PHOENIX

USER MANUAL

Active Silicon

v1.6 14-Sep-2010

Revision History

Version	Comments
v1.0 14-Oct-2005	First Release.
v1.1 25-Oct-2005	Updated the Acquisition Trigger Control section.
v1.2 28-Mar-2007	Added PCI Express boards. Added block diagrams to Camera Control section.
v1.3 2-May-2008	Added PoCL LED section.
v1.4 16-Jul-2008	Updated some example code.
v1.5 05-Jan-2010	Updated Phoenix board overview. New US address.
v1.6 14-Sep-2010	New US address.

Disclaimer

Copyright Notice

Trademarks

Part Information

Part Number: PHX-MAN-USER

Version v1.6 14-Sep-2010

Contact Details

Web www.activesilicon.com
Sales sales@activesilicon.com
Support phoenix.support@activesilicon.com

Europe:

Active Silicon Limited
Pinewood Mews, Bond Close, Iver,
Bucks, SL0 0NA, UK.

Tel: +44 (0)1753 650600
Fax: +44 (0)1753 651661

North America:

Active Silicon, Inc.
479 Jumpers Hole Road, Suite 301
Severna Park, MD 21146, USA

Tel +1 410-696-7642
Fax +1 410-696-7643

Table of Contents

Introduction

SCOPE

This document provides overview information regarding the Phoenix series of frame grabbers and answers some of the most frequently asked questions posed to phoenix.support@activesilicon.com.

DOCUMENT STRUCTURE

The document is broken down into the following major sections

- Documentation Overview
- Phoenix Board Overview
- SDK Overview
- SDK Examples
- Phoenix Board Control
- SDK Concepts
- Camera Control
- Acquisition Control
- Image Control

Documentation Overview

GETTING STARTED

- **Quickstart Guide**. There are a number of quickstart guides for using Phoenix on various operating systems.
- **Installation Guide** (part number PHX-MAN-IG). This document describes how to install the software, configure your computer and get your Phoenix board running.
- **User Manual** (part number PHX-MAN-USER). This document!

FRAME GRABBER DATASHEETS

- **D24CL PCI Camera Link.** The *Phoenix-D24CL* is a PCI board for the acquisition of digital data from a variety of Camera Link sources, including digital frame capture and line scan cameras. It supports all the formats of Base configuration, i.e. single 8 to 16 bit data, through 8 bit RGB, to dual tap 12 bit sources.
- **D24CL PMC Camera Link.** This is the PMC version of the above board.
- **D24CL PC/104-Plus Camera Link.** This is the PC/104-Plus version of the above board.
- **D24CL PCI Express Camera Link.** This is the PCI Express version of the above board.
- **D36 PCI LVDS.** The *Phoenix-D36* is a PCI board for the acquisition of digital data from a variety of sources, including digital frame capture and line scan cameras. It has 36 bits used for input data, with 4 bits for control. This provides support for a single 12 bit RGB or 32 bit mono data source, including multi-tap cameras. The 4 bit control inputs are dedicated as Frame Enable, Line Enable, Data Enable and Pixel Clock. Alternatively four of the data inputs can be re-assigned as an additional control port, thus allowing two independent 16 bit mono cameras to be supported. Data widths up to the above maximums are also handled, i.e. 8, 10 or 12 bit RGB and 8, 10, 12, 14, 16 or 32 bit mono.
- **D36 PCI Express .** This is the PCI Express version of the above board.
- **D48CL PCI Camera Link.** The *Phoenix-D48CL* is a PCI board for the acquisition of digital data from a variety of Camera Link sources, including digital frame capture and line scan cameras. It supports all the formats of the Base and Medium configurations, i.e. single 8 to 16 bit data, through 12 bit RGB, to four tap 12 bit sources, as well as dual Base configuration, i.e. acquisition from two asynchronous Base cameras.
- **D48CL CPCI Camera Link.** This is the CompactPCI version of the above board.
- **D48CL PCI Express Camera Link.** This is the PCI Express version of the above board.
- **D48CL PCI/104-Express Camera Link.** This is a PCI/104-Express version of the above board.
- **D64CL PCI Express Camera Link.** The *Phoenix-D64CL* is a PCI Express board which supports acquisition from Base, Medium and Full Camera Link configurations, i.e. single tap 8 to 16 bit data, through 12-bit RGB to eight tap 8 bit sources, as well as ten tap 8-bit and eight tap 10-bit configurations.
- **D10HDSDI PCI Express HD-SDI.** The *Phoenix-D10HDSDI* is a PCI Express board for the acquisition of SDI and HD-SDI data from a variety of sources. It supports HD-SDI video sources up to and including 1080i@60 data rates.
- **D20HDSDI PCI Express HD-SDI.** This is a dual channel version of the above boards. It supports two HD-SDI video sources up to and including 1080i@60 data rates.

HARDWARE MANUALS

- **Interface Guide** (part number PHX-MAN-IFG). This manual describes how to interface a Phoenix Digital product to cameras and other external devices, i.e. acquisition trigger sources, shaft encoders, PLCs, etc. It details information on product pinouts, standard available cables and electrical specifications.
- **Hardware Guide PC/104-Plus**. This document explains the issues involved when using Phoenix in a PC/104-Plus system.
- **Approvals Manual** (part number PHX-MAN-APPR). This document contains EMC statements and Compliance notices.
- **Hardware Failure Report Form**.

SOFTWARE MANUALS

- **Phoenix Library API Reference Manual** (part number PHX-MAN-API). This document describes the functional specification of the PHOENIX software library. This library provides a platform independent interface to the acquisition and control features of a Phoenix image capture device.
- **Buffer Library API Reference Manual** (part number PBL-MAN-API). This document describes the functional specification of the Phoenix Buffer Library (PBL). This library allows the user to easily control the data buffers involved when using a Phoenix image capture device.
- **Image Library API Reference Manual** (part number PIL-MAN-API). This document describes the functional specification of the Phoenix Image Library (PIL). This library allows the user to easily process the data buffers involved when using a Phoenix image capture device.
- **Display Library API Reference Manual** (part number PDL-MAN-API). This document describes the functional specification of the Phoenix Display Library (PDL). This library allows the user to easily display images captured using a Phoenix image capture device. PDL is closely coupled to the Phoenix image capture library (PHX), configuring itself using the parameters of the associated Phoenix instance.
- **Control Class API Reference Manual** (part number PCC-MAN-API). This document describes the functional specification of the Phoenix Control Class (PCC). This class provides a property sheet control, which allows the user to easily control the Phoenix image capture device.
- **VxWorks Library Developer's Manual** (part number PHX-MAN-VXW). This manual describes the low-level functions of the VxWorks driver library for the Phoenix series of acquisition cards. This library provides a hardware platform independent method of accessing Phoenix cards, but allows the user to add enhanced functionality specific to a particular platform via a set of installable callout functions.

Phoenix Board Overview

VARIANTS AND FORM FACTORS

The following products are available in the Phoenix range:

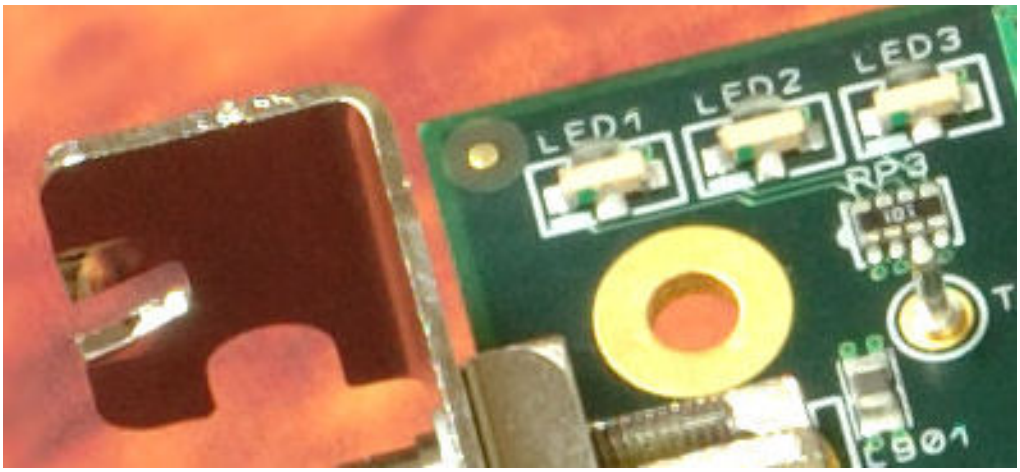
- **D24CL** : Base only Camera Link frame grabber,
- **D48CL** : Base, Dual Base and Medium Camera Link frame grabber,
- **D10HDSDI** : Single input HD-SDI frame grabber,
- **D20HDSDI** : Dual input HD-SDI frame grabber,
- **D36** : 36-bit LVDS frame grabber

The Phoenix boards are available in PCI Express, PCI, CompactPCI, PMC, PCI/104-Express and PC/104-Plus form factors.

FEATURE LIST

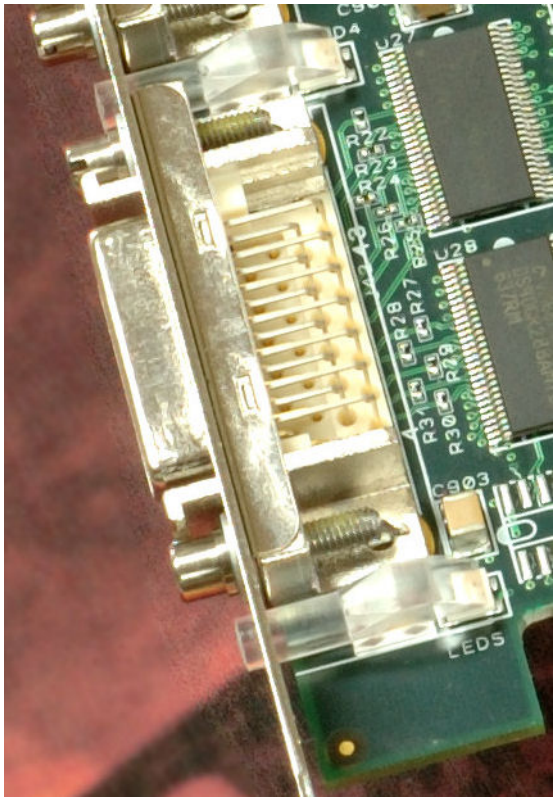
	PHX-D24CL	PHX-D48CL	PHX-D64CL	DxHDSDI	PHX-D36
PCI Express (x1)	✓	✓	-	✓	✓
PCI Express (x4)	-	✓	✓	✓	
PCI (32-bit / 33MHz)	✓	-	✓	-	✓
LVDS Input	-	-	-	-	36-bit
Camera Link Base	1	2	1	-	-
Camera Link Medium	-	1	1	-	-
Camera Link Full	-	-	1	-	-
HD-SDI up to 1080i60.	-	-	-	✓	-
Max Pixel Clock	85MHz	85MHz	60MHz	-	60MHz
LVDS I/O	4	4	4	4	4
Camera Link Controls	4	8	-	-	-
24V Opto-isolated I/O	4	4	4	4	4
TTL I/O	16	16	16	16	16
Serial Ports	1	2	2	2	2
Stereo Cameras	-	✓	✓	✓	✓
DMA to Host or Display	✓	✓	✓	✓	✓

THE THREE LEDS (1, 2, 3)



- **LED1** : Indicates that the board has powered up. When the board is inactive, it stays illuminated. For each PCI access to Phoenix, this LED will briefly extinguish.
- **LED2** : Flashes to indicate that captured image data from **channel 'A'** is being transferred across the PCI bus. Note that it can appear dimly lit when flashing at a sufficiently high rate.
- **LED3** : This LED performs the same function as LED2, but applies to **channel 'B'**.

THE POCL LEDS (4, 5)



The Phoenix Camera Link boards support the Power over Camera Link (PoCL) functionality and are able to provide power to PoCL enabled cameras via the Camera Link data cable thereby removing the need for a separate power supply. In addition to this the Phoenix implements *SafePower*, an intelligent sense mechanism which detects the presence of a PoCL camera before applying power to it. This safety mechanism ensures that power is not applied to conventional non-PoCL cameras. Phoenix can supply up to

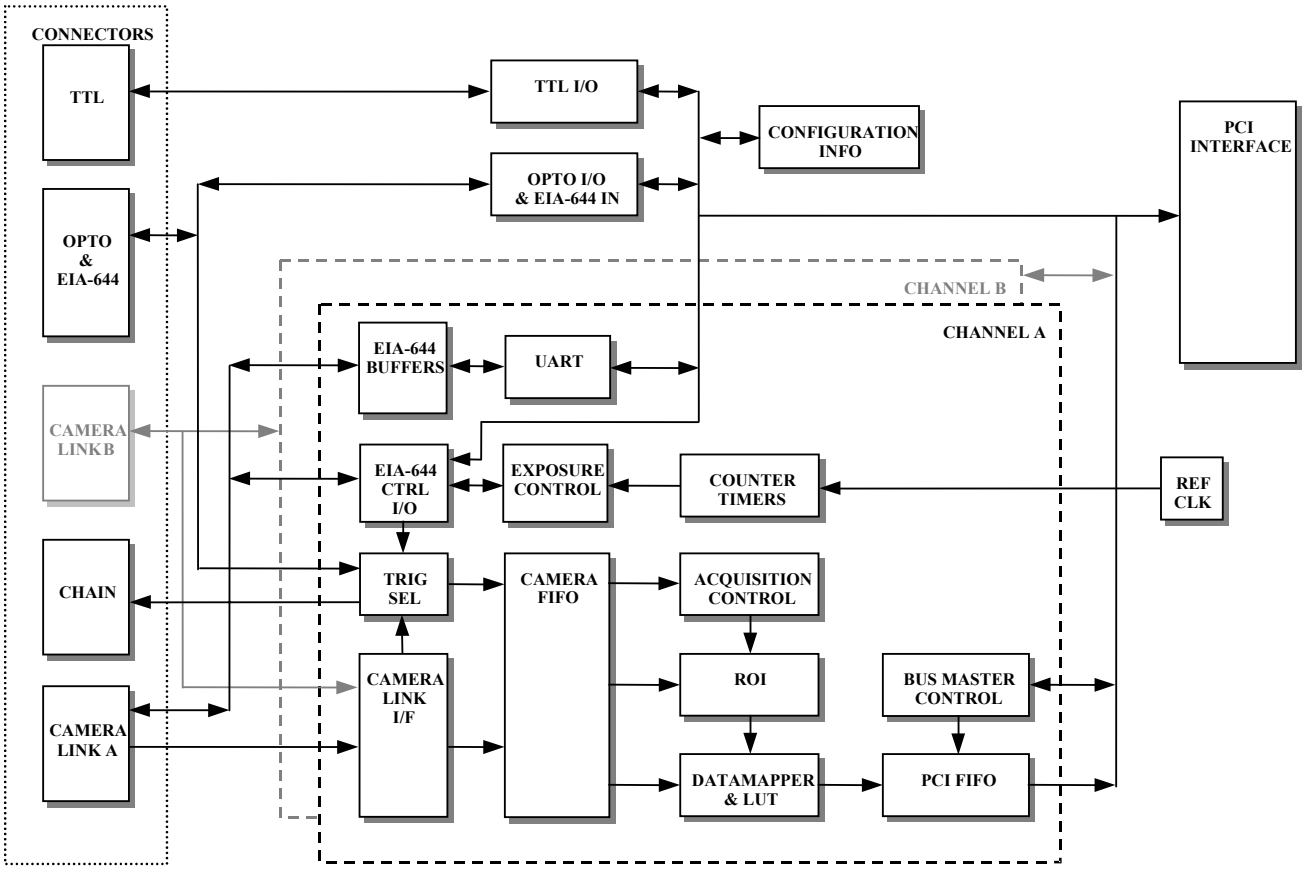
4W at a nominal 12V +/- 1V, as required by the PoCL specification. The PoCL specification also makes allowance for up to a 1V drop in the cable, thus providing 10V to 13V at the camera input. The LEDs operate as follows:

- **Flashing Amber** : Non PoCL camera or cable detected (i.e. pins 1/26 shorted to pins 13/14), but no clock output from the camera.
- **Solid Amber** : Non PoCL camera or cable detected, but with a valid clock output from the camera.
- **Flashing Green** : PoCL cable and camera have been detected (i.e. a 52uA sense current developed a 0.52V voltage across pins 1/26 and 13/14), and 12V power output enabled to the camera, but no valid camera clock yet received.
- **Solid Green** : PoCL cable and camera detected, 12V power output applied to the camera, and a valid clock output from the camera detected.
- **No LEDs** : Sensing mode, whereby the Phoenix board is scanning for any PoCL or non-PoCL camera to be connected.

Note that if the clock is lost whilst the board is in the solid Amber or Green states, Phoenix will revert back to sensing mode to allow users to change between PoCL and non-PoCL cameras.

BLOCK DIAGRAM

The following block diagram shows the Phoenix D48CL board as an example.



Note: This is a simplified block diagram that only shows the main data and control paths.

Camera Clock: ***Phoenix*** supports effective clock rates from DC to the Camera Link maximum of 85MHz, using the Camera Link Strobe (STB) and Data Valid (DVAL) signals.

Camera FIFO: Data from the video source is stored in a FIFO prior to being processed by ***Phoenix***.

Acquisition Control: The acquisition trigger control module is used to determine which video frames to acquire from the camera. The system can be configured for a single trigger event to acquire all subsequent frames, a trigger event per frame, or continuous acquisition irrespective of the trigger condition. The trigger event is programmable between either level or edge sensing on one of the opto-isolated or EIA-644 control inputs.

When running in **linescan** mode, there is an additional mode that uses the active trigger input as an envelope signal. In this mode all lines are acquired whilst the trigger input is asserted. The hardware can also delay the trigger event by a fixed time period or number of lines, and allows the trigger event transducer to be located remotely from the camera.

Region of Interest: The Region Of Interest (ROI) controls which part of the camera output data to acquire. In **areascan** mode, this is a rectangular region with software programmable width, height and x / y offset. **Linescan** mode is similar, allowing control of the width and x offset, with the height control being used to package the data into pseudo frames for subsequent processing by the user's application. ***Phoenix*** supports an additional mode (**DataStream**) whereby data is acquired based upon the control inputs, e.g. all data is acquired when Frame Valid (FVAL) and Line Valid (LVAL) are both asserted. This is necessary for cameras that output their own arbitrary ROIs within a single video frame, or those that vary the amount of data output on each line.

Sub-Sampling: Software controlled hardware sub-sampling is also supported. A factor of x1, x2, x4 or x8 can be independently selected for both x and y directions, e.g. a horizontal factor of x4 and a vertical factor of x2 would acquire every 4th pixel across a line and every 2nd line down the frame.

DataMapper: The raw camera data can be reformatted in hardware for ease of subsequent processing. For example, a mono data source can be converted into 32 bit colour data, ready to be sent directly to graphics card memory, thus reducing the host processor overhead. The optimum use of system resources is determined by the user's application, e.g. packing mono data into 32 bit colour reduces the host processor overhead at the expense of increasing the amount of data transferred across the PCI bus.

The output formats supported include, 8, 16 and 32 bit mono, as well as 15, 16, 24, 32 and 48 bit colour in both RGB and BGR ordering, thus supporting big and little endian processor formats. The data is also pre-packed into a 64 bit stream, prior to being sent across the PCI bus, for maximum transfer performance.

LUT: A 16 bit in, 16 bit out (i.e. 65,536 by 16) LUT per channel allows arbitrary mappings between the input data from the video source and the output data to the destination memory. This allows functions such as gamma correction, brightness, contrast and thresholding to be performed in real time in hardware on a per colour or per camera basis. The LUT may also be used to shift the LSB aligned video data to MSB alignment ready for processing.

PCI FIFO: ***PCI64*** version: A 1024 by 64 bit FIFO per channel provides buffering between the camera and the PCI bus. If the board is being used in single camera mode, then both channels are used providing a 2048 by 64 FIFO.

PCI64U version: Similarly, 512 by 64 bit per channel, or 1024 by 64 bit in single camera mode.

Note that this is not a frame store; ***Phoenix*** uses high speed Bus Mastering (DMA) to transfer the camera data into system memory, and therefore the image size is only limited by the amount of memory available on the host.

Bus Master Bus Master Control is provided by a dedicated RISC processor and a highly optimised

Control: PCI Bus Master (DMA) engine. The RISC processor reads transfer length and destination address instructions across the PCI bus from host memory, and loads these into the PCI Bus Master engine with no host CPU overhead.

The DMA engine then transfers the video data at the full PCI bus rate into host memory, thus achieving the maximum burst rate of 533MBytes/sec. When the current instruction has completed, the RISC processor optionally generates a PCI interrupt to signal that the transfer has completed, before either halting or retrieving the next instruction.

The RISC processor also supports jump instructions that allow a single piece of RISC code to continuously loop without any CPU intervention.

Interrupts: An interrupt signal is available, and can be configured via software to interrupt on a number of different events, including acquisition complete, FIFO overflow, Start/End of Frame/Line, etc.

Counter Timers: Four 32 bit counter timers are available for each channel of the ***Phoenix***. The counter timers are dedicated for the following functions:

1. Astable timer used as a line rate generator for linescan cameras, or as an acquisition trigger for areascan cameras, thus controlling the overall frame rate. The period of the astable can be set from 1µs up to 70 minutes in 1µs increments.
2. Dual monostables for generating two exposure output signals, e.g. ExSync and PRIN. Both monostables are triggered by the same software selectable event but can be programmed with different time periods, once again to 1µs resolution. This provides a flexible exposure control system.
3. Trigger delay counter used to postpone acquisition triggering by a programmable time delay or line count. This allows the acquisition trigger sensor to be mounted remotely from the camera. (Note: As the counter is non-retriggerable, subsequent trigger events will be ignored until a pending event has completed its delay).
4. A versatile event counter is provided to count a number of different events types - Lines (LVAL), Frames (FVAL) or microseconds, within a specified gate condition - Line (LVAL), Frame (FVAL), Acquisition Trigger or Entire Acquisition. The event count provides readings for both the current value, as well as the final value at the end of the previous gate condition. For example the event counter can be configured to provide the current line number within a frame, as well as the total number of lines in the previous frame. Other uses include providing the frame period, the number of lines in the previous acquisition trigger envelope – and hence how much data there is to process, or the number of images processed so far.

Camera Control Outputs: Two 4 bit EIA-644 (LVDS) output ports are provided to interface with the camera. Each bit can be individually set to a logical “1” or “0” under software control or used to drive the camera with exposure control pulses from the counter timer module.

The ports are on the Camera Link connectors.

Opto-Isolated I/O: 4 bits of opto-isolated I/O are provided to interface to external systems. As standard, ***Phoenix*** is configured with 2 bits of input and 2 bits of output, but this can be varied as factory build option. The outputs are designed to sink up to 20mA, and will withstand 24V when “off”. The inputs sense voltages between 3.3V and 24V as a logic high input. A 4.7kΩ current limiting series resistor is fitted on all inputs. The outputs can be individually set and cleared via software, controlled from the internal timer resources, or fed from other input events, e.g. acquisition triggers, etc.

EIA-644 Control In: Two 2 bit EIA-644 (LVDS) input ports are provided to interface with other systems. can be used as additional acquisition trigger sources, or as inputs from shaft encoders, etc.

TTL I/O: Two 8 bit TTL I/O ports are provided to interface with other systems. Each 8 bit port can be independently configured as all input or all output under software control. When used as outputs, each bit can source 24mA at min 2.2V or sink 24mA at max 0.55V. When used as inputs, an applied voltage of between 2V and 5V is read as a logical “1” and an applied voltage of between 0V and 0.8V as a logical “0”.

Serial Port:

Phoenix is fitted with a dual channel Universal Asynchronous Receiver Transmitter (UART), containing 64 character hardware transmit and receive FIFOs for each channel (the software libraries buffer both the transmit and receive data to provide larger user FIFOs). Each channel independently supports 1, 1.5 or 2 stop bits; 5, 6, 7 or 8 data bits; and odd, even or no parity. The baudrate can be configured with standard values from 300 baud up to 115,200 baud. ***Phoenix*** also supports software (XON, XOFF) flow control within the UART without host CPU intervention.

Connectors:

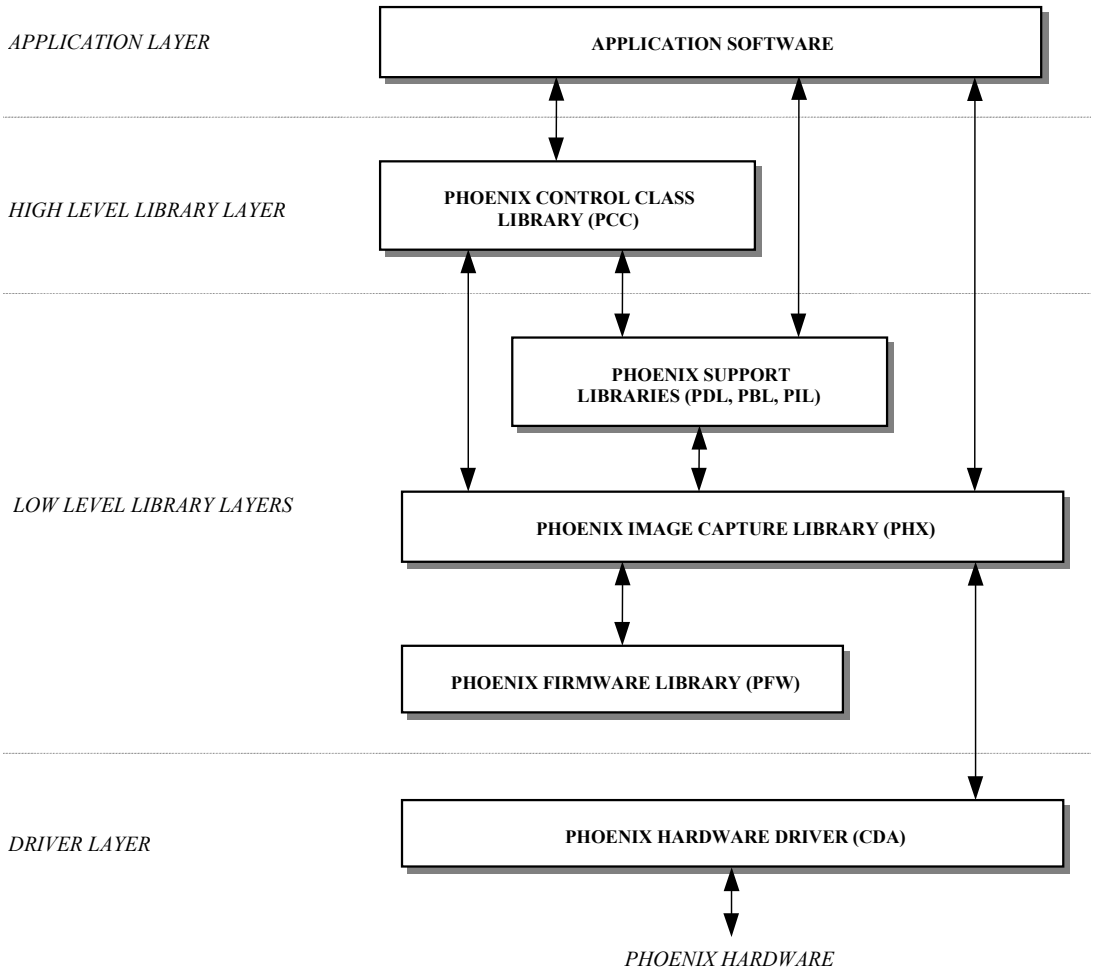
Phoenix is fitted with the 26 way 3M MDR connectors and screwlocks as specified in the Camera Link v1.1 specification.

For opto-isolated, EIA-644 & TTL I/O there are internal 20 & 26 way 0.1" IDC headers, with the option to bring these out to a 50 way mini D on an adjacent PCI slot.

A 10 way 0.1" IDC header ("Chain") allows two ***Phoenix*** boards to be used together to simultaneously acquire from wider sources.

SDK Overview

The Phoenix software architecture diagram (below) shows how the various library components interact.



- The Phoenix library (PHX) provides acquisition and control features for the Phoenix image capture boards.
- The Phoenix Firmware library (PFW) contains the firmware designs and is used by the PHX library to allow the Phoenix board's functionality to be upgraded with new SDK releases.
- The Phoenix Buffer library (PBL) provides method of allocating the destination memory buffers for images acquired from the Phoenix using the PHX library.
- The Phoenix Imaging library (PIL) provides various image format conversion routines as well as Bayer processing.
- The Phoenix Display Library (PDL) allows captured images to be displayed quickly and easily.
- The Phoenix Control Class library (PCC) provides a high level layer which allows applications requiring user control of the Phoenix hardware to be developed quickly and easily.

SDK Examples

Example application code is supplied as part of the Phoenix SDK. These examples illustrate how the Phoenix libraries can be used in various user applications. They are listed in ascending order of complexity, hence the user will benefit by working through them in sequence.

INITIALISING THE PHOENIX BOARD

phxinfo	This example shows how to initialise the Phoenix board and display the board property information. It shows how to programmatically determine board settings, as well as being a useful tool to check the hardware and device driver are correctly installed.
phxconfig	This example continually configures and releases the board resources. This is used under operating systems such as VxWorks, which may require custom platform specific initialisation code, to ensure all the low level read and write caches are configured correctly. If all the firmware data is not correctly sent to the board, it will fail the PHX_CameraConfigLoad call.

PCI BANDWIDTH MEASUREMENT

phxrate	This example calculates the maximum sustainable data rate across the PCI bus segment. The Phoenix hardware is configured to always transfer data across the PCI bus, irrespective of whether there is any data available from the video source. Additionally the hardware is configured to transfer data irrespective of whether the software has finished processing the previous buffer. This ensures that the Phoenix hardware transfers the maximum amount of data in the shortest possible time.

SINGLE ACQUISITION WITH FILE SAVE

phxsnap	This example shows how to acquire a single image from the Phoenix board and save to an output file in raw data format.

LIVE ACQUISITION

phxsimple	This example shows how to initialise the Phoenix board and use the Phoenix library (PHX) to run live acquisition, using a callback function.
phxlive	This example shows how to initialise the Phoenix board and use the display library (PDL) to run live double buffered (also known as ping-pong) acquisition, using a callback function.
phxstereo	This example shows how to use the Phoenix libraries to configure two independent cameras into a single board. For simplicity the code here assumes two identical cameras, though this is NOT a restriction of the libraries. The code does not assume or require any synchronisation between the two channels. Once again the images are displayed using the display library (PDL), as an example of how and where to process the image data. This example uses an application specific structure to contain information about each channel, and hence allow common code to be used for initialisation, data processing, and destruction. This dual channel feature can be regarded as having two virtual frame grabbers on a single piece of hardware.
phxcheckandwait	This example shows how to initialise the Phoenix board and run continuous acquisition using polling. If the camera is turned off or disconnected, a timeout is detected and displayed.
phxtimeout	This example shows how to initialise the Phoenix board and run continuous acquisition using a callback function. If the camera is turned off or disconnected, a timeout is detected and displayed.

BUFFER CONTROL

phxvptrs	This example shows how to initialise the Phoenix board using virtual memory managed by the application. In this simple example, the value of the virtual address buffer is printed to provide user feedback.
phxacqbufferstart	This example shows how to use the PHX_ACQ_BUFFER_START parameter. It performs two single shot acquisitions, starting from a different buffer each time. In this simple example, the value of the virtual address buffer is printed to provide user feedback.

IMAGE CONVERSION

phxconvert	This example shows how to use the image conversion function. It captures into a direct buffer, and then converts the image data into a format suitable for display.
phxbayer	This example shows how to initialise the Phoenix board and use the buffer library (PBL) to run live double buffered (also known as ping-pong) acquisition, using a callback function. Each captured buffer is then converted by the image library (PIL) from Bayer format to the display format and displayed.
phxbayer4	This example shows how to initialise the Phoenix board and use the buffer library (PBL) to run live double buffered (also known as ping-pong) acquisition, using a callback function. Each captured buffer is then converted by the image library (PIL) from Bayer format to Red, Green and Blue buffer format. All four buffer formats are displayed.

SERIAL I/O

phxser	This example is a simple terminal program for serial I/O. It allows character strings to be sent to the camera with either CR and or LF appended to the transmit string, by the use of "\r" and "\n" keys. The application then waits for 1 second before displaying all received characters. To quit the application, type "Exit" at the prompt.

LUT CONTROL

phxlut	This example shows how to use a software call to change the Look-Up Table settings.

TRIGGER CONTROL

phxswtrigger	This example shows how to use a software call to trigger the camera.

MISCELLANEOUS

phxmfc	This is an example Visual C++ project which builds an MFC application using the supplied Phoenix libraries with PCC as the user interface to the Phoenix board.

Phoenix Board Control

CONFIGURING PHOENIX

Before using a particular channel on a Phoenix board, it must be configured with the required settings relevant to the camera, system and application. The function ***PHX_CameraConfigLoad*** is used to configure the Phoenix board and channel. It has the following prototype

```
etStat PHX_CameraConfigLoad( tHandle *phCamera,
                             char* szConfigFile,
                             etCamConfigLoad eCamConfigLoad,
                             void (*pFnErrorHandler)
                             (const char*, etStat, const char*) );
```

These four parameters are explained below

Camera handle

phCamera is a pointer to a camera handle return value. The camera handle is assigned when a particular channel of a board is successfully configured. The camera handle ***hCamera*** is used as the first parameter in most of the Phoenix library (PHX) function calls. It indicates to the Phoenix library which channel of which board is to be accessed.

Phoenix Configuration File

szConfigFile is the name of a Phoenix Configuration File (.PCF). It is a text file which can conveniently store some or all of the parameters which can be set using the ***PHX_ParameterSet*** function. If ***szConfigFile*** is NULL, then the Phoenix channel will be configured with default values for all of its parameters. The Phoenix SDK contains example configuration files for many cameras.

When a Phoenix channel has been successfully configured, and a few of its parameters changed (e.g. by using the ***PHX_ParameterSet*** function), then the function ***PHX_CameraConfigSave*** may be used to save the current settings to a new configuration file.

Configuration Options

eCamConfigLoad is constructed from a bitwise 'OR' of the following options

- **Physical Board Type**. This **must** be set to either ***PHX_DIGITAL*** (in which case any digital Phoenix board will be configured) or one of the specific Phoenix board types, such as ***PHX_D48CL_PCI64*** (in which case only an AS-PHX-D48CL-PCI64-B variant board, or later variant of the same type, will be configured). Note that a digital Phoenix board without the -B variant, or later, marking can only be opened as ***PHX_DIGITAL***. It is considered to be a generic Phoenix board, rather than a specific Phoenix board (e.g. an AS-PHX-D36-PCI32 is a generic Phoenix board, whereas an AS-PHX-D36-PCI32-B is a specific Phoenix board). However, note that the PCI Express boards (e.g. AS-PHX-D24CL-PE1) are specific boards.
- **Board Number**. This is usually ***PHX_BOARD1***. However, if there are a number of Phoenix boards present, it is possible to configure a specific Phoenix board of the above type by specifying ***PHX_BOARD2*** for example.
- **Channel Number**. If this is not specified or is set to ***PHX_CHANNEL_AUTO***, then the first un-configured Phoenix channel on the above board number will be configured. It is possible to configure a specific channel on the above board number by specifying ***PHX_CHANNEL_B*** for example.
- **Board Mode**. If this is not specified or is set to ***PHX_MODE_NORMAL***, then both the Phoenix acquisition engine and the Phoenix communication port are opened for use on the current channel. If this is set to ***PHX_ACQ_ONLY***, then only the Phoenix acquisition engine is opened. This allows the Phoenix communication port to be opened by another application (e.g. a camera link serial .dll). If this is set to ***PHX_COMMS_ONLY***, then only the Phoenix communication port is opened for use

on the current channel. This allows another application to access the Phoenix acquisition engine of the current channel. Note that **Board Mode** is only available when used with a **Physical Board Type** which corresponds to an AS-PHX-*-B board variant or later (e.g. AS-PHX-D48CL-PCI64-B), or a PCI Express board (AS-PHX-*-PE1).

Error Handler

pFnErrorHandler is a pointer to an error handler function. It is called whenever a Phoenix library function encounters an error condition. It can be set to

- ***PHX_ErrHandlerDefault***, in which case the default internal error handler will be invoked.
- ***NULL***, in which case no error handler will be invoked.
- A pointer to a user defined custom error handler function.

Examples

To find and initialise the first un-configured channel on the first Phoenix board in the system using the camera configuration file provided, using the default supplied error handler:

```
PHX_CameraConfigLoad( &hCamera, "file.pcf", PHX_DIGITAL |,
                      PHX_BOARD1, &PHX_ErrHandlerDefault );
```

To find and initialise the first un-configured channel on the first Phoenix board in the system using the camera configuration file provided, using a custom error handler function:

```
void ErrorHandler( const char* pszFnName, etStat eErrCode,
                   const char* pszDescString )
{
   printf( "%s:%x:%s\n", pszFnName, eErrCode, pszDescString );
}

PHX_CameraConfigLoad( &hCamera, "file.pcf", PHX_DIGITAL |,
                      PHX_BOARD1, &ErrorHandler );
```

To find and initialise channel A on the second Phoenix board in the system using the camera configuration file provided, without any error handler installed:

```
PHX_CameraConfigLoad( &hCamera, "file.pcf", PHX_DIGITAL |
                      PHX_BOARD2 | PHX_CHANNEL_A, NULL );
```

To find and initialise channel A on the first AS-PHX-D24CL-PCI32-B board found using default settings, without any error handler installed:

```
PHX_CameraConfigLoad( &hCamera, NULL, PHX_D24CL_PCI32 |
                      PHX_BOARD1 | PHX_CHANNEL_A, NULL );
```

To find and initialise channel B on the first AS-PHX-D24CL-PCI32-B board found using default settings, without any error handler installed:

```
PHX_CameraConfigLoad( &hCamera, NULL, PHX_D24CL_PCI32 |
                      PHX_BOARD1 | PHX_CHANNEL_B, NULL );
```

To find and initialise channel A on the second AS-PHX-D24CL-PCI32-B board found using default settings, without any error handler installed, leaving the Phoenix communication port available for use by another application:

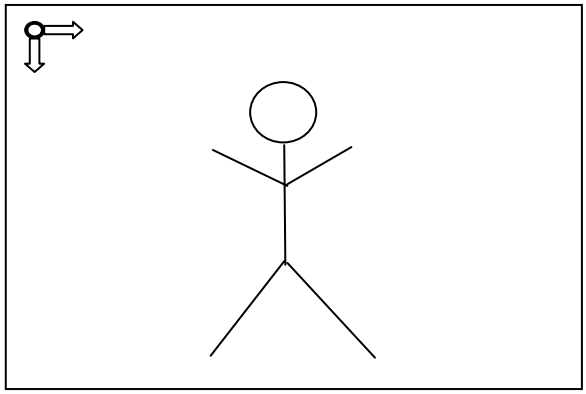
```
PHX_CameraConfigLoad( &hCamera, NULL, PHX_D24CL_PCI32 |
                      PHX_BOARD2 | PHX_CHANNEL_A | PHX_ACQ_ONLY, NULL );
```

CAMERA TAP CONFIGURATION

A *tap* is an image data channel provided by the camera. A camera may have just one tap, or it may have multiple taps i.e. it can generate image data from multiple areas of the image in parallel. Each vertical tap provides image data from a particular line. Each of these vertical taps may in turn have multiple horizontal taps which provide image data from a particular point in that line.

The following examples illustrate possible camera tap configurations and explain how the associated parameters relate to these.

Single tap

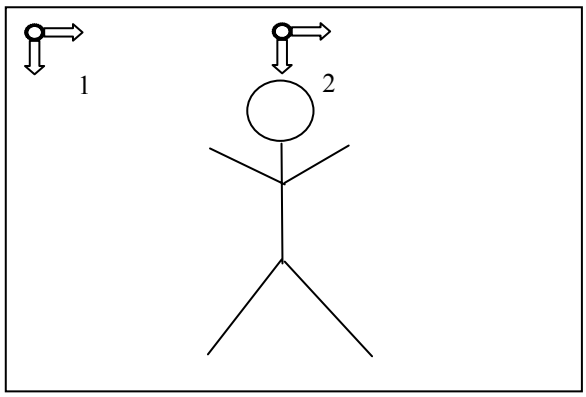


The box above represents the image area from the camera with pixel (0,0) at top left. The small circle indicates a tap and the arrows show its direction. In this case there is a single vertical tap moving from the top of the image to the bottom, left to right across each line. This vertical tap has a single horizontal tap extracting pixels from left to right across the image. So, with reference to the parameters below, the settings are :-

```
PHX_CAM_HTAP_NUM   = 1
PHX_CAM_HTAP_DIR   = PHX_CAM_HTAP_LEFT
PHX_CAM_HTAP_TYPE  = PHX_CAM_HTAP_LINEAR
PHX_CAM_HTAP_ORDER = N/A

PHX_CAM_VTAP_NUM   = 1
PHX_CAM_VTAP_DIR   = PHX_CAM_VTAP_TOP
PHX_CAM_VTAP_TYPE  = PHX_CAM_VTAP_LINEAR
PHX_CAM_VTAP_ORDER = N/A
```

Dual horizontal taps



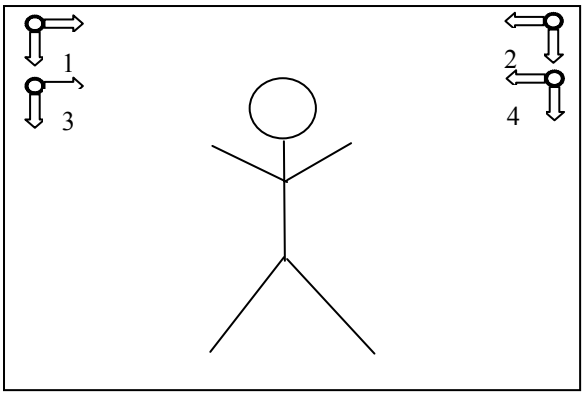
Here, again there is a single top to bottom vertical tap, but consisting of 2 offset horizontal taps extracting image data from two points in the line simultaneously. Each tap generates one half of the line data. It is

necessary to specify which half of the line is generated by each of the taps since cameras vary in their definition of tap ordering. In this case we specify the ordering as ascending which means that tap 1 generates the left hand side of the line (i.e. it is earlier in time), and tap 2 the right hand. Tap 1 therefore provides pixels 0, 1, 2… and tap 2 pixels n, (n+1), n+2)…. The parameters for this configuration are :-

```
PHX_CAM_HTAP_NUM   = 2
PHX_CAM_HTAP_DIR   = PHX_CAM_HTAP_LEFT
PHX_CAM_HTAP_TYPE  = PHX_CAM_HTAP_OFFSET
PHX_CAM_HTAP_ORDER = PHX_CAM_HTAP_ASCENDING

PHX_CAM_VTAP_NUM   = 1
PHX_CAM_VTAP_DIR   = PHX_CAM_VTAP_TOP
PHX_CAM_VTAP_TYPE  = PHX_CAM_VTAP_LINEAR
PHX_CAM_VTAP_ORDER = N/A
```

Dual horizontal and dual vertical taps



Now there are 2 top to bottom vertical taps, each providing alternate lines of the image. The vertical taps consist of 2 horizontal taps which are moving towards each other. Again, the physical ordering of the taps must be specified, both horizontally and vertically. In this case the camera provides the left hand of the image on taps 1 and 3, and the right side on 2 and 4, so it is horizontally ascending. But it generates even lines on taps 3 and 4, and odd lines on taps 1 and 2; therefore it is vertically descending. The parameters for this configuration are :-

```
PHX_CAM_HTAP_NUM   = 2
PHX_CAM_HTAP_DIR   = PHX_CAM_HTAP_CONVERGE
PHX_CAM_HTAP_TYPE  = PHX_CAM_HTAP_OFFSET
PHX_CAM_HTAP_ORDER = PHX_CAM_HTAP_ASCENDING

PHX_CAM_VTAP_NUM   = 2
PHX_CAM_VTAP_DIR   = PHX_CAM_VTAP_TOP
PHX_CAM_VTAP_TYPE  = PHX_CAM_VTAP_LINEAR
PHX_CAM_VTAP_ORDER = PHX_CAM_VTAP_DESCENDING
```

SDK Concepts

USING THE PHX_CACHE_FLUSH PARAMETER

The Phoenix library uses a cache to optimise access to the hardware. Commands are sent to the board using the ***PHX_ParameterSet*** call. These updates are stored and any necessary hardware writes are only performed with a call to ***PHX_ParameterSet*** with the ***PHX_CACHE_FLUSH*** parameter combined with a bitwise 'OR' of the parameter being set.

For example:

```
etParamValue eParamValue;
ui32 dwValue;

eParamValue = PHX_DST_FORMAT_Y8;
PHX_ParameterSet( hCam, PHX_DST_FORMAT, &eParamValue );

dwValue = 2048;
PHX_ParameterSet( hCam, (etParam)( PHX_BUF_DST_XLENGTH | PHX_CACHE_FLUSH ),
                  &dwValue );
```

The board is not updated until the final call to ***PHX_ParameterSet*** with the ***PHX_CACHE_FLUSH***. Alternatively you can force a cache flush with the following call.

```
PHX_ParameterSet( hCam, (etParam)( PHX_DUMMY_PARAM | PHX_CACHE_FLUSH ), NULL );
```

It is not advised to use the ***PHX_CACHE_FLUSH*** on every call to ***PHX_ParameterSet*** as each time it is used the board will recalculate its internal settings. It is best to set-up the board with the settings you require and then flush them all at the appropriate point.

Note: If either ***PHX_ParameterGet*** or ***PHX_Acquire(, PHX_START,);*** is called then the cache is automatically flushed.

PHOENIX ERROR HANDLING

Error Status Codes

All of the Phoenix SDK functions return an enumerated value of type ***etStat*** which is used to return the status (success or otherwise) of the function being called. These values are defined in the ***phx_api.h*** header file, and their meanings are outlined below.

PHX_OK	Successful completion, without errors.
PHX_ERROR_BAD_PARAM	The eParam passed to the function is either unrecognised, or not supported for that function.
PHX_ERROR_BAD_PARAM_VALUE	The eParamValue passed to ***PHX_ParameterGet*** or ***PHX_ParameterSet*** functions is either unrecognised, or not supported for the eParam value.
PHX_ERROR_READ_ONLY_PARAM	The eParam passed to the ***PHX_ParameterSet*** function is read only.
PHX_ERROR_BAD_HANDLE	The handle value is invalid.
PHX_ERROR_OPEN_FAILED	Failed to find the hardware.
PHX_ERROR_INCOMPATIBLE	An illegal combination of parameters was encountered.
PHX_ERROR_HANDSHAKE	Serial communications handshake failure.

PHX_ERROR_INTERNAL_ERROR	An internal error has occurred in the library. Please contact Active Silicon for further support.
PHX_ERROR_OVERFLOW	A data overflow has occurred. This could be a PCI FIFO overflow, or a comms Receive/Transmit buffer overflow for example.
PHX_ERROR_NOT_IMPLEMENTED	The requested feature is not available in the current release of software.
PHX_ERROR_HW_PROBLEM	The hardware is not responding.
PHX_ERROR_NOT_SUPPORTED	The requested option is not supported. This can be caused by invalid options for the hardware, i.e. analogue sync settings for a PHX_DIGITAL product, or for features which are not implemented on the hardware variant, i.e. access to a second serial port.
PHX_ERROR_OUT_OF_RANGE	A parameter is not within the permitted range.
PHX_ERROR_MALLOC_FAILED	Failed to allocate system memory.
PHX_ERROR_SYSTEM_CALL_FAILED	A call to the underlying OS failed.
PHX_ERROR_FILE_OPEN_FAILED	Could not find or read file.
PHX_ERROR_FILE_CLOSE_FAILED	Could not close file after reading or writing.
PHX_ERROR_FILE_INVALID	The file contained errors.
PHX_ERROR_BAD_MEMBER	The parameter stored within the libraries is invalid. This is a specific case of PHX_ERROR_INTERNAL_ERROR. Please contact Active Silicon for further support.
PHX_ERROR_HW_NOT_CONFIGURED	The RISC code has not been generated.
PHX_ERROR_INVALID_FLASH_PROPERTIES	The onboard flash memory contains an invalid property string.
PHX_ERROR_ACQUISITION_STARTED	An attempt was made to start an acquisition when there was already one in progress.
PHX_ERROR_INVALID_POINTER	A pointer was passed to the library which does not reference valid memory.
PHX_ERROR_LIB_INCOMPATIBLE	An attempt was made to use an out of date software library with the PHX library.
PHX_ERROR_DISPLAY_CREATE_FAILED	PDL failed to create a display instance.
PHX_ERROR_DISPLAY_DESTROY_FAILED	PDL failed to destroy a display instance.
PHX_ERROR_DDRAW_INIT_FAILED	PDL failed to initialise a display instance.
PHX_ERROR_DISPLAY_BUFF_CREATE_FAILED	PDL failed to create a display buffer instance.
PHX_ERROR_DISPLAY_BUFF_DESTROY_FAILED	PDL failed to destroy a display buffer instance.
PHX_ERROR_DDRAW_OPERATION_FAILED	A PDL DirectDraw operation failed (e.g. blit)
PHX_WARNING_TIMEOUT	The requested operation has timed out.
PHX_WARNING_FLASH_RECONFIG	The onboard flash memory has been

	reprogrammed successfully with new firmware.
PHX_WARNING_ZBT_RECONFIG	The board firmware has been temporarily updated with new firmware.
PHX_WARNING_NOT_PHX_COM	No Phoenix OS specific Communications Ports have been found, e.g. win32 PHX serial ports.
PHX_WARNING_NO_PHX_BOARD_REGISTERED	No Phoenix boards have been registered with an OS specific structure, e.g. a win32 Registry entry was not found.

Error Information

The libraries also keep a copy of the first error and most recent error returned in the following read only parameters.

- PHX_ERROR_FIRST_ERRNUM
- PHX_ERROR_LAST_ERRNUM
- PHX_ERROR_FIRST_ERRSTRING
- PHX_ERROR_LAST_ERRSTRING

The PHX_ERROR_FIRST_ERRNUM and PHX_ERROR_LAST_ERRNUM parameters return the ***etStat*** values describing the error. The PHX_ERROR_FIRST_ERRSTRING and PHX_ERROR_LAST_ERRSTRING parameters return a string describing the error.

The PHX_ERROR_LAST_ERRNUM and PHX_ERROR_LAST_ERRSTRING parameters return the most recent error encountered. The PHX_ERROR_FIRST_ERRNUM and PHX_ERROR_LAST_ERRNUM parameters return the first error encountered since that parameter was queried.

After PHX_ERROR_FIRST_ERRNUM or PHX_ERROR_FIRST_ERRSTRING has been queried, the next error to occur will then become the new first error.

Error Handler Function

When configuring a Phoenix board, the fourth parameter of ***PHX_CameraConfigLoad*** is a pointer to an error handler function. This gets called when an error is encountered within the PHX library. The function has the following prototype

```
void PHX_ErrHandlerDefault( const char *pszFnName, etStat eErrCode,
                            const char *pszDescString );
```

pszFnName – Name of the function in which the error occurred.
eErrCode – The error generated.
pszDescString – A descriptive string describing the error in more detail.

The user may provide his/her own custom error handler, or specify ***PHX_ErrHandlerDefault*** which is the built in PHX error handler. This function generates a popup message box (under Win32) describing the error. Under other operating systems such as linux for example, it writes a printf statement to the console window.

Decoding Error Codes

The function ***PHX_ErrCodeDecode*** is provided to decode an ***etStat*** error code into a human readable string. The function is defined as follows

```
void PHX_ErrCodeDecode( char *pszDescString, etStat eErrCode );
```

pszDescString – Target descriptive string (user allocated)
eErrCode – Error code to decode.

I/O PORT ACCESS

Phoenix has a number of I/O ports which can be used for controlling external equipment, sensing etc. The following ports are available:

LVDS Phoenix board	Camera Link Phoenix board
Two 8 bit bidirectional TTL ports (A and B).	Two 8 bit bidirectional TTL ports (A and B).
Two 2 bit bidirectional opto isolated ports (A and B).	Two 2 bit bidirectional opto isolated ports (A and B).
Two 2 bit bidirectional Camera Control I/O ports (A and B).	Two 4 bit Camera Control output ports (A and B).

Absolute / Relative I/O Port Access

There are two ways of accessing each I/O port:

Absolute mode	Allows either port to be accessed irrespective of which channel is currently open. E.g. ***PHX_IO_TTL_A*** or ***PHX_IO_TTL_B***.
Relative mode	Allows only the port associated with the currently open channel to be accessed. E.g. ***PHX_IO_TTL***.

Hence if Channel A is currently open for use, then TTL port A can be accessed via either ***PHX_IO_TTL_A*** (Absolute mode) or ***PHX_IO_TTL*** (Relative mode). Similarly, if Channel B is currently open for use, then TTL port B can be accessed via either ***PHX_IO_TTL_B*** (Absolute mode) or ***PHX_IO_TTL*** (Relative mode). Using I/O ports in Relative mode allows common application code to be written, such that when applied to different channels, the corresponding I/O ports associated with that channel are correctly accessed.

Note : The **D24CL** Phoenix board only has one camera channel (i.e. channel 'A'). Hence, the channel 'B' I/O ports on this board must be accessed in Absolute mode.

etPhxIoMethod for I/O Port Access

When setting or clearing bits in the I/O ports, various etPhxIoMethods have been defined to allow greater flexibility when using the I/O control parameters:

PHX_IO_METHOD_WRITE	This method causes all of the I/O bits to be overwritten with the bits defined in the lower bits of the parameter value.
PHX_IO_METHOD_BIT_SET	This method sets only the I/O bits defined in the lower bits of the parameter value.
PHX_IO_METHOD_BIT_CLR	This method clears only the I/O bits defined in the lower bits of the parameter value.

The BIT_SET and BIT_CLR methods are useful for application code which is only concerned with specific bits in an I/O port, and has no knowledge of the other bits in that port.

The CCIO ports on the LVDS board (CCOUT for Camera Link boards) and Opto port bits can be set up to output a programmable width pulse using the internal astable timer (useful for exposure control for example). The following two etPhxIoMethods control this mode of operation:

PHX_IO_METHOD_BIT_TIMER_POS	The I/O bits defined in the lower bits of the parameter value are used as a mask to set the relevant bits to be timer controlled. The timer is triggered from either the line trigger or acquisition trigger and the outputs are active high.
PHX_IO_METHOD_BIT_TIMER_NEG	The I/O bits defined in the lower bits of the parameter value are used as a mask to set the relevant bits to be timer controlled. The timer is triggered from either the line trigger or acquisition trigger and the outputs are active low.

The CCIO ports on the LVDS board (CCOUT for Camera Link boards) can also be driven directly by the acquisition trigger signal. The following two etPhxIoMethods control this mode of operation:

PHX_IO_METHOD_BIT_ACQTRIG_POS	The I/O bits defined in the lower bits of the parameter value are used as a mask to set the relevant bits to be timer controlled. The outputs are derived from the acquisition trigger and are active high.
PHX_IO_METHOD_BIT_ACQTRIG_NEG	The I/O bits defined in the lower bits of the parameter value are used as a mask to set the relevant bits to be timer controlled. The outputs are derived from the acquisition trigger and are active low.

Examples

The following example shows how to set all of the bits high on TTL port A. Note the use of ***PHX_CACHE_FLUSH*** to ensure that the bits are written to the port when that line of code is executed. If the port is to be toggled, then ***PHX_CACHE_FLUSH*** must be used with each call to ***PHX_ParameterSet***.

```
/* Ensure the port direction is set to OUTPUT */
eParamValue = (etParamValue) PHX_ENABLE;
PHX_ParameterSet( hCam, PHX_IO_TTL_A_OUT, &eParamValue );

/* Set all of the bits */
dwValue = 0xff;
PHX_ParameterSet( hCam, (etParam)( PHX_IO_TTL_A | PHX_CACHE_FLUSH ), &dwValue );
```

The following example shows how to clear both bits of the current CCIO port.

```
/* Ensure the port direction is set to OUTPUT */
eParamValue = (etParamValue) PHX_ENABLE;
PHX_ParameterSet( hCam, PHX_IO_CCIO_OUT, &eParamValue );

/* Clear both bits */
eParamValue = (etParamValue) PHX_IO_METHOD_BIT_CLR | 0x03;
PHX_ParameterSet( hCam, (etParam)( PHX_IO_CCIO | PHX_CACHE_FLUSH ),
                  &eParamValue );
```

HOW TO USE THE DISPLAY LIBRARY

The **Display Library API Reference Manual** describes how to use the display functions. This section describes parts of the display functionality that may not be obvious but are important when evaluating the performance of the system.

Speed and data rate issues

Displaying captured image data can dramatically affect the load of a system if not performed correctly. When a display buffer is allocated using the ***PDL_BufferCreate*** function, the buffer format will match the current display mode.

What this means is that if the display depth is set to 32-bit mode then the buffer created by the library, either directly on the display or in system memory, will be 32-bits. This makes the copying of the image to the display more efficient.

It is likely that the captured image is not 32-bit, it will more than likely be 8, 10, 12, 16-bits etc. The Phoenix board will automatically convert the captured image data to the destination format and transfer this over the PCI bus.

This means that if the display mode is 32-bit and the camera source is 8-bit then the Phoenix board will actually transfer 32-bits/pixel.

Therefore in some cases, when displaying the captured image, the transferred data rate may exceed the PCI bandwidth and image corruption can occur (see **IMAGE CORRUPTION (FIFO OVERFLOW)** for more information). To avoid this problem, reduce the display pixel depth to a mode whereby the overall data rate is reduced.

When capturing the image to host for processing, the destination buffer can be set to match the captured image format. This can be done using the ***PHX_ParameterSet*** function with the ***PHX_DST_FORMAT*** set to the relevant bit depth.

Double Buffering

Phoenix can make use of double buffering to help improve the performance of the system.

On creation a display is associated with a board, see below.

```
PDL_DisplayCreate( &DisplayHandle, HwndHandle, PhxBoardHandle, ErrHandler );
```

Once this is complete the display buffers can be created, these are associated with the newly created display, see below.

```
PDL_BufferCreate( &DisplayBuffer1, DisplayHandle, PDL_BUFF_VIDCARD_MEM_DIRECT );

PDL_BufferCreate( &DisplayBuffer2, DisplayHandle, PDL_BUFF_VIDCARD_MEM_DIRECT );
```

In this instance the buffers are created on the graphics card by using the parameter, ***PDL_BUFF_VIDCARD_MEM_DIRECT***. The next section explains this option in more detail.

After successfully creating the two display buffers they need to be initialised. This sets the depth of each buffer according to the current display depth. The X and Y size is equal to the image size set in the configuration file used to configure the board.

```
PDL_DisplayInit( DisplayHandle );
```

This will inform the libraries to use the two display buffers for the captured image.

When ***PHX_Acquire*** is called, and the capture has completed, a Callback message will be generated. It is from within the Callback function that the current image ID can be retrieved and used to transfer the newly captured image to the display. The ***PDL_BufferPaint*** function is called to transfer the new image to the onscreen graphics memory. The following command shows how this is done.

```
PDL_BufferPaint( (tHandle)NewBuffer.pvContext );
```

The NewBuffer variable is a structure that contains the current image details; the pvContext member identifies the buffer to use when updating the display. The other member of the structure is the physical address; this is not required for the display update.

While this is happening the Phoenix board is acquiring data in to the second buffer.

One thing to note is that before we exit the Callback, but after we have painted the buffer, we have to release the buffer. This tells the board that it can use this area of memory for the next capture.

```
PHX_Acquire( PhxBoardHandle, PHX_BUFFER_RELEASE, NULL );
```

Display buffer destinations.

The display buffer can be allocated in one of 4 different modes, these are:

- PDL_BUFF_SYSTEM_MEM_DIRECT
- PDL_BUFF_SYSTEM_MEM_INDIRECT

- PDL_BUFF_VIDCARD_MEM_DIRECT
- PDL_BUFF_VIDCARD_MEM_INDIRECT

A '_DIRECT' buffer is one that will receive DMA data from the Phoenix board. These buffers are used to directly display the captured image.

An '_INDIRECT' buffer is a displayable buffer that will be used to accept processed data. The usual scenario is that you have captured your image to memory (in a '_DIRECT' buffer), it is then processed and the resultant data is copied to the '_INDIRECT' buffer. The address of the displayable buffer can be obtained with a call to ***PDL_BufferParameterGet***.

After the displayable buffer has been written to then it can be displayed with a call to ***PDL_BufferPaint***.

PHX_EVENTCOUNT / PHX_EVENTCOUNT_AT_GATE

The ***PHX_EVENTCOUNT*** parameter returns a read only value which is always the **current** value of the event counter. The event counter counts either elapsed image lines, elapsed image frames, or elapsed time in microseconds, as controlled by the ***PHX_EVENTCOUNT_SRC*** parameter.

The ***PHX_EVENTCOUNT_AT_GATE*** parameter returns a read only value which is the **gated** value (or **buffered** value) of the event counter, **sampled** when the gating signal is active. The gating signal is controlled by the ***PHX_EVENTGATE_SRC*** parameter. The gating signal can either be the acquisition trigger, the frame enable signal, the line enable signal, or when an image acquisition is in progress.

The event counter is reset on the active edge of the gating signal. The gated count is the same as the event count while the gating signal is active. When the gating signal is removed, the gated count is latched. The gated value of the event counter remains valid until the gating signal next occurs, and the event counter value is reset, and counting resumes from zero.

HOW TO CREATE A SMALLER EXECUTABLE BY LINKING FIRMWARE OBJECT FILES

Each Phoenix Frame Grabber Board contains a reconfigurable FPGA device. When a board is powered up, the FPGA device is loaded with a factory default firmware design. When the board is configured using the Phoenix Libraries, the firmware is automatically updated. This upgrades the board's functionality to match the library release.

Each Phoenix board needs a particular FPGA design depending on the PCI bus width, speed and voltage. Each firmware design is typically between 160KB and 230KB. All of the designs for the complete range of Phoenix boards are contained in the Phoenix Firmware Library. This is currently over 4MB in size. An application that links to the static version of the Phoenix Firmware Library, as used under DOS32 with DJGPP, Linux and VxWorks operating systems, is therefore typically 4MB larger than it could be.

The size of the executable can be reduced using the following steps:

1. Extract the required firmware object file(s) for the required Phoenix board / PCI bus combination(s).
2. Link the application to these object files, instead of the Phoenix Firmware Library.

Phoenix Firmware Object Files

The following table shows the object files contained in the Phoenix Firmware Library archive.

Object File	Board	PCI Bus			Library Symbol
		Width (bits)	Speed (MHz)	Volts	
pfw_api.o	All boards				
pfw_g.o	All boards				

d2p3m3v3.obj	D24CL	32	33	3.3	abAsPhxData_D24Cl_P32_M33_V3
d2p3m3v5.obj	D24CL	32	33	5	abAsPhxData_D24Cl_P32_M33_V5
d2pe1.obj	D24CL-PE1	PCI Express			abAsPhxData_D24Cl_PE1
d3p3m3v3.obj	D36-PCI32	32	33	3.3	abAsPhxData_D36_P32_M33_V3
d3p3m3v5.obj	D36-PCI32	32	33	5	abAsPhxData_D36_P32_M33_V5
d3p6m3v3.obj	D36-PCI64	64	33	3.3	abAsPhxData_D36_P64_M33_V3
d3p6m6v3.obj	D36-PCI64	64	66	3.3	abAsPhxData_D36_P64_M66_V3
d3p3m3u3.obj	D36-PCI64U	32	33	3.3	abAsPhxData_D36_P32_M33_V3_U
d3p3m3u5.obj	D36-PCI64U	32	33	5	abAsPhxData_D36_P32_M33_V5_U
d3p6m3u3.obj	D36-PCI64U	64	33	3.3	abAsPhxData_D36_P64_M33_V3_U
d3p6m3u5.obj	D36-PCI64U	64	33	5	abAsPhxData_D36_P64_M33_V5_U
d3p6m6u3.obj	D36-PCI64U	64	66	3.3	abAsPhxData_D36_P64_M66_V3_U
d3pe1.obj	D36-PE1	PCI Express			abAsPhxData_D36_PE1
d4p3m3v3.obj	D48CL-PCI64	32	33	3.3	abAsPhxData_D48Cl_P32_M33_V3
d4p6m3v3.obj	D48CL-PCI64	64	33	3.3	abAsPhxData_D48Cl_P64_M33_V3
d4p6m6v3.obj	D48CL-PCI64	64	66	3.3	abAsPhxData_D48Cl_P64_M66_V3
d8p6m6v3.obj	D48CL-PCI64	64	66	3.3	abAsPhxData_D48Cl85_P64_M66_V3
d4p3m3u3.obj	D48CL-PCI64U	32	33	3.3	abAsPhxData_D48Cl_P32_M33_V3_U
d4p3m3u5.obj	D48CL-PCI64U	32	33	5	abAsPhxData_D48Cl_P32_M33_V5_U
d4p6m3u3.obj	D48CL-PCI64U	64	33	3.3	abAsPhxData_D48Cl_P64_M33_V3_U
d4p6m3u5.obj	D48CL-PCI64U	64	33	5	abAsPhxData_D48Cl_P64_M33_V5_U
d4p6m6u3.obj	D48CL-PCI64U	64	66	3.3	abAsPhxData_D48Cl_P64_M66_V3_U
d4pe1.obj	D48CL-PE1	PCI Express			abAsPhxData_D48Cl_PE1

Note:

The file ‘d8p6m6v3.obj’ is for Camera Link operation with a pixel clock of up to 85MHz.

Extract the Firmware Object File(s)

The following gcc librarian/archiver command will extract the object file ‘pfw_api.o’ from the Phoenix Firmware Library archive file ‘libpfw.a’.

```
ar x libpfw.a pfw_api.o
```

Link the Application to the Required Firmware Object File(s)

The following makefile shows how to build the ‘phxinfo’ example application under DOS32 (using DJGPP), assuming that the Phoenix board is a D24CL in a 32-bit, 33MHz, 3.3volt PCI slot.

```
vpath %.c .. ../../common
INCDIR1 = ../../../../include
INCDIR2 = ../../common
```

```
vpath %.h $(INCDIR1) $(INCDIR2)

CPPFLAGS += -I$(INCDIR1) -I$(INCDIR2) -D_PHX_DOS32

phxinfo.exe : phxinfo.o common.o
  $(CC) -o phxinfo.exe phxinfo.o common.o pfw_api.o pfw_g.o d2p3m3v3.obj -lphx -
L. \
-Wl,--defsym -Wl,_abAsPhxData_D24Cl_P32_M33_V5=0 \
-Wl,--defsym -Wl,_abAsPhxData_D24Cl_PE1=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_P32_M33_V3=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_P32_M33_V5=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_P32_M33_V3_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_P32_M33_V5_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_P64_M33_V3=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_P64_M66_V3=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_P64_M33_V3_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_P64_M33_V5_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_P64_M66_V3_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D36_PE1=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl_P32_M33_V3=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl_P32_M33_V3_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl_P32_M33_V5_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl_P64_M33_V3=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl_P64_M66_V3=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl_P64_M33_V3_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl_P64_M33_V5_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl_P64_M66_V3_U=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl85_P64_M66_V3=0 \
-Wl,--defsym -Wl,_abAsPhxData_D48Cl_PE1=0

clean:
  del common.o
  del phxinfo.o
```

Notes:

- The unused firmware library symbols are redefined to zero using the ‘defsym’ option.
- The files ‘pfw_api.o’ and ‘pfw_g.o’ are always required.
- More than one firmware file may be linked in to the application.
- This applies to DOS32 with DJGPP, Linux and VxWorks operating systems.

Camera Control

EXPOSURE CONTROL

Phoenix has the ability to trigger the camera in order to operate in asynchronous reset mode. Most modern cameras have the ability to reset the sensor and internal timings so that it is ready to take another picture. This requires the frame grabber to drive a specific signal to the camera and with certain cameras we can also control the amount of light that is integrated on the sensor by controlling the width of the reset pulse. In certain situations the camera reset pulse width (integration time) may need to be short in order to capture fast moving images.

Exposure Control Output Signals

Two signals are output by Phoenix which may be used to control the exposure of LVDS cameras. These signals are CcIo_1 and CcIo_2.

- CcIo_1 is generally used to initiate the exposure (i.e. reset the image sensor). Some cameras call this PRIN (pixel reset).
- CcIo_2 generally determines the length of the exposure (EXSYNC).

However, the use of these signals varies from camera to camera. E.g. some cameras do not have a CcIo_1 type input signal, and the exposure is controlled entirely by CcIo_2.

The following table associate the specific software I/O bits ***PHX_IO_CCIO*** with their associated hardware exposure control signals.

Exposure Control Signal		D36 LVDS Camera connector		Notes
		Pins	Signal Name	
PHX_IO_CCIO bit 0		93,94 43,44	CcIoA1+, CcIoA1- CcIoB1+, CcIoB1-	PRIN (A) if configured for channel A PRIN (B) if configured for channel B
PHX_IO_CCIO bit 1		95,96 45,46	CcIoA2+, CcIoA2- CcIoB2+, CcIoB2-	EXSYNC if configured for channel A EXSYNC if configured for channel B
PHX_IO_CCIO_A bit 0		93,94	CcIoA1+, CcIoA1-	PRIN for channel A
PHX_IO_CCIO_A bit 1		95,96	CcIoA2+, CcIoA2-	EXSYNC for channel A
PHX_IO_CCIO_B bit 0		43,44	CcIoB1+, CcIoB1-	PRIN for channel B
PHX_IO_CCIO_B bit 1		45,46	CcIoB2+, CcIoB2-	EXSYNC for channel B

With Camera Link cameras, the exposure is controlled by one of the CCn signals on the Camera Link connector. These are controlled using the ***PHX_IO_CCOUT*** parameter within the Phoenix Library (PHX).

Exposure Control Signal		Camera Link connector		Notes
		Pins	Signal Name	
PHX_IO_CCOUT bit 0		5,18	CC1+, CC1-	CC1 for the current channel (A or B)
PHX_IO_CCOUT bit 1		17,4	CC2+, CC2-	CC2 for the current channel (A or B)
PHX_IO_CCOUT bit 2		3,16	CC3+, CC3-	CC3 for the current channel (A or B)
PHX_IO_CCOUT bit 3		15,2	CC4+, CC4-	CC4 for the current channel (A or B)
PHX_IO_CCOUT_A bit 0		5,18	CC1+, CC1-	
PHX_IO_CCOUT_A bit 1		17,4	CC2+, CC2-	

PHX_IO_CCOUT_A bit 2		3,16	CC3+, CC3-	
PHX_IO_CCOUT_A bit 3		15,2	CC4+, CC4-	
PHX_IO_CCOUT_B bit 0		5,18	CC1+, CC1-	
PHX_IO_CCOUT_B bit 1		17,4	CC2+, CC2-	
PHX_IO_CCOUT_B bit 2		3,16	CC3+, CC3-	
PHX_IO_CCOUT_B bit 3		15,2	CC4+, CC4-	

Exposure Control Methods

The exposure control signals can be driven in one of three ways.

Direct software control:	Either the ***PHX_IO_METHOD_WRITE***, ***PHX_IO_METHOD_BIT_SET*** or ***PHX_IO_METHOD_BIT_CLR*** method can be used.
Internal timer control:	Either the ***PHX_IO_METHOD_BIT_TIMER_POS*** or ***PHX_IO_METHOD_BIT_TIMER_NEG*** method can be used.
Acquisition trigger control:	Either the ***PHX_IO_METHOD_BIT_ACQTRIG_POS*** or ***PHX_IO_METHOD_BIT_ACQTRIG_NEG*** method can be used.

To illustrate the internal timer control method, for example, the following line in a PCF file will drive the CcIo_1 signal with its corresponding internal timer (timer_1 active low).

```
PHX_IO_CCIO,          PHX_IO_METHOD_BIT_TIMER_NEG, 1
```

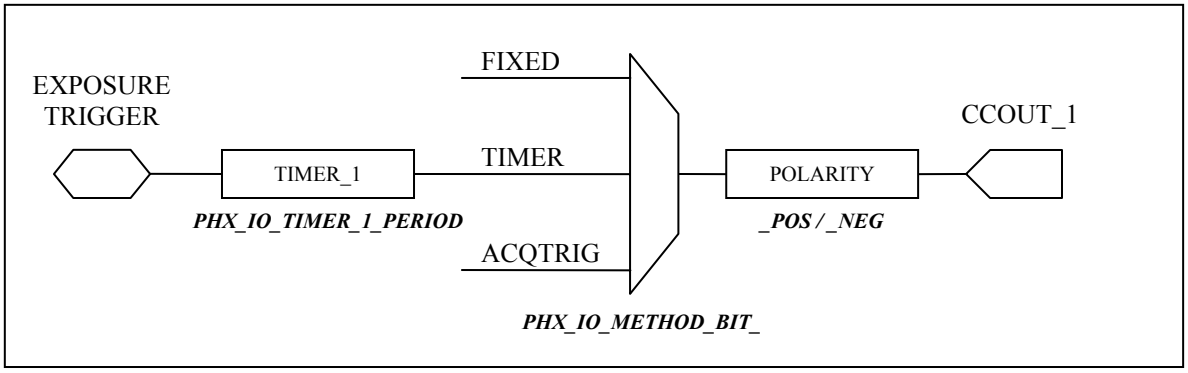
Also, the following lines of code, for example, will drive the CcIo_2 signal with its corresponding internal timer (timer_2 active high).

```
eParamValue = (etParamValue) PHX_IO_METHOD_BIT_TIMER_POS | 0x02;
PHX_ParameterSet( hCamera, PHX_IO_CCIO, &eParamValue );
```

For Camera Link cameras, the ***PHX_IO_CCOUT*** parameter should be used instead of ***PHX_IO_CCIO***.

Exposure Control Output Block Diagram

The following example block diagram shows how the CcOut_1 signal can be generated from either the Exposure Trigger, the Acquisition Trigger or be set to a fixed value (high or low).

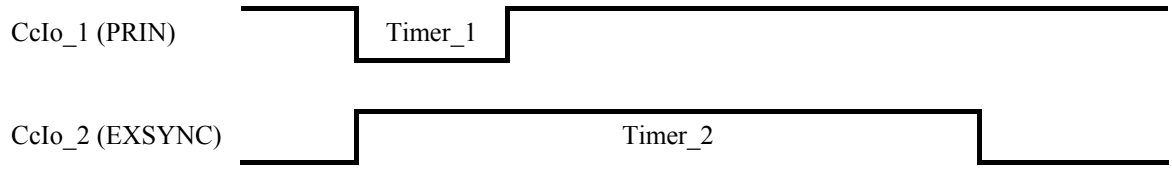


Exposure Control Internal Timers

Two internal timers (per channel) can be used to control the timing of the exposure control signals. The following table associates each timer with its corresponding exposure control signals.

Internal Timer		Exposure control signal	
		LVDS	Camera Link
Timer_1		CcIo_1	CC1, CC3
Timer_2		CcIo_2	CC2, CC4

The following diagram illustrates how a typical PRIN and EXSYNC pair of signals can be generated on CcIo_1 and CcIo_2 respectively.



Here, the width of CcIo_1 is defined by the width of Timer_1, which is set using the ***PHX_IO_TIMER_1_PERIOD*** parameter in the Phoenix Library. The width of CcIo_2 is defined by the width of Timer_2, which is set using the ***PHX_IO_TIMER_2_PERIOD*** parameter in the Phoenix Library. Note that the exposure time in this case is defined by duration that both of the above signals are high. It is the difference between Timer_2 period and Timer_1 period,
i.e. ***PHX_IO_TIMER_2_PERIOD - PHX_IO_TIMER_1_PERIOD***.

Initiating an Exposure

The start of an exposure causes the falling edge of CcIo_1 (PRIN) in the above diagram (and also the rising edge of CcIo_2 (EXSYNC)), for example. Exposure control can be initiated in one of four ways, as defined by the ***PHX_EXPTRIG_SRC*** parameter in the Phoenix Library.

PHX_EXPTRIG_ACQTRIG: The acquisition trigger (***PHX_ACQTRIG_SRC***) can initiate an exposure.

PHX_EXPTRIG_SWTRIG: A software trigger can initiate an exposure. This is achieved with a call to `PHX_Acquire( hCamera, PHX_EXPOSE, NULL );`

Note that this call returns immediately (rather than when the exposure pulse has finished), and cannot be queued, i.e. if an exposure pulse is currently being output then the PHX_Acquire call will have no effect.

PHX_EXPTRIG_TIMER: The internal timer (***PHX_LINETRIG_TIMER***) can be programmed to initiate exposure events repetitively. Note that ***PHX_LINETRIG_SRC*** must be set to ***PHX_LINETRIG_TIMER***. The line trigger timer can then be controlled using ***PHX_LINETRIG_TIMER_CTRL*** and ***PHX_LINETRIG_TIMER_PERIOD***.

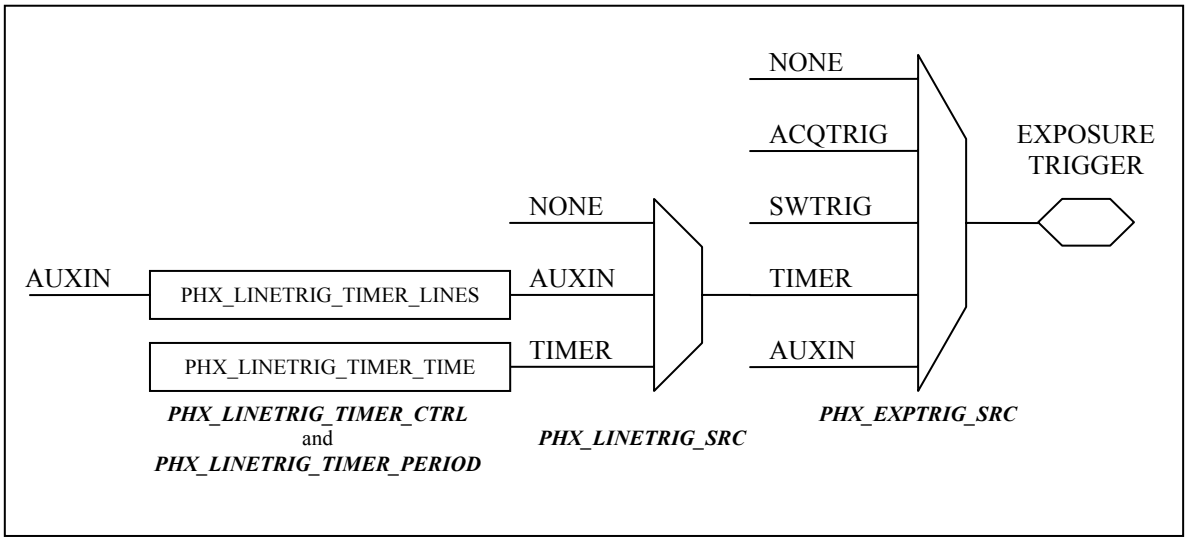
PHX_EXPTRIG_AUXIN_1_RISING:
PHX_EXPTRIG_AUXIN_1_FALLING:
PHX_EXPTRIG_AUXIN_2_RISING:
PHX_EXPTRIG_AUXIN_2_FALLING:
The two Relative signal inputs available are AuxIn1 and AuxIn2, and either of these can be selected to initiate an exposure. These should be connected to external devices such as a shaft encoder or other trigger devices.

Exposure Trigger Block Diagram

The following block diagram shows how the internal Exposure Trigger signal is generated.

When ***PHX_LINETRIG_TIMER_CTRL*** is ***PHX_LINETRIG_TIMER_TIME***, the timer outputs a square wave of period ***PHX_LINETRIG_TIMER_PERIOD*** microseconds.

When ***PHX_LINETRIG_TIMER_CTRL*** is ***PHX_LINETRIG_TIMER_LINES***, the timer divides the AUXIN signal by ***PHX_LINETRIG_TIMER_PERIOD*** lines.



ACQUISITION TRIGGER CONTROL

In many vision applications you may need to trigger the frame grabber from an external source to acquire a new image. The source could arrive from a variety of sources and therefore Phoenix has number of different options for trigger inputs.

External Trigger Source

The following tables associate the specific software parameter for ***PHX_ACQTRIG_SRC*** with its associated hardware input

PHX_ACQTRIG_SRC Value		Opto-Isolated connector		Notes
		Pins	Signal Name	
PHX_ACQTRIG_OPTO_1		1,2 11,12	OptoA1 Signal, GND OptoB1 Signal, GND	If configured for channel A If configured for channel B
PHX_ACQTRIG_OPTO_2		3,4 13,14	OptoA2 Signal, GND OptoB2 Signal, GND	If configured for channel A If configured for channel B
PHX_ACQTRIG_OPTO_A1		1,2	OptoA1 Signal, GND	
PHX_ACQTRIG_OPTO_A2		3,4	OptoA2 Signal, GND	
PHX_ACQTRIG_OPTO_B1		11,12	OptoB1 Signal, GND	
PHX_ACQTRIG_OPTO_B2		13,14	OptoB2 Signal, GND	
PHX_ACQTRIG_AUXIN_1		5,6 15,16	AuxInA1+, AuxInA1- AuxInB1+, AuxInB1-	If configured for channel A If configured for channel B
PHX_ACQTRIG_AUXIN_2		7,8 17,18	AuxInA2+, AuxInA2- AuxInB2+, AuxInB2-	If configured for channel A If configured for channel B
PHX_ACQTRIG_AUXIN_A1		5,6	AuxInA1+, AuxInA1-	
PHX_ACQTRIG_AUXIN_A2		7,8	AuxInA2+, AuxInA2-	
PHX_ACQTRIG_AUXIN_B1		15,16	AuxInB1+, AuxInB1-	
PHX_ACQTRIG_AUXIN_B2		17,18	AuxInB2+, AuxInB2-	

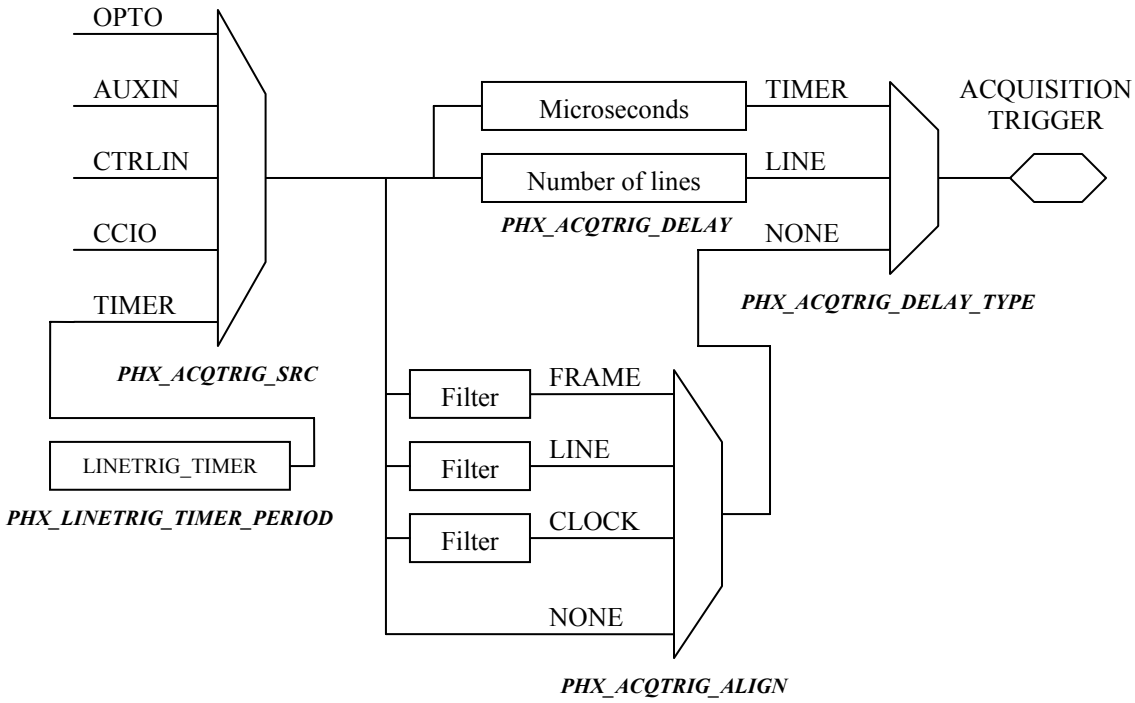
PHX_ACQTRIG_SRC Value		D36 LVDS Camera connector		Notes
		Pins	Signal Name	
PHX_ACQTRIG_CTRLIN_1 (frame enable)		85,86 35,36	CtrlInA1+, CtrlInA1- CtrlInB1+, CtrlInB1-	If configured for channel A If configured for channel B
PHX_ACQTRIG_CTRLIN_2 (line enable)		87,88 37,38	CtrlInA2+, CtrlInA2- CtrlInB2+, CtrlInB2-	If configured for channel A If configured for channel B
PHX_ACQTRIG_CTRLIN_3 (data enable)		89,90 39,40	CtrlInA3+, CtrlInA3- CtrlInB3+, CtrlInB3-	If configured for channel A If configured for channel B
PHX_ACQTRIG_CCIO_1		93,94 43,44	CcIoA1+, CcIoA1- CcIoB1+, CcIoB1-	If configured for channel A If configured for channel B
PHX_ACQTRIG_CCIO_2		95,96 45,46	CcIoA2+, CcIoA2- CcIoB2+, CcIoB2-	If configured for channel A If configured for channel B
PHX_ACQTRIG_CCIO_A1		93,94	CcIoA1+, CcIoA1-	
PHX_ACQTRIG_CCIO_A2		95,96	CcIoA2+, CcIoA2-	

PHX_ACQTRIG_CCIO_B1		43,44	CcIoB1+, CcIoB1-	
PHX_ACQTRIG_CCIO_B2		45,46	CcIoB2+, CcIoB2-	
PHX_ACQTRIG_TIMER		-		*

***** The ***PHX_ACQTRIG_TIMER*** option is a special case, no external trigger is needed. The acquisition rate is driven by the onboard timer.

Acquisition Trigger Block Diagram

The following block diagram shows how the internal Acquisition Trigger signal is generated.



Internal Control

Phoenix has various options over which edges or levels of the trigger source are used, as controlled by the ***PHX_ACQTRIG_TYPE*** parameter. Note that the default setting for the acquisition trigger type is ***PHX_ACQTRIG_NONE***, i.e. Phoenix does not wait for an acquisition trigger before starting an image capture. Hence this parameter must be set to a valid edge or level in order to use acquisition trigger control.

The ***PHX_ACQTRIG_ALIGN*** parameter allows the trigger source to be aligned to the pixel clock, line enable or frame enable signals if required.

An internal programmable timer can be used to delay the start of the image capture (after receiving an external trigger) by a number of lines or a time period. This is useful where the camera is physically offset from the trigger source for example. The relevant parameters are ***PHX_ACQTRIG_DELAY_TYPE*** and ***PHX_ACQTRIG_DELAY***.

LINE SCAN CONTROL

The ***PHX_CAM_TYPE*** parameter should be set to ***PHX_LINESCAN_ROI*** for example in order set the Phoenix board into line scan mode. In line scan mode, Phoenix uses the Line Enable signal to initiate the capture of image data instead of the Frame Enable signal used during area scan mode.

The relevant Exposure control and / or Acquisition Trigger control settings must then be used, depending upon the setup of the camera and system. The following Phoenix Configuration File is shown as an example.

```
# ============================
# Phoenix Configuration File
# Example Linescan
# ============================

# Camera Specific Settings
# ------------------------
# Standard 8 bit mono 1024 length linescan camera setting.
# Replace these with specific settings for your camera.
PHX_BOARD_TYPE,             PHX_DIGITAL
PHX_CAM_TYPE,               PHX_CAM_LINESCAN_ROI
PHX_CAM_SRC_DEPTH,          8
PHX_CAM_SRC_COL,            PHX_CAM_SRC_MONO
PHX_CAM_ACTIVE_XOFFSET,     0
PHX_CAM_ACTIVE_XLENGTH,     1024
PHX_CAM_HTAP_DIR,           PHX_CAM_HTAP_LEFT
PHX_CAM_HTAP_TYPE,          PHX_CAM_HTAP_LINEAR
PHX_CAM_HTAP_NUM,           1
PHX_CAM_VTAP_DIR,           PHX_CAM_VTAP_TOP
PHX_CAM_VTAP_TYPE,          PHX_CAM_VTAP_LINEAR
PHX_CAM_VTAP_NUM,           1
PHX_CAM_CLOCK_POLARITY,     PHX_CAM_CLOCK_POS


# System Specific Settings
# ------------------------
# Exposure Control
# ----------------
#                           Enable CamIo outputs
PHX_IO_CCIO_OUT,            PHX_ENABLE
#                           Start exposure with AuxIn2 source, ie shaft encoder
PHX_EXPTRIG_SRC,            PHX_EXPTRIG_AUXIN_2_RISING
#                           CcIo_1 is ExpCtrl to Camera,
#                           CcIo_2 is NewFrame to camera
PHX_IO_CCIO,                PHX_IO_METHOD_BIT_TIMER_NEG, 3
#                           10us ExpCtrl width
PHX_IO_TIMER_1_PERIOD,      10
#                           510us NewFrame width, ie 500us fixed exposure
PHX_IO_TIMER_2_PERIOD,      510
# Acquisition Trigger Control
# ---------------------------
#                           Frame Enable A is AcqTrig source
PHX_ACQTRIG_SRC,            PHX_ACQTRIG_CTRLIN_A1
#                           Trigger on each rising edge of the AcqTrig source
PHX_ACQTRIG_TYPE,           PHX_ACQTRIG_EACH_POS_EDGE


# Add Application Specific Settings
# ---------------------------------
```

Acquisition Control

USING A CALLBACK

The Phoenix libraries use interrupt driven events to inform the application when a specific acquisition event happens on the Phoenix board. The recommended method for handling these events is to use a Callback function. This function is used to process or handle the capture of images.

The definition of the Callback function is given below.

```
void CallBack( tHandle hCamera, ui32 dwMask, void* pvParams )
```

The hCamera value is the camera handle.

The dwMask value is used to distinguish between the possible interrupt events described below. It must be noted that more than one interrupt can occur at the same time.

The pvParams value can be used to pass in the address of a user defined context.

The supported interrupt events are listed below.

Value	Description
PHX_INTRPT_BUFFER_READY	Generated on every frame buffer that is filled.
PHX_INTRPT_CAPTURE_COMPLETE	Sequence Capture complete*****.
PHX_INTRPT_FRAME_START	Start of frame, including all of the active area.
PHX_INTRPT_FRAME_END	End of frame, including all of the active area.
PHX_INTRPT_LINE_START	Start of Line.
PHX_INTRPT_LINE_END	End of Line.
PHX_INTRPT_ACQ_TRIG_START	Start of Acquisition trigger, when the trigger is asserted.
PHX_INTRPT_ACQ_TRIG_END	End of Acquisition trigger.
PHX_INTRPT_FIFO_OVERFLOW	FIFO overflow******.
PHX_INTRPT_TIMEOUT	Timeout.
PHX_INTRPT_GLOBAL_ENABLE	Enable all of the above interrupts.

***** This is only generated if ***PHX_ACQ_CONTINUOUS*** is disabled.

****** This occurs if the Phoenix cannot transfer the data to the destination fast enough and therefore data is lost.

Implementation

Any of the above interrupts can be enabled or disabled using the ***PHX_ParameterSet*** with the ***PHX_INTRPT_SET*** parameter. You can either call this function multiple times with the individual parameter or 'OR' the required parameters together.

The camera configuration files supplied contain two calls to set the interrupts, these are the minimum required to enable Callback functionality.

PHX_INTRPT_GLOBAL_ENABLE **must** be set to enable any interrupt to be generated.

PHX_INTRPT_BUFFER_READY (previously known as PHX_INTRPT_DMA) is set so that an interrupt is generated every time an image buffer is filled.

In addition to these you could, for example, set the Start and End of frame interrupts.

```
eParamValue = PHX_INTRPT_FRAME_START | PHX_INTRPT_FRAME_END;
PHX_ParameterSet( hCamera, PHX_INTRPT_SET, &eParamValue );
```

The Callback function is registered on the call to ***PHX_Acquire*** and will be called on each interrupt that has been set.

```
PHX_Acquire( hCamera, PHX_START, Callback );
```

Example

This example demonstrates how a callback function can be registered to be executed following the end of each image capture. In this example, continuous acquisition is used to repeatedly capture images into the 10 buffers. In addition, the default error handler is registered in order to respond to any errors that might occur.

```
/* Define an application specific structure to hold user information */
typedef struct
{
   /* Event counters */
   volatile ui32 nBufferReadyCount;

   /* Control Flags */
   volatile tFlag fFifoOverFlow;
} tPhxExample;

/* Callback function to handle image capture events. */
void Callback( tHandle hCamera, ui32 dwMask, void* pvParams )
{
   tPhxExample *psPhxExample = (tPhxExample*) pvParams;

   (void) hCamera;

   /* Handle the Buffer Ready event */
   if ( PHX_INTRPT_BUFFER_READY & dwInterruptMask ) {
      /* Increment the Buffer Ready Count */
      psPhxExample->nBufferReadyCount++;
   }

   /* Fifo Overflow */
   if ( PHX_INTRPT_FIFO_OVERFLOW & dwInterruptMask ) {
      psPhxExample->fFifoOverFlow = TRUE;
   }
}


/* Main function. */

tHandle      hCamera = 0;
ui32         dwNumberOfImages;
etParamValue eParamValue;
stImageBuff  stBuffer;
tPhxExample  sPhxExample;            /* User defined Event Context */

/* Initialise the user defined Event context structure */
memset( &sPhxExample, 0, sizeof( tPhxExample ) );

/* Configure the board with the config file */
PHX_CameraConfigLoad( &hCamera, "camera.pcf", PHX_BOARD_AUTO | PHX_DIGITAL,
                                              PHX_ErrHandlerDefault );

/* Sequence capture */
dwNumberOfImages  = 10;
PHX_ParameterSet( hCamera, PHX_ACQ_NUM_IMAGES, &dwNumberOfImages );

/* Continuous capture to image memory */
eParamValue = PHX_ENABLE
PHX_ParameterSet( hCamera, PHX_ACQ_CONTINUOUS, &eParamValue );
```

```
/* Define events which activate the callback */
eParamValue = PHX_INTRPT_BUFFER_READY | PHX_INTRPT_FIFO_OVERFLOW;
PHX_ParameterSet( hCamera, PHX_INTRPT_SET, &eParamValue );

/* Setup our own event context */
PHX_ParameterSet( hCamera, PHX_EVENT_CONTEXT, (void *) &sPhxExample );

/* Start acquiring a sequence of 10 images continuously and register the
 * callback
 */
PHX_Acquire( hCamera, PHX_START, CallBack );

/* Callback() is called with each event */
while( (   100  > sPhxExample.nBufferReadyCount ) &&
       ( FALSE == sPhxExample.fFifoOverFlow     ) )
{
   /* Get the info for the last acquired buffer */
   PHX_Acquire( hCamera, PHX_BUFFER_GET, &stBuffer );

   /* Process the newly acquired buffer,
    * the video data can be accessed at stBuffer.pvAddress
    */
   printf("EventCount = %5d\r", sPhxExample.nBufferReadyCount );

   /* Having processed the data, release the buffer
    * ready for further image data
    */
   PHX_Acquire( hCamera, PHX_BUFFER_RELEASE, NULL );
}

/* Release camera resources */
PHX_CameraRelease( &hCamera );
```

SEQUENCE CONTROL

Before starting an acquisition, Phoenix will link together a series of acquisition buffers to form a sequence. The number of images in the sequence is defined by the ***PHX_ACQ_NUM_IMAGES*** parameter. If this is 1 for example, then only a single image will be captured. If it is 5 for example, then there will be 5 buffers in the sequence.

The ***PHX_ACQ_CONTINUOUS*** parameter controls how Phoenix behaves when it gets to the end of the sequence.

- If this is ***PHX_DISABLE***, then acquisition will stop (this is known as **single shot** acquisition).
- If this is ***PHX_ENABLE***, then the last buffer in the sequence is linked to the first buffer in the sequence, and the acquisition runs continuously (this is known as **continuous** acquisition, or can also be known as **live** acquisition).

During the continuous acquisition of an image sequence, after each buffer has been filled once, there is a mechanism which controls whether or not subsequent buffers are overwritten. This concept is known as acquisition **blocking**. The ***PHX_ACQ_BLOCKING*** parameter controls how this happens, in conjunction with a buffer release mechanism. It either permits or prevents the acquisition buffers being overwritten by new image data.

- If this is ***PHX_ENABLE***, then the next buffer in the sequence is only overwritten if it has previously been released (i.e. with a call to ***PHX_Acquire(, PHX_BUFFER_RELEASE,);***). If the next buffer in the sequence has not been released, then acquisition will halt, and only continue when the buffer has been released. Hence, when an application is processing a particular buffer, the data in that buffer is guaranteed to be contiguous (i.e. not contain part of two images, a concept known as **tearing**).

- If this is ***PHX_DISABLE***, then the next buffer in the sequence will always be overwritten, irrespective of whether or not it has been released.

The ***PHX_ACQ_BUFFER_START*** parameter can be used to specify which buffer of a sequence will be captured into first.

The ***PHX_BUFFER_READY_COUNT*** parameter allows an application to reduce the number of ***PHX_INTRPT_BUFFER_READY*** events that are issued if necessary. For example, with a sequence capture of 10 images and a ***PHX_BUFFER_READY_COUNT*** of 2, then a ***PHX_INTRPT_BUFFER_READY*** event would only be signalled after each pair of frames were captured. In other words, 5 interrupts would be generated during each 10 frame sequence.

Examples

The following example demonstrates the capture of a single image.

```
tHandle hCamera = 0;
ui32 dwNumberOfImages, dwEvent;
stImageBuff stBuffer;


/* Configure the hardware from a configuration file
 * (sets up camera parameters, ROI etc.)
 */
PHX_CameraConfigLoad( &hCamera, "camera.pcf", PHX_BOARD_AUTO | PHX_CHANNEL_A |
                                              PHX_DIGITAL, NULL );

/* Single image capture */
dwNumberOfImages  = 1;
PHX_ParameterSet( hCamera, PHX_ACQ_NUM_IMAGES, &dwNumberOfImages );

/* Start the acquisition and wait until complete */
PHX_Acquire( hCamera, PHX_START, NULL );
PHX_Acquire( hCamera, PHX_CHECK_AND_WAIT, &dwEvent );

/* Obtain a pointer to the newly acquired image buffer structure */
if( PHX_INTRPT_BUFFER_READY & dwEvent )
   PHX_Acquire( hCamera, PHX_BUFFER_GET, &stBuffer );

/* Image processing goes here … */

/* Release the camera resources */
PHX_CameraRelease( &hCamera );
```

This is similar to the above example except that a sequence of 10 images is acquired.

```
/* Set the number of images to capture */
dwNumberOfImages = 10;
PHX_ParameterSet( hCamera, PHX_ACQ_NUM_IMAGES, &dwNumberOfImages );

/* Start the acquisition of all 10 images */
PHX_Acquire( hCamera, PHX_START, NULL );

/* Wait for the first image capture to complete and obtain a pointer to it */
PHX_Acquire( hCamera, PHX_CHECK_AND_WAIT, &dwEvent );
if( PHX_INTRPT_BUFFER_READY & dwEvent )
   PHX_Acquire( hCamera, PHX_BUFFER_GET, &stBuffer );

/* Repeat from the above PHX_CHECK_AND_WAIT for the remaining nine images */

/* Release camera resources *.
PHX_CameraRelease( &hCamera );
```

HOW PHOENIX USES MEMORY BUFFERS

For maximum flexibility, all the Phoenix products support the use of three different types of destination memory; Internal, Virtual and Physical. These different memory types are selected via the ***PHX_ParameterSet*** function called with the ***PHX_DST_PTR_TYPE*** parameter.

Internal Memory

As the name suggests, Internal memory is memory allocated and controlled by the Phoenix libraries. The libraries automatically calculate the required buffer size based upon the source image width and height, and the destination format. The libraries then request this amount of system memory from the operating system and ensure that this memory is physically available, i.e. not swapped out to disk. This memory is then retained either until the board is released (i.e. ***PHX_CameraRelease*** is called) or a parameter is changed which results in the need for different sized or numbers of buffers.

This method is the simplest, as all the memory allocation is transparent to the user. However there are occasions when finer control of the memory usage is required, as discussed below.

Retrieving the Buffer Details

At any time the user's application can call ***PHX_Acquire*** with the ***PHX_BUFFER_GET*** parameter to obtain the address of the buffer. This is usually performed as part of the user installed event handler (callback) function when an end of DMA event is signalled. Having finished processing an individual memory buffer, it is important to call ***PHX_Acquire*** with ***PHX_BUFFER_RELEASE*** so that the buffer may be re-used.

The following code snippet gives an example of how to retrieve the current buffer information.

```
/* Allocate an stImageBuff structure to contain the buffer info */
stImageBuff sImageBuffs;


/* Get the info for the current buffer
 * stImageBuff.pvAddress and stImageBuff.pvContext contain the virtual address
 * and user defined context for the current buffer.
 * Note that pvContext can be set by the user's application for Virtual or
 * Physical buffers. For Internal buffers, the value is set by the Phoenix
 * libraries as the buffer number.
 */
PHX_Acquire( hPhoenix, PHX_BUFFER_GET, (void *) &stImageBuff );


/* Process Video Data Here */
{ .. }


/* Release the buffer */
PHX_Acquire( hPhoenix, PHX_BUFFER_RELEASE, NULL );
```

As it is recommended that the user's application does not perform any data processing with in the event handler (callback) routine, the above code is best called from within a separate processing thread. However it is allowed to make the ***PHX_BUFFER_GET*** call within the user's event handler (callback) function and the ***PHX_BUFFER_RELEASE*** call from the processing thread, if required.

Virtual Memory

Virtual memory is used when the user wishes to explicitly control the memory allocation. In this case the user allocates memory, and passes the virtual address pointer to Phoenix. This technique is most commonly used in conjunction with applications which must use predefined memory buffers, for example with third party drivers, and removes the need for inefficient memory copying.

In order to do this the following steps must be performed in the user's application.

```
/* Allocate an array of stImageBuff structures to contain the virtual addresses.
 */
stImageBuff asImageBuffs[kiNumOfBuffs + 1];
etParamValue eParamValue;


/* Assign the data to each stImageBuff instance.
 * Note the pvAddress member is assigned with the buffer address, and
 * the pvContext member is assigned with a user specified value.
 * The user specified pvContext value is returned by Phoenix
 * so that each buffer can be distinguished, if required.
 */
for ( int i = 0; i < kiNumOfBuffs; i++ ) {
   asImageBuffs[i].pvAddress = (void *) malloc( kiImageSizeInBytes );
   asImageBuffs[i].pvContext = (void *) i;
}


/* Terminate the end of the array with NULL parameters */
asImageBuffs[kiNumOfBuffs].pvAddress = (void *) NULL;
asImageBuffs[kiNumOfBuffs].pvContext = (void *) NULL;


/* Pass the buffer information to Phoenix, and instruct it to use these buffers.
 */
PHX_ParameterSet( hPhoenix, PHX_DST_PTRS_VIRT, (void *) asImageBuffs );
eParamValue = PHX_DST_PTR_USER_VIRT;
PHX_ParameterSet( hPhoenix, PHX_DST_PTR_TYPE, (void *) &eParamValue );
```

Physical Memory

This method is used when transferring data to another physical device on the PCI bus rather than system memory. For example, this method could be used to send data directly to a DSP card for processing. The other device does not have to be on the same PCI bus segment, but it must be directly addressable from the PCI bus containing Phoenix.

Note that the physical addresses referred to here are the addresses output on the PCI bus in order to access the target device. On most operating systems this address is constant for all sources, but on others (most notably VxWorks) the target physical address varies depending upon on the source, and any intermediate bridge chip configurations.

Below is some example code for using Physical address buffers:

```
/* Allocate an array of stImageBuff structures to contain the
 * virtual addresses of the physical address and length pairs lists
 * (also known as Scatter Gather Tables).
 */
stImageBuff asImageBuffs[kiNumOfBuffs + 1];
etParamValue eParamValue;


/* Assign the data to each stImageBuff instance.
 * Note the pvAddress member is assigned with the virtual address of the
```

```
 * physical address and length pairs list.
 * The user specified pvContext value is returned by Phoenix so that each
 * buffer can be distinguished, if required.
 */
for ( int i = 0; i < kiNumOfBuffs; i++ ) {
   /* Add the entry to the list of buffers */
   ui32 *pdwPhysAddrLenList = (ui32*) malloc( ( kiNumPhysAddrEntries + 1) * 8 );
   asImageBuffs[i].pvAddress = (void *) pdwPhysAddrLenList;
   asImageBuffs[i].pvContext = (void *) i;

   /* Add each Physical Address and Length Pair entry to the list */
   *pdwPhysAddrLenList++ = (ui32) dwPhysAddress1;
   *pdwPhysAddrLenList++ = (ui32) dwPhysLength1;
   *pdwPhysAddrLenList++ = (ui32) dwPhysAddress2;
   *pdwPhysAddrLenList++ = (ui32) dwPhysLength2;
   ..
   *pdwPhysAddrLenList++ = (ui32) dwPhysAddressN;
   *pdwPhysAddrLenList++ = (ui32) dwPhysLengthN;
   /* Terminate the list */
   *pdwPhysAddrLenList++ = (ui32) NULL;
   *pdwPhysAddrLenList++ = (ui32) 0;
}

/* Terminate the array of buffers with NULL parameters */
asImageBuffs[kiNumOfBuffs].pvAddress = (void *) NULL;
asImageBuffs[kiNumOfBuffs].pvContext = (void *) NULL;

/* Pass the buffer information to Phoenix, and instruct it to use these buffers.
 */
PHX_ParameterSet( hPhoenix, PHX_DST_PTRS_PHYS, (void *) asImageBuffs );
eParamValue = PHX_DST_PTR_USER_PHYS;
PHX_ParameterSet( hPhoenix, PHX_DST_PTR_TYPE, (void *) &eParamValue );
```

IMAGE CORRUPTION (FIFO OVERFLOW)

During image acquisition, Phoenix places captured image data into an onboard PCI FIFO. The image data is then transferred across the PCI bus from the FIFO to a system memory buffer. If there is sufficient PCI bus bandwidth available, then all of the image data from a particular frame will be transferred to the correct memory buffer.

Image Slip

In certain circumstances, access to the PCI bus may be temporarily denied to the Phoenix board, and as a result, data in the PCI FIFO can be overwritten before it is transferred across the PCI bus. This is termed a FIFO OVERFLOW. Data is lost, and the resulting image in the memory buffer will be corrupted, as it will contain data from a subsequent frame in place of the data that was lost. The effect of this is that subsequent images will appear to have shifted up and / or left. See the following example image :-

When this happens, a ***PHX_INTRPT_FIFO_OVERFLOW*** event is generated by the SDK. The user should then call

```
PHX_Acquire( hCamera, PHX_ABORT, NULL );
```

to abort the current acquisition, and then restart acquisition with

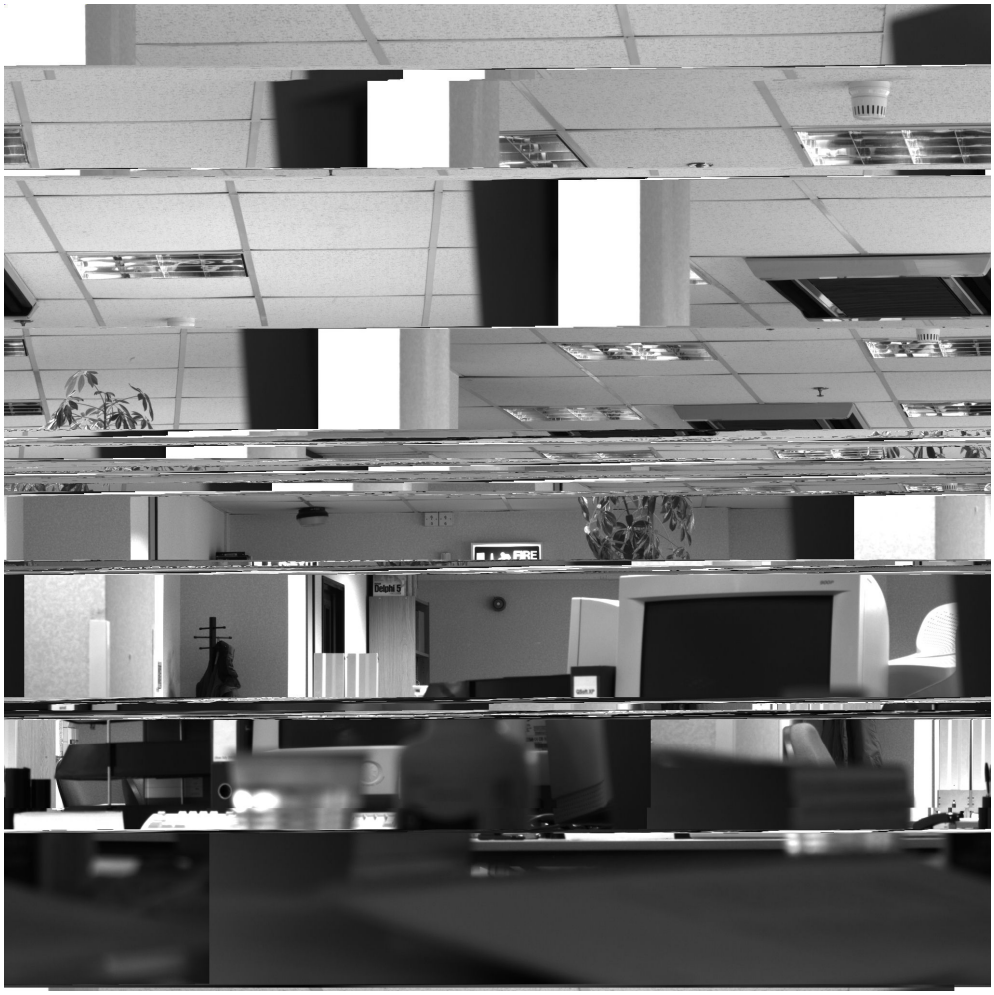
```
PHX_Acquire( hCamera, PHX_START, pvVar );
```

in order to resume the capture of normal images.

Severe Image Corruption

By default, Phoenix will convert the captured image data from the camera output format to the required destination format of the buffer allocated on the host or graphics memory. If the destination buffer is to be displayed, its pixel depth is usually set to 32-bits. Hence Phoenix will transfer 4-bytes for every captured pixel even if the camera data is only 8-bits.

If a large image is being transferred at a high frame rate, such that the data rate exceeds the PCI bus bandwidth, (most commonly seen when using a 32-bit PCI bus), then severe image corruption can be seen, as shown in the following image :-



In this instance, simply aborting and restarting acquisition will make no difference. The solution is to reduce the data rate across the PCI bus.

Within an application, the user should change the format of the destination buffer (***PHX_DST_FORMAT***) so that it matches the camera's output format (for example an 8-bit camera will need ***PHX_DST_FORMAT_Y8***, whereas a 16-bit camera will need ***PHX_DST_FORMAT_Y16***). If the image still needs to be displayed, then a display buffer can be created, and the destination buffer can be converted into the display buffer with a call to

```
PIL_Convert( hSrcBuffer, hDstBuffer );
```

Alternatively, from within the PhoenixCapture application :-

- go to the "Sequence" tab,
- check the "Camera Format Transfer Enable" option,
- then select the required "Processed Buffer Format" from the drop down list.
- "Apply" these settings and press "Live".

The image should now no longer be corrupted.

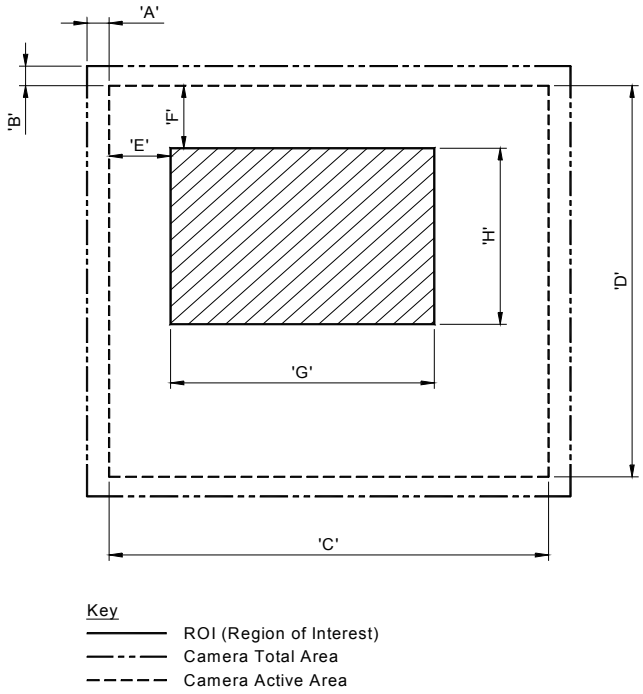
Image Control

IMAGE DATA CONTROL

The following parameters are used to describe the format of the physical image data generated by the camera, and its relationship to that captured in memory.

Camera image data

This diagram shows the format of the data generated by the camera.



Key

A - PHX_CAM_ACTIVE_XOFFSET

B - PHX_CAM_ACTIVE_YOFFSET

C - PHX_CAM_ACTIVE_XLENGTH

D - PHX_CAM_ACTIVE_YLENGTH

E - PHX_ROI_SRC_XOFFSET

F - PHX_ROI_SRC_YOFFSET

G - PHX_ROI_XLENGTH

H - PHX_ROI_YLENGTH

The outer box designates the total image data generated by the camera. This is dictated by the camera hardware and cannot be altered using the Phoenix libraries.

The box within the outer box designates the active area. This is the valid image data generated by the camera. Its size is defined using parameters C and D and its position relative to the total image data origin using parameters A and B. The settings for these parameters can be obtained from the technical specifications of the camera being interfaced.

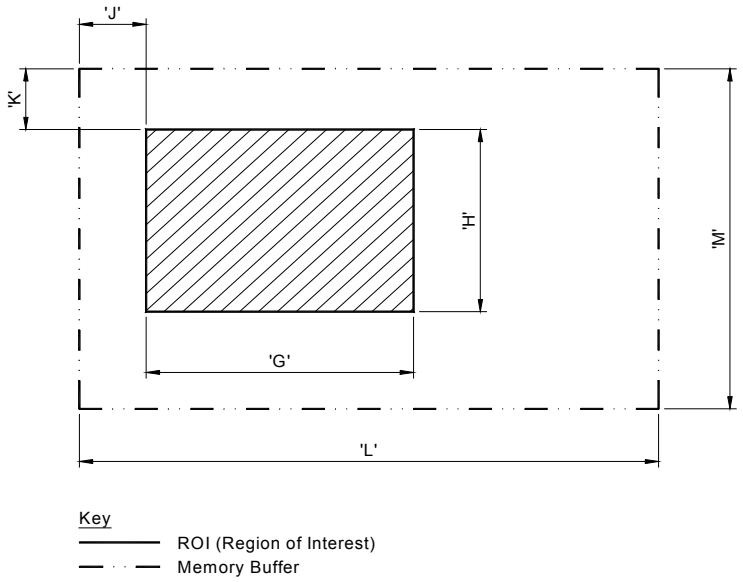
The shaded region designates the Region Of Interest (ROI). This is the image data the user is interested in capturing. Its size is defined using parameters G and H, and its position relative to the active area origin by

E and F. The ROI cannot be positioned in such a way that it exceeds the boundary defined by the active area parameters.

The source image data may be sub-sampled by setting the parameters ***PHX_ACQ_XSUB*** and ***PHX_ACQ_YSUB***. Thus the effective ROI can be reduced for faster update speed / lower data rates etc.

Destination buffer

The following diagram shows how the ROI specified above is positioned when captured in a memory buffer.



Key

G - PHX_ROI_XLENGTH

H - PHX_ROI_YLENGTH

J - PHX_ROI_DST_XOFFSET

K - PHX_ROI_DST_YOFFSET

L - PHX_BUF_DST_XLENGTH

M - PHX_BUF_DST_YLENGTH

The outer box designates the destination memory buffer used to store image data. This can either be allocated by the user or by the Phoenix libraries (see ***PHX_DST_PTR_TYPE***).

With a user allocated buffer, the size is defined using parameters L and M. The ROI may be placed anywhere within this buffer, its offset from the buffer origin defined using parameters J and K.

With a library allocated buffer the size is identical to the source ROI (unless subsampling is being used) so J and K are zero and L and M are identical to G and H (changing these parameters will have no effect).

The destination format of the output image is defined using ***PHX_DST_FORMAT***. The camera source data is mapped in hardware by the Phoenix board before being written to the destination buffer. Note that the user should be aware that using high pixel rate cameras with certain destination formats (e.g. ***PHX_DST_FORMAT_RGBX32***) can cause loss of image data/image breakup. This is simply due to the physical limitation of the bandwidth of the PCI bus. In these cases, a lower data rate output format should be selected, or subsampling/binning utilised.

In order to capture image data into user allocated buffers, ***PHX_DST_PTR_TYPE*** should be set to ***PHX_DST_PTR_USER_VIRT***, and a buffer(s) provided using the ***PHX_DST_PTRS*** parameter. When using user allocated buffers, the library automatically frees up any internally allocated buffers and captures

image data into the user's buffers. Note that unlike with internally allocated buffers, it is the user's responsibility to free the user allocated buffers when no longer required.

LOOK-UP TABLE CONTROL

Phoenix provides hardware Look-Up Tables (LUTs) for manipulating the camera image data prior to being mapped into the destination image format. This may be used for performing image correction (brightness, contrast, gamma etc), or other custom user mappings, such as dynamic range cropping and binary thresholding in real time.

Multiple LUTs can be defined per image line, and per camera tap/colour component. Since Phoenix only has a finite amount of physical memory available for holding the hardware LUTs, the number available is dictated by the camera source image depth. See ***PHX_LUT_COUNT*** for a definition of available LUTs.

Upon initialisation, the libraries create a set of straight through LUTs based on default settings.

A LUT is described by the following structure.

```
struct tLutInfo {
   ui32  dwLut;
   ui32  dwColour;
   ui32  dwTap;
   ui32  dwBrightness;
   ui32  dwContrast;
   ui32  dwGamma;
   ui32  dwFloor;
   ui32  dwCeiling;
   ui16  *pwLutData;
   ui32  dwSize;
}
```

The user may call ***PHX_ParameterGet*** using the ***PHX_LUT_INFO*** parameter and a pointer to a tLutInfo structure to inquire the current settings of a particular internal LUT.

The elements of tLutInfo are defined as follows.

LUT identifiers

dwLut, dwColour and dwTap identify the particular LUT and can take the following values :-

dwLut	The LUT number across a line. 0 to PHX_LUT_COUNT. 0 is the first LUT across the line.
dwColour	The colour component. Always 0 for a mono camera. 0 to 3 for RGB (0 is R, 1 is G and 2 is B).
dwTap	The camera tap number. Always 0 for an RGB camera. 0 or 1 for a mono camera.

LUT settings

dwBrightness, dwContrast, dwGamma, dwFloor and dwCeiling are the current LUT settings. They can take the following values :-

dwBrightness	The brightness of the LUT. Range 0 to 200. Default 100.
dwContrast	The contrast of the LUT. Range 0 to 200. Default 100.
dwGamma	The gamma of the LUT. Range 1 to 400. Default 100.
dwFloor	Defines the minimum data value of the LUT. If any LUT value calculated using the brightness, contrast and gamma settings above is less than the dwFloor value, it will be clamped to the dwFloor value. Range 0 to ((1 << camera bit depth) – 1). Default 0.
dwCeiling	Defines the maximum value of the LUT. If any LUT value calculated using the brightness, contrast and gamma settings is greater than the dwCeiling value, it will be clamped to the dwCeiling value. Range 0 to ((1 << camera bit depth) – 1). Default (1 << camera bit depth) – 1.

The above settings may be modified and ***PHX_ParameterSet*** called using the ***PHX_LUT_INFO*** parameter. This will cause the library to regenerate the LUT using the new settings.

User allocated LUT data

In order to use a custom LUT, the user creates a buffer containing the mapping data (each entry being 16 bits regardless of camera depth), and sets pwLutData to point to this.

pwLutData	Pointer to a user allocated LUT buffer.

PHX_ParameterSet is then called using the ***PHX_LUT_INFO*** parameter which causes the library to destroy its internal LUT and use the custom LUT data. Note that in this case, dwBrightness, dwContrast, dwGamma, dwFloor and dwCeiling are ignored.

When retrieving the details of a LUT using ***PHX_ParameterGet***, pwLutData is always returned as NULL if the LUT is internal. Thus the user has no access to any internally allocated LUT data.

Setting the Brightness of an Internal LUT

The following example demonstrates how to inquire the current settings of an internal LUT and modify its brightness.

```
tHandle hCamera = 0;
struct tLutInfo sLut;


PHX_CameraConfigLoad( &hCamera, NULL, PHX_BOARD1 | PHX_DIGITAL, NULL );

/* Inquire the current settings of an internal LUT */
sLut.dwLut = 0;
sLut.dwColour = 0;
sLut.dwTap = 0;
PHX_ParameterGet( hCamera, PHX_LUT_INFO, &sLut );

/* Modify the brightness setting */
sLut.dwBrightness = 200;

/* Force the libraries to recreate this LUT with the new setting */
PHX_ParameterSet( hCamera, PHX_LUT_INFO | PHX_CACHE_FLUSH, &sLut );
```

Using a Custom LUT

Demonstrates how to force the libraries to destroy an internally created LUT and use a custom one.

```
tHandle hCamera = 0;
ui16 *pwCustomLut;
struct tLutInfo sLut;
ui32 dwCamDepth


PHX_CameraConfigLoad( &hCamera, NULL, PHX_BOARD1 | PHX_DIGITAL, NULL );

/* Inquire the current settings of an internal LUT */
sLut.dwLut = 0;
sLut.dwColour = 0;
sLut.dwTap = 0;
PHX_ParameterGet( hCamera, PHX_LUT_INFO, &sLut );

/* Create a custom LUT
 * Get the camera source depth (this dictates the size of the LUT)
 */
```

```
PHX_ParameterGet( hCamera, PHX_CAM_SRC_DEPTH, &dwCamDepth );

/* Each LUT entry is a 16 bit value (regardless of the camera depth) */
pwCustomLut = malloc( ( 1 << dwCamDepth ) * sizeof(ui16) );

/* Fill the above buffer with custom LUT data here */
…

/* Force the libraries to destroy the internal LUT and use the custom one */
sLut.pwLutData = pwCustomLut;
PHX_ParameterSet( hCamera, PHX_LUT_INFO | PHX_CACHE_FLUSH, &sLut );
```

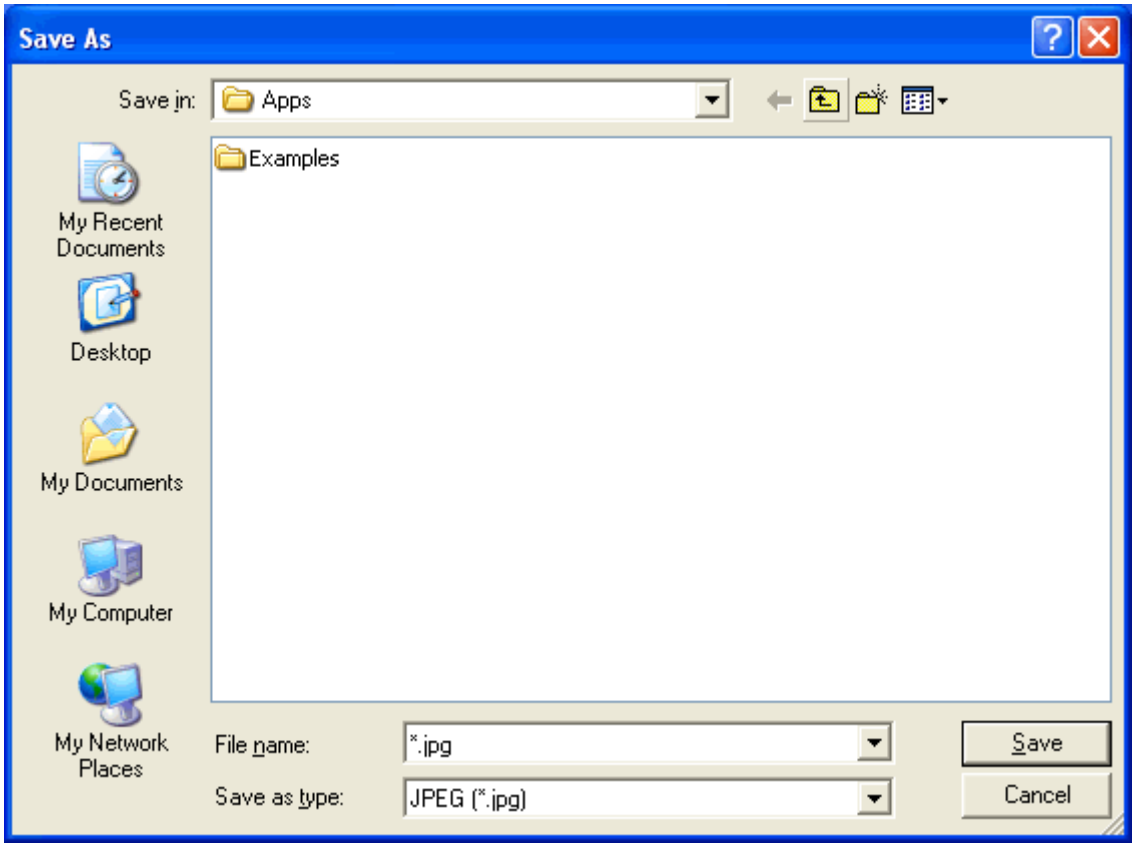
DISPLAYING IMAGES WITH GREATER THAN 8-BIT DATA

When displaying captured images, the image data buffer format needs to be compatible with the computer's graphics display setting. This is usually a 32-bit colour format (e.g. ***PHX_DST_FORMAT_XRGB32***). In such a colour format buffer, each displayed colour component (i.e. red, green or blue) can only be a maximum of eight bits.

If the captured image has a data depth of more than eight bits, then it is not possible to display all of this data using a 32-bit colour format data buffer. Hence it is important to note that only the top eight bits of each colour component will be visible.

SAVING IMAGES WITH GREATER THAN 8-BIT DATA

When using the Sequence tab on the Phoenix Control Class property sheet (under win32) to save a captured image buffer (or complete sequence), the following dialog box is displayed:



The "Save as type:" options are: JPEG (*.jpg); TIFF (*.tif); BMP (*.bmp) or RAW (*.raw). For each of these formats, the following bit depths are saved:

Format	Description
JPEG	Monochrome images are saved as 8-bit mono JPEG. Colour images are saved as 24-bit colour JPEG.
TIFF	Monochrome images of greater than 8-bit depth are saved as 16-bit mono TIFF. Monochrome images of 8-bits are saved as 8-bit mono TIFF. Colour images are saved as 24-bit colour TIFF.
BMP	Monochrome images are saved as 8-bit mono BMP. Colour images are saved as 24-bit colour BMP.
RAW	All images are saved in full bit-depth RAW format.

Hence, monochrome images with greater than 8-bit data can be saved in TIFF or RAW formats without losing any data. Colour images with greater than 8-bit data should be saved in RAW format in order to preserve all of the data.