JAI SDK から eBUS SDK への移行ガイド

概要
本書では、現在 JAI SDK を使用されているお客様が Pleora 社の eBUS SDK にアプリケーションを移行するにあたって役立つ情報を紹介しています。JAI SDK と eBUS SDK の機能は 1 対 1 では対応していません。本書では、各 SDK で類似したカメラ操作を示しながら、大きく異なる点について解説します。

本書は、読者が JAI SDK と GenICam 準拠のカメラの使用方法に精通していることを前提としています。サンプルコードは、双方の SDK で最も普及している C/C++ API を使って解説します。

1. JAI SDK と eBUS SDK の相違点

	JAI SDK	eBUS SDK
プラットフォーム	Windows、Linux（x86）	Windows
コスト	無料	無料 ただし、ライセンスのないカメラで使用した場合は、画像に透かしが入ります。
ライセンス	不要	JAI のカメラ以外で使用する場合はライセンスが必要
API 言語	C、.NET 言語	C++、.NET 言語
Windows 開発要件	Visual Studio 2005～2012	Visual Studio 2008、2010、2012、2013、2015
サポートしているカメラインタフェース	GigE、USB3、GenTL（CXP、CameraLink）	GigE、USB3
サポートしているカメラフォーマット	JAI カメラで使用されているすべてのフォーマット、ほとんどの PFNC フォーマット	JAI カメラで使用されているすべてのフォーマット、ほとんどの PFNC フォーマット
オブジェクト指向プログラミング	.NET API のみがクラスベース	全ての API がクラスベース

Figure 1 - SDK の基本機能の比較

2. 現在の JAI SDK にあって、eBUS SDK にはない機能

- 自動 ForceIP
- ホスト PC 上での HDR 合成
- 画像の反転と回転
- カラーヒストグラム
- ホスト PC 上にルックアップテーブルを定義
- 色補正やレンズ歪み補正などの特殊な画像処理機能

3. 基本操作の比較

3.1. カメラの列挙とオープン

JAI SDK では、ファクトリオブジェクトがシステムについての完全なナレッジを持ち、どのようにカメラを検出してアクセスするかを指定します。eBUS SDK では、PvSystem オブジェクトが同様の役割を担います。

ご注意：オブジェクトは、JAI SDK ではハンドルを介してのみアクセスが可能ですが、eBUS SDK ではユーザーがアクセス可能なクラスとなります。

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. カメラ機能へのアクセス

JAI SDK ではカメラ機能へのアクセスは比較的容易ですが、eBUS SDK では少々セットアップが必要です。

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 ストリーミングの設定と画像の取得

JAI SDK では、DataStream オブジェクトがカメラからホスト PC への画像の流れを設定し、新しい画像バッファが届いたことをユーザーに通知する役目を担っています。新しいバッファを待機して検索し、最終的にそれを再キューイングするという手順は、バッファの割り当て／解放と同様にユーザーに任されます。取得コールバック関数はすべてのタスクを自動的に処理しますが、取得スレッドはユーザーによって明示的に実行される必要があります。

eBUS SDK では、JAI データストリームオブジェクトと同様に機能する PvStream クラスがあります。ただし、PvPipeline という追加のクラスがあり、JAI データストリームオブジェクトが処理しないバッファ関連のタスクの一部を自動化します。PvStream に関連付けられた PvPipeline オブジェクトは、基本的に新しいバッファを継続してチェックするというループ処理のオブジェクトです。新しいバッファが届いたときに、ユーザーが RetrieveNextBuffer などの関数を介して要求していると、ユーザーにバッファが渡されます。そうでない場合は、直ちにストリームに再キューイングされます。

3.3.1 Using callback functions コールバック関数の使用

2 つの SDK の最大の相違点として、コールバック関数を使用した取得の処理方法があります。JAI SDK には、新しいバッファの準備ができたときに必ず呼び出されるコールバック関数を登録するための 1 つの関数があります。コールバック関数による取得は容易ですが、関数が戻る前に各バッファのコピーを作成しなければならないため、バッファのハンドリングやバッファ処理をする方法としては一般的に効率的であるとはいえません。一方 eBUS SDK でも、コールバック関数の設定と使用がより複雑になるため、使い勝手に影響を与えることもあります。

JAI SDK と同等の機能性を得るには、仮想クラスである PvPipelineEventSink をサブクラス化し、パイプラインによってストリームから検索されるたびに呼び出される OnBufferReady 関数を実装する必要があります。

[ご注意：OnBufferReady は別のクラスに属し、パイプラインへのポインタを受け取るだけなので、どの表示ウィンドウにも直接アクセスできず、画像の表示もできません。詳しくは、サンプルコード

の StreamCallbackSample を参照し、カスタムのコンストラクタを処理する方法を確認してください。]

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 スレッドの使用

スレッド関数は OS 固有であるため、本書では Windows で実装されているスレッドを例にとって説明します。

JAI SDK で取得スレッドを使用するには、カメラに付加された DataStream オブジェクトと、EVENT_NEW_BUFFER イベントに登録された DataStream イベントを作成する必要があります。これらの変数がスレッド内でアクセス可能であれば、スレッドの外部または内部で実行できます。

次に、関数呼び出しの J_Event_WaitForCondition をループし、条件が満たされるのを待ちます。その後、バッファ情報をフィールドごとに J_tIMAGE_INFO 構造体に読み出すことが可能になります。この画像バッファは、スレッド内で処理することも、その他の処理にポイントとして渡すこともできます。バッファの処理が終了したら、J_DataStream_QueueBuffer を使ってバッファを再度キューに入れ、再びドライバを使用できるようにする必要があります。

.

[ステップ数が多いため、ここでは説明を省略します。各ステップについては、JAI のサンプルプログラム StreamThreadSample または ConsoleExampleFullAcq を参照してください。]

これとは対照的に、eBUS SDK ではセットアップと作業の多くはすでに PvPipeline オブジェクトによって処理されているため、JAI SDK に比べてはるかに容易です。ほとんどのスレッドは、新しいバッファが届くまで PvPipeline::ReceiveNextBuffer をループおよびブロックするだけとなります。

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. リソースの解放とカメラのクローズ

JAI SDK では、クリーンアップは、ハンドルのクローズと適切なクローズ関数の呼び出しで構成されます。つまり、JAI オブジェクトの作成とは逆の手順となります。バッファが手動で割り当てられている場合は、ドライバでの使用から削除して、解放する必要があります。

eBUS SDK では、クリーンアップは比較的容易です。オブジェクトの作成とは逆の手順で停止またはクローズする必要がありますが、デストラクタが割り当てられているリソースを解放します。

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 の相違点

PvDotNet クラスは C++のラッパークラスであるため、双方の API に大きな違いはありません。C++のクラスやメソッドを.NET の同等のものに変換するのは比較的容易です。

4. FAQ - よくあるご質問

4.1. CoaXPress インタフェース搭載の JAI 製カメラに対して、eBUS SDK を使うことはできますか？

使えません。eBUS SDK は GenTL をサポートしていないため、サードパーティ製のフレームグラバを使って取得するカメラ制御には対応しておりません。

4.2. eBUS SDK は Cognex VisionPro を共存させることはできますか？

残念ながら使えません。VisionPro が使用している Pleora 社の Universal Pro GigE ドライバのライセンスバージョンは、eBUS に含まれているバージョンとは互換性がありませんので、両パッケージが同じシステムにインストールされないようになっています。

Author: Gordon Rice (gr@jai.com)執筆者：Gordon Rice（gr@jai.com）

リビジョン履歴

リビジョン	日付	変更内容
0	2019/04/02	新規リリース