PixeLINK

MAKING DIGITAL IMAGING SIMPLE

PixeLINK™ Camera API Reference

Version 4.0

PixeLINK
3030 Conroy Road • Ottawa, Ontario • K1G 6C2 • Canada • Tel: (613) 247-1211 • Fax: (613) 247-2001
Email: info@pixelink.com • Web Site: www.pixelink.com

PixeLINK™ Camera API Reference

Copyright Notice

Version 4.0

By purchasing this product, the Purchaser(s) and/or any subsequent legitimate owner(s) of the product, henceforth referred to as "the Purchaser," agree(s) to abide by the terms of this Agreement and read and recognize the following set of definitions appertaining to the intellectual-property items and trademark references as can be found throughout this API Reference.

PixeLINK provides the information and data included in this document for the Purchaser's benefit, but it is not possible for PixeLINK to entirely verify and test all of this information in all circumstances, particularly information relating to non-PixeLINK manufactured products. PixeLINK makes no warranties or representations relating to the quality, content, or adequacy of this information. Every effort has been made to ensure the accuracy of this manual; however, PixeLINK assumes no responsibility for any errors or omissions in this document. PixeLINK shall not be held liable for any errors or for incidental or consequential damages in connection with the furnishing, performance, or use of this API Reference or the examples herein.

PixeLINK assumes no responsibility for any damage or loss resulting from the use of this API Reference, loss or claims by third parties which may arise through the use of this product, any damage or loss caused by deletion of data as a result of malfunction or repair, or any other damage related to the use of this product or associated documentation. The information in this document is subject to change without notice.

Definitions of Intellectual Property and Trademark Attributions

This Section is intended to ensure proper attribution and honoring of any and all trademarks and intellectual-property items in terms of attribution to their respective owners as mentioned in this API Reference. The reader is encouraged to consult this Section whenever uncertainty presents itself as to the terms, their meaning within the API Reference, and the trademarks and intellectual-property items they stand to identify, whether by themselves or in conjunction with other terms and items.

PixeLINK is either a trademark or a registered trademark of PixeLINK in Canada and/or other countries; *IEEE* is a registered trademark or service mark of the Institute of Electrical and Electronics Engineers, Incorporated in the United States and/or other countries; *FireWire* is a trademark of Apple Computer, Inc., registered in the U.S. and other countries; *Microsoft* and *Windows* are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries; *TIFF* is a trademark or registered trademark of Adobe Systems Incorporated in the United States and/or other countries. All other products, brand names, company names are trademarks or registered trademarks of their respective owners.

Definition of Terms

This Section is intended to define certain terminology used in this API Reference, while ensuring proper attribution and honoring of any and all trademarks and intellectual-property items in terms of attribution to their respective owners as mentioned in this API Reference.

The Purchaser shall hereby recognize the following definitions set herein, as can be found throughout this API Reference: *Camera* shall henceforth refer to a PixeLINK FireWire Camera; *API* shall henceforth refer to the PixeLINK Camera Application Programming Interface; *Kit* shall henceforth refer to a PixeLINK Camera Kit; *FireWire* shall henceforth refer to the IEEE 1394a interface specification; *Windows* shall henceforth refer to a Microsoft Windows operating system.

The above Sections set forth Terms and Conditions, compliance with which constitutes a mandatory prerequisite for owning and/or using the product for which the manual was created. It is the Purchaser's responsibility to ensure that the information contained within the Sections is maintained as a part of the API Reference at all times—should the Purchaser discover that the page(s) containing the Sections is (are) missing, and/or was not provided with the API Reference, and/or become illegible, PixeLINK should be contacted as soon as possible and the Sections requested. PixeLINK shall not be held liable for any and all copyright violations that may ensue in relation to its products and/or the consequences of their intended and unintended usage.October 2003
Part Number: 04326-01

Table of Contents

List of Figures

List of Tables

1

Introduction

1.1 The PixeLINK Camera Application Programming Interface, Version 4.0

The PixeLINK Camera Application Programming Interface (API) offers application software developers a means to adapt existing programs or develop new imaging applications for PixeLINK cameras. It allows rapid development of custom applications for camera operation by simplifying the most common tasks associated with configuring and controlling the cameras.

The PixeLINK Camera API Version 4.0 is a dynamic link library (DLL) that provides:

- A powerful, easy to use control interface
- A generic command set that can be used to control different camera models
- Fast and flexible access to streaming video
- The ability to save high quality still images and video clips

The API also provides a consistent programming interface regardless of the electrical interface of the product (that is, FireWire or USB). Functions may be called from C/C++ or Visual Basic.

Note that the PixeLINK API is designed to control PixeLINK cameras only.

1.2 Topics Covered in this Manual

This manual provides a reference for the PixeLINK Camera API and related software.

Sections include:

- Basics principles of using the PixeLINK Camera API—Section ☐ (page 2)
- A summary of the PixeLINK Camera API functions—Section 3.1 (page 12)
- Descriptions of individual API functions—Section 3.2 (page 16)
- Sample code—Section 5 (page 65)

Most references in the PDF version of this manual are hyperlinked for easy navigation and access.

Note:

This manual assumes that the reader has a basic understanding of the C language.

1.3 Compatibility with Other PixeLINK Products

Applications created with the PixeLINK Camera API can be used with any Version 4.0 PixeLINK camera using either a FireWire or a Hi-Speed USB interface.

To convert programs created with an earlier version of the PixeLINK Camera API, refer to Appendix E (on page 79) for a table of function conversions and equivalences.

1.4 Related PixeLINK Documentation

- **PixeLINK Camera User's Manual**

  This manual describes the functionality of the PixeLINK camera hardware and software, including the *PixeLINK Developers Application*.

  Users should consult the User's Manual

  … if this is the first time installing FireWire or USB hardware
  … before installing PixeLINK software
  … as a guide when using the *PixeLINK Developers Application*

  Sections include:

  - Installation of the camera hardware and PixeLINK software
  - Features and operation of the *PixeLINK Developers Application*

- **PixeLINK Camera System Guide**

  This Guide provides a reference for camera's hardware and connectors, and the control options available through IIDC 1.3.

  Users should consult this Guide …

  … before using the camera with an IIDC-compliant application
  … when mounting the camera within a system or enclosure
  … when planning to use an external trigger or GPO device (e.g., strobe, pulse) with the camera

  Sections include:

  - A description of the camera hardware
  - An overview of IIDC compliance and features available with the camera
  - A list of available triggering and GPO modes
  - IIDC features
  - A description of the camera's operating (exposure) modes and shutter types

2

API Usage

2.1 Basic Strategy

Controlling a PixeLINK camera with the PixeLINK API, from connection of the camera until disconnection, follows this basic strategy:

1. Select and initialize a camera—Section 2.2 (page 3)
2. Determine and set camera features—Section 2.3 (page 4)
3. Retrieve image data (performing changes to the feature settings as required)—Section 2.4 (page 5)
4. Uninitialize the camera—Section 2.5 (page 10)

See the PixeLINK Camera System Guide for limitations on the number of cameras that can be connected or operated at a time.

Note:

A camera cannot be controlled by more than one application at once.

2.2 Selecting and Initializing the Camera

The API functions allow an application to control multiple cameras simultaneously.

1. Use the PxLGetNumberCameras function (on page 32) to determine the number of the cameras connected to the system and the serial number of each camera.
2. Use the PxLInitialize function (on page 33) to initialize and access a specific camera, identified by its serial number. The PxLInitialize function returns a "handle" for the camera, which is used to identify it in other function calls. Repeat for the desired number of cameras.

2.3 Determining, Setting, and Getting Camera Features

Each camera supports a subset of the list of available camera features (on page 52). The specific features supported by the camera are encoded in the camera hardware.

The PxLGetCameraFeatures function (on page 23) allows the application to enumerate the list of features applicable to the camera and the range of values appropriate for the each feature.

Camera features and their properties are passed as parameters in two generic functions, PxLSetFeature (on page 40) and PxLGetFeature (on page 29). These functions access the camera to set and get the camera feature properties.

1. Use the PxLGetCameraFeatures function (on page 23) to enumerate the features of a specified camera. (PxLGetCameraFeatures also returns the ranges of these features.)
2. Use the PxLSetFeature (on page 40) and PxLGetFeature (on page 29) functions to set and get settings and properties of specific functions on the camera.

Prior to capturing images, be sure to use PxLSetFeature (on page 40) to set the camera to return the desired pixel format (for example, PIXEL_FORMAT_RGB24 or PIXEL_FORMAT_BAYER16). For more information on pixel formats, see page 61.

Applying and Managing Sets of Camera Feature Properties

Feature properties stored in non-volatile memory:

Sets of camera feature properties can be stored in and loaded from the camera's non-volatile memory using PxLSaveSettings (on page 36) and PxLLoadSettings (on page 34).

Sets of feature properties returned during image acquisition:

In addition to the image data, PixeLINK API's the image acquisition process returns the set of camera feature properties applied to each frame and stores them in a buffer or in the PixeLINK data stream file. This set of properties, called a **descriptor**, can be passed from function to function for informative purposes.

Applying sets of feature properties automatically on a frame-by-frame basis:

Using advanced features in the API, **user-defined descriptors** can be created and loaded into the camera. Multiple descriptors can be defined and applied automatically on a frame-by-frame basis.

For general information about descriptors, see Section 2.6 (on page 10). For information about descriptor advanced functions, see Appendix D (on page 70).

2.4 Retrieving Image Data from the Camera

After the camera has been initialized and the camera features determined and set appropriately, it is ready for streaming and capturing image data.

(Prior to capturing images, be sure to use PxLSetFeature (on page 40) to set the camera to return the desired pixel format (for example, PIXEL_FORMAT_RGB24 or PIXEL_FORMAT_BAYER16). For more information on pixel formats, see page 61.)

Capture Methods

Using the PixeLINK API, there are four methods of capturing frames or video clips from a camera. These methods are characterized in Table 2.1 (below).

Table 2.1 Capture Methods

Method	Capture Type	Shutter Type	Use of Other Sync. Device Possible? (e.g., Strobe, Flash)
Free-Running Continuous (default method)	Continuous	Rolling	No
Internally Triggered Free-Running Continuous	Continuous	Synchronous	No
Hardware Triggered Frame-on-Demand	Frame-on-Demand	Synchronous	Yes
Software Triggered Frame-on-Demand	Frame-on-Demand	Synchronous	Yes

Not all methods can be used by all camera models. Use the PxLGetCameraFeatures function (on page 23) to check the trigger features of a specific camera.

Capture Type

Continuous

Image data is transmitted continuously from the camera to the host computer.

Frame-on-Demand

No image data is captured until a frame is requested using a trigger signal (hardware or software).

Shutter Type

Rolling Shutter

With a Rolling Shutter, only a few rows of pixels are exposed at one time. The camera builds a frame by reading out the most exposed row of pixels (and ceasing exposure of

that row), starting exposure of the next unexposed row down in the **Region of Interest** (**ROI**; the user-specified active area on the imager), then repeating the process on the next most exposed row and continuing until the frame is complete. After the bottom row of the ROI starts its exposure, the process "rolls" to the top row of the ROI to begin exposure of the next frame's pixels.

The exposure down each frame, and from frame-to-frame, remains consistent due to this continuous read-out.

The row read-out rate is constant, so the longer the exposure setting, the greater the number of rows being exposed, or **integrated**, at a given time. ("Integrated" means that the pixels are building up, or integrating, an electrical charge in response to the photons hitting them.) Rows are added to the exposed area one at a time. The more time that a row spends being integrated, the greater the electrical charge built up in the row's pixels and the brighter the output pixels will be. As each fully exposed row is read out, another row is added to the set of rows being integrated.

Example: A very short exposure may be obtained by having only three rows of integration (see Figure 2.1, page 6). This means that as each row is being read out, the three rows ahead of it are being exposed. As each row is read out, another row is added to the group of rows being integrated.

Note that the read-out direction is *reversed* when the camera's "vertical flip" mode is enabled (see the PxLSetFeature function on page 40). In this case, the read-out happens from the bottom to the top of the ROI, rather than from top to bottom.

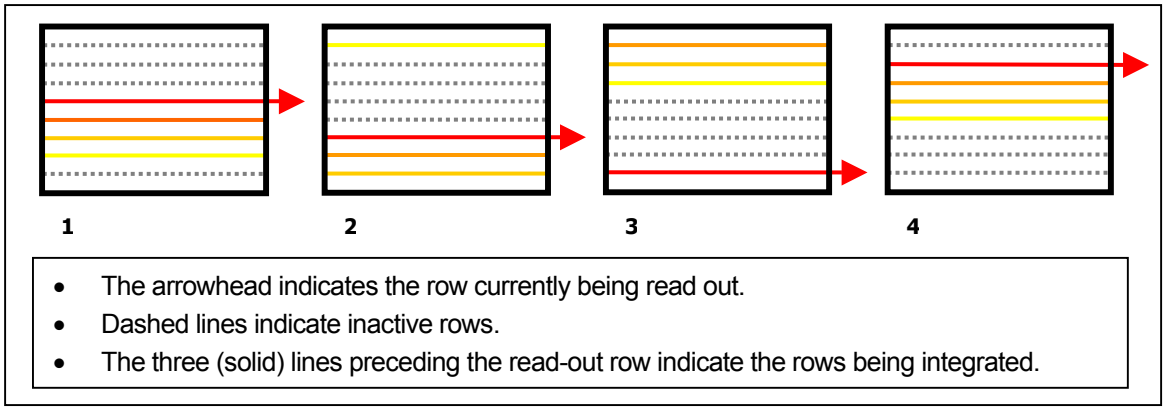

Figure 2.1 Rolling Shutter Integration and Read-Out

Frames are kept in a circular buffer—that is, one in which the oldest frame is constantly being overwritten with new frame data—until the camera receives request for image data. At that point, the frame data is transmitted to the host computer, starting with the oldest frame in the buffer.

Because Rolling Shutter exposes rows in the integration area while reading out fully exposed rows (that is, it does not stop exposure to perform read-out), it provides evenly exposed image data with the greatest possible speed (under the given parameters). Use

it when a continuous sequence of frames is required, such as in the capture of smooth video clips.

Each row of pixels has a slightly different exposure start and end times from the adjacent rows, so Rolling Shutter can produce a jagged or blurred effect in fast-action images unless the exposure time is sufficiently short. Under this condition, consider using Synchronous Shutter if the circumstances allow it.

Synchronous Shutter

With Synchronous Shutter, all rows in the ROI are reset then exposed simultaneously for a specified time. At the end of the exposure time, each pixel value is transferred immediately to an adjacent storage area to await read-out. The pixel values are then read out row-by-row from storage, building the frame. This use of intermediate storage reduces the gradual overexposure that can occur down the image when the rows are read out directly from the active area.

Because all rows are exposed simultaneously, Synchronous Shutter avoids the jagged or blurred affect produced by Rolling Shutter for fast action images. However, because it stops exposure to perform read-out, it does not provide the fastest possible sequence of frames. If speed is the main consideration, use Rolling Shutter if the circumstances allow it.

Synchronous Shutter requires a trigger event to reset the pixel data and start exposure for the entire ROI (as opposed to Rolling Shutter, in which exposure is an on-going process).

Triggers

Free-Running (Internal) Trigger—The camera creates synchronously exposed frames as frequently as possible under the given parameters, resulting in continuous image capture.

Hardware Trigger—The frame is not created until the camera receives a signal from an attached hardware trigger.

Software Trigger—The frame is not created until the camera receives a specific function call (PxLGetNextFrame, on page 31)).

2.4.1 Starting the Video Stream

Before images can be captured, video image data must be streaming from the camera to the host computer.

- Call PxLSetStreamState (on page 44) set to START_STREAM, to take the camera out of an idle ("stopped") or paused state and open the video stream. After this call, the camera can send image data to the host computer, either continuously or as Frame-on-Demand.

2.4.2 Starting the Image Preview (Optional)

When the video stream is running, the image data can be previewed on the monitor prior to image capture, although this is not necessary for capture.

- To start the image preview, call PxLSetPreviewState (on page 43) set to START_PREVIEW.

2.4.3 Capturing a Single-Frame Image

2.4.3.1 Free-Running Continuous (Default method)

Because the camera is using a Rolling Shutter with this method, the camera will be collecting and buffering frame data before the capture command is received. This method essentially captures the oldest frame in the camera's buffer.

1. Call PxLGetNextFrame (on page 31) to transfer a frame immediately to a user-specified buffer on the host computer.
2. Call PxLFormatImage (on page 21) to convert the frame to an end-user format (.bmp, .tif, .psd. jpg).

Because the frame data is buffered, this may be repeated continually. However, depending on the system speed, some incoming data can be lost if the frame rate is too high for the system to keep up.

2.4.3.2 Internally Triggered Free-Running Continuous

1. Call PxLSetFeature (on page 40) to configure the camera to use the free-running trigger (FEATURE_TRIGGER; TRIGGER_TYPE_FREE_RUNNING).
2. Call PxLGetNextFrame (on page 31). The function will not complete until a valid frame has been captured into the buffer.
3. Call PxLFormatImage (on page 21) to convert the frame to an end-user format (.bmp, .tif, .psd. jpg).

2.4.3.3 Hardware Triggered Frame-on-Demand

1. Call PxLSetFeature (on page 40) to configure the camera to use the hardware trigger (FEATURE_TRIGGER; TRIGGER_TYPE_HARDWARE).
2. Call PxLGetNextFrame (on page 31). The function will not complete until a hardware trigger has been detected and a valid frame has been captured into the buffer.
3. Call PxLFormatImage (on page 21) to convert the frame to an end-user format (.bmp, .tif, .psd. jpg).

2.4.3.4 Software Triggered Frame-on-Demand

1. Call PxLSetFeature (on page 40) to configure the camera to use the software trigger (FEATURE_TRIGGER; TRIGGER_TYPE_SOFTWARE).

2. Call PxLGetNextFrame (on page 31). This activates the software trigger. The function will not complete until a valid frame has been captured into the buffer.
3. Call PxLFormatImage (on page 21) to convert the frame to an end-user format (.bmp, .tif, .psd. jpg).

2.4.4 Capturing a Video Clip

Use the same methods as for capturing a single–frame image (Section 2.4.3, on page 8), with the following changes:

1. Instead of calling PxLGetNextFrame, call PxLGetClip (on page 26) to capture multiple frames into a PixeLINK data stream file (see Appendix C on page 69 for the file format).

   A Hardware-Triggered Frame-on-Demand video capture will require multiple trigger signals to be received before the function can complete.

   A Software-Triggered Frame-on-Demand video capture is similar to, but slower than, an *Internally Triggered Free-Running Continuous video capture*. PxLGetClip will issue the software triggers automatically by calling PxLGetNextFrame the required number of times. These function calls are slower than using the internal trigger.
2. Instead of calling PxLFormatImage, call PxLFormatClip (on page 20) to convert the PixeLINK data stream file into an end-user format (.avi).

2.4.5 Changes to Camera Settings

In general, changes to camera settings are applied to the next frame. The only exception occurs when the camera is performing a Frame-on-Demand capture (that is, a capture using a hardware or software trigger).

With a Frame-on-Demand capture:

- The camera is in a "wait state" while waiting for a trigger. Any requests to change camera settings received during the wait state are acted upon immediately, but while the changes are being enabled, the camera video stream is stopped momentarily, activating a "busy signal." While this busy signal is active, trigger signals are ignored. When the changes in camera settings are complete, the busy signal is deactivated, the video stream is restored to its previous state, and the camera is left in the wait state ready to respond to the trigger.
- The camera is in a "busy state" from the time the trigger is received until any delay period has passed, exposure (integration) is complete, the image data has been read from the sensor, and the camera is made ready to accept the next trigger. Any requests to change to camera settings received during the busy state are postponed and applied at the end of the busy state.

2.5 Uninitializing the Camera

- Call PxLUninitialize (on page 45) to uninitialize the camera.

While a camera is uninitialized, it is in a standby state. It draws power, but it does not collect image data or act on function calls except for PxLGetNumberCameras (on page 32) and PxLInitialize (on page 33).

2.6 Changing, Tracking, and Recording Frame Features in "Descriptors"

A **descriptor** is the set of feature properties that characterize a frame of image data by recording the camera features in effect at the time that frame was delivered from the camera. Under the PixeLINK Camera API Version 4, each frame has a descriptor as metadata. Descriptors are returned and passed by function calls, and custom descriptors may be constructed and applied automatically on a frame-by-frame basis to captured images (including clips) or to the preview window.

If a custom descriptor is not applied to a frame, then the frame's descriptor is constructed from the current camera properties at the time that the frame is delivered. Contents of the descriptor are defined in the descriptor structure, FRAME_DESC, as defined on page 55.

When a frame or video clip is captured using PxLGetNextFrame (on page 31) or PxLGetClip (on page 26), the descriptor for each frame is stored with it. When the frame or clip is converted to an end-user format using PxLFormatImage (on page 21) or PxLFormatClip (on page 20), a custom descriptor can be applied to change the image's width, height, decimation, or pixel format. The end-user file does not contain a statement of the descriptor data.

For many applications, descriptors are informative only and do not play an active role, needing only to be passed from function to function.

Creating and Applying Custom Descriptors

Some applications require specific, automatic changes to frame properties on a frame-by-frame basis for display or storage purposes. This can be achieved by creating and applying custom descriptors to the video stream.

Descriptors are applied, in order, on a frame-by-frame basis. If more than one frame is being previewed or captured, the descriptors are applied in a cyclic fashion (that is, after the last descriptor is applied, the first descriptor is applied to the next frame, then the second descriptor is applied to the following frame, and so on).

Note that custom descriptors require advanced handling techniques. For more information, refer to Appendix D: Advanced Functions—Creating and Using Custom Descriptors (on page 70).

2.7 Lookup Tables (LUTs)

The PixeLINK API accommodates the use of a user-specified lookup table (LUT) for custom image adjustment. The LUT is an array that maps each possible pixel value to a user-specified pixel value. Thus, the number of elements in the LUT array is equal to the number of possible pixel values in the input, which is a function of the pixel depth of the data stream. The number of elements is 2^n, where "n" is the number of bits per pixel. So, for a camera with 10-bit pixels, the LUT would have $2^{10} = 1024$ elements.

The values of the original pixel data are integers from 0 to $2^n - 1$. (So, for a 10-bit camera, the values range from 0 to 1023.)

Examples of uses of lookup tables include pixel depth reduction (say, from 10-bit to 8-bit), grayscale inversion, and custom filtering.

In the PixeLINK API, LUTs are treated as camera features by the set and get functions. An LUT is loaded as a feature using the PxLSetFeature function (on page 40) and applied to the camera, allowing the processing to the performed in hardware for optimal speed. However, unlike other features, LUTs are not stored in descriptors (see Section 2.6, on page 10).

An LUT can be read from the camera using the PxLGetFeature function (on page 29).

For more information on using features, refer to PxLSetFeature and PxLGetFeature. For more information on using FEATURE_LOOKUP_TABLE, see page 54.

3

API Reference

3.1 Summary of API Functions

Figure 3.1 (on page 13) lists the API commands and shows the relationships between functional groups of commands.

Table 3.1 (on page 14) lists the API functions by functional group. Table 3.2 (on page 15) lists the API functions alphabetically. Designers should consult the referenced pages for detailed descriptions of the API functions.

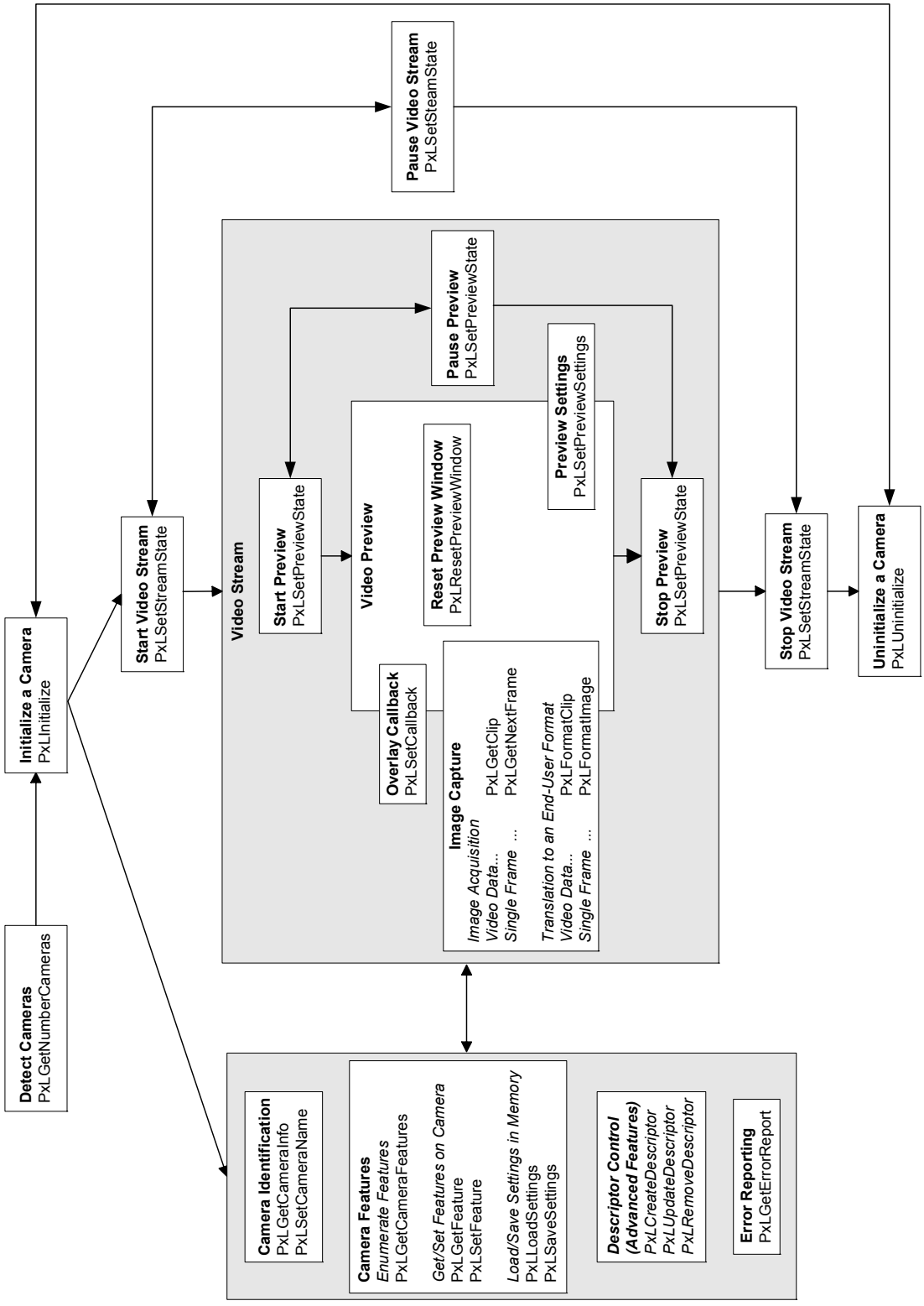

Figure 3.1 Chart of PixeLINK Camera API Functions

Table 3.1 PixeLINK Camera API Functions, Listed by Functional Group

Detect Cameras

API Function Name	Description	Page
PxLGetNumberCameras ()	Get the number of cameras currently connected to the bus	32

Initialize a Camera

API Function Name	Description	Page
PxLInitialize ()	Initialize a camera and return the camera handle	33

Camera Identification

API Function Name	Description	Page
PxLGetCameraInfo()	Get information about the specified camera	25
PxLSetCameraName()	Set a name for the specified camera	39

Camera Features

API Function Name	Description	Page
Enumerate Features		
PxLGetCameraFeatures()	Get the list of the features supported by the specified camera	23
Get/Set Features on Camera		
PxLGetFeature()	Get the current value of the specified feature	29
PxLSetFeature()	Set the value of the specified feature	40
Load/Save Settings in the Camera's Non-Volatile Memory		
PxLLoadSettings()	Load settings from non-volatile memory on the camera	34
PxLSaveSettings()	Save the current settings to non-volatile memory on the camera	36

Video Stream

API Function Name	Description	Page
PxLSetStreamState()	Set the current state of the video stream to stopped, started or paused	44

Overlay Callback

API Function Name	Description	Page
PxLSetCallback()	Specify a callback function to modify the video data in the preview window or as it is translated into an end-user format	37

Preview

API Function Name	Description	Page
PxLSetPreviewSettings()	Set the preview window settings for the specified camera	41
PxLSetPreviewState()	Set the current state of the preview window to playing, stopped, or paused	43
PxLResetPreviewWindow()	Reset the size of the preview window to the size of the streaming video (thereby optimizing display performance)	35

Image Capture

API Function Name	Description	Page
Image Data Acquisition		
[*Video Data*] PxLGetClip()	Get a video clip and save it as a PixeLINK datastream (..pds) intermediate file	26
[*Single Frame*] PxLGetNextFrame()	Get the next image frame from the camera and put it in an image buffer	31
Translation to an End-User Format		
[*Video Data*] PxLFormatClip()	Convert a PixeLINK datastream (.pds) file to a video file (.avi)	20
[*Single Frame*] PxLFormatImage()	Convert a raw frame residing in an image buffer to an image file (.bmp, .tif, .psd, .jpg)	21

Error Reporting

API Function Name	Description	Page
PxLGetErrorReport()	Returns details about the last error that occured	28

Uninitialize a Camera

API Function Name	Description	Page
PxLUninitialize()	Uninitialize the specified camera	45

Descriptor Control (Advanced Features, listed in Appendix D on page 70)

API Function Name	Description	Page
PxLCreateDescriptor()	Create a descriptor for the specified camera	74
PxLUpdateDescriptor()	Set the update mode for the specified descriptor	77
PxLRemoveDescriptor()	Remove the descriptor from the specified camera	76

Table 3.2 PixeLINK Camera API Functions, Listed Alphabetically

API Function Name	Description	Page
PxLCreateDescriptor()	Create a descriptor for the specified camera	74
PxLFormatClip()	Convert a PixeLINK datastream (.pds) file to a video file (.avi)	20
PxLFormatImage()	Convert a raw frame residing in an image buffer to an image file (.bmp, .tif, .psd, .jpg)	21
PxLGetCameraFeatures()	Get the list of the features supported by the specified camera	23
PxLGetCameraInfo()	Get information about the specified camera	25
PxLGetClip()	Get a video clip and save it as a PixeLINK datastream (..pds) intermediate file	26
PxLGetErrorReport()	Returns details about the last error that occured	28
PxLGetFeature()	Get the current value of the specified feature	29
PxLGetNextFrame()	Get the next image frame from the camera and put it in an image buffer	31
PxLGetNumberCameras ()	Get the number of cameras currently connected to the bus	32
PxLInitialize ()	Initialize a camera and return the camera handle	33
PxLLoadSettings()	Load settings from non-volatile memory on the camera	34
PxLRemoveDescriptor()	Remove the descriptor from the specified camera	76
PxLResetPreviewWindow()	Reset the size of the preview window to the size of the streaming video	35

API Function Name	Description	Page
	(thereby optimizing display performance)	
PxLSaveSettings()	Save the current settings to non-volatile memory on the camera	36
PxLSetCallback()	Specify a callback function to modify the video data in the preview window or as it is translated into an end-user format	37
PxLSetCameraName()	Set a name for the specified camera	39
PxLSetFeature()	Set the value of the specified feature	40
PxLSetPreviewSettings()	Set the preview window settings for the specified camera	41
PxLSetPreviewState()	Set the current state of the preview window to playing, stopped, or paused	43
PxLSetStreamState()	Set the current state of the video stream to stopped, started or paused	44
PxLUninitialize()	Uninitialize the specified camera	45
PxLUpdateDescriptor()	Set the update mode for the specified descriptor	77

3.2 API Functions

This section contains a detailed description of each function in the API.

3.2.1 Notational Conventions

In this section,

- *italics* are used to represent variables, function names, and program names,
- `courier` is used to represent source code given as examples.

3.2.2 Using the API Functions with C/C++ or Visual Basic

To use the API functions within a C/C++ program, the program must include the header PixeLINKApi.h and link to PxLApi40.lib. To use within a Visual Basic program, add the file PixeLINKApi.bas to the Visual Basic project.

3.2.3 Obtaining the Imager Object Handle

Most API functions in this set require a camera to be selected with PxLInitialize (p.33) first. PxLInitialize provides the camera Object Handle (or "Camera Handle") required by these functions. The camera is specified by the required serial number which is made available through PxLGetNumberCameras (on page 32).

3.2.4 Return Values

All return values are listed in Appendix B: API Return Codes (on page 67).

3.2.5 Data Types

The PixeLINK API uses a variety of simple and structured data types.

BOOLEAN A 32 bit variable set to 0 for FALSE and non-zero for TRUE.

Visual Basic has a different definition of Boolean and the application should use LONG instead.

CAMERA_FEATURES

A structure used to define the supported camera features. See page 48 for a detailed description.

Visual Basic – CAMERA_FEATURES

CAMERA_INFO

A structure used to define camera-specific version information. See page 49 for a detailed description.

Visual Basic – CAMERA_INFO

float A 32 bit floating point number

Visual Basic –DOUBLE

FRAME_DESC

A structure containing the settings for a frame. See page 55 for a detailed description.

Visual Basic – FRAME_DESC

HANDLE A pointer to a structure that allows access to the camera driver. It is usually an unsigned 32 bit value.

For application languages other than C/C++, the HANDLE type may be considered a 32-bit signed or unsigned integer.

Visual Basic –LONG

LPVOID A pointer to any type (for example, typedef void *LPVOID;)

Visual Basic – does not deal in pointers. In this case, the best way is to create a dynamic array of type BYTE and access its first member By Ref.

LPSTR This indicates a pointer to type CHAR (typedef CHAR *LPSTR;)

CHAR is an 8 bit variable.

For an OUT variable, a typical "pointer" can be created by created an array (e.g. char pName[33];)

Visual Basic – This is STRING By Ref

For an OUT variable, then the application must ensure that a NUL terminated string of appropriate length is available by using String$(NECESSARY_SIZE, vbNulChar)

PCAMERA_FEATURES
A pointer to CAMERA_FEATURES
Visual Basic – CAMERA_FEATURES By Ref

PCAMERA_INFO
A pointer to CAMERA_INFO
Visual Basic – CAMERA_INFO By Ref

PFRAME_DESC
A pointer to FRAME_DESC
Visual Basic – FRAME_DESC By Ref

PU32 A pointer to a 32 bit unsigned integer (U32)
Visual Basic – Usually use LONG By Ref. In some cases (such as PxLGetFeature, on page 29), a dynamic array is required instead. The size of the array is based on the number of LONGs required.

U32 A 32 bit unsigned integer.
Visual Basic does not have unsigned values. Use LONG.

3.2.6 Data Direction

The PixeLINK function descriptions also indicate the data direction:

IN An input parameter that will not be modified by the function. A constant may be used.

OUT A parameter or pointer location to be returned by the function.

IN OUT An input parameter or pointer location to be modified by the function.

3.2.7 Reserved Messages

The API reserves Windows messages from WM_USER to WM_USER+100. If user-defined messages are required for user applications, WM_USER+101 should be specified.

3.2.8 Format of API Function Descriptions

The following sample (MyFunction) illustrates the general format of each entry and describes the information contained within each subsection.

MyFunction

Syntax:

`PXL_RETURN_CODE MyFunction(modifier` *`parameter`*`[,...]);`

This statement demonstrates the declaration syntax of the function. Each parameter is *italicized*.

Description:

This summary of the function's purpose is followed by descriptions of its parameters and other relevant information pertaining to the function.

Default: [*Selected functions only*]

The default value(s) of camera properties listed here are valid after initialization or reset.

Usage: [*Selected functions only*]

A sample is provided to demonstrate usage, where necessary.

Comments, Restrictions and Limitations:

This area contains additional comments about the function.

PxLFormatClip

Syntax:

```
PXL_RETURN_CODE PxLFormatClip(
                              IN LPSTR pInputFileName,
                              IN LPSTR pOutputFileName,
                              IN U32 uOutputFormat);
```

Description:

This function converts a PixeLINK data stream file (.pds) into a standard Windows video clip file.

- *pInputFileName* is the name of the PixeLINK data stream file to be converted.
- *pOutputFileName* is the name of the video clip file that to be created.
- *uOutputFormat* is the format of the output video file. Valid formats are:

  CLIP_FORMAT_AVI — Convert to standard Windows .avi file.

  This value is defined in the file PixeLINKTypes.h.

Comments, Restrictions and Limitations:

To create the input file (in PixeLINK data stream format), use PxLGetClip (p.26).

In addition to video image data, the PixeLINK data stream file includes descriptors containing sets of properties for each frame in the file. For more information on descriptors, see Section 2.6 (p.10).

If the height and width of the frames in the source file varies, the smallest values are used and any larger frames are cropped.

For an .avi output file (CLIP_FORMAT_AVI), the frame timing is determined from the descriptors in the .pds input file.

For more information about the PixeLINK data stream file format, see Appendix C (p.69).

Overlay callback:

To add an overlay callback function to each frame of the output file, first specify the callback function with PxLSetCallback (p.37), setting the usage case to OVERLAY_FORMAT_CLIP.

PxLFormatImage

Syntax:

```
PXL_RETURN_CODE PxLFormatImage(
                                IN LPVOID pSrc,
                                IN PFRAME_DESC pFrameDesc,
                                IN U32 uOutputFormat,
                                OUT LPVOID pDest,
                                IN OUT PU32 pDestBufferSize,
```

Description:

This function converts a frame (stored in a buffer) into an end-user format and stores it in a buffer.

- *pSrc* is the pointer to the buffer containing the image data to be converted.
- *pFrameDesc* is the pointer to the frame descriptor. This is normally provided by PxLGetNextFrame (p.31), but the application may also apply a custom descriptor to the output frame. See below for more information.
- *uOutputFormat* is the output format desired. Valid values are:

  IMAGE_FORMAT_BMP — Convert to bitmap (.bmp) format

  IMAGE_FORMAT_TIFF — Convert to TIFF (.tif)

  IMAGE_FORMAT_PSD — Convert to Adobe Photoshop (.psd) format

  IMAGE_FORMAT_JPEG — Convert to JPEG (.jpg) format (high quality; 5% compression).

  These values are defined in the file PixeLINKTypes.h.
- *pDest* is a pointer to the destination buffer. Set to NULL to return the required output buffer size in *pDestBufferSize.*
- *pDestBufferSize* points to the size of the *pDest* buffer (in bytes). On output, the function returns the actual buffer size in this parameter.

Usage:

Recommended method of using PxLFormatImage:

1. Call the function with *pDest* set to NULL. The required size of the destination buffer is returned in *pDestBufferSize.*
2. Allocate the required number of bytes of memory to the destination buffer.
3. Call the function with *pDest* set to point to the allocated destination buffer.

Comments, Restrictions and Limitations:

Output buffer size:

Prior to calling this function, sufficient memory space should be allocated to the destination buffer. If the destination buffer is too small, the application will return an "ApiBufferTooSmall" message (see Appendix B, p.67).

Regardless of the return code generated, PxLFormatImage will return the size of the output frame in *pDestBufferSize*, so to determine the appropriate amount of the memory to be allocated to the destination buffer without running the risk of generating an error message, call PxLFormatImage with *pDest* set to NULL. The required number of bytes

will be returned in *pDestBufferSize*, and the destination buffer can be allocated prior to calling the function with *pDest* pointing at the buffer location.

Input buffer size:

The size of the input buffer is assumed to be width × height × number of bytes per pixel.

Custom frame descriptor:

Use of a custom descriptor requires the advanced functions listed in Appendix D (p.70).

Note that *pFrameDesc* applies only the following four features from a descriptor:

- Width
- Height
- Decimation
- Pixel format

For more information, see page 55 for an explanation of the FRAME_DESC structure (defined in the file PixeLINKTypes.h).

Overlay callback:

To add an overlay callback function to the output image, first specify the callback function with PxLSetCallback (p.37) setting the usage case to OVERLAY_FORMAT_IMAGE.

PxLGetCameraFeatures

Syntax:

```
PXL_RETURN_CODE PxLGetCameraFeatures(
                                IN HANDLE hCamera,
                                IN U32 uFeatureId,
                                OUT PCAMERA_FEATURES pFeatureInfo,
                                IN OUT PU32 pBufferSize);
```

Description:

This function identifies the list of features supported by the camera and enumerates the minimum and maximum values of all features. It can also retrieve the minimum and maximum values of a specified feature.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uFeatureId* is the ID of the specified feature, or set to FEATURE_ALL to return information on all supported features. Information is returned in the CAMERA_FEATURES structure in the *pFeatureInfo* buffer.
- *pFeatureInfo* is the pointer to the buffer to be filled with the CAMERA_FEATURES structure containing the feature information. Set to NULL to return the required size of the buffer in *pBufferSize*.
- *pBuffersize* is the number of bytes of memory (in the buffer) pointed to by *pFeatureInfo*. If *pFeatureInfo* is set to NULL then the required size of *pFeatureInfo* will be returned in *pBufferSize*.

Possible feature IDs are listed starting on page 52. The CAMERA_FEATURES structure is described on page 48.

Feature IDs, CAMERA_FEATURES, and FEATURE_ALL are defined in the file PixeLINKTypes.h

Default:

Camera dependent.

Usage:

Recommended method of using PxLGetCameraFeatures:

1. Call the function with *pFeatureInfo* set to NULL. The required size of the destination buffer is returned in *pBuffersize.*
2. Allocate the required number of bytes of memory to the buffer.
3. Call the function with *pFeatureInfo* set to point to the allocated buffer.

Example:

```
// Variables
PCAMERA_FEATURES   pFeatures = NULL;
U32                BufferSize;

// Determine the size of memory required
nRetValue = PxLGetCameraFeatures(hCamera, FEATURE_ALL, NULL,
                &BufferSize);
```

```
… Error checking
// Get memory
pFeatures = (PCAMERA_FEATURES)malloc(BufferSize);
… Error Checking

/* Make the request to enumerate all the camera features */
nRetValue = PxLGetCameraFeatures(hCamera, FEATURE_ALL, pFeatures,
                &BufferSize);
/* If no error – show feature IDs */
if (ApiSuccess == nRetValue)
{
       for (i = 0; i < pFeatures->uNumberOfFeatures; i++)
       {
       printf("Feature %d = %d\n", i, Features->Feature[i].uFeatureId);
       }
}

// When finished
free(pFeatures);
pFeatures = NULL;
```

Comments, Restrictions and Limitations:

Features are camera dependent. Use this function to identify the presence and range of all features in the camera. The returned information identifying the features can be used to qualify calls to the PxLGetFeature (p.29) and PxLSetFeature (p.40) functions prior to use.

PxLGetCameraInfo

Syntax:

```
PXL_RETURN_CODE PxLGetCameraInfo(
                                IN HANDLE hCamera,
                                OUT PCAMERA_INFO pInformation);
```

Description:

This function returns version information about the PixeLINK hardware and firmware.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *pInformation* is the pointer to CAMERA_INFO structure, which contains strings of information about PixeLINK hardware and firmware. The CAMERA_INFO structure is described on page 49 and defined in the file PixeLINKTypes.h.

Usage:

```
/* Make the request */
nRetValue = PxLGetCameraInfo(hCamera, &Information);

/* Check for error */
if (ApiSuccess == nRetValue)
{
   printf("Camera Name %s\n", Information.CameraName);
   printf("Vendor %s\n", Information.VendorName);
   printf("Model ID %s\n", Information.ModelName);
   printf("Description %s\n", Information.Description);
   printf("Serial Number %s\n", Information.SerialNumber);
   printf("Firmware Version %s\n", Information.FirmwareVersion);
   printf("FPGA Version %s\n", Information.FpgaVersion);
}
```

Comments, Restrictions and Limitations:

None.

PxLGetClip

Syntax:

```
PXL_RETURN_CODE PxLGetClip(
                        IN HANDLE hCamera,
                        IN U32 uNumberOfFrames,
                        IN LPSTR pFileName,
                        IN U32 (_stcall * TerminationFunction)(
                            IN HANDLE hCamera,
                            IN U32 uNumberOfFramesCaptured,
                            IN PXL_RETURN_CODE uRetCode));
```

Description:

This function saves a clip of video to a PixeLINK data stream (.pds) file. This file can be converted to an end-user file using the function PxLFormatClip (p.20).

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uNumberOfFrames* is the number of frames to be captured in the clip.
- *pFileName* is the name of the PixeLINK data stream file to be created.
- (_stcall * TerminationFunction)(IN HANDLE *hImager*, IN PXL_RETURN_CODE *uRetCode*) is the pointer to the termination function to be called when:
  - the function has acquired the required number of frames,
  - there is possibility of disk overflow, or
  - the application requests termination by stopping the video stream (using the PxLSetStreamState function (p.44) with the stream state set to STOP_STREAM).
  - *hImager* is the handle of the camera sending the data.
  - *uNumberOfFramesCaptured* is the actual number of frames captured into the clip.
  - *uRetCode* is the returned value indicating the status of the call. Per the reasons given above, the possible responses are:
    - "ApiSuccess" (function completed successfully),
    - "ApiDiscOverflowError" (process forced to halt because of possible disk overflow), or
    - "ApiStreamStoppedError" (the application has stopped the stream).

  Return values are listed in Appendix B (p.67).

Usage:

```
/* Make the request */
nRetValue = PxLGetClip(hCamera, uNumberOfFrames, pFileName,
                TerminationFunction);

/* Define overlay function */
U32 stdcall TerminationFunction(IN HANDLE hImager,
                    IN U32 uNumberOfFramesCaptured,
                    IN PXL_RETURN_CODE uRetCode)
```

```
{
   switch(uRetCode)
   {
      case ApiSuccess:
         printf("Function completed successfully.\n");
         break;
      case ApiDiscOverflowError:
         printf("Process halted – possible disk overflow.\n");
         break;
      case ApiStreamStoppedError:
         printf("Application requested termination.\n");
         break;
      default:
         return uRetCode;
   } // End of switch of return code
   return ApiSuccess;
} // End of TerminationFunction
```

Comments, Restrictions and Limitations:

For a description of the PixeLINK data stream file format, see Appendix C (p.69).

Before this function is called, the camera defined by the handle *hCamera* must be streaming video. Use the PxLSetStreamState function (p.44) with the stream state set to START_STREAM.

If the video stream is stopped by the PxLSetStreamState function (with the stream state set to STOP_STREAM) before the PxLGetClip function (p.26) can complete the clip, the clip will be saved with the number of frames captured up to the point that the video stream is stopped.

The PixeLINK data stream file (.pds) that is created can be converted to an AVI file using the function PxLFormatClip (p.20).

In addition to video image data, the PixeLINK data stream file includes descriptors containing sets of properties for each frame in the file. For general information on descriptors, see Section 2.6 (p.10).

PxLGetClip operates using two separate threads: One thread captures the video data and places it in a queue, allocating more memory if necessary (that is, if the queue is full); the other thread copies video frames from the queue and saves them, along with their corresponding descriptors, to disk as a PixeLINK data stream file. Using a dual-thread architecture reduces the likelihood of dropping frames while capturing long video clips, although it increases memory usage.

PxLGetErrorReport

Syntax:

```
PXL_RETURN_CODE PxLGetErrorReport (
                                IN HANDLE hCamera,
                                OUT PERROR_REPORT pErrorReport);
```

Description:

This function returns information the most recent error occurrence. When a function call returns an error, it caches the error code and a report on the likely cause of the error. This information is retrieved by a call to PxLGetErrorReport. The information in the error report is overwritten the next time a function call returns an error.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *pErrorReport* is the pointer to a ERROR_REPORT structure, which contains strings of information about the most recent error occurrence. This structure is described on page 51 and defined in the file PixeLINKTypes.h.

Usage:

```
_An error occurs

/* Make the request */
nRetValue = PxLGetErrorReport(hCamera, &ErrorReport);

/* Check for error */
if (ApiSuccess == nRetValue)
{
   /* Display error information */
   printf("Last function to return error was %s\n",
      ErrorReport.strFunctionName);
   printf("Last Error was 0x%X (%s)\n", ErrorReport.uReturnCode,
      ErrorReport.strReturnCode);
   printf("Description %s\n", ErrorReport.strReport);
}
```

Comments, Restrictions and Limitations:

If no errors have occurred then the return code field of the error report structure is set to *ApiSuccess* and all other fields are set to null strings.

PxLGetFeature

Syntax:

```
PXL_RETURN_CODE PxLGetFeature(
                                IN HANDLE hCamera,
                                IN U32 uFeatureId,
                                OUT PU32 pFlags,
                                IN OUT PU32 pNumberParms,
                                OUT float* pParms);
```

Description:

This function returns the current value of the selected feature. Features are camera-dependent.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uFeatureId* is the ID of the selected feature. Features and feature IDs are listed starting on page 52 and defined in the file PixeLINKTypes.h.
- *pFlags* is the pointer to the current flag for the feature. Feature flags are described on page 58 and are defined in the file PixeLINKTypes.h.
- *pNumberParms* is the pointer to the number of entries in the *pParms* array. If *pParms* is set to NULL then the required number of array entries will be returned in *pNumberParms*.
- *pParms* is the pointer to an array of values that to be filled with the current parameters for the specified feature. Set to NULL to return the required number of array entries in *pNumberParms*.

Default:

Camera dependent.

Usage:

Recommended method of using PxLGetFeature:

1. Call the function with *pParms* set to NULL. The required number of array entries is returned in *pNumberParms.*
2. Allocate 4 bytes of memory to *pFlags*. Allocate the required number of bytes of memory to *pParms*: that is, *pNumberParms* × *sizeof(float)*. [The value of *sizeof(float)* is usually 4 bytes, but this depends on the host computer.]
3. Call the function with *pFlags* and *pParms* set to point to their own areas of allocated memory.

Example:

```
// Variables
float*    pParameters = NULL;
U32       uNumberParms = 0;
U32       uFeatureId;  // Set to the desired feature
U32       uFlags;

// Determine the size of memory required
nRetValue = PxLGetFeature(hCamera, uFeatureId, &uFlags, &uNumberParms,
```

```
                NULL);
… Error checking

// Get memory
pFeatures = (float*)malloc( uNumberParms * sizeof(float) );
… Error Checking

/* Make the request */
nRetValue = PxLGetFeature(hCamera, uFeatureId, &uFlags,
                &uNumberParms, pParameters);
/* If no error – show feature values */
if (ApiSuccess == nRetValue)
{
      for (i = 0; i < uNumberParms; i++)
      {
      printf("Parameter %d = %f\n", i, pParameters[i]);
      }
}

// When finished
free(pParameters);
pParameters = NULL;
```

Comments, Restrictions and Limitations:

Features

Features are camera dependent. The PxLGetCameraFeatures function (p.23) should be used to identify the presence and range of all features present in the camera.

Features and feature IDs are listed starting on page 52. Feature flags are described on page 58. All are defined in the file PixeLINKTypes.h.

Using PxLGetFeature with descriptors:

PxLGetFeature can read the settings of any custom descriptors that are in use. Custom descriptors are managed using the advanced functions listed in Appendix D (on page 70).

Note that if all descriptors are in focus at once (rather than just a single descriptor), calling PxLGetFeature will generate an error. For more information on bringing a descriptor in focus, see page 71.

PxLGetNextFrame

Syntax:

```
PXL_RETURN_CODE PxLGetNextFrame(
                                IN HANDLE hCamera,
                                IN U32 uBufferSize,
                                OUT LPVOID pFrame,
                                OUT PFRAME_DESC pDescriptor);
```

Description:

This function returns a frame of image data and the feature descriptor for that frame from the video stream. If the camera has a trigger feature (see PxLGetCameraFeatures, p.23 and Features, p.52) and the trigger mode is set to TRIGGER_TYPE_SOFTWARE (see Trigger Type, p.64), then this function also causes a software trigger. Trigger types are defined in the file PixeLINKTypes.h.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uBufferSize* is the number of bytes of memory allocated for the frame. This value is used as a check to prevent memory overflow
- *pFrame* is the pointer to the array receiving the frame of image data.
- *pDescriptor* is the pointer to a frame descriptor. For an explanation of the FRAME_DESC structure, see page 55. The structure is defined in the file PixeLINKTypes.h.

Comments, Restrictions and Limitations:

Before this function is called, memory space should be allocated for the image data *pFrame* (width × height × number of bytes per pixel) and to the FRAME_Desc structure for *pDescriptor*.

Before this function is called, the camera defined by the handle *hCamera* must be streaming video. Use the PxLSetStreamState function (p.44) with the stream state set to START_STREAM.

Trigger modes

When the camera is free-running (that is, frames are output continuously, driven by an internal trigger that occurs as frequently as possible), this function returns the next available frame from the camera. Loss of frames can occur if the application takes more time to make the next request than is allowable with the requested frame rate of the camera.

When in software trigger mode, PxLGetNextFrame returns the frame acquired after the software trigger has occurred.

Trigger types are described on page 64 and defined in the file PixeLINKTypes.h.

Dropped frame

One can determine if a frame was dropped by looking at the frame number field in the feature descriptor. For an explanation of the FRAME_DESC structure, see page 55. The structure is defined in the file PixeLINKTypes.h.

PxLGetNumberCameras

Syntax:

```
PXL_RETURN_CODE PxLGetNumberCameras(
                              OUT PU32 pSerialNumbers,
                              IN OUT PU32 pNumberSerial);
```

Description:

This function determines the number of PixeLINK cameras connected to the system.

- *pSerialNumbers* is the pointer to an array of the serial numbers of the cameras connected to the system. To ignore, set to NULL—the function will return the number of connected cameras (that is, the required number of array entries for *pSerialNumbers*) in *pNumberSerial*., for the purpose of memory space allocation.
- *pNumberSerial* is the pointer to the number of cameras connected (that is, the number of entries in the array pointed to by *pSerialNumbers*). Each entry is 4 bytes in size.

Comments, Restrictions and Limitations:

Before this function is called, memory space should be allocated for the *pSerialNumbers* array (that is, *pNumberSerial* × 4 bytes). Set *pSerialNumbers* to NULL to return *pNumberSerial* to determine the space allocation.

This function is typically used at the beginning of a session, to determine the number of cameras connected to the system. It requires a longer response time than other functions, although it is well within the time needed to open an application or for a user to change settings within a GUI. It is expected that this function will be called when high speed is not critical to the performance of the application, such as on startup or reinitialization, when the response time will be transparent to the user. It is not typically necessary to use this function within image display and capture routines. To avoid undue system delays, this function should be used only when necessary.

The list of serial numbers can be used with PxLInitialize (p.33) to specify a specific camera to initialize.

PxLInitialize

Syntax:

```
PXL_RETURN_CODE PxLInitialize(
                                IN U32 uSerialNumber,
                                OUT HANDLE* phCamera);
```

Description:

This function initializes the camera in its entirety and obtains the camera handle required for subsequent API function calls.

- *uSerialNumber* is the serial number of the required camera. If there is only one camera connected to the system, this value is ignored. If there is more than one camera connected to the system, this value is used to select the camera.
- *phCamera* is the pointer to the variable receiving the camera handle.

Comments, Restrictions and Limitations:

To determine the serial numbers of the connected cameras, use PxLGetNumberCameras (p.32).

An application cannot call PxLInitialize twice for the same camera; nor can two applications have the same camera initialized at the same time.

If there is no camera attached with the specified serial number, then the function will return an "ApiInvalidSerialNumberError" message (see Appendix B, p.67).

PxLLoadSettings

Syntax:

```
PXL_RETURN_CODE PxLLoadSettings(
                              IN HANDLE hCamera,
                              IN U32 uChannel);
```

Description:

This function loads camera settings from the specified memory channel of the camera's non-volatile memory.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uChannel* is memory channel number from which the settings will be loaded. Valid values are between 0 (FACTORY_DEFAULT_MEMORY_CHANNEL) and MAX_MEMORY_CHANNEL (as defined by FEATURE_MEMORY_CHANNEL).

  FEATURE_MEMORY_CHANNEL is described on page 54 and defined in the file PixeLINKTypes.h

Comments, Restrictions and Limitations:

All settings **except** FEATURE_LOOKUP_TABLE are stored in the memory channel.

This function overwrites all current camera settings **except** FEATURE_LOOKUP_TABLE with the ones stored in the specified memory channel. Because FEATURE_LOOKUP_TABLE is not saved in the memory channel, it will not change when this function is called.

If multiple descriptors are being used, then the current descriptor is loaded with the new settings. Settings pertaining to multiple descriptors (such as the number of descriptors and their order) are not affected.

The FACTORY_DEFAULT_MEMORY_CHANNEL (0) is read-only. It cannot be set using PxLSaveSettings (p.36). User-definable memory channels start at 1.

PxLResetPreviewWindow

Syntax:

```
PXL_RETURN_CODE PxLResetPreviewWindow(
                                IN HANDLE hCamera);
```

Description:

This function redefines the dimensions of the preview window's client rectangle to match those of the camera's region of interest (ROI).

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).

Comments, Restrictions and Limitations:

The preview window must be running first. For this function to operate properly, the PxLSetPreviewState function (p.43) must be invoked prior to this function call, with the preview window state set to START_PREVIEW or PAUSE_PREVIEW .

If the preview window has been minimized, it will not be restored by this function call.

PxLSaveSettings

Syntax:

```
PXL_RETURN_CODE PxLSaveSettings(
                              IN HANDLE hCamera,
                              IN U32 uChannel);
```

Description:

This function stores the current settings of the camera in a specified memory channel of the camera's non-volatile memory.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uChannel* is memory channel number to save the settings to. This value starts at 1 and goes to MAX_MEMORY_CHANNEL (as defined by FEATURE_MEMORY_CHANNEL).

  FEATURE_MEMORY_CHANNEL is described on page 54 and defined in the file PixeLINKTypes.h.

Comments, Restrictions and Limitations:

All settings **except** FEATURE_LOOKUP_TABLE are saved by PxLSaveSettings.

This function overwrites any settings already stored in the specified memory channel.

The read-only memory channel, FACTORY_DEFAULT_MEMORY_CHANNEL (0), cannot be set using PxLSaveSettings (p.36). User-definable memory channels start at 1.

If using multiple descriptors, only the contents of the current descriptor are saved. Settings pertaining to multiple descriptors (such as the number of descriptors and their order).

PxLSetCallback

Syntax:

```
PXL_RETURN_CODE PxLSetCallBack (
                    IN HANDLE hCamera,
                    IN U32 uOverlayUse,
                    IN U32 (_stcall * DataProcessFunction)(
                       IN HANDLE hCamera,
                       IN OUT LPVOID pFrameData,
                       IN U32 uDataFormat,
                       IN PFRAME_DESC pDescriptor));
```

Description:

This function specifies a callback function, such as a time stamp or logo, to be used with the video preview window or frame formatting functions (PxLFormatImage, p.21; PxLFormatClip, p.20). The callback function overlays an image on the frame(s) in accordance with to the case(s) specified in *uOverlayUse*. The callback function can be called:

- just before a frame is displayed in the preview,
- just before formatted video data is returned from PxLFormatImage (p.21),
- just before a frame is saved to disk in PxLFormatClip (p.20), or
- just after a raw frame is received by the driver.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uOverlayUse* provides the case(s) in which this overlay function should be used.

  Valid flags are:

  OVERLAY_PREVIEW—The callback is applied to the data displayed in the preview window.

  OVERLAY_FORMAT_IMAGE—The callback is applied to the data converted by PxLFormatImage.

  OVERLAY_FORMAT_CLIP—The callback is applied to the data saved by PxLFormatClip.

  OVERLAY_FRAME—The callback is applied to the raw data received by the driver.

- (_stcall * DataProcessFunction)(IN HANDLE *hImager*, OUT LPVOID *pFrameData*, IN U32 *uDataFormat*, IN PFRAME_DESC *pDescriptor*) is the pointer to the callback function.
  - *hImager* is the handle of the camera sending the data.
  - *pFrameData* is the pointer to the frame data to be modified by the callback function. Note that this can be in one of several different data formats, depending on when this function is called.
  - *uDataFormat* is the format of the frame data. Valid values are:

    PIXEL_FORMAT_MONO8—Grayscale, 8 bits per pixel.

    PIXEL_FORMAT_MONO16—Grayscale, 16 bits per pixel, with the first byte as the most significant byte.

    PIXEL_FORMAT_YUV422—Color, 16 bits per pixel, with a coding pattern of U0, Y0, V0, Y1, U2, Y2, V2, Y3, …

PIXEL_FORMAT_BAYER8—Color, 8 bits per pixel, with data in Bayer pattern format.

PIXEL_FORMAT_BAYER16—Color ,16 bits per pixel, with data in Bayer pattern format.

PIXEL_FORMAT_RGB24—Color, 24 bits per pixel.

PIXEL_FORMAT_RGB48—Color, 48 bits per pixel.

- *pDescriptor* is the pointer to the frame descriptor. This is normally provided by PxLGetNextFrame (p.31), but the application may also apply a custom descriptor to the output frame. See below for more information.

Comments, Restrictions and Limitations:

If the *DataProcessFunction* parameter is set to NULL, then the callback function will be removed for the specified case(s). Otherwise, the current callback function will be replaced with the new one.

To prevent the frame rate from dropping, limit the time for the callback function when applied to the preview window.

uOverlayUse: If the OVERLAY_FRAME flag is set, the function PxLGetNextFrame (p.31) cannot be called for that camera until the OVERLAY_FRAME flag is reset to NULL.

Custom frame descriptor:

Use of a custom descriptor requires the advanced functions listed in Appendix D (p.70).

Note that *pDescriptor* applies only the following four features from a descriptor:

- Width
- Height
- Decimation
- Pixel format

For more information, see page 55 for an explanation of the FRAME_DESC structure (defined in the file PixeLINKTypes.h).

PxLSetCameraName

Syntax:

```
PXL_RETURN_CODE PxLSetCameraName(
                                IN HANDLE hCamera,
                                IN LPSTR pCameraName);
```

Description:

This function sets the name of the camera.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *pCameraName* is the string that holds the new name of the camera.

Usage:

```
/* Make the request */
nRetValue = PxLSetCameraName(hCamera, "Bottom-left camera");

/* Check for error */
if (ApiSuccess != nRetValue)
{
      ;
}
```

Comments, Restrictions and Limitations:

The camera name can have up to 32 characters.

The camera name resides in non-volatile memory and is valid from the time that it is set by the PxLSetCameraName function until another PxLSetCameraName command is executed, even if the unit is powered off.

Note: This function should not be used in a general application as it causes the firmware in the camera to operate in a different mode. It should only be used in a configuration application to set the required camera name.

PxLSetFeature

Syntax:

```
PXL_RETURN_CODE PxLSetFeature(
                              IN HANDLE hCamera,
                              IN U32 uFeatureId,
                              IN U32 uFlags,
                              IN U32 uNumberParms,
                              IN PF32 pParms);
```

Description:

This function sets the value and flags of the selected camera feature.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uFeatureId* is the feature ID. For a list of the possible values, see PixeLINKTypes.h or the types and structures section in this document.
- *uFlags* are the properties to be set for the feature. Feature flags are described on page 58 and are defined in the file PixeLINKTypes.h. **Note:** Exactly one of the following flags must be set when the function is called:

  FEATURE_FLAG_MANUAL—The application controls the feature by sending parameters to the camera.

  FEATURE_FLAG_AUTO—The camera controls the feature continuously. No intervention by the camera is possible.

  FEATURE_FLAG_ONEPUSH—The camera sets the feature only once and then returns to manual operation.
- *uNumberParms* is the number of parameters in the *pParms* array
- *pParms* is the pointer to the array of values for the feature.

Comments, Restrictions and Limitations:

Features

Features are camera dependent. The PxLGetCameraFeatures function (p.23) should be used to identify the presence and range of all features present in the camera.

Features and feature IDs are listed starting on page 52. Feature flags are described on page 58. All are defined in the file PixeLINKTypes.h.

When using this function, remember to review the error return codes. Setting a value out of range, setting an unsupported flag or accessing an unsupported feature will cause an error. See PxLGetErrorReport (p.28).

Using PxLSetFeature with descriptors:

PxLSetFeature can change the settings of any custom descriptors that are in use. Custom descriptors are managed using the advanced functions listed in Appendix D (on page 70).

Note that if all descriptors are in focus at once (rather than just a single descriptor), calling PxLSetFeature set the properties in all descriptors simultaneously. For more information on bringing a descriptor in focus, see page 71.

PxLSetPreviewSettings

Syntax:

```
PXL_RETURN_CODE PxLSetPreviewSettings(
                                IN HANDLE hCamera,
                                IN LPSTR pTitle="PixeLINK
                                   Preview",
                                IN U32 uStyle=
                                   WS_OVERLAPPEDWINDOW|WS_VISIBLE,
                                IN U32 uLeft=CW_USEDEFAULT,
                                IN U32 uTop=CW_USEDEFAULT,
                                IN U32 uWidth=CW_USEDEFAULT,
                                IN U32 uHeight=CW_USEDEFAULT,
                                IN HWND hParent=NULL,
                                IN U32 uChildId=0);
```

Description:

This function defines a preview window if none is present or redefines the settings of the current preview window for the specified camera. Note that except for *hCamera,* all parameters are optional.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *pTitle* is the title of the preview window.
- *uStyle* is the preview window style. See the CreateWindow function of the Windows API for more information.
- *uLeft* specifies the left coordinate of the preview window. This value is relative to the left side of the parent window or of the screen if there is no parent window.
- *uTop* specifies the top coordinate of the preview window. This value is relative to the top of the parent window or of the screen if there is no parent window.
- *uWidth* is the width of the client rectangle of the preview window.
- *uHeight* is the height of the client rectangle of the preview window.
- *hParent* is the handle to the parent window.
- *uChildId* is the ID of the child window.

Usage:

1) Define all values:

```
/* Make the request */
nRetValue = PxLSetPreviewSettings(hCamera, "PixeLINK Camera Preview",
                style, x, y, width, height, parent, ChildId);

/* Check for error */
if (ApiSuccess != nRetValue)
{
      ;
}
```

2) Start the preview as a child window. The preview may be placed as a child window within another window.

```
CWnd* pParent = ??;   // Need a parent
CWnd* pChildWnd = pParent->GetDlgItem(IDC_PICTURE);
RECT  rectChildWindow;

if (!pChildWnd)
{
      AfxMessageBox("Unable to find IDC_PICTURE", MB_OK, 0);
      return;
} // End of test on GetDlgItem

// Get the screen co-ordinates of the Child window
pChildWnd->GetWindowRect(&rectChildWindow);

// Change to parent's client co-ordinates
pParent->ScreenToClient(&rectChildWindow);

/* Start the preview as a child window */
nRetValue = PxLSetPreviewSettings(hCamera,
                  "PixeLINK Camera Preview",  //title
                  WS_CHILD|WS_VISIBLE,     // style
                  rectChildWindow.left,    // x
                  rectChildWindow.top,     // y
                  rectChildWindow.width,   // width
                  rectChildWindow.height,  // height
                  pParent->m_hWnd,         //parent handle
                  ChildId);                // Child ID

/* Check for error */
if (ApiSuccess != nRetValue)
{
      ;
}
```

Comments, Restrictions and Limitations:

Before this function is called, the camera defined by the handle *hCamera* must be streaming video. Use the PxLSetStreamState function (p.44) with the stream state set to START_STREAM.

Once this call has been executed, use PxLSetPreviewState (p.43) to start, stop or pause the stream. Changing the preview settings while the preview window is running will destroy the current preview window and create a new one with the new settings.

Overlay callback:

To add an overlay callback function to the preview, first specify the callback function with PxLSetCallback (p.37), setting the usage case to OVERLAY_PREVIEW.

PxLSetPreviewState

Syntax:

```
PXL_RETURN_CODE PxLSetPreviewState(
                                IN HANDLE hCamera,
                                IN U32 uPreviewState,
                                OUT HWND *pHWnd);
```

Description:

This function establishes the operating state of the preview window.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uPreviewState* is the new preview state. Valid values are:

  START_PREVIEW—The preview window is running.

  PAUSE_PREVIEW—Resources are allocated for the preview window. The preview window, however, is not running and the host CPU is minimizing CPU cycles on this task.

  STOP_PREVIEW—The preview window is stopped and there are no resources allocated to the preview window.

  These values are defined in the file PixeLINKTypes.h.
- *pHWnd* is the pointer to the variable receiving the window handle of the preview window.

Comments, Restrictions and Limitations:

Before this function is called, the camera defined by the handle *hCamera* must be streaming video. Use the PxLSetStreamState function (p.44) with the stream state set to START_STREAM.

While preview is running, avoid calling the PxLGetNextFrame (p.31) function as this will reduce the frame rate to the preview window. To get a separate frame, pause the preview window.

This function can be called without a previous call to the PxLSetPreviewSettings function (p.41). In this case, the default parameters in the PxLSetPreviewSettings function are assumed.

PxLSetStreamState

Syntax:

```
PXL_RETURN_CODE PxLSetStreamState(
                                IN HANDLE hCamera,
                                IN U32 uStreamState);
```

Description:

This function determines the operating state of the video stream coming from the camera.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *uStreamState* is the new stream state. Valid values are:

  START_STREAM—The stream is running.

  PAUSE_STREAM— Resources are allocated for the stream window. The stream, however, is not running and the host CPU is minimizing CPU cycles on this task.

  STOP_STREAM—The stream is stopped and no resources are allocated.

  These values are defined in the file PixeLINKTypes.h.

Comments, Restrictions and Limitations:

None.

.

PxLUninitialize

Syntax:

```
PXL_RETURN_CODE PxLUninitialize(
                                IN HANDLE hCamera);
```

Description:

This function uninitializes the specified camera.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).

Comments, Restrictions and Limitations:

After PxLUninitialize is called, the camera associated with the specified handle cannot be accessed again until a new handle is obtained. Only the PxLInitialize function (p.33) can reinitialize this camera within the application.

4

Definitions, Types and Structures

This section contains all the definitions, enums, typedefs and structures relevant to the PixeLINK API. These definitions are located in the PixeLINKTypes.h header file.

Table 4.1 List of Definitions, Types and Structures

	Definitions/ Structures	Page	Used With Function(s):
Callback Flags	Definitions	47	PxLSetCallback (p.37)
Camera Features	Structures	48	PxLGetCameraFeatures (p.23)
Camera Info	Structure	49	PxLGetCameraInfo (p.25)
Descriptor Update Mode	Definitions	50	PxLCreateDescriptor (p.74), PxLUpdateDescriptor (p.77)
Error Report	Structure	51	PxLGetErrorReport (p.28)
Features	Definitions	52	*Most features are used with the following functions:* PxLGetCameraFeatures (p.23), PxLGetFeature (p.29), PxLSetFeature (p.40) *Exception: FEATURE_MEMORY_CHANNEL is used with:* PxLGetCameraFeatures (p.23), PxLLoadSettings (p.34), PxLSaveSettings (p.36)
Frame Descriptor	Structure	55	PxLFormatImage (p.21), PxLGetNextFrame (p.31)
Feature Flags	Definitions	58	PxLGetFeature (p.29), PxLSetFeature (p.40)
GPIO Mode	Definitions	59	PxLGetFeature (p.29), PxLSetFeature (p.40)
Image File Format	Definitions	60	PxLFormatImage (p.21)
Pixel Format	Definitions	61	PxLGetFeature (p.29), PxLSetFeature (p.40), PxLSetCallback (p.37)
Preview State	Definitions	62	PxLSetPreviewState (p.43)
Stream State	Definitions	63	PxLSetStreamState (p.44).
Trigger Type	Definitions	64	PxLGetFeature (p.29), PxLSetFeature (p.40)
Video Clip File Format	Definitions	65	PxLFormatClip (p.20)

Callback Flags

Type:

Definitions

Description and other comments:

Table 4.2 (below) lists the callback flag definitions.

Table 4.2 Callback Flag Definitions

Value	Description
OVERLAY_PREVIEW	The callback is applied to the data being displayed in the preview window.
OVERLAY_FORMAT_IMAGE	The callback is applied to the data being converted by PxLFormatImage (p.21).
OVERLAY_FORMAT_CLIP	The callback is applied to the data being saved by PxLFormatClip (p.20).
OVERLAY_FRAME	The callback is applied to the raw data being received by the driver.

Used with:

PxLSetCallback (p.37)

Camera Features

Type:

Structures

Description and other comments:

The following structures define information about the supported camera features. The structures are to be located in one contiguous block of memory. Table 4.3 (below) describes the members of these structures.

```
typedef struct _FEATURE_PARAM
{
   float fMinValue;
   float fMaxValue;
}  FEATURE_PARAM, *PFEATURE_PARAM;

typedef struct _CAMERA_FEATURE
{
   U32 uFeatureId;
   U32 uFlags;
   U32 uNumberOfParameters;
   PFEATURE_PARAM pParams;
}  CAMERA_FEATURE, *PCAMERA_FEATURE;

typedef struct _CAMERA_FEATURES
{
   U32 uSize;
   U32 uNumberOfFeatures;
   PCAMERA_FEATURE pFeatures;
}  CAMERA_FEATURES, *PCAMERA_FEATURES;
```

Table 4.3 Members of the FEATURE_PARAM, CAMERA_FEATURE and CAMERA_FEATURES Structures

Member	Description
uSize	The size, in bytes, of the CAMERA_FEATURES structures.
uNumberOfFeatures	The number of entries in the Feature array.
pFeatures	An array of structures describing each feature.
uFeatureId	A value that indicates which feature is being described.
uFlags	A value that indicates which flags the feature supports (example: FEATURE_FLAG_AUTO).
uNumberOfParameters	The number of parameters for this feature, and the size of the pParams array.
pParams	An array of structures describing each parameter
fMinValue	The minimum value for the parameter.
fMaxValue	The maximum value for the parameter.

Used with:

PxLGetCameraFeatures (p.23)

Camera Info

Type:

Structure

Description and other comments:

The following structure defines camera-specific hardware and firmware version information. Table 4.4 (below) describes the members. All members are strings of characters terminated by a '\0'.

```
typedef struct _CAMERA_INFO
{
      U8 VendorName [50];
      U8 ModelName [60];
      U8 Description [80];
      U8 SerialNumber[9];
      U8 FirmwareVersion[12];
      U8 FPGAVersion[3];
      U8 CameraName[33];
} CAMERA_INFO, *PCAMERA_INFO;
```

Table 4.4 Members of the CAMERA_INFO Structure

Member	Description
VendorName	A string identifying the vendor name
ModelName	A string identifying the model name
Description	A textual description of the product.
SerialNumber	A string for the serial number.
FirmwareVersion	A string for the embedded software version. The format is "%02X.%02X.%02X.%02X".
FPGAVersion	A string describing the FPGA version.
CameraName	A string identifying the camera name

Used with:

PxLGetCameraInfo (p.25)

Descriptor Update Mode

Type:

Definitions

Description and other comments:

Table 4.5 (below) lists the descriptor update mode definitions:

Table 4.5 Descriptor Update Mode Definitions

Value	Description
PXL_UPDATE_CAMERA	Camera update mode.
PXL_UPDATE_HOST	Host update mode.

Used with:

PxLCreateDescriptor (p.74), PxLUpdateDescriptor (p.77)

Error Report

Type:

Structure

Description and other comments:

The following structure defines error report information. Table 4.6 (below) describes the members. All members are strings of characters terminated by a '\0'.

```
typedef struct _ERROR_REPORT
{
      PXL_RETURN_CODE uReturnCode;
      S8 strFunctionName[32];
      S8 strReturnCode[32];
      S8 strReport[256];
} ERROR_REPORT, *PERROR_REPORT;
```

Table 4.6 Members of the ERROR_REPORT Structure

Member	Description
uReturnCode	The error code.
strFunctionName	A string identifying the API function that returned the most recent error code (example: "PxLSetFeature").
strReturnCode	A string identifying the error code (example: "ApiUnknownError").
strReport	A string containing a description on the error or a possible cause (example: "Camera must be streaming before calling PxLGetNextFrame").

Used with:

PxLGetErrorReport (p.28)

Features

Type:

Definitions

Description and other comments:

The following features can be accessed through the PixeLINK API, although not all cameras have all features. Most of the features have flags that can be read and modified. All of the features have a value that can be read and modified. See for description of parameters for each feature and the units.

Table 4.7 (below) describes the feature values and their parameters.

Table 4.7 Feature Definitions and Values

Feature	Number of Parameters	Parameters	Units	Restrictions
FEATURE_BRIGHTNESS • Black level of the picture	1	fValue	Percentage	N/A
FEATURE_EXPOSURE • Automatically adjusts several camera features to obtain optimum exposure levels.(that is, autoexposure)	1	fValue	Exposure Value	N/A
FEATURE_SHARPNESS • Sharpness of the picture	1	fValue	Percentage	N/A
FEATURE_WHITE_BAL • White balance	1	fValue	Degrees Kelvin	N/A
FEATURE_HUE • Color phase	1	fValue	Degrees	N/A
FEATURE_SATURATION • Color saturation	1	fValue	Percentage	N/A
FEATURE_GAMMA • Gamma correction	1	fValue	N/A	N/A
FEATURE_SHUTTER • Integration time	1	fValue	Seconds	N/A
FEATURE_GAIN • Camera circuit gain	1	fValue	Decibels (dB)	N/A
FEATURE_IRIS • Mechanical lens iris control	1	fValue	F Number	N/A

Feature	Number of Parameters	Parameters	Units	Restrictions
FEATURE_FOCUS • Lens focus control	1	fValue	Meters	N/A
FEATURE_TEMPERATURE • Temperature inside the camera	1	fValue	Degrees Celcius	N/A
FEATURE_TRIGGER • Trigger delay and mode control	5	fMode	N/A	Integer only
		fType	N/A	Integer only (see Trigger Type, p.64)
		fPolarity	N/A	0 for negative polarity ("Active Low"); 1 for positive polarity ("Active High")
		fDelay	Seconds	N/A
		fParameter	Depends on Mode	N/A
FEATURE_ZOOM • Lens zoom control	1	fValue	Power	N/A
FEATURE_PAN • Camera pan control	1	fValue	Degrees	N/A
FEATURE_TILT • Camera tilt control	1	fValue	Degrees	N/A
FEATURE_OPT_FILTER • Optical filter of the camera lens control	1	fValue	N/A	Integer only
FEATURE_GPIO • General Purpose Input/Output control	6	fStrobeNumber	N/A	Integer only
		fMode	N/A	Integer only (see GPIO Mode, p.59)
		fPolarity	N/A	0 for negative polarity ("Active Low"); 1 for positive polarity ("Active High")
		fParameter1	Depends on Mode	N/A
		fParameter2	Depends on Mode	N/A
		fParameter3	Depends on Mode	N/A
FEATURE_FRAME_RATE • Frame rate of the video stream	1	fValue	Frames/Second	N/A
FEATURE_ROI • Region of interest of the image array	4	fLeft	Pixels	Integer only
		fTop	Pixels	Integer only
		fWidth	Pixels	Integer only
		fHeight	Pixels	Integer only

Feature	Number of Parameters	Parameters	Units	Restrictions
FEATURE_FLIP • Vertical and horizontal flip of the image	2	fHorizontal	N/A (1=On, 0=Off)	1 or 0
		fVertical	N/A (1=On, 0=Off)	1 or 0
FEATURE_DECIMATION • Decimation of the image data	1	fValue	N/A	Integer only
FEATURE_PIXEL_FORMAT • Format of the pixel data	1	fValue	N/A	Integer only (see Pixel Format, p.61)
FEATURE_EXTENDED_SHUTTER • Extended shutter control, allows multiple "knee points"	5	fNumberKnees	N/A	Integer only
		fKnee1	Seconds	N/A
		fKnee2	Seconds	N/A
		fKnee3	Seconds	N/A
		fKnee4	Seconds	N/A
FEATURE_AUTO_ROI • Region of interest for application of AUTO or ONE_SHOT commands	4	fLeft	Pixels	Integer only
		fTop	Pixels	Integer only
		fWidth	Pixels	Integer only
		fHeight	Pixels	Integer only
FEATURE_LOOKUP_TABLE • Lookup table control *FEATURE_LOOKUP_TABLE behaves differently with PxLGetCameraFeatures,* See **Note,** below	1 (See Note, below)	fValue[MAX]	N/A	Integer only
FEATURE_MEMORY_CHANNEL • The memory channel to which the settings can be saved (Read with PxLGetCameraFeatures to determine MAX_MEMORY_CHANNEL)	1	fValue	N/A	Integer only

Note:
PxLGetCameraFeatures (p.23) will report that FEATURE_LOOKUP_TABLE has only one parameter. The maximum value reported for this parameter is actually the number of entries in the lookup table (which is 2^n, where "n" is the number of bits per pixel in the data stream). The range of values for any element in the table is 0 to (number of entries minus one). The number of parameters to specify with PxLGetFeature (p.29) or PxLSetFeature (p.40) when using FEATURE_LOOKUP_TABLE is the number of entries in the table (2^n).

Most features are used with the following functions:

- PxLGetCameraFeatures (p.23)
- PxLGetFeature (p.29)
- PxLSetFeature (p.40)

Exception:

FEATURE_MEMORY_CHANNEL is used with:

- PxLGetCameraFeatures (p.23)
- PxLLoadSettings (p.34)
- PxLSaveSettings (p.36)

Frame Descriptor

Type:

Structure

Description and other comments:

The following structure defines the frame descriptor, which contains the settings for the captured frame. Table 4.8 (on page 57) describes the members.

```
typedef struct _FRAME_DESC
{
   U32 uSize;
   float fFrameTime;
   U32 uFrameNumber;

   struct{
      float fValue;
   } Brightness;

   struct{
      float fValue;
   } AutoExposure;

   struct{
      float fValue;
   } Sharpness;

   struct{
      float fValue;
   } WhiteBalance;

   struct{
      float fValue;
   } Hue;

   struct{
      float fValue;
   } Saturation;

   struct{
      float fValue;
   } Gamma;

   struct{
      float fValue;
   } Shutter;

   struct{
      float fValue;
   } Gain;

   struct{
      float fValue;
   } Iris;
```

```
struct{
   float fValue;
} Focus;

struct{
   float fValue;
} Temperature;

struct{
   float fMode;
   float fType;
   float fPolarity;
   float fDelay;
   float fParameter;
} Trigger;

struct{
   float fValue;
} Zoom;

struct{
   float fValue;
} Pan;

struct{
   float fValue;
} Tilt;

struct{
   float fValue;
} OpticalFilter;

struct{
   float fMode[PXL_MAX_STROBES];
   float fPolarity[PXL_MAX_STROBES];
   float fParameter1[PXL_MAX_STROBES];
   float fParameter2[PXL_MAX_STROBES];
   float fParameter3[PXL_MAX_STROBES];
} GPIO;

struct{
   float fValue;
} FrameRate;

struct{
   float fLeft;
   float fTop;
   float fWidth;
   float fHeight;
} Roi;

struct{
   float fHorizontal;
   float fVertical;
} Flip;

struct{
   float fValue;
```

```
    } Decimation;

    struct{
       float fValue;
    } PixelFormat;

    struct{
       float fKneePoint[PXL_MAX_KNEE_POINTS];
    } ExtendedShutter;

    struct{
       float fLeft;
       float fTop;
       float fWidth;
       float fHeight;
    } AutoROI;

} FRAME_DESC, *PFRAME_DESC;
```

Table 4.8 Members of the FRAME_DESC Structure

Member	Description
uSize	Size of the structure. Set this to sizeof(FRAME_DESC).
fFrameTime	The time point at which the frame was captured, in seconds
uFrameNumber	A unique number (reset to 0 on starting a stream) for each frame good enough to leave the camera
fValue	The value of the feature
fMode	The trigger/GPIO mode
fParameter	The parameter for the trigger
fParameterN	The Nth parameter for the GPIO
fLeft	The left coordinate of the ROI
fTop	The top coordinate of the ROI
fWidth	The width of the ROI
fHeight	The height of the ROI
fHorizontal	The horizontal flip of the image array
fVertical	The vertical flip of the image array
fKneePoint	The knee point for extended shutter control

Used with:

PxLFormatImage (p.21), PxLGetNextFrame (p.31)

Feature Flags

Type:

Definitions

Description and other comments:

Table 4.9 (below) lists the Feature Flag Definitions.

Table 4.9 Feature Flag Definitions

Value	Description
FEATURE_FLAG_PRESENCE	(Read only) When this flag is set, the feature is supported.
FEATURE_FLAG_READ_ONLY	(Read only) When this flag is set, the feature can only be read. Any attempts to set the feature's value will return an error.
FEATURE_FLAG_DESC_SUPPORTED	(Read only) When this flag is set, the feature can be set to different values in different descriptors. Otherwise, the feature has the same value in all descriptors.
FEATURE_FLAG_MANUAL	When this flag is set, the application can control the feature by sending parameters to the camera.
FEATURE_FLAG_AUTO	When this flag is set, the camera controls the feature continuously. No intervention by the camera is possible.
FEATURE_FLAG_ONEPUSH	When this flag is set, the camera sets the feature only once and then returns to manual operation.
FEATURE_FLAG_OFF	When this flag is set, the feature is set to the last known state and cannot be controlled by the application.

Used with:

PxLGetFeature (p.29), PxLSetFeature (p.40)

GPIO Mode

Type:

Definitions

Description and other comments:

Table 4.10 (below) lists the GPIO Mode Definitions

Table 4.10 GPIO Mode Definitions

Value	Description
GPIO_MODE_STROBE	The GPIO acts as a strobe that reacts to the trigger input.
GPIO_MODE_NORMAL	The GPIO can either be set or cleared.
GPIO_MODE_PULSE	A series of pulses can be driven on the GPIO.
GPIO_MODE_BUSY	The GPIO is set whenever the camera is unable to respond to a trigger.

Used with:

PxLGetFeature (p.29), PxLSetFeature (p.40)

Image File Format

Type:

Definitions

Description and other comments:

These are values used to specify an image file format for PxLFormatImage (p.21). Table 4.11 (below) lists the image file format definitions.

Table 4.11 Image File Format Definitions

Value	Description
IMAGE_FORMAT_BMP	The file format is bitmap (.bmp).
IMAGE_FORMAT_TIFF	The file format is TIFF (.tif).
IMAGE_FORMAT_PSD	The file format is the Adobe Photoshop format (.psd).
IMAGE_FORMAT_JPEG	The file format is JPEG (high quality; 5% compression).

Used with:

PxLFormatImage (p.21)

Pixel Format

Type:

Definitions

Description and other comments:

These are values used to specify the format of a pixel. Table 4.12 (below) lists the possible values and their descriptions.

Table 4.12 Pixel Format Definitions

Value	Description
PIXEL_FORMAT_MONO8	Grayscale, 8 bits per pixel
PIXEL_FORMAT_MONO16	Grayscale, 16 bits per pixel, with the first byte is the most significant byte
PIXEL_FORMAT_YUV422	Color, 16 bits per pixel, with a coding pattern of U0, Y0, V0, Y1, U2, Y2, V2, Y3, …
PIXEL_FORMAT_BAYER8	Color, 8 bits per pixel, with data in Bayer pattern format
PIXEL_FORMAT_BAYER16	Color, 16 bits per pixel, with data in Bayer pattern format
PIXEL_FORMAT_RGB24	Color, 24 bits per pixel
PIXEL_FORMAT_RGB48	Color, 48 bits per pixel

Used with:

PxLGetFeature (p.29), PxLSetFeature (p.40), PxLSetCallback (p.37)

Preview State

Type:

Definitions

Description and other comments:

Table 4.13 (below) lists the preview state definitions.

Table 4.13 Preview State Definitions

Value	Description
START_PREVIEW	The preview window is running.
PAUSE_PREVIEW	All resources are allocated for the preview window. The preview window, however, is not running and the host CPU is minimizing CPU cycles on this task.
STOP_PREVIEW	The preview window is stopped and there are no resources allocated to the preview window.

Used with:

PxLSetPreviewState (p.43)

Stream State

Type:

Definitions

Description and other comments:

Table 4.14 (below) lists the stream state definitions.

Table 4.14 Stream State Definitions

Value	Description
START_STREAM	The stream is running.
PAUSE_STREAM	The stream is paused.
STOP_STREAM	The stream is stopped and there are no resources allocated to the preview window.

Used with:

PxLSetStreamState (p.44)

Trigger Type

Type:

Definitions

Description and other comments:

Table 4.15 (below) lists the trigger type definitions.

Table 4.15 Trigger Type Definitions

Value	Description
TRIGGER_TYPE_FREE_RUNNING	Frames are output continuously, driven by an internal trigger that occurs as frequently as possible.
TRIGGER_TYPE_SOFTWARE	Frames are output when PxLGetNextFrame (p.31) is called, which generates a software trigger (using the IIDC One_Shot feature).
TRIGGER_TYPE_HARDWARE	Frames are output on receipt of valid external hardware triggers.

Used with:

PxLGetFeature (p.29), PxLSetFeature (p.40)

Video Clip File Format

Type:

Definitions

Description and other comments:

These are values used to specify a video image file format for PxLFormatClip (p.20). Table 4.16 (below) lists the video clip file format definitions.

Table 4.16 Video Clip File Format Definitions

Value	Description
CLIP_FORMAT_AVI	The file format is standard Windows .avi.

Used with:

PxLFormatClip (p.20)

Appendix A. Glossary

API Application Programming Interface

.bmp, BMP Bitmap

bpp Bits Per Pixel

Camera "Camera" refers to a Version 4.0 PixeLINK camera or imaging module.

DCAM IIDC 1394-based Digital Camera Specification

Descriptor Descriptors are an advanced feature provided in certain cameras. Descriptors are used to change camera settings (or features) on a frame-by-frame basis. Descriptors are cycled through continuously. Copies of the descriptors are kept on the host and on the camera and can be updated on the host before sending the changes to the camera. For a more detailed description, see Section 2.6 (on page 10).

DLL Dynamic Link Library

Feature Camera specific setting or attribute

FireWire Apple Computer's trademark for the IEEE 1394 digital bus protocol

fps Frames Per Second

frame One complete image from the camera

IEEE Institute of Electrical and Electronics Engineers

IIDC 1.3 Instrumentation and Industrial Digital Camera Specification v1.3

MB Megabyte

MHz Megahertz

Pixel Basic unit of light or color information in an image

Preview Window "Preview window" refers to the window used to display images on the screen. The size and position are independent of the imager ROI.

.psd (Adobe) Photoshop data file

RGB Standard for encoding color images (Red, Green, Blue)

RGB24 In this API reference, unless otherwise specified "RGB24" refers to 24-bit RGB pixel data in Windows DIB format—that is, ordered as 8 bits each of blue, green, and red (RGBTRIPLE), and upside down. Several PixeLINK API functions import or export image data in this format. The "RGB24" notation is used as shorthand, both for convenience and to distinguish the pixel size.

UNIX software and some third-party Windows software packages may require "true" RGB24 output ordered as 8 bits each of red, green and blue, and rightside up. Further conversion of the image output, outside of the PixeLINK API, is required for these applications.

48-bit RGB For the convenience of users of that require 48-bit image data, the PixeLINK API exports images in this format. **Note**: Unlike "RGB24", the pixels in 48-bit RGB are ordered as 16 bits each of red, green and blue, and rightside up, suitable for conversion to 48-bit TIFF.

ROI "Region of Interest" refers to the area read out from the camera's imager.

Streaming The API obtains video and still image data from the camera. This video data can be displayed in the preview window or stored as still images or video clips. The imaging area (region of interest or ROI) and preview window can be moved and resized as necessary.

.tif, TIFF Tagged Image File Format

Appendix B. API Return Codes

Return code	Value	Explanation	Possible reasons
ApiSuccess	0x00000000	The function completed successfully.	
ApiUnknownError	0x80000001	Unknown error	Contact PixeLINK for help
ApiInvalidHandleError	0x80000002	The handle parameter is invalid	The camera was not initialized or the initialization was not successful
ApiInvalidParameterError	0x80000003	Invalid parameter	See the appropriate function description to determine the admissible parameter values
ApiBufferTooSmall	0x80000004	A buffer passed as parameter is too small.	If possible, use the function to return the required size of the buffer before filling the buffer
ApiInvalidFunctionCallError	0x80000005	The function cannot be called at this time	The function was called at an incorrect point in the calling sequence; another function may need to be called beforehand
ApiNotSupportedError	0x80000006	The API cannot complete the request	This camera does not support the requested functionality.
ApiCameraInUseError	0x80000007	The camera is already being used by another application	Another application is running and using the camera.
ApiNoCameraError	0x80000008	There is no response from the camera	The camera is disconnected
ApiHardwareError	0x80000009	The Camera responded with an error	Unknown hardware error—contact PixeLINK for help
ApiCameraUnknownError	0x8000000A	The API does not recognize the camera	Incompatible firmware, driver or API version
ApiOutOfBandwidthError	0x8000000B	There is not enough 1394 bandwidth to start the stream	Too many other devices are using the bus, or too much video data is streaming
ApiOutOfMemoryError	0x8000000C	The API can not allocate the required memory	The system is out of memory
ApiOSVersionError	0x8000000D	The API cannot run on the current operating system	The current version of the operating system is not supported by the Kit software

Return code	Value	Explanation	Possible reasons
ApiNoSerialNumberError	0x8000000E	The serial number could not be obtained from the camera	
ApiInvalidSerialNumberError	0x8000000F	A camera with that serial number could not be found	
ApiDiskFullError	0x80000010	Not enough disk space to complete an IO operation	
ApiIOError	0x80000011	An error occurred during an IO operation	
ApiStreamStopped	0x80000012	Application requested streaming termination	
ApiNullPointerError	0x80000013	The pointer parameter=NULL	The necessary memory buffer has not been allocated
ApiCreatePreviewWndError	0x80000014	Error creating the preview window	
ApiSuccessParametersChanged	0x00000015	Indicates that a set feature is successful but one or more parameter had to be changed (ROI)	
ApiOutOfRangeError	0x80000016	Indicates that a feature set value is out of range	
ApiNoCameraAvailableError	0x80000017	There is no camera available	
ApiInvalidCameraName	0x80000018	Indicated that the name specified is not a valid camera name	
ApiGetNextFrameBusy	0x80000019	PxLGetNextFrame (p.31) cannot be called at this time because it is in use by an overlay callback function	

Appendix C. PixeLINK Data Stream File Format

The PixeLINK Data Stream file (.pds) is a binary file with the following format:

<START OF FILE>

[a U32 constant indicating this is a .pds file (0x04040404)]

[a U32 stating the number of frames in the file]

[FRAME_DESC structure for Frame #1]

[Data for Frame #1 (size can be calculated from FRAME_DESC)]

[FRAME_DESC structure for Frame #2]

[Data for Frame #2]

…

[FRAME_DESC structure for Frame N]

[Data for Frame N]

<END OF FILE>

That is:

```
{
   CONST U32    MajicNumber=0x04040404;
         U32    NumberOfFrames;
         {
            FRAME_DESC  Descriptor;
            PVOID       FrameData[FrameSize];
         }  Frames [NumberOfFrames];
};
```

Appendix D. Advanced Functions—Creating and Using Custom Descriptors

D.1 Overview

A descriptor is a collection of camera feature properties that is applied to a frame. By defining multiple descriptors, the camera properties can be changed automatically on a frame-by-frame basis for each frame in the video stream. A copy of each descriptor is kept on the host computer and can be updated on the host before sending changes to the camera.

Descriptors are applied to the video stream in the order in which they were created. That is, the first descriptor created is applied to the first frame returned after the start of the video stream, the second descriptor created is applied to the second frame, and so on. If the number of frames exceeds the number of descriptors, the descriptors are applied to the remaining frames in a cyclic fashion (that is, after the last descriptor is applied, the first descriptor is applied to the next frame, then the second descriptor is applied to the following frame, and so on).

Note that a descriptor is applicable only to the camera for which it was created.

D.2 Camera and Host Update Modes

Because of the time required to communicate and enable property changes between the host computer and the camera, the PixeLINK API offers the ability to change a set of descriptor properties on the host computer first, then update the new set of properties on the camera in a single operation. This greatly reduces the risk of streaming frames that incorporate partially applied property changes.

When including descriptor controls in an interactive GUI, the host update mode can act as a "scratch pad" while the user changes the descriptor properties within the GUI window. The set of properties can then be updated on the camera in one operation with a single button click. This technique is used in the Descriptor tab of the *PixeLINK Developers Application*.

The update mode is set when the descriptor is created (see Section D.3 on page 71). It can be changed by updating the descriptor (see Section D.6 on page 71).

Host Mode: All function calls to change or read the feature properties are done to or from a cache on the host computer. They do not affect the camera settings until the update mode is changed to camera mode.

Camera Mode: This is the default update mode. All function calls to change or read the feature properties are done to or from the camera itself. If the update mode is changed from host mode to camera mode when updating the descriptor, the camera is updated with the descriptor's property set cached on the host computer.

D.3 Creating a Descriptor

Create a new descriptor using PxLCreateDescriptor (on page 74). For information on update modes, see Section D.2 (on page 70).

D.4 Bringing a Descriptor in Focus

To read or change the settings of a descriptor, it must be **in focus.** At any given time, either a single descriptor can be in focus or all descriptors can be in focus simultaneously.

A descriptor can be put in focus in one of three ways:

- By calling PxLUpdateDescriptor (on page 77) and specifying the descriptor to be brought in focus (this function is also used bring all descriptors in focus simultaneously);
- By creating a new descriptor using PxLCreateDescriptor (on page 74)—This descriptor stays in focus until another descriptor is created using PxLCreateDescriptor or brought in focus using PxLUpdateDescriptor; or
- By deleting the descriptor currently in focus—the focus reverts to the *first* descriptor created. (Descriptors are deleted using PxLRemoveDescriptor, on page 76.)

D.5 Reading Descriptor Settings

1. Bring the descriptor in focus (PxLUpdateDescriptor, on page 77). A newly created descriptor is automatically in focus until PxLUpdateDescriptor or PxLCreateDescriptor (on page 74) is called.
2. Call PxLGetFeature (on page 29) to read the settings.

Note that if all descriptors are in focus at once (rather than just a single descriptor), calling PxLGetFeature will generate an error.

D.6 Changing Descriptor Settings

1. Bring the descriptor in focus (PxLUpdateDescriptor, on page 77). (To bring all descriptors in focus simultaneously, set the descriptor handle to NULL.) A new

descriptor is automatically in focus until PxLUpdateDescriptor or PxLCreateDescriptor (on page 74) is called.

2. Call PxLSetFeature (on page 40) to change the settings.

PxLUpdateDescriptor is also used to change the descriptor's update mode. For information on update modes, see Section D.2 (on page 70).

D.7 Deleting a Descriptor

- Call PxLRemoveDescriptor (on page 76) to delete a descriptor.

Focus: If the descriptor currently in focus is deleted, the focus reverts to the *first* descriptor created.

All descriptors deleted: If all custom descriptors are deleted (that is, no descriptors are defined), the descriptor stored with any image data subsequently viewed or captured is constructed from the camera feature properties in effect at the time.

D.8 Features Not Controlled By Descriptors

Not all features can be controlled by descriptors.

If the FEATURE_FLAG_DESC_SUPPORTED flag is set for a feature, the feature can be controlled by descriptors. Otherwise, the feature will have the same properties in all descriptors.

Some features cannot have the FEATURE_FLAG_DESC_SUPPORTED flag set. These features include FEATURE_LOOKUP_TABLE (on page 54) and FEATURE_AUTO_ROI (on page 54).

Feature flags are described on page 58.

D.9 Examples

D.9.1 Example 1 – Creating Two Simple Descriptors

```
// Create two descriptors
PxLCreateDescriptor ( hCamera, &hDesc1, PXL_UPDATE_CAMERA );
PxLCreateDescriptor ( hCamera, &hDesc2, PXL_UPDATE_CAMERA );

// Set the focus to Descriptor 1
PxLUpdateDescriptor ( hCamera, hDesc1, PXL_UPDATE_CAMERA );

// Change the integration time to xx ms
PxLSetFeature ( hCamera, FEATURE_SHUTTER, … );
```

```
// Set the focus to Descriptor 2
PxLUpdateDescriptor ( hCamera, hDesc2, PXL_UPDATE_CAMERA );

// Change the integration time to yy ms
PxLSetFeature ( hCamera, FEATURE_SHUTTER, … );

// Start the camera streaming
PxLSetStreamState ( hCamera, START_STREAM );
```

The camera will now be streaming data, alternating between two frames, one at xx ms of integration time and the other at yy ms of integration time.

D.9.2 Example 2 – Using a Descriptor to Change Many Settings At Once

```
// Create one descriptor
PxLCreateDescriptor ( hCamera, &hDesc, PXL_UPDATE_CAMERA );

// Set the focus for the descriptor and set it to Host update mode
PxLUpdateDescriptor ( hCamera, hDesc, PXL_UPDATE_HOST );

// Set several features in the descriptor
PxLSetFeature ( hCamera, FEATURE_SHUTTER, … );
PxLSetFeature ( hCamera, FEATURE_GAMMA, … );
PxLSetFeature ( hCamera, FEATURE_ROI, … );
…

// Set the descriptor to Camera update mode
PxLUpdateDescriptor ( hCamera, hDesc, PXL_UPDATE_CAMERA );
```

[All the changes to the feature settings are applied at once here.]

```
…
// Remove the descriptor
PxLRemoveDescriptor ( hCamera, &hDesc);
```

D.10 API Descriptor Functions

Three advanced functions, listed in Table D.1 (below) directly control descriptor creation and management.

Table D.1 API Descriptor Functions

API Function Name	Description	Page
PxLCreateDescriptor()	Create a descriptor for the specified camera	74
PxLUpdateDescriptor()	Set the update mode for the specified descriptor	77
PxLRemoveDescriptor()	Remove the descriptor from the specified camera	76

PxLCreateDescriptor

Advanced Feature

Syntax:

```
PXL_RETURN_CODE PxLCreateDescriptor(
                                IN HANDLE hCamera,
                                OUT HANDLE* pDescriptorHandle,
                                IN U32 uUpdateMode);
```

Description:

This function creates a descriptor on the host computer or on the camera, depending on the update mode specified. This descriptor is created using the current settings on the host.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *pDescriptorHandle* is a pointer to a variable receiving the descriptor handle.
- *uUpdateMode* is the initial update mode of the descriptor being created. Valid values are:

  PXL_UPDATE_CAMERA — Camera update mode.

  PXL_UPDATE_HOST — Host update mode.

  These values are defined in the file PixeLINKTypes.h. See below for more information.

Default:

PXL_UPDATE_CAMERA

Comments, Restrictions and Limitations:

For more information about descriptors, including definitions of the terms used in this function description and examples of creating and using descriptors, refer to the text starting on page 70.

Descriptors are applied to the video stream in the order in which they were created. That is, the first descriptor created is applied to the first frame returned after the start of the video stream, the second descriptor created is applied to the second frame, and so on. If the number of frames exceeds the number of descriptors, the descriptors are applied to the remaining frames in a cyclic fashion (that is, after the last descriptor is applied, the first descriptor is applied to the next frame, then the second descriptor is applied to the following frame, and so on).

A descriptor is applicable only to the camera for which it was created.

Focus:

The newly created descriptor is in focus (that is, its settings can be changed using PxLSetFeature, p.40, or read using PxLGetFeature, p.29) until one of the following occurs:

- another descriptor is created,
- another descriptor is brought in focus using PxLUpdateDescriptor, or
- the descriptor is deleted using PxLRemoveDescriptor (p.76).

All descriptors can be brought in focus simultaneously by calling PxLUpdateDescriptor with the descriptor handle set to NULL. Otherwise, only one descriptor may be in focus at a time. Note that if all descriptors are in focus at once (rather than just a single descriptor), calling PxLGetFeature will generate an error.

Update mode:

If *uUpdateMode* is set to PXL_UPDATE_HOST, then the descriptor is created (cached) on the host computer only. It will not be applied to the camera until PxLUpdateDescriptor (p.77) is called with the update mode set to PXL_UPDATE_CAMERA.

If *uUpdateMode* is set to PXL_UPDATE_CAMERA, then the descriptor is cached on the host computer and applied immediately to the camera.

See PxLUpdateDescriptor for a more detailed description of the update modes.

PxLRemoveDescriptor

Advanced Feature

Syntax:

```
PXL_RETURN_CODE PxLRemoveDescriptor (
                                IN HANDLE hCamera,
                                IN HANDLE hDescriptor);
```

Description:

This function removes a descriptor from the camera and host.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *phDescriptor* is the a pointer to the handle of the descriptor to be removed. This handle is returned by PxLCreateDescriptor (p.74). Set to NULL to remove all descriptors at once.

Comments, Restrictions and Limitations:

For more information about descriptors, including examples of creating and using descriptors, refer to the text starting on page 70.

If a descriptor is in focus when it is removed, the focus reverts to the first descriptor created.

All descriptors deleted: If all custom descriptors are deleted (that is, no descriptors are defined), the descriptor stored with any image data subsequently viewed or captured is constructed from the camera feature properties in effect at the time.

PxLUpdateDescriptor

Advanced Feature

Syntax:

```
PXL_RETURN_CODE PxLUpdateDescriptor(
                                IN HANDLE hCamera,
                                IN HANDLE hDescriptor,
                                IN U32 uUpdateMode);
```

Description:

This function puts a specified descriptor in focus and sets its update state.

- *hCamera* is the camera handle. This value is returned by PxLInitialize (p.33).
- *hDescriptor* is the descriptor handle. This value is returned by PxLCreateDescriptor (p.74). Set to NULL to select all descriptors.
- *uUpdateMode* indicates the update mode for this descriptor. . Valid values are:

  PXL_UPDATE_CAMERA — Camera update mode.

  PXL_UPDATE_HOST — Host update mode.

  These values are defined in the file PixeLINKTypes.h.

Comments, Restrictions and Limitations:

For more information about descriptors, including definitions of the terms used in this function description and examples of creating and using descriptors, refer to the text starting on page 70.

A descriptor is applicable only to the camera for which it was created.

The specified descriptor is in focus (that is, its settings can be changed using PxLSetFeature, p.40, or read using PxLGetFeature, p.29) until one of the following occurs:

- a new descriptor is created using PxLCreateDescriptor (p.74),
- another descriptor is brought in focus using PxLUpdateDescriptor, or
- the descriptor is deleted using PxLRemoveDescriptor (p.76), in which case the focus reverts to the first descriptor created.

When a descriptor exists and is in focus, all calls of PxLGetFeature (p.29) and PxLSetFeature (p.40) refer to the settings of that descriptor.

Bringing all descriptors into focus simultaneously:

All descriptors may be brought in focus simultaneously by setting *hDescriptor* to NULL. Otherwise, only one descriptor may be in focus at a time. Note that if all descriptors are in focus at once (rather than just a single descriptor), calling PxLGetFeature will generate an error.

Update mode:

If *uUpdateMode* is set to PXL_UPDATE_HOST, then the descriptor is updated on the host computer only. Changes will not be applied to the camera until PxLUpdateDescriptor is called for that descriptor with *uUpdateMode* set to PXL_UPDATE_CAMERA.

If *uUpdateMode* is set to PXL_UPDATE_CAMERA, then the descriptor is updated on the host computer and changes are applied immediately to the camera.

If the descriptor update mode was originally set to PXL_UPDATE_HOST and *uUpdateMode* is set to PXL_UPDATE_CAMERA, the descriptor is transferred to the

camera and all the changes to settings are applied at once. This is done using the PixeLINK extension to IIDC.

Recommendation: Update the descriptor settings in PXL_UPDATE_HOST mode, then call PxLUpdateDescriptor with *uUpdateMode* set to PXL_UPDATE_CAMERA to apply all changes to the camera settings at once. After this is done, return the descriptor to host mode by calling PxLUpdateDescriptor with *uUpdateMode* set to PXL_UPDATE_HOST.

Appendix E. IIDC 1.3 and PixeLINK Extensions

The IIDC 1394-based Digital Camera Specification (Version 1.30) requires that the camera hardware retain a record of all the features the camera supports. Software and drivers used to control the camera can query the camera to get a list of supported features and other information. The IIDC specification allows for the definition of advanced features that are camera specific.

Features Controlled by PixeLINK Extensions to IIDC Specifications

Camera information retrieval: This feature reports the camera's hardware and firmware information, specifically the firmware version, the FPGA version, the camera's serial number, and the product ID.

Trigger features: The standard IIDC specification is used whenever possible. However, the standard does not allow the time to be set between a trigger event and the start of integration, so a PixeLINK extension is used to enable this feature.

GPIOs: The IIDC specification does not apply to strobes, flashes, or LEDs. A PixeLINK extension allows control of multiple strobes and a general purpose I/O. The extension also allows control of the time between a trigger event and the activation of a strobe, the duration of the strobea and the polarity of the strobe signal.

Color coding: The IIDC specification allows a color camera to output only YUV and RGB formats. A PixeLINK extension accommodates raw (Bayer) color output.

Extended shutter (Knee points): This feature allows for multiple-slope-exposure captures to enable a larger dynamic range.

Lookup Table (LUT): This feature allows a user-specified lookup table to be applied to the image data, enabling custom filtering and image processing.

Automatic Region of Interest (Auto ROI): This feature applies Auto or One Shot settings (like White Balance) to a specific area within the current region of interest (ROI).

Descriptors: A descriptor is a collection of camera feature properties that is applied to a frame. Since each frame can have a unique descriptor, custom descriptors can be used to change camera settings on a frame-by-frame basis for each frame in the video stream.

Appendix F. Relationship Between R3.1 and R4.0 API Functions

The following pages list the relationships between the functions in the previous PixeLINK Camera API (Release 3.1, shaded) and the current PixeLINK Camera API (Release 4.0).

Detect Cameras

PimMegaGetNumberDevices

USE: PxLGetNumberCameras **PAGE:** 32

Initialization

PimMegaInitialize

USE: PxLInitialize **PAGE:** 33

Identification

HARDWARE/ SOFTWARE

PimMegaGetImagerType

USE: PxLGetCameraInfo **PAGE:** 25

PimMegaGetHardwareVersion

USE: PxLGetCameraInfo **PAGE:** 25

NAMING

PimMegaGetImagerName	PimMegaSetImagerName
USE: PxLGetCameraInfo **PAGE:** 25	**USE:** PxLSetCameraName **PAGE:** 39

Stream Specifications

FRAME RATE

PimMegaGetCurrentFrameRate

USE: PxLGetFeature **PAGE:** 29
Usage: Select FEATURE_FRAME_RATE (p.53)

DATA TRANSFER SIZE

PimMegaGetDataTransferSize	PimMegaSetDataTransferSize
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_PIXEL_FORMAT (p.54). See Pixel Format (p.61)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_PIXEL_FORMAT (p.54) and one of PIXEL_FORMAT_MONO8, PIXEL_FORMAT_MONO16, PIXEL_FORMAT_BAYER8, PIXEL_FORMAT_BAYER16. See Pixel Format (p.61)

CLOCK SPEED

PimMegaGetImagerClocking	PimMegaSetImagerClocking
USE: Not applicable	**USE:** Not applicable

TIMEOUT PERIOD

PimMegaGetTimeout	PimMegaSetITimeout
USE: Not applicable	**USE:** Not applicable

Subwindow Coordinates and Decimation

SUBWINDOW SIZE, POSITION, AND DECIMATION

PimMegaGetSubWindow	PimMegaSetSubwindow
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_ROI (p.53) and FEATURE_DECIMATION (p.54)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_ROI (p.53) and FEATURE_DECIMATION (p.54)

SUBWINDOW POSITION

PimMegaGetSubWindowPos	PimMegaSetSubwindowPos
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_ROI (p.53)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_ROI (p.53)

SUBWINDOW SIZE AND DECIMATION

PimMegaGetSubWindowSize	PimMegaSetSubwindowSize
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_ROI (p.53) and FEATURE_DECIMATION (p.54)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_ROI (p.53) and FEATURE_DECIMATION (p.54)

Video Stream Controls

PimMegaStartVideoStream

USE: PxLSetStreamState **PAGE:** 44
Usage: Select START_PREVIEW_STATE. See Preview State (p.62)

PimMegaPauseVideoStream

USE: PxLSetStreamState **PAGE:** 44
Usage: Select PAUSE_PREVIEW_STATE. See Preview State (p.62)

PimMegaStopVideoStream

USE: PxLSetStreamState **PAGE:** 44
Usage: Select STOP_PREVIEW_STATE. See Preview State (p.62)

Video Mode

PimMegaGetVideoMode	PimMegaSetVideoMode
USE: Not applicable	**USE:** Not applicable

Overlay Callback

PimMegaSetOverlayCallBack

USE: PxLSetCallback **PAGE:** 37

Image Control

GAMMA

PimMegaGetGamma	PimMegaSetGamma
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_GAMMA (p.52)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_GAMMA (p.52)

FLIP

PimMegaGetImageFlip	PimMegaImageFlip
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_FLIP (p.54)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_FLIP (p.54)

EXPOSURE

PimMegaAutoExposure

USE: PxLSetFeature **PAGE:** 40
Usage: Select FEATURE_SHUTTER (p.52) and FEATURE_FLAG_ONEPUSH (see Feature Flags (p.58)

PimMegaGetExposureGain	PimMegaSetExposureGain
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_GAIN (p.52)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_GAIN (p.52)
PimMegaGetExposureTime	PimMegaSetExposureTime
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_SHUTTER (p.52)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_SHUTTER (p.52)

PimMegaGetMaxExposureTime

USE: PxLGetCameraFeatures **PAGE:** 23
Usage: Select FEATURE_SHUTTER (p.52)

PimMegaRequestContinuousAutoExposure

USE: PxLGetFeature **PAGE:** 29
Usage: Select FEATURE_SHUTTER (p.52) and FEATURE_FLAG_AUTO (see Feature Flags (p.58)

WHITE BALANCE

PimMegaWhiteBalance

USE: PxLSetFeature **PAGE:** 40
Usage: Select FEATURE_WHITE_BAL (p.52) and FEATURE_FLAG_AUTO (see Feature Flags (p.58)

BLUE GAIN

PimMegaGetBlueGain	PimMegaSetBlueGain
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_HUE (p.52), FEATURE_WHITE_BAL (p.52)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_HUE (p.52), FEATURE_WHITE_BAL (p.52)

GREEN GAIN

PimMegaGetGreenGain	PimMegaSetGreenGain
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_HUE (p.52), FEATURE_WHITE_BAL (p.52)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_HUE (p.52), FEATURE_WHITE_BAL (p.52)

RED GAIN

PimMegaGetRedGain	PimMegaSetRedGain
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_HUE (p.52), FEATURE_WHITE_BAL (p.52)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_HUE (p.52), FEATURE_WHITE_BAL (p.52)

MONO GAIN

PimMegaGetMonoGain	PimMegaSetMonoGain
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_GAIN (p.52)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_GAIN (p.52)

SATURATION

PimMegaGetSaturation	PimMegaSetSaturation
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_SATURATION (p.52)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_SATURATION (p.52)

LIGHT SOURCE

PimMegaGetLightSource	PimMegaSetLightSource
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_WHITE_BAL (p.52)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_WHITE_BAL (p.52)

Image Acquisition

RAW FRAME DATA

PimMegaReturnVideoData

USE: PxLGetNextFrame **PAGE:** 31

PimMegaReturnStillFrame

USE: PxLGetNextFrame **PAGE:** 31

PREVIEW WINDOW CONTROLS

PimMegaStartPreview

USE: PxLSetPreviewState **PAGE:** 43
Usage: Select START_PREVIEW. See Preview State (p.62)

PimMegaPausePreview

USE: PxLSetPreviewState **PAGE:** 43
Usage: Select PAUSE_PREVIEW. See Preview State (p.62)

PimMegaStopPreview

USE: PxLSetPreviewState **PAGE:** 43
Usage: Select STOP_PREVIEW. See Preview State (p.62)

PimMegaSetPreviewColorConversion

USE: Not applicable

PREVIEW WINDOW SIZE AND POSITION

PimMegaSetPreviewWindow

USE: PxLSetPreviewSettings **PAGE:** 41

PimMegaGetPreviewWindowPos

USE: Not applicable

PimMegaGetPreviewWindowSize

USE: Not applicable

PimMegaResetPreviewWindow

USE: PxLResetPreviewWindow **PAGE:** 35

Image Capture

DIRECTLY FROM CAMERA

PimMegaCaptureFrameToBitmap

USE: PxLGetNextFrame **PAGE:** 31
PxLFormatImage **PAGE:** 21
Usage: For PxlFormatImage, select IMAGE_FORMAT_BMP. See Image File Format (p.60).

FROM FRAME BUFFER

PimMegaSaveFrameAsBitmap

USE: PxLFormatImage **PAGE:** 21
Usage: Select IMAGE_FORMAT_BMP. See Image File Format (p.60).

PimMegaSaveFrameAsJpeg

USE: PxLFormatImage **PAGE:** 21
Usage: Select IMAGE_FORMAT_JPEG. See Image File Format (p.60).

PimMegaSaveFrameAsTiff

USE: PxLFormatImage **PAGE:** 21
Usage: Select IMAGE_FORMAT_TIFF. See Image File Format (p.60).

Video Data Buffers

PimMegaFlushVideoData

USE: Not applicable

Raw to RGB Conversion

PimMegaConvertColor16BppTo24Bpp

USE: PxLFormatImage **PAGE:** 21

PimMegaConvertColor16BppTo48Bpp

USE: PxLFormatImage **PAGE:** 21

PimMegaConvertColor8BppTo24Bpp

USE: PxLFormatImage **PAGE:** 21

PimMegaConvertMono16BppTo24Bpp

USE: PxLFormatImage **PAGE:** 21

PimMegaConvertMono8BppTo24Bpp

USE: PxLFormatImage **PAGE:** 21

Enhanced Control Commands

REGISTERS

PimMegaReadExtI2CRegister	PimMegaWriteExtI2CRegister
USE: Not applicable	**USE:** Not applicable

SERIAL CONTROL

PimMegaReadCommand	PimMegaWriteCommand
USE: Not applicable	**USE:** Not applicable

GPO, GPIO

PimMegaGetGpo	PimMegaSetGpo
USE: PxLGetFeature **PAGE:** 29 **Usage:** Select FEATURE_GPIO (p.53)	**USE:** PxLSetFeature **PAGE:** 40 **Usage:** Select FEATURE_GPIO (p.53)
PimMegaReadGpio	PimMegaWriteGpio
USE: Not applicable	**USE:** Not applicable

External Trigger Commands

PimMegaReturnFrameAfterTrigger

USE: PxLSetFeature **PAGE:** 40
PxLGetNextFrame **PAGE:** 31
Usage: For PxlSetFeature, select FEATURE_TRIGGER (p.53) (see Trigger Type (p.64)). Select the appropriate feature flag when putting the camera into trigger mode (see Feature Flags (p.58)).

PimMegaStopTriggerMode

USE: Not Applicable *See PimMegaReturnFrameAfterTrigger (above).*

Uninitialization

PimMegaUninitialize

USE: PxLUninitialize **PAGE:** 45

Technical Support

Tech Notes

Tech Notes on a variety of topics are available on our Web site, at http://www.pixelink.com/.

Please follow these steps before contacting our technical support team:

1. Review the relevant sections of the documentation.
2. Check the PixeLINK Web site for applicable Tech Notes.
3. Carefully document the problem you are experiencing, noting any warning or error messages that may appear during operation.
4. Have your camera's serial number and other identifying information at hand. The serial number can be found on the camera; other information can be found on the "About" screen in PixeLINK applications or via the PixeLINK Camera API function PxLGetCameraInfo (on page 25)

PixeLINK Customer Support

Email: support@pixelink.com

Web: http://www.pixelink.com/

Sample Code Available

Source code for the *PixeLINK Developer's Application* is available as a ZIP file on the PixeLINK SDK (Developer's Kit) CD-R and at

http://support.pixelink.com/

Index