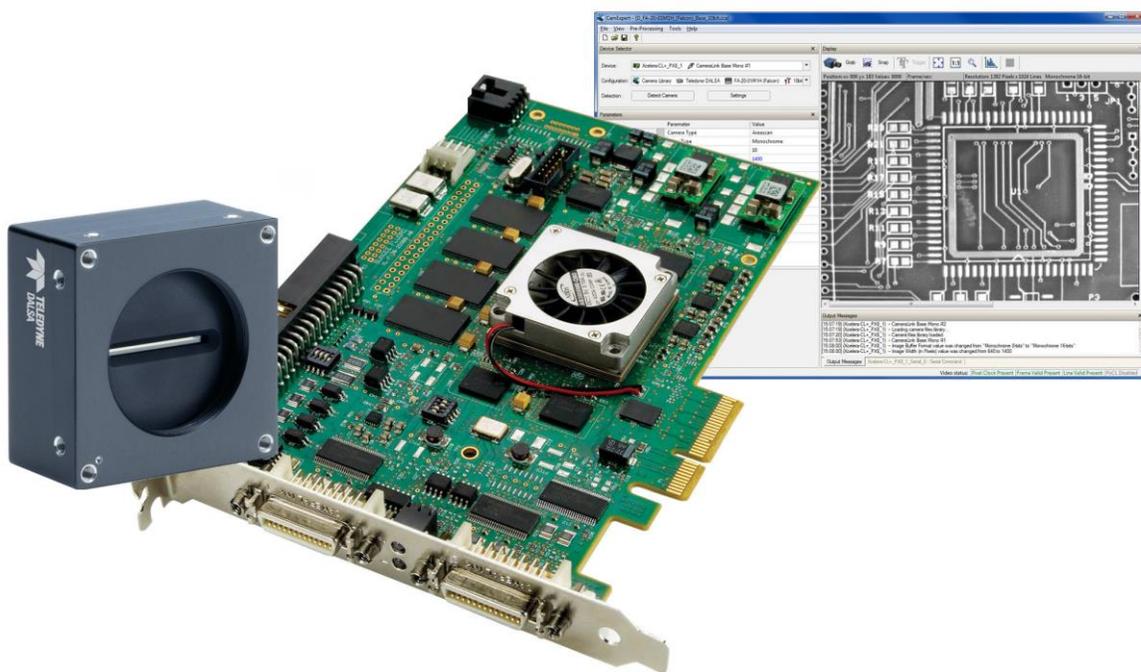


Sapera LT™ 8.60

User's Manual

sensors | cameras | frame grabbers | processors | **software** | vision solutions



P/N: OC-SAPM-USER0
www.teledynedalsa.com

NOTICE

© 2020 Teledyne Digital Imaging, Inc. All rights reserved.

This document may not be reproduced nor transmitted in any form or by any means, either electronic or mechanical, without the express written permission of TELEDYNE DALSA. Every effort is made to ensure the information in this manual is accurate and reliable. Use of the products described herein is understood to be at the user's risk. TELEDYNE DALSA assumes no liability whatsoever for the use of the products detailed in this document and reserves the right to make changes in specifications at any time and without notice.

Microsoft® is a registered trademark; Windows®, Windows® 7, Windows® 8 and Windows® 10 are trademarks of Microsoft Corporation.

All other trademarks or intellectual property mentioned herein belongs to their respective owners.

Printed on October 1, 2020

Document Number: OC-SAPM-USER0

Printed in Canada

About This Manual

This manual exists in Windows Help, and Adobe Acrobat® (PDF) formats (printed manuals are available as special orders). The Help and PDF formats make full use of hypertext cross-references. The Teledyne DALSA home page on the Internet, located at <http://www.teledynedalsa.com/imaging>, contains documents, software updates, demos, errata, utilities, and more.

About Teledyne DALSA

Teledyne DALSA, a business unit of Teledyne Digital Imaging Inc., is an international high performance semiconductor and electronics company that designs, develops, manufactures, and markets digital imaging products and solutions, in addition to providing wafer foundry services.

Teledyne Digital Imaging offers the widest range of machine vision components in the world. From industry-leading image sensors through powerful and sophisticated cameras, frame grabbers, vision processors and software to easy-to-use vision appliances and custom vision modules.

Contents

SAPERA LT ARCHITECTURE	4
APPLICATION ARCHITECTURE	4
<i>Library Architecture</i>	4
DEFINITION OF TERMS	6
SAPERA LT ++ AND SAPERA LT .NET CLASSES	7
<i>Sapera LT ++ Basic Classes by Subject</i>	7
<i>Sapera LT .NET Basic Classes by Subject</i>	8
<i>Sapera LT ++ and Sapera LT .NET Class Descriptions</i>	9
TRIGGER-TO-IMAGE-RELIABILITY FRAMEWORK	13
TELEDYNE DALSA ACQUISITION DEVICE FUNCTIONAL ARCHITECTURE.....	13
WHAT IS TRIGGER-TO-IMAGE RELIABILITY (T2IR)?.....	14
T2IR WITH A TYPICAL APPLICATION	14
ELEMENTS OF TRIGGER-TO-IMAGE RELIABILITY FRAMEWORK	15
THE RIGHT TARGET IMAGE ACQUISITION	16
MANAGING EXTERNAL TRIGGERS.....	19
TRACKING AND TRACING IMAGES	21
MONITORING THE ACQUISITION PROCESS	23
<i>Sapera Events</i>	23
OVERCOMING TOO MUCH DATA	25
ADVANCED DIAGNOSTICS	27
<i>Sapera Monitor</i>	27
<i>External LEDs</i>	28
<i>Sapera LogViewer</i>	28
<i>Sapera PCI Diagnostic Tool</i>	30
<i>Xtium Frame Grabber Diagnostic Tool</i>	31
<i>Diagnostic Tool Main Windo</i>	31
SAPERA LT API OVERVIEW	33
THE THREE SAPERA LT APIS	33
SAPERA LT ++ – CREATING AN APPLICATION	33
SAPERA LT .NET – CREATING AN APPLICATION	33
SAPERA LT ++ – OBJECT INITIALIZATION AND CLEANUP	33
<i>Example with SapBuffer Class Objects</i>	33
SAPERA LT .NET – OBJECT INITIALIZATION AND CLEANUP	34
<i>Example with SapBuffer Class Objects in C#:</i>	34
<i>Equivalent Code for Visual Basic .NET:</i>	35
<i>Equivalent Code for C++:</i>	35
SAPERA LT ++ – ERROR MANAGEMENT	36
<i>Setting the Current Reporting Mode</i>	36
<i>Monitoring Errors</i>	36
SAPERA LT .NET – ERROR MANAGEMENT	37
<i>Setting the Current Reporting Mode</i>	37
<i>Monitoring Errors</i>	39
CAPABILITIES AND PARAMETERS.....	40
<i>What is a Capability?</i>	40
<i>What is a Parameter?</i>	40
ACQUIRING IMAGES	41
REQUIRED CLASSES.....	41
FRAME-GRABBER ACQUISITION – REQUIRED STEPS	41
SAPERA LT ++ – SAMPLE ACQUISITION CODE	41
<i>Example Program using Sapera LT ++</i>	42

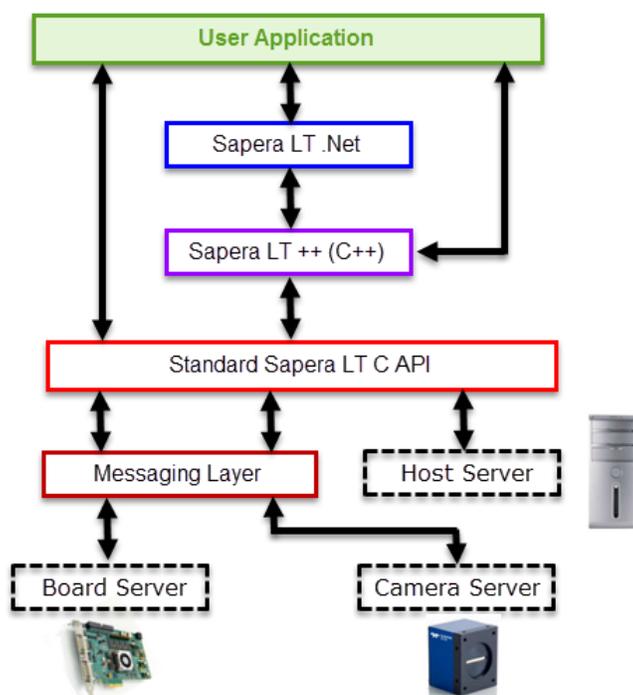
SAPERA LT .NET – SAMPLE ACQUISITION CODE	43
<i>Example Program using C#</i>	43
<i>Equivalent Program using Visual Basic .NET</i>	44
<i>Equivalent Example using C++</i>	45
SAPERA LT ++ – MODIFYING FRAME-GRABBER PARAMETERS.....	46
<i>Modifying Parameters Individually</i>	46
<i>Triggered Acquisition Example</i>	46
<i>Modifying Parameters by Group</i>	46
SAPERA LT .NET – MODIFYING FRAME-GRABBER PARAMETERS	47
<i>Modifying Parameters Individually</i>	47
<i>Triggered Acquisition Example</i>	48
<i>Modifying Parameters by Group</i>	48
SAPERA LT ++ – USING AN INPUT LOOKUP TABLE	49
<i>Sample Code</i>	49
SAPERA LT .NET – USING AN INPUT LOOKUP TABLE	50
<i>Sample Code for C#</i>	50
<i>Equivalent Code for Visual Basic .NET</i>	51
<i>Equivalent Code for C++</i>	51
SAPERA LT ++ – CAMERA ACQUISITION EXAMPLE.....	51
<i>Sample Code</i>	52
SAPERA LT .NET – CAMERA ACQUISITION EXAMPLE.....	53
<i>Sample Code for C#</i>	53
<i>Equivalent Code for Visual Basic .NET</i>	54
<i>Equivalent Code for C++</i>	55
SAPERA LT ++ – MODIFYING CAMERA FEATURES.....	56
<i>Accessing Feature Information and Values</i>	56
<i>Writing Feature Values by Group</i>	60
SAPERA LT .NET – MODIFYING CAMERA FEATURES.....	61
<i>Accessing Feature Information and Values</i>	61
<i>Writing Feature Values by Group</i>	69
DISPLAYING IMAGES	71
REQUIRED CLASSES.....	71
DISPLAY EXAMPLES.....	71
<i>Example using the Sapera LT ++ API</i>	71
<i>Example Code for C# using Sapera LT .NET</i>	72
<i>Equivalent Code for Visual Basic .NET using Sapera LT .NET</i>	72
<i>Equivalent Code for C++ using Sapera LT .NET</i>	72
SAPERA LT ++ – DISPLAYING IN A WINDOWS APPLICATION	73
<i>Sample Code Using the Visual C++'s MFC library</i>	73
SAPERA LT .NET – DISPLAYING IN A WINDOWS APPLICATION	74
<i>Partial C# Listing of a Windows Form Application</i>	75
<i>Equivalent Code for Visual Basic .NET</i>	76
<i>Equivalent Code for C++</i>	77
WORKING WITH BUFFERS	78
ROOT AND CHILD BUFFERS	78
<i>Sapera LT ++ Example – Parent Buffer with Two Children</i>	78
BUFFER TYPES	81
MULTIFORMAT IR BUFFERS	82
READING AND WRITING A BUFFER	83
<i>Sapera LT ++ – Access of a Buffer Object</i>	83
<i>Sapera LT .NET – Access of a Buffer Element</i>	84
<i>Sapera LT .NET – Access of a Buffer by an Array of Elements</i>	86
<i>Sapera LT .NET – Access of a Buffer via a Pointer</i>	88
PROCESSING BUFFERS	89
<i>Buffer State</i>	90
<i>Auto-Empty Mechanism</i>	90

<i>Transfer Cycling Modes</i>	91
<i>Execution flow for processing and displaying images</i>	95
SAPFLATFIELD COEFFICIENT CALIBRATION	98
<i>Flat Field File Format</i>	99
<i>Flat Field Correction Formula</i>	99
OFFSET COEFFICIENTS.....	100
GAIN COEFFICIENTS	100
PIXEL REPLACEMENT	101
TO CALIBRATE THE CAMERA'S FLAT FIELD COEFFICIENTS:	101
CODE SAMPLES USING SAPERA LT	102
DEPLOYING A SAPERA APPLICATION	103
RUNTIME INSTALLATIONS	103
<i>Installing Sapera LT Runtimes and Sapera LT Compatible Drivers</i>	103
TELEDYNE DALSA INSTALLERS	104
<i>Silent Mode Installation</i>	105
<i>Silent Mode Uninstall</i>	107
COMPILER RUN-TIME REDISTRIBUTION.....	108
CONTACT INFORMATION	109
SALES INFORMATION.....	109
TECHNICAL SUPPORT.....	109

Sapera LT Architecture

Application Architecture

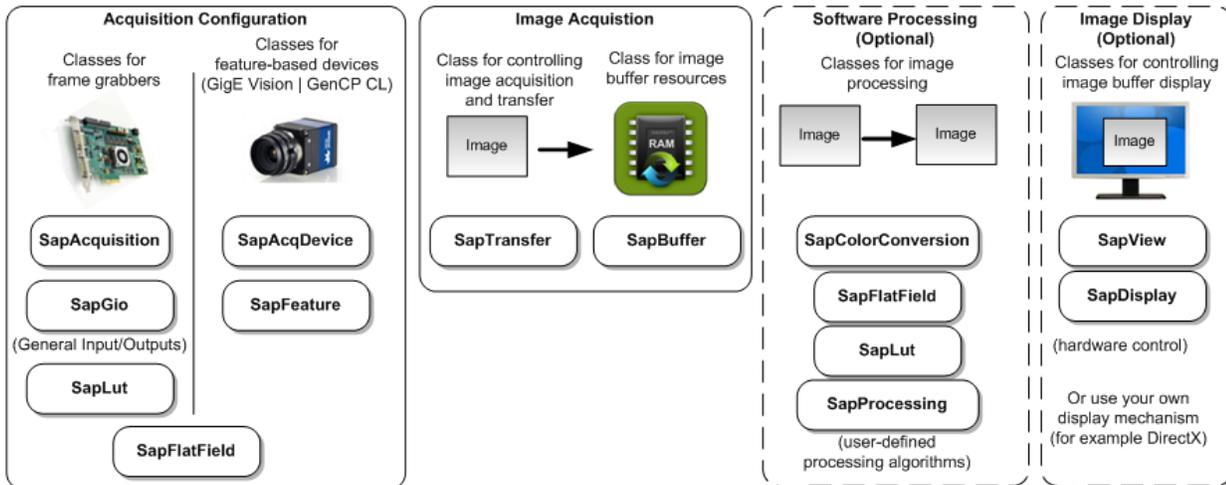
Whichever API is used (Sapera LT ++, Sapera LT .NET, or Standard C), the Sapera LT modular architecture allows applications to be distributed on different Sapera LT servers. Each server can run either on the host computer or on a Teledyne DALSA device. Sapera LT calls are routed to different servers via the Sapera LT messaging layer in a fashion completely independent of the underlying hardware.



Library Architecture

The typical machine vision application requires configuration of acquisition resources, image capture and transfer to memory buffers. These image buffers can then be processed or displayed, analyzed, with results determining subsequent processes. Events can also be monitored to trigger appropriate responses. The Sapera LT library architecture is organized around these basic machine vision functional blocks.

The following block diagram, while not exhaustive of all the classes available in Sapera LT, illustrates the major functional blocks with the corresponding classes.



The **Sapera LT ++** and **Sapera LT .NET Programmer's Manuals** provide a complete reference for all classes in the Spera API.



It is always recommended to use the source code provided with the demos and examples as both a learning tool and a starting point for your applications. For a complete list and description of the demos and examples included with Spera LT see the Spera LT Getting Started for Frame Grabbers Manual and Spera LT Getting Started for GigE Cameras and 3D Sensors Manual.

Definition of Terms

What is a server?

A Sopera LT server is an abstract representation of a physical device like a frame grabber, a processing board, a GigE camera or a desktop PC. In general, a Teledyne DALSA board is a server. Some processing boards, however, may contain several servers; this is true when using multi-processor boards.

A server allows Sopera LT applications to interact with the server's resources.

What is a static resource?

Resources attached to a physical device are called static resources. For example, a frame grabber can have an acquisition resource and a transfer resource. These resources can be manipulated to control a physical device through a Sopera LT server.

What is a dynamic resource?

A dynamic resource is an abstract representation of data storage (such as a buffer, lookup table, and so forth), or links that connect the data storage to static resources. Unlike static resources, dynamic resources are not dependent on physical devices; therefore, users on a specified server can freely create dynamic resources.

What is a module?

A module is a set of functions used to access and/or control a static or a dynamic resource. The complete Sopera LT Standard C API is composed of a series of modules organized in a particular architecture. See the *Sopera Basic Modules Reference Manual* for details.

Sopera LT ++ as a series of C++ classes or Sopera LT .NET as a series of .NET classes, encapsulate all of these concepts to offer the following benefits compared to the Standard API:

- Easier server management
- Consistent programming interface for static and dynamic resources
- Grouping of modules inside one class whenever appropriate

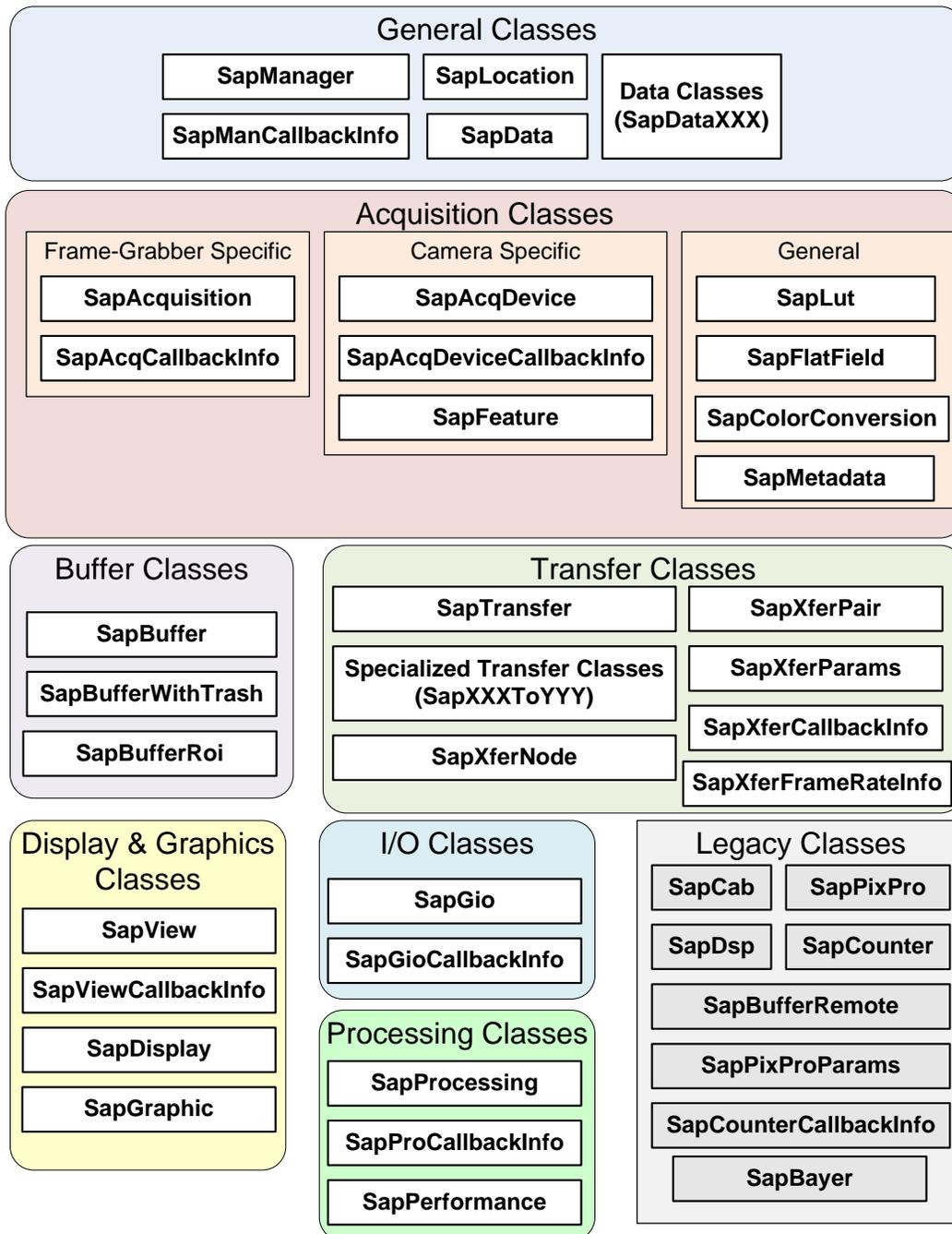
See the *Sopera LT ++ Programmer's Manual* for a hierarchy chart of all the Sopera LT ++ classes. See the *Sopera LT .NET Programmer's Manual* for details on the Sopera LT .NET Framework for Visual Studio.

Sapera LT ++ and Sapera LT .NET Classes

This section provides information on Sapera LT ++ classes and Sapera LT .NET classes. Class grouping diagrams are presented for each API followed by class descriptions which are often common to both Sapera LT ++ and Sapera LT .NET.

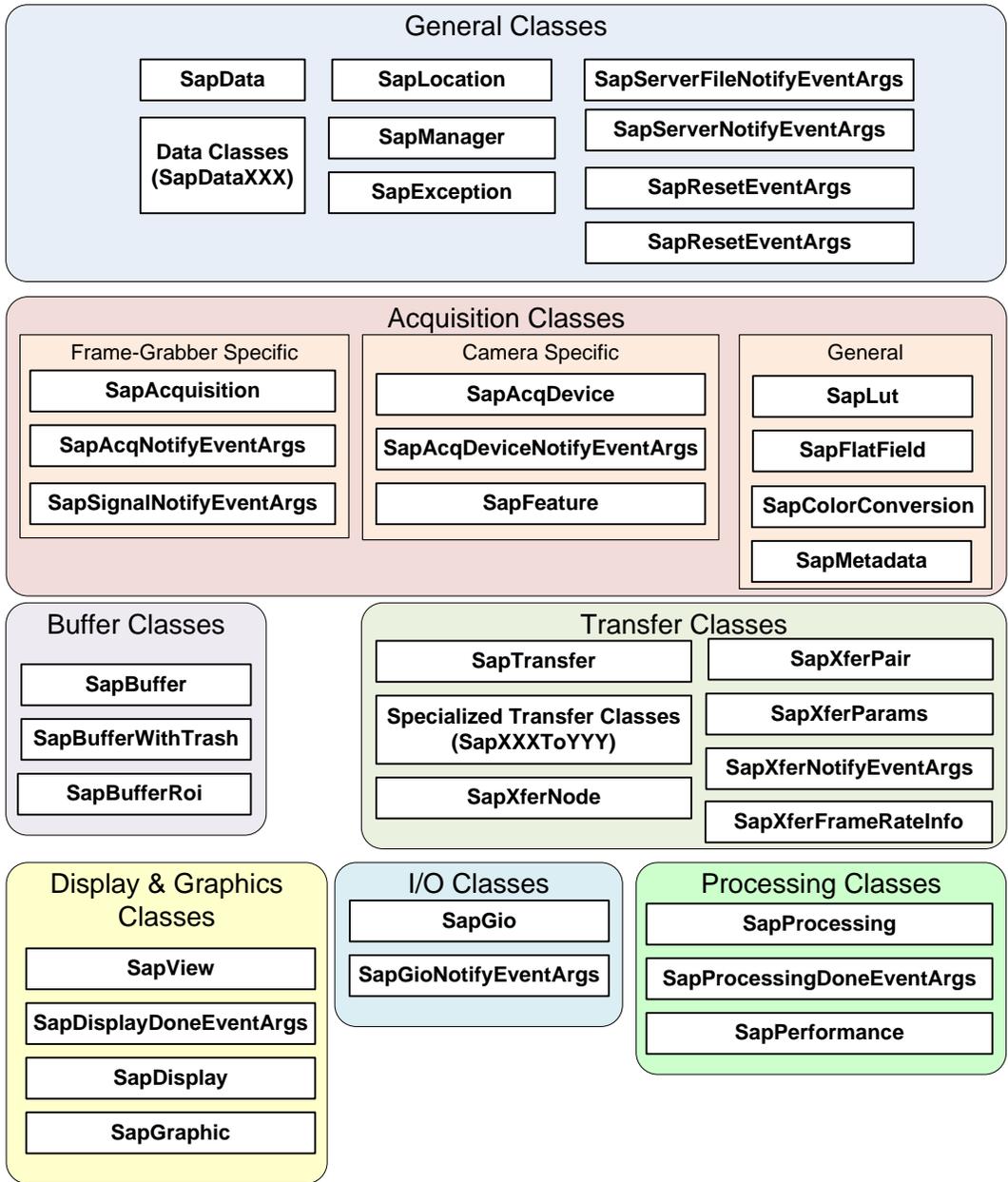
Sapera LT ++ Basic Classes by Subject

The following figure shows the main Sapera LT ++ classes that implement access to the Standard API module resources, as well as their relationship to other classes.



Sapera LT .NET Basic Classes by Subject

Below is a diagram along with a brief description of the main Sapera LT .NET classes, as well as their relationship to other classes.



Sapera LT ++ and Sapera LT .NET Class Descriptions

Sapera LT class descriptions cover the purpose of each class and mention any associated Sapera class used. Most classes apply to both C++ and .NET. Classes that are specific to an API are grouped together.

C++/.NET Class	Description
SapAcqDevice	The SapAcqDevice class includes the functionality to control an acquisition device on any Teledyne DALSA camera (for example, Genie M640). It is used as a source transfer node to allow data transfers from an acquisition resource to another transfer node, such as SapBuffer. It is used by the SapTransfer class.
SapAcqToBuf, SapAcqDeviceToBuf, SapBufToBuf, SapMultiAcqToBuf	<p>These specialized transfer classes are a set derived from SapTransfer that allow easy creation of the most commonly used transfer configurations.</p> <p>For example, setting up a transfer configuration from a SapAcquisition object (frame grabber) to a SapBuffer object normally requires many lines of code which call various functions in the SapTransfer class. Using the specialized class SapAcqToBuf instead reduces this to just one line of code.</p>
SapAcquisition	The SapAcquisition class includes the functionality to control an acquisition device on any Teledyne DALSA board with an acquisition section (for example, X64 Xcelera-CL PX4). It is used as a source transfer node to allow data transfers from an acquisition resource to another transfer node, such as SapBuffer. It is used by the SapTransfer class.
SapBuffer	The SapBuffer class includes the functionality to manipulate an array of buffer resources. A SapBuffer object can be used by a SapTransfer object as a destination transfer node to allow data transfers from a source node, such as SapAcquisition or SapAcqDevice. It may also be used as a source transfer node to allow transferring data to another SapBuffer. A SapBuffer object can be displayed using the SapView class and processed using the SapProcessing class.
SapBufferRoi	The purpose of the SapBufferRoi class is to create a rectangular region of interest (ROI) inside an existing SapBuffer object. The ROI has the same origin and dimensions for all buffer resources in the object.
SapBufferWithTrash	The SapBufferWithTrash class creates an additional resource called the trash buffer used when transferring data in real-time applications. The trash buffer is an emergency buffer used when the data transfer is faster than a processing task performed on the buffers. When processing is not fast enough to keep up with the incoming data, images are transferred temporarily into the trash buffer until stability is reestablished.
SapColorConversion	The purpose of the SapColorConversion class is to support Bayer conversion on images acquired from a camera, as well as other color image formats. When using any Teledyne DALSA board with an acquisition section, this class supports the color conversion function within the acquisition hardware, if available. Else this class also supports software-based conversion executed on the host PC.

SapData and SapDataXxx	SapData and its derived classes act as wrappers for Sapera LT data types, where each class encapsulates one data element of a specific type. They are used as property values, method arguments, or return values in various Sapera LT ++ and Sapera LT .NET classes.
SapDisplay	The SapDisplay class includes functionality to manipulate a display resource on the system display device (your computer video card) or any Teledyne DALSA board supporting a display section. There is at least one such resource for each display adapter (VGA board) in the system. Note that SapView objects automatically manage an internal SapDisplay object for the default display resource. However, you must explicitly manage the object yourself if you need a display resource other than the default one.
SapFeature	The SapFeature class includes the functionality to retrieve the feature information from the SapAcqDevice class. Each feature supported by the SapAcqDevice class provides a set of properties such as name, type, access mode, and so forth, that can be obtained through the feature module.
SapFlatField	The purpose of the SapFlatField class is to perform flat-field correction on images acquired from a camera or loaded from a disk. It supports this functionality both from the acquisition hardware (if supported) or from a software implementation.
SapGio	The purpose of the SapGio class is to control a block of general inputs and outputs—a group of I/Os that may be read and/or written all at once.
SapLocation	The SapLocation class identifies a Sapera server/resource pair.
SapLut	The SapLut class implements lookup table management. It is usually used together with the SapAcquisition and SapView classes to respectively manipulate acquisition and display lookup tables.
SapManager	The SapManager class includes methods for describing the Sapera resources present on the system. It also includes error management capabilities.
SapMetadata	The SapMetadata Class provides functions to manage GigE-Vision camera metadata (for Genie-TS and Linea GigE). When enabled, supported metadata (for example, the timestamp or device ID) is contained in the SapBuffer object.
SapPerformance	The SapPerformance class implements basic benchmarking functionality. It is used by the SapProcessing Class to evaluate the time it takes to process one buffer. You may also use it for your own benchmarking needs.
SapProcessing	The SapProcessing class allows you implement your own processing through a derived class.
SapTransfer	The SapTransfer class implements functionality for managing a generic transfer process—the action of transferring data from one source node to a destination node. The following classes are considered to be transfer nodes: SapAcquisition, SapAcqDevice, and SapBuffer.
SapView	The SapView class includes the functionality to show the resources of a SapBuffer object in a window through a SapDisplay object. An 'auto empty' mechanism allows synchronization between SapView and

	SapTransfer objects in order to show buffers in realtime without missing any data.
SapXferFrameRateInfo	The SapXferFrameRateInfo class provides frame rate statistics for the associated SapTransfer object. It is created automatically when a SapTransfer object is constructed.
SapXferNode	The SapXferNode class is the base class used to represent a source or destination transfer node involved in a transfer task managed by the SapTransfer class. The actual class for the node can be SapAcqDevice, SapAcquisition, or SapBuffer.
SapXferPair	The SapXferPair class describes a pair of source and destination nodes for the SapTransfer class.
SapXferParams	The SapXferParams class stores parameters needed by a transfer task managed by the SapTransfer class.

C++ Only Class	Description
SapAcqCallbackInfo	The SapAcqCallbackInfo class acts as a container for storing all arguments to callback functions for the SapAcquisition class.
SapAcqDeviceCallbackInfo	The SapAcqDeviceCallbackInfo class acts as a container for storing all arguments to callback functions for the SapAcqDevice class.
SapGioCallbackInfo	The SapGioCallbackInfo class acts as a container for storing all arguments to callback functions for the SapGio class.
SapManCallbackInfo	The SapManCallbackInfo class acts as a container for storing all arguments to callback functions for the SapManager class.
SapProCallbackInfo	The SapProCallbackInfo class acts as a container for storing all arguments to callback functions for the SapProcessing class.
SapViewCallbackInfo	The SapViewCallbackInfo class acts as a container for storing all arguments to callback functions for the SapView class.
SapXferCallbackInfo	The SapXferCallbackInfo class acts as a container for storing all arguments to callback functions for the SapTransfer class.

.NET Only Class	Description
SapAcqDeviceNotifyEventArgs	The SapAcqDeviceNotifyEventArgs class stores arguments for the AcqDeviceNotify event of the SapAcqDevice class.
SapAcqNotifyEventArgs	The SapAcqNotifyEventArgs class stores arguments for the AcqNotify event of the SapAcquisition class.
SapDisplayDoneEventArgs	The SapDisplayDoneEventArgs class stores arguments for the DisplayDone event of the SapView class.
SapErrorEventArgs	The SapErrorEventArgs class stores arguments for the Error event of the SapManager class.
SapException	The SapException class is the base class common to the SapLibraryException and SapNativePointerException classes.
SapGioNotifyEventArgs	The SapGioNotifyEventArgs class stores arguments for the GioNotify event of the SapGio class.
SapLibraryException	The SapLibraryException class is thrown when error conditions reported as exceptions occur in the Sapera LT libraries.
SapManVersionInfo	The SapManVersionInfo class includes version information corresponding to the currently installed copy of Sapera LT.
SapNativePointerException	The SapNativePointerException class is thrown when internal pointer error conditions occur.
SapProcessingDoneEventArgs	The SapProcessingDoneEventArgs class stores arguments for the ProcessingDone event of the SapProcessing class.
SapResetEventArgs	The SapResetEventArgs class stores arguments for the Reset event of the SapManager class.
SapServerFileNotifyEventArgs	The SapServerFileNotifyEventArgs contains the arguments to the application handler method for the ServerFileNotify event of the SapManager class.
SapServerNotifyEventArgs	The SapServerNotifyEventArgs class stores arguments for the ServerNotify event of the SapManager class.
SapSignalNotifyEventArgs	The SapSignalNotifyEventArgs class stores arguments for the SignalNotify event of the SapAcquisition class.
SapXferCounterStampInfo	The SapXferCounterStampInfo class stores information about the counter-stamp capabilities for a specific transfer pair.
SapXferNotifyEventArgs	The SapXferNotifyEventArgs class stores arguments for the XferNotify event of the SapTransfer class.

Trigger-to-Image-Reliability Framework

Machine vision systems are used for inspection, recognition and guidance applications in different types of manufacturing and process industries. The vision systems incorporate area and line scan, color and monochrome cameras and frame grabbers to provide systems that operate autonomously to perform inspection tasks on 100% of the objects. The vision systems must produce reliable results under a variety of operating conditions to help improve quality of products and processes. Teledyne DALSA's cameras and frame grabbers incorporate the Trigger-to-Image Reliability (T2IR) technology framework to ensure data reliability from the time an object is placed in front of camera until a decision is made to accept or reject the inspected objects. The T2IR framework is rooted in hardware and software design principles to ensure reliability and is delivered as hardware features and capabilities, standalone GUI based tools, and programming API. The T2IR framework permits applications to track, trace, debug, recover and prevent any data loss.

Teledyne DALSA Acquisition Device Functional Architecture

Let us take brief look at the main building blocks of acquisition device architecture of Teledyne DALSA hardware devices:

Acquisition Control Unit (ACQ): This conceptual functional block is responsible for control of the image acquisition capabilities and features. It is responsible for ensuring that correct images are generated and at the correct moment. The ACQ is responsible for managing camera control signals under software control. In addition, the ACQ provides running status of the image acquisition sequence.

Data Transfer Engine (DTE): The DTE is responsible for moving data in and out from onboard memory to the host memory. The functional block consists of intelligent DMA architecture and scales with performance specifications of the hardware. The DTE is also responsible for generating all notifications necessary to manage image flow as per the T2IR framework.

Image Processing Unit (IPU): The IPU performs real-time embedded image processing. The capabilities of IPU vary based on the price performance criteria targeted for the acquisition hardware. The embedded processing varies in complexity from color space conversion in simple frame grabbers and cameras to image analysis to controlling external devices on vision processors.

With the Teledyne DALSA image acquisition device functional architecture in mind, let us take a closer look at T2IR framework to understand what is it, its principal building blocks and how it helps reduce costs.

What is Trigger-to-Image Reliability (T2IR)?

The process of image acquisition for machine vision applications begins by sending a signal, known as an external trigger, to the camera to start generating images. As such, no matter the nature of vision system, a trigger signal represents a crucial starting point of the image acquisition sequence to enable image processing and analysis for decisions further down the inspection process.

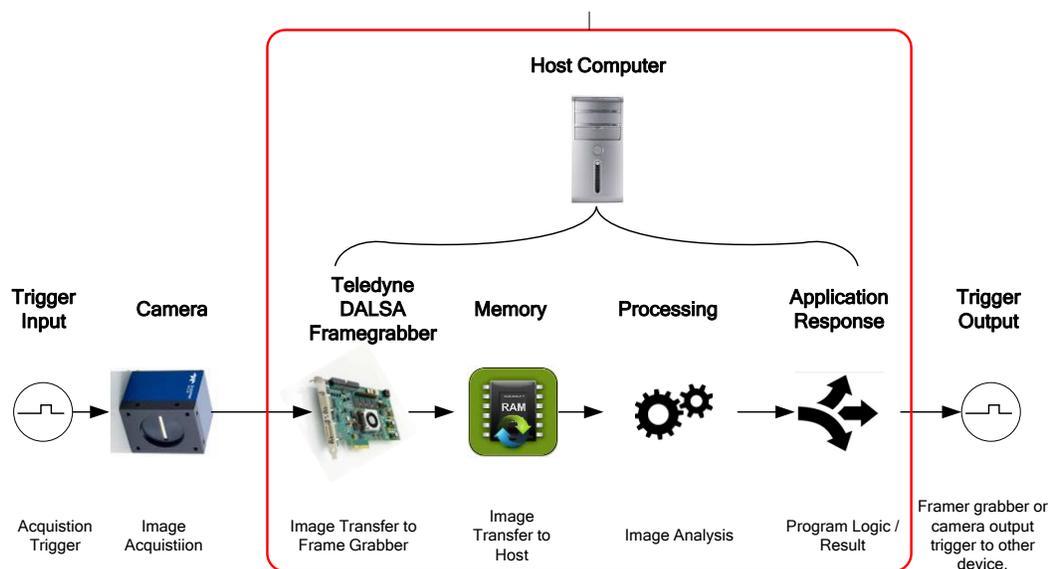
The reliability of a vision system is reflected by its ability to handle both predictable and unpredictable trigger signals. The parts of vision system – image acquisition and control - must operate in harmony to achieve this reliability. A controlled response to system events is directly related to the quality of information needed to produce products with consistent quality. This helps lower costs by increasing the system uptime and yield.

T2IR is a combination of hardware and software features that work together to improve the reliability of your vision system. T2IR features deliver full system level monitoring, control, and diagnostics capability. It lets you reach inside your vision system to audit and debug image flow. You can trace the flow of data from image capture right through transfer to host memory. You can even store images temporarily in the onboard memory to overcome unexpected transfer bottlenecks. That means no lost data, no false data and a clear source to identify and track any errors. Sapera T2IR features accomplish these tasks in a non-intrusive manner that does not interfere with the applications.

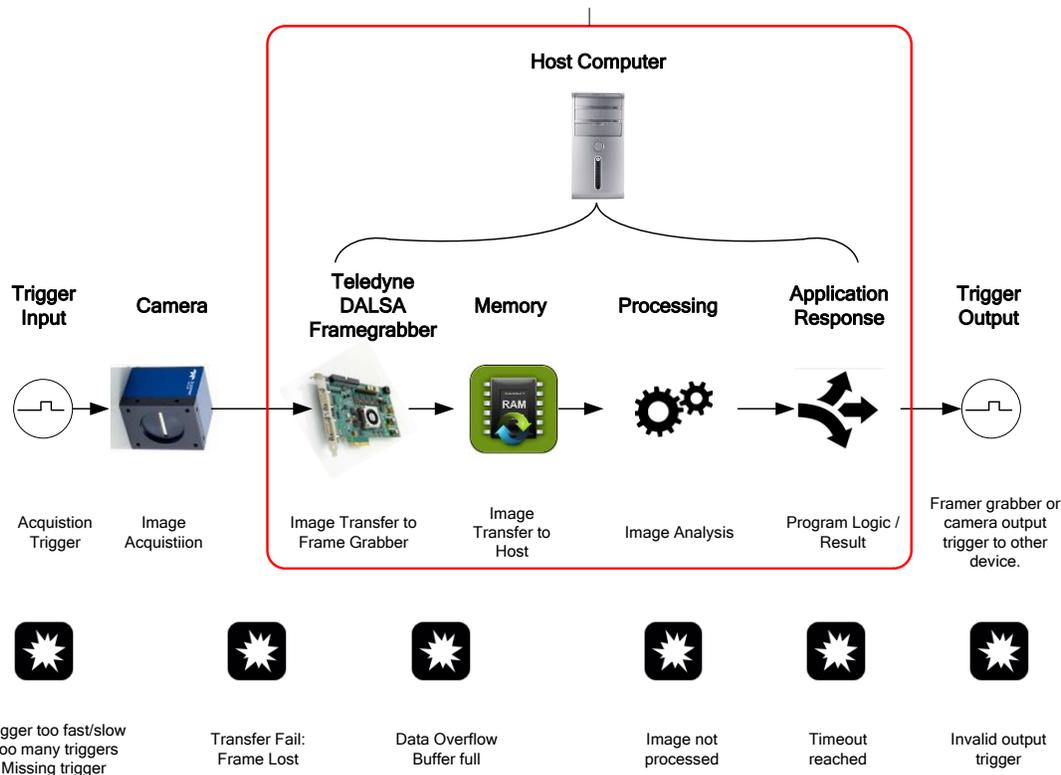
T2IR ensures robust and reliable operations to produce repeatable results.

T2IR with a Typical Application

A typical imaging application follows a processing chain similar to the one illustrated below:



T2IR aims to handle the common breakdown points in this chain such that corrective or preventative action can be taken and to eliminate the possibility of unknown faults/application failure.



Elements of Trigger-to-Image Reliability Framework

T2IR framework capabilities are available in three principal ways:

- Sapera API programming functions: integrated in user applications for dynamic inline tracking, tracing and control.
- Standalone GUI based tools: enable advanced diagnostics that can run concurrently with Sapera applications without performance impact.
- Visual indicators – provide indispensable internal device status details from the time the system powers up to operating mode.

The functionality of the T2IR GUI tools is also available as part of the Sapera API. Users can access this functionality directly from their own application using the Sapera SDK. As part of T2IR all Teledyne DALSA hardware incorporates LEDs to indicate the device operating status. These visual indicators are indispensable before any host application can run or when camera and host are located some distance apart.

The following table summarizes the key benefits offered by various functional elements of T2IR framework:

T2IR Elements	Benefit
The right target image acquisition	Acquire the best quality images with object details critical to make correct decisions.
Managing External Triggers	Ensures synchronization between image acquisition and object motion. Reduced image artifacts due to motion and provide control response to expected and unexpected external events.
Tracking and Tracing Images	Continuous coverage of the entire images flow reduces waste and improves up time.
Monitoring the Acquisition and Transfer Process	Enables preventive action if resource usage exceeds a predetermined threshold, selectively keeping or discarding images to sustain processing speed.
Overcoming Too Much Data	Handle peak loads to avoid data loss, ensure smooth operations.
Ensuring Data Quality	Helps increase uptime and reduce waste .
Advanced Diagnostics	Rapid pinpointing of errors for speedy diagnostic and preventive actions.

The Right Target Image Acquisition

Sapera LT supports programmable delay timers on strobe and trigger signals to precisely control the image acquisition timing to acquire the right target image.

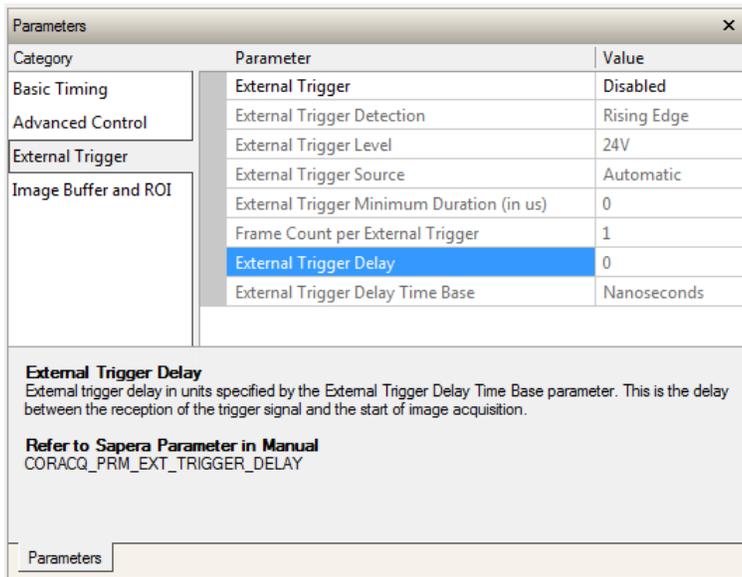
Teledyne DALSA camera and frame grabber products incorporate various levels of control functions for automating imaging applications. A good starting example is the integration of the trigger and strobe control functions into onboard hardware.

This sounds simple enough: a trigger input generates a strobe output for lighting control and camera exposure. However, there are circumstances in which a delay between the trigger input and the strobe output is required; for example, if the camera and lighting units are not in the same position on a conveyor as the trigger sensor. Coordinating these two events through software is almost impossible and certainly not reliable (especially given the variations in command execution of the Windows operating system). To solve this problem Teledyne DALSA has incorporated programmable delay timers between these two signals.

The delay timers give developers a mechanism for establishing a precise delay between the trigger input and firing of the lighting and camera exposure. However, this amount of programmed delay is calculated based on the theoretical speed of the production line. If the actual speed is not constant (a common occurrence), the position of the object in the resulting image may not be suitable for analysis. Therefore, for reliable image acquisition the delay has to be linked to the speed of the object. This is done using the pulse output from an encoder attached to a rotating part of the conveyor system. Expressing the delay in terms of encoder ticks synchronizes it with the actual speed of the production line. As a result, the object is always at the same location in the image regardless of the speed of production line.

The easiest way to program trigger parameters is to use Sapera CamExpert. Sapera CamExpert is camera configuration tool that offers intuitive graphical user interface and live image display for faster camera setup. CamExpert works with all Teledyne DALSA frame grabbers, GigE Vision and GenCP compliant cameras.

For example, the External Trigger parameters are all grouped in one category in the Parameters panel (shown here for the Xtium-CL PX4 frame grabber):



When you are satisfied with all the parameters settings these parameters can be saved in a configuration file and later retrieved by the application at run time.

The example below shows how to access previously stored camera configuration file for the Xtium frame grabber in C++:

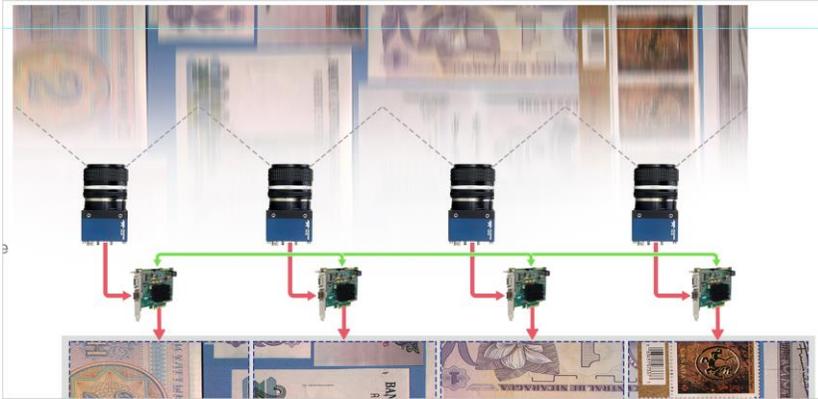
```
// Allocate acquisition object
SapAcquisition *pAcq = new SapAcquisition(SapLocation ("Xtium-CL_PX_1", 0),
"MyCamera.ccf");
```

This synchronization achieves the first goal of Trigger-to-Image Reliability: the camera is properly controlled to capture the image of the target being inspected. Of course, these hardware features are under software control, but, once initialized, they act independently of any software execution, leading to predictable results.

System designers want to build systems that offer scalable performance while minimizing costs. In some cases it might more economical to combine multiple lower resolution cameras and optics to construct higher resolution images while in some other it might be necessary to distribute very high speed images across multiple computers to minimize image processing and analysis.

In all cases when multiple acquisition devices are used, it is important that all devices operate synchronously to produce images that are error free and ready to use. T2IR framework capabilities permit this by incorporating critical features to achieve image acquisition synchronization in hardware and software, without the need for external synchronization and data replicating devices., This T2IR synchronization feature also permits implementation of different image processing setups to achieve a target processing time. Let us closely look at some of the commonly used system configurations.

For example, one application can combine images from two cameras in one buffer or split the image from one camera across multiple frame grabbers to overcome processing bandwidth limitations.

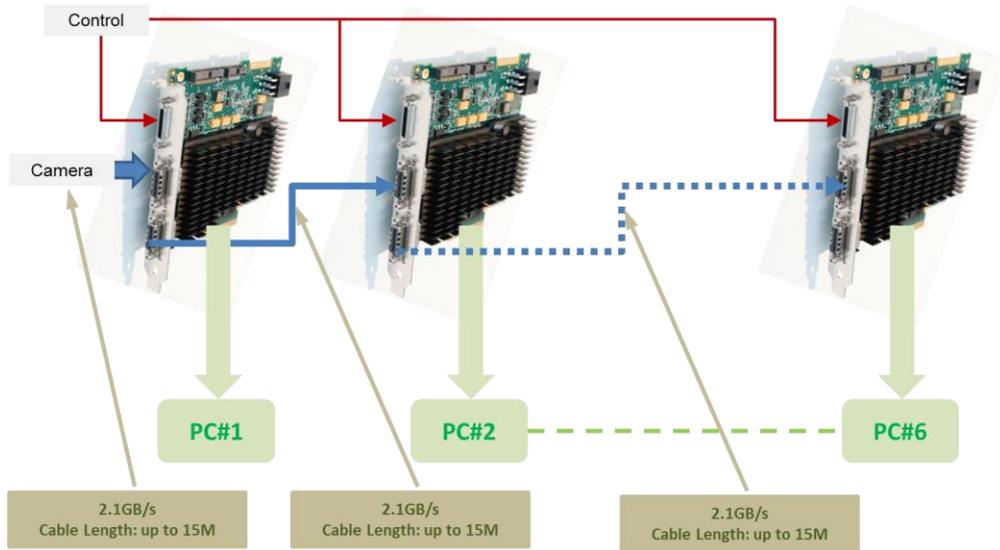


In cases where images from different devices must be combined in one buffer, Teledyne DALSA GigE Vision cameras, (such as Genie Nano and Linea GigE) and frame grabbers (such as Xtium-CL MX4) incorporate the necessary hardware to work under Sapera LT to capture images in one seamless Sapera buffer. Teledyne DALSA’s Xcelera and Xtium series frame grabbers, for example, offer dedicated hardware signals to synchronize multiple boards and cameras together. The trigger source can be easily set using CamExpert. Sapera LT SDK also provides dedicated demo applications with source code to jump start the development efforts.

Parameters		
Category	Parameter	Value
Basic Timing	External Trigger	Enable
Advanced Control	External Trigger Detection	Rising Edge
	External Trigger Level	24V
External Trigger	External Trigger Source	Automatic
Image Buffer and ROI	External Trigger Minimum Durati...	Automatic
	Frame Count per External Trigger	External Trigger #1 External Trigger #2
	External Trigger Delay	Board Sync #1
	External Trigger Delay Time Base	Board Sync #2

Similarly, Genie Nano and Linea GigE cameras series are also capable of accepting external input signals that can be distributed to other cameras for synchronization.

The Xtium-CLHS series also includes a dedicated image data forwarding port. This allows image processing to be distributed across multiple computers or capture images in the same PC using two frame grabbers when camera bandwidth exceeds the 2.1GB/s limit of the CLHS cable.



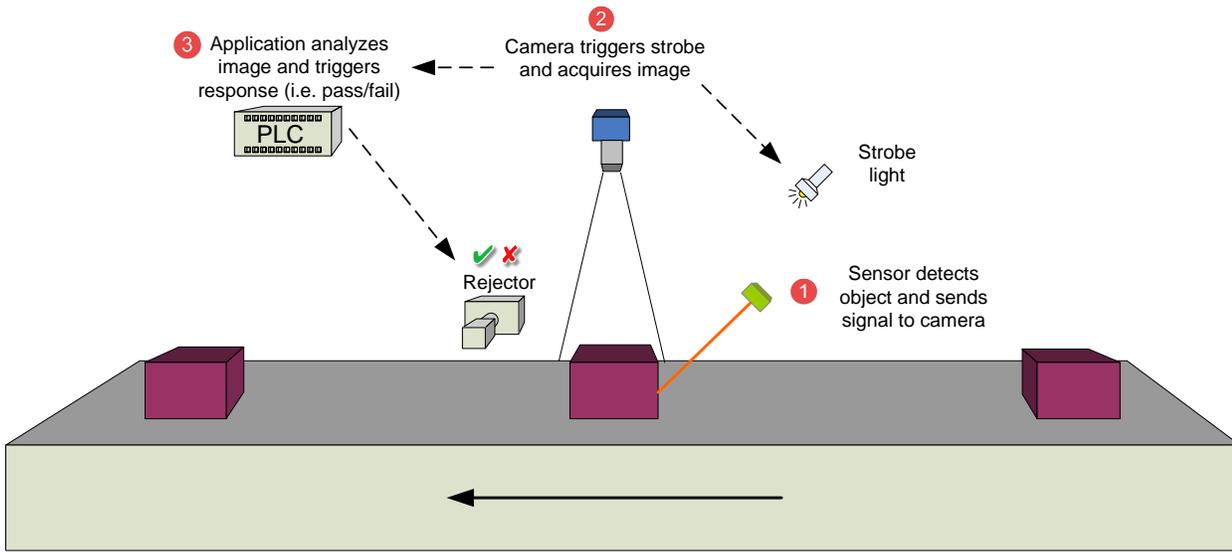
Managing External Triggers

Detecting Valid and Invalid Triggers

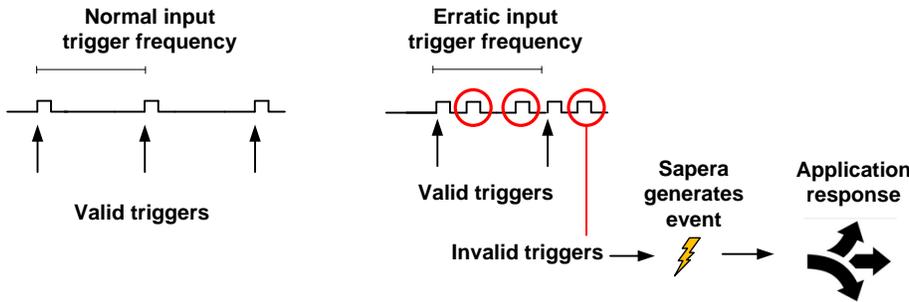
External trigger management involves functions and capabilities that are essential to ensure reliability of the trigger signals seen by the vision system. It involves managing situations when the system receives too many triggers for it to properly handle situations when the signal does not truly represent a trigger event. Let us see how T2IR handles both predictable and unpredictable triggers to ensure reliability of a vision system.

A first criterion for a valid trigger is that a trigger has to represent an actual “part-in place” for inspection. A false trigger is a signal that is not associated with a part in place. False triggers can be caused by jitter resulting from electrical noise or glitches associated with mechanical actuators and motors. T2IR capabilities offer an effective way to reduce faulty triggers by ensuring that the signal remains active for a minimum duration before it can be considered as valid for the acquisition. For added flexibility Teledyne DALSA products offer this T2IR feature as a user programmable parameter.

The following figure illustrates a typical imaging application.



The following figure illustrates valid and invalid triggers.



After the probability of spurious triggers is minimized, user applications can be programmed to handle the other extreme, appropriately called “over trigger” conditions. An over-trigger condition occurs when the camera receives a trigger but is busy acquiring previous image. Care must be given to the fact that ,in some cases, sending a trigger while grabbing the previous line or frame is desirable to minimize the dead time between frames or lines(in case of line scan cameras).

Typical causes for an over-trigger state can be that the image generated from the previous trigger is still being processed, or the sensor is currently being readout or exposed for the next image (note that some cameras support exposing the sensor during readout, which allows for a higher frame rate than otherwise possible).

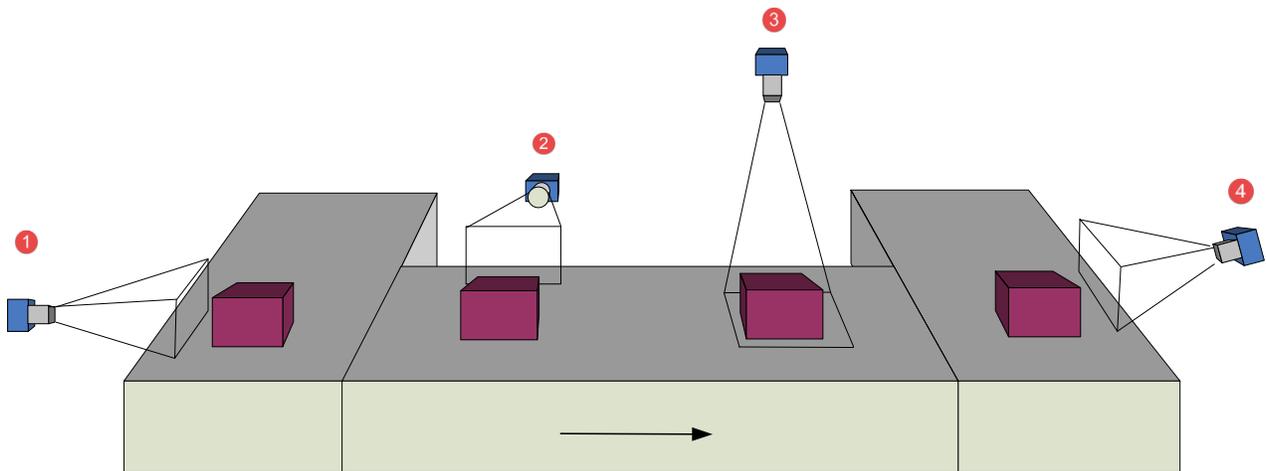
The T2IR capabilities allow applications to tolerate over-trigger situations and track them if a system starts to lose images. When frames are lost, T2IR capabilities notify Sapera based user applications with event messages for remedial actions. T2IR framework helps applications to maintain control despite timing fluctuations in trigger generation.

Tracking and Tracing Images

While we have progressed in our discussion from the point of detecting our targets, to triggering strobe lights and camera acquisition when the target is in the right location and reading the correct image data from the sensor, this is only the start of designing a reliable machine vision system. Another major issue is coordinating the collection of image data and correlating these images with physical objects moving through a material-handling system.

Trigger-to-Image Reliability uses an important design concept to assist engineers in creating reliable and repeatable systems: **image tagging** or **timestamps**. To illustrate, let us use an example of a material-handling unit processing up to 3,600 parts per minute (ppm). For factory production lines to work at maximum speed and each image must be tagged such that the downstream decision to keep, discard or re-inspect is carried out on the correct object.

More advanced applications may require inspection from multiple views. Continuing our previous example, let us assume the object has to be inspected on each side, each with different lighting, at the same frame rate. Now the constraints evolve from inspecting 3600 parts per minute to handling 14,400 images per minute. In this scenario, the imaging system must correlate four different acquisitions before making the final decision to accept, reject or re-inspect the object.



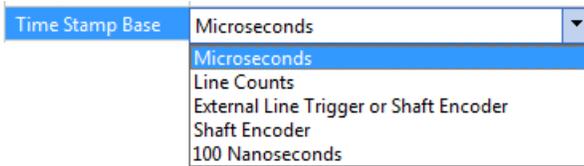
With synchronized acquisition timestamps, the 4 images for each item are:

- Image 1 timestamp = ①
- Image 2 timestamp = ① + x ticks = ②
- Image 3 timestamp = ① + y ticks = ③
- Image 4 timestamp = ① + z ticks = ④

Where x, y and z are the expected intervals between acquisitions.

Figure 4: Object Tracing

The image tags (timestamps) are generated either from an onboard hardware clock, the PC clock or increments using an external signal, be it a trigger, encoder tick or another pulse input at the time of image acquisition and/or image transfer to the host. For example, the Xtium-CL MX4 provides the following hardware timestamps:



The acquisition frame start timestamps from the device and host (issued when it starts to receive the frame) are saved in Sapera buffers with the images. These timestamps can be retrieved by the host applications using Sapera functions for analysis.

Since there is a time lag between image capture and analysis, the image timestamps can be used to ensure that the system acts on the correct object. Timestamps can also be used to precisely measure the acquisition or processing rates. It can also be used to determine if any loss of data has occurred by comparing the time lapse between successive frames.

In C++, callback functions are used to access the timestamps; whenever registered events occur, the associated callback function is executed.

```
//Register acquisition events
success = pAcqDevice->RegisterCallback("FrameStart", MyAcquisitionCallback, pBuffer)
...
//Callback function for events
void MyAcquisitionCallback(SapAcqDeviceCallbackInfo *pInfo)
{
...
pInfo->GetAuxiliaryTimeStamp(&myAuxTimeStampValue);
pInfo->GetHostTimeStamp(&myHostTimeStampValue);
...
}
```

For .NET, a similar mechanism uses the EnableEvent method and AcqDeviceNotify event to call the associated event handler.

```
//Enable acquisition event
device.EnableEvent("FrameStart");
...
//Create event handler to execute callback for enabled events
device.AcqDeviceNotify += new SapAcqDeviceNotifyHandler(AcqDeviceCallback);
...
//Callback function for events
static void AcqDeviceCallback(Object sender, SapAcqDeviceNotifyEventArgs args)
{
myVariableA = args.AuxTimeStamp);
myVariableB = args.HostTimeStamp;
```

Monitoring the Acquisition Process

When the machine vision system is capturing the right data and tracking objects throughout the cycle for acceptance or rejection, it is now time to transfer the image data from the onboard memory to system memory.

Trigger-to-Image Reliability framework includes a set of software tools to ensure that all required images were captured accurately into onboard memory. While it is possible to continuously check the status to monitor system operations, in practice it comes at the expense of system performance. T2IR uses the concept of events that are issued by the acquisition devices to notify the application if certain status flags have changed. This allows applications to operate more optimally as it gets interrupted from its main processing task only when an event has occurred. Since, these notifications are handled at the user application level, the applications have complete freedom to decide how best to handle them.

The table below summarizes the Sapera events associated with image capture and transfer sequences into the host memory.

Sapera Events

Event	Description
EndOfEven	End of even field
EndOfField	End of field (odd or even)
EndOfFrame	End of frame
EndOfLine	After a specific line number <i>eventType</i> = EndOfLine <i>lineNum</i>
EndOfNLines	After a specific line number (linescan cameras only) <i>eventType</i> = EndOfNLines <i>numLines</i>
EndOfOdd	End of odd field
EndOfTransfer	End of transfer, that is, after all frames have been transferred following calls to SapTransfer.Snap or SapTransfer.Grab/SapTransfer.Freeze.
FieldUnderrun	The number of active lines per field received from a video source is less than it should be.
LineUnderrun	The number of active pixels per line received from a video source is less than it should be.
StartOfEven	Start of even field
StartOfField	Start of field (odd or even)
StartOfFrame	Start of frame
StartOfOdd	Start of odd field

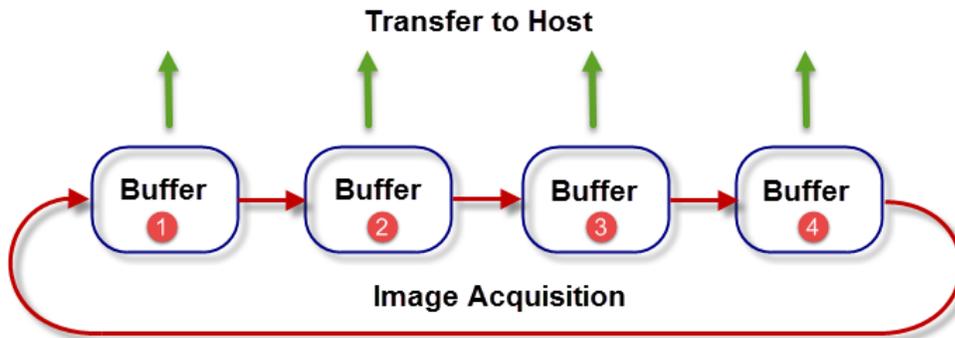
In addition to these events the status of the following acquisition signals can be monitored in the host application. Note that the availability of status signals varies with the hardware used and this availability can be verified programmatically. The SapAcquisition::GetSignalStatus(.....) function can be used to monitor these signals.

Status signal to inquire.	Description
SapAcquisition::SignalNone	No signal
SapAcquisition::SignalHSyncPresent	Horizontal sync signal (analog video source) or line valid (digital video source)
SapAcquisition::SignalVSyncPresent	Vertical sync signal (analog video source) or frame valid (digital video source)
SapAcquisition::SignalPixelClkPresent /	Pixel clock signal. For CameraLink devices, this status returns true if a clock signal is detected on the base cable.
SapAcquisition::SignalPixelClk1Present	
SapAcquisition::SignalPixelClk2Present	Pixel clock signal. For CameraLink devices, this status returns true if a clock signal is detected on the medium cable.
SapAcquisition::SignalPixelClk3Present	Pixel clock signal. For CameraLink devices, this status returns true if a clock signal is detected on the full cable.
SapAcquisition::SignalPixelClkAllPresent	Pixel clock signal. For Camera Link devices, true if all required pixel clock signals have been detected by the acquisition device based on the CameraLink configuration selected.
SapAcquisition::SignalChromaPresent	Color burst signal (valid for NTSC and PAL)
SapAcquisition::SignalHSyncLock	Successful lock to an horizontal sync signal, for an analog video source
SapAcquisition::SignalVSyncLock	Successful lock to a vertical sync signal, for an analog video source
SapAcquisition::SignalPowerPresent	Power is available for a camera. This does not necessarily mean that power is used by the camera, it only indicates that power is available at the camera connector, where it might be supplied from the board PCI bus or from the board PC power connector. The returned value value is FALSE if the circuit fuse is blown, therefore power cannot be supplied to any connected camera.
SapAcquisition::SignalPoCLActive	Power to the camera is present on the Camera Link cable
SapAcquisition::SignalPixelLinkLock	Lane lock signal. For HSLink and CLHS devices, true if all required lane lock signals have been detected by the acquisition device based on the HSLink or CLHS configuration selected.

Overcoming Too Much Data

Tracks Occurrences of Trashed Frames

Let us build on our previous example: the system processing 3600 parts per minute that involves image acquisition from 4 sides simultaneously results in a machine vision system acquiring, processing and analyzing 14,400 images per minute. Proper system design dictates that a certain amount of over-capacity be built into the system to handle peak loads. Trigger-to-Image Reliability framework delivers peak load capacity through the concept of circular buffers. It also combines this with user notifications for continuous tracking. While handling peak-load, it is important to monitor the image queue to ensure that various parts of the system stay in-sync and if any variation occurs it is identified and promptly communicated to the user application.



The scalable nature of T2IR framework has allowed Teledyne DALSA to add sophisticated parameter switching capability in its hardware products that are well suited for use with circular buffers. Teledyne DALSA Genie cameras, for example, allow users to change trigger delay, strobe outputs, exposure delay and duration, gain, LUTs and FFC (flat field coefficients) on a frame by frame basis. Similarly, the Xtium-CL MX4 frame grabber allows users to switch flat-field and LUTs on a frame by frame basis. When activated, these advanced switching features operate entirely in the acquisition device without using the host CPU resources. Furthermore, the images generated while switching parameters can be saved as a sequence of images.

T2IR provides a broad range of options to handle situations involving too much data. It provides users with necessary information to discard images safely while preserving the accuracy of results from images that were processed. When every image counts discarding images inevitably leads to reduced throughput. Thus, even when discarding images care must be given to minimize the impact on throughput. The T2IR framework allows applications to discard images early in the acquisition pipeline if it is determined that the system won't be able to handle the images subsequently. The T2IR framework uses a concept of "trash" buffers to discard incoming images efficiently. When a system is not able to handle the incoming data, the acquired images are transferred into the "trash buffer".

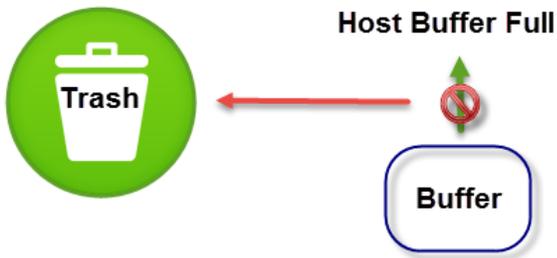


Image Acquisition

When this occurs, the user application is notified through a data-overflow event. The Xtium family of frame grabbers when transferring images to the host buffers, for example, monitors downstream bottlenecks and immediately discards images at the acquisition source instead of transferring all the way in the host memory and then discarding it. This allows the acquisition section to return to a ready state immediately to capture next image.

Ensuring Data Quality

Generally machine vision cameras are responsible for handling over-trigger situations elegantly and in a predictable fashion. The over-trigger situation for a camera occurs when the rate of triggers for a camera exceeds its maximum frame rate or line rate capability. In cases where the camera does not respond properly and stops sending images at all, T2IR function provide means to recover from this situation and generates notifications to the user application. This is a standard functionality on all Teledyne DALSA frame grabbers. Trapping and handling lost lines or frames is an important factor to determine the reliability of the acquisition system and has a direct impact on the accuracy of results. For example, for a line scan camera, a missing line alters the aspect ratio of the object in the image, causing the processing algorithm to produce incorrect results. For area-scan cameras, similarly, it could imply missing objects.

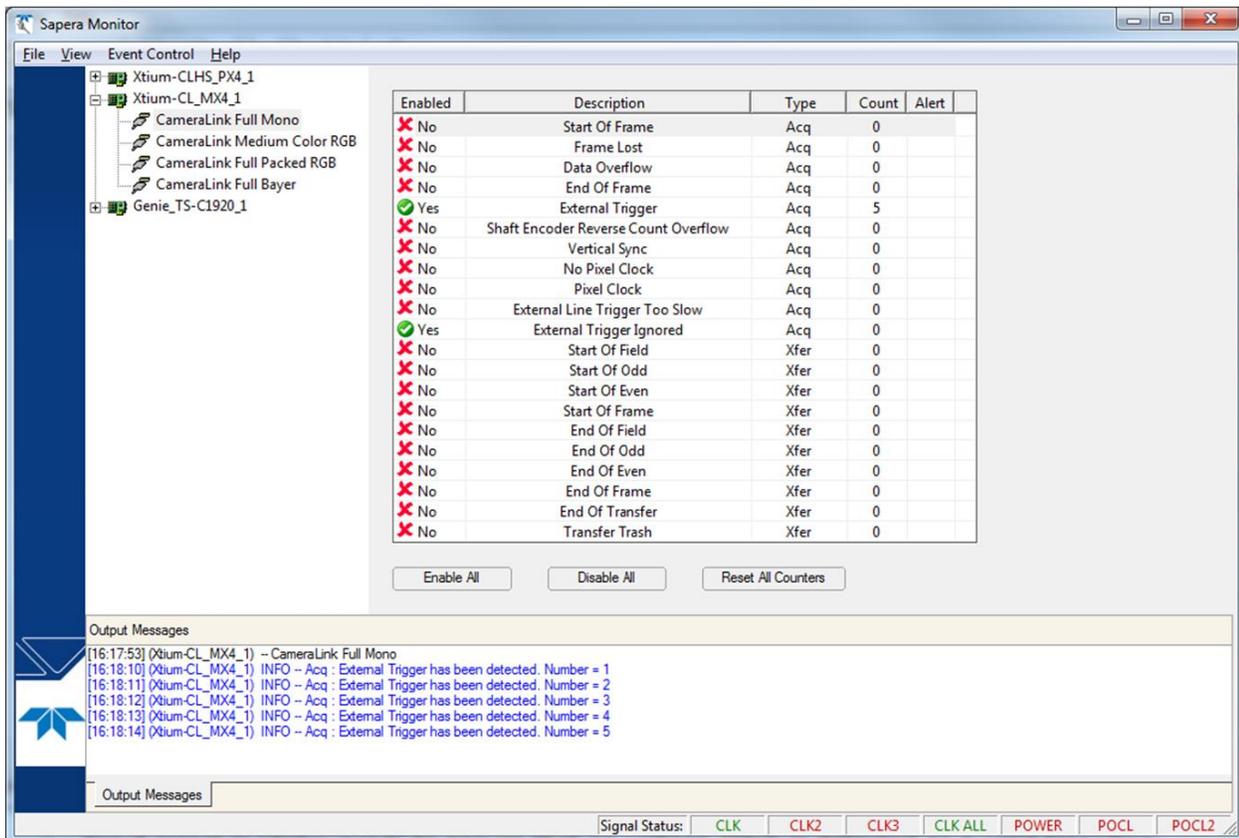
Advanced Diagnostics

Sapera LT's T2IR framework includes powerful GUI based tools for continuous monitoring and rapid pinpoint of errors that are hard to trace back. This continuous system monitoring and deep debugging tools help reduce downtime. This is done with the help of the following tools:

- Sapera Monitor
- External LEDs
- Sapera LogViewer
- Sapera PCI Diagnostic Tool
- Sapera Networking Tool
- Sapera Configuration
- Xtium Diagnostic Tool

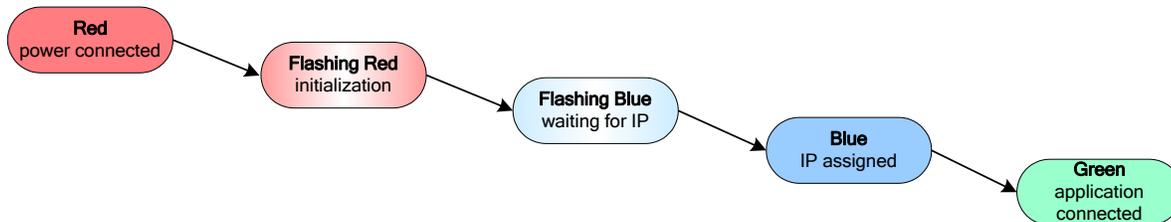
Sapera Monitor

The Sapera Monitor Tool allows users to view the acquisition and transfer events generated by an acquisition device in real-time. Sapera Monitor is a standalone application that is based on the Sapera LT T2IR functions. It allows users to see how their application is reacting to various events pertaining to the acquisition system and helps identify and debug problems without having to modify their application.



External LEDs

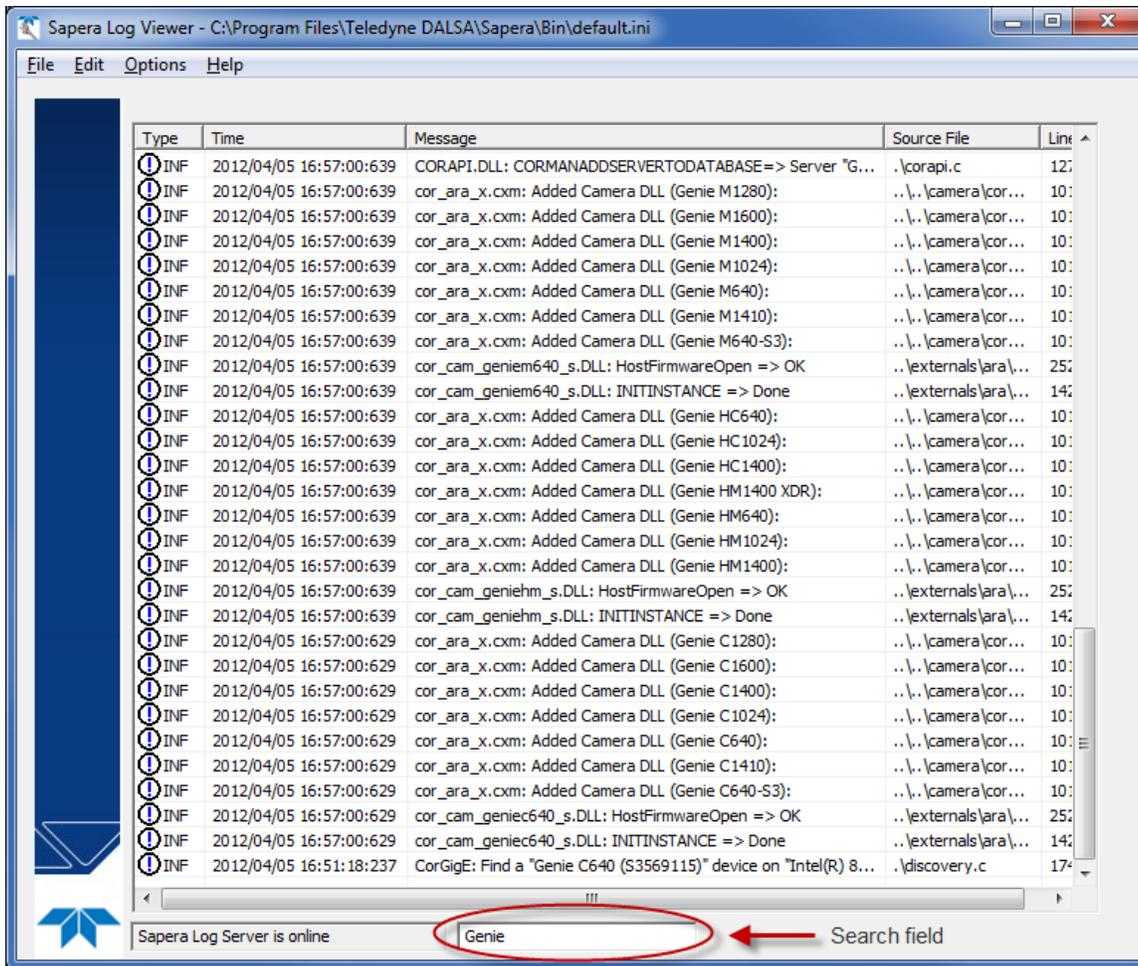
Visual indicators are indispensable features that permit continuous system monitoring right from the power up to full operations. External LEDs are available on the frame grabber bracket and camera back panel. Xtium series frame grabbers, for example, indicate crucial information during boot-up to indicate the board's detection status, PCIe version, lane configuration, and, during operation, presence of camera and acquisition status. For cameras, the status LED indicates boot-up and connection information. For example, the following LED sequence occurs when the Genie is powered up connected to a network with installed Genie Framework software.



Sapera LogViewer

The Teledyne DALSA Sapera Log Viewer utility bundled with Sapera LT installations provides an easy way to view the Sapera messages sent to the Teledyne DALSA acquisition devices and operating system. The Log Viewer provides critical insight into interactions between the host application and Sapera modules. Its detailed message listing offers crucial system wide information thus making it an indispensable tool to pinpoint hard to isolate, infrequent errors.

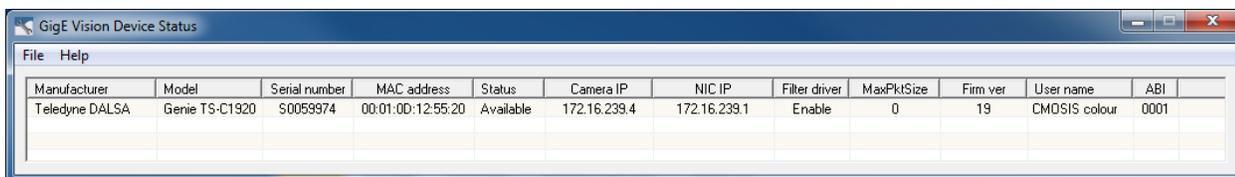
Sapera LogViewer runs transparently in the background without impacting the application performance and stores entire message communications and results. This allows analysis of the log even after the error has occurred. LogViewer configuration options allow users to set the type of results that are logged. For example, users can choose only to log "Error" messages and ignore "Warnings" or "Info" messages to conserve space. The resulting logs can be dynamically filtered and/or searched for key terms to pinpoint the messages resulting in errors, for example.



Furthermore, it is possible to run and customize multiple instances of the Log Viewer at the same time; therefore users, when dealing with multiple Teledyne DALSA acquisition devices, only view the messages of interest in each instance.

Sapera GigE Vision Device Status

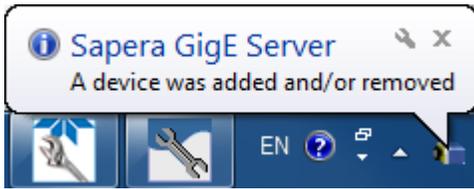
The GigE Vision Device Status application provides a quick method to view all the Teledyne DALSA GigE devices on your system.



It is available directly from the taskbar.

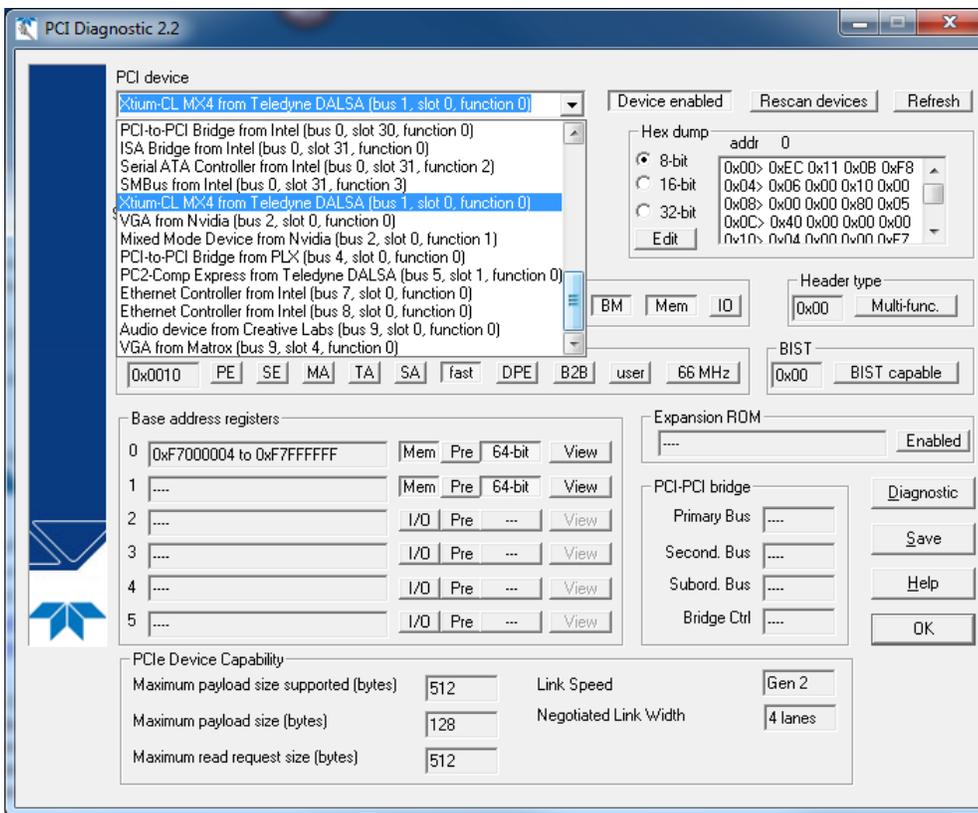


It continuously monitors the system and when a device is added or removed, a message box is displayed:



Sapera PCI Diagnostic Tool

The PCI Diagnostic tool allows you to view the low-level hardware resources allocated to the PCI devices on the system. For frame grabbers, you can quickly verify the device capabilities such as the bus and slot utilized, device ID, link speed (for example, Gen 1 or Gen 2) and payload size.



Xtium Frame Grabber Diagnostic Tool

The Xtium Board Diagnostic Tool provides a quick method to see board status and health of Xtium family of frame grabbers. Additionally, it provides live monitoring of FPGA temperature and voltages, which may help in identifying problems.

Diagnostic Tool Main Window

The main window provides a comprehensive view of the installed Xtium board. Toolbar buttons execute the board self-test function and open a FPGA live status window.

The screenshot shows the Xtium Diagnostic Tool interface. On the left, a blue sidebar contains several callout boxes: 'Generate Report', 'Save Report', 'Computer Slot Identification', 'FPGA Monitors', and 'Sapera & System Monitors'. The main window has a toolbar with 'Execute Self Test' and 'Live FPGA Monitors' buttons. The 'Channel Signal Integrity' callout points to the 'TEST' button. The 'PCI Slot Type' callout points to the 'Xtium-CLHS_PX8_1' dropdown menu. The 'Data Transfer Performance' callout points to the 'PCI Info' section of the table. The 'FPGA Monitors' callout points to the 'FPGA Temperature' section. The 'Sapera & System Monitors' callout points to the 'System Resource' and 'Sapera Memory' sections. A tooltip for 'Right Click: Refresh Rate & Reset Lanes Stats Menu.' is visible over the 'CLHS Lanes Stats' table.

Field/Value	Value	Max	Min
Driver Version	1.00.01.0113		
Serial Number	H0359003		
PCI Info			
Bus #	4		
Slot #	0		
Function #	0		
Bus Total Lanes	8		
Bus Bit Transfer Rate	Gen 2		
Bus Payload Size (bytes)	128		
Bus Request Size (bytes)	512		
PCIe Bandwidth (MB/s)			
Achieved Bandwidth	2169		
Maximum Theoretical	3500		
FPGA Temperature (°C)			
Current	52.29	52.29	52.29
Recommended		100.00	0.00
Voltage Aux (V)			
Current	1.76	1.76	1.76
Recommended		1.89	1.71
Voltage Int (V)			
Current	0.98	0.98	0.98
Recommended		1.03	0.97

CLHS Lanes Stats	CRC Error	Video MSG	Packet Buffer Overflow	Resend Flag	8b/10b Error
Lane 0	0	4836028	0	0	0
Lane 1	0	4836028	0	0	0
Lane 2	0	4836028	0	0	0
Lane 3	0	4836028	0	0	0
Lane 4	0	4836028	0	0	0
Lane 5	0	4836028	0	0	0
Lane 6	0	4836028	0	0	0

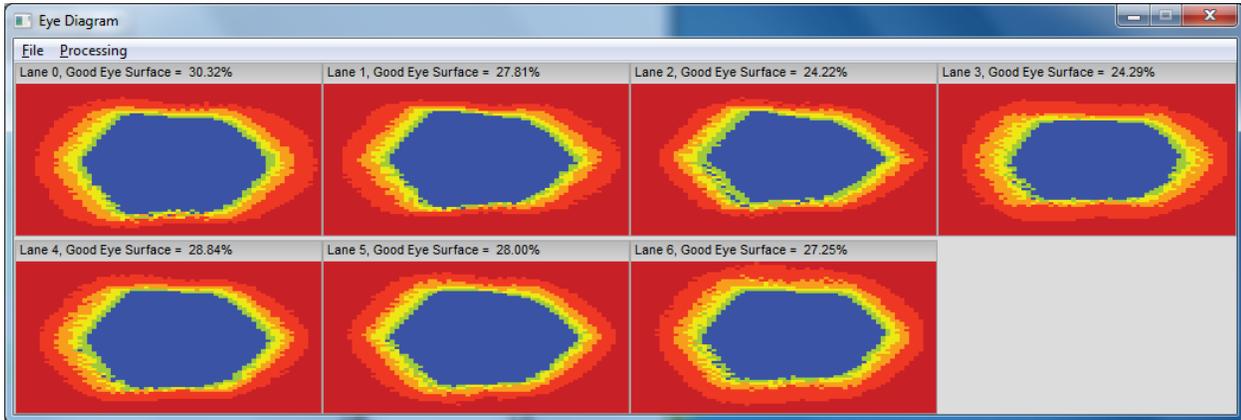
System Resource	Total (MB/KB)	Free (MB/KB)	Handles	Process	Thread
Physical Memory	16325/ 16717332	14011/ 14348240			
Page File	32649/ 33432824	30308/ 31035720			
Virtual Memory	8388607/ 8589934464	8388445/ 8589767712			
Total			18906	67	948

Sapera Memory	Free (KB/B)	Used (KB/B)	Free Blocks	Largest Free Block (KB/B)	Used Blocks	Largest Used Block (KB/B)
Message Memory	6144/ 6291456	0/ 0	1	6144/6291456	0	0/ 0
Buffer Memory	5120/ 5242880	0/ 0	1	5120/5242880	0	0/ 0

Important parameters include the PCI Express bus transfer supported by the host computer and the internal Xtium FPGA temperature. The bus transfer defines the maximum data rate possible in the computer, while an excessive FPGA temperature may explain erratic acquisitions due to poor computer ventilation.

Camera Input Eye Diagram Monitor

An Eye diagram is a graphical representation of signal between camera and frame grabber data lanes. This tool can be used to determine if the cable performance starts to degrade over a long period of use. The screen capture below shows a camera with 7 data lanes, where each digital signal is repetitively sampled and overlaid over itself, showing relative low-high transitions of the differential signal. Interpreting the results is easy, the bigger the blue area (eye surface) the better the signal integrity. When all the blue areas are similar in size and shape, it indicates that each wire pair has similar performance.



The closure (collapse or horizontal shortening) of the eye surface would indicate problems such as poor signal to noise, high cable capacitance, multipath interference, among many possible digital transmission faults.

Sapera LT API Overview

The Three Sapera LT APIs

Three different APIs are available under Sapera LT:

- Sapera LT ++ classes (based on C++ language)
- Sapera LT .NET classes (based on .NET languages)
- Sapera LT Standard API (based on C language)

The following sections demonstrate **Sapera LT ++** and **Sapera LT .NET**. For C API information consult the Sapera Basic Modules Reference Manual.

Sapera LT ++ – Creating an Application

See the Using Sapera LT ++ chapter of the **Sapera LT ++ Programmer's Manual** for a description of the steps needed for creating a Sapera LT ++ application.

Sapera LT .NET – Creating an Application

See the Using Sapera LT .NET chapter of the **Sapera LT .NET Programmer's Manual** for a description of the steps needed for creating a Sapera LT .NET application.

Sapera LT ++ – Object Initialization and Cleanup

Sapera LT ++ objects that encapsulate management of Standard API resources are initialized and refreshed in a uniform way, which consists of the following steps:

- Allocate memory for the object
- Create the resources needed by the object through the Create method
- Destroy the resources for the object through the Destroy method
- Release the memory for the object

Example with SapBuffer Class Objects

There is more than one way to do this, as shown next for SapBuffer class objects:

```
// The usual way to create the object is through a pointer
SapBuffer *pBuffer = new SapBuffer(1, 512, 512);

if (pBuffer->Create())
{
    // Buffer object is correctly initialized
}

// Destroy the buffer resources after checking if it is still initialized
// through the 'operator BOOL' for the SapBuffer class
if (*pBuffer)
{
    pBuffer->Destroy();
}

// Release the object memory
delete pBuffer;
pBuffer = NULL;
```

```

// Create the object on the stack
SapBuffer buffer(1, 512, 512);

if (buffer.Create())
{
    // Buffer object is correctly initialized
    // Destroy the buffer resources
    buffer.Destroy();
}

// The object memory is automatically released when it goes out of scope

```

```

// Create the object from an existing object
SapBuffer buffer(1, 512, 512);
SapBuffer *pBuffer = new SapBuffer(buffer);

if (pBuffer->Create())
{
    pBuffer->Destroy();
}

// Release the object memory
delete pBuffer;
pBuffer = NULL;

```

Sapera LT ++ objects that do not encapsulate management of Standard API resources are correctly initialized as soon as their constructor has been called.

```

SapDataMono data(123);

// The object memory is automatically released when it goes out of scope

```

Sapera LT .NET – Object Initialization and Cleanup

Sapera LT .NET objects are initialized and cleaned up in a uniform way, which is described by the following steps:

- Allocate memory for the object
- Create the resources needed by the object through the Create method
- Destroy the resources for the object through the Destroy method
- Release unmanaged memory used internally through the Dispose method

Example with SapBuffer Class Objects in C#:

```

SapBuffer buffer = new SapBuffer(1, 512, 512, SapFormat.Mono8,
    SapBuffer.MemoryType.ScatterGather);

if (buffer.Create())
{
    // Buffer object is correctly initialized
}

// Destroy the buffer resources after checking if it is still initialized
if (buffer.Initialized)
{
    buffer.Destroy();
}

// Release unmanaged memory used internally
buffer.Dispose();

```

Equivalent Code for Visual Basic .NET:

```
Dim buffer As SapBuffer = New SapBuffer(1, 512, 512, SapFormat.Mono8, _
    SapBuffer.MemoryType.ScatterGather)

If buffer.Create() Then
    ' Buffer object is correctly initialized
End If

' Destroy the buffer resources after checking if it is still initialized
If buffer.Initialized Then
    buffer.Destroy()
End If

' Release unmanaged memory used internally
buffer.Dispose()
```

Equivalent Code for C++:

```
SapBuffer^ pBuffer = gcnew SapBuffer(1, 512, 512, SapFormat::Mono8,
    SapBuffer::MemoryType::ScatterGather);

if (pBuffer->Create())
{
    // Buffer object is correctly initialized
}

// Destroy the buffer resources after checking if it is still initialized
if (pBuffer->Initialized)
{
    pBuffer->Destroy();
}

// Release unmanaged memory used internally
// Note that the delete operator actually calls the Dispose method
delete pBuffer;
pBuffer = nullptr;
```

Sapera LT ++ – Error Management

Most Sapera LT ++ methods return a Boolean TRUE/FALSE result to indicate success or failure. However, the actual errors conditions are still reported as soon as they happen using one of five predefined reporting modes:

- Error messages are sent to a popup window (the default)
- Error messages are sent to the Sapera Log Server (can be displayed using the Sapera Log Viewer)
- Error messages are sent to the active debugger if any
- Error messages are generated internally
- Error messages are sent to the application through a callback function

Setting the Current Reporting Mode

Use the `SapManager::SetDisplayStatusMode` method to set the current reporting mode, as follows:

```
// Send error messages to the Sapera Log Server
SapManager::SetDisplayStatusMode(SapManager::StatusLog);

// Send error messages to the debugger
SapManager::SetDisplayStatusMode(SapManager::StatusDebug);

// Simply generate error messages
SapManager::SetDisplayStatusMode(SapManager::StatusCustom);

// Send errors to application using a callback function
SapManager::SetDisplayStatusMode(SapManager::StatusCallback);

// Restore default reporting mode
SapManager::SetDisplayStatusMode(SapManager::StatusNotify);
```

Monitoring Errors

No matter which reporting mode is currently active, it is always possible to retrieve the latest error message. If the error happened when Sapera LT ++ called a Standard API function, then a related numeric code is also available. In order to retrieve this information, call the `SapManager::GetLastStatus` method as follows:

```
// Get the latest error message
char errorDescr[256];
strcpy(errorDescr, SapManager::GetLastStatus());

// Get the latest error code
// See the Sapera Basic Modules Reference Manual for details
SAPSTATUS lastError;
SapManager::GetLastStatus(&lastError);
```

In addition, the Sapera Log Viewer utility program, included with Sapera LT, provides an easy way to view error messages. It includes a list box that stores these messages as soon as the errors happen. Available options allow you to modify the different fields for display.

During development it is recommended to start the Log Viewer before your application and then let it run so it can be referred to any time a detailed error description is required. However, errors are actually stored by the Sapera Log Server (running in the background), even if the utility is not running. Therefore it is possible to start the Log Viewer only when a problem occurs with your application.

Sapera LT .NET – Error Management

Most Sapera LT .NET methods return a boolean result to indicate success or failure. However, the actual errors conditions are still reported as soon as they happen using one of five predefined reporting modes:

- Error messages are sent to a popup window (the default)
- Error messages are sent to the Sapera Log Server (can be displayed using the Sapera Log Viewer)
- Error messages are sent to the application through an event
- Error messages are sent to the application through an exception
- Error messages are generated internally, but not reported immediately

Setting the Current Reporting Mode

Use the `DisplayStatusMode` property of the `SapManager` class to set the current reporting mode as follows:

Example of Error Management with C#:

```
// Send error messages to the Sapera Log Server
SapManager.DisplayStatusMode = SapManager.StatusMode.Log;

// Send errors to application through the Error event
SapManager.DisplayStatusMode = SapManager.StatusMode.Event;
SapManager.Error += new SapErrorHandler(SapManager_Error);

SapManager.Error -= new SapErrorHandler(SapManager_Error);

// Send errors to application through an exception
SapManager.DisplayStatusMode = SapManager.StatusMode.Exception;

// try
{
    // Code that possibly generates an error
}
catch (SapLibraryException exception)
{
    // Exception handling code
}

// Just generate error messages
SapManager.DisplayStatusMode = SapManager.StatusMode.Custom;

// Restore default reporting mode
SapManager.DisplayStatusMode = SapManager.StatusMode.Popup;
```

Equivalent Code for Visual Basic .NET:

```
' Send error messages to the Log Viewer
SapManager.DisplayStatusMode = SapManager.StatusMode.Log

' Send errors to application through the Error event
SapManager.DisplayStatusMode = SapManager.StatusMode.Event
AddHandler SapManager.Error, AddressOf SapManager_Error

RemoveHandler SapManager.Error, AddressOf SapManager_Error

' Send errors to application through an exception
SapManager.DisplayStatusMode = SapManager.StatusMode.Exception

Try
    ' Code that possibly generates an error
Catch exception As SapLibraryException
    ' Exception handling code
End Try

' Just generate error messages
SapManager.DisplayStatusMode = SapManager.StatusMode.Custom

' Restore default reporting mode
SapManager.DisplayStatusMode = SapManager.StatusMode.Popup
```

Equivalent Code for C++:

```
// Send error messages to the Log Viewer
SapManager::DisplayStatusMode = SapManager::StatusMode::Log;

// Send errors to application through the Error event
SapManager::DisplayStatusMode = SapManager::StatusMode::Event;
SapManager::Error += gcnew SapErrorHandler(SapManager_Error);

SapManager::Error -= gcnew SapErrorHandler(SapManager_Error);

// Send errors to application through an exception
SapManager::DisplayStatusMode = SapManager::StatusMode::Exception;

// try
{
    // Code that possibly generates an error
}
catch (SapLibraryException^ exception)
{
    // Exception handling code
}

// Just generate error messages
SapManager::DisplayStatusMode = SapManager::StatusMode::Custom;

// Restore default reporting mode
SapManager::DisplayStatusMode = SapManager::StatusMode::Popup;
```

Event Handling Method Definition for C#:

```
public static void SapManager_Error(Object sender, SapErrorEventArgs args)
{
    // Code to handle the Error event of the SapManager class
}
```

Equivalent Code for Visual Basic .NET:

```
Sub SapManager_Error(ByVal sender As Object, ByVal args As SapErrorEventArgs)
    ' Code to handle the Error event of the SapManager class
End Sub
```

Equivalent Code for C++:

```
static void SapManager_Error(Object^ sender, SapErrorEventArgs^ args)
{
    // Code to handle the Error event of the SapManager class
}
```

Monitoring Errors

No matter which reporting mode is currently active, it is always possible to retrieve the latest error message. If the error happened when Sapera LT .NET called a Standard API function, then a related numeric code is also available. In order to retrieve this information use the `LastStatusMessage` and `LastStatusCode` properties of the `SapManager` class.

Example to Monitor Errors in C#:

```
// Get the latest error message
string lastMessage = SapManager.LastStatusMessage;

// Get the latest error code
// See the Sapera Basic Modules Reference Manual for details
SapStatus lastCode = SapManager.LastStatusCode;
```

Equivalent Code for Visual Basic .NET:

```
' Get the latest error message
Dim lastMessage As String = SapManager.LastStatusMessage

' Get the latest error code
' See the Sapera Basic Modules Reference Manual for details
Dim lastCode As SapStatus = SapManager.LastStatusCode
```

Equivalent Code for C++:

```
// Get the latest error message
String^ lastMessage = SapManager::LastStatusMessage;

// Get the latest error code
// See the Sapera Basic Modules Reference Manual for details
SapStatus lastCode = SapManager::LastStatusCode;
```

In addition, the Sapera Log Viewer utility program included with Sapera LT provides an easy way to view error messages. It includes a list box that stores these messages as soon as the errors happen. Available options allow you to modify the different fields for display.

During development it is recommended to start the Log Viewer before your application and then let it run so it can be referred to any time a detailed error description is required. However, errors are actually stored by the Sapera Log Server (running in the background), even if the utility is not running. Therefore it is possible to start the Log Viewer only when a problem occurs with your application.

Capabilities and Parameters

Sapera LT ++ and Sapera LT .NET already include all the functionality necessary for most Sapera LT applications. However, some features are only available in the Standard API such as the devices's capabilities and parameters. Together these API define a resource device's ability and current state.

See the *Sapera Basic Modules Reference Manual* for a description of all capabilities and parameters, and their possible values.

What is a Capability?

A capability as its name implies, is a value or set of values that describe what a resource can do. Capabilities are used to determine the possible valid values that can be applied to a resource's parameters. They are read-only.

A capability can be obtained from a resource by using the `GetCapability` method in the corresponding class. See the *Sapera LT ++ Programmer's Manual* or the *Sapera LT .NET Programmer's Manual* for details.

What is a Parameter?

A parameter describes a current characteristic of a resource. It can be read/write or read-only.

A parameter for a resource can be obtained or set by using the `GetParameter` and `SetParameter` methods in the corresponding class. See the *Sapera LT ++ Programmer's Manual* or the *Sapera LT .NET Programmer's Manual* for details.

Acquiring Images

Required Classes

You need three Sapera LT ++ or Sapera LT .NET classes to initiate the acquisition process:

- **SapAcquisition or SapAcqDevice:** Use the SapAcquisition class if you are using a frame grabber. Use the SapAcqDevice class if you are using a camera directly connected to your PC, such as a Teledyne DALSA Genie camera.
- **SapBuffer:** Used to store the acquired data. Should be created using the ScatterGather (preferable) or Contiguous buffer type to enable the transfer. See the "Working with Buffers" section for more information about contiguous memory and scatter-gather.
- **SapTransfer:** Used to link the acquisition device to the buffer and to synchronize the acquisition operations.

Frame-Grabber Acquisition – Required Steps

- Specify the acquisition device and corresponding camera configuration file using the SapAcquisition class.
- Create a buffer in memory to store the acquired image using the SapBuffer class.
- Allocate a view object to display the image using the SapView class, if required.
- **For Sapera LT ++:** Allocate a transfer object to link the acquisition to the image buffer using the SapTransfer class. A transfer callback function should be registered if images need to be processed and displayed while grabbing.
- **For Sapera LT .NET:** Allocate a transfer object to link the acquisition to the image buffer using the SapTransfer class. Use a transfer event method if images need to be processed and/or displayed while grabbing.
- Allocate the resources for all objects (acquisition, view, buffer, and transfer), using the respective Create function of the class used to create the objects.
- Grab images, using the SapTransfer class.
- Destroy all created resources when grabbing is completed.

Sapera LT ++ – Sample Acquisition Code

This sample code demonstrates how to grab a live image into a buffer allocated in system memory using the X64-CL board as an acquisition device. See section *Supported Systems & Devices of the Sapera LT User's Introduction* manual for a list of Teledyne DALSA boards equipped with an acquisition section.

Acquiring an image requires one file (the CCF file) to configure the acquisition hardware. It defines both the characteristics of the camera and how it will be used with the acquisition hardware. Refer to *Using the CamExpert Tool in the Sapera LT User's Introduction* manual for information on generating this file. Resource parameters can also be accessed individually.

After the acquisition module is initialized using the CCF file, a compatible buffer can be created using settings taken directly from the acquisition.

Before starting the actual transfer, you must create a transfer object to link the acquisition and the buffer objects. Furthermore when stopping a transfer, you must call the SapTransfer::Wait method to wait for the transfer process to terminate.

Example Program using Sapera LT ++

```
// Transfer callback function is called each time a complete frame is transferred.
// The function below is a user defined callback function.

void XferCallback(SapXferCallbackInfo *pInfo)
{
    // Display the last transferred frame
    SapView *pView = (SapView *) pInfo->GetContext();
    pView->Show();
}
// Example program
//
main()
{
    // Allocate acquisition object
    SapAcquisition *pAcq =
        new SapAcquisition(SapLocation ("X64-CL_1", 0), "MyCamera.ccf");

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

    // Allocate view object, images will be displayed directly on the desktop
    SapView *pView = new SapView(pBuffer, SapHwndDesktop);

    // Allocate transfer object to link acquisition and buffer
    SapTransfer *pTransfer = new SapTransfer(XferCallback, pView);
    pTransfer->AddPair(SapXferPair(pAcq, pBuffer));

    // Create resources for all objects
    BOOL success = pAcq->Create();
    success = pBuffer->Create();
    success = pView->Create();
    success = pTransfer->Create();

    // Start a continuous transfer (live grab)
    success = pTransfer->Grab();
    printf("Press any key to stop grab\n");
    getch();

    // Stop the transfer and wait (timeout = 5 seconds)
    success = pTransfer->Freeze();
    success = pTransfer->Wait(5000);
    printf("Press any key to terminate\n");
    getch();

    // Release resources for all objects
    success = pTransfer->Destroy();
    success = pView->Destroy();
    success = pBuffer->Destroy();
    success = pAcq->Destroy();

    // Free all objects
    delete pTransfer;
    delete pView;
    delete pBuffer;
    delete pAcq;

    return 0;
}
```

For more details, see the *Sapera LT ++ Programmer's Manual* and the source code for the demos and examples included with Sapera LT.

Sapera LT .NET – Sample Acquisition Code

The following sample code demonstrates grabbing a live image into a buffer allocated in system memory using the X64 Xcelera-CL PX4 board as the acquisition device.

Acquiring an image requires one file (the CCF file) to configure the acquisition hardware. It defines both the characteristics of the camera and how it will be used with the acquisition hardware. Refer to *Using the CamExpert Tool in the Sapera LT User's Introduction* manual for information on generating this file. Resource parameters can also be accessed individually.

After the acquisition module is initialized using the CCF file, a compatible buffer can be created using settings taken directly from the acquisition.

Before initiating the actual transfer you must create a transfer object to link the acquisition and the buffer objects. Furthermore when stopping a transfer, you must call the Wait method in the SapTransfer class to wait for the transfer process to terminate.

Example Program using C#

```
// Transfer event handler is called each time a complete frame is transferred
static void SapTransfer_XferNotify(object sender, SapXferNotifyEventArgs args)
{
    SapView view = args.Context as SapView;
    view.Show();
}

static void Main(string[] args)
{
    // Allocate acquisition object
    SapAcquisition acq = new SapAcquisition(
        new SapLocation("Xcelera-CL_PX4_1", 0), "MyCamera.ccf");

    // Allocate buffer object, taking settings directly from the acquisition
    SapBuffer buffer = new SapBuffer(1, acq, SapBuffer.MemoryType.ScatterGather);

    // Allocate view object, images will be displayed directly on the desktop
    SapView view = new SapView(buffer);

    // Allocate transfer object to link acquisition and buffer
    SapTransfer transfer = new SapTransfer();
    transfer.AddPair(new SapXferPair(acq, buffer));

    transfer.Pairs[0].EventType = SapXferPair.XferEventType.EndOfFrame;
    transfer.XferNotify += new SapXferNotifyHandler(SapTransfer_XferNotify);
    transfer.XferNotifyContext = view;

    // Create resources for all objects
    bool success = acq.Create();
    success = buffer.Create();
    success = view.Create();
    success = transfer.Create();

    // Start a continuous transfer (live grab)
    success = transfer.Grab();
    Console.WriteLine("Press any key to stop grab");
    Console.ReadKey(true);

    // Stop the transfer and wait (timeout = 5 seconds)
    success = transfer.Freeze();
    success = transfer.Wait(5000);
    Console.WriteLine("Press any key to terminate");
    Console.ReadKey(true);

    // Release resources for all objects
    success = transfer.Destroy();
    success = view.Destroy();
    success = buffer.Destroy();
    success = acq.Destroy();

    // Free all objects
    transfer.Dispose();
}
```

```

view.Dispose();
buffer.Dispose();
acq.Dispose();
}

```

Equivalent Program using Visual Basic .NET

```

' Transfer event handler is called each time a complete frame is transferred
Sub SapTransfer_XferNotify(ByVal sender As Object, _
    ByVal args As SapXferNotifyEventArgs)
    Dim view As SapView = args.Context
    view.Show()
End Sub

Sub Main()
    ' Allocate acquisition object
    Dim acq As SapAcquisition = New SapAcquisition( _
        New SapLocation("Xcelera-CL_PX4_1", 0), "MyCamera.ccf")

    ' Allocate buffer object, taking settings directly from the acquisition
    Dim buffer As SapBuffer = New SapBuffer(1, acq, SapBuffer.MemoryType.ScatterGather)

    ' Allocate view object, images will be displayed directly on the desktop
    Dim view As SapView = New SapView(buffer)

    ' Allocate transfer object to link acquisition and buffer
    Dim transfer As SapTransfer = New SapTransfer()
    transfer.AddPair(New SapXferPair(acq, buffer))

    transfer.Pairs(0).EventType = SapXferPair.XferEventType.EndOfFrame
    AddHandler transfer.XferNotify, AddressOf SapTransfer_XferNotify
    transfer.XferNotifyContext = view

    ' Create resources for all objects
    Dim success As Boolean = acq.Create()
    success = buffer.Create()
    success = view.Create()
    success = transfer.Create()

    ' Start a continuous transfer (live grab)
    success = transfer.Grab()
    Console.WriteLine("Press any key to stop grab")
    Console.ReadKey(True)

    ' Stop the transfer and wait (timeout = 5 seconds)
    success = transfer.Freeze()
    success = transfer.Wait(5000)
    Console.WriteLine("Press any key to terminate")
    Console.ReadKey(True)

    ' Release resources for all objects
    success = transfer.Destroy()
    success = view.Destroy()
    success = buffer.Destroy()
    success = acq.Destroy()

    ' Free all objects
    transfer.Dispose()
    view.Dispose()
    buffer.Dispose()
    acq.Dispose()
End Sub

```

Equivalent Example using C++

```
// Transfer event handler is called each time a complete frame is transferred
static void SapTransfer_XferNotify(Object^ sender, SapXferNotifyEventArgs^ args)
{
    SapView^ pView = safe_cast<SapView^>(args->Context);
    pView->Show();
}
int main(array<String ^>^ args)
{
    // Allocate acquisition object
    SapAcquisition^ pAcq = gcnew SapAcquisition(
        gcnew SapLocation("Xcelera-CL_PX4_1", 0), "MyCamera.ccf");

    // Allocate buffer object, taking settings directly from the acquisition
    SapBuffer^ pBuffer = gcnew SapBuffer(1, pAcq,
        SapBuffer::MemoryType::ScatterGather);

    // Allocate view object, images will be displayed directly on the desktop
    SapView^ pView = gcnew SapView(pBuffer);

    // Allocate transfer object to link acquisition and buffer
    SapTransfer^ pTransfer = gcnew SapTransfer();
    pTransfer->AddPair(gcnew SapXferPair(pAcq, pBuffer));

    pTransfer->Pairs[0]->EventType = SapXferPair::XferEventType::EndOfFrame;
    pTransfer->XferNotify += gcnew SapXferNotifyHandler(SapTransfer_XferNotify);
    pTransfer->XferNotifyContext = pView;

    // Create resources for all objects
    bool success = pAcq->Create();
    success = pBuffer->Create();
    success = pView->Create();
    success = pTransfer->Create();

    // Start a continuous transfer (live grab)
    success = pTransfer->Grab();
    Console::WriteLine("Press any key to stop grab");
    Console::ReadKey(true);

    // Stop the transfer and wait (timeout = 5 seconds)
    success = pTransfer->Freeze();
    success = pTransfer->Wait(5000);
    Console::WriteLine("Press any key to terminate");
    Console::ReadKey(true);

    // Release resources for all objects
    success = pTransfer->Destroy();
    success = pView->Destroy();
    success = pBuffer->Destroy();
    success = pAcq->Destroy();

    // Free all objects
    // Note that the delete operator actually calls the Dispose method
    delete pTransfer;
    delete pView;
    delete pBuffer;
    delete pAcq;
    return 0;
}
```

For detailed information see the source code for the Sopera LT .NET demos and examples included with Sopera LT.

Sapera LT ++ – Modifying Frame-Grabber Parameters

The code samples below are examples for individual and group parameter changes, including a triggered acquisition sample. For more details, see the *Sapera LT ++ Programmer's Manual* and the *Sapera Basic Modules Reference Manual*.

Modifying Parameters Individually

Acquisition parameters can be modified individually by using the `SapAcquisition::SetParameter` method. When a new parameter value is requested that value is verified against the current state of the acquisition module and the acquisition module capabilities. If the modification request is denied because the parameter is dependent on other parameters, then all the parameters in question must be modified by group.

```
// Allocate and create resources for acquisition object
SapAcquisition *pAcq =
    new SapAcquisition(SapLocation("X64-CL_1", 0), "MyCamera.ccf");
BOOL success = pAcq->Create();

// Try changing the sync source to Composite Sync
success = pAcq->SetParameter(CORACQ_PRM_SYNC, CORACQ_VAL_SYNC_COMP_SYNC);

// Release resources for acquisition object, and free it
success = pAcq->Destroy();
delete pAcq;
```

Triggered Acquisition Example

The following code sample demonstrates how to set individual parameters, using the `SapAcquisition` class `SetParameter` function, to perform a triggered acquisition (area scan camera). These parameters are set after creating your acquisition object and resources, and before grabbing.

```
pAcq->SetParameter(CORACQ_PRM_EXT_TRIGGER_LEVEL, CORACQ_VAL_LEVEL_TTL);
pAcq->SetParameter(CORACQ_PRM_EXT_TRIGGER_ENABLE, CORACQ_VAL_EXT_TRIGGER_ON);
pAcq->SetParameter(CORACQ_PRM_EXT_TRIGGER_DETECTION, CORACQ_VAL_RISING_EDGE);
```

Modifying Parameters by Group

Acquisition parameters can be modified by groups using the optional *updateNow* parameter to the `SapAcquisition::SetParameter` method. When a new set of values is written, all modified parameters are verified against the given state and capabilities of the acquisition object.

```
// Allocate and create resources for acquisition object
SapAcquisition *pAcq =
    new SapAcquisition(SapLocation("X64-CL_1", 0), "MyCamera.ccf");
BOOL success = pAcq->Create();

// Try changing the cropping and scaling parameters
success = pAcq->SetParameter(CORACQ_PRM_CROP_WIDTH, 640, FALSE);
success = pAcq->SetParameter(CORACQ_PRM_CROP_HEIGHT, 480, FALSE);
success = pAcq->SetParameter(CORACQ_PRM_SCALE_HORZ, 640, FALSE);
success = pAcq->SetParameter(CORACQ_PRM_SCALE_VERT, 480, TRUE);

// Release resources for acquisition object, and free it
success = pAcq->Destroy();
delete pAcq;
```

Sapera LT .NET – Modifying Frame-Grabber Parameters

The code samples below are examples for individual and group parameter changes, including a triggered acquisition sample. These are presented for C#, Visual Basic .NET, and C++.

Modifying Parameters Individually

Acquisition parameters can be modified individually by using the SetParameter method of SapAcquisition. When a new parameter value is requested that value is verified against the current state of the acquisition module and the acquisition module capabilities. If the modification request is denied because the parameter is dependent on other parameters, then all the parameters in question must be modified by group.

Sample Code for C#

```
// Allocate and create resources for acquisition object
SapAcquisition acq =
    new SapAcquisition(new SapLocation("Xcelera-CL_PX4_1", 0), "MyCamera.ccf");
bool success = acq.Create();

// Change the sync source to Composite Sync
success = acq.SetParameter(SapAcquisition.Prm.SYNC,
    SapAcquisition.Val.SYNC_COMP_SYNC, true);

// Release resources for acquisition object, and free it
success = acq.Destroy();
acq.Dispose();
```

Equivalent Code for Visual Basic .NET

```
' Allocate and create resources for acquisition object
Dim acq As SapAcquisition = _
    New SapAcquisition(New SapLocation("Xcelera-CL_PX4_1", 0), "MyCamera.ccf")
Dim success As Boolean = acq.Create()

' Change the sync source to Composite Sync
success = acq.SetParameter(SapAcquisition.Prm.SYNC, _
    SapAcquisition.Val.SYNC_COMP_SYNC, True)

' Release resources for acquisition object, and free it
success = acq.Destroy()
acq.Dispose()
```

Equivalent Code for C++

```
// Allocate and create resources for acquisition object
SapAcquisition^ pAcq =
    gcnew SapAcquisition(gcnew SapLocation("Xcelera-CL_PX4_1", 0), "MyCamera.ccf");
bool success = pAcq->Create();

// Change the sync source to Composite Sync
success = pAcq->SetParameter(SapAcquisition::Prm::SYNC,
    SapAcquisition::Val::SYNC_COMP_SYNC, true);

// Release resources for acquisition object, and free it
// Note that the delete operator actually calls the Dispose method
success = pAcq->Destroy();
delete pAcq;
```

For more details, see the *Sapera Basic Modules Reference Manual*.

Triggered Acquisition Example

The following shows how to set individual parameters using the `SetParameter` method of `SapAcquisition`, to perform a triggered acquisition (area scan camera). These parameters are set after creating the `SapAcquisition` object and before grabbing.

Sample code for C#

```
success = acq.SetParameter(SapAcquisition.Prm.EXT_TRIGGER_LEVEL,
    SapAcquisition.Val.LEVEL_TTL, true);
success = acq.SetParameter(SapAcquisition.Prm.EXT_TRIGGER_ENABLE,
    SapAcquisition.Val.EXT_TRIGGER_ON, true);
success = acq.SetParameter(SapAcquisition.Prm.EXT_TRIGGER_DETECTION,
    SapAcquisition.Val.RISING_EDGE, true);
```

Equivalent Code for Visual Basic .NET

```
success = acq.SetParameter(SapAcquisition.Prm.EXT_TRIGGER_LEVEL, _
    SapAcquisition.Val.LEVEL_TTL, True)
success = acq.SetParameter(SapAcquisition.Prm.EXT_TRIGGER_ENABLE, _
    SapAcquisition.Val.EXT_TRIGGER_ON, True)
success = acq.SetParameter(SapAcquisition.Prm.EXT_TRIGGER_DETECTION, _
    SapAcquisition.Val.RISING_EDGE, True)
```

Equivalent Code for C++

```
success = pAcq->SetParameter(SapAcquisition::Prm::EXT_TRIGGER_LEVEL,
    SapAcquisition::Val::LEVEL_TTL, true);
success = pAcq->SetParameter(SapAcquisition::Prm::EXT_TRIGGER_ENABLE,
    SapAcquisition::Val::EXT_TRIGGER_ON, true);
success = pAcq->SetParameter(SapAcquisition::Prm::EXT_TRIGGER_DETECTION,
    SapAcquisition::Val::RISING_EDGE, true);
```

Modifying Parameters by Group

Acquisition parameters can be modified by groups using the `updateNow` argument of the `SetParameter` method of `SapAcquisition`. When a new set of values is written all modified parameters are verified against the given state and capabilities of the `SapAcquisition` object.

Sample Code for C#

```
// Change the cropping and scaling parameters
success = acq.SetParameter(SapAcquisition.Prm.CROP_WIDTH, 640, false);
success = acq.SetParameter(SapAcquisition.Prm.CROP_HEIGHT, 480, false);
success = acq.SetParameter(SapAcquisition.Prm.SCALE_HORZ, 640, false);
success = acq.SetParameter(SapAcquisition.Prm.SCALE_VERT, 480, true);
```

Equivalent Code for Visual Basic .NET

```
' Change the cropping and scaling parameters
success = acq.SetParameter(SapAcquisition.Prm.CROP_WIDTH, 640, False)
success = acq.SetParameter(SapAcquisition.Prm.CROP_HEIGHT, 480, False)
success = acq.SetParameter(SapAcquisition.Prm.SCALE_HORZ, 640, False)
success = acq.SetParameter(SapAcquisition.Prm.SCALE_VERT, 480, True)
```

Equivalent Code for C++

```
// Change the cropping and scaling parameters
success = pAcq->SetParameter(SapAcquisition::Prm::CROP_WIDTH, 640, false);
success = pAcq->SetParameter(SapAcquisition::Prm::CROP_HEIGHT, 480, false);
success = pAcq->SetParameter(SapAcquisition::Prm::SCALE_HORZ, 640, false);
success = pAcq->SetParameter(SapAcquisition::Prm::SCALE_VERT, 480, true);
```

For more details, see the *Sapera Basic Modules Reference Manual*.

Sapera LT ++ – Using an Input Lookup Table

When you call the Create method for a SapAcquisition object an internal lookup table object (SapLut) is automatically created inside the object, if the acquisition hardware supports lookup tables.

You may then retrieve it using the SapAcquisition::GetLut method, manipulate it using the methods in the SapLut Class, and reprogram it using the SapAcquisition::ApplyLut method.

The internal SapLut object is automatically destroyed when you call the SapAcquisition::Destroy method. The following code is an example of these steps.

Sample Code

```
// Allocate and create resources for acquisition object
SapAcquisition pAcq =
    new SapAcquisition(SapLocation("X64-CL_1", 0), "MyCamera.ccf");
BOOL success = pAcq->Create();

// Try changing the acquisition lookup table using a custom mapping
// The GetLut method returns NULL if there is no acquisition LUT
SapLut pLut = pAcq->GetLut();

// Allocate working memory for the new lookup table data
char *pNewData = new char[pLut->GetTotalSize()];
// Fill contents of data buffer ...

// Write new data to lookup table
pLut->Write(0, pNewData, pLut->GetTotalSize());

// Program the acquisition hardware with the new LUT data
pAcq->ApplyLut();

// Free working memory
delete [] pNewData;

// Release resources for acquisition object, and free it
success = pAcq->Destroy();
delete pAcq;
```

Sapera LT .NET – Using an Input Lookup Table

When you call the Create method for a SapAcquisition object an internal lookup table object (SapLut) is automatically created inside the object, if the acquisition hardware supports lookup tables.

You may then retrieve it using the Lut property of SapAcquisition, manipulate it using the methods in SapLut, and reprogram it using the ApplyLut method of SapAcquisition.

The internal SapLut object is automatically destroyed when you call the Destroy method of SapAcquisition.

Sample Code for C#

```
// Allocate and create resources for acquisition object
SapAcquisition acq =
    new SapAcquisition(new SapLocation("Xcelera-CL_PX4_1", 0), "MyCamera.ccf");
bool success = acq.Create();

// Change the first acquisition lookup table using a custom mapping.
// The Luts property returns null if there is no acquisition LUT.
SapLut lut = acq.Luts[0];

// C# needs an unsafe block to work with pointers
unsafe
{
    // Allocate working memory for the new lookup table data
    byte[] newData = new byte[lut.TotalSize];

    fixed (byte* pNewData = newData)
    {
        // Fill contents of data buffer ...

        // Write new data to lookup table
        success = lut.Write(0, pNewData, lut.TotalSize);
    }
}

// Program the acquisition hardware with the new LUT data
success = acq.ApplyLut(true, 0);

// Release resources for acquisition object, and free it
success = acq.Destroy();
acq.Dispose();
```

Equivalent Code for Visual Basic .NET

```
' Allocate and create resources for acquisition object
Dim acq As SapAcquisition = _
    New SapAcquisition(New SapLocation("Xcelera-CL_PX4_1", 0), "MyCamera.ccf")
Dim success As Boolean = acq.Create()

' Change the first acquisition lookup table using a custom mapping.
' The Luts property returns Nothing if there is no acquisition LUT.
Dim lut As SapLut = acq.Luts(0)

' Visual Basic cannot use methods with pointer arguments like the Write
' method of SapLut. Use one of the predefined LUT mappings instead.
success = lut.Reverse()

' Program the acquisition hardware with the new LUT data
success = acq.ApplyLut(True, 0)

' Release resources for acquisition object, and free it
success = acq.Destroy()
acq.Dispose()
```

Equivalent Code for C++

```
// Allocate and create resources for acquisition object
SapAcquisition^ pAcq = gcnew SapAcquisition(
    gcnew SapLocation("Xcelera-CL_PX4_1", 0), "MyCamera.ccf");
bool success = pAcq->Create();

// Change the first acquisition lookup table using a custom mapping.
// The Lut property returns nullptr if there is no acquisition LUT.
SapLut^ pLut = pAcq->Luts[0];

// Allocate working memory for the new lookup table data
char *pNewData = new char[pLut->TotalSize];

// Fill contents of data buffer

// Write new data to lookup table
success = pLut->Write(0, pNewData, pLut->TotalSize);

// Program the acquisition hardware with the new LUT data
success = pAcq->ApplyLut(true, 0);

// Free working memory
delete [] pNewData;

// Release resources for acquisition object, and free it
// Note that the delete operator actually calls the Dispose method
success = pAcq->Destroy();
delete pAcq;
```

Sapera LT ++ – Camera Acquisition Example

The camera acquisition example demonstrates how to grab a live image into a buffer allocated within system memory using the Genie M640 camera as an acquisition device. Acquiring an image can be performed either by using the camera default settings (feature values stored in the camera) or by loading a configuration file. The configuration file can be generated using *CamExpert*.

After the *SapAcqDevice* class is initialized (with or without using a configuration file), certain parameters are retrieved from it (acquisition width, height, and format) to create a compatible buffer.

Before starting the transfer, you must create a transfer path between the *SapAcqDevice* class and the *SapBuffer* class using one of the *SapTransfer*'s derived classes (*SapAcqDeviceToBuf* in this case). Furthermore when requesting a transfer stop, you must call *SapTransfer::Wait* to wait for the transfer process to terminate completely.

Sample Code

```
// Transfer callback function is called each time a complete frame is transferred
//
void XferCallback(SapXferCallbackInfo *pInfo)
{
    // Display the last transferred frame
    SapView *pView = (SapView *) pInfo->GetContext();
    pView->Show();
}

// Example program
//
main()
{
    // Allocate acquisition object
    SapAcqDevice *pAcq =
        new SapAcqDevice("Genie_M640_1", FALSE); // uses camera default settings
        //new SapAcqDevice("Genie_M640", "MyCamera.ccf"); // loads configuration file

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

    // Allocate view object to display in an internally created window
    SapView *pView = new SapView(pBuffer, (HWND)-1);

    // Allocate transfer object to link acquisition and buffer
    SapAcqDeviceToBuf *pTransfer =
        new SapAcqDeviceToBuf(pAcq, pBuffer, XferCallback, pView);

    // Create resources for all objects
    BOOL success = pAcq->Create();
    success = pBuffer->Create();
    success = pView->Create();
    success = pTransfer->Create();

    // Start a continuous transfer (live grab)
    success = pTransfer->Grab();
    printf("Press any key to stop grab\n");
    getch();

    // Stop the transfer and wait (timeout = 5 seconds)
    success = pTransfer->Freeze();
    success = pTransfer->Wait(5000);
    printf("Press any key to terminate\n");
    getch();

    // Release resources for all objects
    success = pTransfer->Destroy();
    success = pView->Destroy();
    success = pBuffer->Destroy();
    success = pAcq->Destroy();

    // Free all objects
    delete pTransfer;
    delete pView;
    delete pBuffer;
    delete pAcq;
    return 0;
}
```

Sapera LT .NET – Camera Acquisition Example

The camera acquisition example demonstrates how to grab a live image into a buffer allocated within system memory using the Genie M640 camera as an acquisition device. Acquiring an image can be performed either by using the camera default settings (feature values stored in the camera) or by loading a configuration file (which can be generated using CamExpert).

After the SapAcqDevice object is initialized (with or without a configuration file), certain parameters are retrieved from it (acquisition width, height, and format) to create a compatible buffer.

Before starting the transfer, you must create a transfer path between the SapAcqDevice and the SapBuffer objects using the SapTransfer class or one of the specialized transfer classes (SapAcqDeviceToBuf in this case). Furthermore when requesting a transfer stop, you must call the Wait method of SapTransfer to wait for the transfer process to terminate completely.

Sample Code for C#

```
// Transfer event handler is called each time a complete frame is transferred
static void SapTransfer_XferNotify(object sender, SapXferNotifyEventArgs args)
{
    SapView view = args.Context as SapView;
    view.Show();
}
static void Main(string[] args)
{
    // Allocate acquisition object using default camera settings
    SapAcqDevice acqDevice = new SapAcqDevice(new SapLocation("Genie_M640_1", 0));

    // Allocate acquisition object using a camera configuration file
    //SapAcqDevice acqDevice = new SapAcqDevice(new SapLocation("Genie_M640_1", 0), "MyCamera.ccf");

    // Allocate buffer object, taking settings directly from the acquisition
    SapBuffer buffer = new SapBuffer(1, acqDevice,
        SapBuffer.MemoryType.ScatterGather);

    // Allocate view object to display in an internally created window
    SapView view = new SapView(buffer);

    // Allocate transfer object to link acquisition and buffer
    SapAcqDeviceToBuf transfer = new SapAcqDeviceToBuf(acqDevice, buffer);

    transfer.Pairs[0].EventType = SapXferPair.XferEventType.EndOfFrame;
    transfer.XferNotify += new SapXferNotifyHandler(SapTransfer_XferNotify);
    transfer.XferNotifyContext = view;

    // Create resources for all objects
    bool success = acqDevice.Create();
    success = buffer.Create();
    success = view.Create();
    success = transfer.Create();

    // Start a continuous transfer (live grab)
    success = transfer.Grab();
    Console.WriteLine("Press any key to stop grab\n");
    Console.ReadKey(true);

    // Stop the transfer and wait (timeout = 5 seconds)
    success = transfer.Freeze();
    success = transfer.Wait(5000);
    Console.WriteLine("Press any key to terminate\n");

    // Release resources for all objects
    success = transfer.Destroy();
    success = view.Destroy();
    success = buffer.Destroy();
    success = acqDevice.Destroy();

    // Free all objects
    transfer.Dispose();
    view.Dispose();
    buffer.Dispose();
    acqDevice.Dispose();
}
```

Equivalent Code for Visual Basic .NET

```
' Transfer event handler is called each time a complete frame is transferred
Sub SapTransfer_XferNotify(ByVal sender As Object, _
    ByVal args As SapXferNotifyEventArgs)
    Dim view As SapView = args.Context
    view.Show()
End Sub

Sub Main()
    ' Allocate acquisition object using default camera settings
    Dim acqDevice As SapAcqDevice = _
        New SapAcqDevice(New SapLocation("Genie_M640_1", 0))

    ' Allocate acquisition object using a camera configuration file
    Dim acqDevice As SapAcqDevice = New SapAcqDevice( _
        'New SapLocation("Genie_M640_1", 0), "MyCamera.ccf")

    ' Allocate buffer object, taking settings directly from the acquisition
    Dim buffer As SapBuffer = _
        New SapBuffer(1, acqDevice, SapBuffer.MemoryType.ScatterGather)

    ' Allocate view object to display in an internally created window
    Dim view As SapView = New SapView(buffer)

    ' Allocate transfer object to link acquisition and buffer
    Dim transfer As SapAcqDeviceToBuf = _
        New SapAcqDeviceToBuf(acqDevice, buffer)

    transfer.Pairs(0).EventType = SapXferPair.XferEventType.EndOfFrame
    AddHandler transfer.XferNotify, AddressOf SapTransfer_XferNotify
    transfer.XferNotifyContext = view

    ' Create resources for all objects
    Dim success As Boolean = acqDevice.Create()
    success = buffer.Create()
    success = view.Create()
    success = transfer.Create()

    ' Start a continuous transfer (live grab)
    success = transfer.Grab()
    Console.WriteLine("Press any key to stop grab\n")
    Console.ReadKey(True)

    ' Stop the transfer and wait (timeout = 5 seconds)
    success = transfer.Freeze()
    success = transfer.Wait(5000)
    Console.WriteLine("Press any key to terminate\n")

    ' Release resources for all objects
    success = transfer.Destroy()
    success = view.Destroy()
    success = buffer.Destroy()
    success = acqDevice.Destroy()

    ' Free all objects
    transfer.Dispose()
    view.Dispose()
    buffer.Dispose()
    acqDevice.Dispose()
End Sub
```

Equivalent Code for C++

```
// Transfer event handler is called each time a complete frame is transferred
static void SapTransfer_XferNotify(Object^ sender,
    SapXferNotifyEventArgs^ args)
{
    SapView^ pView = safe_cast<SapView^>(args->Context);
    pView->Show();
}

int main(array<String ^>^ args)
{
    // Allocate acquisition object using default camera settings
    SapAcqDevice^ pAcqDevice =
        gcnew SapAcqDevice(gcnew SapLocation("Genie_M640_1", 0));

    // Allocate acquisition object using a camera configuration file
    //SapAcqDevice^ pAcqDevice = gcnew SapAcqDevice(
        //gcnew SapLocation("Genie_M640_1", 0), "MyCamera.ccf");

    // Allocate buffer object, taking settings directly from the acquisition
    SapBuffer^ pBuffer =
        gcnew SapBuffer(1, pAcqDevice, SapBuffer::MemoryType::ScatterGather);

    // Allocate view object to display in an internally created window
    SapView^ pView = gcnew SapView(pBuffer);

    // Allocate transfer object to link acquisition and buffer
    SapAcqDeviceToBuf^ pTransfer =
        gcnew SapAcqDeviceToBuf(pAcqDevice, pBuffer);

    pTransfer->Pairs[0]->EventType = SapXferPair::XferEventType::EndOfFrame;
    pTransfer->XferNotify +=
        gcnew SapXferNotifyHandler(SapTransfer_XferNotify);
    pTransfer->XferNotifyContext = pView;

    // Create resources for all objects
    bool success = pAcqDevice->Create();
    success = pBuffer->Create();
    success = pView->Create();
    success = pTransfer->Create();

    // Start a continuous transfer (live grab)
    success = pTransfer->Grab();
    Console::WriteLine("Press any key to stop grab\n");
    Console::ReadKey(true);

    // Stop the transfer and wait (timeout = 5 seconds)
    success = pTransfer->Freeze();
    success = pTransfer->Wait(5000);
    Console::WriteLine("Press any key to terminate\n");

    // Release resources for all objects
    success = pTransfer->Destroy();
    success = pView->Destroy();
    success = pBuffer->Destroy();
    success = pAcqDevice->Destroy();

    // Free all objects
    // Note that the delete operator actually calls the Dispose method
    delete pTransfer;
    delete pView;
    delete pBuffer;
    delete pAcqDevice;

    return 0;
}
```

Sapera LT ++ – Modifying Camera Features

The following section describes how to modify camera features individually or by group.

Accessing Feature Information and Values

The following example shows how camera features can be accessed. Information such as type, range and access mode can be retrieved for each supported feature. The *SapAcqDevice* class also allows modifying the feature values by directly writing to the camera. To get more information on a feature, the SapFeature object can be retrieved for a specific feature using the SapAcqDevice::GetFeatureInfo function. The SapFeature class can then be used to determine the properties of the feature.

In some circumstances, a set of feature values are tightly coupled together and must be written to the camera at the same time. The next section shows how to proceed in such a case.

Sample Code for Sapera LT ++

```
//
// Callback Function
//
void CameraCallback(SapAcqDeviceCallbackInfo* pInfo)
{
    BOOL status;
    int eventCount;
    int eventIndex;
    char eventName[64];

    // Retrieve count, index and name of the received event
    status = pInfo->GetEventCount(&eventCount);
    status = pInfo->GetEventIndex(&eventIndex);
    status = pInfo->GetAcqDevice()->GetEventNameByIndex(eventIndex, eventName,
        sizeof(eventName));

    // Check for "Feature Value Changed" event
    if (strcmp(eventName, "Feature Value Changed") == 0)
    {
        // Retrieve index and name of the feature that has changed
        int featureIndex;
        char featureName[64];
        status = pInfo->GetFeatureIndex(&featureIndex);
        status = pInfo->GetAcqDevice()->GetFeatureNameByIndex(featureIndex, featureName,
            sizeof(featureName));
    }
}

//
// Main Program
//
main()
{
    BOOL status;

    // Create a camera object
    SapAcqDevice camera("Genie_M640_1");
    status = camera.Create();

    // Get the number of features provided by the camera
    int featureCount;
    status = camera.GetFeatureCount(&featureCount);

    // Create an empty feature object (to receive information)
    SapFeature feature("Genie_M640_1");
    status = feature.Create();

    //
    // Example 1 : Browse through the feature list
    //
    int featureIndex;
    for (featureIndex = 0; featureIndex < featureCount; featureIndex++)
    {
        char featureName[64];
```

```
SapFeature::Type featureType;
SapFeature::AccessMode featureAccessMode;

// Get information from current feature
// Get feature object
status = camera.GetFeatureInfo(featureIndex, &feature);

// Extract name and type from object
status = feature.GetName(featureName, sizeof(featureName));
status = feature.GetType(&featureType);
status = feature.GetAccessMode(&featureAccessMode);

// Get/set value from/to current feature
switch (featureType)
{
    // Feature is a 64-bit integer
    case SapFeature::TypeInt64:
    {
        UINT64 value;
        if (featureAccessMode == SapFeature::AccessRW)
        {
            status = camera.GetFeatureValue(featureIndex, &value);
            value += 10;
            status = camera.SetFeatureValue(featureIndex, value);
        }
    }
    break;
}
```

```

    // Feature is a boolean
    case SapFeature::TypeBool:
    {
        BOOL value;
        if (featureAccessMode == SapFeature::AccessRW)
        {
            status = camera.GetFeatureValue(featureIndex, &value);
            value = !value;
            status = camera.SetFeatureValue(featureIndex, value);
        }
    }
    break;

    // Other feature types
    // ...
}

//
// Example 2 : Access specific feature (integer example)
//
// Get feature object
status = camera.GetFeatureInfo("Gain", &feature);

// Extract minimum, maximum and increment values
UINT32 min, max, inc;
status = feature.GetMin(&min);
status = feature.GetMax(&max);
status = feature.GetInc(&inc);

// Read, modify and write value
UINT32 value;
status = camera.GetFeatureValue("Gain", &value);
value += 10;
status = camera.SetFeatureValue("Gain", value);

//
// Example 3 : Access specific feature (enumeration example)
//
// Get feature object
status = camera.GetFeatureInfo("ExposureMode", &feature);

// Get number of items in enumeration
int enumCount;
status = feature.GetEnumCount(&enumCount);

int enumIndex, enumValue;
char enumStr[64];
for (enumIndex = 0; enumIndex < enumCount; enumIndex++)
{
    // Get item string and value
    status = feature.GetEnumString(enumIndex, enumStr, sizeof(enumStr));
    status = feature.GetEnumValue(enumIndex, &enumValue);
}

```

```

// Read a value and get its associated string
status = camera.GetFeatureValue("ExposureMode", &enumValue);
status = feature.GetEnumStringFromValue(enumValue, enumStr, sizeof(enumStr));

// Write a value corresponding to known string
status = feature.GetEnumValueFromString("Programmable", &enumValue);
status = camera.SetFeatureValue("ExposureMode", enumValue);

//
// Example 4 : Access specific feature (LUT example)
//
// Select a LUT and retrieve its size and format
UINT32 lutNEntries, lutFormat;

status = camera.GetFeatureValue("LUTNumberEntries", &lutNEntries);
status = camera.GetFeatureValue("LUTFormat", &lutFormat);

// Create and generate a compatible software LUT
SapLut lut(lutNEntries, lutFormat);
status = lut.Create();
status = lut.Reverse();

// Write LUT values to camera
status = camera.SetFeatureValue("LUTData", &lut);

//
// Example 5 : Callback management
//
// Browse event list
int numEvents;
status = camera.GetEventCount(&numEvents);

int eventIndex;
for (eventIndex = 0; eventIndex < numEvents; eventIndex++)
{
    char eventName[64];
    status = camera.GetEventNameByIndex(eventIndex, eventName, sizeof(eventName));
}

// Register event by name
status = camera.RegisterCallback("Feature Value Changed", CameraCallback, NULL);

// Modified a feature (Will trigger callback function)
status = camera.SetFeatureValue("Gain", 80);

// Unregister event by name
status = camera.UnregisterCallback("Feature Value Changed");
}

```

Writing Feature Values by Group

When a series of features are tightly coupled, they are difficult to modify without following a specific order. For example, a region-of-interest (ROI) has four values (OffsetX, OffsetY, Width and Height) that are inter-dependent and must be defined as a group. To solve this problem, the *SapAcqDevice* class allows you to temporarily set the feature values in an “internal cache” and then download the values to the camera at the same time. The following code illustrates this process using an ROI example.

Sample Code for Sapera LT ++

```
...
// Set manual mode to update features
success = pAcq->SetUpdateFeatureMode(SapAcqDevice::UpdateFeatureManual);

// Set buffer left position (in the internal cache only)
success = pAcq->SetFeatureValue("OffsetX", 50);

// Set buffer top position (in the internal cache only)
success = pAcq->SetFeatureValue("OffsetY", 50);

// Set buffer width (in the internal cache only)
success = pAcq->SetFeatureValue("Width", 300);

// Set buffer height (in the internal cache only)
success = pAcq->SetFeatureValue("Height", 300);

// Write features to device (by reading values from the internal cache)
success = pAcq->UpdateFeaturesToDevice();

// Set back the automatic mode
success = pAcq->SetUpdateFeatureMode(SapAcqDevice::UpdateFeatureAuto);

...
```

For more details, see the *Sapera LT ++ Programmer's Manual*.

Sapera LT .NET – Modifying Camera Features

The following section describes how to modify camera features individually or by group.

Accessing Feature Information and Values

The following example shows how features of the camera can be accessed. Information such as type, range and access mode can be retrieved for each supported feature. The SapAcqDevice class also allows modifying the feature values by directly writing to the camera. In some circumstances a set of feature values are tightly coupled together and must be written to the camera at the same time. The next section shows how to proceed in such a case.

Sample Code for C#

```
// Event handler
public static void
AcqDevice_AcqDeviceNotify(Object sender, SapAcqDeviceNotifyEventArgs args)
{
    SapAcqDevice acqDevice = sender as SapAcqDevice;

    // Retrieve count, index and name of the received event
    int eventCount = args.EventCount;
    int eventIndex = args.EventIndex;
    string eventName = acqDevice.EventNames[eventIndex];

    // Check for "Feature Value Changed" event
    if (eventName == "Feature Value Changed")
    {
        // Retrieve index and name of the feature that has changed
        int featureIndex = args.FeatureIndex;
        string featureName = acqDevice.FeatureNames[featureIndex];
    }
}

static void Main(string[] args)
{
    // Allocate acquisition object using default camera settings,
    // and create resources
    SapAcqDevice acqDevice =
        new SapAcqDevice(new SapLocation("Genie_M640_1", 0));
    bool success = acqDevice.Create();

    // Get the number of features provided by the camera
    int featureCount = acqDevice.FeatureCount;

    // Create an empty feature object (to receive information),
    // and create resources
    SapFeature feature = new SapFeature(new SapLocation("Genie_M640_1", 0));
    success = feature.Create();

    //
    // Example 1 : Browse through the feature list
    //
    for (int featureIndex = 0; featureIndex < featureCount; featureIndex++)
    {
        // Get information from current feature
        // Get feature object
        success = acqDevice.GetFeatureInfo(featureIndex, feature);

        // Extract name and type from object
        string featureName = feature.Name;
        SapFeature.Type featureDataType = feature.DataType;
        SapFeature.AccessMode featureAccessMode = feature.DataAccessMode;

        // Get/set value from/to current feature
        switch (featureDataType)
        {
            // Feature is a 64-bit integer
            case SapFeature.Type.Int64:
            {
                long localFeatureValue;
                if (featureAccessMode == SapFeature.AccessMode.RW)
```

```

        {
            success = acqDevice.GetFeatureValue(
                featureIndex, out localFeatureValue);
            localFeatureValue += 10;
            success = acqDevice.SetFeatureValue(
                featureIndex, localFeatureValue);
        }
    }
    break;

    // Feature is a boolean
    case SapFeature.Type.Bool:
    {
        bool localFeatureValue;
        if (featureAccessMode == SapFeature.AccessMode.RW)
        {
            success = acqDevice.GetFeatureValue(
                featureIndex, out localFeatureValue);
            localFeatureValue = !localFeatureValue;
            success = acqDevice.SetFeatureValue(
                featureIndex, localFeatureValue);
        }
    }
    break;

    // Other feature types
    // ...
}

//
// Example 2 : Access specific feature (integer example)
//
// Get feature object
success = acqDevice.GetFeatureInfo("Gain", feature);

// Extract minimum, maximum and increment values
int featureValueMin;
int featureValueMax;
int featureValueIncrement;

success = feature.GetValueMin(out featureValueMin);
success = feature.GetValueMax(out featureValueMax);
success = feature.GetValueIncrement(out featureValueIncrement);

// Read, modify and write value
int featureValue;
success = acqDevice.GetFeatureValue("Gain", out featureValue);
featureValue += 10;
success = acqDevice.SetFeatureValue("Gain", featureValue);

//
// Example 3 : Access specific feature (enumeration example)
//
// Get feature object
success = acqDevice.GetFeatureInfo("ExposureMode", feature);

// Get number of items in enumeration
int featureEnumCount = feature.EnumCount;

// Get all enumeration strings and values
string[] featureEnumText = feature.EnumText;
int[] featureEnumValues = feature.EnumValues;

// Get individual enumeration strings and values
string enumText;
int enumValue;

for (int featureEnumIndex = 0; featureEnumIndex < featureEnumCount;
    featureEnumIndex++)
{
    enumText = featureEnumText[featureEnumIndex];
    enumValue = featureEnumValues[featureEnumIndex];
}

// Read a value and get its associated string
success = acqDevice.GetFeatureValue("ExposureMode", out enumValue);
success = feature.GetEnumTextFromValue(enumValue, out enumText);

```

```

// Write a value corresponding to known string
success = feature.GetEnumValueFromText("Programmable", out enumValue);
success = acqDevice.SetFeatureValue("ExposureMode", enumValue);

//
// Example 4 : Access specific feature (LUT example)
//
// Select a LUT and retrieve its size and format
int numLutEntries;
int lutFormat;

success = acqDevice.GetFeatureValue("LUTNumberEntries", out numLutEntries);
success = acqDevice.GetFeatureValue("LUTFormat", out lutFormat);

// This cast is OK, because the "LUTFormat" feature uses the same values
// as the SapFormat enumeration
SapFormat saperaLutFormat = (SapFormat)lutFormat;

// Create and generate a compatible software LUT
SapLut lut = new SapLut(numLutEntries, saperaLutFormat);
success = lut.Create();
success = lut.Reverse();

// Write LUT values to camera
success = acqDevice.SetFeatureValue("LUTData", lut);

//
// Example 5 : Callback management
//
// Get all event names
string[] eventNames = acqDevice.EventNames;

// Browse event list
int numEvents = acqDevice.EventCount;

for (int eventIndex = 0; eventIndex < numEvents; eventIndex++)
{
    string eventName = eventNames[eventIndex];
}

// Enable event by name
success = acqDevice.EnableEvent("Feature Value Changed");
acqDevice.AcqDeviceNotify +=
    new SapAcqDeviceNotifyHandler(AcqDevice_AcqDeviceNotify);

// Modify a feature (will trigger an event)
success = acqDevice.SetFeatureValue("Gain", 80);

// Disable event by name
acqDevice.AcqDeviceNotify -=
    new SapAcqDeviceNotifyHandler(AcqDevice_AcqDeviceNotify);

// Release resources for all objects
success = lut.Destroy();
success = feature.Destroy();
success = acqDevice.Destroy();

// Free all objects
lut.Dispose();
feature.Dispose();
acqDevice.Dispose();
}

```

Equivalent Code for Visual Basic .NET

```
' Event handler
Sub AcqDevice_AcqDeviceNotify(ByVal sender As Object, _
    ByVal args As SapAcqDeviceNotifyEventArgs)
    Dim acqDevice As SapAcqDevice = sender

    ' Retrieve count, index and name of the received event
    Dim eventCount As Integer = args.EventCount
    Dim eventIndex As Integer = args.EventIndex
    Dim eventName As String = acqDevice.EventNames(eventIndex)

    ' Check for "Feature Value Changed" event
    If eventName = "Feature Value Changed" Then
        ' Retrieve index and name of the feature that has changed
        Dim featureIndex As Integer = args.FeatureIndex
        Dim featureName As String = acqDevice.FeatureNames(featureIndex)
    End If
End Sub

Sub Main()
    ' Allocate acquisition object using default camera settings,
    ' and create resources
    Dim acqDevice As SapAcqDevice = _
        New SapAcqDevice(New SapLocation("Genie_M640_1", 0))
    Dim success As Boolean = acqDevice.Create()

    ' Get the number of features provided by the camera
    Dim featureCount As Integer = acqDevice.FeatureCount

    ' Create an empty feature object (to receive information),
    ' and create resources
    Dim feature As SapFeature = _
        New SapFeature(New SapLocation("Genie_M640_1", 0))
    success = feature.Create()

    '
    ' Example 1 : Browse through the feature list
    '
    For featureIndex As Integer = 0 To featureCount - 1
        ' Get information from current feature
        ' Get feature object
        success = acqDevice.GetFeatureInfo(featureIndex, feature)

        ' Extract name and type from object
        Dim featureName As String = feature.Name
        Dim featureDataType As SapFeature.Type = feature.DataType
        Dim featureAccessMode As SapFeature.AccessMode = feature.DataAccessMode

        ' Get/set value from/to current feature
        Select Case featureDataType
            ' Feature is a 64-bit integer
            Case SapFeature.Type.Int64
                Dim localFeatureValue As Long
                If featureAccessMode = SapFeature.AccessMode.RW Then
                    success = acqDevice.GetFeatureValue( _
                        featureIndex, localFeatureValue)
                    localFeatureValue += 10
                    success = acqDevice.SetFeatureValue( _
                        featureIndex, localFeatureValue)
                End If

            ' Feature is a boolean
            Case SapFeature.Type.Bool
                Dim localFeatureValue As Boolean
                If featureAccessMode = SapFeature.AccessMode.RW Then
                    success = acqDevice.GetFeatureValue( _
                        featureIndex, localFeatureValue)
                    localFeatureValue = Not localFeatureValue
                    success = acqDevice.SetFeatureValue( _
                        featureIndex, localFeatureValue)
                End If

            Case Else
                ' Other feature types
                , ...
        End Select
    End For
End Sub
```

```

Next
,
' Example 2 : Access specific feature (integer example)
,
' Get feature object
success = acqDevice.GetFeatureInfo("Gain", feature)

' Extract minimum, maximum and increment values
Dim featureValueMin As Integer
Dim featureValueMax As Integer
Dim featureValueIncrement As Integer

success = feature.GetValueMin(featureValueMin)
success = feature.GetValueMax(featureValueMax)
success = feature.GetValueIncrement(featureValueIncrement)

' Read, modify and write value
Dim featureValue As Integer
success = acqDevice.GetFeatureValue("Gain", featureValue)
featureValue = featureValue + 10
success = acqDevice.SetFeatureValue("Gain", featureValue)
,
' Example 3 : Access specific feature (enumeration example)
,
' Get feature object
success = acqDevice.GetFeatureInfo("ExposureMode", feature)

' Get number of items in enumeration
Dim featureEnumCount As Integer = feature.EnumCount

' Get all enumeration strings and values
Dim featureEnumText() As String = feature.EnumText
Dim featureEnumValues() As Integer = feature.EnumValues

' Get individual enumeration strings and values
Dim enumText As String = Nothing
Dim enumValue As Integer

For featureEnumIndex As Integer = 0 To featureEnumCount - 1
    enumText = featureEnumText(featureEnumIndex)
    enumValue = featureEnumValues(featureEnumIndex)
Next

' Read a value and get its associated string
success = acqDevice.GetFeatureValue("ExposureMode", enumValue)
success = feature.GetEnumTextFromValue(enumValue, enumText)

' Write a value corresponding to known string
success = feature.GetEnumValueFromText("Programmable", enumValue)
success = acqDevice.SetFeatureValue("ExposureMode", enumValue)
,
' Example 4 : Access specific feature (LUT example)
,
' Select a LUT and retrieve its size and format
Dim numLutEntries As Integer
Dim lutFormat As Integer

success = acqDevice.GetFeatureValue("LUTNumberEntries", numLutEntries)
success = acqDevice.GetFeatureValue("LUTFormat", lutFormat)

' This cast is OK, because the "LUTFormat" feature uses the same values
' as the SapFormat enumeration
'Dim saperaLutFormat As SapFormat = lutFormat

' Create and generate a compatible software LUT
Dim lut As SapLut = New SapLut(numLutEntries, lutFormat)
success = lut.Create()
success = lut.Reverse()

' Write LUT values to camera
success = acqDevice.SetFeatureValue("LUTData", lut)
,
' Example 5 : Callback management
,

```

```

' Get all event names
Dim eventNames() As String = acqDevice.EventNames

' Browse event list
Dim numEvents As Integer = acqDevice.EventCount

For eventIndex As Integer = 0 To numEvents - 1
    Dim eventName As String = eventNames(eventIndex)
Next

' Enable event by name
success = acqDevice.EnableEvent("Feature Value Changed")
AddHandler acqDevice.AcqDeviceNotify, AddressOf AcqDevice_AcqDeviceNotify

' Modified a feature (will trigger an event)
success = acqDevice.SetFeatureValue("Gain", 80)

' Disable event by name
RemoveHandler acqDevice.AcqDeviceNotify, _
    AddressOf AcqDevice_AcqDeviceNotify

' Release resources for all objects
success = lut.Destroy()
success = feature.Destroy()
success = acqDevice.Destroy()

' Free all objects
lut.Dispose()
feature.Dispose()
acqDevice.Dispose()
End Sub

```

Equivalent Code for C++

```

// Event handler
static void AcqDevice_AcqDeviceNotify(Object^ sender,
    SapAcqDeviceNotifyEventArgs^ args)
{
    SapAcqDevice^ pAcqDevice = safe_cast<SapAcqDevice^>(sender);

    // Retrieve count, index and name of the received event
    int eventCount = args->EventCount;
    int eventIndex = args->EventIndex;
    String^ eventName = pAcqDevice->EventNames[eventIndex];

    // Check for "Feature Value Changed" event
    if (String::Compare(eventName, "Feature Value Changed") == 0)
    {
        // Retrieve index and name of the feature that has changed
        int featureIndex = args->FeatureIndex;
        String^ featureName = pAcqDevice->FeatureNames[featureIndex];
    }
}

// Example program
int main(array<String ^>^ args)
{
    // Allocate acquisition object using default camera settings,
    // and create resources
    SapAcqDevice^ pAcqDevice =
        gcnew SapAcqDevice(gcnew SapLocation("Genie_M640_1", 0));
    bool success = pAcqDevice->Create();

    // Get the number of features provided by the camera
    int featureCount = pAcqDevice->FeatureCount;

    // Create an empty feature object (to receive information),
    // and create resources
    SapFeature^ pFeature =
        gcnew SapFeature(gcnew SapLocation("Genie_M640_1", 0));
    success = pFeature->Create();
    //
    // Example 1 : Browse through the feature list
    //
    for (int featureIndex = 0; featureIndex < featureCount; featureIndex++)
    {
        // Get information from current feature
    }
}

```

```

// Get feature object
success = pAcqDevice->GetFeatureInfo(featureIndex, pFeature);

// Extract name and type from object
String^ featureName = pFeature->Name;
SapFeature::Type featureDataType = pFeature->DataType;
SapFeature::AccessMode featureAccessMode = pFeature->DataAccessMode;

// Get/set value from/to current feature
switch (featureDataType)
{
    // Feature is a 64-bit integer
    case SapFeature::Type::Int64:
    {
        __int64 featureValue;
        if (featureAccessMode == SapFeature::AccessMode::RW)
        {
            success = pAcqDevice->GetFeatureValue(
                featureIndex, featureValue);
            featureValue += 10;
            success = pAcqDevice->SetFeatureValue(
                featureIndex, featureValue);
        }
        break;

        // Feature is a boolean
        case SapFeature::Type::Bool:
        {
            bool featureValue;
            if (featureAccessMode == SapFeature::AccessMode::RW)
            {
                success = pAcqDevice->GetFeatureValue(
                    featureIndex, featureValue);
                featureValue = !featureValue;
                success = pAcqDevice->SetFeatureValue(
                    featureIndex, featureValue);
            }
            break;

            // Other feature types
            // ...
        }
    }
}
//
// Example 2 : Access specific feature (integer example)
//
// Get feature object
success = pAcqDevice->GetFeatureInfo("Gain", pFeature);

// Extract minimum, maximum and increment values
int featureValueMin;
int featureValueMax;
int featureValueIncrement;

success = pFeature->GetValueMin(featureValueMin);
success = pFeature->GetValueMax(featureValueMax);
success = pFeature->GetValueIncrement(featureValueIncrement);

// Read, modify and write value
int featureValue;
success = pAcqDevice->GetFeatureValue("Gain", featureValue);
featureValue += 10;
success = pAcqDevice->SetFeatureValue("Gain", featureValue);
//
// Example 3 : Access specific feature (enumeration example)
//
// Get feature object
success = pAcqDevice->GetFeatureInfo("ExposureMode", pFeature);

// Get number of items in enumeration
int featureEnumCount = pFeature->EnumCount;

// Get all enumeration strings and values
array<String^>^ featureEnumText = pFeature->EnumText;
array<int>^ featureEnumValues = pFeature->EnumValues;

```

```

// Get individual enumeration strings and values
String^ enumText;
int enumValue;

for (int featureEnumIndex = 0; featureEnumIndex < featureEnumCount;
     featureEnumIndex++)
{
    enumText = featureEnumText[featureEnumIndex];
    enumValue = featureEnumValues[featureEnumIndex];
}

// Read a value and get its associated string
success = pAcqDevice->GetFeatureValue("ExposureMode", enumValue);
success = pFeature->GetEnumTextFromValue(enumValue, enumText);

// Write a value corresponding to known string
success = pFeature->GetEnumValueFromText("Programmable", enumValue);
success = pAcqDevice->SetFeatureValue("ExposureMode", enumValue);
//
// Example 4 : Access specific feature (LUT example)
//
// Select a LUT and retrieve its size and format
int numLutEntries;
int lutFormat;

success = pAcqDevice->GetFeatureValue("LUTNumberEntries", numLutEntries);
success = pAcqDevice->GetFeatureValue("LUTFormat", lutFormat);

// This cast is OK, because the "LUTFormat" feature uses the same values
// as the SapFormat enumeration
SapFormat saperaLutFormat = static_cast<SapFormat>(lutFormat);

// Create and generate a compatible software LUT
SapLut^ pLut = gcnew SapLut(numLutEntries, saperaLutFormat);
success = pLut->Create();
success = pLut->Reverse();

// Write LUT values to camera
success = pAcqDevice->SetFeatureValue("LUTData", pLut);

//
// Example 5 : Callback management
//
// Get all event names
array<String^>^ eventNames = pAcqDevice->EventNames;

// Browse event list
int numEvents = pAcqDevice->EventCount;

for (int eventIndex = 0; eventIndex < numEvents; eventIndex++)
{
    String^ eventName = eventNames[eventIndex];
}

// Enable event by name
success = pAcqDevice->EnableEvent("Feature Value Changed");
pAcqDevice->AcqDeviceNotify +=
    gcnew SapAcqDeviceNotifyHandler(AcqDevice_AcqDeviceNotify);

// Modified a feature (will trigger an event)
success = pAcqDevice->SetFeatureValue("Gain", 80);

// Disable event by name
pAcqDevice->AcqDeviceNotify
    gcnew SapAcqDeviceNotifyHandler(AcqDevice_AcqDeviceNotify);

// Release resources for all objects
success = pLut->Destroy();
success = pFeature->Destroy();
success = pAcqDevice->Destroy();

// Free all objects
// Note that the delete operator actually calls the Dispose method
delete pLut;
delete pFeature;
delete pAcqDevice;
return 0;
}

```

Writing Feature Values by Group

When a series of features are tightly coupled, they are difficult to modify without following a specific order. For example, a region-of-interest (ROI) has four values (OffsetX, OffsetY, Width and Height) that are inter-dependent and must be defined as a group. To solve this problem, the SapAcqDevice class allows you to temporarily set the feature values in an "internal cache" and then download the values to the camera at the same time. The following code samples illustrate this process using an ROI example.

Sample Code for C#

```
// Set manual mode to update features
acqDevice.UpdateMode = SapAcqDevice.UpdateFeatureMode.Manual;

// Set buffer left position (in the internal cache only)
success = acqDevice.SetFeatureValue("OffsetX", 50);

// Set buffer top position (in the internal cache only)
success = acqDevice.SetFeatureValue("OffsetY", 50);

// Set buffer width (in the internal cache only)
success = acqDevice.SetFeatureValue("Width", 300);

// Set buffer height (in the internal cache only)
success = acqDevice.SetFeatureValue("Height", 300);

// Write features to device (by reading values from the internal cache)
success = acqDevice.UpdateFeaturesToDevice();

// Set back the automatic mode
acqDevice.UpdateMode = SapAcqDevice.UpdateFeatureMode.Auto;
```

Equivalent Code for Visual Basic .NET

```
' Set manual mode to update features
acqDevice.UpdateMode = SapAcqDevice.UpdateFeatureMode.Manual

' Set buffer left position (in the internal cache only)
success = acqDevice.SetFeatureValue("OffsetX", 50)

' Set buffer top position (in the internal cache only)
success = acqDevice.SetFeatureValue("OffsetY", 50)

' Set buffer width (in the internal cache only)
success = acqDevice.SetFeatureValue("Width", 300)

' Set buffer height (in the internal cache only)
success = acqDevice.SetFeatureValue("Height", 300)

' Write features to device (by reading values from the internal cache)
success = acqDevice.UpdateFeaturesToDevice()

' Set back the automatic mode
acqDevice.UpdateMode = SapAcqDevice.UpdateFeatureMode.Auto
```

Equivalent Code for C++

```
// Set manual mode to update features
pAcqDevice->UpdateMode = SapAcqDevice::UpdateFeatureMode::Manual;

// Set buffer left position (in the internal cache only)
success = pAcqDevice->SetFeatureValue("OffsetX", 50);

// Set buffer top position (in the internal cache only)
success = pAcqDevice->SetFeatureValue("OffsetY", 50);

// Set buffer width (in the internal cache only)
success = pAcqDevice->SetFeatureValue("Width", 300);

// Set buffer height (in the internal cache only)
success = pAcqDevice->SetFeatureValue("Height", 300);

// Write features to device (by reading values from the internal cache)
success = pAcqDevice->UpdateFeaturesToDevice();

// Set back the automatic mode
pAcqDevice->UpdateMode = SapAcqDevice::UpdateFeatureMode::Auto;
```

Displaying Images

Required Classes

The following three Sapera LT ++ / Sapera LT .NET classes are required to initiate a display process:

- **SapDisplay:** Manages the actual resources on the hardware display device.
- **SapBuffer:** Contains the data to display. Several type options may be chosen when allocating the buffer to be compatible with the different display modes (see the "Working with Buffers" section for more information about these options).
- **SapView:** Links the display to the buffer and synchronizes the display operations.

Display Examples

The example below illustrates how to display an image contained within a system buffer on the computer VGA card. The buffer is transferred to the Windows Desktop using the DIB mode (automatically detected by the SapView Class). When using this mode, a Windows Device-Independent Bitmap (DIB) is first created before being sent to VGA memory.

For more details, see the *Sapera LT ++ Programmer's Manual* or the *Sapera LT .NET Programmer's Manual*.

Example using the Sapera LT ++ API

```
// Allocate and create a 640x480x8 buffer object
SapBuffer *pBuffer = new SapBuffer(1, 640, 480, SapFormatMono8);
BOOL success = pBuffer->Create();

// Allocate and create view object, images will be displayed directly on the desktop
SapView *pView = new SapView(pBuffer, SapHwndDesktop);
success = pView->Create();

// Display the image on the desktop
pView->Show();

// Release resources for all objects
success = pView->Destroy();
success = pBuffer->Destroy();

// Free all objects
delete pView;
delete pBuffer;
```

Example Code for C# using Sapera LT .NET

```
// Allocate and create a 640x480x8 buffer object
SapBuffer buffer = new SapBuffer(1, 640, 480, SapFormat.Mono8,
    SapBuffer.MemoryType.ScatterGather);
bool success = buffer.Create();

// Allocate and create view object, images will be displayed
// directly on the desktop
SapView view = new SapView(buffer);
success = view.Create();

// Display the image on the desktop
view.Show();

// Release resources for all objects
success = view.Destroy();
success = buffer.Destroy();

// Free all objects
view.Dispose();
buffer.Dispose();
```

Equivalent Code for Visual Basic .NET using Sapera LT .NET

```
' Allocate and create a 640x480x8 buffer object
Dim buffer As SapBuffer = New SapBuffer(1, 640, 480, SapFormat.Mono8, _
    SapBuffer.MemoryType.ScatterGather)
Dim success As Boolean = buffer.Create()

' Allocate and create view object, images will be displayed
' directly on the desktop
Dim view As SapView = New SapView(buffer)
success = view.Create()

' Display the image on the desktop
view.Show()

' Release resources for all objects
success = view.Destroy()
success = buffer.Destroy()

' Free all objects
view.Dispose()
buffer.Dispose()
```

Equivalent Code for C++ using Sapera LT .NET

```
// Allocate and create a 640x480x8 buffer object
SapBuffer^ pBuffer = gcnew SapBuffer(1, 640, 480, SapFormat::Mono8,
    SapBuffer::MemoryType::ScatterGather);
bool success = pBuffer->Create();

// Allocate and create view object, images will be displayed
// directly on the desktop
SapView^ pView = gcnew SapView(pBuffer);
success = pView->Create();

// Display the image on the desktop
pView->Show();

// Release resources for all objects
success = pView->Destroy();
success = pBuffer->Destroy();

// Free all objects
// Note that the delete operator actually calls the Dispose method
delete pView;
delete pBuffer;
```

Sapera LT ++ – Displaying in a Windows Application

The SapView Class includes the three methods **OnPaint**, **OnMove**, and **OnSize**. When your Windows application is based on **MFC**, you should call these from within the OnPaint, OnMove, and OnSize handlers in your application program. This ensures that Sapera LT ++ is aware about the changes to the display area.

However, these methods often do not offer, especially for scroll bars, a sufficient level of management following changes to the display area. So it is recommended that you use the methods with the same names in the CImageWnd Class instead. Note that this class is part of the GUI classes. Below is a partial listing of a dialog-based Windows application.

Sample Code Using the Visual C++'s MFC library

```
// Declarations from class header file
SapBuffer *m_pBuffer;
SapView *m_pView;
CImageWnd *m_pImageWnd;
// End declarations from class header file

CSaperaAppDlg::CSaperaAppDlg()
{
    m_pBuffer = NULL;
    m_pView = NULL;
    m_pImageWnd = NULL;

    // Other initialization
    ...
}

BOOL CSaperaAppDlg::OnInitDialog()
{
    // Call default handler
    CDialog::OnInitDialog();

    // Other initialization
    ...

    // Allocate and create a 640x480x8 buffer object
    *m_pBuffer = new SapBuffer(1, 640, 480, SapFormatMono8);
    BOOL success = m_pBuffer->Create();

    // Allocate and create view object, images will be displayed in the MFC CWnd
    // object identified by m_ViewWnd
    m_pView = new SapView(m_pBuffer, m_ViewWnd.GetSafeHwnd());
    success = m_pView->Create();

    // Create image window object. The m_HorizontalScr and m_VerticalScr arguments
    // are MFC CscrollBar objects representing the scroll bars.
    m_pImageWnd = new CImageWnd(m_pView, &m_ViewWnd,
                                &m_HorizontalScr, &m_VerticalScr, this);

    return TRUE;
}

CSaperaAppDlg::OnDestroy()
{
    // Release resources for all objects
    BOOL success = m_pView->Destroy();
    success = m_pBuffer->Destroy();

    // Free all objects
    delete m_pImageWnd;
    delete m_pView;
    delete m_pBuffer;
}
```

```

void CSaperaAppDlg::OnPaint()
{
    if (IsIconic())
    {
        ...
    }
    else
    {
        // Call the default handler to paint a background
        CDialog::OnPaint();
        // Display last acquired image
        m_pImageWnd->OnPaint();
    }
}

void CSaperaAppDlg::OnSize(UINT nType, int cx, int cy)
{
    // Call default handler
    CDialog::OnSize(nType, cx, cy);

    // Make appropriate adjustment in image window
    m_pImageWnd->OnSize();
}

void CSaperaAppDlg::OnMove(int x, int y)
{
    // Call default handler
    CDialog::OnMove(x, y);

    // Make appropriate adjustment in image window
    m_pImageWnd->OnMove();
}

```

For more details, see the *Sapera LT ++ Programmer's Manual* and the source code for the demos and examples included with Sapera LT.

Sapera LT .NET – Displaying in a Windows Application

The SapView Class includes the three methods **OnPaint**, **OnMove**, and **OnSize**. When your application is based on **Windows Forms**, you should call these from within the Paint, Move, and Size event handlers in your application program. This ensures that Sapera LT .NET is aware about the changes to the display area.

However, these methods often do not offer, especially for scroll bars, a sufficient level of management following changes to the display area. The Sapera LT .NET demos include an ImageBox class (with included source code) for this purpose. Below are partial listings which do use the ImageBox class.

Partial C# Listing of a Windows Form Application

```
public static void Form_Paint(Object sender, PaintEventArgs args)
{
    // Find the SapView object corresponding to the form for this event
    Form form = sender as Form;
    SapView view = SapView.FindView(form);
    view.OnPaint();
}

public static void Form_Resize(Object sender, EventArgs args)
{
    // Find the SapView object corresponding to the form for this event
    Form form = sender as Form;
    SapView view = SapView.FindView(form);
    view.OnSize();
}

public static void Form_Move(Object sender, EventArgs args)
{
    // Find the SapView object corresponding to the form for this event
    Form form = sender as Form;
    SapView view = SapView.FindView(form);
    view.OnMove();
}

static void Main(string[] args)
{
    // Allocate and create a 640x480x8 buffer object
    SapBuffer buffer = new SapBuffer(1, 640, 480, SapFormat.Mono8,
        SapBuffer.MemoryType.ScatterGather);
    bool success = buffer.Create();

    // Create the form object for showing images
    Form form = new Form();

    // Allocate and create view object, images will be displayed in the form
    SapView view = new SapView(buffer, form);
    success = view.Create();

    // Display the image in the form
    view.Show();

    // Register events for the view
    form.Paint += new PaintEventHandler(Form_Paint);
    form.Resize += new EventHandler(Form_Resize);
    form.Move += new EventHandler(Form_Move);

    // Show the form, code that follows will execute
    // when the form is manually closed
    Application.Run(form);

    // Release resources for all objects
    success = view.Destroy();
    success = buffer.Destroy();

    // Free all objects
    view.Dispose();
    buffer.Dispose();
}
```

Equivalent Code for Visual Basic .NET

```
Sub Form_Paint(ByVal sender As Object, ByVal args As PaintEventArgs)
    ' Find the SapView object corresponding to the form for this event
    Dim form As Form = sender
    Dim view As SapView = SapView.FindView(form)
    view.OnPaint()
End Sub

Sub Form_Resize(ByVal sender As Object, ByVal args As EventArgs)
    ' Find the SapView object corresponding to the form for this event
    Dim form As Form = sender
    Dim view As SapView = SapView.FindView(form)
    view.OnSize()
End Sub

Sub Form_Move(ByVal sender As Object, ByVal args As EventArgs)
    ' Find the SapView object corresponding to the form for this event
    Dim form As Form = sender
    Dim view As SapView = SapView.FindView(form)
    view.OnMove()
End Sub

Sub Main()
    ' Allocate and create a 640x480x8 buffer object
    Dim buffer As SapBuffer = New SapBuffer(1, 640, 480, SapFormat.Mono8, _
        SapBuffer.MemoryType.ScatterGather)
    Dim success As Boolean = buffer.Create()

    ' Create the form object for showing images
    Dim form As Form = New Form()

    ' Allocate and create view object, images will be displayed in the form
    Dim view As SapView = New SapView(buffer, form)
    success = view.Create()

    ' Display the image in the form
    view.Show()

    ' Register events for the view
    AddHandler form.Paint, AddressOf Form_Paint
    AddHandler form.Resize, AddressOf Form_Resize
    AddHandler form.Move, AddressOf Form_Move

    ' Show the form, code that follows will execute
    ' when the form is manually closed
    Application.Run(form)

    ' Release resources for all objects
    success = view.Destroy()
    success = buffer.Destroy()

    ' Free all objects
    view.Dispose()
    buffer.Dispose()
End Sub
```

Equivalent Code for C++

```
static void Form_Paint(Object^ sender, PaintEventArgs^ args)
{
    // Find the SapView object corresponding to the form for this event
    Form^ pForm = safe_cast<Form^>(sender);
    SapView^ pView = SapView::FindView(pForm);
    pView->OnPaint();
}

static void Form_Resize(Object^ sender, EventArgs^ args)
{
    // Find the SapView object corresponding to the form for this event
    Form^ pForm = safe_cast<Form^>(sender);
    SapView^ pView = SapView::FindView(pForm);
    pView->OnSize();
}

static void Form_Move(Object^ sender, EventArgs^ args)
{
    // Find the SapView object corresponding to the form for this event
    Form^ pForm = safe_cast<Form^>(sender);
    SapView^ pView = SapView::FindView(pForm);
    pView->OnMove();
}

// Example program
int main(array<String ^>^ args)
{
    // Allocate and create a 640x480x8 buffer object
    SapBuffer^ pBuffer = gcnew SapBuffer(1, 640, 480, SapFormat::Mono8,
        SapBuffer::MemoryType::ScatterGather);
    bool success = pBuffer->Create();

    // Create the form object for showing images
    Form^ pForm = gcnew Form();

    // Allocate and create view object, images will be displayed in the form
    SapView^ pView = gcnew SapView(pBuffer, pForm);
    success = pView->Create();

    // Display the image in the form
    pView->Show();

    // Register events for the view
    pForm->Paint += gcnew PaintEventHandler(Form_Paint);
    pForm->Resize += gcnew EventHandler(Form_Resize);
    pForm->Move += gcnew EventHandler(Form_Move);

    // Show the form, code that follows will execute
    // when the form is manually closed
    Application::Run(pForm);

    // Release resources for all objects
    success = pView->Destroy();
    success = pBuffer->Destroy();

    // Free all objects
    // Note that the delete operator actually calls the Dispose method
    delete pView;
    delete pBuffer;

    return 0;
}
```

For more details, see the *Sapera LT .NET Programmer's Manual* and the source code for the demos and examples included with Sapera LT.

Working with Buffers

Root and Child Buffers

A buffer object is created in one of two ways: either as a root SapBuffer object (with no parent) or as a child SapBufferRoi object (with a parent). The parent of the child may also be a child itself, which allows you to build a buffer hierarchy with no restriction on the number of levels. A SapBuffer object can have more than one child SapBufferRoi object.

A SapBufferRoi object shares the same memory space as its parent, and it defines an adjustable rectangular region of interest. A child may be used by acquisition to reduce bandwidth requirements, or by a processing function in order to process a specific region.



Note: SapBufferRoi objects must be destroyed before their parent.

Sapera LT ++ Example – Parent Buffer with Two Children

```
// Allocate and create a 640x480x8 buffer object
SapBuffer *pBuffer = new SapBuffer(1, 640, 480, SapFormatMono8);
BOOL success = pBuffer->Create();

// Allocate and create a 320x240 child in the upper-left corner
SapBufferRoi *pChild1 = new SapBufferRoi(pBuffer, 0, 0, 320, 240)
success = pChild1->Create();

// Allocate and create a 320x240 child in the upper-right corner
SapBufferRoi *pChild2 = new SapBufferRoi(pBuffer, 320, 0, 320, 240)
success = pChild2->Create();

// Use buffers
...

// Release resources for all objects
success = pChild2->Destroy();
success = pChild1->Destroy();
success = pBuffer->Destroy();

// Free all objects
delete pChild2;
delete pChild1;
delete pBuffer;
```

You may modify the origin and dimensions of the region of interest for a child buffer object before calling its Create method. The following example demonstrates this concept.

```
// Swap left and right children, and make their height the same as the parent
success = pChild1->SetRoi(320, 0, 320, 480);
success = pChild2->SetRoi(0, 0, 320, 480);
```

Sapera LT .NET C# Example – Parent Buffer with Two Children

```
// Allocate and create a 640x480x8 buffer object
SapBuffer buffer = new SapBuffer(1, 640, 480, SapFormat.Mono8,
    SapBuffer.MemoryType.ScatterGather);
bool success = buffer.Create();

// Allocate and create a 320x240 child in the upper-left corner
SapBufferRoi child1 = new SapBufferRoi(buffer, 0, 0, 320, 240);
success = child1.Create();

// Allocate and create a 320x240 child in the upper-right corner
SapBufferRoi child2 = new SapBufferRoi(buffer, 320, 0, 320, 240);
success = child2.Create();

// Use buffers
// ...

// Release resources for all objects
success = child1.Destroy();
success = child2.Destroy();
success = buffer.Destroy();

// Free all objects
child1.Dispose();
child2.Dispose();
buffer.Dispose();

//*****
//
// You may modify the origin and dimensions of the region of interest for a child //buffer object before
calling its Create method. The following C# example //demonstrates this concept.

// Allocate a 320x240 child in the upper-left corner
SapBufferRoi child1 = new SapBufferRoi(buffer, 0, 0, 320, 240);

// Allocate a 320x240 child in the upper-right corner
SapBufferRoi child2 = new SapBufferRoi(buffer, 320, 0, 320, 240);

// Swap left and right children, and make their height the same as the parent
success = child1.SetRoi(320, 0, 320, 480);
success = child2.SetRoi(0, 0, 320, 480);

// Create child buffers
success = child1.Create();
success = child2.Create();
```

Sapera LT .NET – Equivalent Code for Visual Basic .NET

```
' Allocate and create a 640x480x8 buffer object
Dim buffer As SapBuffer = New SapBuffer(1, 640, 480, SapFormat.Mono8, _
    SapBuffer.MemoryType.ScatterGather)
Dim success As Boolean = buffer.Create()

' Allocate and create a 320x240 child in the upper-left corner
Dim child1 As SapBufferRoi = New SapBufferRoi(buffer, 0, 0, 320, 240)
success = child1.Create()

' Allocate and create a 320x240 child in the upper-right corner
Dim child2 As SapBufferRoi = New SapBufferRoi(buffer, 320, 0, 320, 240)
success = child2.Create()

' Use buffers
' ...

' Release resources for all objects
success = child1.Destroy()
success = child2.Destroy()
success = buffer.Destroy()

' Free all objects
child1.Dispose()
child2.Dispose()
buffer.Dispose()
```

```

' *****
'
' You may modify the origin and dimensions of the region of interest for a child
' buffer object before calling its Create method. The following Visual Basic .NET
' example demonstrates this concept.

' Allocate a 320x240 child in the upper-left corner
Dim child1 As SapBufferRoi = New SapBufferRoi(buffer, 0, 0, 320, 240)

' Allocate a 320x240 child in the upper-right corner
Dim child2 As SapBufferRoi = New SapBufferRoi(buffer, 320, 0, 320, 240)

' Swap left and right children, and make their height the same as the parent
success = child1.SetRoi(320, 0, 320, 480)
success = child2.SetRoi(0, 0, 320, 480)

' Create child buffers
success = child1.Create()
success = child2.Create()

```

Sapera LT .NET – Equivalent Code for C++:

```

// Allocate and create a 640x480x8 buffer object
SapBuffer^ pBuffer = gcnew SapBuffer(1, 640, 480, SapFormat::Mono8,
    SapBuffer::MemoryType::ScatterGather);
bool success = pBuffer->Create();

// Allocate and create a 320x240 child in the upper-left corner
SapBufferRoi^ pChild1 = gcnew SapBufferRoi(pBuffer, 0, 0, 320, 240);
success = pChild1->Create();

// Allocate and create a 320x240 child in the upper-right corner
SapBufferRoi^ pChild2 = gcnew SapBufferRoi(pBuffer, 320, 0, 320, 240);
success = pChild2->Create();

// Use buffers
// ...

// Release resources for all objects
success = pChild1->Destroy();
success = pChild2->Destroy();
success = pBuffer->Destroy();

// Free all objects
// Note that the delete operator actually calls the Dispose method
delete pChild1;
delete pChild2;
delete pBuffer;

//*****
//
// You may modify the origin and dimensions of the region of interest for a child
//buffer object before calling its Create method. The following C++ example
//demonstrates this concept.

// Allocate a 320x240 child in the upper-left corner
SapBufferRoi^ pChild1 = gcnew SapBufferRoi(pBuffer, 0, 0, 320, 240);

// Allocate a 320x240 child in the upper-right corner
SapBufferRoi^ pChild2 = gcnew SapBufferRoi(pBuffer, 320, 0, 320, 240);

// Swap left and right children, and make their height the same as the parent
success = pChild1->SetRoi(320, 0, 320, 480);
success = pChild2->SetRoi(0, 0, 320, 480);

// Create child buffers
success = pChild1->Create();
success = pChild2->Create();

```

Buffer Types

You may create a SapBuffer object using one of many predefined types. This has an effect on how the actual buffer resources are allocated, and on how the object may be used with other classes, such as SapTransfer and SapView.

Contiguous Memory Buffers (SapBuffer::TypeContiguous)

Buffers are allocated in Sapera LT Contiguous Memory, which is one large chunk of non-pageable and non-moveable memory reserved by Sapera LT at boot time. Buffer data is thus contained in a single memory block (not segmented). These buffers may be used as source and destination for transfer resources.

Scatter-Gather Memory Buffers (SapBuffer::TypeScatterGather)

Buffers are allocated in noncontiguous memory (paged pool). Pages are locked in physical memory so that a scatter-gather list may be built. This allows allocation of very large buffers to be used as source and destination for transfer resources. The maximum amount of memory that may be allocated depends on available memory, the operating system, and the application(s) used. For 32-bit Windows, if the amount of system memory exceeds four GB, Sapera LT automatically uses TypeScatterGatherPhysical instead.

Virtual Buffers (SapBuffer::TypeVirtual)

Similar to TypeScatterGather, except that the memory pages are not locked. This allows allocation of very large buffers, but they cannot be used as source or destination for transfer resources.

Dummy Buffers (SapBuffer::TypeDummy)

Dummy buffers do not have any data memory. They may be used as placeholders by transfer resources when there is no physical data transfer.

Physical Memory Buffers (SapBuffer::TypeUnmapped)

Buffers are allocated as a series of non-contiguous chunks of physical memory. You may not access their data until they have been mapped to virtual memory addresses using the SapBuffer::GetAddress method. This type of buffer is useful if the total amount of needed buffer data exceeds the amount of available virtual memory addresses (2 GB under 32-bit Windows). To avoid a shortage of virtual memory addresses, use the SapBuffer::ReleaseAddress method as soon as you are done accessing their data. Note that you cannot acquire images into these buffers.

This buffer type is not supported in Sapera LT for 64-bit Windows.

Physical Scatter-Gather Memory Buffers (SapBuffer::TypeScatterGatherUnmapped)

These buffers are similar to TypePhysical, except that you can acquire images into them.

This buffer type is not supported in Sapera LT for 64-bit Windows.

Physical Scatter-Gather Memory Buffers (special case) (SapBuffer::TypeScatterGatherPhysical)

These buffers are needed in 64-bit Windows for some frame grabbers (e.g. X64-CL iPro) which feature DMA transfers to the host using 32-bit addresses. These frame grabbers do not support acquisition in regular scatter-gather buffers (SapBuffer::TypeScatterGather), because they require all physical addresses used during DMA transfers to be limited to 32-bit values.

Multiformat IR Buffers



Note: References to functions in the following section use the .NET syntax, however the related C++ functions use similar names.

The multiformat buffer types `SapFormat.RGB888_MONO8` and `SapFormat.RGB161616_MONO16` contain both RGB and monochrome (typically near infrared (IR)) information, in 32-bit or 64-bit format respectively. Use the `SapBuffer.IsMultiformat` method to verify if a buffer is a multiformat buffer.

The `SapBuffer.Copy` or `SapBuffer.SplitComponents` methods can be used to extract either the RGB or monochrome (IR) component into a separate buffer. The `SapBuffer.MergeComponents` method can be used to merge separate components into the multiformat buffer type.

When using the `SapBuffer.ReadElement` method to access the buffer, the data is extracted as a `SapDataRGBA` object, with the A component representing the monochrome (IR) portion.

If accessing the memory directly, for each line in the buffer, the first $\frac{3}{4}$ (left side) represents the RGB data and the last $\frac{1}{4}$ (right side) represents the monochrome (IR) data.

Multiformat buffers use 2 pages; one page for RGB component and one page for the monochrome (IR) component. When displaying multiformat buffers with the `SapView` class, use the `AllPage` and `Page` properties to manage the current (active) page of the buffer (RGB or monochrome) to display. The active page only applies when choosing which format to display when calling the `SapView.Show` function.

For load and save operations, multiformat buffers only support the CRC and RAW formats.

Reading and Writing a Buffer

The simplest way to read or write data to a buffer resource is by accessing it element by element. The ReadElement and WriteElement methods fulfill this purpose.

Sapera LT ++ – Access of a Buffer Object

The following demonstrates how to access data in an 8-bit monochrome SapBuffer object.

Accessing an 8-bit Buffer by Element

```
// Allocate and create a 640x480x8 buffer object
SapBuffer *pBuffer = new SapBuffer(1, 640, 480, SapFormatMono8);
BOOL success = pBuffer->Create();

// Write a constant value in the buffer. SapDataMono is a Sapera LT ++ class
// that encapsulates a Standard API monochrome data type.
success = pBuffer->WriteElement(100, 200, SapDataMono(128));

// Read back the value
SapDataMono data;
success = pBuffer->ReadElement(100, 200, &data);

// Release and free resources for SapBuffer object
success = pBuffer->Destroy();
delete pBuffer;
```

Accessing a Buffer by an Array of Elements

Accessing buffer data in this way is quite straightforward, but, unfortunately, it considerably slows down access time. Alternately, you can access data by reading/writing an array of elements with only one function call through the Read and Write methods. Here is a sample of their usage.

```
// Allocate and create a 640x480x8 buffer object
SapBuffer *pBuffer = new SapBuffer(1, 640, 480, SapFormatMono8);
BOOL success = pBuffer->Create();

// Create an array large enough to hold all buffer data
int size = 640 * 480 * sizeof(BYTE);
BYTE *dataBuf = new BYTE [size];

// Fill array with some values
...

// Write array to buffer resource
success = pBuffer->Write(0, size, dataBuf);

// Read back buffer data
success = pBuffer->Read(0, size, dataBuf);

// Release and free resources for SapBuffer object
success = pBuffer->Destroy();
delete pBuffer;

// Free the data buffer
delete [] dataBuf;
```

Accessing a Buffer Directly Through a Pointer

Although this is faster than the previous method, performance is still an issue because of the data copying operations involved.

The fastest way to access buffer data is to obtain direct access through a pointer. The `GetAddress` and `ReleaseAddress` methods initiate and end direct data access, respectively. The drawback of this method is that you need to know the buffer dimensions, format, and pitch in order to correctly access the data. The following code illustrates this.

```
// Allocate and create a 640x480 RGB 5-6-5 buffer object
SapBuffer *pBuffer =
    new SapBuffer(1, 640, 480, SapFormatRGB565, SapBuffer::TypeScatterGather);
BOOL success = pBuffer->Create();

// Get the buffer pitch in bytes
int pitch = pBuffer->GetPitch();

// Get the buffer data address
BYTE *pData;
success = pBuffer->GetAddress(&pData);

// Access the buffer data
for (int lineNum = 0; lineNum < 480; lineNum++)
{
    WORD *pLine = (WORD *) (pData + lineNum * pitch);
    for (pixelNum = 0; pixelNum < 640; pixelNum++)
    {
        // Process the current line
    }
}

// Release and free resources for SapBuffer object
success = pBuffer->Destroy();
delete pBuffer;
```

For more information on buffer data access functionality, see the *Sapera LT ++ Programmer's Manual*.

Sapera LT .NET – Access of a Buffer Element

The following demonstrates how to access data in an 8-bit monochrome `SapBuffer` object.

Accessing an 8-bit `SapBuffer` Object by Element Using C#

```
// Allocate and create a 640x480x8 buffer object
SapBuffer buffer = new SapBuffer(1, 640, 480, SapFormat.Mono8,
    SapBuffer.MemoryType.ScatterGather);
bool success = buffer.Create();

// Write a constant value in the buffer
success = buffer.WriteElement(100, 200, new SapDataMono(128));

// Read back the value
SapDataMono data = new SapDataMono();
success = buffer.ReadElement(100, 200, data);

// Release resources for SapBuffer object
success = buffer.Destroy();

// Free all objects
data.Dispose();
buffer.Dispose();
```

Equivalent Code for Visual Basic .NET

```
' Allocate and create a 640x480x8 buffer object
Dim buffer As SapBuffer = New SapBuffer(1, 640, 480, SapFormat.Mono8, _
    SapBuffer.MemoryType.ScatterGather)
Dim success As Boolean = buffer.Create()

' Write a constant value in the buffer
success = buffer.WriteElement(100, 200, New SapDataMono(128))

' Read back the value
Dim data As SapDataMono = New SapDataMono()
success = buffer.ReadElement(100, 200, data)

' Release resources for SapBuffer object
success = buffer.Destroy()

' Free all objects
data.Dispose()
buffer.Dispose()
```

Equivalent Code for C++

```
// Allocate and create a 640x480x8 buffer object
SapBuffer^ pBuffer = gcnew SapBuffer(1, 640, 480, SapFormat::Mono8,
    SapBuffer::MemoryType::ScatterGather);
bool success = pBuffer->Create();

// Write a constant value in the buffer
success = pBuffer->WriteElement(100, 200, gcnew SapDataMono(128));

// Read back the value
SapDataMono^ pData = gcnew SapDataMono();
success = pBuffer->ReadElement(100, 200, pData);

// Release resources for SapBuffer object
success = pBuffer->Destroy();

// Free all objects
// Note that the delete operator actually calls the Dispose method
delete pData;
delete pBuffer;
```

Sapera LT .NET – Access of a Buffer by an Array of Elements

Accessing buffer data in this way is quite straightforward, but, unfortunately, it considerably slows down access time. Alternately, you can access data by reading/writing an array of elements with only one function call through the Read and Write methods.

Example Code in C#

```
// Allocate and create a 640x480x8 buffer object
SapBuffer buffer = new SapBuffer(1, 640, 480, SapFormat.Mono8,
    SapBuffer.MemoryType.ScatterGather);
bool success = buffer.Create();

// Create an array large enough to hold all buffer data
int size = 640 * 480;
byte[] dataBuf = new byte[size];

// Fill array with some values
// ...

// Pin the array to avoid Garbage collector move it
GCHandle dataBufHandle = GCHandle.Alloc(dataBuf, GCHandleType.Pinned);
IntPtr dataBufAddress = dataBufHandle.AddrOfPinnedObject();

// Write array to buffer resource
success = buffer.Write(0, size, dataBufAddress);

// Read back buffer data
success = buffer.Read(0, size, dataBufAddress);

// Unpin the array
dataBufHandle.Free()

// Release resources and free SapBuffer object
success = buffer.Destroy();
buffer.Dispose();
```

Equivalent Code for Visual Basic .NET

```
' Allocate and create a 640x480x8 buffer object
Dim buffer As New SapBuffer(1, 640, 480, SapFormat.Mono8, SapBuffer.MemoryType.ScatterGather)
Dim success As Boolean = buffer.Create()

' Create an array large enough to hold all buffer data
Dim size As Integer = 640 * 480
Dim dataBuf As Byte() = New Byte(size - 1) {}

' Fill array with some values
' ...
' Pin the array to avoid Garbage collector move it
Dim dataBufHandle As GCHandle = GCHandle.Alloc(dataBuf, GCHandleType.Pinned)
Dim dataBufAddress As IntPtr = dataBufHandle.AddrOfPinnedObject()

' Write array to buffer resource
success = buffer.Write(0, size, dataBufAddress)

' Read back buffer data
success = buffer.Read(0, size, dataBufAddress)

' Unpin the array
dataBufHandle.Free()

' Release resources and free SapBuffer object
success = buffer.Destroy()
buffer.Dispose()
```

Equivalent Code for C++

```
// Allocate and create a 640x480x8 buffer object
SapBuffer^ pBuffer = gcnew SapBuffer(1, 640, 480, SapFormat::Mono8,
    SapBuffer::MemoryType::ScatterGather);
bool success = pBuffer->Create();

// Create an array large enough to hold all buffer data
int size = 640 * 480 * sizeof(char);
char* dataBuf = new char[size];

// Fill array with some values
// ...
// Write array to buffer resource
success = pBuffer->Write(0, size, dataBuf);

// Read back buffer data
success = pBuffer->Read(0, size, dataBuf);

// Release resources for SapBuffer object
success = pBuffer->Destroy();

// Free all objects
// Note that, for the SapBuffer object, the delete operator
// actually calls the Dispose method
delete pBuffer;
delete [] dataBuf;
```

Sapera LT .NET – Access of a Buffer via a Pointer

Although this is faster than the previous method, performance is still an issue because of the data copying operations involved.

The fastest way to access buffer data is to obtain direct access through a pointer. The `GetAddress` and `ReleaseAddress` methods initiate and end direct data access, respectively. The drawback of this method is that you need to know the buffer dimensions, format, and pitch in order to correctly access the data.

Example Code in C#

```
// Allocate and create a 640x480 RGB 5-6-5 buffer object
SapBuffer buffer = new SapBuffer(1, 640, 480, SapFormat.RGB565,
    SapBuffer.MemoryType.ScatterGather);
bool success = buffer.Create();

// Get the buffer pitch in bytes
int pitch = buffer.Pitch;

// Get the buffer data address
IntPtr address;
success = buffer.GetAddress(out address);

unsafe
{
    char* pData = (char*)address.ToPointer();

    // Access the buffer data
    for (int lineNum = 0; lineNum < 480; lineNum++)
    {
        System.UInt16* pLine = (System.UInt16 *) (pData + lineNum * pitch);
        for (int pixelNum = 0; pixelNum < 640; pixelNum++)
        {
            // Process the current line
        }
    }
}

// Release and free resources for SapBuffer object
// Note that the delete operator actually calls the Dispose method
success = buffer.Destroy();
buffer.Dispose();
```



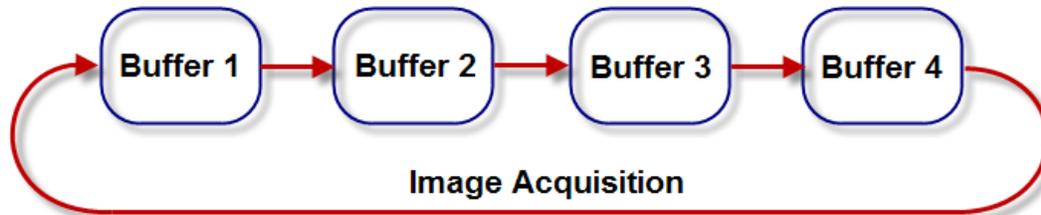
Note: Direct access through a memory pointer is *not available for Visual Basic .NET*.

For more information on buffer data access functionality, see the Sapera LT .NET Programmer's Manual.

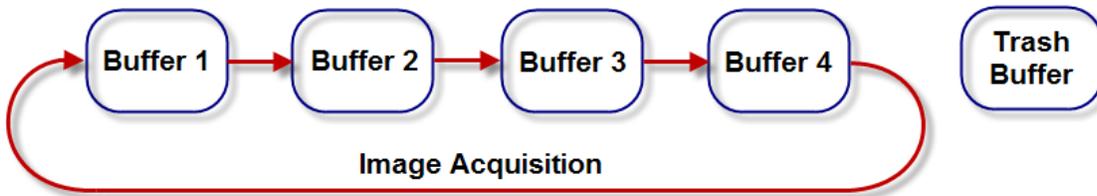
Processing Buffers

To process images while other are being acquired, more than one image buffer is required. When these buffers are linked to an acquisition device using a transfer resource, images are acquired into these multiple buffers in a circular progression, writing into the first buffer, then the second, and so forth, until the last buffer is reached, whereupon the first buffer is overwritten and the cycle continues. While the first buffer is being processed, images are being acquired into the next buffers in the group.

For example, with 4 image buffers:



However, this only allows for processing of all acquired images when the average processing time is less than the time required to acquire one image. When processing cannot keep up with the acquisition frame rate, it is often useful to have a special buffer, not part of the circular list, for throwing away images that cannot be processed. In Sapera LT, this is called the **trash buffer**.



API support for allocating buffers:

- Use one instance of the **SapBuffer** or **SapBufferWithTrash** classes, both of which can be given a buffer count

API support for linking buffers together:

- Use one instance of the **SapTransfer** class, or one of its derived classes, for example, **SapAcqToBuf** or **SapAcqDeviceToBuf**.

Buffer State

Buffers can be in one of two states:

- **Empty**, meaning that images **may be acquired** in the buffer
- **Full**, meaning that unprocessed **data is still present** in the buffer

Buffer state changes between empty and full as follows:

- All buffers are **initially empty**
- As **images are acquired** into the buffers, the **transfer hardware** sets their state to **full**
- When the transfer hardware needs a new buffer for an image, it can either consider or ignore the current buffer state, depending on the transfer cycling mode
- It is the **responsibility of application code** to set the state back to **empty** when the buffers are available again for image acquisition

Note that this is irrelevant for trash buffers.

API support for managing buffer state:

- **Sapera LT ++:** Although you can use the **GetState** and **SetState** methods of the SapBuffer and SapBufferWithTrash classes, it is generally preferable to rely on the **auto-empty mechanism** in the SapTransfer, SapProcessing, and SapView classes.
- **.NET:** Although you can use the **State** property of the SapBuffer and SapBufferWithTrash classes, it is generally preferable to rely on the **auto-empty mechanism** in the SapTransfer, SapProcessing, and SapView classes.

Auto-Empty Mechanism

Refers to an application configurable mechanism by which **buffer state is automatically set to empty**.

There are four possible scenarios:

- Buffer state is **set to empty by the transfer** (SapTransfer), just after the transfer callback function (if any) in the application returns. This is the **default behavior**.
- Buffer state is **set to empty by the processing** (SapProcessing), right after the image in the buffer has been processed, and just before calling the processing callback function (if any) in the application.
- Buffer state is **set to empty by the view** (SapView), right after the image in the buffer has been displayed, and just before calling the view callback function (if any) in the application.
- Buffer state is **set to empty directly in the application**.

API support for managing the auto-empty mechanism:

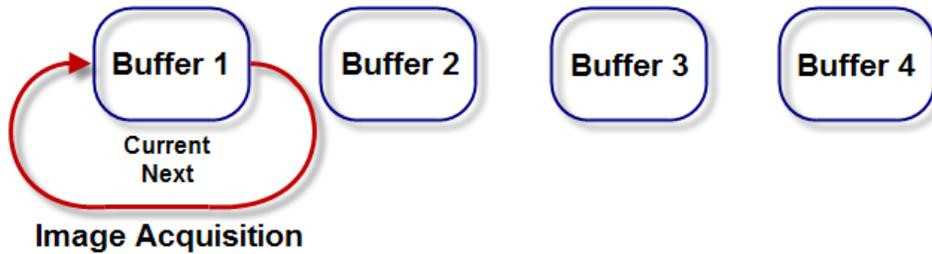
- **Sapera LT ++:** Use the **SetAutoEmpty** and **GetAutoEmpty** methods in the SapTransfer, SapProcessing, and SapView classes.
- **.NET:** Use the **AutoEmpty** property in the SapTransfer, SapProcessing, and SapView classes.

Transfer Cycling Modes

Refers to the criteria used by the transfer when deciding in which buffer the next image will be stored.

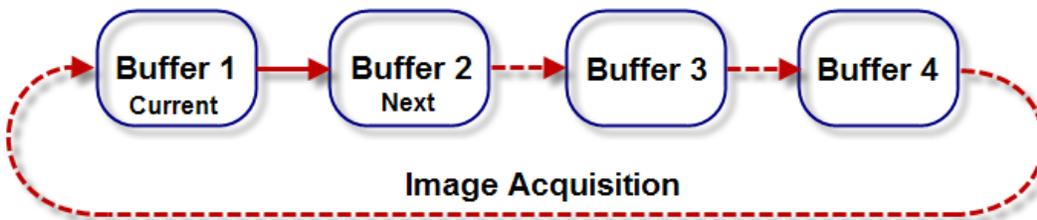
In the **Off** mode:

- **Always** transfer to the **current** buffer
- **Ignore** the buffer **state**
- **Ignore** the presence of a **trash** buffer



In the **Asynchronous** mode:

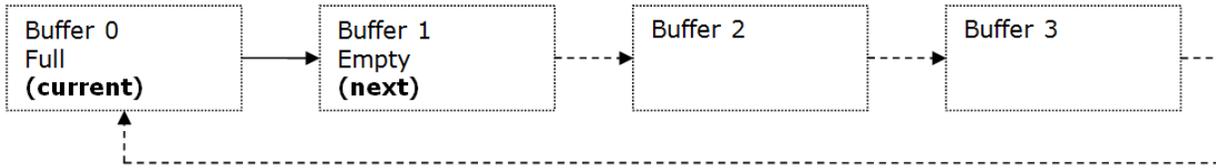
- **Always** transfer to the **next** buffer
- **Ignore** the buffer **state**
- **Ignore** the presence of a **trash** buffer



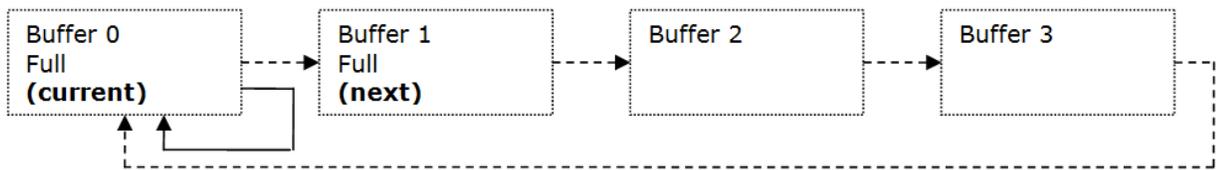
In the **Synchronous** mode:

- **(Case 1)** If the **next** buffer is **empty**, then transfer to the **next** buffer
- **(Case 2)** If the **next** buffer is **full**, then transfer to the **current** buffer
- **Ignore** the presence of a **trash** buffer

Example of **case 1**:



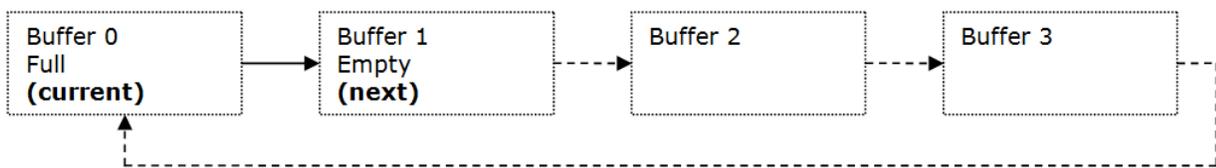
Example of **case 2**:



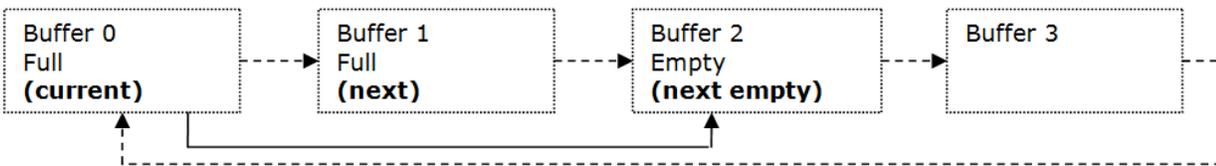
In the **Next Empty** mode:

- **(Case 1)** If the **next** buffer is **empty**, then transfer to the **next** buffer
- **(Case 2)** If the **next** buffer is **full**, transfer to **next empty** buffer in the list
- **(Case 3)** If **all** buffers are **full**, then transfer to the **current** buffer
- **Ignore** the presence of a **trash** buffer

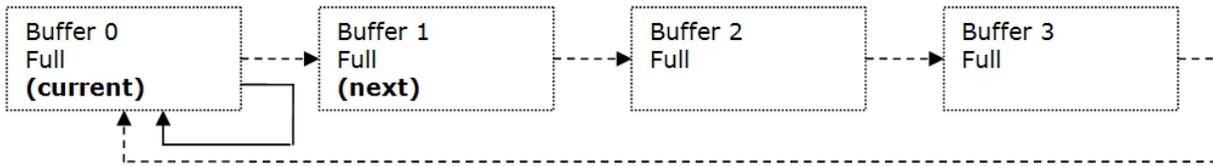
Example of **case 1**:



Example of **case 2**:



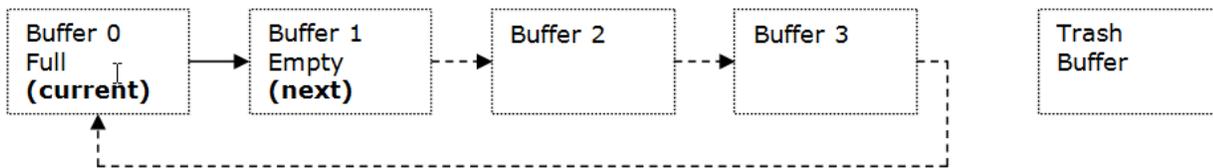
Example of **case 3**:



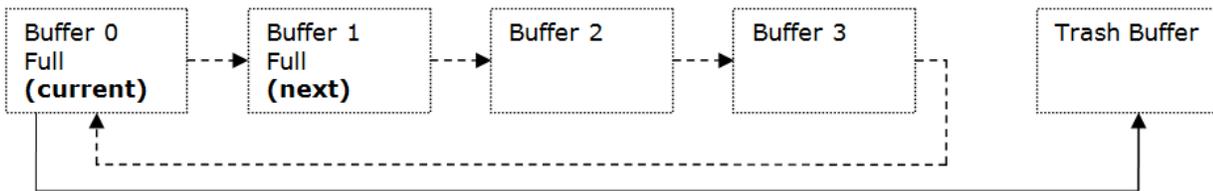
In the **Synchronous With Trash** mode:

- **(Case 1)** If the **next** buffer is **empty**, then transfer to the **next** buffer
- **(Case 2)** If the **next** buffer is **full**, then transfer to the **trash** buffer
- **(Case 3) Repeat** transferring to the **trash** buffer as long as the **next** buffer is **full**
- Buffer **state** is **irrelevant** for the **trash** buffer

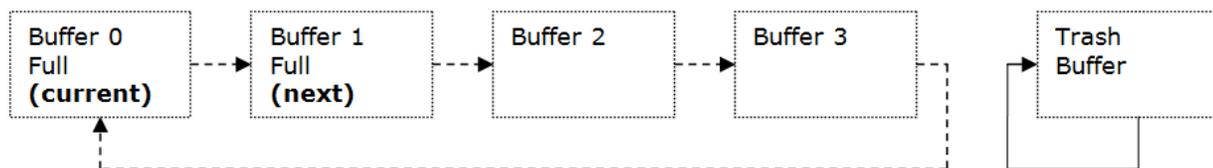
Example of **case 1**:



Example of **case 2**:



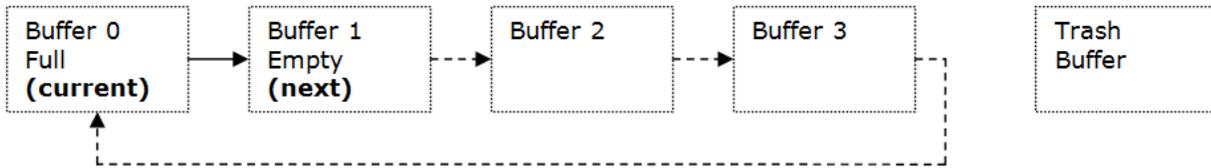
Example of **case 3**:



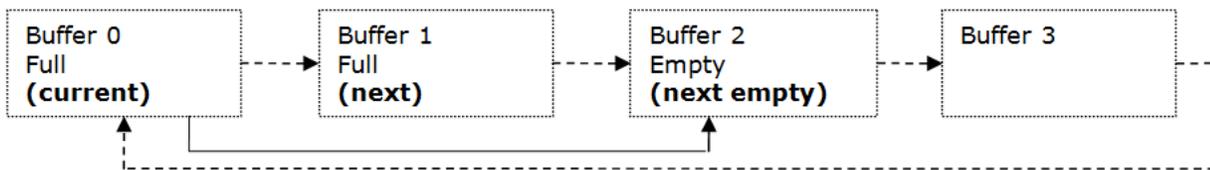
In the **Synchronous Next Empty With Trash** mode:

- **(Case 1)** If the **next** buffer is **empty**, then transfer to the **next** buffer
- **(Case 2)** If the **next** buffer is **full**, transfer to **next empty** buffer in the list
- **(Case 3)** If **all** buffers are **full**, then transfer to **trash** buffer
- **(Case 4)** **Repeat** transferring to the **trash** buffer as long as **all** buffers are **full**
- Buffer state is irrelevant for the trash buffer

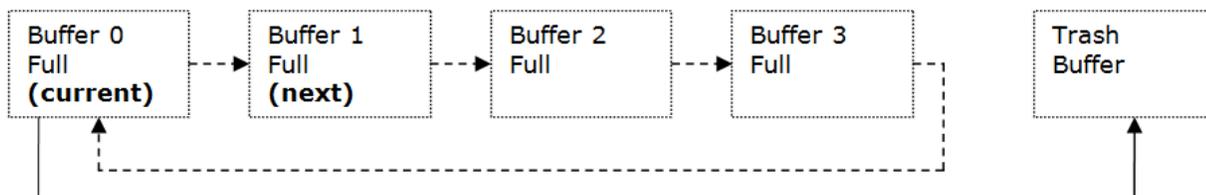
Example of **case 1**:



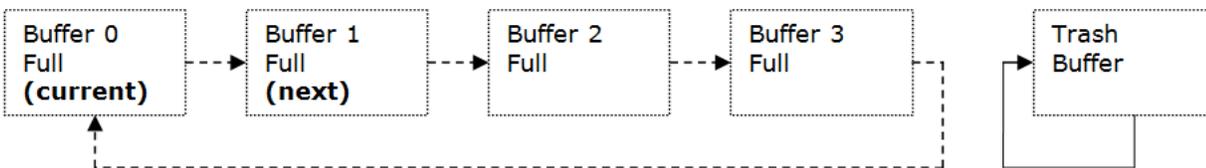
Example of **case 2**:



Example of **case 3**:



Example of **case 4**:



API support for managing the transfer cycling mode:

- **(Sapera LT ++)** Use the **GetCycleMode** and **SetCycleMode** methods in the SapXferPair class
- **(.NET)** Use the **Cycle** property in the SapXferPair class

Execution flow for processing and displaying images

Example 1: The application only needs to read time stamp information of acquired images

With **Sapera LT ++**:

- The application transfer callback function gets called
- This function calls the GetCounterStamp function of the SapBuffer class to retrieve the time stamp, and returns
- The buffer state is automatically set to empty by the SapTransfer class

With the **.NET** API:

- The application handler function for the XferNotify event of the SapTransfer class gets called
- This function reads the CounterStamp property of the SapBuffer class to retrieve the time stamp, and returns
- The buffer state is automatically set to empty by the SapTransfer class

Example 2: The application only needs to **process** acquired images (**no display**)

With **Sapera LT ++** (initialization):

- Create a new class derived from SapProcessing, the overridden Run function of this class will handle the actual processing
- Call the SetAutoEmpty(FALSE) method of the SapTransfer class to prevent buffers from being set to empty in this class
- Call the SetAutoEmpty(TRUE) method of the SapProcessing class to set buffers to empty in this class

With **Sapera LT ++** (after each acquired image):

- The application transfer callback function gets called
- This function calls the Execute function of the SapProcessing class, and returns
- This eventually calls the overridden Run function in the application
- This function performs the actual processing on the image, and returns
- The buffer state is automatically set to empty by the SapProcessing class
- The application processing callback function (if any) gets called

With the **.NET** API (initialization):

- Create a new class derived from SapProcessing, the overridden Run function of this class will handle the actual processing
- Set the AutoEmpty property of the SapTransfer class to False to prevent buffers from being set to empty in this class
- Set the AutoEmpty property of the SapProcessing class to True to set buffers to empty in this class

With the **.NET** API (after each acquired image):

- The application handler function for the XferNotify event of the SapTransfer class gets called
- This function calls the Execute function of the SapProcessing class, and returns
- This eventually calls the overridden Run function in the application
- This function performs the actual processing on the image, and returns
- The buffer state is automatically set to empty by the SapProcessing class
- The application handler function for the ProcessingDone event of the SapProcessing class (if any) gets called

Example 3: The application needs to **process** acquired images before **displaying** the resulting processed images

With **Sapera LT ++** (initialization):

- Create a new class derived from SapProcessing, the overridden Run function of this class will handle the actual processing
- Call the SetAutoEmpty(FALSE) method of the SapTransfer class to prevent buffers from being set to empty in this class
- Call the SetAutoEmpty(FALSE) method of the SapProcessing class to prevent buffers from being set to empty in this class
- Call the SetAutoEmpty(TRUE) method of the SapView class to set buffers to empty in this class

With **Sapera LT ++** (after each acquired image):

- The application transfer callback function gets called
- This function calls the Execute function of the SapProcessing class, and returns
- This eventually calls the overridden Run function in the application
- This function performs the actual processing on the image, and returns
- The application processing callback function gets called
- This function calls the Show function of the SapView class to display the processed image, and returns
- After image display is done, the buffer state is automatically set to empty by the SapView class
- The application view callback function (if any) gets called

With the **.NET** API (initialization):

- Create a new class derived from SapProcessing, the overridden Run function of this class will handle the actual processing
- Set the AutoEmpty property of the SapTransfer class to False to prevent buffers from being set to empty in this class
- Set the AutoEmpty property of the SapProcessing class to False to prevent buffers from being set to empty in this class
- Set the AutoEmpty property of the SapView class to True to set buffers to empty in this class

With the **.NET** API (after each acquired image):

- The application handler function for the XferNotify event of the SapTransfer class gets called
- This function calls the Execute function of the SapProcessing class, and returns
- This eventually calls the overridden Run function in the application
- This function performs the actual processing on the image, and returns
- The application handler function for the ProcessingDone event of the SapProcessing class gets called
- This function calls the Show function of the SapView class to display the processed image, and returns
- The buffer state is automatically set to empty by the SapView class
- The application handler function for the DisplayDone event of the SapView class (if any) gets called

SapFlatField Coefficient Calibration

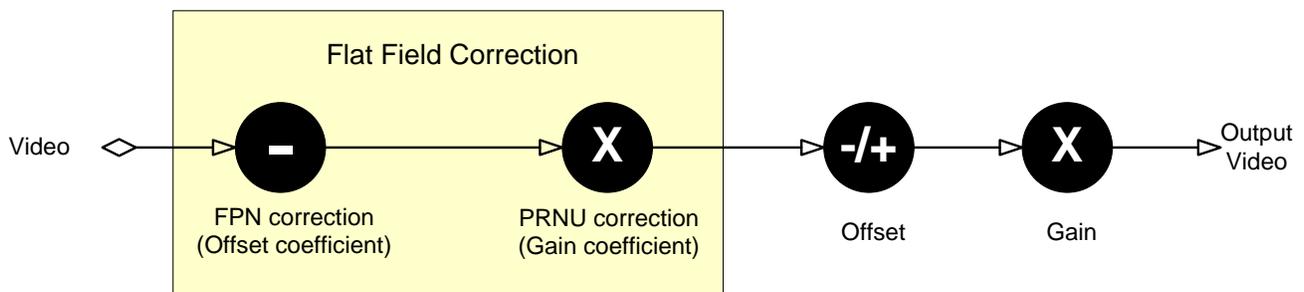
A number of Teledyne DALSA frame grabbers and cameras support hardware flat field calibration. The following section provides an overview of how to use the Sopera LT SapFlatField class to perform flat field calibration.

Flat field correction uses 2 coefficients (offset and gain) per pixel to compensate for fixed pattern noise (FPN) and photo response non-uniformity (PRNU).

- **FPN** is the variation in pixel response without incident light (also known as dark current). It is noise signal generated by the background voltage present in the sensor. The flat field **offset** coefficients are used to correct for this noise. To perform FPN calibration using **SapFlatField::ComputeOffset**, a number of dark images are averaged (i.e, all light is blocked from entering the sensor using the lens cap). The percentage of zero pixels allowed in the averaged images can be set using the SapFlatField::SetBlackPixelPercentage (too many zero pixels indicates the camera's black level is too high and information is being clipped; adjust the camera settings accordingly).
- **PRNU** is the variation in pixel response to a uniform amount of light. The flat field **gain** coefficients are used to correct for this response non-uniformity such that all pixels output the same value when exposed to the same incident light. To perform PRNU calibration using **SapFlatField::ComputeGain**, a number of white images are averaged, such that the camera is close to, but not at saturation, is used. The gain coefficient is calculated for each pixel such that it reaches a specified target value below saturation.

For both FPN and PRNU calibration, the greater the number of images averaged reduces the effects of random noise.

Typical Digital Processing Chain
(monochrome)



The ComputeOffset function must be called before the ComputeGain function. To apply the software flat field correction on an image, use the **SapFlatField::Execute** function. For hardware flat field correction, the flat field correction file is loaded to the device and enabled on the hardware; refer to the device documentation for more information.

The system offset and gain applied after the flat field correction are typically used to maximize the image dynamic range for the typical image scene for the application.

Flat Field File Format

Flat field calibration creates an 8 or 16-bit TIF file that contains the offset and gain coefficients. The buffer is the same width as the acquired image but twice the height of the acquired image height (the first half contains the offset coefficients, the second half the gain coefficients).

The 8 or 16 bit format is determined by the format of the buffer passed to the SapFlatField::ComputeOffset / ComputeGain functions. 16-bit files are used for 10, 12, 14, or 16 bit output format. In general, the sensor's highest output format should be used to calibrate the flat field coefficients. A 16-bit flat field coefficient file can be used with lower output formats by setting an offset factor (**SapFlatField::SetOffsetFactor**).

TIFF File Structure as used by Sopera LT

- TIFF header — as per TIFF 6.0 specification
- Image data width is same as acquired image width
- Image data height is twice the acquired image height
- Upper half of image is offset data
- Lower half is gain data
- Image pixel format is same as acquired image format

Flat Field Correction Formula

For each pixel, flat-field correction is performed according to the following formula:

$$\text{correctedValue} = (\text{originalValue} - \text{offset}) * (\text{gain} / \text{gainDivisor} + \text{gain base})$$

Gain Divisor

For 8-bit gain coefficients, the gain divisor is typically equal to 128, so that a gain value between 0 and 255 becomes a value between 0 and 2. It is then set to the acquisition device gain divisor value when calling the Create method (the SapFlatField::SetGainDivisor method is only used when operating without hardware support). The gainDivisor and gain base are used to convert a floating point gain value to an integer value that can be saved in a .TIFF image.

Gain Base

For gain base, if supported by the acquisition device (for example, the Genie TS), it is retrieved from the device after calling the Create method. For all other acquisition devices, and for software based flat field correction, the initial value for this attribute is 0, and the application code can call SetGainBase if required.

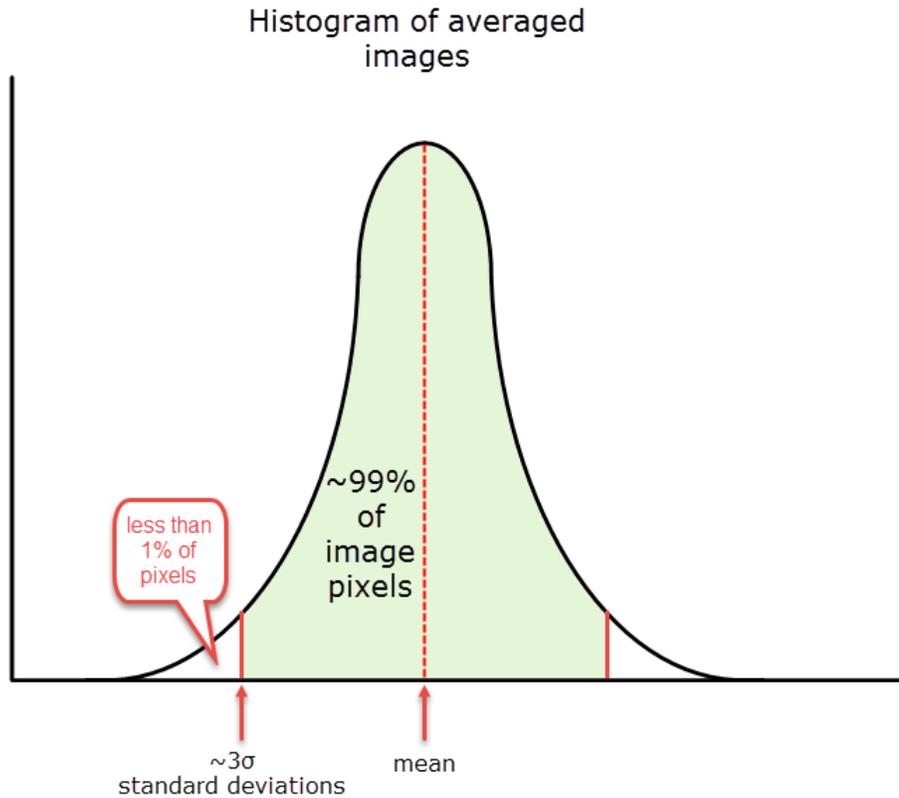
Offset Coefficients

The offset coefficient for FPN correction is calculated on a per pixel basis using the average pixel value at X_n, Y_n minus the DN value corresponding to the lower bound representing less than 1% of the pixel distribution (calculated using $\sim 3\sigma$ standard deviations from the histogram mean of the averaged black images).

Offset coefficient $(X_n, Y_n) = \text{average pixel value } (X_n, Y_n) - \text{DN value of } \sim 3\sigma$.

This method preserves the dynamic range and reduces the number of pixels that are clipped at zero (which results in loss of image data, even if offset and gain are subsequently applied to adjust the black threshold).

The **SapFlatField::SetOffsetMinMax** can be used to limit the possible gain values. If pixels reach this limit, they are flagged as defective when **SapFlatField::EnableClippedGainOffsetDefects** = TRUE (default).



Offset coefficient $(X_n, Y_n) = \text{average pixel value } (X_n, Y_n) - \text{DN value of } \sim 3\sigma$

Gain Coefficients

Gain coefficients are calculated after offset coefficients are applied. Gain coefficients are calculated such that all pixels reach the specified target value (or the maximum pixel value in the white image). The **SapFlatField::SetGainMinMax** can be used to limit the possible gain values. If pixels reach this limit, they are flagged as defective when

SapFlatField::EnableClippedGainOffsetDefects = TRUE (default).

Pixel Replacement

For the black and white images, pixel values higher/lower than
(*average pixel value +/- deviationMax*)

are considered as defective pixels. By default, the maximum deviation is 0.25 x maximum pixel value (for example for 8-bit images the maximum deviation is 63).

The maximum deviations for the black and white images are set using the SapFlatField::SetDeviationMaxBlack / SetDeviationMaxWhite functions.

Pixel replacement is enabled/disabled using **SapFlatField::EnablePixelReplacement**. Pixels are replaced using the pixel to its immediate left, other than the first pixel of a line, which uses the pixel to the right.

To calibrate the camera's flat field coefficients:

1. Configure the camera to the required frame rate and exposure timing, plus adjust the light level for normal operation. If used, any horizontal or vertical binning should also be applied.
2. The lens should be at the required magnification and aperture and slightly unfocused to avoid introducing granularity or details in the reference image (when calibration is complete, refocus the lens).
3. As the white reference is located at the object plane, any markings or contaminants on its surface (that is, dust, scratches, smudges) will end up in the calibration profile of the camera. To avoid this, use a clean white plastic or ceramic material rather than trying to rely on a paper reference. (Ideally, the white object will be moving during the calibration process, as the averaging process of the camera will diminish the effects of any small variation in the white reference.)
4. Adjust the system gain until the peak intensity is at the desired DN level and then calibrate the fixed pattern noise (FPN) using the SapFlatField::ComputeOffset function. Use a lens cap to ensure that no light reaches the sensor.
5. Once complete, remove the lens cap and perform a photo response non-uniformity (PRNU) calibration using SapFlatField::ComputeOffset using the desired target value (in DN) . You want all the pixels to match. This target value should be higher than the peak values you saw while first setting up the camera.
6. The system gain remains as first set.

Code Samples using Sapera LT

The following C++ code sample shows Flat Field manual calibration using Sapera LT.

```
// Rely on the SapFlatField class to automatically create the offset and gain buffers with the
// correct dimensions and format, but perform the calibration manually
// pAcquisition is an existing SapAcquisition object
// pBuffer is an existing SapBuffer object containing an acquired image.

SapFlatField* pFlatField = new SapFlatField(pAcquisition);
BOOL success = pFlatField->Create();

SapBuffer* pBufferOffset = pFlatField->GetBufferOffset();
SapBuffer* pBufferGain = pFlatField->GetBufferGain();

// Can also use the following:
//     int bufWidth = pBufferOffset->GetWidth();
//     int bufWidth = pBufferGain->GetWidth();

int bufWidth = pBuffer->GetWidth();

// Can also use the following:
//     int bufHeight = pBufferOffset->GetHeight();
//     int bufHeight = pBufferGain->GetHeight();

int bufHeight = pBuffer->GetHeight();

// Can also use the following:
//     int bufFormat = pBufferOffset->GetFormat();
//     int bufFormat = pBufferGain->GetFormat();

int bufFormat = pBuffer->GetFormat();

// This is for 8-bit buffers.

BYTE* pBufData;
success = pBuffer->GetAddress(&pBufData);

BYTE* pOffsetData;
success = pBufferOffset->GetAddress(&pOffsetData);

BYTE* pGainData;
success = pBufferGain->GetAddress(&pGainData);

int gainDivisor = pFlatField->GetGainDivisor();

// Code to perform manual calibration using pBufData, pOffsetData,
// pGainData, and gainDivisor goes here
// ...

success = pFlatField->Destroy();
delete pFlatField;
```

Deploying a Sapera Application

When your application is ready to be delivered, you may need to create a procedure that will install the appropriate component to the target system. The following sections detail the tasks that your installation program needs to perform.

Runtime Installations

Two types of Sapera LT runtime installations are available when deploying your application:

- Sapera LT only
- Sapera LT with CamExpert

The type of runtime installation required depends on whether your application requires the CamExpert utility.

The appropriate device driver must be installed along with the installation of the Sapera LT runtimes. This topic is further discussed in this section.

Installing Sapera LT Runtimes and Sapera LT Compatible Drivers

The Sapera LT Installation Program automatically handles the task of copying files and making Windows Registry changes, as dictated by your selection of Sapera LT (full development or runtime components only) and the selected device drivers. When integrating Sapera LT and the Sapera LT device driver installations into your system, Teledyne DALSA suggests that the Sapera LT Install Program be invoked within your own software installation program.

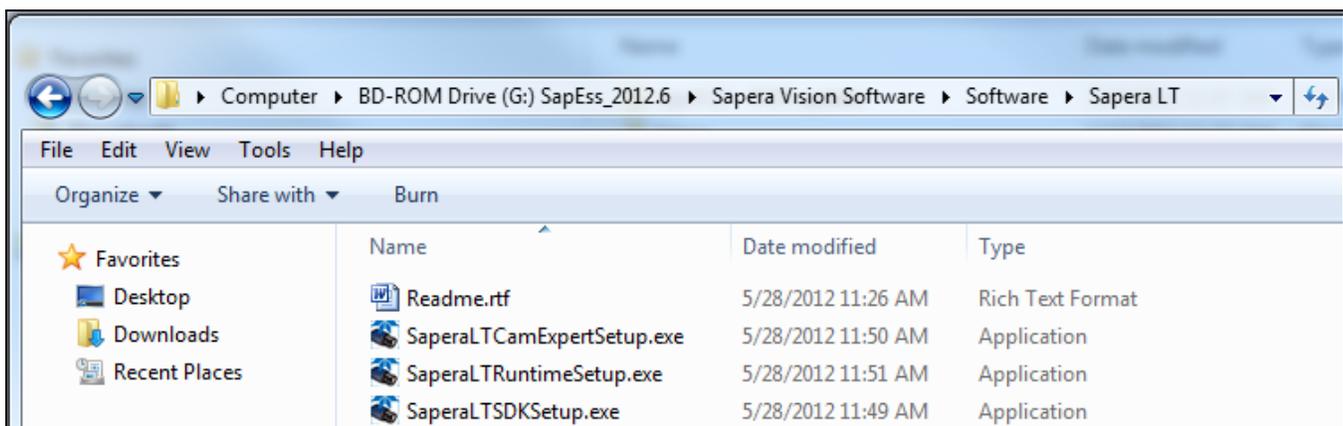
The Sapera LT installation is supplied in three forms:

- Developer full installations
- Runtime only installation
- CamExpert tool only installation

The Developer installation is accessible from the Sapera LT DVD browser and contains all Sapera LT components (demos, tools, documentation, and the Sapera LT runtimes).

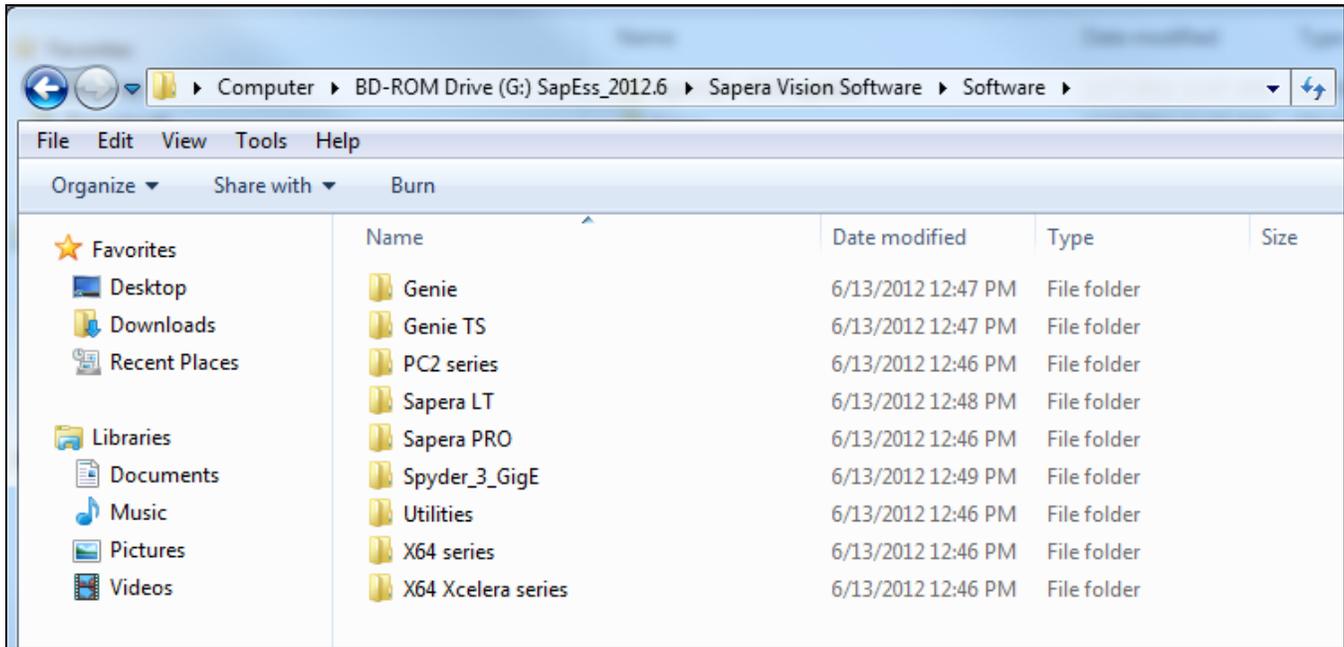
The Runtime version contains only the components required to execute your application. It is recommended that you install the Runtime components only on systems not used for development.

You will find the Sapera LT installations in the **Sapera Vision Software\Software\Sapera LT** directory on the Sapera Essential, Sapera Architect, or Sapera Nitrous distribution disk.



Unlike the Sapera LT installation, there is only one way to perform Sapera LT driver installations. A driver installation contains the device driver, the device's user's manual and, in some cases, some device specific demos.

Frame grabber product drivers or GigE Vision camera framework packages are found on the Sapera distribution disk within their own folders as shown in the following screen image.



Teledyne DALSA Installers

Both Sapera LT and the driver installations share the same installer technology. As a result, the following discussion applies to both.



Note: You must reboot after the installation of Sapera LT. However, to streamline the installation process, you may install Sapera LT (without rebooting), the required device drivers and then reboot.

Teledyne DALSA's installers can be started in two ways:

1. Normal Mode
This is the interactive mode provided by default. It is initiated by invoking the **setup.exe** program. The installation proceeds normally as if it was started from Windows Explorer or the Windows command line.
2. Silent Mode
This mode requires no user interaction. Any user input is provided through a "response" file. Nothing is displayed by the installer.



Note: During driver installation, Windows Digital Signature and Logo Testing warnings can be safely ignored.

Silent Mode Installation

Silent Mode installation is recommended when integrating Teledyne DALSA products into your software installation. The silent installation mode allows the Sopera LT installation to proceed without the need for mouse clicks or other input from a user.

Two steps are required:

- Preparation of a response file to emulate a user.
- Invoking the Sopera LT installer with command options to use the prepared response file.

Creating a response file

The installer response file is created by performing a Sopera LT installation with a command line switch "-r". The response file is automatically named `setup.iss` which is saved in the `\windows` folder. One simple method is to execute the Sopera LT installer from within a batch file. The batch file will have one command line.

As an example, the command line is:

```
SoperaLTSDKSetup -r
```

Running a Silent Mode Installation

A Sopera LT silent installation, whether done alone or within a larger software installation requires the Sopera LT executable and the generated response file `setup.iss`.

Execute the Sopera LT installer with the following command line:

```
SoperaLTSDKSetup -s -f1".\setup.iss"
```

where the `-s` switch specifies the silent mode and the `-f1` switch specifies the location of the response file. In this example, the switch `-f1".\setup.iss"` specifies that the `setup.iss` file is in the same folder as the Sopera LT installer.



Note: For Sopera LT 8.10, the installation process has been modified; a new prompt has been added for installing 'Teledyne Dalsa frame grabbers and CameraLink cameras' only, 'GigE-Vision cameras and the Sopera Network Imaging Package' only, or 'All acquisition components'. Therefore existing response files for previous versions need to be updated and replaced.

Installer Error Codes

The following table describes the error codes returned by the installer during a silent installation.

Return code	Description
0	Success
-1	General error
-2	Invalid mode
-3	Required data not found in the Setup.iss file (response file)
-4	Not enough memory available
-5	File does not exist
-6	Cannot write to response file
-7	Unable to write to the log file
-8	Path to the InstallShield silent response file is not valid
-9	Not a valid list type (string or number)
-10	Data type is not valid
-11	Unknown error during set up
-12	Dialog boxes are out of order
-51	Cannot create the specified folder
-52	Cannot access the specified file or folder
-53	A selected option is not valid

Silent Mode Uninstall

Similar to a silent installation, a response file must be prepared first as follows.

Creating a Response File

The installer response file is created by performing a software un-installation with a command line switch "-r". The response file is automatically named **setup_uninstall.iss** which is saved in the \windows folder. If a specific directory is desired, the switch "-f1" is used.

As an example, to save a response file in the same directory as the installation executable of the Sapera LT SDK, the command line would be:

```
SaperaLTSDKSetup.exe -r -f1".\setup_uninstall.iss"
```

Running a Silent Mode Uninstall

Similar to the device driver silent mode installation, the un-installation requires the device driver executable and the generated response file **setup.iss**.

Execute the Sapera LT SDK installer with the following command line:

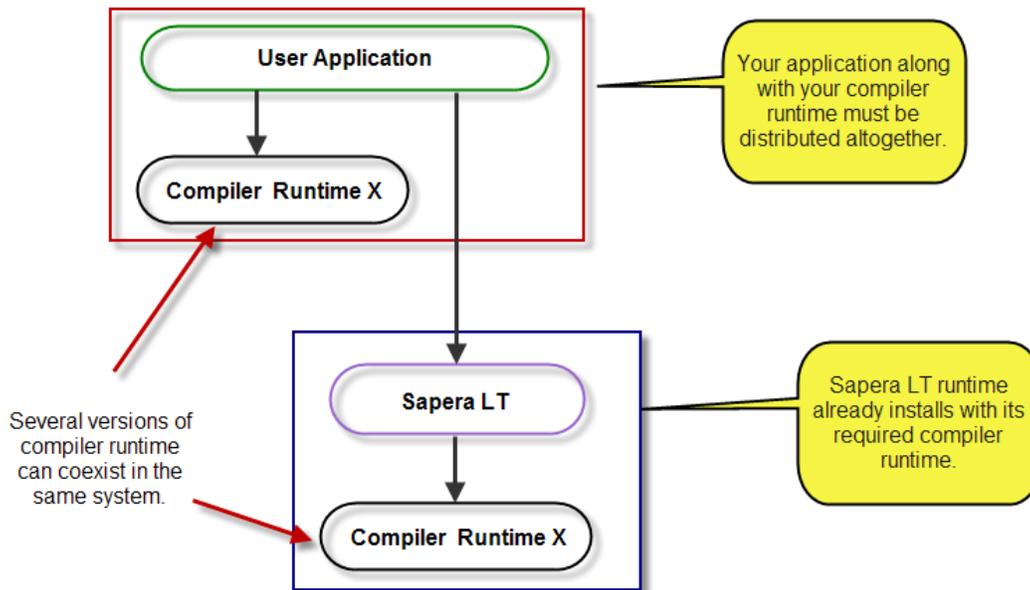
```
SaperaLTSDKSetup.exe -s -f1".\setup_uninstall.iss"
```

Where the **-s** switch specifies the silent mode and the **-f1** switch specifies the location of the response file. In this example, the switch **-f1".\setup_uninstall.iss"** specifies that the **setup_uninstall.iss** file be in the same folder as the software installer.

Compiler Run-time Redistribution

When deploying your application you also need to redistribute the runtime required by the compiler used for generating your application. In fact, the Sapera LT runtime installation program does install the version of compiler runtime used by Sapera LT libraries but not necessarily the one used by your application.

The diagram below shows the runtime architecture dependency. "Compiler runtime X" provides support for the calls made by your application to the standard compiler libraries (direct calls) while "Compiler runtime Y" provides support for the calls made by Sapera LT library to the standard compiler libraries (indirect calls). Several compiler versions can coexist in the same target system.



Contact Information



TELEDYNE DALSA
Everywhere you look™

The following sections provide sales and technical support contact information.

Sales Information

Visit our web site:

www.teledynedalsa.com/en/contact/contact-sales/

Email:

<mailto:info@teledynedalsa.com>

Technical Support

Submit any support question or request via our web site:

Technical support form via our web page:

Support requests for imaging product installations

Support requests for imaging applications

Camera support information

Product literature and driver updates

www.teledynedalsa.com/en/support/options/

When encountering hardware or software problems, please have the following documents included in your support request:

- The Sopera Log Viewer .txt file
- The PCI Diagnostic PciDiag.txt file (for frame grabbers)
- The Device Manager BoardInfo.txt file (for frame grabbers)



Note, the Sopera Log Viewer and PCI Diagnostic tools are available from the Windows start menu shortcut **Start•All Programs•Teledyne DALSA•Sopera LT**. The Device Manager utility is available as part of the driver installation for your Teledyne DALSA device and is available from the Windows start menu shortcut **Start•All Programs•Teledyne DALSA•<Device Name>•Device Manager**.