JAI SDK to eBUS SDK Migration Guide

Overview

This guide is designed to help current users of the JAI SDK migrate their applications to the new eBUS SDK by Pleora. Since there's no one-to-one correspondence between the functions in the two SDKs, this guide aims to shows how similar camera operations as a whole can be performed in each SDK and points out any significant differences.

The guide assumes that the reader is familiar with the JAI SDK and how to use it with GenICam cameras. Code examples will be shown using the C/C++ API in both SDKs since this is most commonly used API.

1. SDK Differences in a Nutshell

	JAI SDK	eBUS SDK
Platforms	Windows, Linux (x86)	Windows
Cost	Free	Free but unlicensed cameras have watermarked images
Licensing	No license required	License required for non-JAI cameras
API languages	C, .NET languages	C++, .NET languages
Windows development requirements	Visual Studio 2005 through 2012	Visual Studio 2008, 2010, 2012, 2013, and 2015
Camera interfaces supported	GigE, USB3, GenTL (CXP, CameraLink)	GigE, USB3
Pixel Formats supported	All formats used by JAI cameras, most PFNC formats	All formats used by JAI cameras, most PFNC formats
Object Oriented Programming	Classes only in .NET API	All APIs are class-based

Figure 1 - SDK basic features comparison

2. Current JAI SDK functionality not in the eBUS SDK

- Automatic ForceIP
- HDR on host
- Image flip and rotate
- Color histograms
- Lookup Tables on host
- Special image processing functions like color correction and lens distortion correction

3. Comparison of basic operation between the SDKs

3.1 Enumerating and opening cameras

In the JAI SDK the factory object has complete knowledge of the system and is how cameras can be discovered and accessed. In the eBUS SDK the PvSystem object plays a similar role.

Note that in the JAI SDK objects can only be accessed through handles while in eBUS they are user-accessible classes.

JAI SDK:

```
J_STATUS_TYPE rc;
FACTORY_HANDLE hFactory;
CAM_HANDLE hCamera;
bool8_t bHasChanged;
uint32_t iNumCameras;
int8_t sCameraId[J_CAMERA_ID_SIZE];
uint32_t size;

// Open the Factory
rc = J_Factory_Open((int8_t*)"" , &hFactory);

// Search for cameras on all interfaces
rc = J_Factory_UpdateCameraList(hFactory, &bHasChanged);

// Get the number of cameras
rc = J_Factory_GetNumOfCameras(hFactory, &iNumCameras);

// Get camera ID of first camera
size = sizeof(sCameraId);
rc = J_Factory_GetCameraIDByIndex(hFactory, 0, sCameraId, &size);

// And open the camera and get a handle to it
rc = J_Camera_Open(hFactory, sCameraId, &hCamera);
```

eBUS SDK:

```
PvResult lResult;
PvSystem lSystem;
uint32_t iNumCameras;
PvDevice* plDevice;

// Search for cameras on all interfaces
lResult = lSystem.Find();

// Get the number of cameras
iNumCameras = lSystem.GetDeviceCount();

// Get pointer to DeviceInfo of first camera
const PvDeviceInfo* plDeviceInfo = lSystem.GetDeviceInfo(0);

// Create and connect to camera as a PvDevice
```

```
// Note: To access GigE or USB3-specific camera attributes the pointer to
PvDevice will
// need to be explicitly cast to the subclass PvDeviceGEV or PvDeviceU3V
plDevice = PvDevice::CreateAndConnect(plDeviceInfo, &lResult);
```

3.2 Accessing camera features

Accessing camera features is relatively trivial with the JAI SDK while eBUS requires a bit more setup to access a feature.

JAI SDK:

```
// Set exposure time to 5000us, note that ExposureTime is a float feature
rc = J_Camera_SetValueDouble(hCamera, "ExposureTime", 5000.0);
```

eBUS SDK:

```
// Access the exposure time node as a pointer to a GenICam float feature and
set it to 5000 us
PvGenParameterArray*  plDeviceGenParams = plDevice->GetParameters();
PvGenFloat* plExposureTime = dynamic_cast<PvGenFloat *>(plDeviceGenParams-
>Get("ExposureTime"));
lResult = plExposureTime->SetValue(5000.0);
// Note that this could also be done in a single line like so:
//lResult = dynamic_cast<PvGenFloat *>(plDevice->GetParameters()-
>Get("ExposureTime"))->SetValue(5000.0);
```

3.3 Setting up streaming and acquiring images

In the JAI SDK the DataStream object is responsible for setting up the flow of images from the camera to the host and notifying the user that a new image buffer has arrived. The mechanics of waiting on a new buffer, retrieving it, and ultimately re-queuing it are left up to the user as well as allocating/freeing buffers. An acquisition callback function will handle all of these tasks automatically but in an acquisition thread they must be done explicitly by the user.

In eBUS SDK there is a PvStream class which functions similarly to the JAI data stream object. However, there is an additional class, PvPipeline, which manages acquired buffers and automates some of the buffer-related tasks that the JAI data stream object does not handle. A PvPipeline object associated with a PvStream is basically a loop that continually checks for new buffers. If a new buffer arrives and the user has requested one through a function like RetrieveNextBuffer, it is passed to the user. Otherwise the buffer is immediately re-queued to the stream. This keeps the image output queue from backing up and the driver from running out of free buffers. The PvPipeline class also handles automatically re-sizing buffers if the size of streamed images changes.

3.3.1 Using callback functions

One of the biggest differences in the two SDKs is how acquisition using callback functions is handled. The JAI SDK has a single function for registering a callback function that will be called

whenever a new buffer is ready. Acquisition by callback function is easy but generally not the most efficient way of handling and processing buffers since a copy of each buffer must be made before the function returns. In the eBUS SDK setting up and using a callback function is more involved which is probably a subtle way of discouraging its use.

To get the equivalent functionality as the JAI SDK it's necessary to sub-class the virtual class PvPipelineEventSink and implement the function OnBufferReady which will be called whenever a new buffer has been retrieved from the stream by the pipeline.

[Note that since OnBufferReady is in a separate class and only receives a pointer to the pipeline, it will not have direct access to any display windows and cannot display images. See the sample StreamCallbackSample to see how can be handled via a custom constructor.]

JAI SDK:

```
uint32_t iImageSize; // size of image in bytes
THRD_HANDLE hThread;

// Register the acquisition callback function and then open stream
void *vfptr = reinterpret_cast<void*>(AcquisitionCBFunc);
J_IMG_CALLBACK_FUNCTION *cbfptr =
reinterpret_cast<J_IMG_CALLBACK_FUNCTION*>(&vfptr);
rc = J_Image_OpenStream(hCamera, 0, NULL, *cbfptr, &hThread, iImageSize);

// Acquisition call back function
static void __stdcall AcquisitionCBFunc(J_tIMAGE_INFO *pAqImageInfo)
{
      // Image data is available in pAqImageInfo->pImageBuffer
      // Do processing of image here and return when done
}
```

eBUS SDK:

```
// Derived class to handle pipeline events
class MyPipelineEventSink : public PvPipelineEventSink
{
public:
      MyPipelineEventSink(void);

      // New buffers will be received and displayed in this function
      void OnBufferReady(PvPipeline *poPipeline);
};


…

// Callback function that's called when a new buffer has been
delivered to the pipeline
void MyPipelineEventSink::OnBufferReady(PvPipeline *poPipeline)
{
      PvBuffer* poBuffer = NULL;
```

```
      PvResult oResult, oOperationResult;

      // Get next available buffer, timeout after 1000ms
      oResult = poPipeline->RetrieveNextBuffer(&poBuffer, 1000,
&oOperationResult);
      if (oResult.IsOK() && oOperationResult.IsOK())
      {
            // Do something with buffer here
            //
      }
      else
      {
            // Handle error(s)
      }
}
```

3.3.2 Using threads

Since thread functions are very OS-specific, we have chosen to show threads as implemented in Windows.

To use an acquisition thread in the JAI SDK it's necessary to create a DataStream object attached to the camera and a DataStream event that is registered to the EVENT_NEW_BUFFER event. This can be done outside of the thread or within the thread as long as these variable are accessible in the thread.

Then loop on the function call J_Event_WaitForCondition and wait for the condition to be met. Then the buffer information can be read out field by field into a J_tIMAGE_INFO structure. This image buffer can be processed within the thread or passed as a pointer to some other processing. When you are finished with the buffer, it's necessary to re-queue it using J_DataStream_QueueBuffer to make it available for the driver to use again.

[Because of the number of steps involved rather than describe each step, I will refer the user to the JAI sample programs StreamThreadSample or ConsoleExampleFullAcq.]

By contrast using acquisition threads with eBUS is much simpler because much of the setup and work is already handled by the PvPipeline object. Most of the thread consists of just looping and blocking on PvPipeline::ReceiveNextBuffer until a new buffer arrives.

JAI SDK:

```
int main(int argc, _TCHAR* argv[])
{
      ...
      // Create the thread
      hAcqStreamThread = CreateThread(NULL, NULL,
(LPTHREAD_START_ROUTINE)AcquisitionThread, &index, NULL, NULL);
```

```
	// Start acquisition on the camera, thread should be ready to
receive buffers
	rc = J_Camera_ExecuteCommand(hCamera,
(int8_t*)"AcquisitionStart");
	...
}

// Acquisition thread function
void AcquisitionThread(LPVOID lpdwThreadParam)
{
	J_STATUS_TYPE rc;

	STREAM_HANDLE hDataStream = (STREAM_HANDLE) lpdwThreadParam;
	uint32_t iSize;
	BUF_HANDLE iBufferID;
	HANDLE hCondition;
	EVT_HANDLE hStreamEvent;
	J_COND_WAIT_RESULT WaitResult;
	EVENT_NEW_BUFFER_DATA eventData; // Struct for EventGetData
	J_tIMAGE_INFO tAqImageInfo = {0, 0, 0, 0, NULL, 0, 0, 0, 0, 0, 0};

	// Create the condition used for signalling the new image event
	rc = J_Event_CreateCondition(&hCondition);

	// Create a stream event for new frame notification
	gtCamInfo[iCamNum].hStreamEvent = CreateEvent(NULL, true, false,
NULL);

	// Register the event and associated condition with the
acquisition engine
	rc = J_DataStream_RegisterEvent(hDataStream, EVENT_NEW_BUFFER,
hCondition, &hStreamEvent);

	// Start image acquisition
	rc = J_DataStream_StartAcquisition(hDataStream,
ACQ_START_NEXT_IMAGE, ULLONG_MAX);

	// Acquisition loop
	while (gbAcqThreadEnabled) {
		// Wait for Buffer event (or kill event) or timeout after
1000ms
		rc = J_Event_WaitForCondition(hCondition, 1000, &WaitResult);

		// Did we get a new buffer event?
		if (J_COND_WAIT_SIGNAL == WaitResult) {
			uint64_t iFramesPending = 0;
			uint64_t iRawPixelFormat;
			uint64_t iReadValue;
```

```
                // Get the Buffer Handle from the event
                iSize = (uint32_t)sizeof(EVENT_NEW_BUFFER_DATA);
                rc = J_Event_GetData(hStreamEvent, &eventData,
&iSize);
                iBufferID = eventData.BufferHandle;

                // Fill in complete tAqImageInfo structure field by
field
                // Get frame width
                iSize = sizeof (size_t);
                rc =
J_DataStream_GetBufferInfo(gtCamInfo[iCamNum].hDataStream, iBufferID,
BUFFER_INFO_WIDTH, &iReadValue, &iSize); CHECK_RC(rc,
"J_DataStream_GetBufferInfo failed");
                tAqImageInfo.iSizeX = (uint32_t) iReadValue;
                // Get frame height
                iSize = sizeof (size_t);
                rc =
J_DataStream_GetBufferInfo(gtCamInfo[iCamNum].hDataStream, iBufferID,
BUFFER_INFO_HEIGHT, &iReadValue, &iSize); CHECK_RC(rc,
"J_DataStream_GetBufferInfo failed");
                tAqImageInfo.iSizeY = (uint32_t) iReadValue;
                ....


                      // Do any processing with image buffer here
                      //
                      // Then queue this buffer again for reuse in
acquisition engine
                      // or pass the buffer pointer/index on to some
other thread that will requeue it when done
                      rc = J_DataStream_QueueBuffer(hDataStream,
iBufferID);


            }
      }

      // Stop streaming
      rc = J_DataStream_StopAcquisition(hDataStream,
ACQ_STOP_FLAG_KILL);

      // Unregister new buffer event
      rc = J_DataStream_UnRegisterEvent(hDataStream, EVENT_NEW_BUFFER);


      // Free the event object
      J_Event_Close(hStreamEvent);
```

```
        // Free the Condition
        J_Event_CloseCondition(hCondition);

        // End of thread function
}
```

eBUS SDK:

```
// Create and open camera stream
PvStream *lStream = PvStream::CreateAndOpen(plDeviceInfo, &lResult);
// Cast to specific stream interface type
PvStreamGEV *lStreamGEV = static_cast<PvStreamGEV *>(lStream);
// Configure device streaming destination (only needed for GigE cameras)
lResult = lDeviceGEV->SetStreamDestination(lStreamGEV-
>GetLocalIPAddress(), lStreamGEV->GetLocalPort());

// Create pipeline object
PvPipeline* lPipeline = new PvPipeline(lStream);
if (lPipeline != NULL)
{
    // And set the Buffer size and the Buffer count
    lPipeline->SetBufferSize(lDeviceGEV->GetPayloadSize());
    lResult = lPipeline->SetBufferCount(BUFFER_COUNT);
}

// Start acquisition thread here and pass in pointers to PvDeviceGEV,
PvStream, and PvPipeline via lpParameter
hAcqStreamThread = CreateThread(NULL, NULL,
(LPTHREAD_START_ROUTINE)AcquisitionThread, lpParameters, NULL, NULL);
…

// Thread function which continually acquires frames from a camera
void AcquisitionThread(LPVOID lpParameters)
{
    PvResult lResult;
    PvDeviceGEV *lDevice = (PvDeviceGEV*)lGEVDevice;
    // Obtain pointers PvDeviceGEV *lGEVDevice, PvPipeline* lPipeline,
PvStream* lStream from lpParameters somehow


    // Get device parameters and map the AcquisitionStart and
AcquisitionStop commands
    PvGenParameterArray *lDeviceParams = lDevice->GetParameters();
    PvGenCommand *lAcqStart = dynamic_cast<PvGenCommand
*>(lDeviceParams->Get("AcquisitionStart"));
    PvGenCommand *lAcqStop = dynamic_cast<PvGenCommand *>(lDeviceParams-
>Get("AcquisitionStop"));
```

```
    // Start pipeline
    lResult = lPipeline->Start();

    // Enable streaming
    lResult = lDevice->StreamEnable();

    // Send Start command
    lResult = lAcqStart->Execute();

    // Loop and block until the next buffer is available, timeout after
1000ms
    PvBuffer *lBuffer = NULL;
    PvResult lOperationResult;
    while (bLoopCondition == true)
    {
        lResult = lPipeline->RetrieveNextBuffer(&lBuffer, 1000,
&lOperationResult);
        if (lResult.IsOK() && lOperationResult.IsOK())
        {
            // Do something with buffer
            //

            // Release the buffer back to the pipeline
            lResult = lPipeline->ReleaseBuffer(lBuffer);
        }
    }

    // Now send Stop command
    lResult = lAcqStop->Execute();

    // Disable streaming on the device
    lResult = lDevice->StreamDisable();

    // Stop the pipeline
    lResult = lPipeline->Stop();
}
```

3.4 Freeing resources and closing cameras

In the JAI SDK cleanup consists of closing any handles and calling the appropriate close function for JAI objects in reverse order of their creation. If buffers were manually allocated, these will also need to be removed from driver use and then freed.

In eBUS cleanup is relatively straightforward. Any created objects should be stopped or closed in reverse order of their creation and their destructors will handle the freeing of any allocated resources.

JAI SDK:

```
// Stop acquisition and wait for any in-flight buffers to arrive
rc = J_Camera_ExecuteCommand(hCamera, "AcquisitionStop");
Sleep(300); // 300ms should be more than enough

// Handle buffer cleanup
// Flush image queues in case there are images pending then unprepare
and delete buffers
J_DataStream_FlushQueue(hDataStream, ACQ_QUEUE_INPUT_TO_OUTPUT);
J_DataStream_FlushQueue(hDataStream, ACQ_QUEUE_OUTPUT_DISCARD);
for(i = 0 ; i < NUM_OF_BUFFERS; i++) {
      // Remove each buffer from the acquisition engine
      void *pBufferPtr, *pPrivateInfo;
      J_DataStream_RevokeBuffer(hDataStream, pAcqBufferID[i],
&pBufferPtr , &pPrivateInfo);
      if (pAcqBuffer[i]) {
            delete pAcqBuffer[i];
      }
      pAcqBuffer[i] = NULL;
      pAcqBufferID[i] = 0;
}

// Close image stream thread handle
CloseHandle(hAcqStreamThread);

// Close data stream
rc = J_DataStream_Close(hDataStream);

// Close view window
rc = J_Image_CloseViewWindow(hView);

// Close the camera
rc = J_Camera_Close(hCamera);

// Close the factory
rc = J_Factory_Close(hFactory);
```

eBUS SDK:

```
// Send AcquisitionStop command
lResult = lAcqStop->Execute();

// Disable streaming on the device
lResult = lDevice->StreamDisable();

// Stop the pipeline
lResult = lPipeline->Stop();

// Clean up display window if created
```

```
if (poDisplay) {
      poDisplay->Close();
      delete poDisplay;
      poDisplay = NULL;
}

// Close and free stream
lResult = lStreamList[i]->Close();
PvStream::Free(lStreamList[i]);
lStreamList[i] = NULL;

// Disconnect device
PvDevice *lDevice = (PvDevice*)lGEVDeviceList[i];
PvDevice::Free(lDevice);
lGEVDeviceList[i] = NULL;
```

3.5 .NET API differences

Since the PvDotNet classes are just wrapper classes around the C++ classes, there are almost no appreciable differences between the two APIs. It should be relatively easy to translate C++ classes and methods to their .NET equivalents.

4. FAQ – Additional Questions

4.1 I have JAI CXP cameras – can I use the eBUS SDK with them?

No. eBUS does not support GenTL so it cannot be used to control cameras that use third-party vendors' framegrabbers for acquisition.

4.2 Can eBUS co-exist with Cognex VisionPro?

Unfortunately, it cannot. VisionPro uses a licensed version of Pleora's Universal Pro GigE driver that is not compatible with the version included with eBUS and this prevents both packages from being installed on the same system.

Author: Gordon Rice (gr@jai.com)

End.

Revision History

Revision	Date	Changes
1	2019/04/02	New release