changed and the buffers will be allocated with the correct size automatically.

3rd generation stack

```
devices = DeviceFactory.discover_from_root(cvb.DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.GenTL) as
device:
    stream = device.stream(ImageStream)
    nodes = device.node_maps["Device"]
    pixel_format = nodes["Std::PixelFormat"]
    pixel_format.value = "Mono10"
    stream.start()
    for _ in range(10):
        image, status, node_maps = stream.wait()
        with image:
            pass
    stream.abort()
```

After Streaming

If camera settings, that affect the buffers, are changed after streaming, the buffers will need to be allocated with the new layout. As the memory structure has changed, the updating of the buffers to the new size is done differently in 3rd generation stack.

2nd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.Vin) as
device:
    stream = device.stream()
    # do some acquisition here
    nodes = device.node_maps["Device"]
    pixel_format = nodes["Std::PixelFormat"]
    pixel_format.value = "Mono10"
    stream.image_rect.update(DeviceUpdateMode.UpdateDeviceImage)
    stream.start()
    for _ in range(10):
        image, status = stream.wait()
        if status == WaitStatus.Ok:
            pass
    stream.abort()
```

(7) The camera setting pixel format is changed, which means, that the buffers will need to be resized.

(8) This update function call resizes the buffers. The result of this operation is that the currently active device image is moved to the new device image internally. As a side effect, the current image buffer is lost and replaced by an empty buffer.

© 2022 Stemmer Imaging AG

3rd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.GenTL) as
device:
    stream = device.stream(ImageStream)
    # do some acquisition here
    nodes = device.node_maps["Device"]
    pixel_format = nodes["Std::PixelFormat"]
    pixel_format.value = "Mono10"
    stream.deregister_flow_set_pool()
    stream.start()
    for _ in range(10):
        image, status, node_maps = stream.wait()
        with image:
            pass
    stream.abort()
```

(7) The camera setting pixel format is changed, which means, that the buffers will need to be resized.

(8) In the 3rd generation stack the buffers are organized in a flow set pool. To reallocate the buffers, an existing flow set pool has to be removed from the acquisition engine. Removing a flow set pool can be useful in between subsequent registrations to reduce memory consumption. Find more information on flow set pools in Ringbuffer vs Flow Set Pool.

1.4.5 Point Cloud

This example is pretty much the same as in Single Stream, only the stream type is a point cloud stream in this case.

Code Example

3rd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.GenTL) as
device:
    stream = device.stream(PointCloudStream)
    stream.start()
    for _ in range(10):
        cloud, status, node_maps = stream.wait()
        with cloud:
            pass
    stream.abort()
```

(3) From the device a data stream is instantiated. In our assumed scenario, only one data stream is involved, so by default, the stream at index zero is queried. The stream type is defined by the

© 2022 Stemmer Imaging AG

template specialization of the stream query function. With 3rd generation GenTL acquisition stack it is possible to get different stream types, which are described in Stream Types.

1.4.6 Multi Part Image

Multi part basically means that the stream returns a **composite** instead of an image which may itself consist of one or more image(s), planes of point cloud(s) etc. In other words: A composite is a way to iterate over and interpret the delivered data and it will in fact be necessary to do so in order to make sense of the acquired data.

The use of a composite element's data is represented by the **composite purpose**, which can be:

- Custom: an unspecified custom composite - consult the source of the image for an interpretation.
- Image: one image, potentially with extra data.
- ImageList: multiple images grouped together.
- MultiAoi: multiple images that represent multiple AOIs in one frame.
- RangeMap: a 2.5D image, potentially with extra data.
- PointCloud: 3D data, potentially with extra data.
- ImageCube: a (hyper spectral) image cube.

The different **part types** are:

- A standard image
- A single plane of data holding, e.g for point clouds confidence data
- A collection of grouped planes, e.g. R,G and B planes
- A generic raw buffer
- A raw buffer following Pixel Format Naming Convention (PFNC) for e.g. packed data, which is not representable as an image.

Code Example

3rd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.GenTL) as
device:
    stream = device.stream(ImageStream)
    stream.start()
    for _ in range(10):
        image, status, node_maps = stream.wait()
        with image:
            for part in image:
                if isinstance(part, Image):
                    pass
    stream.abort()
```

(8) Get the parts of the image.

© 2022 Stemmer Imaging AG

(9) The type of this element can be verified by the isinstance function.

Summary

As has been shown, acquiring multi part data from a camera is not actually any more complex than acquiring a composite - the actual effort with multi part data goes into parsing the content of the received composite. Usually this will be done with the properties of the hardware in mind, i.e. like in the example above, the code usually contains assumptions about the content of the composite (that should, of course, be verified) and therefore tends to be somewhat device-specific.

1.4.7 Multiple Streams

This example expands the use case of the single data stream example to **multi stream devices**, for example devices that potentially deliver more than just one data stream with potentially different data types per stream and potentially asynchronously delivered data. An example for such a device is a multispectral camera providing multiple streams transferring different spectral information.

Code Example

3rd generation stack

```
devices = DeviceFactory.discover_from_root(DiscoverFlags.IgnoreVins)
with DeviceFactory.open(devices[0].access_token, AcquisitionStack.GenTL) as
device:
    streams = [device.stream(ImageStream, i) for i in
range(device.stream_count)]
    for stream in streams:
        stream.engine_start()
    for stream in streams:
        stream.device_start()
    for _ in range(10):
        images = [stream.wait() for stream in streams]
        try:
            pass
        finally:
            for image, status, nodes in images:
                with image:
                    pass
    for stream in streams:
        stream.device_abort()
    for stream in streams:
        stream.engine_abort()
```

(3) Where previously only one data stream was accessed, now all available streams on the device are accessed. Therefore, the index based stream fetching is used. The number of available streams gets queried with the stream count function on the device - the queried streams are collected as image stream type. This enables parallel streaming over all streams.

© 2022 Stemmer Imaging AG

(5) The acquisition engine on all streams is started.

(7) The device acquisition is started in all streams.

(17) The device acquisition has to be either stopped or aborted. Following the reverse order compared to starting the stream, now the stream control is stopped before the acquisition engines for each stream.

(19) Finally the acquisition engine is stopped / aborted on all streams.

Summary

When using multi stream devices, the sequence of actions necessary on a single stream device simply needs to be extended from 1 stream to N streams. This means that start, stop and wait for on the stream need to be called in a loop over all the required streams. When working with asynchronous streams it should be considered to put the `Wait()` calls into dedicated threads to make sure that the streams don't stall each other.

1.4.8 Cancel Wait Call

The **cancellation token** gives the possibility to cancel individual wait calls instead of the whole acquisition. Restarting the whole acquisition takes much longer, than to simply call wait functions again. A use case for example would be if an external stop signal is received by the application, the fastest reaction to stop the acquisition and to restart it, would be to abort the wait function with a cancellation token. The wait call returns with the wait status abort in this case. The token itself can be checked if it has been canceled.

Code Example

3rd generation stack

```
source = cvb.CancellationTokenSource
token = source.token
image, status, node_maps = stream.wait(token)
source.cancel()
token.is_canceled()
```

(1) Create a cancellation token source object, which provides tokens and the signal to cancel.

(2) Get a cancellation token.

(3) Pass the cancellation token to the wait function.

(4) Cancel the wait function by using the cancellation token.

(5) Check, whether the cancellation token has been canceled.

1.5 Service and Troubleshooting

© 2022 Stemmer Imaging AG

Technical Support

Before contacting the Technical Support please

- save logs and hardware configuration settings.

Please also have the version numbers of Common Vision Blox, Operating System and the devices serial numbers ready.

✓ Request Hotline Support

✓ Discuss in User Forum

© 2022 Stemmer Imaging AG

- 2 -

2nd Generation Acquisition Stack 3

- 3 -

3rd Generation Acquisition Stack 3

- A -

Access Token (C#) 22
Access Token (C++) 5
Access Token (Python) 34
Acquisition 3
Acquisition Stack 3
Architecture 3

- C -

C# 3
C++ 3
Camera Configuration (C#) 28
Camera Configuration (C++) 14
Camera Configuration (Python) 39
Cancel Wait (C#) 34
Cancel Wait (C++) 21
Cancel Wait (Python) 44
Cancellation Tokens 3
CancellationToken (C#) 34
CancellationToken (C++) 21
CancellationToken (Python) 44
Composite Purpose (C#) 31
Composite Purpose (C++) 17
Composite Purpose (Python) 42
Composite Stream 3
Contact 44
CVB.Net 3
CVB++ 3
CVBpy 3

- D -

Device 3
Discover (C#) 22
Discover (C++) 5
Discover (Python) 34
Documentation 44

- E -

External Flow Set Pool 10, 26, 37

- F -

Flow Set Pool (Python) 37
Flow Set Pool (C#) 26
Flow Set Pool (C++) 10
Forum 44

- H -

Hotline 44

- I -

Image Stream 3

- M -

Multi Part Image (C#) 31
Multi Part Image (C++) 17
Multi Part Image (Python) 42
Multiple Streams (C#) 32
Multiple Streams (C++) 19
Multiple Streams (Python) 43

- N -

Nodemap (C#) 28
Nodemap (C++) 14
Nodemap (Python) 39
Nodes (C#) 28
Nodes (C++) 14
Nodes (Python) 39

- O -

Open (C#) 22
Open (C++) 5
Open (Python) 34

- P -

Part (C#) 31
Part (C++) 17
Part (Python) 42

© 2022 Stemmer Imaging AG

Point Cloud (C#) 30
Point Cloud (C++) 16
Point Cloud (Python) 41
Point Cloud Stream 3
Preconditions 3
Purpose 3
Python 3

- R -

Ringbuffer (C#) 26
Ringbuffer (C++) 10
Ringbuffer (Python) 37

- S -

Single Stream (C#) 23
Single Stream (C++) 6
Single Stream (Python) 35
Stream 3
Stream Types 3
Support 44

- T -

Technical Support 44

- U -

Use Cases 3
User Forum 44
User-Allocated Memory 10, 26, 37

© 2022 Stemmer Imaging AG