日立国際電気

GigE カメラ SDK

マニュアル

Version 1.01

1．SDK の構造

インストールマニュアル（別紙）に従って HitachiGigECamera.exe のインストールが完了すると、次の場所に SDK ファイルが展開されます。

C:¥Program Files¥Hitachi_GigE¥SDK　　：32bit OS にインストールした場合のデフォルト展開フォルダ
C:¥Program Files (x86)¥Hitachi_GigE¥SDK　　：64bit OS にインストールした場合のデフォルト展開フォルダ

フォルダ階層は次のようになっています。

```
*SDK -- *bin ----------- *Release -------- GigaCam_siple.exe, Sample.exe, Sample_vc6.exe :サンプル実行ファイル
     - *Common ------ Version.h : バージョン情報取得用ヘッダファイル
     - *Doc ---------- GigEConnectSDK.chm        : SDK の説明ヘルプファイル英文／和文
                       GenApi.chm                : EMVA 提供の GenAPI リファレンス

     - *GigaCam_simple : ビュアーソフト GigaCam（簡易版）ソースコード
     - *Includes ------- registers.h : カメラレジスタ用ヘッダファイル
     - *Lib ----------- TLI.lib     : ライブラリ
     - *Sample         : VC7/VC8/VC9 用サンプルソースコード
     - *Sample_VB6     : VB6 用サンプルソースコード
     - *Sample_VC6     : VC6 用サンプルソースコード
     - *Sample_VCS     : VC#用サンプルソースコード
     - *ZipLib         : サンプルに使用されているフリーウェア ZIP ソースコード
```

＊: フォルダを表します

２．GenICam API

GenICam は様々なインタフェースを使用してカメラとのやり取りが可能です。代表的なものを下記に挙げます。

Integer	：	整数を扱います
Enumeration	：	列挙値を扱います
Float	：	実数を扱います
String	：	文字列を扱います
Boolean	：	ブール数を扱います
Command	：	実行形式のコマンドです

※GenICam コマンドは Visual C++6 と Visual Basic 6 では使用できません。

これらのインタフェースのプロパティについて次ページから説明します。GenAPI についての詳細は"GenApi.chm"（英文）をご参照ください。

➢ Integer interface

```
interface   IInteger : virtual public IValue
{
    // Set node value
    virtual void SetValue(int64_t Value, bool Verify = true) = 0;

    // Set node value
    virtual IInteger& operator=(int64_t Value) = 0;

    // Get node value
    virtual int64_t GetValue(bool Verify = false) = 0;

    // Get node value
    virtual int64_t operator()() = 0;

    // Get node value
    virtual int64_t operator*() = 0;

    // Get minimum value allowed
    virtual int64_t GetMin() = 0;

    // Get maximum value allowed
    virtual int64_t GetMax() = 0;

    // Get increment
    virtual int64_t GetInc() = 0;

    // Get recommended representation
    virtual  ERepresentation GetRepresentation() = 0;

};

//! SmartPointer for IInteger interface pointer
typedef CPointer<IInteger> CIntegerPtr;
```

➢ Enumeration interface

```
interface   IEnumeration  : virtual public IValue
{
    // Get list of symbolic Values
    virtual void GetSymbolics(StringList_t & Symbolics) = 0;

    // Get list of entry nodes
    virtual void GetEntries(NodeList_t & Entries) = 0;

    // Set string node value
    virtual IEnumeration& operator=(const GenICam::gcstring& ValueStr) = 0;

    // Set integer node value
    virtual void SetIntValue(int64_t Value, bool Verify = true) = 0;

    // Get string node value
    virtual GenICam::gcstring operator*() = 0;

    // Get integer node value
    virtual int64_t GetIntValue(bool Verify = false) = 0;

    // Get an entry node by name
    virtual IEnumEntry *GetEntryByName(const GenICam::gcstring& EntryName) = 0;

};

//! SmartPointer for IEnumeration interface pointer
typedef CPointer<IEnumeration> CEnumerationPtr;
```

➢ Float interface

```
interface   IFloat  : virtual public IValue
{
    // Set node value
    virtual void SetValue(double Value, bool Verify = true) = 0;

    // Set node value
    virtual IFloat& operator=(double Value) = 0;

    // Get node value
    virtual double GetValue(bool Verify = false) = 0;

    // Get node value
    virtual double operator()() = 0;

    // Get node value
    virtual double operator*() = 0;

    // Get minimum value allowed
    virtual double GetMin() = 0;

    // Get maximum value allowed
    virtual double GetMax() = 0;

    // Get recommended representation
    virtual  ERepresentation GetRepresentation() = 0;

    // Get the physical unit name
    virtual GenICam::gcstring GetUnit() const = 0;

};

//! SmartPointer for IFloat interface pointer
typedef CPointer<IFloat> CFloatPtr;
```

➢ String interface

```
interface   IString  : virtual public IValue
{
    // Set node value
    virtual void SetValue(const GenICam::gcstring& Value, bool Verify = true) = 0;

    // Set node value
    virtual IString& operator=(const GenICam::gcstring& Value) = 0;

    // Get node value
    virtual GenICam::gcstring GetValue(bool Verify = false) = 0;

    // Get node value
    virtual GenICam::gcstring operator()() = 0;

    // Get node value
    virtual GenICam::gcstring operator*() = 0;

};

//! SmartPointer for IString interface pointer
typedef CPointer<IString> CStringPtr;
```

➢ Boolean interface

```
interface   IBoolean   : virtual public IValue
{
    // Set node value
    virtual void SetValue(bool Value, bool Verify = true) = 0;

    // Set node value
    virtual void operator=(bool Value){ SetValue( Value ); }

    // Get node value
    virtual bool GetValue() const = 0;

    // Get node value
    virtual bool operator()() const { return GetValue(); }

};

//! SmartPointer for IBoolean interface pointer
typedef CPointer<IBoolean> CBooleanPtr;
```

➢ Command interface

```
interface ICommand   : virtual public IValue
{
    // Execute the command
    virtual void Execute() = 0;

    // Execute the command
    virtual void operator()() = 0;

    // Query whether the command is executed
    virtual bool IsDone() const = 0;

};

//! SmartPointer for ICommand interface pointer
 typedef CPointer<ICommand> CCommandPtr;
```

3. 画像取り込みのチュートリアル

カメラの認識から画像取り込みを行うまでのもっともシンプルな手順を紹介します。

○ 必須用件

アプリケーションをコンパイルしたりリンクしたりするためには

- GenTL インクルードファイル（Client.h も含む）
- TLI.lib インポートライブラリ
- TLI.dll 取り込み DLL
- GenAPI インクルードヘッダファイル

が必要です。

○ 関数の戻り値

全てのライブラリ関数は関数の処理が成功したかどうかを表す GC_ERROR 型のステータスを返します。成功時の戻り値は ERR_SUCCESS(0)で、他の戻り値は失敗の種類によって変わります。

本チュートリアルでは

```
GC_ERROR gc;
```

と定義されたローカル変数 gc を使用します。

○ 名前空間

GenICam モデルに従って、全ての関数、変数およびデータ型が定義されるいくつかの名前空間をライブラリが定義しています。

本チュートリアルでは便宜上名前空間接頭語の代わりに#include 記述の後ろに下記のコードを記述します：

```
using namespace GenICam;
using namespace GenICam::Client;
using namespace GenApi;
```

○ ライブラリのオープン

ライブラリを開くためにコード上で TLOpen()を呼びます。パラメータはライブラリへのハンドルを持たせる変数へのポインタです。

```
TL_HANDLE hTL;
gc = TLOpen(&hTL);
```

○ インタフェースの列挙

最初に、利用可能なインタフェースの数を獲得します。これは TLGetNumInterfaces()を呼ぶことで達成されます：

```
uint32_t num_ifaces;
gc = TLGetNumInterfaces( hTL, &num_ifaces) ;
```

次に、TLGetInterfaceName()を呼んでインタフェースの名前を獲得します：

```
char iface_name[256];
size_t cb = sizeof(iface_name);
gc = TLGetInterfaceName(hTL, 0, iface_name, &cb );
```

第 2 引数に 0 を指定しているのは最初に見つかったものの名前を獲得するということです。

以降 n 番目に見つかったものの名前を取得したい時は n-1 を指定してください。

○ インタフェースのオープン
インタフェースを開くために、ライブラリへのハンドルとインタフェース名を指定して TLOpenInterface()を呼びます：

```
INTERFACE_HANDLE hIF;
gc = TLOpenInterface( hTL, iface_name, &hIF );
```

○ デバイスの列挙
インタフェース上のデバイスを列挙することはインタフェースを列挙することとよく似ています。最初に TLGetNumDevices()を呼びます：

```
Uint32_t num_devices;
gc = TLGetNumDevices( hIF, &num_devices );
```

次に、指定したインデックス（このケースでは最初に見つかったもの：0）のデバイスの名前を TLGetDeviceName()で獲得します：

```
char dev_name[256];
gc = TLGetDeviceName( hIF, 0, dev_name, sizeof(dev_name) );
```

※ デバイスの IP アドレス指定
見つかったデバイスのサブネットが接続しているインタフェースのサブネットと異なるときは、デバイスに対して IP アドレスを設定します：

```
char mac_address[32];   // フォーマット：  "00:40:31:XX:XX:XX"          XX はカメラによって異なる
uint32_t ip_address;
uint32_t subnet_mask;
uint32_t default gateway
gc = TLSetIPAddres( hIF, mac_address, ip_address, subnet_mask, default_gateway );
```

○ デバイスのオープン
インタフェースとデバイスの名前を獲得した後にデバイスを開く準備をします。これは TLOpenDevice()を呼ぶことで達成されます：

```
DEV_HANDLE hDev;
gc = TLOpenDevice( hIF, device_name, &hDev );
```

○ デバイスレジスタへのアクセス
デバイスレジスタに Write することでカメラを操作することができ、デバイスレジスタを Read することでデバイスのパラメータを獲得することができます。デバイスの独立性を達成しかつアクセスをはるかに簡単にするために、数値レジスタではなく文字列レジスタ名を通してアクセス可能です。
ライブラリはデバイスレジスタを Read/Write するための関数：GCReadData()/GCWriteData()を提供します。これらの関数は低レベルインタフェースの一部であり、そのためレジスタアドレスを指定する必要があります。
しかし、本チュートリアルではこれらの関数を直接使用する代わりに、文字列名を対応するレジスタアドレスへと効果的に変換させる GenAPI ライブラリを使います。デバイスレジスタにアクセスするために GenAPI ライブラリは 2 つのことを必要とします：

- GCReadData()と GCWriteData()を使って実データ転送を実行するために GenAPI によって呼ばれるメソッドを持つ、ユーザが生成するポートオブジェクト
- レジスタ記述 XML ファイル

最初に、実レジスタ I/O を実行するために GenAPI によって呼ばれるポートオブジェクトを生成します。IPort インタフェースとそのメソッドである Read()と Write()を実装するクラスを次のように宣言します：

```
class MyPort: public IPort
{
public:
	MyPort()
	{
	};
	void SetDeviceHandle( DEV_HANDLE hDev )
	{
		m_hDev = hDev;
	}
protected:
	virtual void Read(void *pBuffer, int64_t Address, int64_t Length)
	{
		if( GC_ERR_SUCCESS != GCReadData( m_hDev, Address, pBuffer, (size_t*)&Length ) )
			throw ACCESS_EXCEPTION( "Failure reading device data" );
	};
	virtual void Write(const void *pBuffer, int64_t Address, int64_t Length)
	{
		if( GC_ERR_SUCCESS != GCWriteData( m_hDev, Address, pBuffer, (size_t*)&Length ) )
			throw ACCESS_EXCEPTION( "Failure writing device data" );
	};
	virtual EAccessMode GetAccessMode() const
	{
		return RW;
	};
```

Read()メソッドとWrite()メソッドはデバイスレジスタへの実際の Read/Write を行います。それを完了させるためにはただ対応するライブラリ関数を呼ぶだけです。
GetAccessMode()メソッドはポートアクセスモードを獲得するために GenAPI によって必要とされます。このクラスは RW(Read/Write)を返します。
このポートクラス宣言の次のステップは、GenAPI は正確にレジスタネームをアドレスへと変換できるようにデバイスレジスタを記述している XML の取得です。これは、ファイルの位置を獲得するために最初に DevGetURL()を呼ぶことで達成されます。

```
char url[2048];
size_t url_len = sizeof( url );
gc = GCGetURL( hDev, url, &url_len );
```

現在、レジスタ記述ファイルの URL を持っているのでファイルの中身を読むべきです。
本例では、返された URL は Local:記述子を持ちます。これはカメラ内の XML ファイルを使用することを意味します。他には Http:(インターネット上にある XML を使用する)と File:(接続している PC 内にある XML を使用する)があります。記述子が Local:のとき URL は"Local:ファイル名.拡張子;アドレス;データ長"となっているので、次のようにファイル名、アドレス、データ長を取り出します:

```
char* file_name = strtok( ( url + strlen("Local:") + 1 ), ";" );
char* str_address = strtok( ( url + strlen("Local:") + 1 + strlen(file_name) + 1 ), ";" );
char* str_length = url + strlen("Local:") + 1 + strlen(file_name) + 1 + strlen(str_address) + 1;
uint64_t address = strtoul( str_address, NULL, 16 ):
size_t length = strtoul( str_length, NULL, 16);
```

そして、アドレスからデータ長分リードするとカメラ内の XML ファイルを読み込みます:

```
char* xml = NULL
xml = new char [length + 1];
gc = GCReadData( hDev, address, xml, &length );
```

ファイル名の拡張子"zip"が示すように XML ファイルは zip 形式で圧縮されているので zlib ライブラリを使用して XML ファイルを解凍します：

```
HZIP hZip = OpenZip( xml, (DWORD)length, 0 );
ZIPENTRY ze;

ZRESULT zr = GetZipItem( hZip, 0, &ze );
length = ze.unc_size;
char* uxml = new char[length + 1];
zr = UnzipItem( hZip, 0, uxml, ze.unc_size );
CloseZip( hZip );

uxml[length] = 0;
```

次に、GenAPI ノードマップオブジェクトのインスタンスを生成して XML をロードします：

```
CNodeMapRef node_map;
node_map._LoadXMLFromString( uxml );
```

そして、ポートオブジェクトのインスタンスを生成してノードマップを接続します。

```
MyPort port;
port.SetDeviceHandle( hDev );
node_map._Connect( &port, "CameraPort" );
```

こうすることで GenAPI がデバイスのレジスタにアクセスするために使われる準備ができます。

○ データストリームの生成

デバイスを開いてレジスタアクセスの準備を終えたら次に画像取得に必要なデータストリームを生成します。これは CreateDataStream()を通してなされます：

```
DS_HANDLE hDS;
gc = DevCreateDataStream( hDev, 0, &hDS );
```

現在は 1 つのデータストリームしかサポートしていないことに注意してください(第 2 引数は 0 を指定してください)。

○ 画像バッファの割り当て

画像取得は、入ってくる画像データで満たされるいくつかのバッファの割り当てを必要とします。各バッファはアナウンスされ使用準備のために待ち行列に入れられるべきです。

重要な点はバッファサイズの計算です。サイズは 3 つのパラメータに依存します：画像の横幅、高さ、そしてピクセルフォーマットです。これらは全て、上記 GenAPI ノードマップオブジェクトを使用することで読むことができるデバイスパラメータです：

```
CIntegerPtr pWidth = node_map._GetNode("Width");
CIntegerPtr pHeight = node_map._GetNode("Height");
CEnumerationPtr pPixelFormat = node_map._GetNode("PixelFormat");
```

これら 3 つのパラメータを取得したら、求める画像サイズは Width × Height × Factor で計算できます。Factor はピクセルフォーマットに依存します：RGB8Packed では Factor = 3、Mono8 では Factor = 1 というように。

バッファサイズの計算が終わるとバッファを割り当てて、DSAnnounceBuffer()を使用してグラバにそれをアナウンスします：

```
PVOID pBuffer;
BUFFER_HANDLE hBuffer;
pBuffer = new BYTE[buf_size];
gc = DSAnnounceBuffer( hDS, pBuffer, buf_size, NULL, &hBuffer );
```

次に、アナウンスされたバッファは DSQueueBuffer()を呼ぶことで必ず待ち行列に入れられなければいけません：

```
gc = DSQueueBuffer( hDS, hBuffer );
```

バッファの数はアプリケーションによって指定され、通常はユーザが設定可能です。

○ 取り込み開始

取り込み開始は次の 3 つのステップにかかわります：

1. グラバに取り込みを開始することを伝える
2. 画像到着イベントの登録
3. デバイスに映像転送の開始を伝える

1.は DSStartAcquisition()を呼ぶことでなされます：

```
gc = DSStartAcquisition( hDS );
```

画像取り込みを開始する前に、画像が取り込まれるたびに発火するイベントを登録しなければいけません。
2.は GCRegisterEvent()を通してなされます：

```
HANDLE hEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
EVENT_HANDLE hEvt;
gc = GCRegisterEvent( hDS, EVENT_NEW_BUFFER, hEvent, &hEvt );
```

後述のように Windows イベントオブジェクトは WaitForSingleObject によって実際に待機に使用されます。

3.はデバイスへのコマンドを発行することで達成されます：

```
CCommandPtr pStart = node_map._GetNode("AcquisitionStart");
pStart->Execute();
```

○ 画像の取り込み

カメラから送信される映像データを取得するために、イベントを登録したときに指定した Windows イベントオブジェクト上で待機します：

```
WaitForSingleObject( hEvent, 10000 );
```

イベントがシグナル状態になると、取得した画像を含むバッファのハンドルを検索します：

```
BUFFER_HANDLE hBuf;
cb = sizeof(hBuf);
gc = GCGetEventData( hEvt, &hBuf, &cb );
```

画像の使用後、例えばスクリーンに表示した後はグラバによる使用のために再度バッファを待ち行列に入れなければいけません：

```
gc = DSQueueBuffer( hDS, hBuf );
```

○ 取り込み停止

取り込み開始と同様に、取り込み停止も 3 ステップからなされます：

1. デバイスに映像転送の停止を伝える
2. 映像データ取得イベントの登録を抹消する
3. グラバに取り込みを停止することを伝える

1.はデバイスコマンドを実行することで達成されます：

```
CCommandPtr pStop = node_map._GetNode("AcquisitionStop");
pStop->Execute();
```

2.は GCUnregisterEvent ()を呼ぶことでなされます：

```
gc = GCUnregisterEvent( hEvt, EVENT_NEW_BUFFER );
```

3.は DSStopAcquisition()を呼ぶことでなされます：

```
gc = DSStopAcquisition( hDS );
```

○ 後処理

デバイスの取り扱いが終わったら、次のステップを実行します：

- データストリームを閉じる：

```
gc = DSClose( hDS );
```

- デバイスを閉じる：

```
gc = DevClose( hDev );
```

- ライブラリを閉じる：

```
gc = TLClose( hTL );
```

4. サンプルプログラムの説明

○ GigaCam_simple

製品添付の GigaCam.exe の簡易版です。機能の差は次のようになっています。

機能	GigCam_simple	GigaCam
カメラの認識	○	○
カメラのオープン	○	○
画像取り込み/表示	○	○
画面サイズ変更		○
画像保存		○
カメラコントロール	○(*1)	○
GenICam コントロール	○	○
波形モニタ		○
ヒストグラム		○
ベクトルスコープ		○
IP アドレス設定		○

(*1) 一部機能のみ

○ Sample

チュートリアルに従ったプログラムです。

画像を 100 枚連続で取り込みます(画像表示は行いません)。

○ Sample_VC6

画像を取り込んで表示させるプログラムです。

画面表示はモノクロ/カラーのどちらかを選択し、1 枚取り込みと連続取り込みが可能です。

○ Sample_VB6

画像を 1 枚取り込んで表示/保存するプログラムです。

モノクロカメラ専用です。

○ Sample_VCS

チュートリアルに従ったプログラムです。

画像を 1～100 枚連続で取り込み BMP 形式で保存します(画像表示は行いません)。

注意: 64bit OS でこれらのサンプルをビルドする時は「ソリューションプラットフォーム」が「Win32」もしくは「x86」でないと正常に動作しないことがあります。

５．変更来歴

バージョン	年月日	内容
1.00	2010.9.1	第 1 版公開
1.01	2011.10.26	サンプルソースに「VCS」を追加 チュートリアルの手順一部修正

日立国際電気

2011 年 10 月 26 日