Teledyne DALSA • 880 Rue McCaffrey • St-Laurent, Québec, H4T 2C7 • Canada
https://www.teledynedalsa.com/imaging/products/software/

SAP-AN0008: Using Frame Grabbers & Sapera LT with Matrox Imaging Library (MIL) Processing API

Transferring Images from Sapera LT Buffers to Matrox MIL Processing Buffers

Overview

This application note describes how to use a Teledyne Dalsa frame grabber and Sapera LT to acquire images and perform image processing using the Matrox Imaging Library (MIL) API. This allows seamless migration of existing MIL applications to a Teledyne DALSA frame grabber (or other acquisition devices, such as Genie Nano cameras).

It demonstrates how to pass a buffer created with Sapera LT to MIL, or from MIL to Sapera LT. The application note refers to the C++ implementation, however the concepts are applicable to .NET as well.

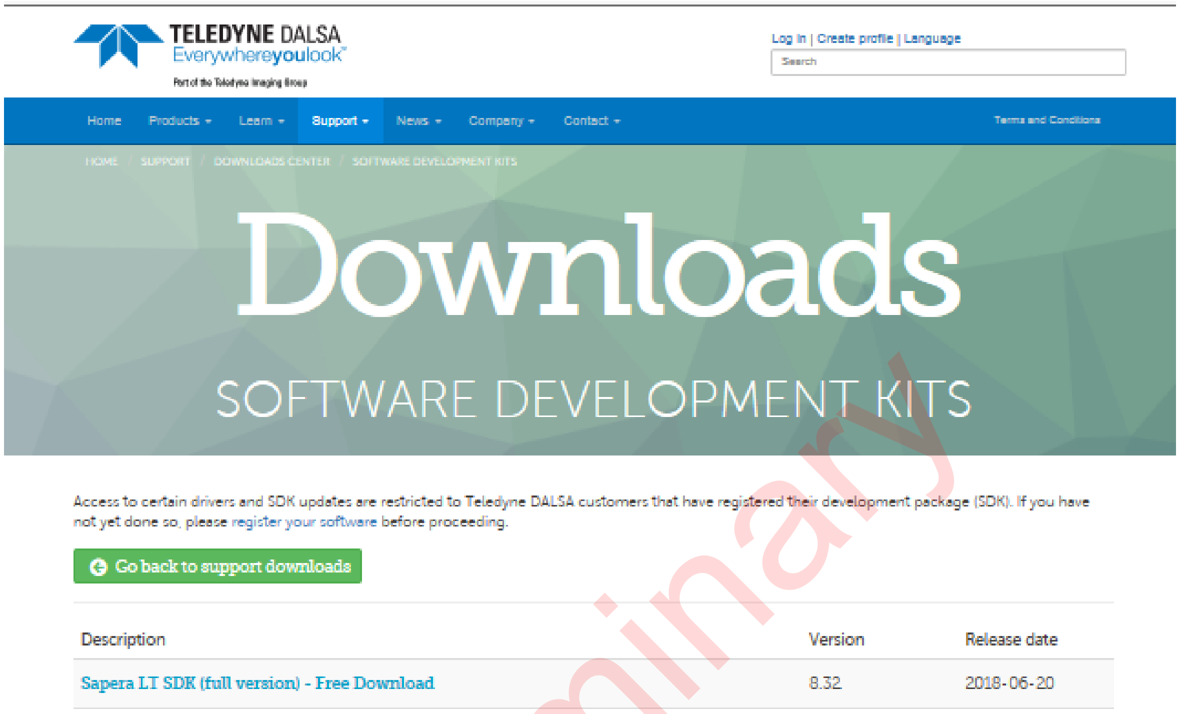
For more information on processing images within callbacks or using multiple threads, contact Teledyne DALSA.

Acquisition and Control Libraries

Sapera LT is a **free** image acquisition and control software development toolkit (SDK) for Teledyne DALSA'S cameras and frame grabbers. Hardware independent in nature, Sapera LT offers a rich development ecosystem for machine vision OEMs and system integrators. Sapera LT supports image acquisition from cameras and frame grabbers based on standards including GigE Vision™, CameraLink®, CameraLink HS™, and CoaXpress®.

Sapera LT SDK (full version), the image acquisition and control SDK for Teledyne DALSA cameras and frame grabbers is available for download from the Teledyne DALSA website:

http://teledynedalsa.com/imaging/support/downloads/sdks/


TELEDYNE DALSA
Everywhereyoulook
Log in | Create profile | Language
Search
Home
Products
Learn
Support
News
Company
Contact
Terms and Conditions
HOME / SUPPORT / DOWNLOADS CENTER / SOFTWARE DEVELOPMENT KITS
Downloads
SOFTWARE DEVELOPMENT KITS
Go back to support downloads

Access to certain drivers and SDK updates are restricted to Teledyne DALSA customers that have registered their development package (SDK). If you have not yet done so, please register your software before proceeding.

Description	Version	Release date
Sapera LT SDK (full version) - Free Download	8.32	2018-06-20

Note: Administrative privileges are required to perform the Sapera LT software installation described in this application note.

Upgrading MIL Applications to Use Teledyne DALSA Acquisition Devices

Teledyne DALSA image acquisition devices supported by Sapera LT include both frame grabbers and cameras that support Camera Link, CoaXPress (CXP) and Camera Link HS (CLHS) standards.

To take advantage of Trigger-To-Image-Reliability (T2IR) features with Teledyne DALSA products, Sapera LT is required.

To use TurboDrive with supported Teledyne DALSA GigE Vision cameras Sapera LT is required.

Information on the full line of available products is available on the Teledyne DALSA website:

https://www.teledynedalsa.com/en/products/imaging/frame-grabbers/

For example, the Xtium family is recommended when migrating applications to Teledyne DALSA frame grabbers.

Standard	**Recommend Teledyne DALSA Frame Grabber**
Camera Link	Xtium-CL MX4
CoaXPress (CXP)	Xtium-CXP PX8
Camera Link HS	Xtium2-CLHS

Teledyne DALSA Installation Prerequisites

When installing a Teledyne DALSA frame grabber in your system, the following software is required:

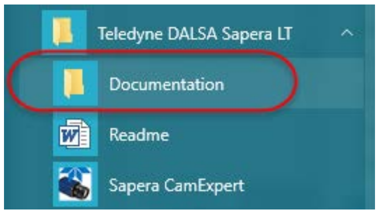
- Sapera LT SDK (full version 8.32 or higher)
- For applications that use Teledyne DALSA frame grabbers, the device driver must be installed.

Sapera LT SDK and all device drivers are available for free download from the Teledyne DALSA website:

https://www.teledynedalsa.com/en/support/downloads-center/software-development-kits/

https://www.teledynedalsa.com/en/support/downloads-center/device-drivers/

Key documentation provided with the installation of the Sapera LT SDK includes the **Getting Started Manual For Frame Grabbers** or **Getting Start Manual for GigE Vision Cameras**.



ⓘ	To compile sample application code verify that all necessary library and header files are included. Error checking and acquisition format validation should also be performed.

Creating a MIL Buffer from a Sapera LT Buffer

MIL buffers can be created using the memory already allocated by Sapera LT buffers. This allows MIL buffers to access the same memory location without copying.

To pass a Sapera LT buffer to MIL the address of the Sapera LT buffer is required; the MIL buffer is then constructed using this Sapera LT buffer address.

In Sapera LT, acquisition buffers are created using the SapBuffer class. The SapBuffer class includes the SapBuffer::GetAddress() function to return the virtual (logical) memory address assigned to the buffer.

SapBuffer::GetAddress() can only be called after the SapBuffer object has been constructed and created. For example, to get the address of a SapBuffer *pBuffer*:

```
// Allocate Sapera buffer object taking settings directly from acquisition object
SapBuffer *pBuffer = new SapBuffer(1, pAcq);

// Create resources for Sapera buffer object
success = pBuffer->Create();

// Get the Sapera buffer object
void *pSaperaBufferAddress;
pBuffer->GetAddress(&pSaperaBufferAddress);
```

The SapBuffer address is passed to the MIL **MbufCreate2d** function which creates a MIL buffer without copying the data. The SapBuffer pitch and height is also required (the pitch is used since the actual buffer size may differ from the image width).

```
// Create the required MIL identifiers
MIL_ID   MilApplication,      /* Application identifier.  */
         MilSystem,           /* System identifier.       */
         MilDisplay,          /* Display identifier.      */
         MilImage;            /* Image buffer identifier. */

// Allocate the MIL resources
MappAllocDefault(M_DEFAULT, &MilApplication, &MilSystem, &MilDisplay, M_NULL,
M_NULL);
// Allocate a MIL img buffer with the same size and format as the Sapera buffer
int bufferPitch = pBuffer->GetPitch(); //Sapera LT SapBuffer::GetPitch function
int bufferHeight = pBuffer->GetHeight();//Sapera LT SapBuffer::GetHeight function
// MIL MbufCreate2d function
MbufCreate2d(MilSystem, bufferPitch, bufferHeight, 8+M_UNSIGNED,
M_GRAB+M_IMAGE+M_PROC+M_DISP, M_HOST_ADDRESS+M_PITCH,M_DEFAULT, pSaperaBuffer,
&MilImage);
```

The MIL buffer can now be processed using the MIL API.

The MIL **MbufPut** group of functions can also be used to copy the Sapera LT buffer if necessary.

Example Code with Processing

The following example code demonstrates how to create SapBuffers to acquire images into. Two MIL buffers are created that access the buffer memory locations allocated by Sapera LT and a simple MIL processing function is applied to one MIL buffer.

```
// SaperaLT_MIL.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include "SapClassBasic.h"
#include <iostream>
#include <mil.h>

using namespace std;

// Transfer callback function is called each time a complete frame is transferred
void saperaXferCallback(SapXferCallbackInfo *pInfo)
{
   // Display the last transferred frame
   SapView *pView = (SapView *)pInfo->GetContext();
   pView->Show();
}
// Example program
//
int main()
{
   //BOOL success;

   // Allocate acquisition object (in this case, a frame grabber)
   SapAcquisition saperaAcq(SapLocation("Xtium-CL_MX4_1", 0));

   // To allocate a camera a SapAcqDevice object is used
   //SapAcqDevice saperaAcq("Nano-M2590_1", FALSE); // uses camera default settings
   //SapAcqDevice saperaAcq("Genie_M640", "MyCamera.ccf");  // loads configuration file

   // Allocate buffer objects, taking settings directly from the acquisition
   SapBuffer saperaBuffer(2, &saperaAcq);

   // Allocate view object to display in an internally created window
   SapView saperaView(&saperaBuffer, (HWND)-1);

   // Allocate transfer object to link acquisition and buffer
   SapAcqToBuf saperaTransfer(&saperaAcq, &saperaBuffer, saperaXferCallback, &saperaView);

   // Create the required MIL identifiers
   MIL_ID   MilApplication, /* Application identifier.  */
   MilSystem,               /* System identifier.       */
   MilDisplay,              /* Display identifier.      */
   MilImage,                /* Image buffer identifier. */
   MilImageToProcess;       /* Image buffer identifier. */

                            // Allocate the MIL resources
   MappAllocDefault(M_DEFAULT, &MilApplication, &MilSystem, &MilDisplay, M_NULL, M_NULL);

   // Create resources for all objects
   saperaAcq.Create();
   saperaBuffer.Create();
   saperaView.Create();
   saperaTransfer.Create();

   // Start a continuous transfer (live grab)
   saperaTransfer.Grab();
   cout << "Press any key to stop grab" << endl;
   cin.get();

   // pause to see the unprocessed image displayed
   // Stop the transfer and wait (timeout = 5 seconds)
   saperaTransfer.Freeze();
   saperaTransfer.Wait(5000);
```

```
    // MIL processing

    // Create an array to hold the buffer addresses returned using SapBuffer::GetAddress()

    //vector of void pointers :address of list containing buffer addresses
    std::vector<void*> virtualAddress_Buffer(saperaBuffer.GetCount());
    for(int bufferIndex= 0; bufferIndex < saperaBuffer.GetCount(); bufferIndex++)
        saperaBuffer.GetAddress(bufferIndex, &virtualAddress_Buffer[bufferIndex]);

    // Get buffer size using the Sapera LT SapBuffer::GetPitch and SapBuffer::GetHeight functions
    int bufferPitch = saperaBuffer.GetPitch();
    int bufferWidth = saperaBuffer.GetWidth();
    int bufferHeight = saperaBuffer.GetHeight();

    // Allocate MIL image buffers (here 8-bit monochrome; user applications should validate format).
    MbufCreate2d(MilSystem, bufferWidth , bufferHeight, 8 + M_UNSIGNED, M_GRAB + M_IMAGE + M_PROC +
    M_DISP, M_HOST_ADDRESS, bufferPitch, virtualAddress_Buffer[0], &MilImage);
    MbufCreate2d(MilSystem, bufferWidth, bufferHeight, 8 + M_UNSIGNED, M_GRAB + M_IMAGE + M_PROC +
    M_DISP, M_HOST_ADDRESS, bufferPitch, virtualAddress_Buffer[1], &MilImageToProcess);

    cout << "Show unprocessed image in MIL display - press any key to continue " << endl;
    cin.get(); // wait until a key has been hit

    // Display an unprocessed image buffer using MIL display function.
    MdispSelect(MilDisplay, MilImage);
    cout << "Show processed image (MimFlip) in MIL display - press any key to continue " << endl;
    cin.get(); // wait until a key has been hit

    //MIL processing function MimFlip
    MimFlip(MilImageToProcess, MilImageToProcess, M_FLIP_HORIZONTAL, M_DEFAULT);

    // Display the processed image buffer using MIL display function.
    MdispSelect(MilDisplay, MilImageToProcess);
    cout << "Press any key to terminate" << endl;
    cin.get(); // wait until a key has been hit

    // Release resources for all objects
    saperaTransfer.Destroy();
    saperaView.Destroy();
    saperaBuffer.Destroy();
    saperaAcq.Destroy();

    /* Free all allocations. */
    MbufFree(MilImage);
    MbufFree(MilImageToProcess);
    MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);

    return 0;
}
```

Using Sapera LT to Acquire Images into an Existing MIL Buffer

To pass a MIL buffer to Sapera LT to use to acquire images into, the address of the MIL buffer is required; the Sapera LT buffer is then constructed using this MIL buffer address.

The MIL buf class includes the MbufInquire() function which can return the virtual address of the MIL buffer. For example, to create a MIL buffer and get its address:

```
// Create the required MIL identifiers
MIL_ID   MilApplication,       /* Application identifier.  */
         MilSystem,            /* System identifier.       */
         MilDisplay,           /* Display identifier.      */
         MilImage;             /* Image buffer identifier. */

// Allocate the MIL resources
MappAllocDefault(M_DEFAULT, &MilApplication, &MilSystem, &MilDisplay, M_NULL,
M_NULL);

// Allocate a MIL img buffer.
// This sample code uses a 1280x1024 unsigned 8-bit monochrome buffer
MbufAlloc2d(MilSystem, 1280, 1024, 8+M_UNSIGNED,
M_GRAB+M_IMAGE+M_PROC+M_DISP, &MilImage);


//Get MIL buffer address
void *pMILBuffer= NULL;
MbufInquire(MilImage, M_HOST_ADDRESS, &pMILBuffer);
```

It is also necessary to get the pitch of the buffer, as the MIL buffer does not correspond directly to the width of the image.

```
//Get MIL buffer pitch
MIL_INT bufferPitch;
MbufInquire(MilImage, M_PITCH, &bufferPitch);
```

The MIL buffer address is passed to the Sapera LT SapBuffer constructor which creates a Sapera buffer object without copying the data. The buffer pitch size is passed as the width of the buffer:

```
SapBuffer *pBuffer = new SapBuffer(1, &pMILBuffer, bufferPitch, bufferHeight,
SapFormatMono8, SapBuffer::TypeScatterGather);
```

Images can now be acquired by Teledyne DALSA devices directly into the memory location used by MIL to process the buffer.

Example Code with Processing

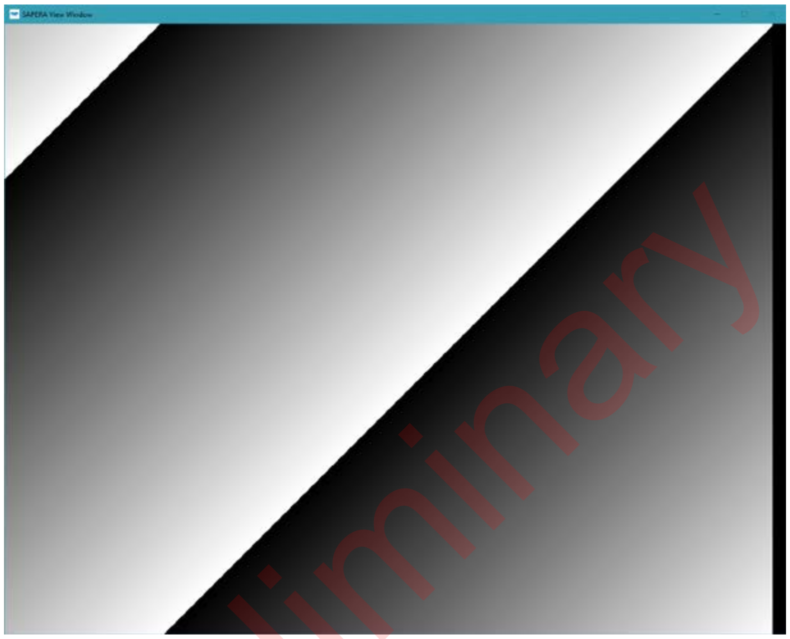
The following example code demonstrates how to create SapBuffers to grab into memory already allocated by MIL buffers. 2 buffers are created and a simple MIL processing function is applied to one MIL buffer.

```
// SaperaLT_MIL.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include "SapClassBasic.h"
#include <iostream>
#include <mil.h>

using namespace std;

// Transfer callback function is called each time a complete frame is transferred
void saperaXferCallback(SapXferCallbackInfo *pInfo)
{
   // Display the last transferred frame
   SapView *pView = (SapView *)pInfo->GetContext();
   pView->Show();
}
// Example program
//
int main()
{
   // Create the required MIL identifiers
   MIL_ID   MilApplication,       /* Application identifier.  */
      MilSystem,            /* System identifier.       */
      MilDisplay,           /* Display identifier.      */
      MilImage,             /* Image buffer identifier. */
      MilImageToProcess;    /* Image buffer identifier. */

                            // Allocate the MIL resources
   MappAllocDefault(M_DEFAULT, &MilApplication, &MilSystem, &MilDisplay, M_NULL, M_NULL);

   // Create variables for the buffer width and height (here a 1280x1024 image is used)
   int bufferWidth = 1280;
   int bufferHeight = 1024;

   // Allocate MIL image buffers.
   MbufAlloc2d(MilSystem, bufferWidth, bufferHeight, 8 + M_UNSIGNED, M_GRAB + M_IMAGE + M_PROC +
   M_DISP, &MilImage);
   MbufAlloc2d(MilSystem, bufferWidth, bufferHeight, 8 + M_UNSIGNED, M_GRAB + M_IMAGE + M_PROC +
   M_DISP, &MilImageToProcess);

   //Get MIL buffer pitch
   MIL_INT bufferPitch;
   MbufInquire(MilImage, M_PITCH, &bufferPitch);

   // Create an array to hold the buffer addresses
   void* virtualAddress_Buffer[2];
   virtualAddress_Buffer[0] = (void *)MbufInquire(MilImage, M_HOST_ADDRESS, M_NULL);
   virtualAddress_Buffer[1] = (void *)MbufInquire(MilImageToProcess, M_HOST_ADDRESS, M_NULL);

   // Allocate Sapera acquisition object (in this case, a frame grabber)
   SapAcquisition saperaAcq(SapLocation("Xtium-CL_MX4_1", 0));

   // To allocate a camera a SapAcqDevice object is used
   //SapAcqDevice saperaAcq("Nano-M2590_1", FALSE); // uses camera default settings
   //SapAcqDevice saperaAcq("Genie_M640", "MyCamera.ccf");//loads configuration file
   // Create SapBuffers (can change count if necessary)
   SapBuffer saperaBuffer(2, virtualAddress_Buffer, bufferPitch, bufferHeight,
      SapFormatMono8, SapBuffer::TypeScatterGather);
   // Allocate view object to display in an internally created window
   SapView saperaView(&saperaBuffer, (HWND)-1);
```

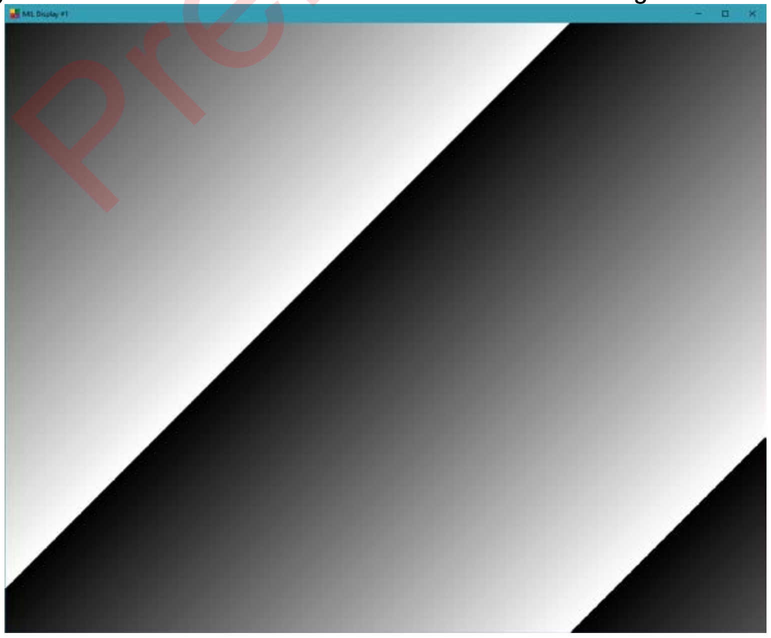
```
    // Allocate transfer object to link acquisition and buffer
    SapAcqToBuf saperaTransfer(&saperaAcq, &saperaBuffer, saperaXferCallback, &saperaView);
    // This is for a camera object
    //SapAcqDeviceToBuf *pTransfer = new SapAcqDeviceToBuf(pAcq, pBuffer, XferCallback, pView);

    // Create resources for all objects
    saperaAcq.Create();
    saperaBuffer.Create();
    saperaView.Create();
    saperaTransfer.Create();

    // Start a continuous transfer (live grab)
    saperaTransfer.Grab();
    cout << "Press any key to stop grab" << endl;
    cin.get();

    // pause to see the unprocessed image displayed
    // Stop the transfer and wait (timeout = 5 seconds)
    saperaTransfer.Freeze();
    saperaTransfer.Wait(5000);
    // MIL processing
    cout << "Show unprocessed image in MIL display - press any key to continue " << endl;
    cin.get(); // wait until a key has been hit

               // Display an unprocessed image buffer using MIL display function.
    MdispSelect(MilDisplay, MilImage);
    cout << "Show processed image (MimFlip) in MIL display - press any key to continue " << endl;
    cin.get(); // wait until a key has been hit

               //MIL processing function MimFlip
    MimFlip(MilImageToProcess, MilImageToProcess, M_FLIP_HORIZONTAL, M_DEFAULT);

    // Display the processed image buffer using MIL display function.
    MdispSelect(MilDisplay, MilImageToProcess);
    cout << "Press any key to terminate" << endl;
    cin.get(); // wait until a key has been hit

               // Release resources for all objects
    saperaTransfer.Destroy();
    saperaView.Destroy();
    saperaBuffer.Destroy();
    saperaAcq.Destroy();

    /* Free all allocations. */
    MbufFree(MilImage);
    MbufFree(MilImageToProcess);
    MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);

    return 0;
}
```

Note: SapBuffers are allocated with the MIL buffer's pitch as the width. This results in padding pixels when displaying the Sapera buffer with SapView. The MIL display mechanism compensates automatically for the difference in width and pitch when display the MIL buffer. It is recommended not to manipulate buffer areas outside the image boundaries since MIL may use this region for internal purposes.

The Sapera buffer uses the MIL buffer's pitch as the width (in this example, 32 pixels wider than the image width).



The MIL display mechanism accounts for the difference in image width and pitch.


MIL Display #1