Frequently Asked Questions

How to get PixelFormats available in a camera.

#4300-0308 Version 1.0.0.1

TOSHIBA TELI CORPORATION

Information contained in this document is subject to change without prior notice.

 www.toshiba-teli.co.jp

Document in Japanese

このドキュメントはカメラで使用できる PixelFormat の取得の仕方を紹介するドキュメントです。
日本語ドキュメントは英語ドキュメントの後にあります。

End of document in Japanese

1. How to get PixelFormats available in a camera.

This document shows how to get PixelFormats available in a camera.

There are two ways to get PixelFormats available in a camera. Note that there are no ways to get PixelFormats available in a camera during image acquisition is active, because changing PixelFormat is not allowed during image acquisition is active. Please get PixelFormats during image acquisition is not active.

A) Checks access mode of GenAPI EnumEntry nodes for PixelFormat node (or PixelCoding and PixelSize node).

EnumEntry nodes are nodes that contain information of Enumeration value used in parent Enumeration node. When access mode of an EnumEntry node is “Not Installed” of “Not Available”, the Enumeration value corresponding to the EnumEntry node is not available in the parent Enumeration node.

B) Checks ListOfElements registers of PixelCoding FeatureCSR and PixelSize FeatureCSR in IIDC2 register map.

ListOfElements registers are 16 bytes registers in Enumeration type FeatureCSR. (Feature Control and Status Register is a structure of registers defined in IIDC2.) Each bit of ListOfElements registers shows that a value corresponding to the bit is available in the feature or not. When bit value is 1, application can use corresponding value for the Enumeration type feature.

The later way is available in cameras that supports IIDC2 register map.

USB3 Vision cameras have PixelCoding and PixelSize registers for setting pixel format. Application should check access mode of EnumEntry nodes of PixelCoding node and PixelSize node, instead of EnumEntry nodes of PixelFormat node. Note that access mode of PixelSize EnumEntry node is influenced by PixelCoding node value.

The following code is an example for getting PixelFormats available in a camera.

GetAvailablePxlFmts_GenAPI()	:	Example function checking EnumEntry nodes.
GetAvailablePxlFmts_IIDC2()	:	Example function checking ListOfElements registers.
GetPxlFmtFromPxlCodingSize()	:	Example function getting PixelFormat ID corresponding to combination of PixelCoding and PixelSize value.

Document in Japanese

1. カメラで使用できる PixelFormats の取得の仕方

このドキュメントはカメラで使用できる PixelFormat の取得の仕方を紹介するドキュメントです。

PixelFormat の取得の仕方は２種類あります。撮像中は PixelFormat が変更できないため、カメラで使用できる PixelFormat の情報を取得することはできません。撮像停止中に PixelFormat の情報を取得してください。

A) GenAPI の PixelFormat ノード（又は PixelCoding ノードと PixelSize ノード）の EnumEntry ノードのアクセスモードを調査する方法。

EnumEntry ノードは親 Enumeration ノードが使用する Enumeration 値に関する情報を保有するノード群です。各 EnumEntry ノードは１個の Enumeration 値に関する情報を保有しています。EnumEntry ノードのアクセスモードが“Not Installed” または “Not Available”のときは、その EnumEntry ノードに対応した Enumeration 値は、少なくともその時点では親 Enumeration ノードの値として使用できません。

B) IIDC2 レジスタマップの PixelCoding FeatureCSR と PixelSize FeatureCSR 内の ListOfElements レジスタを調査する方法。

ListOfElements レジスタは Enumeration 型 FeatureCSR（IIDC2 で定義されている Feature Control and Status Register。カメラ機能の設定値を取得・制御するための各種情報を持つレジスタの集合体。）内の１６バイトのレジスタです。ListOfElements レジスタ内の各ビットは Enumeration 型の設定値にそれぞれ関連付けられており、bit 値が１の場合、対応する値が Enumeration 型 Feature の値として使用できます。

後者の方法は IIDC2 レジスタマップをサポートするカメラで使用できます。

USB3 Vision カメラは PixelCoding と PixelSize の２レジスタを使用して画像形式を設定することになっています。使用できる PixelFormat を調べるときは、PixelFormat ノードの EnumEntry ノードを調べるのではなく、PixelCoding ノードと PixelSize ノードの EnumEntry ノードを調べてください。PixelSize ノードの各 EnumEntry ノードのアクセスモードは PixelCoding ノードの設定値の影響をうけますので、PixelCoding ノードの値を設定してから PixelSize ノードを調査してください。

以下に、カメラで使用できる PixelFormat を取得する関数の例を記載します。

GetAvailablePxlFmts_GenAPI() ： EnumEntry 調査タイプ PixelFormat 取得関数例。

GetAvailablePxlFmts_IIDC2() ： ListOfElements 調査タイプ PixelFormat 取得関数例。

GetPxlFmtFromPxlCodingSize() ： PixelCoding 値と PixelSize 値の組み合わせに対応する PixelFormat 値を取得する関数の例。

End of document in Japanese

2. Example code

```
#include "TeliCamAPI.h"
#include "XmlFeatures.h"
#include "RegisterMap_BU.h"                  // PixelCoding, PixelSize
#include <vector>                            // vector
#include <algorithm>                         // find

using namespace Teli;

//Declaration of GetPxlFmtFromPxlCodingSize().
CAM_PIXEL_FORMAT GetPxlFmtFromPxlCodingSize(uint32_t uiPxlCoding, uint32_t uiPxlSize);


////////////////////////////////////////////////////////////////////////////////
// Example code.
// Gets pixel formats available in the camera using GenAPI.
//
// if NULL is specified to auiPxlFmt then number of available pixel format will be returned to piCount.
// if length of auiPxlFmt is not enough for writing available pixel format,
//   CAM_API_STS_BUFFER_TOO_SMALL will be returned as result status.
//
// GenAPI library is necessary for running this function.
//
//
// <<Usage>>
// CAM_HANDLE hCam;
// uint32_t uiCamIdx = 0;
// Cam_Open(uiCamIdx, &hCam);
// CAM_PIXEL_FORMAT auiPxlFmt[16];
// int iCount = 16
// CAM_API_STATUS uiSts = GetAvailablePxlFmts_GenAPI(hCam, auiPxlFmt, &iCount)
//
//
// [in]     hCam           Camera handle.
// [in/out] auiPxlFmt      PixelFormat data array.
// [in/out] piCount        In: Length of auiPxlFmt. Out: Number of available pixel format.
//
// return value            Result status.
//
////////////////////////////////////////////////////////////////////////////////
CAM_API_STATUS GetAvailablePxlFmts_GenAPI(CAM_HANDLE hCam, CAM_PIXEL_FORMAT *auiPxlFmt, int *piCount)
{
    if(0L == hCam)
    {
        return CAM_API_STS_INVALID_CAMERA_HANDLE;
    }
    if(NULL == piCount)
    {
        return CAM_API_STS_INVALID_PARAMETER;
    }

    std::vector<CAM_PIXEL_FORMAT> arrayFmt;
    CAM_API_STATUS          uiSts = CAM_API_STS_SUCCESS;
    CAM_TYPE                eCamType = CAM_TYPE_UNKNOWN;
    CAM_NODE_HANDLE         hNode = NULL;
    CAM_NODE_HANDLE         hEntry = NULL;
    TC_NODE_ACCESS_MODE     eAcc;
    uint32_t                uiNum = 0;
    int                     iCount = 0;

    // Clears PixelFormat buffer.
    arrayFmt.clear();

    // Gets camera interface type.
    uiSts = GetCamTypeFromCamHandle(hCam, &eCamType);

    if(CAM_API_STS_SUCCESS == uiSts)
    {
        if(CAM_TYPE_U3V == eCamType)
        {
            // USB3 Vision camera.
            // PixelFormat node is Read-Only node.
```



```
// Checks PixelCoding node and PixelSize node for getting available PixelFormats.

CAM_PIXEL_FORMAT uiPxlFmt = 0;

// GenApi type coding.
CAM_NODE_HANDLE    hNodeS    = NULL;
CAM_NODE_HANDLE    hEntryS   = NULL;
uint32_t uiEntryCount    = 0;
uint32_t uiEntryCountS   = 0;
int64_t tmp                 = 0L;
int64_t tmpS           = 0L;
int64_t llPixelCoding  = 0L;
int64_t llPixelSize         = 0L;

// Gets PixelCoding node.
uiSts = Nd_GetNode(hCam, XmlFeatures::XF_ID_PIXEL_CODING, &hNode);
if(CAM_API_STS_SUCCESS == uiSts)
{    // Backs up current PixelCoding setting.
     uiSts = Nd_GetEnumIntValue(hCam, hNode, &llPixelCoding);
}

if(CAM_API_STS_SUCCESS == uiSts)
{
     uiSts = Nd_GetNumOfEnumEntries(hCam, hNode, &uiEntryCount);
}


// Gets PixelSize node.
uiSts = Nd_GetNode(hCam, XmlFeatures::XF_ID_PIXEL_SIZE, &hNodeS);
if(CAM_API_STS_SUCCESS == uiSts)
{    // Backs up current PixelSize setting.
     uiSts = Nd_GetEnumIntValue(hCam, hNodeS, &llPixelSize);
}

if(CAM_API_STS_SUCCESS == uiSts)
{
     // Checks each PixelCoding EnumEntries.
     for(uint32_t i = 0; i < uiEntryCount; ++i)
     {
          // Gets a PixelCoding entry for checking access mode of it.
          uiSts = Nd_GetEnumEntryByIndex(hCam, hNode, i, &hEntry);

          // Checks access mode of target PixelCoding EnumEntry node.
          eAcc = TC_NODE_ACCESS_MODE_NI;
          if(CAM_API_STS_SUCCESS == uiSts)
          {
               uiSts = Nd_GetAccessMode(hCam, hEntry, &eAcc);
          }

          // Gets available PixelSizes if the PixelCoding EnumEntry value is available.
          if((CAM_API_STS_SUCCESS == uiSts) && (TC_NODE_ACCESS_MODE_NI != eAcc) &&
             (TC_NODE_ACCESS_MODE_NA != eAcc))
          {
               // Gets integer value of thet PixelCoding entry.
               uiSts = Nd_GetEnumEntryIntValue(hCam, hEntry, &tmp);

               // Sets target PixelCoding value temporary.
               uiSts = Nd_SetEnumIntValue(hCam, hNode, tmp);

               // Gets PixelSize node.
               uiSts = Nd_GetNode(hCam, XmlFeatures::XF_ID_PIXEL_SIZE, &hNodeS);

               // Gets entry count of PixelSize node.
               uiSts = Nd_GetNumOfEnumEntries(hCam, hNodeS, &uiEntryCountS);

               // Checks each PixelSize EnumEntries.
               for(int j = uiEntryCountS - 1; j >= 0 ; --j)
               {
                    // Gets a PixelSize entry to check its access mode.
                    uiSts = Nd_GetEnumEntryByIndex(hCam, hNodeS, j, &hEntryS);

                    // Checks access mode of the PixelSize EnumEntry node.
                    eAcc = TC_NODE_ACCESS_MODE_NI;
                    if(CAM_API_STS_SUCCESS == uiSts)
                    {
                         uiSts = Nd_GetAccessMode(hCam, hEntryS, &eAcc);
```



```
                        }

                        // If the PixelSize EnumEntry value is available,
                        // PixelFormat corresponding to the the PixelCoding EnumEntry value and
                        // the PixelSize EnumEntry value is available.
                        if((CAM_API_STS_SUCCESS == uiSts) && (TC_NODE_ACCESS_MODE_NI != eAcc) &&
                            (TC_NODE_ACCESS_MODE_NA != eAcc))
                        {
                            if(CAM_API_STS_SUCCESS == uiSts)
                            {   // Gets integer value of the EnumEntry node.
                                uiSts = Nd_GetEnumEntryIntValue(hCam, hEntryS, &tmpS);
                            }
                            if(CAM_API_STS_SUCCESS == uiSts)
                            {
                                // Stores PixelFormat converted from PixelCoding and PixelSize
                                // to PixelFormat temporary buffer.
                                uiPxlFmt = GetPxlFmtFromPxlCodingSize((uint32_t)tmp,
                                                                      (uint32_t)tmpS);
                                if(PXL_FMT_Unknown != uiPxlFmt)
                                {
                                    arrayFmt.push_back(uiPxlFmt);
                                }
                            }
                        }
                    }
                }

                // Resumes original PixelCoding value.
                Nd_SetEnumIntValue(hCam, hNode, llPixelCoding);
                // Resumes original PixelSize value.
                Nd_SetEnumIntValue(hCam, hNodeS, llPixelSize);
            }
        }
        else
        {
            // GigE Vision cameras.
            // Checks access mode of PixelFormat EnumEntry nodes to get available PixelFormats.

            int64_t llPixelFormat  = 0L;

            // Gets PixelFormat node.
            uiSts = Nd_GetNode(hCam, XmlFeatures::XF_ID_PIXEL_FORMAT, &hNode);
            if(CAM_API_STS_SUCCESS == uiSts)
            {   // Gets number of EnumEntry nodes.
                uiSts = Nd_GetNumOfEnumEntries(hCam, hNode, &uiNum);
            }

            if(CAM_API_STS_SUCCESS == uiSts)
            {
                // Checks each EnumEntry nodes.
                for(uint32_t i = 0; i < uiNum; ++i)
                {
                    // Gets a EnumEntry to check its access mode.
                    uiSts = Nd_GetEnumEntryByIndex(hCam, hNode, i,  &hEntry);
                    if(CAM_API_STS_SUCCESS != uiSts)
                    {
                        break;
                    }

                    // Gets access mode of the EnumEntry node.
                    uiSts = Nd_GetAccessMode(hCam, hEntry, &eAcc);
                    if(CAM_API_STS_SUCCESS != uiSts)
                    {
                        break;
                    }

                    // If the EnumEntry node is availlable, the correspoinding PixelFormat is available.
                    if((TC_NODE_ACCESS_MODE_RO == eAcc) || (TC_NODE_ACCESS_MODE_RW == eAcc))
                    {
                        // Gets integer value of the EnumEntry node.
                        uiSts = Nd_GetEnumEntryIntValue(hCam, hEntry, &llPixelFormat);
                        if(CAM_API_STS_SUCCESS == uiSts)
                        {   // Registers the integer value to the available PixelFormat array.
                            arrayFmt.push_back(static_cast<CAM_PIXEL_FORMAT>(llPixelFormat));
                        }
```



```
                            else
                            {
                                break;
                            }
                        }
                    }
                }
            }
        }

        // Sets available PixelFormat data to the arguments.
        if(CAM_API_STS_SUCCESS == uiSts)
        {
            if(NULL == auiPxlFmt)
            {   // Returns number of available PixelFormat.
                *piCount = static_cast<int>(arrayFmt.size());
            }
            else
            {
                if(*piCount < static_cast<int>(arrayFmt.size()))
                {
                    uiSts = CAM_API_STS_BUFFER_TOO_SMALL;
                }
                *piCount = static_cast<int>(arrayFmt.size());

                if(CAM_API_STS_SUCCESS == uiSts)
                {
                    try
                    {
                        for(int i = static_cast<int>(arrayFmt.size()) - 1; i >= 0; --i)
                        {
                            auiPxlFmt[i] = arrayFmt.at(i);
                        }
                    }
                    catch(...)
                    {
                        uiSts = CAM_API_STS_UNSUCCESSFUL;
                    }
                }
            }
        }
        return uiSts;
}


////////////////////////////////////////////////////////////////////////////////
// Example code.
// Gets pixel formats available in the camera using IIDC2 register map for BU/DU series.
//
// if NULL is specified to auiPxlFmt then number of available pixel format will be returned to piCount.
// if length of auiPxlFmt is not enough for writing abvailable pixel format,
//   CAM_API_STS_BUFFER_TOO_SMALL will be returned as result status.
//
//
// <<Usage>>
// CAM_HANDLE hCam;
// uint32_t uiCamIdx = 0;
// Cam_Open(uiCamIdx, &hCam);
// CAM_PIXEL_FORMAT auiPxlFmt[16];
// int iCount = 16
// CAM_API_STATUS uiSts = GetAvailablePxlFmts_IIDC2(hCam, auiPxlFmt, &iCount)
//
//
// [in]     hCam           Camera handle.
// [in/out] auiPxlFmt      PixelFormat data array,
// [in/out] piCount        In: Length of auiPxlFmt. Out: Number of available pixel format.
//
// return value            Result status.
//
////////////////////////////////////////////////////////////////////////////////
CAM_API_STATUS GetAvailablePxlFmts_IIDC2(CAM_HANDLE hCam, CAM_PIXEL_FORMAT *auiPxlFmt, int *piCount)
{
    if(0L == hCam)
    {
        return CAM_API_STS_INVALID_CAMERA_HANDLE;
```



```
    }
    if(NULL == piCount)
    {
        return CAM_API_STS_INVALID_PARAMETER;
    }

    std::vector<CAM_PIXEL_FORMAT> arrayFmt;
    CAM_API_STATUS          uiSts = CAM_API_STS_SUCCESS;
    CAM_TYPE            eCamType = CAM_TYPE_UNKNOWN;
    CAM_NODE_HANDLE         hNode = NULL;
    CAM_NODE_HANDLE         hEntry = NULL;
    uint32_t            uiNum = 0;
    int                     iCount = 0;

    // Clears PixelFormat buffer.
    arrayFmt.clear();

    // Gets camera interface type.
    uiSts = GetCamTypeFromCamHandle(hCam, &eCamType);

    if(CAM_API_STS_SUCCESS == uiSts)
    {
        if(CAM_TYPE_U3V == eCamType)
        {
            CAM_PIXEL_FORMAT uiPxlFmt = 0;

            RegMapBU::REGVAL_PIXEL_CODING   ePCodingCurent;
            RegMapBU::REGVAL_PIXEL_SIZE     ePSizeCurent;
            uint32_t LoePCoding[4];
            uint32_t LoePSize[4];
            uint32_t tmpPCoding;
            uint32_t tmpPSize;
            uint32_t tmpC;
            uint32_t tmpS;

            // Gets current PixelCoding and PixelSize value.
            uiSts = Cam_ReadReg(hCam, RegMapBU::ADR_IMGFMT_PIXEL_CODING_I, 1, &ePCodingCurent);
            if(CAM_API_STS_SUCCESS == uiSts)
            {
                uiSts = Cam_ReadReg(hCam, RegMapBU::ADR_IMGFMT_PIXEL_SIZE_I, 1, &ePSizeCurent);
            }

            // Gets List of element register of PixelCoding featureCSR.
            if(CAM_API_STS_SUCCESS == uiSts)
            {
                uiSts = Cam_ReadReg(hCam, RegMapBU::ADR_IMGFMT_PIXEL_CODING_ENUM32B +
                                    RegMapBU::OFS_FCSR_ENUM_LIST_OF_ELEMENT_B16, 4, &LoePCoding[0]);
            }

            if(CAM_API_STS_SUCCESS == uiSts)
            {
                // Searches available PixelCoding.
                tmpPCoding = 0;
                for(int i = 0; i < 4; ++i)
                {
                    tmpC = 1;
                    for(int bit = 0 ; bit < 32; ++bit)
                    {
                        if((tmpC & LoePCoding[i]) == tmpC)
                        {   // Found available PixelCoding value.
                            // Sets PixelCoding value temporary.
                            uiSts = Cam_WriteReg(hCam, RegMapBU::ADR_IMGFMT_PIXEL_CODING_I, 1,
                                                 &tmpPCoding);
                            if(CAM_API_STS_SUCCESS != uiSts)
                            {
                                break;
                            }

                            // Gets available PixelSize.
                            uiSts = Cam_ReadReg(hCam, RegMapBU::ADR_IMGFMT_PIXEL_SIZE_ENUM32B +
                                                RegMapBU::OFS_FCSR_ENUM_LIST_OF_ELEMENT_B16, 4,
                                                &LoePSize[0]);
                            if(CAM_API_STS_SUCCESS != uiSts)
                            {
                                break;
                            }
```



```
                            // Searches available PixelSize.
                            tmpPSize = 0;
                            for(int j = 0; j < 4; ++j)
                            {
                                tmpS = 1;
                                for(int bit = 0 ; bit < 32; ++bit)
                                {
                                    if((tmpS & LoePSize[j]) == tmpS)
                                    {   // Found available PixelSize value.
                                        // Stores PixelFormat converted from PixelCoding and PixelSize
                                        // to PixelFormat temporary buffer.
                                        uiPxlFmt = GetPxlFmtFromPxlCodingSize(tmpPCoding, tmpPSize);
                                        if(PXL_FMT_Unknown != uiPxlFmt)
                                        {
                                            // Checks if the PixelFormat is already registered or not.
                                            if(arrayFmt.size() == 0)
                                            {
                                                arrayFmt.push_back(uiPxlFmt);
                                            }
                                            else
                                            {
                                                std::vector<CAM_PIXEL_FORMAT>::iterator cIter =
                                                                    std::find(arrayFmt.begin(),
                                                                              arrayFmt.end(),
                                                                              uiPxlFmt);
                                                if(cIter == arrayFmt.end())
                                                {
                                                    arrayFmt.push_back(uiPxlFmt);
                                                }
                                            }
                                        }
                                    }
                                    // Prepares for the next check.
                                    tmpS += tmpS;
                                    ++tmpPSize;
                                }
                            }
                        }
                        // Prepares for the next check.
                        tmpC += tmpC;
                        ++tmpPCoding;
                    }

                    if(CAM_API_STS_SUCCESS != uiSts)
                    {
                        break;
                    }
                }

                // Resumes original PixelCoding value.
                Cam_WriteReg(hCam, RegMapBU::ADR_IMGFMT_PIXEL_CODING_I, 1, &ePCodingCurent);
                // Resumes original PixelSize value.
                Cam_WriteReg(hCam, RegMapBU::ADR_IMGFMT_PIXEL_SIZE_I, 1, &ePSizeCurent);
            }
        }
    }

    if(CAM_API_STS_SUCCESS == uiSts)
    {
        if(NULL == auiPxlFmt)
        {   // Returns number of available PixelFormat.
            *piCount = static_cast<int>(arrayFmt.size());
        }
        else
        {
            if(*piCount < static_cast<int>(arrayFmt.size()))
            {
                uiSts = CAM_API_STS_BUFFER_TOO_SMALL;
            }
            *piCount = static_cast<int>(arrayFmt.size());

            if(CAM_API_STS_SUCCESS == uiSts)
            {
                try
                {
```



```
                        for(int i = static_cast<int>(arrayFmt.size()) - 1; i >= 0; --i)
                        {
                            auiPxlFmt[i] = arrayFmt.at(i);
                        }
                    }
                    catch(...)
                    {
                        uiSts = CAM_API_STS_UNSUCCESSFUL;
                    }
                }
            }
        }
        return uiSts;
}


////////////////////////////////////////////////////////////////////////////
//  Gets pixel formats corresponding to the argument PixelCoding and PixelSize value.
//
//
// [in] uiPixelCoding      PixelCoding.
// [in] uiPixelSize            PixelSize.
//
// return value                PixelFormat.
//
////////////////////////////////////////////////////////////////////////////
CAM_PIXEL_FORMAT GetPxlFmtFromPxlCodingSize(uint32_t uiPxlCoding, uint32_t uiPxlSize)
{
    CAM_PIXEL_FORMAT uiVal = PXL_FMT_Unknown;

    switch(uiPxlCoding)
    {
    case RegMapBU::RV_IMGFMT_MONO:                // Mono (0)
        switch(uiPxlSize)
        {
        case 8:
            uiVal = PXL_FMT_Mono8;
            break;

        case 10:
            uiVal = PXL_FMT_Mono10;
            break;

        case 12:
            uiVal = PXL_FMT_Mono12;
            break;

        case 16:
            uiVal = PXL_FMT_Mono16;
            break;
        }
        break;
    //case RegMapBU::RV_IMGFMT_RGBPACKED:             // RGBPacked(34)
    case RegMapBU::RV_IMGFMT_RGBPACKED:           // RGBPacked(34)
        switch(uiPxlSize)
        {
        case 24:
            uiVal = PXL_FMT_RGB8;
            break;

        case 30:
//          uiVal = PXL_FMT_RGB10;
            uiVal = 0x02300018;
            break;

        case 36:
//          uiVal = PXL_FMT_RGB12;
            uiVal = 0x0230001A;
            break;
        }
        break;


    case 42:                                      // BGRPacked(42)
        switch(uiPxlSize)
```



```
        {
        case 24:
             uiVal = PXL_FMT_BGR8;
             break;

        case 30:
             uiVal = PXL_FMT_BGR10;
             break;

        case 36:
             uiVal = PXL_FMT_BGR12;
             break;
        }
        break;

    case RegMapBU::RV_IMGFMT_YUV411PACKED:        // YUV411Packed(66)
    case 64:                                      // YUV411_8 in BU series.
        switch(uiPxlSize)
        {
        case 12:
             uiVal = PXL_FMT_YUV411_8;
             break;
        }
        break;

    case RegMapBU::RV_IMGFMT_YUV422PACKED:        // YUV422Packed(74)
    case 72:                                      // YUV422_8 in BU series.
        switch(uiPxlSize)
        {
        case 16:
             uiVal = PXL_FMT_YUV422_8;
             break;
        }
        break;

    case RegMapBU::RV_IMGFMT_YUV444PACKED:        // YUV444Packed(82)
        switch(uiPxlSize)
        {
        case 24:
             uiVal = PXL_FMT_YUV8;
                  break;
        }
        break;

    case RegMapBU::RV_IMGFMT_BAYER_GR:            // BayerGR(96)
        switch(uiPxlSize)
        {
        case 8:
             uiVal = PXL_FMT_BayerGR8;
             break;

        case 10:
             uiVal = PXL_FMT_BayerGR10;
             break;

        case 12:
             uiVal = PXL_FMT_BayerGR12;
             break;
        }
        break;

    case RegMapBU::RV_IMGFMT_BAYER_RG:            // BayerRG(99)
        switch(uiPxlSize)
        {
        case 8:
             uiVal = PXL_FMT_BayerRG8;
             break;

        case 10:
             uiVal = PXL_FMT_BayerRG10;
             break;

        case 12:
             uiVal = PXL_FMT_BayerRG12;
             break;
        }
```



```
        break;

    case RegMapBU::RV_IMGFMT_BAYER_GB:          // BayerGB(102)
        switch(uiPxlSize)
        {
        case 8:
            uiVal = PXL_FMT_BayerGB8;
            break;

        case 10:
            uiVal = PXL_FMT_BayerGB10;
            break;

        case 12:
            uiVal = PXL_FMT_BayerGB12;
            break;
        }
        break;

    case RegMapBU::RV_IMGFMT_BAYER_BG:          // BayerBG(105)
        switch(uiPxlSize)
        {
        case 8:
            uiVal = PXL_FMT_BayerBG8;
            break;

        case 10:
            uiVal = PXL_FMT_BayerBG10;
            break;

        case 12:
            uiVal = PXL_FMT_BayerBG12;
            break;
        }
        break;
    }

    return uiVal;
}
```


3. Others

3.1. Revision History

Date	Version	Description
2016/05/24	1.0.0	Created the initial version

3.2. Disclaimer

The disclaimer of this document including example code is described in “License Agreement TeliCamSDK Eng.pdf” in TeliCamSDK installation folder.
Make sure to read this Agreement carefully before using it.
Refer to TeliCamSDK installation folder/Documents/License folder

