

PXD 1000

Digital Frame Grabber

User's Guide

Version 1

Copyright © 1999, Imgenation Corporation. All rights reserved.
Imgenation Corporation
P.O. Box 276
Beaverton, OR 97075-0276

December 1999

P/N MN-1000-00

Table of Contents

| | |
|---|-----------|
| Chapter 1 | |
| Introduction | 7 |
| Technical Support | 7 |
| What is the PXD1000 | 8 |
| Chapter 2 | |
| Installation | 11 |
| What's in the Box | 12 |
| System Requirements | 12 |
| Basic Elements of an Image Capture System | 15 |
| Hardware Installation | 16 |
| Hardware Setup for Windows NT, 98 or 95 | 19 |
| Hardware Setup for DOS | 21 |
| Installing the PXD Software | 22 |
| Troubleshooting the Installation | 27 |
| Chapter 3 | |
| Using the PXD1000 | 29 |
| View Live Video Stream | 30 |
| Capture a Portion of an Image | 33 |
| Capture and Save Snapshots | 34 |
| Cascade or tile all open windows | 34 |
| Apply a Text Overlay to an Image | 35 |

Display the PXD Revision36

Chapter 4

Adding a Camera37

Creating a Camera Configuration File38

1. Starting PXDConfig38

2. Setting the Camera Information40

3. Image Geometry41

4. Setting the Exposure Timing42

5. Setting the Video Timing46

6. Setting the Advanced Video Timing48

7. Saving the Camera Configuration File50

Building a Cable51

The Raw Cable51

Cable Kits from Imagenation53

Making a Data Cable for 20-bit and Smaller Cameras54

Making a Data Cable for up to 32-bit Cameras54

The I/O Connector54

Chapter 5

Developing Applications for the PXD100059

Runtime Environments60

Development Environments61

32-bit Windows61

32-bit DOS62

The Development Environment62

Board Configuration62

Tools Organization64

ilib_32.lib64

Frame Grabber Library64

Frame Library64

Building a Simple Application65

Typical Program Flow65

The Sample Application65

| | |
|---|-----|
| Accessing the Imagination Libraries from C | 74 |
| Requesting Access to Frame Grabbers | 78 |
| Setting the Destination for Image Captures | 80 |
| Grabbing and Displaying Images | 81 |
| Advanced Topics | 89 |
| Initializing and Exiting Libraries | 89 |
| Sending Images Directly to Another PCI Device | 91 |
| Grabbing Images | 92 |
| Cropping Images | 94 |
| Timing the Execution of Functions | 94 |
| Using Flags with Function Calls | 99 |
| Digital I/O | 100 |
| Controlling the Input Lines | 101 |
| Controlling the Output Lines | 102 |
| Reading Frame Grabber Information | 103 |
| Accessing Captured Image Data | 105 |
| Frame and File Input/Output | 106 |
| Working with Camera Configurations | 109 |
| Setting the Camera Configuration | 111 |
| Video Timing | 113 |
| Line Scan Cameras | 117 |
| LUTs | 117 |
| Frame Status | 118 |
| Resynchronization and Frame Dropping | 118 |
| Camera Adjustment through RS-232 | 119 |
| Manual Buffer Locking | 119 |
| Scaling and Cropping | 120 |
| QualifyGrabs | 120 |
| Monitoring Grab Status | 122 |

Table of Contents

| | |
|----------------------------|------------|
| Chapter 6 | |
| PXD Library | 125 |
| Chapter 7 | |
| Frame Library | 273 |
| Appendix A | |
| Glossary | 335 |
| Index | 341 |

Chapter 1

Introduction

Thank you for purchasing an Imagination PXD1000 Digital Frame Grabber. The purpose of this chapter is to introduce you to the capabilities of the PXD1000 and the resources available to help make your project a success.

Technical Support

Before we start describing the capabilities of the PXD1000, a few words about technical support. We have attempted to make this manual the best technical resource for the PXD1000. Take a few minutes to read this chapter and browse the table of contents. We hope most of your questions about the capabilities and techniques for using them will be answered here. The PXD1000 User's Guide is the best, fastest path for answers your questions.

Beyond this guide, Imagenation technical support is available from 8 a.m. to 5 p.m. (Pacific Time zone) to answer your questions. If you have access to e-mail, mail your questions to:

support@imagenation.com

In our experience, e-mail provides the next best mechanism for support because it allows us to apply our all of our technical resources to help answer your questions.

You can also reach us by phone at:

(503) 495-2200 or (800) 366-9131.

What is the PXD1000

The PXD1000 is a digital frame grabber. It provides a 32-bit parallel connection between a digital camera and the system memory of an Intel X86-based computer through the computer's PCI bus. While several attempts have been made to standardize the both the logical (data format, timing and control) and physical (cables) interface to digital cameras, today there are almost as many connection schemes between a camera and frame grabber as there are cameras and frame grabbers. This means that one of the most important characteristics of the PXD1000 is its flexibility.

The PXD1000 can be configured to operate with many different types of digital cameras. While complex interactions between separate performance parameters may complicate the determination, generally, the PXD1000 can do the following:

- Accommodate cameras delivering up to 32 bit of parallel data. It will handle up to four separate channels of data, consistent with the 32-bit data path width. It will capture data at up to 40 MHz /channel and it will scan-line reorder image data from a variety of channel formats. The data can be sent using RS422 or EIA644 signaling standards.

- Synchronize to timing information provided by the camera or it can generate the timing information for the camera using an on-board clock synthesizer.
- Provide exposure control for the camera using a single exposure signal with either the length of a single pulse or a separate start and stop pulse defining the exposure period. The PXD1000 can also provide exposure control by sending timing information on two signal paths (see the strobe functions in the PXD library reference Chapter 6).
- Synchronize to external events using the trigger input. Two general purpose input and two general purpose outputs are also available for interfacing to external equipment.

You can capitalize on this broad range of capabilities by customizing the PXD1000 for your particular camera. A PXD1000 camera configuration file holds all the information necessary to program the PXD1000 to operate with a specific camera. We supply configuration files for a number of popular cameras, or you can create your own camera configuration files with **ConfigPXD**, armed with only the camera's technical manual. You won't need to know anything about the details of the PXD1000 to create a configuration file. The data structures created by **ConfigPXD** consists of a set of human-readable text entries that you can change in your application or a text editor if necessary. Once you've got a camera configuration file, you can use the **ViewPXD** program to capture and save images from your imaging system or you can use our development libraries to create your own application.

Since the cabling needs of each camera are different, we have used a very general 100-pin camera connector on the PXD1000. With this large number of pins, we can provide many of the often mutually exclusive signals needed by different cameras on one connector. On the one hand this means that no camera exactly matches the connector on the PXD1000, on the other, the flexibility provided by this diversity of signals allows you to build cables that will attach to almost any camera.

We supply standard cables for a number of high-volume digital cameras. If your camera is not one for which we supply an off-the-shelf cable, we also supply cable kits in which one half of the cable (the part that connects to the PXD1000) has been completed. The other half can be custom-wired to meet your needs.

Just as the interface details of digital cameras differ from model to model, the programming needs differ from project to project. We provide support for the most popular operating systems. These include Windows NT, 98 and 95. Support is also provided for DOS using the Watcom DOS4GW extender.

At this point, you are probably anxious to get started. The manual is organized in the following way:

- Chapter 2 describes the installation process.
- Chapter 3 is devoted to using the PXD1000 with the **ViewPXD** application.
- Chapter 4 describes adding a configuration file and a cable for a new camera.
- In Chapter 5 and beyond you will learn about creating software applications for the PXD1000.

This has been but the briefest of introductions to the PXD1000. As you explore the PXD1000 and incorporate it into your application, we welcome your feedback on making the PXD1000 more useful to you in the future. Please send your suggestions to support@imagination.com and thank you once again for purchasing your digital frame grabber from Imagenation.

Chapter 2

Installation

This chapter describes the process for installing the PXD1000 and setting up an image capture system. This minimal installation will work for the following cameras:

- Basler Line Scan and Area Scan cameras
- Dalsa CA-D4 and CA-D7
- Hitachi KPF100
- Kodak Megaplug ES1.0 and Megaplug 1.4i
- Pulnix TM1300, TM1001-02 and TM9701
- Reticon LD2040

If you are using a camera not listed above, some additional configuration will be required. Once you complete the hardware and software installation, turn to Chapter 4 for instructions on adding a camera.

What's in the Box

- The PXD1000 Digital Frame Grabber board
- I/O ribbon cable used to bring the I/O signals (triggers, strobes, etc.) from a connector on the top of the PXD1000 to the back panel of your computer
- Power adapter to connect your computer's power supply to the PXD1000. This is used to supply power to a camera or external equipment.
- Quick Installation Card
- User's Guide
- CD containing **ConfigPXD**, **ViewPXD**, drivers for Windows NT, 98 and 95 and software development libraries and tools for 32-bit Windows and DOS4GW.
- As there is a wide variety of cable connectors used by digital camera manufacturers, it is not possible to include a "standard" cable with the PXD1000. We do however, stock optional camera cables for a number of popular cameras. Call your Imagenation sales representative for a list of cables currently available.

System Requirements

- A computer with a Plug-and-Play BIOS, running a 32-bit version of Windows (Windows NT, 98 or 95).

Note We support DOS4GW as well, but all of our applications and sample code are designed for Windows.

Note The PXD1000 can not operate in a PCI computer that does not have a Plug-and-Play BIOS. Check the documentation for your

mother board to determine if the BIOS is Plug-and-Play, or press F1, F2, or DELETE (depending on your BIOS manufacturer) at boot time to display the setup screen. If your computer does not have a Plug-and-Play BIOS then you will need to get a BIOS upgrade from your computer manufacturer or a different motherboard.

- An unused PCI socket, capable of supporting a bus-mastering PCI card.

Note The PXD1000 is a bus-mastering PCI card. This means it takes control of the PCI bus to rapidly transfer images. In virtually all computers manufactured today, any PCI socket can host a bus-master card, but in some, especially those with more than four PCI sockets, only a few sockets can support bus mastering. This is particularly likely in computers based on a passive back-plane motherboard. You may need to consult your motherboard manual to determine which sockets can support a bus-master if you have a motherboard with more than four PCI sockets or are using a passive back-plane motherboard.

Note The board is 7.875 inches long, once fully connected, so the slot does not need to be full length.

- An unused interrupt (or one that can be shared) – The PCI Plug-and-Play BIOS will assign an interrupt to the PXD1000, unfortunately a frame grabber is seldom the first peripheral to request an interrupt in the computer.

Note Having too many ISA cards can use up all the available interrupts. Normally interrupts 3, 4, 5, 7, 9, 10, 11, 12, 14, 15 are under PCI plug-and-play control. In theory if these get used up, PCI cards can share interrupts. In practice, however, many PCI cards are not very friendly about sharing. It is the task of a PCI card driver to respond to interrupts; when the shared interrupt is not for that card, the driver is responsible for passing it on to the next card using the same interrupt.

Under normal circumstances, the PCI BIOS allocates interrupts by some reproducible scheme. If you change the cards around the interrupts may change. Some PCI BIOSs let you select the interrupt for each slot and that remains fixed. Others let you select the order in which they are allocated (for example, from the AGP slot outwards).

Some BIOSs let you reserve interrupts that are used only for ISA devices: The PCI PnP BIOS cannot detect ISA NoA PnP cards, so you have to tell it which interrupts not to use.

- An unused back-panel punch-out, either at an unused card-slot or a DB25 cutout on the back panel of your computer. This is used for the I/O connector that brings the trigger and strobe signals from the PXD1000 to the outside world.
- About 5 MB of free space on your hard drive for the software.
- CD ROM drive for installing the software.

Basic Elements of an Image Capture System

- **PXD1000 Digital Frame Grabber** the PXD1000 contains both a dedicated camera connector that carries all the signals necessary to interface with popular digital cameras and a separate I/O connector that provides easy access to the signals you will need to interface your image capture application to the outside world.
- **Digital camera** the PXD1000 comes with configuration files for the following cameras. If your camera is not listed, see Chapter 4 for instructions on adding a camera.
 - Dalsa CA-D4 and CA-D7
 - Hitachi KPF100
 - Kodak Megaplug ES1.0 and Megaplug 1.4i
 - Pulnix TM1300, TM1001-02 and TM9701
 - Reticon LD2040
- **Camera lens** this is usually an option from the camera manufacturers although most cameras use standard lens mounts that are compatible with lenses from many sources
- **Camera power supply** the PXD1000 supplies +12 volts (fused at 800 mA) which can be used to power some cameras. However, most cameras have more complex power requirements and may require an additional power supply (available from the camera manufacturer).
- **Camera cable to connect the camera to the PXD1000** cables for the above-mentioned cameras are available from Imagenation. If your camera is not listed, see Chapter 4 for instructions on building a cable.
- **PXD software** located on the CD that came with the PXD1000. The software should be installed on a computer running Windows NT, 98, or 95.

Hardware Installation

1. Turn off your computer.
2. Unplug the power supply from the AC power outlet.
3. Remove the cover to access the PCI bus sockets.
4. If you have an anti-static wrist-strap, put it on and attach the alligator-clip to piece of bare metal on the computer chassis. If you don't have an anti-static strap, ground yourself by touching a piece of bare metal on the case before each of the next six steps.
5. Locate an unused PCI socket on the motherboard. As mentioned above in "System Requirements," you might need to choose this socket carefully. The PXD1000 is a bus-mastering PCI card. This means it takes control of the PCI bus to rapidly transfer images. In most computers any PCI slot can host a bus-master card, but in some, especially those with more than four PCI sockets, only a few sockets can support bus mastering. This is particularly likely in computers based on a passive back-plane motherboard. You might want to consult your motherboard literature if you have a motherboard with more than four PCI sockets or are using a passive back-plane motherboard.
6. On the back panel of your computer, remove the cover plate for the PCI socket you have chosen.
7. Remove the PXD1000 from the anti-static bag.

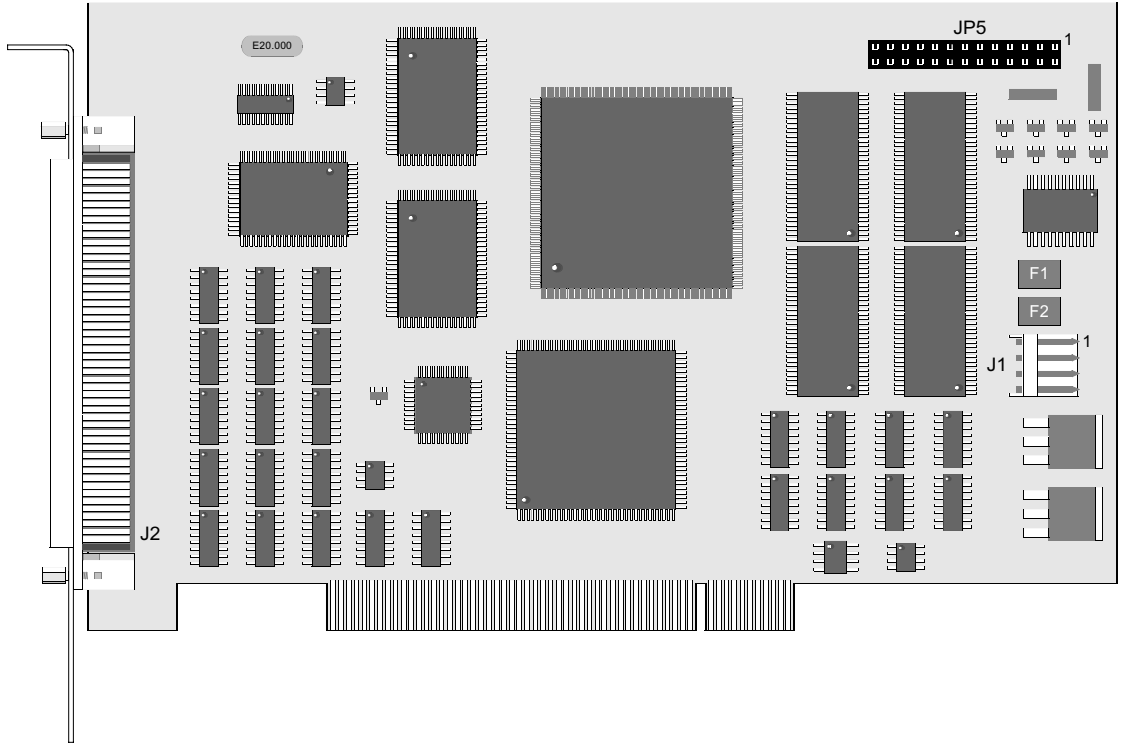


Figure 2.1

8. *Optional:* If you plan to use the PXD1000 for setting triggers, switches, etc., attach the I/O ribbon cable to the JP5 connector on the PXD1000 card (see Figure 2.1 above).

Note It is essential that this cable be attached correctly. The red side of the ribbon cable should face *away* from the back plane of the PXD1000 and should line up with pin #1 (see Figure 2.1).

9. Insert the PXD1000 card into the PCI slot. Make sure the card is seated completely in the socket.

10. Attach the back plane of the PXD1000 to the back panel of the computer, using the same method that was used to attach the original blank cover plate (e.g. use the same screw and screw the card to the back panel.)
11. *Optional:* If you are using the I/O ribbon cable, attach the loose end of the cable (the end with the back plane) to an unused card slot or to an unused parallel port punch out.
12. Attach a power connector from your PC's power supply to the J1 connector on the right side of the PXD1000 board (see Figure 2.1). The pin out of J1 is shown below. On the cable from your PC's power supply the +5 volt supply will typically be a red wire, +12 volt will be yellow and Ground will be black.

Note If necessary, use the power adapter cable that came with the PXD1000 to convert the standard hard disk power plug to the miniature (1/2") plug used on the PXD1000.

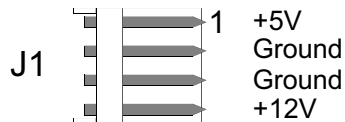


Figure 2.2

13. Replace the cover to your computer case.
14. Attach the camera cable to the PXD at the camera connector on the back plane.

This 100-pin connector is an AMP Amplimite .050 Series Right Angle Receptacle (AMP PN: 787170-9) or equivalent and contains all the camera data and timing signals as well as +12 volts fused at 800 ma which can be used for powering some digital cameras.

15. Choose a lens and install it on your camera.

16. Plug in your computer (it is amazing how many people forget this step).
17. Turn on the computer and proceed with the hardware setup for either Windows or DOS (see below).

Hardware Setup for Windows NT, 98 or 95

1. Turn on your computer. (for Windows NT, log on with Administrator's rights.) As soon as the power-on self-test completes, the Plug-and-Play BIOS takes control of your computer. It scans each PCI slot and builds a device table containing the Device and Vendor ID of each PCI card it finds (the manufacturer programs these numbers into every PCI card). The BIOS also assigns an interrupt and a range of memory addresses to each card. When Windows finally starts, it loads the device driver that has previously been associated with each entry in the device table. If an entry has no device driver associated then Windows will run the New Hardware Wizard.

Note If the New Hardware Wizard does not run, the most common cause is that your computer does not have a Plug-and-Play BIOS. The BIOS is responsible for assigning an interrupt and a memory portal to the frame grabber. If your computer lacks a Plug-and-Play BIOS, you will need to get an upgrade from your motherboard manufacturer.

Note Normally the primary purpose of the new hardware wizard is to install a device driver for the new hardware. In the case of the PXD1000 the main result is to simply assure Windows that the PXD1000 is indeed a valid PCI device.

2. The New Hardware Wizard will identify the PXD1000 as a new PCI Multimedia Device and bring up a screen like the one below:



3. Insert the Imagination CD into the CD-ROM drive and select the “Search for the best driver for your device” option.

4. Choose the CD-ROM drive as the place to search for the new driver from the screen shown below. Click **Next**.



Hardware Setup for DOS

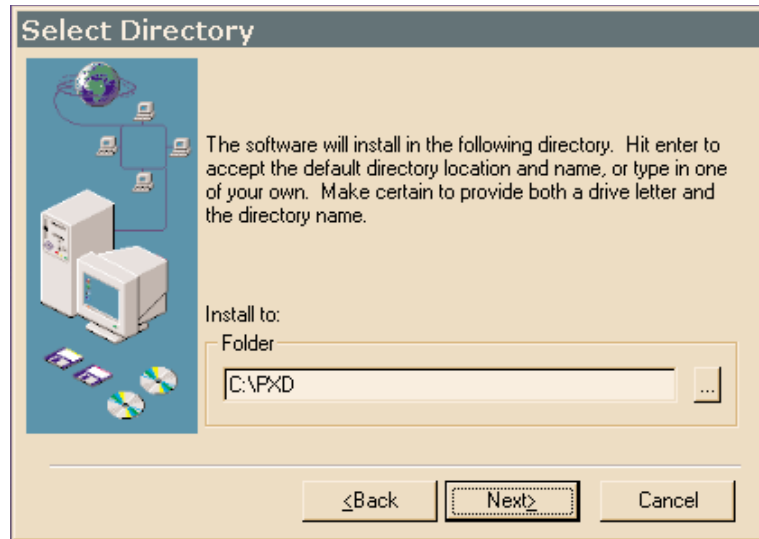
1. After installing the PXD1000 hardware reboot your computer and load the CD into the drive.
2. From the top level of the CD type **Setup** to begin the software installation and follow the on screen instructions to complete the installation.

Installing the PXD Software

1. Put the disk in your CD drive. Click the **Start Menu** and choose **Run**. Click **Browse** to choose **Setup** from the CD drive. The following window appears. Click **Next**.

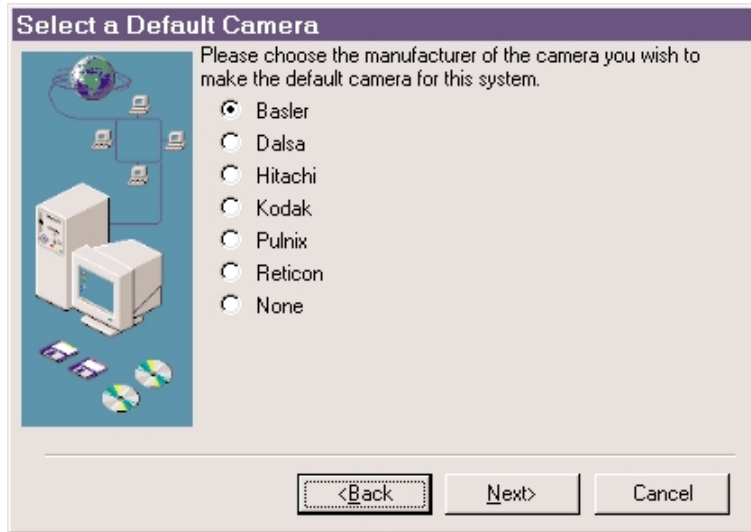


2. Choose a directory in which to install the PXD software, or click **Next** to accept the default directory.



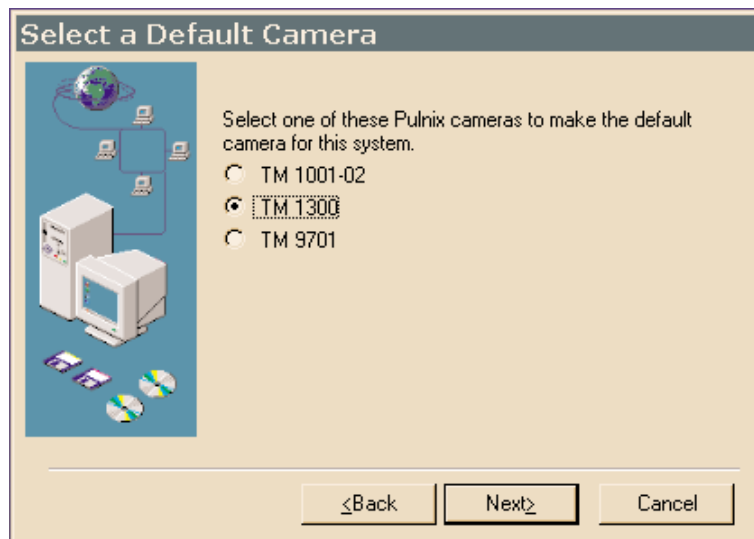
3. If you have one of the cameras on the list, select it. Otherwise select **Pulnix** for the default camera manufacturer (this will allow you to run non-camera-specific applications) and click **Next**.

Note If your camera is not on this list, see Chapter 4 *Adding a Camera* for instructions on configuring your camera.



4. Choose your camera **model** and click **Next**. At this stage the installer will start copying the files to your hard drive.

Note If you selected **Pulnix** in the previous step, select **TM1300** for the default camera and click **Next**.

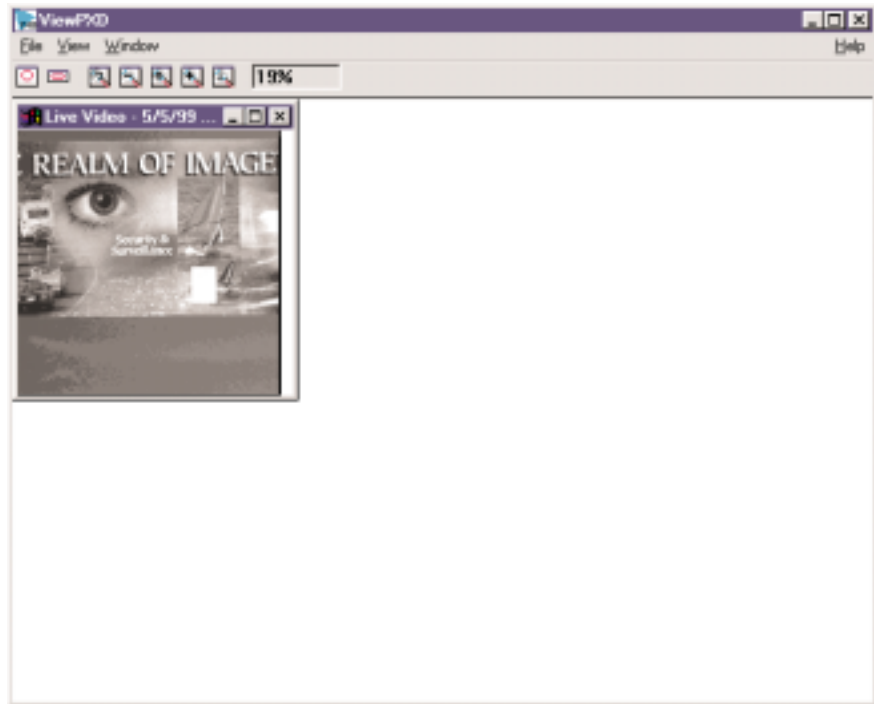


You will be given the option of putting the path to the Imagenation **BIN** directory (where the executables are stored) in the **PATH** statement of your **AUTOEXEC.BAT** file and adding an environment variable called **IMAGENATION** which tells the driver where to look for **PXD1000** firmware file and the default camera definition file.

If you are running Windows NT or have allowed Setup to modify the **AUTOEXEC.BAT** file you will be given the option to reboot your computer. You must reboot for the changes to take effect.

Upon completion the Setup application will have:

- Copied all of the Imagenation software to the directory you specified
 - Modified the registry to enable the driver to automatically load on boot (Windows NT only, there are no registry modifications for Windows 98 or 95.)
6. When the installation is complete, restart your computer.
 7. Go to the **PXD/bin** directory and double click on the **ViewPXD** application to start it.
 8. Go to the File menu. Select **PXD** and then **PXD1** on the cascaded menu. You should see video on your screen!



Troubleshooting the Installation

If you have trouble with your image capture system...

First check that all the components are in place.

- The board must be correctly installed in a PCI slot and an unused interrupt allocated to it by the PCI Plug-and-Play BIOS. Interrupts 3, 4, 5, 7, 9, 10, 11, 12, 14, 15 are under PCI plug-and-play control. Under normal circumstances, the PCI BIOS allocates an interrupt by some reproducible scheme. If you change the cards around the interrupt may change. If you install a new card the interrupt may change unless you change cards or muck with the BIOS, some PCI BIOSes let you select the interrupt for each slot and that remains fixed, others let you select the order in which they are allocated (from the AGP slot outwards).
- A driver must be installed – this can happen at boot time or be initiated by the application
- Firmware must be loaded onto the board – the driver does this when it loads.
- The runtime libraries need to be accessible – this means either in a place known to the operating system or in the same folder as the application.
- A camera definition file must be used to configure the board for a particular camera – this is the responsibility of the application. It can be a data structure included in a header file or dynamically loaded by the application using the LoadConfig and SetConfig functions

The firmware contained in `pxd1000.pxd` turns the board into a high-speed 32-bit digital frame grabber. The firmware is loaded when the driver for the board is loaded by the operating system. The firmware file can contain mul-

multiple copies of the firmware. As new firmware is released the new version is added onto the old so a driver can always load a particular version.

Once the board is installed and the driver and firmware loaded, the board is ready to run.

Problem: I ran the application for the first time and the computer locks up (no mouse control, nothing).

The most likely cause is that the PXD software has enabled the DMA engine on the PXD1000 to start data transfer but the PCI slot is not capable of giving the PXD1000 control of the BUS.

Solution: Make sure you have plugged the PXD1000 into a slot capable of bus mastering.

Verify that your machine meets the system requirements by contacting your PC vendor.

Problem: I started the application, but I'm getting scrambled or incomplete video data.

In this event, the PCI bus itself is not capable of handling the PXD1000's current data rate. This is typically caused by a poorly performing PCI bridge chip. The PXD1000 works with the Intel line of PCI bridges 430TX or better. Some earlier PCI products may cause problems.

Solution: Verify that the PCI bridge is set up correctly. You may need to adjust the interrupt latency timer.

You may need a new motherboard.

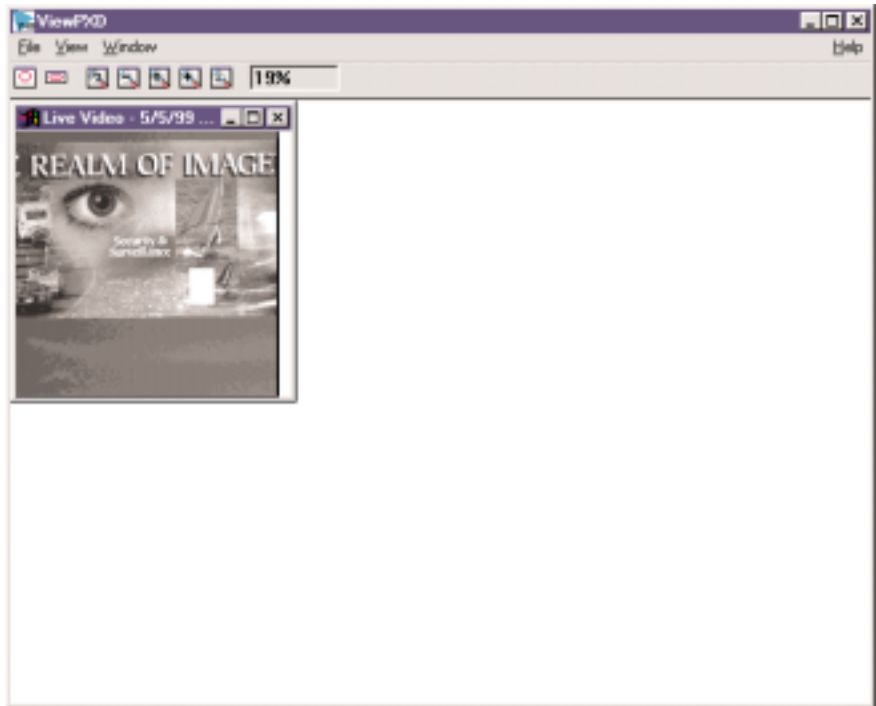
Chapter 3

Using the PXD1000

The PXD1000 comes with **ViewPXD**, a software application for Windows that allows you to view and store images from your camera. It is loaded onto your computer as part of the standard installation process.

View Live Video Stream

1. To start **ViewPXD**, click the **Start** button on the taskbar, choose **Programs**, then select **ViewPXD** from the **PXD** menu.
2. From the **File** menu, choose **PXD** and then **PXD 1**. If you have more than one PXD1000 board installed, you can choose which board to use (**PXD 1**, **PXD 2**, etc.). A window opens containing the live video stream.



- This button displays the entire image, maintaining its original aspect ratio.
- This button stretches the image to completely fill the window, which may change the aspect ratio. This effect is for viewing only and does not affect the image to be saved.

The other icons on the toolbar allow you to zoom in and out at different rates. You can also choose **Zoom** from the **View** menu to access these commands.

Note **ViewPXD** uses the default camera configuration defined in the installation procedure (**default.cam**). **ViewPXD** searches for this file in the following locations (in order of first to last searched):

- **Windows\System32** (Windows NT only)
- **Windows\System** (Windows 98/95 only)
- any folders listed in the **Imagination** environment variable
- any folders in the **PATH** variable
- in the current directory

If you would like to load a different camera configuration file, choose **Grabber Configuration** from the **View** menu. In the dialog box, click the browse button under **Configuration File** and select the new **.cam** file.

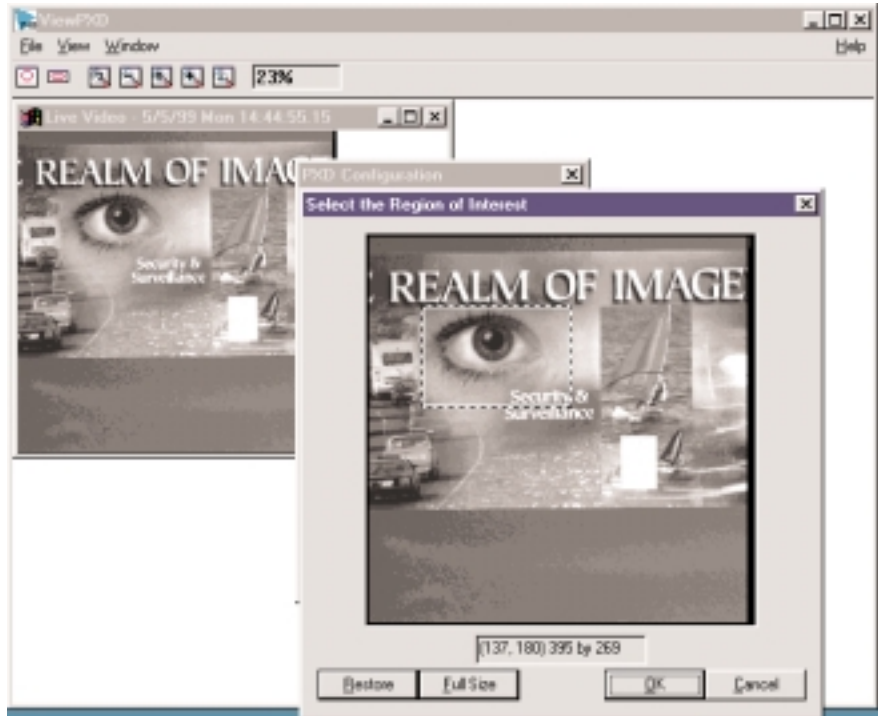
Note If you would like to load a Look Up Table (LUT), choose **Grabber Configuration** from the **View** menu and click the **Lookup Tables...** button.

A LUT description file contains a list of numbers, each on a separate line:

- The first line is the number of bits - 8 or 16.
- The second line is the starting entry number.
- The third line is the ending entry number.
- The remaining lines are the LUT values, starting with entry number **START**, through entry number **END**.

Capture a Portion of an Image

1. From the **View** menu, click **Grabber Configuration**, then click on the icon next to **Capture Region**.



2. Drag to select the region you want and click **OK**. In the example above, the **Live Video Window** would capture only the region of the image containing the eye.
3. To capture the image at full size, return to the **Capture Region** window and click the **Full Size** button.

Capture and Save Snapshots

1. Choose **Snapshot Live Image** from the **View** menu to capture a snapshot of the live video. The snapshot will appear in a new window. You can create as many snapshots as you like.
2. To save a snapshot to file, select the window containing the snapshot and from the **File** menu, choose **Save** or **Save As**. You can save the file in either **.BMP** (RGB) or **.PNG** (grayscale) format.

Note If you save the **Live Video Window**, a snapshot of the image will be saved to a file, rather than the live video stream itself.

Cascade or tile all open windows

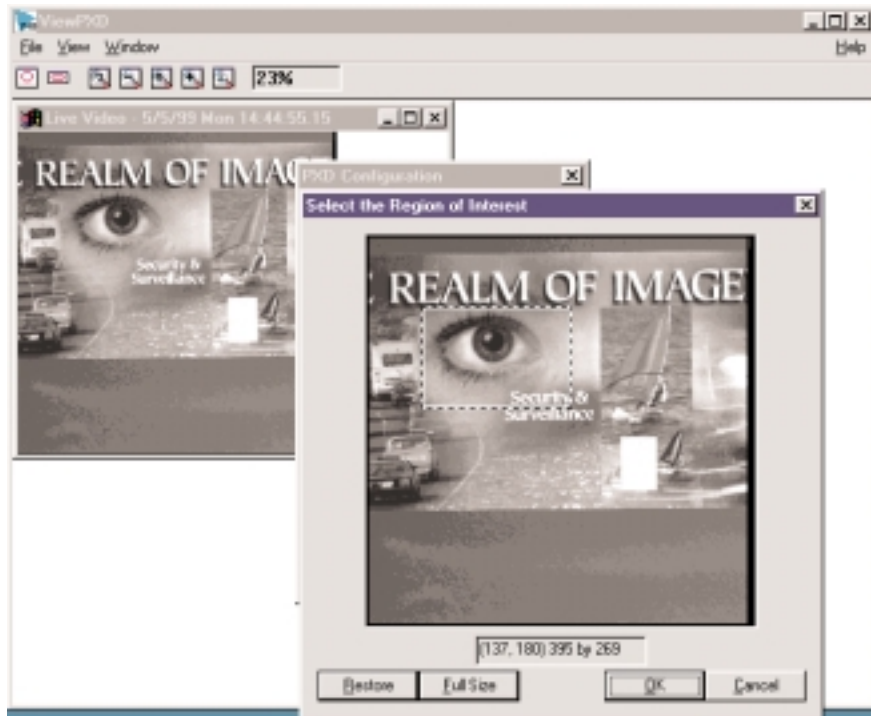
If you capture several snapshots, you can view them more easily by cascading or tiling the windows:

1. From the **Window** menu, choose **Tile Horizontally**, **Tile Vertically**, or **Cascade**.
2. To cycle through the images, use **CTRL F6** for the next window and **SHIFT F6** for previous window.

Apply a Text Overlay to an Image

You can add a text overlay to a snapshot.

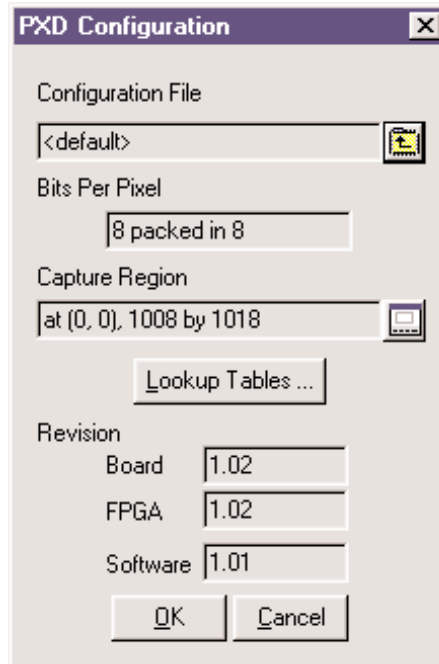
1. Select a snapshot window. (You cannot overlay text on the live video window).
2. From the **View** menu, select **Text Box Edit**.



3. Type your text in the edit box on the lower left. Choose the font, size, color, and background.
4. Drag the text overlay box on the image to move or resize it.
5. When the text overlay is the way you want it, click **OK**.

Display the PXD Revision

You can determine the revision numbers of your PXD1000 board or software by selecting **Grabber Configuration** from the **View** menu. The revision numbers are located at the bottom of the dialog box.



Adding a Camera

The PXD1000 provides direct support for the following cameras:

- Basler Line Scan and Area Scan cameras
- Dalsa CA-D4 and CA-D7
- Hitachi KPF100
- Kodak Megaplug ES1.0 and Megaplug 1.4i
- Pulnix TM1300, TM1001-02 and TM9701
- Reticon LD2040

Imagination provides Camera Guides for each of these cameras, which include general instructions for configuring the PXD1000 for the camera, as well as guidance for using the different modes on each camera. These Camera Guides are located on the CD that came with the PXD1000, as well as on our website: www.imagination.com.

If your camera is not listed above, you will need to create a camera definition file and you may need to build a cable to connect it to the PXD1000 (see *Building a Cable* later in this chapter for more information on cables).

The **PXDConfig** application (installed during the installation procedure described in Chapter 2) allows you to create a camera configuration file for any digital camera. You can also use it to customize settings for your camera, taking advantage of any advanced features, such as variable exposure and integration mode.

Creating a Camera Configuration File

The example below uses the Dalsa CA-D7 camera. Although this is one of the cameras already supported by the PXD1000, you can use the same procedure to create a configuration file for any other digital camera.

You will need your camera manual to determine the following information:

- Pixel bit depth of video output
- Pixel clock speed or data output rates per channel
- Active image resolution
- Cable pin-out for your camera cable

This information is necessary for the PXD1000 to work with the camera. The rest of the information requested in the camera configuration is not critical and can be experimented with to fine tune the settings.

1. Starting PXDConfig

PXDConfig is located under **PXD** in your **Start Menu**. Select it to start the application. When the application starts, it loads the **default.cam** configuration file. You can use the instructions that follow to modify this file or create a new file using **Save As**.

The screen contains three windows:

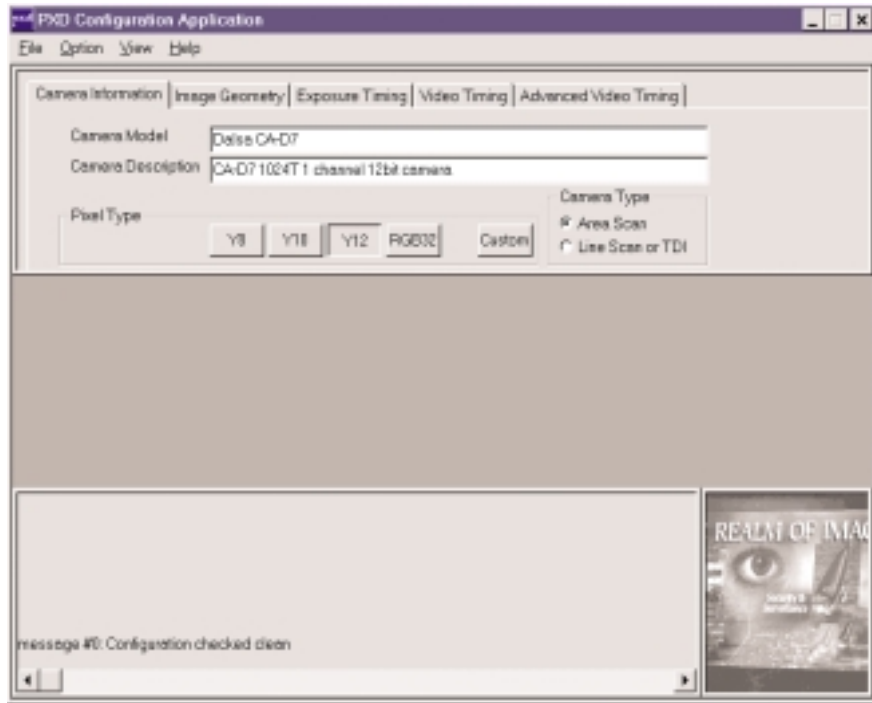
- Parameters window with five tabs for different configuration parameters. (This window extends downward to display or hide

advanced parameters. To display the advanced parameters, click the **Custom** button when available.)

- Video window in the lower right that displays the live video stream when configuration is correct. You can double-click this window to open a larger window for a more detailed view of the image. This larger window allows you to zoom in and out as well as scroll through the image.
- Message window that displays a log of status and error messages. This log can be cleared by choosing **Clear Message Queue** from the **Option** menu.

2. Setting the Camera Information

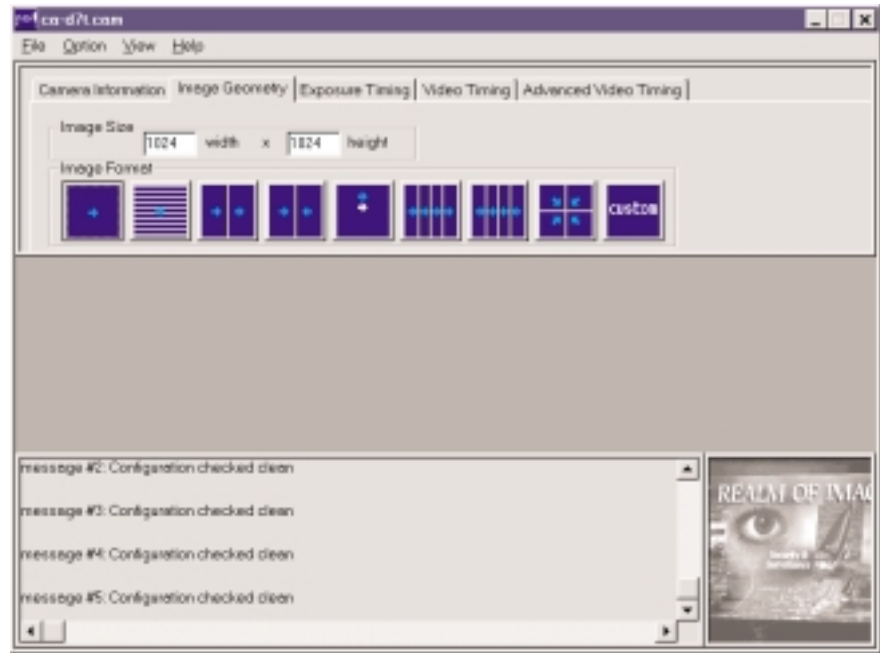
- a. Select the **Camera Information** tab and type in the **Camera Model** and **Camera Description** in the fields provided.



- b. Select the **Pixel Type** (also known as bit depth). In this case, **Y12** is selected for a 12-bit image.
- c. Select the **Camera Type**. This particular camera is an area scan camera.

3. Image Geometry

- a. Select the **Image Geometry** tab and enter the camera resolution under **Image Size**. In this case, the camera is 1024x1024.

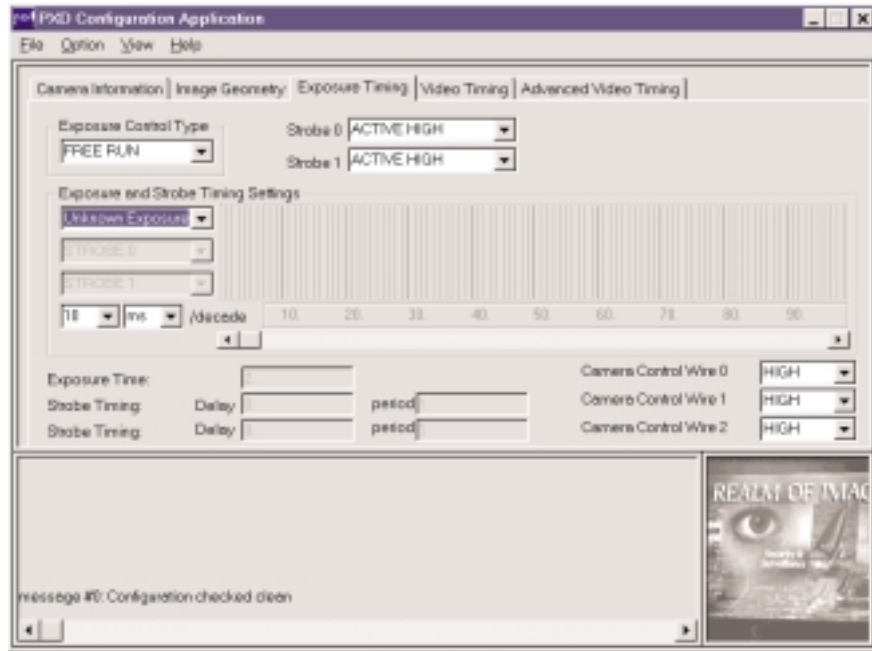


- b. Choose the **Image Format** appropriate for your camera.

- The first button is for single-channel, non-interlaced cameras.
- The second button is for single-channel, interlaced cameras.
- The next three are for two-channel cameras.
- The next three are for four channels.
- The **Custom** button allows you to define a new format.

4. Setting the Exposure Timing

- a. Select the **Exposure Timing** tab and select the **Exposure Control Type**. This camera is set to **Free Run** for continuous live video.



- **ASYNC** for cameras that generate frames that are occasionally controlled by something other than the frame grabber. For example, this setting can be used for a camera that generates a frame only when it is triggered by some external mechanism. (This is snapshot mode.) The frame grabber waits for that frame.
- **Free Run** for cameras that output continuous frames of video either based on internal timing or synced to the drive signals from the grabber. (This is continuous live video mode.)
- **Strobes** to set the frame grabber to generate strobe signals at the strobe0 and strobe1 pins. This can be used to program the expo-

sure or integration time, or it can be used as a general-purpose strobe signal. (See section 4.1 *Controlling the Exposure Time* below.) The strobes repeat to provide continuous video. Single shot, non-repetitive strobes can only be programmed through the API. (See Chapter 4, *Developing Applications for the PXD1000*.)

- b. Set **Strobe 0** and **Strobe 1**. This sets the polarity of the strobe output to HIGH true or LOW true.
- c. Set the **Camera Control0, 1, 2**. These are used as general purpose control pins (for example, to control the camera mode). Check your camera manual to determine if they are relevant to your camera.

The Dalsa CA-D7 has a binning feature, which causes the camera to output only half the resolution but at a faster frame rate. For this example, we turned off the binning feature by setting the pin connected to **Camera Control 0** to **High** and the pin connected to **Camera Control 1** to **Low**.

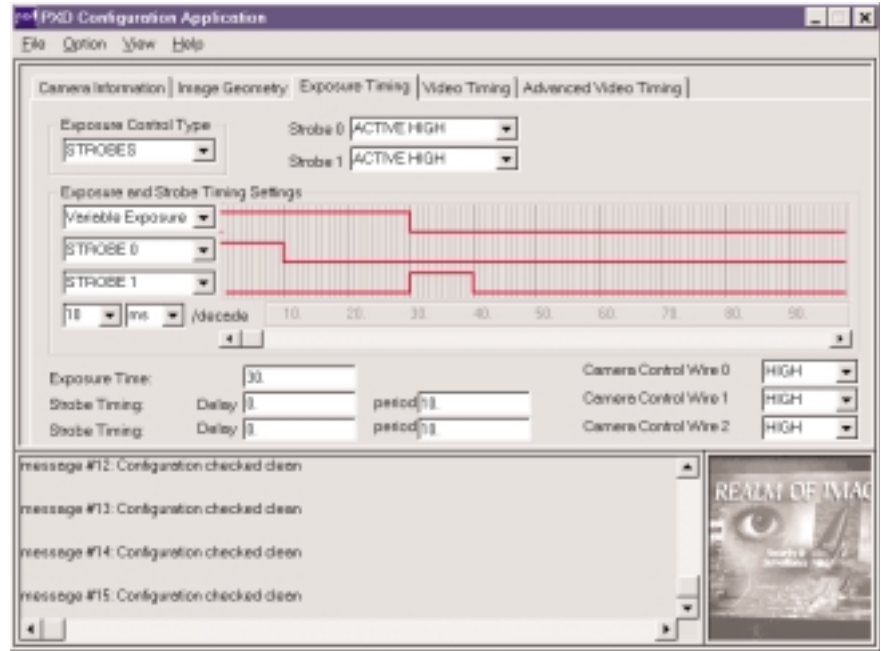
4.1 To Control the Exposure Time (or Integration Time)

- a. Set the **Exposure Control Type** to **Strobes**.
- b. Determine which pins on your camera control the exposure time and set **Strobe 0** and **Strobe 1** to mimic these pins.

For example, the Dalsa CA-D7 camera uses the EXSYNC and PRIN pins to control its exposure time (integration time). Both pins are active on LOW. Since the cable for this camera has EXSYNC connected to strobe #0 pin and PRIN connected to strobe #1 pin on the frame grabber, set **Strobe0** and **Strobe1** to **Active Low**.

- c. Select an exposure type under **Exposure** and **Strobe Timing Settings**.
 - **Fixed Exposure** means that the camera's exposure is not adjustable. You can still program the strobes, but the exposure time features are not available, and the **SetExposureTime()** API function will not adjust the strobe timing.
 - **Unknown Exposure** is used if the exposure time is unknown or not programmable. In this case, the strobes do not control the exposure and are used for general purposes only.
 - **Variable Exposure** allows you to set the exposure timing with a software application. (See Chapter 5, *Developing Applications with the PXD1000* for information on creating your own software.) The exposure time is either added to the strobe duration (if only 1 strobe is enabled) or to the time between the strobes (if 2 strobes are enabled).

- d. Use the **Strobe 0** and **Strobe 1** controls to mimic EXSYNC and PRIN as indicated on the timing diagram in the camera manual.



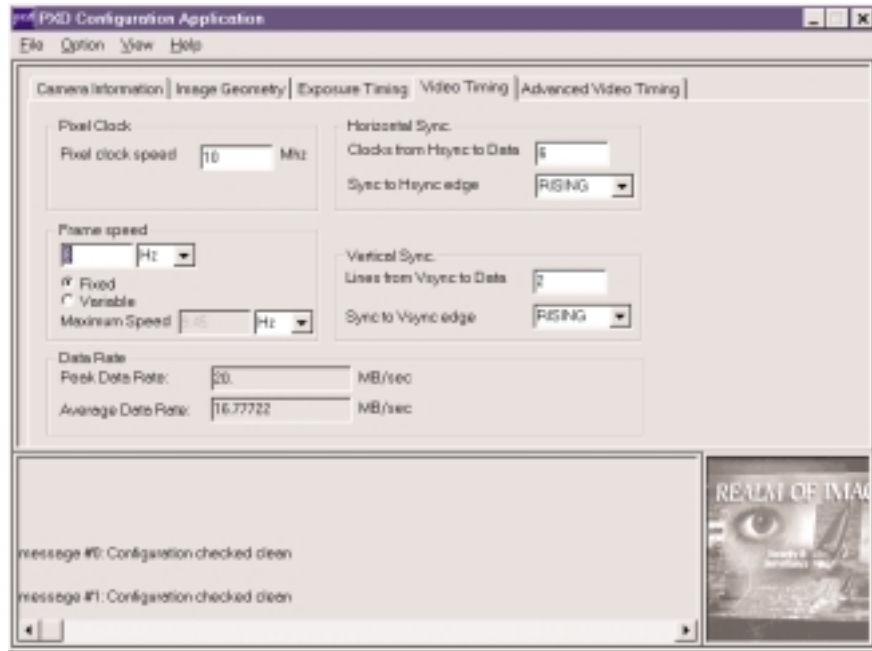
Adding a Camera

This timing diagram is a representation of what you would see on an oscilloscope connected to Strobe #0 and Strobe #1 pins.

Note You can produce two pulses on one strobe by enabling the same strobe for both waveforms.

5. Setting the Video Timing

- a. Select the **Video Timing** tab and enter the **Pixel clock speed**. For this example, the speed is set to 10Mhz.



The pixel clock speed is used to set the clock generator of the frame grabber. This setting must be correct in order for the camera to work properly.

- b. Set the **Frame Speed**. The Dalsa CA-D7 camera produces a frame speed of 8Hz at a fixed rate.

If the exposure time on your camera (which should be listed in your camera manual) is longer than the minimum frame period, use **Variable**.

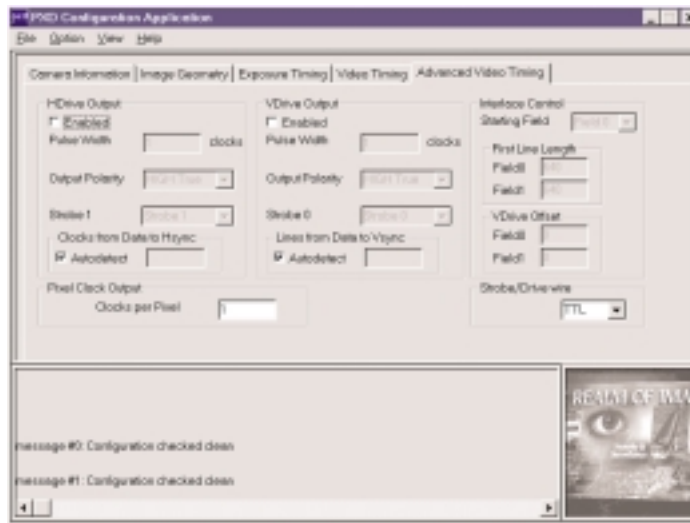
- c. Set the **Horizontal** and **Vertical Sync**. For this camera, there are about 6 clocks before valid pixels from the horizontal sync and about 2 lines before valid image lines from the vertical sync.

Note You will most likely find the horizontal and vertical sync information in your camera's manual or you can derive it from the timing diagram in the camera manual. You can also experiment with the numbers and see what works best.

6. Setting the Advanced Video Timing

Use this section only if your camera requires a horizontal or vertical drive, if it requires a clock from the frame grabber, or if it is an interlaced camera. Otherwise skip to section 7. *Saving the Camera Configuration File*.

- a. Select the **Advanced Video Timing** tab and enter the appropriate information.



- For cameras that use Horizontal Drive, the frame grabber generates an Hdrive pulse and the camera synchronizes to it. The camera then generates an LDV (Line Data Valid) signal and sends it back to the frame grabber. The frame grabber uses the LDV to detect the starting pixel of each line from the camera.
 - **Pulse Width:** in number of clocks
 - **Output polarity:** Low or High true

- **Strobe1 wire:** Output the HDrive signal to strobe1 and strobe1++1 wires in addition to the HDrive wire

Note Outputting the HDrive signal to the strobe1 wire will cause an error if strobe1 is enabled in the exposure timing window.

- **Clocks from Data to Hsync:** number of clocks from last pixel on the line to following horizontal sync.
- For cameras that use Vertical Drive, the frame grabber generates a Vdrive pulse and the camera synchronizes to it. The camera then generates an FDV (Frame Data Valid) signal and sends it back to the frame grabber. The frame grabber uses the FDV to detect the starting line of each frame from the camera.

- **Pulse Width:** in number of clocks
- **Output polarity:** Low or High true
- **Strobe0 wire:** Output the VDrive signal to strobe0 and strobe0++1 wires in addition to the VDrive wire

Note Outputting the VDrive signal to the strobe0 wire will cause an error if strobe1 is enabled in the exposure timing window.

- **Clocks from Data to Vsync:** number of lines from last line to following vertical sync.
- b. If your camera is an interlaced camera, enter the appropriate information under **Interlace Control**.
- **Starting Field:** either 0 or 1
 - **First Line Length:** Length of the first line on field 0 and field 1. Both fields are usually set to the width of the image size.
 - **Vdrive offset:** Number of pixel clocks after the edge of the Hdrive that Vdrive goes active.

- c. If your camera requires a clock from the PXD1000 set the **Clocks per Pixel**. Most cameras require one clock per pixel, but some require two. Refer to your camera manual for the correct setting.
- d. Set the **Strobe/Drive Signal**. This is the type of output signals generated by the frame grabber (either RS422 or TTL), for HDrive, VDrive, strobe0, and strobe1.

7. Saving the Camera Configuration File

- a. Once you have set all the necessary parameters, the video window should display a clear image. If it does not, check the message window for clues on what parameter to adjust.
- b. Once you are satisfied with the results, choose **Save** or **Save As** from the **File** menu. You can save the file either as a camera configuration file (.cam) or in C header format (.h).

Building a Cable

Imagination supplies cables for a number of popular digital cameras. For cameras where standard cables are not available, you can purchase an Imagination cable-kit and build one or make your own cable from scratch.

Consult your camera manual for the part numbers of the connectors used on the camera. These connectors are often available from the camera manufacturer. The mating cable plug for the PXD1000 is an AMP Amplimite .050 Series Cable Plug Connector, Series III (AMP PN: 749621-9) or equivalent.

The Raw Cable

Since most digital cameras send and receive differential signals, the raw cable should be built up of twisted-pairs of wires. If you choose to use a raw cable with wires that are not twisted into pairs for the differential signals, you will lose much of the noise immunity that the differential signaling schemes provide.

For example, suppose you plan to use a single channel 10-bit camera; the image data is transmitted on 10 twisted pairs (20 wires). If the camera is generating the timing for the frame grabber then you need at least Line-Data-Valid (LDV), Frame-Data-Valid (FDV) and the camera pixel clock (CAMERACLK). This means three more twisted pairs. If the camera uses a single exposure control implemented with EIA-644 (for example), and you want the frame grabber to control it, then another twisted pair is required. All in all, even this simple application requires at least 14 twisted pairs of wires.

Signal locations for the data connector (J2) are shown in Figure 2.

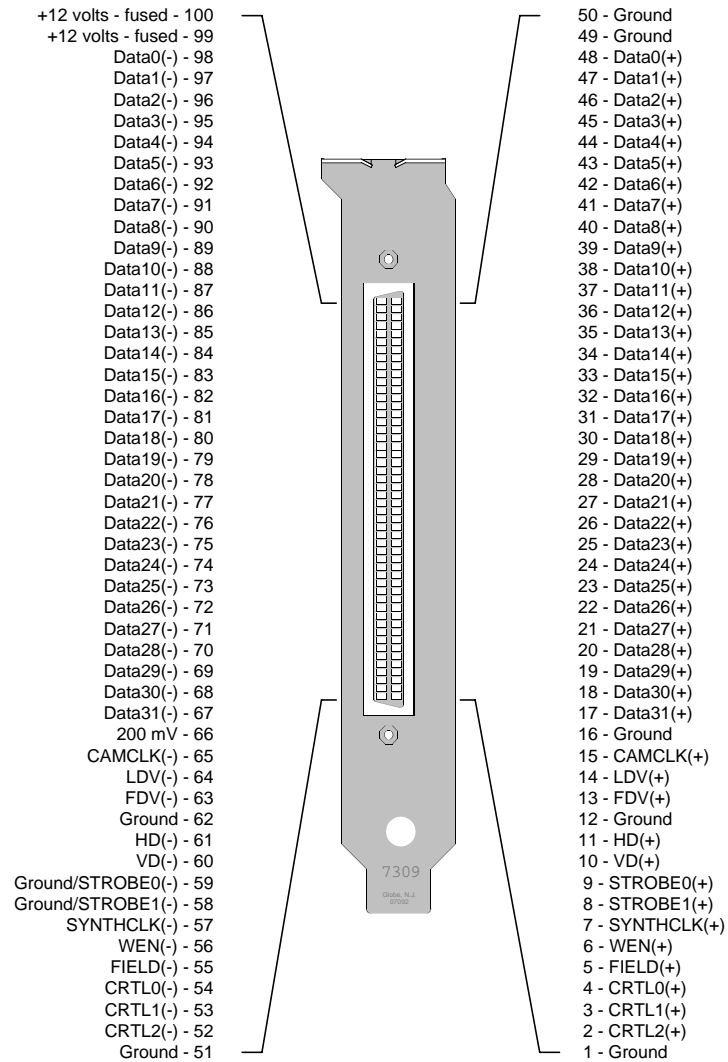


Figure 4.1 The data connector (J2)

In building a cable, there are two steps which are often overlooked that can reduce the difficulty and hasten the process.

1. Make a map of the cable connections, like the table below, and check off the connections as you make them.

| | Camera Pin Name | Camera Connector Pin | Wire Color | Frame Grabber Pin Name | Grabber Connector Pin |
|---|-----------------|----------------------|------------|------------------------|-----------------------|
| √ | AMSB(+) | 2 | green/pink | Data8(+) | 40 |
| √ | AMSB(-) | 36 | pink/green | Data8(-) | 90 |
| | AMSB1(+) | 3 | tan/green | Data9(+) | 39 |
| | AMSB1(-) | 37 | green/tan | Data9(-) | 89 |
| | | | | | |
| | | | | | |

2. Test the cable before using it. A multi-meter works well for testing end-to-end connections.

Poor soldering and incorrect connections can cause many unnecessary hours of debugging, which can be avoided by these simple steps.

Cable Kits from Imagenation

If you would like to make your own cable, two cable kits are available from Imagenation. Each contains a 3-meter cable with the PXD1000 plug (AMP Amplimite 749621-9) attached. The other end is uncommitted and contains stripped (0.5 cm.) and tinned wires. The two cable kits are identical except for the number of data wires in each cable.

- Cable Kit CB-012-00 37 twisted pairs
- Cable Kit CB-011-00 50 twisted pairs

If you are using a camera with 20 or fewer data bits, we recommend that you use Cable Kit CB-012-00. Since every unused data line must be termi-

nated to 200 mV (provided at J2), the smaller cable will have fewer wires that need to be terminated.

Making a Data Cable for 20-bit and Smaller Cameras

Cable kit CB-012-00 is appropriate for cameras with up to 20 data bits. This includes all single channel cameras and two channel cameras with up to 10 bits per channel.

The kit contains a cable with 37 twisted pairs of wires terminated on one end by a 100-pin connector which mates to the PXD1000. On the other end the wires have been prepared for attaching to a connector (which you must supply) that mates to your camera.

Making a Data Cable for up to 32-bit Cameras

Cable kit CB-011-00 is appropriate for cameras with up to 32 data bits.

The kit contains a cable with 50 twisted pairs of wires terminated on one end by a 100-pin connector which mates to the PXD1000. On the other end the wires have been prepared for attaching to a connector (which you must supply) that mates to your camera.

The I/O Connector

All other signals (except for camera data and timing) are available at the I/O connector. This includes the trigger, strobes and two general-purpose inputs and two outputs. Figure 3 shows the signals on the I/O connector.

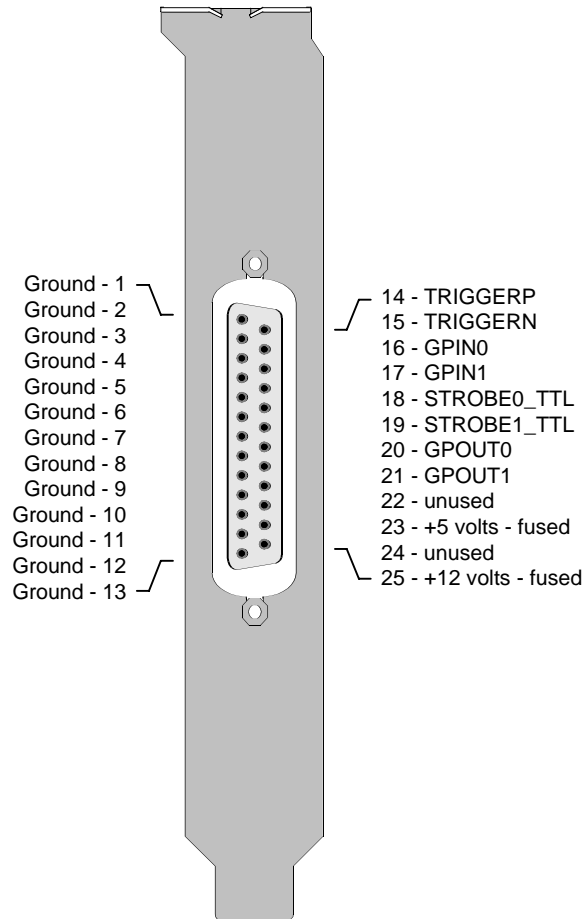


Figure 4.2 The I/O connector

Trigger Group Signals

The PXD1000 contains a single trigger capable of initiating a number of events, from generating an interrupt to capturing an image. A schematic of the trigger input circuitry is shown in Figure 4. The circuitry provides two trigger methods.

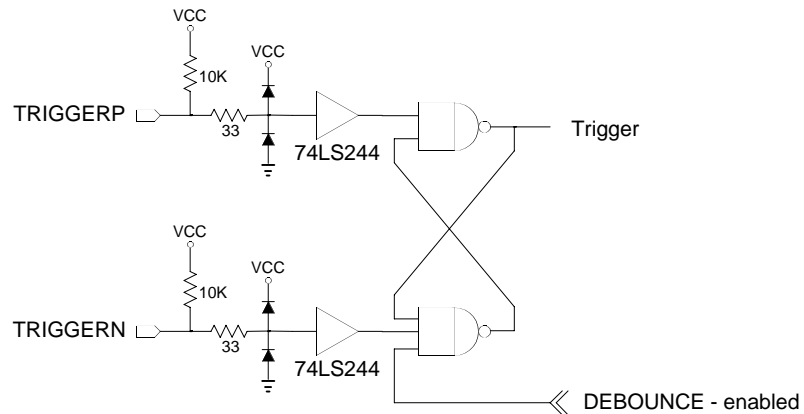


Figure 4.3 The PXD1000 trigger input circuitry.

Single-throw trigger The TRIGGERP input is a standard TTL level trigger. The signal applied to this input should be electrically clean since this input is not debounced. TRIGGERP has a 10K-ohm pull-up, is series terminated with a 33-ohm resistor and is diode protected. Use TRIGGERP with sources like optical encoders.

Double-throw triggers When TRIGGERN is enabled, by calling **SetTriggerSource** with the *DEBOUNCE* constant logically OR'd onto the *type* parameter, TRIGGERP and TRIGGERN become the set and clear inputs to an RS latch that can be used to cleanup noisy trigger signals. Use TRIGGERP and TRIGGERN in the *DEBOUNCE* mode with trigger sources like a double-throw mechanical switch.

General Purpose Input and Output Signals

The PXD1000 contains two general-purpose TTL inputs and two general purpose outputs. These bits are accessed through the **ReadIO** and **WriteImmediateIO** API functions. See Chapter 6, *PXD Library* for a description of these functions.

Each input is series terminated with a 33-ohm resistor and diode protected and drives a 74LS244 buffer. There are no pull-up resistors on the GPIN lines.

Each of the GPOUT lines is driven by a 74LS244 driver.

Strobe Signals

The PXD1000 has two strobe generators. Both strobe generators are available on the data connector (for use in exposure timing for the camera) and on the I/O connector. The only difference between the two strobe outputs on the data connector and on the I/O connector is that on the I/O connector TTL level signals are supplied instead of the RS422 signals on the data connector.

Typically the strobe generators are used to set camera exposure and therefore are not available for other uses.

Chapter 5

Developing Applications for the PXD1000

In this chapter we will describe how to develop custom applications for the PXD1000. We start with a description of the runtime and development environments. Next we talk about the development strategy and library organization. We then walk through a complete but simple application illustrating most of the basic concepts. We close the chapter by discussing several special topics on image capture.

We provide tools so that you can develop imaging applications that will run on Intel-based computers running Windows NT, Windows 98 and 95, as well as under the DOS 4GW extender from Watcom.

Since each development often has unique requirements for third party libraries, we provide specific libraries for use with Microsoft C, Borland C and Watcom development tools.

Runtime Environments

The PXD1000 libraries provide support for developing imaging applications that will run in Windows NT, 98 and 95 as well as 32-bit DOS using the Watcom DOS4GW Extender. Due to the large images typically produced by digital cameras and the awkward memory management issues, we do not directly support application development for the 16-bit DOS platform.

In order for your application to run correctly, drivers and dynamic linked libraries will need to be accessible to the application. The following tables describe the runtime environments for each supported operating system.

Windows NT

| | |
|--------------|---|
| PXD.SYS | in the Windows\System32\Drivers directory |
| PXD_32.DLL | in the Windows\System directory or in your PATH |
| FRAME_32.DLL | in the Windows\System directory or in your PATH |
| PXD1000.PXD | in the Windows\System directory or in your PATH |

For Windows NT, a registry entry is also necessary to tell Windows to load the kernel driver

Windows 98/95

| | |
|--------------|---|
| PXD.VXD | in the <code>Windows\System</code> directory |
| PXD_32.DLL | in the <code>Windows\System</code> directory or in your <code>PATH</code> |
| FRAME_32.DLL | in the <code>Windows\System</code> directory or in your <code>PATH</code> |
| PXD1000.PXD | in the <code>Windows\System</code> directory or in your <code>PATH</code> |

Development Environments

Most of the PXD1000 software tools are designed to run on a 32-bit Windows computer with an Intel-compatible processor. While development in a DOS environment is possible, you will gain greater advantage in dealing with camera configuration files if you have access to a computer running Windows.

You will need to insure that the following libraries and headers are in your compiler's library search path:

32-bit Windows

Microsoft Visual C++ 4.0 or Above

| | |
|--------------------------|--------------------------------|
| <code>pxd.h</code> | for managing the frame grabber |
| <code>iframe.h</code> | for managing image buffers |
| <code>ilib_32.lib</code> | for library management |

Borland C 5.0 or Above

| | |
|---|--|
| <code>pxd.h</code> | for managing the frame grabber |
| <code>iframe.h</code> | for managing image buffers |
| <code>video_32.h</code> and <code>video32b.lib</code> | for copying images to a graphics adapter |
| <code>ilib_32b.lib</code> | for library management |

32-bit DOS

Watcom C Compiler 10.6 or Above

| | |
|--|---|
| <code>pxd.h</code> and <code>pxd_fw.lib</code> | for managing the frame grabber |
| <code>iframe.h</code> and <code>iframe_fw.lib</code> | for managing image buffers |
| <code>video.h</code> and <code>video_fw.lib</code> | for creating user interfaces for DOS applications |

The Development Environment

Board Configuration

The following files are necessary for running the PXD1000

| | |
|--------------------------|---|
| <code>Pxd1000.pxd</code> | the firmware that controls the PXD1000 |
| <code>xxx.cfg</code> | a camera configuration file (if you plan to load camera configurations from a file) |

In order for the PXD1000 to work correctly, make sure these files are placed in the indicated locations

Windows NT

Pxd1000.pxd in the Windows\System32 directory
 or the Windows directory
 or any directories listed in the Imagenation
 environment variable
 or any directories in the PATH
 or current directory

xxx.cfg current directory
 or the full path specified in the filename

Window 98/95

Pxd1000.pxd in the Windows\System directory
 or the Windows directory
 or any directories listed in the Imagenation
 environment variable
 for any directories in the PATH
 or current directory

xxx.cfg current directory
 or the full path specified in the filename

Tools Organization

The Imagination libraries encapsulate all the functionality necessary to build imaging applications using the PXD1000. The functionality has been divided into four separate libraries

ilib_32.lib

This library is a small utility that manages the dynamic loading of the Imagination DLLs. It provides the **imagination_OpenLibrary** and **imagination_CloseLibrary** functions.

Frame Grabber Library

pxd_32.dll (for Windows applications) *or*

pxd_fw.lib (for applications created with the Watcom compiler for use with the DOS4GW extender)

This library contains functions that allow you to control the frame grabber hardware (set timing, camera modes, strobes and triggers) and initiate the capture of images.

Frame Library

frame_32.dll (for Windows applications) *or*

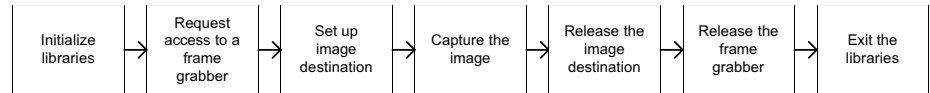
frame_fw.lib (for applications created with the Watcom compiler for use with the DOS4GW extender)

This library contains functions that manipulate images. It provides the ability to copy, load, save, and examine image data.

Building a Simple Application

Typical Program Flow

A basic program for capturing an image with the frame grabber contains the following basic tasks:



In addition, a program might include:

- Using the trigger signal to initiate a capture.
- Queuing functions so the program can do other work while the frame grabber is busy.
- Accessing the captured image data for analysis or processing.

The following sections describe these features in more detail and show you how to use the library functions to accomplish each of these tasks.

The Sample Application

This is a small but complete sample program which uses the PXD1000 to capture and display images under Windows. Several sections below refer to sections of this program as examples when discussing programming techniques.

Do not worry if some of this program is not immediately understandable; we will be going through it piece by piece in the following sections. It is presented complete here for reference.

This sample is in CAPTURE1.C.

```

/* -----
Capture1.c - a basic image capture and
display application using the PXD1000 for
Windows 98
Copyright 1999 Imagenation Corporation
-----*/

/* The displayed image size is hardwired -----*/
#define DISPWIDTH      640
#define DISPHEIGHT     480

#include <windows.h>
#include <time.h>
#include <stdio.h>
#include "pxd.h"
#include "frame.h"

#define FRAMERATECOUNTS 20

/* Function prototypes -----*/

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM) ;
static void ShowFrame(FRAME*, HWND) ;

/* Global variables -----*/
PXD      pxd ; // this struct is kind of
           // large so we don't want it
           // declared on the stack in
           // DOS. It is OK in Windows.
FRAMELIB  frameLib ; // this one is large too
long      hfg = 0 ;
FRAME*    pFrame = 0 ; // the structure that will
           // hold the image

/* WinMain -----*/
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE
hPrevInstance, PSTR szCmdLine, int iCmdShow)
{

```

```

MSG      msg ;
HWND     hwnd = 0 ;
WNDCLASS wndclass ;
short    sType;
TCHAR    szBuffer[64];
short    sFrameCt = 0;
time_t   tStart;
static  TCHAR szAppName[] = TEXT ("Capture1") ;

/* Initialize the Imagenation Libraries -----*/
if (!imagenation_OpenLibrary("pxd_32.dll", &pxd,
sizeof(pxd))) {
    MessageBox(NULL, TEXT ("Frame grabber library not
loaded"), szAppName, MB_ICONERROR) ;
    return 0 ;
}
if (!imagenation_OpenLibrary("frame_32.dll", &frameLib,
sizeof(FRAMELIB))) {
    MessageBox(NULL, TEXT ("Frame library not loaded"),
szAppName, MB_ICONERROR) ;
    return 0 ;
}

/* Request access to a frame grabber */
if (!(hfg = pxd.AllocateFG(-1))) { // the -1 means take the
first frame grabber found
    MessageBox (NULL, TEXT ("pxd frame grabber not found"),
szAppName, MB_ICONERROR) ;
    return 0 ;
}

```

```

    /* Set up image destination - Create a buffer in RAM to
    hold the image that is the same width, height, and frame type
    as the grabber is configured. */
    if (!(pFrame = pxd.AllocateBuffer (pxd.GetWidth(hfg),
pxd.GetHeight(hfg),
pxd.GetPixelType(hfg)))) {
        MessageBox (NULL, TEXT ("Unable to create image buffer"),
szAppName, MB_ICONERROR) ;
        return 0 ;
    }

    /* This next section is not grabber specific - it is just
    to get a windows app */
    /* Create and register a window class -----*/
    wndclass.style = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfWndProc      = WndProc ;
    wndclass.cbClsExtra     = 0 ;
    wndclass.cbWndExtra     = 0 ;
    wndclass.hInstance     = hInstance ;
    wndclass.hIcon          = LoadIcon (NULL,
IDI_APPLICATION) ;
    wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject
(BLACK_BRUSH) ;
    wndclass.lpszMenuName  = NULL ;
    wndclass.lpszClassName = szAppName ;

    if (!RegisterClass (&wndclass)) {
        MessageBox (NULL, TEXT ("Failed to register window
class"), szAppName, MB_ICONERROR) ;
        return 0 ;
    }

    hwnd = CreateWindow (szAppName, TEXT ("Capture1"),
WS_OVERLAPPEDWINDOW,
                        0, 0, DISPWIDTH, DISPHEIGHT,
                        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

    /* we want to grab the image at regular intervals, with
    little intrusion of the normal windows processing. To keep the

```

design simple, we will just modify the typical message processing loop. Instead of using the GetMessage function, we will use PeekMessage, with PM_NOREMOVE. This lets us check if Windows is busy. If it is, we will see that there is a message, and we will process it instead of trying a grab. When we are able to do the grab, we will be busy for 1 full frame period. No messages will be processed during this time, since the grab is not queued.

See QGrab.c for an example of grabbing with queued grabs, which will not hold up Windows processing.

```

*/
tStart = clock();
do
{
    if (PeekMessage (&msg, NULL, 0, 0, PM_NOREMOVE)) {
        if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE)) {
            if (msg.message != WM_QUIT) {
                TranslateMessage (&msg) ;
                DispatchMessage (&msg) ;
            }
        }
    }
    else {
        /* capture the image */
        if (pxd.Grab(hfg, pFrame, 0)) {
            ShowFrame (pFrame, hwnd) ;

            /* display the frame rate, so that it can be
            compared to QGrab.C */
            sFrameCt += 1;
            if (sFrameCt >= FRAMERATECOUNTS)
            {
                sprintf(szBuffer, "%s - %d fps",
                szAppName, FRAMERATECOUNTS * CLOCKS_PER_SEC/(clock() -
                tStart));

                SetWindowText(hwnd, szBuffer);
                sFrameCt = 0;
                tStart = clock();
            }
        }
    }
} while (msg.message != WM_QUIT) ;

/* Release the image destination */

```

```

frameLib.FreeFrame(pFrame) ;
pFrame = NULL ;

/* Release the frame grabber. */
pxd.FreeFG(hfg) ;
hfg = 0 ;

/* Exit the libraries */
imagenation_CloseLibrary (&frameLib) ;
imagenation_CloseLibrary (&pxd) ;

return 1 ;
}

/* WndProc -----*/
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM
wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_CLOSE:
            DestroyWindow (hwnd) ;
            return 0 ; ;
        case WM_DESTROY:
            PostQuitMessage(0) ;
            return 0 ;
    }

    return DefWindowProc (hwnd, message, wParam, lParam) ;
}

/* CreateWindowPalette -----*/
HPALETTE CreateWindowPalette(
    RGBQUAD
    lpPalette[256])
{
    /* from the example "Using DirectDraw Palettes in Windowed
Mode" from the Windows SDK */
    typedef struct {
        WORD        palVersion;
        WORD        palNumEntries;
        PALETTEENTRY palPalEntry[256];
    } XLOGPALETTE;

    DWORD                    dwCounter = 0;

```

```

XLOGPALETTE                Palette;

Palette.palVersion = 0x300;
Palette.palNumEntries = 256;
for (dwCounter = 0; dwCounter < 256; dwCounter++) {
    if (dwCounter < 10 || dwCounter > 245) {

        Palette.palPalEntry[dwCounter].peFlags = PC_EXPLICIT;
        Palette.palPalEntry[dwCounter].peRed = (unsigned
char)dwCounter;
        Palette.palPalEntry[dwCounter].peGreen = 0;
        Palette.palPalEntry[dwCounter].peBlue = 0;
    } else {
        Palette.palPalEntry[dwCounter].peFlags =
PC_NOCOLLAPSE;
        Palette.palPalEntry[dwCounter].peRed =
lpPalette[dwCounter].rgbRed;
        Palette.palPalEntry[dwCounter].peGreen =
lpPalette[dwCounter].rgbGreen;
        Palette.palPalEntry[dwCounter].peBlue =
lpPalette[dwCounter].rgbBlue;
    }
}
return CreatePalette((LOGPALETTE *)&Palette);
}

/* ShowFrame -----*/
static void ShowFrame(FRAME *pFrame, HWND hwnd)
{
    typedef struct {
        BITMAPINFOHEADER    bmiHeader;
        RGBQUAD             bmiColors[256];
    } XBITMAPINFO;
    XBITMAPINFO    bmi;
    BYTE           *pbFrameBuffer = NULL ;
    HDC             hdc ;
    RECT           rect ;
    BYTE           * pbCompressed = NULL;
    BYTE           * pbBuffer;
    HPALETTE       hPalette;
    DWORD          dwCounter;

    hdc = GetDC (hwnd) ;

    /*

```

We are assuming that the digital camera provides monochrome images, packed as either 8 bpp or 16 bpp. We will always display the image as 8 bpp. If 16bpp, we will convert it to an 8 bpp image, by pulling the high byte into another buffer.

If the buffer is other than PBITS_Y8 or PBITS_Y16, the developer will need to implement a different display method.

```

    /*
    memset (&bmi, 0, sizeof (BITMAPINFO)) ;
    bmi.bmiHeader.biSize          = sizeof (BITMAPINFOHEADER)
;
    bmi.bmiHeader.biWidth        = frameLib.FrameWidth(pFrame);
    bmi.bmiHeader.biHeight       = -frameLib.FrameHeight(pFrame)
; // negative paints from top down
    bmi.bmiHeader.biPlanes      = 1 ;
    bmi.bmiHeader.biBitCount     = 8; // force to 8, since DIB's
are 8bpp or RGB24
    bmi.bmiHeader.biCompression = BI_RGB ;
    bmi.bmiHeader.biSizeImage    = frameLib.FrameWidth(pFrame) *
frameLib.FrameHeight(pFrame) ;
    bmi.bmiHeader.biXPelsPerMeter = 0 ;
    bmi.bmiHeader.biYPelsPerMeter = 0 ;
    bmi.bmiHeader.biClrUsed      = 256 ;
    bmi.bmiHeader.biClrImportant = 256 ;

    /*
    Create a palette for the bitmap, and Windows, so that
the image will be displayed with the correct colors.
    */
    for (dwCounter = 0; dwCounter < 256; dwCounter++) {
        bmi.bmiColors[dwCounter].rgbRed = (BYTE)dwCounter;
        bmi.bmiColors[dwCounter].rgbGreen = (BYTE)dwCounter;
        bmi.bmiColors[dwCounter].rgbBlue = (BYTE)dwCounter;
        bmi.bmiColors[dwCounter].rgbReserved = 0;
    }
    if (GetDeviceCaps(hdc, BITSPIXEL) == 8)
    {
        hPalette = CreateWindowPalette(bmi.bmiColors);
        SelectPalette(hdc, hPalette, FALSE);
        RealizePalette(hdc);
    }

    SetStretchBltMode (hdc, COLORONCOLOR) ;
    GetClientRect (hwnd, &rect) ;

```

```

pbFrameBuffer = frameLib.FrameBuffer(pFrame) ;

/*
   If we have a 16 bpp image, convert to 8 bpp by pulling
the high byte into its own buffer.  The bmi.bmiHeader was
already set the the number of bytes in the image.
*/
if (frameLib.FrameType(pFrame) != PBITS_Y8) {
    pbFrameBuffer++; /* skip to the high byte */
    pbCompressed = GlobalAlloc(GPTR,
bmi.bmiHeader.biSizeImage);
    for (dwCounter = 0; dwCounter <
bmi.bmiHeader.biSizeImage; dwCounter++) {
        pbCompressed[dwCounter] = *pbFrameBuffer;
        pbFrameBuffer++;
        pbFrameBuffer++;
    }
    pbBuffer = pbCompressed;
} else {
    pbBuffer = pbFrameBuffer;
}

/*
   We will use StretchDIBits to display the image. It will
stretch the image to fill the window
*/
StretchDIBits (hdc, 0, 0, rect.right, rect.bottom, 0, 0,
                frameLib.FrameWidth(pFrame),
frameLib.FrameHeight(pFrame),
                pbBuffer, (BITMAPINFO *) &bmi, DIB_RGB_COLORS,
SRCCOPY) ;

/* pbCompressed is only a non-NULL value if we allocated it,
since we initially declared it to be NULL
*/
if (pbCompressed != NULL) {
    GlobalFree(pbCompressed);
}

ReleaseDC(hwnd, hdc) ;
if (hPalette) {
    DeleteObject(hPalette);
}
}

```

Customizing the Application

This sample can be used as a template for writing many useful programs. The sections below will suggest places to change this program to add various features. When you first begin to recompile and modify the sample, notice two macros near the top.

```
#define WINDOWWIDTH      640
#define WINDOWHEIGHT     480
```

These are used to define the default window height. You may want this window to be a window in a dialog box, or a window that you create of a different size.

You will also notice that very little message processing is performed in the Windows procedure. Most modifications to the code will occur inside of the Windows procedure, which we have called `WndProc`.

Accessing the Imagination Libraries from C

Using the Imagination libraries from C requires four steps:

1. include the necessary header files
2. declare a structure to represent the library
3. open the library before using it
4. close the library before exiting the program

Header Files

Each library comes with a header file which declares the various functions and symbols the library uses. Each C source file which uses the library must include the header for that library. You may also need to include other header files to use Windows functions or the C standard library.

Our sample calls the PXD frame grabber library, and the Frame library. It also uses Windows system calls to display the window and communicate with the user, the time system calls to measure the frame rate, and standard I/O to provide the text formatting function, **sprintf**. This means we need five header files:

```
#include <windows.h>
#include <time.h>
#include <stdio.h>
#include "pxd.h"
#include "frame.h"
```

Library Structures

Each library which you open must have a library structure declared in your program. This structure holds various information about the library, including a pointer to each function provided by the library. For most programs, declaring these structures as global variables like we have done in the sample works well. If you are writing a large program, and wish to isolate the parts which use the frame grabber, it is also possible to make these structures static within a single file, or even in a single function. Making them stack variables local to a function is generally not a good idea in DOS, however, because they are fairly large. This practice is rarely a problem under Windows, since the stack can be virtualized.

Our sample program needs a PXD structure to use the PXD library, and a FRAMELIB structure to use the Frame library.

We have decided to name our PXD structure “pxd” and our FRAMELIB structure “framelib”. You can change these names if you want to.

```
PXD      pxd ;
FRAMELIB frameLib ;
```

Opening the Libraries

Before the libraries can be used, they must be opened. This is done using the **imagination_OpenLibrary** function under Windows.

Opening a library finds and starts the DLL, and fills in the library structure with various information including pointers to the library's functions. The PXD library will also verify that there is at least one PXD1000 frame grabber installed in the computer, and that the VxD or Kernel Driver is correctly installed. If you attempt to call a library function before opening the library that contains it, your program will probably crash.

Opening the Imagination libraries explicitly gives you several useful abilities.

- You can choose what to do if the library isn't available. Normally, Windows will stop a program from running if any of its libraries aren't available. By explicitly opening the libraries, you can write a program which uses the frame grabber library if it is available, or runs without it and gives the user more limited capabilities if it is not.
- You don't have to worry about symbol name conflicts. Suppose that you are using another library which also happens to have a function named Grab. Because the PXD library's Grab function is always called as a member of the PXD structure, you can use both Grab functions without confusing the C compiler.
- You can write programs which use more than one model of frame grabber. For example, if you want to make a system which uses an Imagination PXD1000 for high resolution image analysis, and a PXC200 for color image previewing, you can declare both a PXD structure and a PXC structure in the same program, and these library structures keep the functions related to each model organized.

Our sample program doesn't use any of these abilities, however. It simply tries to open both the Frame library and the PXD library, and quits with an error message if there is a problem.

```

    /* Open the Imagination Libraries -----*/
    if (!imagination_OpenLibrary("pxd_32.dll", &pxd,
sizeof(pxd))) {
        MessageBox(NULL, TEXT ("Frame grabber library not
loaded"), szAppName, MB_ICONERROR) ;
        return 0 ;
    }
    if (!imagination_OpenLibrary("frame_32.dll", &frameLib,
sizeof(FRAMELIB))) {
        MessageBox(NULL, TEXT ("Frame library not loaded"),
szAppName, MB_ICONERROR) ;
        return 0 ;
    }

```

Closing the Libraries

When you are done using the libraries, they should be closed by calling **imagination_CloseLibrary**. After a library is closed, no more functions should be called from that library. This means that you should be sure to do any other cleaning up which involves library functions before you close the libraries.

In our sample, we close the libraries right before exiting the program. Notice that we do this after calling the **FreeFrame** and **FreeFG** library functions to free other things we've allocated.

```

imagination_CloseLibrary (&frameLib) ;
imagination_CloseLibrary (&pxd) ;

```

Requesting Access to Frame Grabbers

A process must have a handle to a frame grabber to communicate with it. The **AllocateFG()** function returns a handle to the specified frame grabber if it exists and hasn't already been allocated to another process.

Frame grabber handles are specific to the process that allocated them. Don't share a handle between processes; trying to do so will cause unpredictable behavior.

If you're using multiple frame grabbers in a single system, you'll need to determine which frame grabber is which. Due to the design of the PCI bus, bus slot *zero* doesn't necessarily correspond to frame grabber *zero*, and the number of the frame grabber in a particular bus slot can vary between different operating systems. You can determine which frame grabber is which by connecting a video source to only one frame grabber and then using the **ViewPXD** program (under Windows), or your own program, to switch between frame grabbers. If you are only using one frame grabber, you can just use `-1` as the frame grabber number.

When you are finished with a frame grabber, call **FreeFG()**, which frees the specified frame grabber, so it can be allocated by other processes.

```

    if (!(hfg = pxd.AllocateFG(-1))) { // the -1 means take the
first frame grabber found
        MessageBox (NULL, TEXT ("pxd frame grabber not found"),
szAppName, MB_ICONERROR) ;
        return 0 ;
    }

```

Getting Image Information

Once the frame grabber has been programmed to use the correct camera, you can collect some information about the sort of images the camera provides. The three most important pieces of information about the image are the width, height, and pixel data type.

The maximum width and height of image that the frame grabber can supply can be found by calling **GetXResolution** for the width, and **GetYResolution** for the height. The image dimensions are given in pixels, and will typically be anything from a few hundred pixels to several thousand depending on the resolution of the camera. The PXD1000 frame grabber also supports line scan cameras, which have an image height of 1.

The pixel data type is a value which describes the size and format in memory of each pixel in the image. Symbolic names for several pixel data types are provided in *Frame.b*. This table shows those most commonly used.

| Pixel Data | Type Description |
|-------------|--|
| PBITS_Y8 | 8-bit grayscale. |
| PBITS_Y16 | 16-bit grayscale. This type is used for cameras with 10 and 12 bits as well. |
| PBITS_RGB15 | 5 bits each for red, green, and blue, plus one bit for the alpha value. |
| PBITS_RGB16 | 5 bits each for red and blue; 6 bits for green. |
| PBITS_RGB24 | 8 bits each for red, green, and blue. |
| PBITS_RGB32 | 8 bits each for red, green, and blue, plus 8 bits for the alpha value. |

The pixel type of the current camera can be found by calling the **GetPixelFormat** function.

It is important to call these functions after **SetCameraConfig**, because until the camera configuration has been set, the PXD library does not know which camera is being used, and so does not know the correct image size and pixel type.

Our sample creates a frame buffer that is the same dimensions and bit depth that the camera is configured for, so we use the **GetWidth**, **GetHeight**, and **GetPixelFormat** functions.

```
if (!(pFrame = pxd.AllocateBuffer (pxd.GetWidth(hfg),
```

```

        pxd.GetHeight(hfg),
        pxd.GetPixelFormat(hfg))) {
    MessageBox (NULL, TEXT ("Unable to create image buffer"),
szAppName, MB_ICONERROR) ;
    return 0 ;
}

```

Setting the Destination for Image Captures

Library functions send the captured image data to *frames*. Don't confuse this use of the term *frame* with the term *video frame*, which refers to a video image consisting of two fields. A *frame* stores an image and some basic information about it, including the image height, width, and number of bits per pixel.

Allocating and Freeing Frames

The most common way to create a frame for capturing images is with **AllocateBuffer()**. There are other functions which create frames useful for special purposes. These include **AllocateMemoryFrame()**, **AliasFrame()** and **MakeFrameFromPointer()** in the Frame library. See those functions in the library reference for details.

AllocateBuffer() allocates storage for a frame in main memory and calculates the physical address for the storage location, so the frame grabber can send image data directly to the buffer via DMA. The data is organized as an array of pixels in memory. **AllocateBuffer** reserves the correct amount of memory for the image, based on the width, height, and pixel type you specify.

When you want to free memory previously allocated by **AllocateBuffer()**, use the **FreeFrame()** function.

Our sample program is only going to need one image buffer, which needs to be of the right size and type to hold a single captured image. We use the image size data we collected earlier to allocate a frame.

```

if (!(pFrame = pxd.AllocateBuffer (pxd.GetWidth(hfg),
                                pxd.GetHeight(hfg),

```

```

        pxd.GetPixelType(hfg))) {
    MessageBox (NULL, TEXT ("Unable to create image buffer"),
szAppName, MB_ICONERROR) ;
    return 0 ;
}

```

Grabbing and Displaying Images

Once the libraries are opened, the frame grabber is allocated, the camera configuration is set, and a destination frame has been allocated, we're ready to capture an image.

The library includes three functions for capturing images to frames: **Grab()**, **GrabContinuous()**, and **GrabTriggered()**.

Grab() digitizes video and copies the data to the specified frame. It takes three parameters; a frame grabber handle, a destination frame, and a set of flags which tell the function when to execute (see *Using Flags with Function Calls*, page 99). For the moment, we won't be using any of the advanced features provided by the flags, so we'll set them to 0 so that **Grab()** starts digitizing as soon as the command is processed by the frame grabber. See the example CAPTURE1 for a complete application that uses the **Grab()** function in non-queued mode, and see QGrab for a complete application that uses **Grab()** in queued mode.

GrabContinuous() continuously digitizes and transfers video to the specified frame. The frame will always be in the process of being overwritten with newer pixel data, so the data in memory is as up to date as possible. This provides an easy way to keep "live" pixel data in a frame without repeatedly calling Grab, but because you don't have control of when the updates occur, you can never count on the whole frame being a consistent copy of a single image. For a program that displays images this can result in tearing artifacts in the displayed image. It is most useful when grabbing directly into a buffer on a video card that is being displayed live. See the example

CONTGRAB (in the **Samples** folder) for a complete program that uses the **GrabContinuous()** function.

GrabTriggered() waits for an external signal on the frame grabber's I/O connector before beginning image capture. This allows an image to be captured in response to an event with precisely controlled timing. See the example **GRABTRIG** (in the **Samples** folder) for a complete program that uses the **GrabTriggered()** function.

For our sample, we are only interested in capturing images and displaying them, so we just use **Grab()**. The simple **Grab()** function does not occur in the background. Instead, processing is held up until the grab is complete. To allow messages to be processed, we have set the **Grab()** function as part of the messaging loop. To keep the grab at a minimal impact, we only perform a grab once the Windows message queue is complete. This is not a useful technique if the **GrabTriggered()**, since all messaging would be suspended until the trigger completes the grab. We will describe queued operations later.

For the example, we have added a bit of code to compute the average frame rate. You will see that queued grabs can produce about twice the frame rate.

```

do
{
    if (PeekMessage (&msg, NULL, 0, 0, PM_NOREMOVE)) {
        if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE)) {
            if (msg.message != WM_QUIT) {
                TranslateMessage (&msg) ;
                DispatchMessage (&msg) ;
            }
        }
    }
    else {
        /* capture the image */
        if (pxd.Grab(hfg, pFrame, 0)) {
            ShowFrame (pFrame, hwnd) ;

            /* display the frame rate, so that it can be
            compared to QGrab.C */

```

```

        sFrameCt += 1;
        if (sFrameCt >= FRAMERATECOUNTS)
        {
            sprintf(szBuffer, "%s - %d fps",
                    szAppName, FRAMERATECOUNTS *
CLOCKS_PER_SEC/(clock() - tStart));
            SetWindowText(hwnd, szBuffer);
            sFrameCt = 0;
            tStart = clock();
        }
    }
} while (msg.message != WM_QUIT) ;

```

Once we've captured an image, we would like to display it. This requires creating and showing a window. This process requires a lot of code, but it can be done using standard Windows application programming techniques. Our sample uses a very simple window and window procedure.

Once we have an image captured, and a window to display it into, we will use the standard GDI functions. We can do this with the Windows DIB (Device Independent Bitmap) functions. This is possible because the data in the frame buffer is laid out almost the same as it would be for a DIB. The data is laid out in the buffer in a pixel, sequential format. That is, if the image is an 8 bit image, its pixel type is PBITS_Y8, each byte is a sequential pixel. If the image is a 16 bit image, its pixel type is PBITS_Y16, each pair of bytes is a sequential pixel. The pixels are laid out sequentially, starting with the upper-left corner, scanning across and then down, ending with the lower-right corner. This differs from a standard DIB, in that the DIB's are bottom up. They expect the image to be from lower-left, scanning across then up, ending with the upper-right corner. This is easily accounted for with the Windows DIB functions, by pretending the image has a negative height (a top-down DIB has a negative height, a bottom-up DIB has a positive height).

To start with, we create a bitmap header that describes the image.

```

typedef struct {
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD          bmiColors[256];
} XBITMAPINFO;
XBITMAPINFO  bmi;

memset (&bmi, 0, sizeof (BITMAPINFO)) ;
bmi.bmiHeader.biSize          = sizeof (BITMAPINFOHEADER) ;
bmi.bmiHeader.biWidth        = frameLib.FrameWidth(pFrame) ;
bmi.bmiHeader.biHeight       = -frameLib.FrameHeight(pFrame) ;
// negative paints from top down
bmi.bmiHeader.biPlanes       = 1 ;
bmi.bmiHeader.biBitCount     = 8; // force to 8, since DIB's
are 8bpp or RGB24
bmi.bmiHeader.biCompression = BI_RGB ;
bmi.bmiHeader.biSizeImage    = frameLib.FrameWidth(pFrame) *
frameLib.FrameHeight(pFrame) ;
bmi.bmiHeader.biXPelsPerMeter = 0 ;
bmi.bmiHeader.biYPelsPerMeter = 0 ;
bmi.bmiHeader.biClrUsed      = 256 ;
bmi.bmiHeader.biClrImportant = 256 ;

```

Notice that we have a bit of a trick here. The BITMAPINFO defined by Windows does not include a palette of 256 entries. Instead, it has a palette of 1 entry. We create a similar structure, called XBITMAPINFO, that can be cast as a BITMAPINFO and used interchangeably. Also note that we set the height to a negative. This is to accommodate for the top-down image. It is important to realize that a DIB only supports 8 bit palletized images, or an RGB24 image. We will perform a quick conversion from Y8 to Y16 if needed, and also create a palette in the XBITMAPINFO. This palette is a unitary palette – entry 0 has an RGB of (0, 0, 0), entry 1 has an RGB of (1, 1, 1), up to entry 255, which has an RGB of (255, 255, 255).

```

/* CreateWindowPalette -----*/
HPALETTE CreateWindowPalette(
    RGBQUAD
    lpPalette[256])
{
    /* from the example "Using DirectDraw Palettes in Windowed
    Mode" from the Windows SDK */
    typedef struct {
        WORD    palVersion;
        WORD    palNumEntries;
        PALETTEENTRY palPalEntry[256];
    } XLOGPALETTE;

    DWORD          dwCounter = 0;
    XLOGPALETTE    Palette;

    Palette.palVersion = 0x300;
    Palette.palNumEntries = 256;
    for (dwCounter = 0; dwCounter < 256; dwCounter++) {
        if (dwCounter < 10 || dwCounter > 245) {

            Palette.palPalEntry[dwCounter].peFlags = PC_EXPLICIT;
            Palette.palPalEntry[dwCounter].peRed = (unsigned
char)dwCounter;
            Palette.palPalEntry[dwCounter].peGreen = 0;
            Palette.palPalEntry[dwCounter].peBlue = 0;
        } else {
            Palette.palPalEntry[dwCounter].peFlags =
PC_NOCOLLAPSE;
            Palette.palPalEntry[dwCounter].peRed =
lpPalette[dwCounter].rgbRed;
            Palette.palPalEntry[dwCounter].peGreen =
lpPalette[dwCounter].rgbGreen;
            Palette.palPalEntry[dwCounter].peBlue =
lpPalette[dwCounter].rgbBlue;
        }
    }
    return CreatePalette((LOGPALETTE *) &Palette);
}

```

```

HPALETTE hPalette;

/*
   Create a palette for the bitmap, and Windows, so that
   the image will be displayed with the correct colors.
*/
for (dwCounter = 0; dwCounter < 256; dwCounter++) {
    bmi.bmiColors[dwCounter].rgbRed   = (BYTE)dwCounter;
    bmi.bmiColors[dwCounter].rgbGreen = (BYTE)dwCounter;
    bmi.bmiColors[dwCounter].rgbBlue  = (BYTE)dwCounter;
    bmi.bmiColors[dwCounter].rgbReserved = 0;
}
if (GetDeviceCaps(hdc, BITSPIXEL) == 8)
{
    hPalette = CreateWindowPalette(bmi.bmiColors);
    SelectPalette(hdc, hPalette, FALSE);
    RealizePalette(hdc);
}

```

To convert the 16 bpp image to an 8bpp image, we scan down the pixels in the image, pulling the high byte into another buffer that we allocated. The frame grabber always packs the data into the most significant bytes. Thus, if the grabber is configured to grab a 12 bit image, the 12 significant bits would be the top 12 bits of a 16 bit pixel. We can retrieve a pointer to the data in the frame buffer with the `FrameBuffer()` command. We only need to select and realize the windows palette if the desktop is 8 bits per pixel.

```

pbFrameBuffer = frameLib.FrameBuffer(pFrame) ;

/*
   If we have a 16 bpp image, convert to 8 bpp by pulling
   the high byte into its own buffer. The bmi.bmiHeader was
   already set the the number of bytes in the image.
*/
if (frameLib.FrameType(pFrame) != PBITS_Y8) {
    pbFrameBuffer++; /* skip to the high byte */
    pbCompressed = GlobalAlloc(GPTR,
bmi.bmiHeader.biSizeImage);
    for (dwCounter = 0; dwCounter <
bmi.bmiHeader.biSizeImage; dwCounter++) {

```

```

        pbCompressed[dwCounter] = *pbFrameBuffer;
        pbFrameBuffer++;
        pbFrameBuffer++;
    }
    pbBuffer = pbCompressed;
} else {
    pbBuffer = pbFrameBuffer;
}

```

To perform the actual display, we just need to configure how we want the data to be transferred (blitted, for Block Transfer) to the display. It is usually fastest to avoid any technique that averages dropped pixels when shrinking the image. For our example, we will copy a subset of the images if the image is reduced, by calling **SetStretchBltMode** with **COLORONCOLOR**. We then call one of the DIB copy commands – we are using **StretchDIBits**, so that the image will be scaled to fill the window. Other functions that can be used are **SetDIBits** and **SetDIBitsToDevice**.

```

SetStretchBltMode (hdc, COLORONCOLOR) ;
GetClientRect (hwnd, &rect) ;

:
:
    /*
        We will use StretchDIBits to display the image. It will
stretch the image to fill the window
    */
    StretchDIBits (hdc, 0, 0, rect.right, rect.bottom, 0, 0,
        frameLib.FrameWidth(pFrame),
frameLib.FrameHeight(pFrame),
        pbBuffer, (BITMAPINFO *) &bmi, DIB_RGB_COLORS,
SRCCOPY) ;

```

Displaying Live Video in a Dialog Box

Note that we displayed the image to fill a window that we created. It is possible to display live video into dialog box as well. To do this, a few small changes would be required. First of all, we would not create a window, but we would create a dialog box. Also, instead of just calling **DialogBox**, which would create a modal dialog box, we use **CreateDialog**. This allows us to use

our own message processing loop, so that we can process the grabs in the same manner as the other samples. (This is taken from the **DLGSAMP** sample in the **Samples** folder)

```
hwnd = CreateDialog(hInstance, MAKEINTRESOURCE(DIALOG_SAMPLE),  
NULL, DlgProc);
```

DIALOG_SAMPLE is the ID of the dialog box that we created.

To show the live video into This is done by simply providing a different window in the call to **ShowFrame**. For example, assume that you have created a dialog box with some controls and you wish to show live video from the frame grabber. In the example, the dialog box contains a field declared as a type frame, named **FRAME_FIELD**. To display the image into that field, just provide that field's window handle to **ShowFrame** by using the **GetDlgItem** function.

```
ShowFrame (pFrame, GetDlgItem(hwnd, FRAME_FIELD)) ;
```

Advanced Topics

Initializing and Exiting Libraries

Before calling any other library functions, you must explicitly initialize each library by calling the appropriate **OpenLibrary()** function. Following your last call to a library, before your program terminates, you must call the appropriate **CloseLibrary()** function. The actual function names are specific to the operating system and language you are using, and are described in the following sections.

C and Windows Programs

The **OpenLibrary()** and **CloseLibrary()** functions for the PXD1000 Frame Grabber library under Windows NT, 98 and 95 are:

```
imagination_OpenLibrary("pxd_32.dll", &pxd,
                        sizeof(pxd)) ;
imagination_CloseLibrary(&pxd) ;
```

The **OpenLibrary()** and **CloseLibrary()** functions for the Frame library under Windows NT, 98 and 95 are:

```
imagination_OpenLibrary("frame_32.dll", &frm,
                        sizeof(frm)) ;
imagination_CloseLibrary(&frm) ;
```

Where *pxd* and *frm* are the names you will use for the structures for calling library functions. For 16-bit Windows 3.1 and Windows 95 programs, substitute *16* for *95* or *32* in the name of the DLL in the examples above.

In the Windows versions of the libraries, the interrupt handlers are installed by the low-level device drivers; the virtual device drivers (VxDs) in Windows 3.1 and Windows 95. By default, the low-level device driver is loaded when you start Windows, and is uninstalled when you exit Windows.

C and DOS Programs

The **OpenLibrary()** and **CloseLibrary()** functions for the PXD1000 Frame Grabber library and the Frame library for C programs under DOS are:

```
PXD_OpenLibrary(&pxd, sizeof(pxd))  
PXD_CloseLibrary(&pxd)
```

```
FRAME_OpenLibrary(&frm, sizeof(frm))  
FRAME_CloseLibrary(&frm)
```

Where *pxd* and *frm* are the names you will use for the structures for calling library functions.

In the DOS and DOS/4GW versions of the library, initializing the library installs an interrupt handler that is needed for frame grabber communication, and exiting the library uninstalls the interrupt handler. If your program crashes or terminates without calling **CloseLibrary()**, you will probably need to reboot your system, as it may be in an unstable state.

Troubleshooting OpenLibrary()

Check the return value from **OpenLibrary()** to make sure the function was successful (non-zero = success). **OpenLibrary()** functions will fail under Windows if the DLLs or drivers are not present.

The **OpenLibrary()** functions for the Frame library and the DOS VGA Video Display library should fail only when the system has insufficient memory; each function allocates a small amount of memory for internal data structures.

OpenLibrary() for the PXD1000 Frame Grabber library can fail under the following conditions:

- The PCI BIOS does not exist or is malfunctioning. Your computer probably has a hardware problem.
- The PCI BIOS was unable to assign an IRQ to the frame grabber. You may need to modify your CMOS settings to make more IRQs available to the PCI BIOS.
- There is no suitable memory block in upper memory. In DOS, each frame grabber requires a contiguous 4KB block of upper memory, and **OpenLibrary()** will try to find such a block.
- There is insufficient conventional memory. **OpenLibrary()** allocates a small amount of storage for internal data structures.
- There are no Imagenation frame grabbers in your computer, or they are malfunctioning.

Sending Images Directly to Another PCI Device

Some devices, such as high-end PCI video cards, have a physical address where they can receive data via direct memory access (DMA). (Don't confuse this *physical* address with the *logical* addresses or *pointers* that software normally uses. A physical address is a low-level construct that the hardware uses in its internal communication, and is independent of the operating system.) This provides a high-performance path for capturing images directly to the device. For example, some PCI video cards have a *flat addressing mode* that allows DMA transfers to the card without having to swap pages of video memory in and out. With such a card, you should be able to display video in real time. To find out if your video card supports flat addressing, and how to determine the physical address for the card, refer to the documentation that came with the card or contact the manufacturer.

Pixel Formats

The most common pixel formats will be 8, 10, and 12 bit grayscale. 8 bit grayscale is stored in the PBITS_Y8 type, in the obvious way. 10 and 12 bit grayscale are stored in the PBITS_Y16 type, such that the most significant bits are valid data, and the least significant bits are all set to 1. If some other format is desired, such as data in the least significant bits, the LUTs can be programmed to provide this.

Grabbing Images

The library includes two functions for grabbing images to frames: **Grab()** and **GrabContinuous()**.

Grab() captures video and copies the data to the specified frame. You can specify which video field the capture should start on, whether to capture one field or both, and when to execute (see *Using Flags with Function Calls*, page 99). It starts capturing as soon as the command is processed by the frame grabber.

GrabContinuous() continuously digitizes and transfers video to the specified frame.

If the PCI bus is overloaded, it's possible for captured data to be corrupt. Although the Grab functions can't determine when data is being corrupted, **CheckError()** will return the value ERR_CORRUPT.

The most common reasons the **Grab** functions fail are:

- The frame grabber handle or the frame buffer handle is invalid.
- The image specified by **SetWidth()** or **SetHeight()** (or the default image size) is too large in width or height for the frame buffer..

If the **Grab** functions execute successfully, but don't produce the image you expect, the most common reasons are:

- If the captured image is all black or all blue, be sure to check that your video source is fully attached to the frame grabber and that the iris on the video camera is open.
- If you're using a system with an Intel Pentium Pro processor, you might not be able to read valid data from a frame buffer in system memory immediately after grabbing the image. This is due to the processor caching the data, rather than writing the data immediately to memory. Try inserting a delay in your program before reading the data.
- If you get only a few lines of valid video at the top of an image you've grabbed to a frame buffer in system memory, the PCI bus is being overloaded or errors are occurring on the bus. Most Intel 486-based systems don't have a PCI bus that is fast enough for PXD1000 frame grabbers. Run the VGACOPY program to check for errors on the PCI bus.
- The frame grabber can't produce the image specified by **SetHeight()**, **SetWidth()**, **SetXResolution()**, and **SetYResolution()** (see *Cropping Images*, page 94).

Cropping Images

You can grab a subregion of the full image. That is, you can crop images vertically and horizontally as they are grabbed. You crop an image in width by specifying the starting column and number of columns to keep, using the **SetLeft()** and **SetWidth()** functions. You crop an image in height by specifying the starting row and number of rows to keep using the **SetTop()** and **SetHeight()** functions. You can get the current values with **GetLeft()**, **GetWidth()**, **GetTop()**, and **GetHeight()**.

Timing the Execution of Functions

The PXD1000 software library includes some advanced features for applications that are time-critical. These features let you determine whether functions should be executed immediately, or if they should be placed in a queue to execute asynchronously while the program proceeds.

Queued Functions

Frame grabber applications often include a loop that repeatedly grabs a frame and then processes the information in it. For example:

```
/* not a recommended technique for Windows*/
for (;;)
{
pxd.Grab(fgh, fbuf, 0);
Process_Image(fbuf); /* your function */
}
```

where *fgh* identifies the frame grabber, *fbuf* specifies the frame handle, and *0* indicates that `Grab()` is to use the default settings.

This technique of serially grabbing and processing frames is straightforward and easy to implement using the PXD1000 library. However, there are disadvantages to this serial process:

- While the image is being processed, the frame grabber can't grab images, and much of the video image data that the camera is receiving never gets processed.
- While the frame grab is occurring, the computer's CPU can't do any image processing and sits idle waiting for the next frame.

PXD1000 frame grabbers transfer image data to a frame using direct memory access (DMA), which bypasses the computer's operating system. DMA makes it possible to have the frame grabber moving data to one frame, while at the same time the application is processing image data in another frame. The library has been designed to take advantage of this parallel activity. Certain functions can be designated as *queued*, by specifying the QUEUED flag in the function call (see *Using Flags with Function Calls*, page 99). A queued function will return as soon as it puts the necessary information in the queue, without waiting for the operation to execute. This frees the application to continue processing. Here's an example of how you might use this capability:

```
long grab1, grab2;
grab1 = pxd.Grab(fgh, fbuf1, QUEUED);
grab2 = pxd.Grab(fgh, fbuf2, QUEUED);
pxd.WaitFinished(fgh, grab1)
; /* wait until grab 1 has completed */
for (;;)
{
ProcessImage(fbuf1);
grab1 = pxd.Grab(fgh, fbuf1, QUEUED);
pxd.WaitFinished(fgh, grab2)
; /* wait until grab 2 has completed */
ProcessImage(fbuf2);
grab2 = pxd.Grab(fgh, fbuf2, QUEUED);
pxd.WaitFinished(fgh, grab1)
; /* wait until grab 1 has completed */
}
```

The **WaitFinished()** function is used to pause until a function has completed. In the example above, once **WaitFinished()** indicates that the first **Grab()** is complete, the program starts processing the first image. **WaitFinished()**

can check on a specific function in the queue (as in this example), or check to see if all functions in the queue are complete.

If your system has more than one frame grabber installed, each frame grabber has a separate queue, and **WaitFinished()** checks the appropriate queue based on the handle *fgb* that you specify..

Check the sample **GrabQ** (in the **Samples** folder) for an example of a complete application which uses queued grabs. Queued grabs typically yield twice the frame rate of non-queued grabs. For an example of non-queued grabs, see the **CAPTURE1** sample in the **Samples** folder.

Synchronizing Program Execution to Camera Video

The library function **Wait()** can be used to synchronize program execution to incoming video:

Wait() can wait for the end of the next field, the end of the next frame (two complete fields), or the end of a specific field before returning. **Wait()** takes exactly as much time as a **Grab()** with the same parameters. Since the **Wait()** function can be queued, it is most useful for synchronizing queued functions to video.

You can also synchronize program execution based on the state of I/O lines (see *Digital I/O*, page 100).

Purging the Queue

The **KillQueue()** function purges any pending functions in the queue and terminates any that are executing. This function is designed for error recovery and should only be used when the queue appears to have stopped processing functions.

The results of any functions in the queue when **KillQueue()** is called are undefined. For example, if a call to **Grab()** is in the queue when **KillQueue()** is called, the image data in the frame might not be valid.

Immediate Functions

You can specify that a function should only execute if there is nothing in the queue. The library provides the flag IMMEDIATE for this purpose. If a function specified as *immediate* executes when functions are in the queue, it will return failure without doing anything. Otherwise, the function will return when it has completed.

Function Timing Summary

The *queued* and *immediate* settings are not mutually exclusive. A function can be declared to be either one, neither, or both. The behavior of each setting is summarized below:

Neither queued nor immediate Executes when all functions in the queue have completed, and returns when execution is completed. This is the default.

Queued Execution is deferred until previously queued functions have executed. The function returns immediately, and the program continues to the next statement. The frame grabber executes the queued instructions concurrently with the program's execution of any non-frame grabber functions.

Immediate Only executes if there are no functions in the queue. The function returns when execution is completed.

Queued and Immediate Only executes if there are no functions in the queue. The function returns immediately, and program continues to the next statement. The frame grabber executes the queued instructions concurrently with any non-frame grabber functions. If there is a non-queued function in progress, the application doesn't proceed until that function is complete.

Many applications don't require the QUEUED and IMMEDIATE flags. If you don't use either flag, the function executes as soon as the frame grabber has finished the previous operation, and the function returns when the frame grabber has finished executing it..

You can use the QUEUED and IMMEDIATE flags with any of the following functions:

| | | |
|-------------------------|-----------------------|------------------------|
| Grab() | SetCamera() | Wait() |
| GrabContinuous() | SwitchCamera() | WaitAllEvents() |
| SetBrightness() | SwitchGrab() | WaitAnyEvent() |
| SetContrast() | | |

These functions return a *handle* that can be used by **IsFinished()** and **WaitFinished()** to check their progress.

The following functions always wait until all functions in the queue have completed before executing:

| | | |
|------------------------|---------------------------|-------------------------|
| GetFieldCount() | SetWidth() | SetIOType() |
| SetFieldCount() | SetXResolution() | SetPixelFormat() |
| SetHeight() | SetYResolution() | SetVideoDetect() |
| SetLeft() | SetChromaControl() | SetVideoLevel() |
| SetTop() | SetLumaControl() | |

All functions not listed here will execute when they are called and return when they have completed. They may execute concurrently with functions in the queue.

Timeouts

For many queueable operations, there is the possibility that the operation will try to synchronize to an event which never happens. Triggered grabs can wait for a trigger that never comes, and even normal grabs can wait for an image which never comes if the current camera is asynchronous. For applications where these things may happen, the **TimedWaitFinished** function provides a way to abort after a certain time period, or occasionally poll for other events while the queued operation is in progress.

```
short TimedWaitFinished(long fgh, long qh, float timeout);
```

The function returns TRUE if the operation finished, FALSE if time expired.

This function is like **WaitFinished**, except that it will return after the specified timeout occurs, even if the queued operation has not completed. The application can then take appropriate action.

Some examples:

Aborting a triggered grab if no trigger comes for 10 seconds:

```
pxd.GrabTriggered(fgh, frh, 0, QUEUED);
if (!pxd.TimedWaitFinished(fgh, 0, 10.0)) {
    pxd.KillQueue(fgh);
}
```

Polling for user input while a triggered grab is in progress:

```
pxd.GrabTriggered(fgh, frh, 0, QUEUED);
while (!pxd.TimedWaitFinished(fgh, 0, 0.1)) {
    DoInputs();
}
```

Using Flags with Function Calls

Several of the frame grabber control functions take a set of flag bits as one of their parameters. The possible flags are:

| Flag | Description |
|-----------|---|
| EITHER | Operation will start on the next field. |
| IMMEDIATE | Operation will fail if the frame grabber is busy. |
| QUEUED | Operation will be queued for later processing. |

Flags can be combined with the bitwise OR operator.

The default behavior (*flags* = 0) for a function that uses flags is:

- Wait until the frame grabber is not busy.
- Start on the next field.
- Process a two-field, interlaced frame (if the function processes an image).
- Return after the operation is complete.

Not all flags are relevant to each function that has a *flags* parameter. For example, some functions, such as **SetBrightness()** and **SetHue()**, ignore the FIELD choice flags and always operate as if the EITHER flag was specified.

Digital I/O

This section discusses programming the digital I/O lines on the PXD1000. The PXD1000 includes a single digital input (line 0) that lets you synchronize the frame grabber with other devices in the system. The optional Control Package adds I/O lines, for a total of four input lines (lines 0-3) and four output lines (lines 4-7). If your frame grabber doesn't have the Control Package, none of the information on *Controlling the Output Lines*, page 102, applies to your board, and the functions in *Controlling the Input Lines*, page 101, can only be used with line 0.

Strobes, Triggers, and Digital I/O

The PXD frame grabbers include two strobe outputs, one trigger input, three camera control outputs, and two general purpose inputs and outputs. Depending on the camera configuration, some of these may be automatically set up for camera control. The signals which are not put to special uses by the camera configuration are available for application use. The documentation which comes with the configuration files for a particular camera should say which signals are used for camera control and which are available to the user. Note that the software does not stop an application from

reprogramming a signal which has been set up by the camera configuration, so care should be taken to avoid doing this accidentally.

The two strobe signals can be set to a particular value and read back. The frame grabber can also generate pulses automatically on the strobe lines.

The trigger line can be used to start a frame grab or start a strobe pulse sequence, or both. Each of these events can be triggered on either a rising or falling edge of the trigger, and grabs can be triggered on either a high or low level of the trigger.

Controlling the Input Lines

Setting Up and Reading the Input Lines/Using an Input Line as a Trigger

The input lines of the frame grabber can be read by polling them, or can be used to trigger an event. To poll the input lines, that is to read their current state, use the **ReadIO** function. This returns a short, which can be masked against the input macros.

Input macros

| | |
|---------|---|
| TRIGGER | - the Trigger bit |
| WEN | - the Write Enable bit. Provided by some cameras. |
| FIELD | - the interlace field indicator bit. Either 0 or 1. |
| LDV | - the Line Data Valid bit. Provided by some cameras. |
| FDV | - the Frame Data Valid bit. Provided by some cameras. |
| GPIN0 | - TTL input 0 |
| GPIN1 | - TTL input 1 |

Any of these input lines can be used to trigger a grab. This is done with the **SetTriggerSource** function along with the **GrabTriggered** function. (See the sample **GRABTRIG** in the **Samples** folder.) The **SetTriggerSource** function identifies which of the input lines will be used to trigger the grab, and the **GrabTriggered** will perform the grab when the trigger occurs.

```
pxd.SetTriggerSource(hfg, TRIGGER, LATCH_RISING);  
  
qh = pxd.GrabTriggered(hfg, pFrame, 0, QUEUED);
```

Dealing with Trigger Bounce on Input Lines

The grabber has the ability to condition the trigger input signal, to eliminate the bounce from a switch. This requires two trigger switches. One of the switches will raise the trigger signal, the second will drop it. The common method to implement this is to replace a SPST switch (single pole, single throw) with a SPDT switch (single pole, double throw). Thus, instead of just breaking a signal, as the SPST would do, the signal will be routed to one of two locations. One of the trigger control lines is sent to the TRIGGERP, the other to TRIGGERN. The TRIGGERP line is connected to pin 14 of the I/O connector. The TRIGGERN line is connected to pin 15 of the I/O connector. This feature is enabled by including the DEBOUNCE flag when requesting a triggered grab.

```
pxd.SetTriggerSource(hfg, TRIGGER, LATCH_RISING | DEBOUNCE);
```

Controlling the Output Lines

Writing to the Output Lines

Several of the I/O lines can be used for output. These are accessed by the WriteImmediateIO lines, by providing a combination of bit fields. The provided bit fields are:

| | |
|--------------|---|
| HDRIVE_MASK | - the Horizontal Drive pulse |
| VDRIVE_MASK | - the Vertical Drive pulse |
| STROBE0_MASK | - the I/O connector strobe 0 TTL line |
| STROBE1_MASK | - the I/O connector strobe 1 TTL line |
| CTRL0_MASK | - general purpose camera control line 0 |
| CTRL1_MASK | - general purpose camera control line 1 |
| CTRL2_MASK | - general purpose camera control line 2 |
| GPOUT0_MASK | - general purpose I/O connector GPOUT0 |
| GPOUT1_MASK | - general purpose I/O connector GPOIT1 |

To set these lines to specific values the **WriteImmediateIO** command can be used. This function requires a mask field, to identify which IO lines are being controlled, and another bit-field that identifies the state of that line. Any lines that are not included in the mask field are not modified. For example, to turn on the GPOUT0 line, and to turn off the GPOUT1 line, use a command such as

```
pxd.WriteImmediateIO(fgh,
    GPOUT0_MASK | GPOUT1_MASK,
    GPOUT0_MASK);
```

Using the Automatic Strobe Functions

It is possible to trigger the strobes automatically when a trigger occurs. This is useful to use a trigger that goes to the grabber to automatically control another device through the strobes. This is sent from the frame grabber, not the software driver, so latency is within a couple of camera clocks. Thus it is effectively immediate.

An example of sending a strobe as soon as a the general purpose line goes high:

```
pxd.TriggerStrobes(fgh, GPIN0, LATCH_RISING, EITHER);

pxd.GrabTriggered(fgh, 0, 0);
```

The strobe only occurs once per trigger. But the sequence of strobes that were programmed will be sent. This strobe sequence can be controlled with the **SetStrobePeriods** command. The strobes will be send automatically for all following triggers, until the **TriggerStrobes** is cancelled by setting the trigger source to 0.

Reading Frame Grabber Information

From any version of Windows, you can get the frame grabber information – the board revision, the FPGA version, and the software version, from the

ViewPXD application. Run the **ViewPXD** application, and select the frame grabber. From the **View** menu, choose **Grabber Configuration....** (See Chapter 3, *Using the PXD1000* for information on running the **ViewPXD** application.) You can also read the board revision number in the software application you create by using the **ReadRevision()** API function.

Hardware Protection Key

You can request to have your frame grabbers encoded with a unique protection key that your software can read using the **ReadProtection()** function. Checking for the key in software gives you some protection against software piracy, since you can prevent the software from running on systems that you have not supplied.

Serial Number

You can request to have your frame grabbers encoded with a serial number, which can be used to identify a specific board. The **ReadSerial()** function returns the encoded serial number, if any.

Serial Numbers, Protection Keys, and Revision Numbers

A PXD frame grabber is a combination of hardware and firmware, with some of the firmware kept in the driver and uploaded to the hardware at runtime. This means that the revision number of a particular frame grabber must describe both the hardware and the software used by the device. For this reason, the number returned by **ReadRevision()** is 32 bits, with 16 bits specifying the hardware revision, and 16 bits describing the firmware. The firmware revision is further divided into 8 bits of major revision and 8 bits of minor revision. Changes in major revision generally require updated driver software, while changes to minor revision only require a new firmware image. Firmware images are distributed as .PXD files, which must be in the system path in order for the driver to run properly. In general, a single PXD file can contain firmware for several different revisions of the

hardware, so that only one .PXD file is needed even in systems which have several frame grabbers of different ages.

A frame grabber can contain 128 bytes of protection key information, which can be factory set to meet customer requirements. Normally, this data will be 0.

The software API allows each frame grabber to have a unique 16 bit serial number. Normally, the PXD series is manufactured with the serial number set to 0, but this can be factory set to other values to meet customer requirements.

Accessing Captured Image Data

You can access image data stored in a frame in main memory in two ways:

- Use the **FrameBuffer()** function to get a *logical* address (a pointer) to the data and use the pointer to operate directly on the data. You can use **FrameBuffer()** only on frames you create with **AllocateBuffer()**, **AllocateFlatFrame()**, and **AllocateMemoryFrame()**; frames you create with **AllocateAddress()** can't be read by the library, so you can't use **FrameBuffer()** to get a logical address to those frames.
- Use the **GetPixel()**, **GetRectangle()**, **GetRow()**, and **GetColumn()** functions to copy parts of the image data from a frame to a buffer you have created in memory. Use the **PutPixel()**, **PutRectangle()**, **PutRow()**, and **PutColumn()** functions to copy parts of the image data from a buffer you have created in memory to a frame. For languages, such as Visual Basic, that do not have pointers, these functions are the only way to access the data in a frame buffer. These functions will cause unpredictable results if the buffer you are copying to isn't large enough to hold the data.

The following functions are also useful in working with frame data:

CopyFrame()—Copies a rectangular region of pixels from one frame to another frame.

ExtractPlane()—Returns a frame containing one of the planes from a frame containing planar data, such as YUV422P or YUV444P.

FrameHeight(), **FrameWidth()**, and **FrameType()**—Return, respectively, the height, width, and type of pixel data for the specified frame.

AllocateMemoryFrame()—Can allocate frames for any of the pixel data types, including the floating point types PBITS_Yf and PBITS_RGBf. The memory for the frame is not guaranteed to be in one contiguous block.

AllocateFlatFrame()—Can allocate frames for any of the pixel data types, including the floating point types PBITS_Yf and PBITS_RGBf. The memory is guaranteed to be in one contiguous block.

You can use **FrameAddress()** to get the *physical* address for a buffer, but don't try to use this physical address to access data in an application; use the logical address returned by **FrameBuffer()** instead. **FrameAddress()** is provided only for special situations in which a physical address might be needed, as in writing device drivers.

Frame and File Input/Output

The library provides functions for writing and reading image data to and from files. You can read and write unformatted (binary) files, Windows BMP formatted files, and PNG formatted files. Formatted files include information about the image, including the width, height, and number of bits per pixel, while binary files include only the pixel values.

PNG Files

The PNG routines **ReadPng()** and **WritePNG()** read and write frames to image files on disk using the Portable Network Graphics format. Y8 images are written and read with 8-bits-per-pixel. Y16 images are saved at 16-bits-per-pixel, with no conversion or lossy compression. Because the pixel data is stored in a compressed form, but without the data being converted to a new bit depth or colors, this is the preferred storage method.

If a PNG file is read into a frame that does not have room to store the entire PNG image, the image is clipped on the right and bottom edges. If the PNG file image is smaller than the frame, the image is padded on the right and bottom with zeros.

BMP Files

The BMP routines **ReadBMP()** and **WriteBMP()** read and write frames to image files on disk using the Windows BMP formats. Y8 images are written and read as 8-bit-per-pixel BMP files with a grayscale palette. RGB images are written and read as 24-bit, true-color BMP files. In RGB32, the alpha data is ignored.

If a BMP file is read into a frame that does not have room to store the entire BMP image, the image is clipped on the right and bottom edges. If the BMP file image is smaller than the frame, the image is padded on the right and bottom with zeros.

Binary Files

The routines **ReadBin()** and **WriteBin()** read and write unformatted image data to and from files. Unformatted files contain no information on an image's height, width, or pixel type, so you must keep track of that information. For example, nothing prevents you from saving a frame that is 320 pixels wide and 160 pixels tall in an unformatted file, and then reading that file into a frame that is 160 pixels wide and 320 pixels tall, even though each

line of the original frame will occupy two lines in the new frame. If you use unformatted files, keep track of the characteristics of the stored frames.

Video Display and File I/O

Because the PXD software uses FRAME structures compatible with PXD1000 frames, most image manipulation libraries and tools written based on the frame library will work with the PXD. This includes all frame library buffer manipulation and file I/O functions, and the video display library. The PXC DirectDraw library (available with the PXC products) may not get good performance with the PXD series, however.

Although the frame and video libraries will work with the PXD frame grabbers, they do not have support for 10 and 12 bit monochrome pixel types used by some digital cameras. A new version of the frame library is provided with the PXD software which supports the PBITS_Y16 data format, which can hold any pixel type between 9 and 16 bits. It is completely backwards compatible with older versions of the frame library.

Using the new frame library, 10 and 12 bit images can be written to disk as raw binary files using the **WriteBin** function. These files will be twice the size of an 8 bit image at the same resolution, because each pixel is written as a full 16 bits. Bit depths larger than 8 bits can not be written as BMP files, however, because the BMP file format does not support these image types. An additional pair of functions, **WritePNG** and **ReadPNG** have been added to the frame library to manage PNG format files which can support the larger bit depths.

Video display of 10 and 12 bit images poses similar difficulties, because most display cards do not support these bit depths. There are 3 options for displaying deep grayscale images.

- Software translation. The image can be acquired in deep grayscale, and a software routine can be written to translate the image to 8 bit grayscale, or whatever other format is desired. This method is the

most flexible, but it requires an additional 8 bit per pixel buffer to hold the translated image, and for large images can require a lot of processor time.

- **Hardware 8 bit mode.** Most 10 and 12 bit cameras can be used as 8 bit cameras by selecting an appropriate configuration file. An application can switch between the two configurations, using 8 bits for display and 10 or 12 bits to capture images for analysis. This technique provides good update speed to the display because no translation is needed on the 8 bit images, but switching between the two camera modes can take several milliseconds.
- **Pseudo-color lookup table.** The hardware lookup table on the PXD can be used to map the 10 or 12 bit grayscale pixels into the 16 bit RGB color space commonly used by display cards. The pseudo-color image may then be displayed directly. This technique provides good update speed to the display, and pseudo-color images can be helpful when an operator needs to pick low contrast details out of the image. Switching between pseudo color and grayscale requires reloading the lookup table, however, which can take several milliseconds. If occasional grayscale images are needed for processing during pseudo-color display, it may be more efficient to translate backwards from color to grayscale in software rather than reloading the lookup tables. To support this technique, the PXD software allows grabs to any frame which has the correct number of bytes per pixel, even if its type does not match the pixel type specified in the camera configuration.

Working with Camera Configurations

The world of digital cameras is rather complicated. There are many different cameras, with many different features, and no clear standards specifying how they should work. The PXD series is designed to be flexible enough to support a wide variety of cameras. An application program can find the information necessary for the frame grabber to work with a particular cam-

era in two ways –from a configuration file at runtime, or from a header file at compile time.

Configuration files are useful for applications which need to work with a variety of cameras. If the application allows the user to select a configuration file at runtime, then the program does not need to be recompiled to support new cameras; the user can simply download a configuration file for the new camera. On the other hand, managing a large collection of configuration files can be confusing and error prone for the application's users. Application designers which intend to use configuration files in their programs should decide early on a strategy for organizing the files and reacting appropriately if an expected file is missing.

If a program is designed around the features of a specific camera or a small set of cameras, it may be more convenient to have the camera configuration information built into the program itself. This can be done through camera header files. These files can be included in a C program to make the configuration data for a particular camera available as static global data. Unlike most C header files, these files will actually allocate storage for the camera data, so including large numbers of them will increase a program's size.

A particular camera might have several different operating modes. If these modes are different enough, the camera might have a separate configuration file for each mode. For instance, a camera which can run as either a free-running video source or a triggered single frame source might have a configuration file for each mode. Imagenation will supply application notes for the cameras we support describing the supported modes and which configuration files should be used for which modes. These are some common features which might require separate configuration files.

- Interlaced mode or full frame exposure mode.
- Asynchronous resettable mode or free running mode.
- Programmable exposure times.

Several fields of a camera configuration file may be of interest to an applications programmer:

- **pix_type:** This is a recommendation about what type of frames the camera works best with. This value can be passed directly to **AllocateBuffer**.
- **frame_period:** this is a float value which gives the time required (in seconds) for the camera to output one video image. If the camera runs continuously rather than being triggered then $1.0/\text{frame_period}$ is the number of images per second produced by the camera.
- **width, height:** these give the dimensions in pixels of the images produced by the camera. This image can be cropped to a smaller size when it is captured, if desired.

Setting the Camera Configuration

The PXD1000 can be used with a wide variety of digital cameras. The PXD library must be told what sort of camera is connected to the frame grabber in order to properly grab images. One way to do this is with a camera configuration file.

Programming the PXD using a camera configuration file is a two step process. First the configuration file is loaded from disk using the **LoadConfig** function. Then the frame grabber is programmed to use the file's settings with the **SetCameraConfig** function. The camera description created by **LoadConfig** is no longer needed after **SetCameraConfig** has been called. If you expect to change camera configurations later in the program, you can keep it for later use; otherwise it can be freed immediately with the **FreeConfig** function.

Imagination supplies several camera configuration files on the software installation disk. You may need to use a full path name, rather than just a file name for some camera configuration files, for example:

```
#define CAMERA_NAME "c:\\pxd\\cameras\\Hitachi\\kpf100.cam"
```

(A backslash (\) must be typed as a double backslash (\\) inside quoted strings in C.)

```
camera=pxd.LoadConfig(CAMERA_NAME) ;
if (camera) {
    pxd.SetCameraConfig(hfg, camera) ;
    pxd.FreeConfig(camera) ;
}
else {
    MessageBox (NULL, TEXT ("could not load requested camera
file. Trying default configuration."), szAppName, MB_ICONERROR)
;
}
```

The Default Configuration

If **SetCameraConfig** has never been called, or if the configuration you tried to set is not valid for some reason, the PXD library will use the camera configuration from the file DEFAULT.CAM to configure the frame grabber. You do not have to do anything special to set up the frame grabber to use the default camera settings. Our sample program, for example, will use the default settings if **LoadConfig** can't find a camera configuration file, even though there is no special code in the program to cause this to happen.

If you've installed the development software from Imagination's install disk, you were given the opportunity to create a default camera configuration by selecting from a list of supported cameras. Details of how to set up camera configuration files, including the default configuration, are found in Chapter 4 *Adding a Camera*.

Generally, using the default camera configuration to recover from errors, or providing the option of using the default configuration, is a good way to add flexibility to a program. You can also write a program which doesn't ever call **LoadConfig** or **SetCameraConfig** and relies on the default camera configuration as the only way for the PXD library to find out about what camera is installed, but this can cause problems in cases where the program's users do not have a DEFAULT.CAM file, or it is not set up correctly.

Video Timing

In order to achieve high performance, many application programs which use PXD frame grabbers will need to be closely synchronized to the timing of the digital camera. Because of the wide variety of digital cameras as well as the high data rates some of them can supply designing programs which synchronize well with cameras can be a challenge. In particular, some techniques which work well for Imagination's other frame grabbers may not be appropriate for the PXD series.

Timing Differences Between the PXD Series and Other PCI Grabbers

Imagination's other PCI frame grabbers base their synchronization capabilities on a decision point at a fixed time in the camera's vertical blank period. Several assumptions are made about the timing of decision points.

- For stable video, the decision point occurs once per vertical blank, and represents a stable timing reference usually at either 50 or 60 cycles per second.
- At the decision point, all data being captured from the previous field has been written to memory, and no data has yet been captured from the next field.

- All commands in the frame grabber's job queue begin and end execution exactly on a decision point. Some commands take so little time to execute that several of them can occur at a single decision point.
- The frame grabber driver has enough time during vertical blank that it can respond to an interrupt from the frame grabber at each decision point, do a significant amount of processing, and schedule activity for the next field of video. All this happens in the vertical blanking period while the camera is not generating any video data.

Depending on the camera being used, some or all of these assumptions may be inaccurate for the PXD frame grabbers. These are some of the major features of the PXD grabbers which don't fit in well with the decision point model.

- The PXD grabbers support line scan cameras, which provide a continuous stream of horizontal lines of data with no vertical sync information, so there is no correct place to put decision points.
- Some cameras produce small images very rapidly –for instance 256 by 256 pixels at 1000 frames per second. The time required for an application and driver to requeue a new buffer for each frame in these situations is prohibitively large.
- In order to reliably capture images at high data rates, the PXD grabbers have very large internal buffers. The data being written to system memory may be a significant fraction of a second older than the data coming from the camera. Under these conditions, requiring all data from one video frame to be committed to system memory before the next can be captured would prevent real time capture of successive video frames.
- Many digital cameras do not generate stable streams of video, but instead produce single frames in response to trigger events. These frames can be separated by large periods of dead time in which the

frame grabber is not receiving sync information from the camera. For these cameras, vertical sync is not a reliable timing reference.

The PXD Synchronization Model

The PXD frame grabber driver tries to do two things with queued commands. It tries to execute the next command in the queue as soon as possible, and it tries to give grab commands which are sequential in the queue sequential frames of video data.

The frame grabber has a large FIFO for buffering video data to ensure reliable transfers. If the PCI bus is busy, this FIFO can begin to fill up, such that the data being read by the current Grab command is significantly older than the data coming from the camera. In this situation, a queued Grab command is considered finished by the **IsFinished** and **WaitFinished** commands when all data for the grabbed image is in system memory, which might be much later than when all the data has been read from the camera. If the next command in the queue is a Wait, a frame of video will be discarded from the FIFO. This will happen at about the same speed as a grab if the FIFO is mostly empty, or much faster if the FIFO already contains much of the image. As long as there are more operations pending in the queue, they will process sequential video frames (unless the FIFO becomes completely full, at which point frames will be dropped). If the frame grabber's queue becomes empty, all data in the FIFO will be discarded. This ensures that although grabs which are queued together tend to get sequential frames, grabs which are called separately get the newest possible data. **GrabTriggered** will also flush any data which is in the FIFO before waiting for a trigger, to ensure that the data captured in the grab actually represents the frame marked by the trigger.

Lists of Frame Buffers

In a case where the camera is producing frames very quickly, the overhead required to track a Grab operation for each frame captured may be unacceptably high. The PXD frame grabbers can fill multiple buffers from a sin-

gle grab command using buffer lists. A list of several identical frames can be allocated using the **AllocateBufferList** function. This list has a single FRAME pointer representing the entire list, and calling Grab with this frame pointer fills the entire list with successive images. The individual images can be extracted from the list using the frame library's **ExtractPlane** function. Most frame library functions will work on the individual extracted frames but not on the entire list at once. The **WaitMultiple** function can be used to skip multiple frames in the same way that **Wait** is used to skip a single frame.

Timing Techniques

WaitVB as a timer: For the PX and PXC frame grabbers, **WaitVB** and **GetFieldCount** could often be used as general purpose timer and delay functions. This is particularly useful under MS-DOS where the system timer services are only accurate to 55 milliseconds.

Do not use this technique with the PXD series. Neither **WaitVB** nor **GetFieldCount** are supported for the PXD. Modern operating systems generally provide much better timing services –use them instead.

Waiting for a Queued Operation, while polling: It is often useful to wait for a frame grabber operation while occasionally checking for some outside event, such as the user pressing a “cancel” key. On previous frame grabbers this might be done by calling **IsFinished** inside a polling loop with **WaitVB** in the loop to yield processor time to other programs. **IsFinished** can still be used to poll for event completion, but **WaitVB** should not be used as a delay. If you need a delay while polling for frame grabber events, use **TimedWaitFinished** with a timeout of the desired delay.

Line Scan Cameras

The PXD frame grabbers have support for line scan cameras. These cameras output only a single row of pixels for each exposure. Because it is usually inefficient to have a separate frame for each line of data received from a line scan camera, the PXD drivers allow line scan data to be grabbed to a rectangular image buffer. The buffer will be filled from top to bottom with successive line exposures from the camera, and the Grab function will finish when all lines of the rectangular buffer have been captured. An application can use whatever height of frame provides the best solution for a particular problem. Taller frames require less system overhead, but shorter frames allow better response time between when the data for a line is received by the frame grabber and when the application can process it. Assuming that the average data transfer rate of the system is fast enough to keep up with the camera, it is possible to queue two or more frames so that no lines from the camera are lost between the bottom of one frame and the top of the next (the data that is received from the camera during the time when the driver is switching between the two frames is internally buffered by the frame grabber's FIFO, and will not be lost unless the FIFO overflows).

LUTs

The PXD series frame grabbers have hardware lookup tables on the data inputs. On the PXD1000, there are either two 16 to 16 tables or four 8 to 8 tables. (There is no 32 bit to 32 bit table, because this would contain an unwieldy number of entries.) Usually, it is best to use the 8 bit tables with 8 bit per pixel cameras, and the 16 bit per pixel tables with higher bit depth cameras, but there are some cases where other combinations may prove useful. A camera which produces 24 bit parallel RGB, for instance, can be used with the 8 bit LUTs to do a separate translation on each color channel.

The LUT functions allow reading and writing subranges of each LUT, and loading each tap's LUT independently. It is also possible to load all LUTs simultaneously by specifying -1 for the tap number.

Frame Status

Information about the most recent image grabbed to a given frame buffer can be retrieved using **GetLastFrame**. Information about the frame currently being captured can be retrieved using **GetActiveFrame**. The information returned includes the number of lines captured, a 6 bit frame identification number, and any error conditions which occurred during capture. Depending on the camera the lines captured for an incomplete image may or may not be sequential lines starting at the top of the image.

Resynchronization and Frame Dropping

For some high speed cameras, the PCI bus may not be fast enough to transfer all images from the camera in real time. In this case, the frame grabber will respond by occasionally dropping incoming video images so that the average frame rate is small enough for the PCI bus to handle. As long as each individual image is smaller than half the size of the PXD onboard buffer memory, the frame dropping mechanism will result in a regular pattern of dropped and captured frames which captures as many frames as possible with the available PCI bandwidth. If individual images are much larger than half of the buffer size, the frame grabber may begin to drop frames somewhat irregularly, and drop more frames than necessary. If the image size is larger than the onboard buffer, then the frame grabber can no longer guarantee reliable image capture, but it will still attempt to capture as much of the image as possible. All of these issues are only important at very high data rates; if the data rate of the camera is low enough for the PCI bus to transfer data in real time, images larger than the frame grabber's buffers will be captured correctly.

Frame Grabbing and PCI Bus Performance

Data transfers can take advantage of the maximum 132 MB per second burst transfer rate of the PCI bus. Although actual throughput is typically well below the maximum burst rate, a properly-designed system can support real-time transfer and display of at least 8-bit-per-pixel video image data. Actual throughput is affected by the PCI implementation on the motherboard, the design of the PCI video controller or other PCI device, and the load on the bus due to all PCI devices using it.

If the PCI bus is overloaded, it's possible for captured data to be corrupt. Although the Grab functions can't determine if data is being corrupted, **CheckError()** will return the value `ERR_CORRUPT`.

Camera Adjustment through RS-232

Many cameras are programmable using an RS-232 serial interface. We supply no direct support for this, as the PXD frame grabbers do not have RS-232 hardware onboard, and the PXD software does not use RS-232 for anything. It is generally relatively easy to use the operating system's services to program the camera from a serial port elsewhere in the system. Cables we design for cameras with RS-232 will generally have a branch which can be connected to an external serial port.

Manual Buffer Locking

When the frame grabber captures data to a buffer in system memory, that memory needs to be locked, meaning that the operating system has forced the memory to be present at a known location in physical RAM, and that the frame grabber's driver knows the address of the buffer and has updated some internal tables describing the memory. Usually, the applications programmer does not need to worry about whether a buffer is locked or not, since the driver will automatically lock and unlock memory whenever it needs to.

Occasionally, there are times when an application programmer may want to control buffer locking directly. The usual reason for this is that on a virtual memory system, locking a buffer may take an unpredictable amount of time because the operating system may choose to copy data to or from the swap file as part of locking the memory. This can cause grabs to occur later than expected, and may cause programs to miss important events.

For cases where manual locking of frame buffers can improve performance, the API provides the **LockFrame()** and **UnlockFrame()** functions. In general, these should only be used on computers where the programmer knows in advance which programs will be running, and the goal is to tune the entire system for optimum performance for a particular task. Locking frames manually on a computer where many other applications are running can actually reduce the performance of the whole system because it prevents the operating system from efficiently sharing memory between all of the running programs.

Scaling and Cropping

The PXD series does not support scaling or subsampling of the captured image, but does support cropping to capture a subrectangle. The region of the image to be captured can start and stop on any line, but each individual line must be aligned on DWORD boundaries. This means that 8 bit per pixel images can be cropped to an accuracy of 4 horizontal pixels, while 16 bit images can be cropped to any even number of pixels.

QualifyGrabs

The most common way to capture an image in response to an external event is to use **GrabTriggered**. For very high speed captures, however, **GrabTriggered** may be inadequate. When a triggered grab is performed, the frame grabber does not begin waiting for a trigger event until all previous queued operations have completed, and the frame grabber's onboard mem-

ory has emptied. This insures that data is captured correctly for the triggered grab, but loses many of the speed benefits of having the large onboard FIFO. For very fast cameras, there may be several milliseconds between when the camera finishes sending image data and when the data has completely emptied from the FIFO, and with **GrabTriggered**, a new trigger event cannot be accepted during this time.

QualifyGrabs provides a way to do repeated triggered grabs, where a new trigger can occur as soon as the camera has completed the previous image, or even slightly before.

When **QualifyGrabs** is called with a nonzero count, the frame grabber will assume that images are ALWAYS captured in response to a trigger event. The frame grabber will wait for a trigger event, optionally discard one or more incoming images to create a delay, and then attempt to capture one or more images. If a Grab or **GrabContinuous** command is active at this point, these images will be captured. If no Grab command is ready to take the image when the trigger sequences occur, the images are discarded.

GrabTriggered cannot be used when **QualifyGrabs** is active, because all images are already being captured in response to trigger events.

Some examples:

```
/* first describe what kind of trigger event we're interested
in */
pxd.SetTriggerSource(fgh, TRIGGER, LATCH_FALLING);

/* turn on QualifyGrabs. After this line, each trigger will
cause one frame to get captured */
pxd.QualifyGrabs(fgh, 0, 1);

/* each of these five grabs will now wait until a falling edge
occurs on the trigger, and capture the next image */
for (i=0; i<5; i++)
    pxd.Grab(fgh, frh, 0);

/* turn off QualifyGrabs */
pxd.QualifyGrabs(fgh, 0, 0);
```

```
/* these grabs capture any images that the camera supplies, as
usual. */
for (i=0;i<5;i++)
    pxd.Grab(fgh,frh,0);
```

Suppose there is a line scan camera looking at a conveyor belt, and there is a signal attached to GPIN0 which is TRUE whenever an object is passing under the camera. We want to get pictures of the objects, but don't want to digitize a lot of empty space between them. Here's a way to do it:

```
/* this will be used as a circular buffer which will hold the
last 1024 interesting lines */
frh=pxd.AllocateBuffer(LINEWIDTH,1024,PIX_TYPE);

/* set up trigger source and QualifyGrabs so that lines only
get through when GPIN0 is high */
pxd.SetTriggerSource(fgh,GPIN0,IO_INPUT_HIGH);
pxd.QualifyGrabs(fgh,0,1);

pxd.GrabContinuous(fgh,frh,-1,0);
/* at this point whenever GPIN0 is high, lines will be written
into our buffer, and whenever GPIN0 is low, the grabber will
stop and wait */
```

Monitoring Grab Status

There are three functions which collect information about image captures: **CheckError**, **GetLastFrame**, and **GetActiveFrame**.

GetActiveFrame allows the progress of an image capture to be monitored while the grab is still in progress. It reports the handle of the frame currently being captured, the most recently captured line number, the frame index number, and any error conditions which have occurred in this frame.

GetLastFrame returns the same information about the most recently completed image. The line number from **GetLastFrame** will usually be the last line of the image, unless the camera has been reset or an error which caused

the capture to abort occurred. If the image was not completely captured because of an error or resynchronization, the line number will give the last completed line of the image and the error flags will tell why the capture was aborted.

CheckError returns only error flags, not the line number or frame number. The error flags returned by **CheckError** represent a summary of everything that has gone wrong between the last time **CheckError** was called and the most recently completed capture. Error flags accumulate for each capture that occurs, and on each call to **CheckError**, the accumulated errors are all reported and the accumulator is cleared. **CheckError** is most useful in situations where a program needs to check occasionally for whether things are working correctly, but does not need to know exactly when failure occurred.

The line numbers returned by the status functions represents the number of horizontal syncs received from the camera, not the pixel coordinates in the image buffer. For single channel cameras, this distinction usually isn't important, since lines are received in order from the camera. For cameras with multiple channels, one line count in the status functions may represent two or even four lines of data in the image buffer, and these lines may not all be at the top of the image.

For example, the Kodak ES 1.0 camera has two channels, which read out pairs of lines simultaneously. For this camera, each line count in the status represents a pair of lines in the image which were written together.

When using a line scan camera, the line count always represents the number of single lines which have been captured into the image buffer.

The frame index number is a six bit counter which counts the number of vertical syncs received from the camera. There is no way to set this counter to a particular value, but values for two grabs can be meaningfully compared. For example, the following code determines whether two grabs capture sequential frames, or not.

```

pxd.Grab(fgh, frh, 0);
pxd.GetLastFrame(fgh, &frame1, &line1, &count1, &errs1);
pxd.Grab(fgh, frh, 0);
pxd.GetLastFrame(fgh, &frame2, &line2, &count2, &errs2);
if (count2==count1+1) {
    printf("frames were sequential\n");
}
else {
    printf("%d frames skipped\n", count2-count1-1);
}

```

When using status functions with lists of frames, the frame handle returned will be the handle of the entire list, not the handle of an individual sub-frame. The line count will represent the number of lines in the entire list as well. For example, in a list of 10 frames, each with 20 lines, the line counter will range between 0 and 199, with 0..19 representing the first image, 20..39 representing the second, and so forth. The frame index will, as usual, increase for each actual image received from the camera, resulting in 10 different index numbers at various points in the list.

The combination of frame lists and status functions can be used to do double buffered captures with less overhead than multiple queued grabs.

For example:

```

frh=pxd.AllocateBufferList(WIDTH, HEIGHT, TYPE, 2);
pxd.GrabContinuous(fgh, frh, 0);

/* the frame grabber is now continuously updating the two
buffers in the list */
while (1) {
    pxd.GetActiveFrame(fgh, &frame, &line, &count, &errs);
    if (line>=HEIGHT) {
        //process subframe 0 because subframe 1 is being updated
    }
    else {
        //process subframe 1 because subframe 0 is being updated
    }
}

```

Chapter 6

PXD Library

This chapter is a complete, alphabetical function reference for the PXD1000 Frame Grabber Libraries and DLLs. For additional information on using the functions, see Chapter 5, *Developing Applications with the PXD1000*. For reference information on the Frame Library, see Chapter 7.

This function reference is a general guide for using the functions with all operating systems. The functions will work as written for C and C++ with the header files provided.

If you need to construct your own header file, you will need to know the definitions of constants and the sizes of the parameters and the return values for the function calls. You can find the definitions of constants in the header files for C. The following table gives the sizes of the various data types that are used by the PXD1000 library.

| Type | Size |
|--|-------------|
| unsigned char | 8 bits |
| long, unsigned long | 32 bits |
| void *, unsigned char *, int *, char *, LPSTR | 32 bits |
| short | 16 bits |

FRAME and FRAMELIB are defined types; to see how they are defined, refer to the C language header file for the appropriate operating system. Void is a special type. When it is the type for a parameter, the function has no parameters; when it is the type for the return value, the function does not return a value.

AllocateBuffer

```
FRAME* AllocateBuffer (short dx, short dy, short type) ;
```

Return Value

handle to the allocated FRAME structure if successful; 0 on failure

Parameters

dx

width of requested image buffer in pixels. *dx* must be greater than zero.

dy

height of requested image buffer in pixels. *dy* must be greater than zero.

type

data type of pixels that will be stored in the requested image buffer. Valid pixel types are defined in frame.h and listed below:

| | |
|-------------|---|
| PBITS_Y8 | 8-bit gray scale stored in an unsigned char |
| PBITS_Y16 | 16-bit gray scale stored in an unsigned short |
| PBITS_RGB32 | 8/8/8 Red/Green/Blue color stored in the lower 24 bits of a long. This is only valid if a color camera is |

Description

AllocateBuffer allocates memory for an image buffer of size *dx* by *dy* with the specified pixel data *type* from the system memory heap. The start of the buffer will be aligned on a 4-byte boundary. Most often, the frame grabber will use direct memory access (DMA) to directly move an image into this buffer so, for the buffer to be usable by the frame grabber, *dx* and *dy* must be at least as large as the image being grabbed. **FreeFrame** should be used to release the image buffer when it is no longer needed.

When sufficient real memory (RAM) is not available, **AllocateBuffer** will cause storage to be allocated from virtual memory. This can result in extremely slow application performance. **AllocateBuffer** will fail when sufficient memory (of any type) is not available.

Example: AllocateBuffer

```
/* AllocateBuffer Example */

#include "pxd.h"
#include "frame_32.h"
#define IMAGEWIDTH 1024 /* the image width from your camera here */
#define IMAGEHEIGHT 1024 /* the image height from your camera here */

main ()
{
    PXD      pPxd ;          /* a pointer to the pxd library */
    FRAMELIB  pFrameLib ;   /* a pointer to the frame library */
    long      hFG ;         /* a handle to a frame grabber */
    FRAME*    pFrame ;     /* a pointer to a image buffer */

    /* get access to the libraries */
    imagenation_OpenLibrary ("pxd.dll", &pPxd, sizeof (PXD)) ;
    imagenation_OpenLibrary ("frame_32.dll", &pFrameLib, sizeof (FRAMELIB)) ;

    /* allocate an image buffer */
    pFrame = pxd.AllocateBuffer (IMAGEWIDTH, IMAGEHEIGHT, PBITS Y16) ;

    /* get access to a frame grabber */
    hFG = pxd.AllocateFG (-1) ;

    /* capture an image and move it to the image buffer */
    pxd.Grab (hFG, pFrame, 0) ;

    /* do your image processing here */

    /* release the image buffer */
    frameLib.freeFrame (pFrame) ;

    /* release the frame grabber */
    pxd.FreeFG (hFG) ;

    return ;
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

AllocateAddress (not WinNT) (frame library) to allocate an alternate buffer

AllocateBufferList (pxd library) to allocate a list of buffers

AllocateFlatFrame (not WinNT) (frame library) to allocate an alternate buffer

AllocateFG (pxd library) to allocate a frame grabber handle

AllocateMemoryFrame (frame library) to allocate an alternate buffer

FreeBuffer (frame library) to free the buffer

AllocateBufferList

```
FRAME* AllocateBufferList(short x, short y,  
short type, short n);
```

Return Value

a pointer to the allocated FRAME structure if successful; 0 on failure.

Parameters

dx

width of requested image buffer in pixels. *dx* must be greater than zero.

dy

height of requested image buffer in pixels. *dy* must be greater than zero.

type

data type of pixels that will be stored in the requested image buffer. Valid pixel types are defined in frame.h. The most common ones used with digital cameras are listed below:

| | |
|-------------|---|
| PBITS_Y8 | 8-bit gray scale stored in an <i>unsigned char</i> |
| PBITS_Y16 | 16-bit gray scale stored in an <i>unsigned short</i> |
| PBITS_RGB32 | 8/8/8 Red/Green/Blue color stored in the lower 24 bits of a <i>long</i> |

n

the number of buffers to be allocated

Description

AllocateBufferList creates a list of *n* identical frames, stored as planes of a master frame. Grabs to the master frame fill up all elements of the list in order. This is done to allow cameras with very high frame rates to run without the overhead of a new queue operation for each image. Although the FRAME structure supports the notion of arbitrary trees of frames, **AllocateBufferList** does not create trees more than one level deep. Also the library will not recurse more than one level into a frame list (this is to avoid problems with malformed trees that have cycles in them). To access a particular frame in the Buffer List, use **ExtractPlane**.

Example: AllocateBufferList

```

/* AllocateBufferList Example */

#include "pxd.h"
#include "frame_32.h"
#define IMAGEWIDTH 1024 /* the image width from your camera here */
#define IMAGEHEIGHT 1024 /* the image height from your camera here */

main ()
{
    PXD      pPxd ;          /* a pointer to the pxd library */
    FRAMELIB pFrameLib ;    /* a pointer to the frame library */
    long     hFG ;          /* a handle to a frame grabber */
    FRAME*   pFrame ;       /* a pointer to a image buffer */

    /* get access to the libraries */
    imagenation_OpenLibrary ("pxd.dll", &pPxd, sizeof (PXD)) ;
    imagenation_OpenLibrary ("frame_32.dll", &pFrameLib, sizeof (FRAMELIB)) ;

    /* allocate image buffer list of 5 buffers */
    pFrame = pxd.AllocateBufferList (IMAGEWIDTH, IMAGEHEIGHT, PBITS_Y16, 5) ;

    /* get access to a frame grabber */
    hFG = pxd.AllocateFG (-1) ;

    /* capture 5 images with a single grab, since we allocated a list of 5
    buffers
    */
    pxd.Grab (hFG, pFrame, 0) ;

    /* do your image processing here */

    /* release the image buffer */
    frameLib.freeFrame (pFrame) ;

    /* release the frame grabber */
    pxd.FreeFG (hFG) ;

    return ;
}

```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

AllocateAddress (not WinNT) (frame library) to allocate an alternate buffer

AllocateBuffer (pxd library) to allocate a buffer

AllocateFlatFrame (not WinNT) (frame library) to allocate an alternate buffer

AllocateFG (pxd library) to allocate a frame grabber

AllocateMemoryFrame (frame library) to allocate an alternate buffer

FreeBuffer (pxd library) to free the buffer list

Extract Plane (frame library) to access each frame in the list

AllocateFG

```
long AllocateFG (short n) ;
```

Return Value

a handle to the requested frame grabber if successful; otherwise NULL

Parameters

n

the number of the requested PXD1000 frame grabber; -1 requests the first frame grabber found

Description

Use **AllocateFG** to gain access to a frame grabber. The access will be exclusive, i.e. no other program or processes will be able to gain access to an allocated frame grabber until it has been released by the program or process that captured it. You can use a frame grabber number of 0 (zero) in situations where a single PXD frame grabber is installed. A frame grabber number of -1 will return a handle to the first grabber found. If a system has more than one frame grabber installed you can access any one by specifying its number *n* in **AllocateFG**. (see "What is the number of my frame grabber?" at left). If the requested frame grabber is available, **AllocateFG** returns a handle that must be used with other library functions that refer to the frame grabber.

Your program should call **FreeFG** on the frame grabber when it is no longer needed.

AllocateFG will fail if another application has captured the frame grabber. It may also fail if another application has failed to free the frame grabber when finished (or if the application locks).

What is my frame grabber number?

If you have more than one PXD frame grabber installed you will need to know the number of the frame grabber you want to use in order to use **AllocateFG**. When the PXD driver is loaded it searches through the PCI slots looking for PXD frame grabbers. It builds a table of the PXD1000 frame grabbers it finds. When you call `imagegen_OpenLibrary` to open the `pxd.dll`, the value returned is the number of PXD1000 boards in the system. Regardless of the actual slot each board is installed in, the boards are numbered 0 through *m*, where *m* could be as great as the number of PCI slots in your computer minus 1 (if you had filled all the slots with PXD1000 cards). To access a particular PXD1000 card, use the number of the frame grabber. The card installed in the lowest numbered PCI slot will have number 0. If you are unsure how the PCI slots in your computer are numbered, the easiest method is to attach a video source to only a single PXD1000 and use `PXView` to determine which card is receiving the active video.

Example: AllocateFG

```
/* AllocateBuffer Example */

#include "pxd.h"
#include "frame.h"

#define IMAGEWIDTH 1024 /* the image width from your camera here */
#define IMAGEHEIGHT 1024 /* the image height from your camera here */

main ()
{
    PXD    pxd ;          /* a pointer to the pxd library */
    FRAMELIB    framelib ; /* a pointer to the frame library */
    long    hfg ;        /* a handle to a frame grabber */
    FRAME*    pFrame ;   /* a pointer to a image buffer */
    Long    cFG ;       /* number of frame grabbers installed */

    /* open the pxd.dll and find out how many frame grabbers are installed */
    cFG = imagenation OpenLibrary ("pxd.dll", &pxd, sizeof (pxd)) ;

    /* open the frame library */
    imagenation_OpenLibrary ("frame_32.dll", &frameLib, sizeof (FRAMELIB)) ;

    /* allocate an image buffer */
    pFrame = pxd.AllocateBuffer (IMAGEWIDTH, IMAGEHEIGHT, PBITS Y16)) ;

    /* get access to the last frame grabber found */
    hfg = pxd.AllocateFG (cFG - 1) ;

    /* capture an image and move it to the image buffer */
    pxd.Grab (hfg, pFrame, 0) ;

    /* do your image processing here */

    /* release the image buffer */
    frameLib.freeFrame (pFrame) ;

    /* release the frame grabber */
    pxd.FreeFG (hfg) ;

    return ;
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

AllocateAddress (not WinNT) (frame library) to allocate an alternate buffer

AllocateBuffer (pxd library) to allocate a frame buffer

AllocateBufferList (pxd library) to allocate a buffer list

AllocateFlatFrame (not WinNT) (frame library) to allocate an alternate buffer

AllocateMemoryFrame (frame library) to allocate an alternate buffer

FreeFG (pxd Library) – to release a frame grabber

CheckError

```
long CheckError (long hFG) ;
```

Return Value

0 if no errors have occurred; 1 if the frame grabber handle is invalid; or one or more of the following bit-flags if an error has occurred:

| | |
|-------------------|---|
| ERR_CORRUPT | the frame grabber transferred a partial frame because it could not recover from FIFO overflow. This can only happen for frames larger than ~1.5 Megs. |
| ERR_DROPPED_FRAME | the frame grabber dropped an entire frame to make room in the FIFO. |
| ERR_RESYNC | the image from the camera was never completed. This usually means an asynchronous reset of the camera occurred. |
| ERR_LOST_TRIGGER | a second trigger pulse was detected before a GrabTriggered was finished, and the second pulse was discarded |
| ERR_NOT_VALID | hFG is not a valid frame grabber handle. |
| ERR_HW_FAIL | the frame grabber has gotten into an unstable state, most likely because of bad connections or noise in the camera cable. The frame grabber will resynchronize by discarding all data in it's FIFO, marking all currently queued grabs as corrupt and turning off continuous acquire. |

Parameters

hFG

a handle to the frame grabber that you are requesting information about

Description

CheckError returns a long integer with individual bits-flags set for each type of error that has occurred in the frame grabber since the last call to **CheckError**, **Reset** or **AllocateFG**.

Three functions, **CheckError**, **AllocateFG** and **Reset** will reset all the error flags to the no-error (FALSE) state.

GetLastFrame and **GetCurrentFrame** are similar to **CheckError**. Use **GetLastFrame** to see just the errors that occurred on the most recently completed frame. **GetCurrentFrame** reports the errors that have occurred in capturing the current (but not completed) frame.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetLastFrame (pxd library) to identify the last frame

GetActiveFrame (pxd library) to identify the active frame

CloseLibrary

```
void imagenation_CloseLibrary (PXD* pxd) ;
```

Return Value

none

Parameters

pxd

a pointer to a PXD interface structure

Description

CloseLibrary returns to the system any resources that were allocated by **OpenLibrary**. For DOS4GW applications this means unhooking the interrupt that responds to library calls. In Windows a call to **CloseLibrary** releases the handle to `pxd.dll`, which means that if your application is the only one using the library, the library will remove itself from memory.

CloseLibrary should be the last library function called by your program. A program that exits after calling **OpenLibrary**, but before calling **CloseLibrary**, will leave the computer in an unstable state that may cause the operating system to fail.

Example

See the `Capture1` sample in the `Samples` folder for an example of how to use the **OpenLibrary** and **CloseLibrary** functions.

Required Compile-time Headers

`pxd.h` for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

`ilib_32.lib` for Windows NT, 98 and 95

`pxd_fw.lib` for DOS4GW

Required Run-time Libraries

`pxd_32.dll` for Windows NT, 98 and 95

Also See

OpenLibrary (pxd library)

ContinuousStrobes

```
short ContinuousStrobes (long hFG, short iMode) ;
```

Return Value

Non-zero if successful; 0 on failure

Parameters

hFG

a handle to a frame grabber

iMode

selects the strobe mode:

any non-zero value

the preprogrammed strobe sequences repeat forever

0

strobe sequences will execute once for each trigger

Description

If *iMode* is non-zero the strobes will begin their programmed pulse sequences and repeatedly start again as soon as both strobes' pulse sequences are complete. If *iMode* is 0, continuous strobing will stop as soon as the current strobe sequences finish. If continuous strobing is active **TriggerStrobes** and **FireStrobes** have no effect.

The PXD supports two strobe lines. Both of these lines can be programmed to produce up to four pulses, separated by gaps. The pulse-length and gap length are each programmable, from about 1 microsecond to about 3 seconds, accurate to 0.5 microseconds, or about 0.1%

The strobes can be programmed with the `SetStrobePeriods()` function.

The default strobes can be programmed with the PXD Configuration application, in the DEFAULT.CAM.

Required Compile-time Headers

`pxd.h` for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

`ilib_32.lib` for Windows NT, 98 and 95

`pxd_fw.lib` for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SetStrobePeriods (pxd library) to program strobe pulse-sequences

TriggerStrobes (pxd library) to setup hardware triggered strobes

WriteImmediateIO (pxd library) to set the polarity of the strobe outputs

FireStrobes

```
short FireStrobes (long fgh) ;
```

Return Value

Non-zero if successful; 0 on failure

Parameters

fgh

a handle to a frame grabber

Description

The PXD1000 contains two strobe generators, STROBE_0 and STROBE_1 each capable of stepping through a unique pulse-sequence. While the strobes can be programmed individually, they are always fired by the same event. **FireStrobes** is a software-initiated one-time event that causes both strobe lines, on the frame grabber identified by *hFG*, to immediately begin stepping through their preprogrammed pulse sequence.

Hardware events can also be used to fire the strobe outputs. You can use **TriggerStrobes** to specify which hardware event will fire the strobes.

Use **SetStrobePeriods** to define the pulse sequence for each strobe generator. The output polarity of each strobe is programmed with **WriteImmediateIO**.

The default strobe periods are programmed into the configuration file with the PXDConfig application.

FireStrobes will succeed in triggering the start of a new strobe sequence only if the strobe generators are not currently stepping through a strobe sequence. If the strobe generators are in the process of stepping through a sequence, the software trigger generated by FireStrobe will be ignored and FireStrobe will return an error (a 0 value).

FireStrobe will also fail if hFG is not a valid handle of a frame grabber.

Example

See the **FireStro** sample in the **Samples** folder for a detailed example using the PXD1000 strobes

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SetStrobePeriods (pxd library) to program strobe pulse-sequences

TriggerStrobes (pxd library) to setup hardware triggered strobes

WriteImmediateIO (pxd library) to set the polarity of the strobe outputs

FreeConfig

```
void FreeConfig (CAMERA_TYPE *configInMem) ;
```

Return Value

none

Parameters

configInMem

a pointer to a camera configuration structure

Description

FreeConfig releases the data structures created by **LoadConfig** when the data is no longer needed. Do not call **FreeConfig** on a CAMERA_TYPE structure that was not created by **LoadConfig**.

Example: Free Config

```
/* FreeConfig Example - loads camera configuration structure into memory */
CAMERA_TYPE *configInMem          /* create a pointer to a camera
                                configuration data structure */

char configFile [] = "Kodak14.CAM" ; /* a camera configuration file */

configInMem = LoadConfig (configFile) ; /* load structure from file */

/* use the camera configuration here */

FreeConfig (configInMem) ; /* done, release the pointer */
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SaveConfig (pxd library) saves a camera configuration structure to a file

LoadConfig (pxd library) loads a camera configuration structure from a file

SetCameraConfig (pxd library) load a PXD frame grabber with a camera configuration

FreeFG

```
void FreeFG (long hFG) ;
```

Return Value

none

Parameters

hFG

a handle to a frame grabber

Description

FreeFG releases control of a frame grabber previously allocated with **AllocateFG**. It is recommended that you always call **FreeFG** as soon as your program has finished using the frame grabber. If your process completes without calling **FreeFG** the dangling frame grabber handles will be properly cleaned up for you.

Example

See the **AllocateFG** function for an example.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

AllocateFG (pxd library) to allocate the frame grabber handle

FreeFrame

```
void FreeFrame (FRAME *pFrame) ;
```

Return Value

none

Parameters

pFrame

a pointer to a frame (image buffer) in memory

Description

FreeFrame returns memory associated with the *pFrame* to the system. You must free all frames allocated by **AllocateBuffer** before calling **CloseLibrary**. This function is identical to **FreeFrame** in the frame library. Either version of the function can free a frame allocated by either library.

Example

```
void ShutDownFrameGrabber(PXD * pPxd, long fgh, FRAME * pfrh)
{
    // kill the queue, so that the frames are no longer in use
    pPxd->KillQueue(fgh);

    // free the frame handle, since we are no longer grabbing into it
    pPxd->FreeFrame(pfrh);

    // we can free the frame grabber
    pPxd->FreeFG(fgh);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

FreeFrame (frame library) which can also be used to free a frame buffer

GetActiveFrame

```
short GetActiveFrame (long hFG,  
                     FRAME **hFrame,  
                     short *pLine,  
                     short *pCount,  
                     long *pErrors) ;
```

Return Value

non-zero if successful; 0 if unsuccessful

Parameters

hFG

a handle to a frame grabber

hFrame

the handle of the image buffer where the current data is being written

pLine

points to a short that contains the line number of the last line of contiguous of data written to the image buffer (see below)

pCount

points to a short that contains a local frame index number (see below)

pErrors

points to a long that contains the bit-flags for any errors generated in this frame, the bit-flags are:

| | |
|-------------------|---|
| ERR_CORRUPT | the frame grabber transferred a partial frame because it could not recover from FIFO overflow. This can only happen for frames larger than ~1.5 Megs. |
| ERR_DROPPED_FRAME | the frame grabber dropped an entire frame to make room in the FIFO. |
| ERR_RESYNC | the image from the camera was never completed. This usually means an asynchronous reset of the camera occurred. |
| ERR_LOST_TRIGGER | a second trigger pulse was detected before a GrabTriggered was finished, and the second pulse was discarded |

| | |
|---------------|---|
| ERR_NOT_VALID | <i>hFG</i> is not a valid frame grabber handle. |
| ERR_HW_FAIL | the frame grabber has gotten into an unstable state, most likely because of bad connections or noise in the camera cable. The frame grabber will resynchronize by discarding all data in it's FIFO, marking all currently queued grabs as corrupt and turning off continuous acquire. |

Description

GetActiveFrame returns the status of the frame currently being grabbed as part of a **Grab** or **GrabTriggered** operation. If no **Grab** or **GrabTriggered** operation is in progress it returns NULL for *hFrame* and the last valid values for *pLine* and *pCount*. The *pLine* value returned in line is the largest value such that that *pLine* and all previous lines have been written to memory. There may be a few lines of data written out of chronological order, so do not assume that no data has been written to any lines after the one specified. The *pLine* value is a chronological count of scan lines coming from the camera, not the Y offset of the scan line in the image buffer. For example, if a camera scans images from bottom to top, a line value of 5 means that the 6 bottommost lines of the image are done. If a grab has begun, but no lines have been written yet, *pLine* = -1.

**pCount* is the current value of the local frame counter in the PXD1000. The local frame counter is a 6-bit counter that increments on the rising edge of each frame data valid signal (FDV) from the camera.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

CheckError (PXD library) returns accumulated error status since the last CheckError, Reset or AllocateFG

GetLastFrame (PXD library) returns the status from the most recently completed grab

GetBrightness

```
float GetBrightness(long fgh);
```

Return Value

0.0

Description

This function is for compatibility only, and always returns "0.0"

GetCamera

```
short GetCamera(long fgh);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0

GetContrast

```
float GetContrast(long fgh);
```

Return Value

1.0

Description

This function is for compatibility only, and always returns "1.0".

GetExposureTime

```
float GetExposureTime (CAMERA_TYPE *pCamera) ;
```

Return Value

The current exposure time in seconds

Parameters

pCamera

a pointer to a camera configuration structure

Description

GetExposureTime returns the value currently stored in the *exposure_time* field of the camera configuration structure pointed to by *pCamera*. **GetExposureTime** is one of a small set of functions that provide access to specific fields in a camera configuration structure.

The default exposure time is programmed into the default configuration file (DEFAULT.CAM) with the PXD Configuration application.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SetExposureTime (PXD library) sets the *exposure_time* field in a camera configuration structure

GetPixelType (PXD library) reads the *pix_type* field from a camera configuration structure

GetFramePeriod (PXD library) reads the *frame_period* field from a camera configuration structure

SetFramePeriod (PXD library) which sets the *frame_period* field in a camera configuration structure

GetFieldCount

```
long GetFieldCount(long fgh);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

GetFramePeriod

```
float GetFramePeriod (CAMERA_TYPE *pCamera) ;
```

Return Value

the value stored in the *frame_period* field of the camera definition structure pointed to by *pCamera*

Parameters

pCamera
a pointer to a camera definition structure

Description

GetFramePeriod returns the time in seconds currently stored in the *frame_period* field of the camera definition structure pointed to by *pCamera*. **GetFramePeriod** is one of a small set of functions that provide access to specific fields in a camera definition structure.

The default frame period is programmed into the default configuration file (DEFAULT.CAM) with the PXD Configuration application.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SetExposureTime (PXD library) sets the *exposure_time* field in a camera definition structure

GetExposureTime (PXD library) reads the *exposure_time* field from a camera definition structure

GetPixelType (PXD library) reads the *pix_type* field from a camera definition structure

SetFramePeriod (PXD library) – which sets the *frame_period* field in a camera definition structure

GetHeight

```
short GetHeight (long hFG) ;
```

Return Value

the height, in pixels, of the cropping region if successful; 0 if unsuccessful

Parameters

hFG

a handle to a frame grabber

Description

Often, only a portion of the image produced by the camera contains information of interest. Capturing only the area of interest, called the cropping region, instead of the entire image can reduce system memory and PCI bus bandwidth requirements. The pixel coordinates of the left edge and top along with the width and height define the cropping region. **GetHeight** returns the width of the currently selected cropping region.

This function executes immediately regardless of the state of the frame grabber queue.

GetHeight will fail if the handle to the frame grabber, *hFG* is invalid.

Example: GetHeight

```
Void CropToCenterQuarter(PXD * pPxd, long fgh)
{
    short sWidth;
    short sHeight;

    sWidth = pPxd->GetWidth(fgh);
    sHeight = pPxd->GetHeight(fgh);

    pPxd->SetLeft(fgh, sWidth/4);
    pPxd->SetTop(fgh, sHeight/4);
    pPxd->SetWidth(fgh, sWidth/2);
    pPxd->SetHeight(fgh, sHeight/2);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetLeft (PXD library) – returns the left edge of the cropping region

GetTop (PXD library) – returns the top edge of the cropping region

GetWidth (PXD library) – returns the width of the cropping region

GetXResolution (PXD library) – returns the width of the uncropped image

GetYResolution (PXD library) – returns the height of the uncropped image

SetHeight (PXD library) – sets the height of the cropping region

SetLeft (PXD library) – set the left edge of the cropping region

SetTop (PXD library) – sets the top edge of the cropping region

SetWidth (PXD library) – sets the width of the cropping region

GetInputLUT

```
short GetInputLUT(long fgh,short bits,short LUT,long start,long len,void *data);
```

Return Value

Returns non-zero on success, 0 on failure

Parameters

fgh

handle to a frame grabber

bits

the number of bits in the LUT entry. This is either 8 or 16.

LUT

the LUT from which to read

start

the starting entry to read

len

the number of entries to read from the LUT

data

address of the buffer to receive the LUT entries. It must be large enough to receive *len* entries with *bits* per entry.

Description

Reads any of the frame grabber's input lookup tables, or a section of it, into the array *data*. The section to be read is specified by *start* and the number of entries (*length*) desired. The buffer must have at least *length* entries. If the frame grabber is processing queued operations, GetInputLUT() waits for the operations to finish before executing.

In 8 bit modes, the LUT's are 256 bytes long. In 16 bit modes, the LUT's are 65,536 entries long. The PXD does not support 32 bit LUT's.

The LUTs which are applicable, depending upon whether the camera has a single channel, two channels, or 4 channels:

camera has a single channel LUT 0 LUT 1 LUT 2 LUT 3 8 bpp No Yes No Yes 16
bpp Yes No No No

camera has two channels LUT 0 LUT 1 LUT 2 LUT 3 8 bpp No Yes No Yes 16
 bpp Yes Yes No No

camera has four channels LUT 0 LUT 1 LUT 2 LUT 3 8 bpp Yes Yes Yes Yes 16
 bpp No No No No

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SetInputLUT (PXD library) sets the LUT

LoadConfig (PXD library) reads the configuration file, to identify the number of channels

GetInterface

```
const void *GetInterface(long handle);  
#define GetInterface(h) (*(void * *) (h))
```

Return Value

Pointer to the interface structure for a given frame grabber handle.

Parameters

handle

handle to a frame grabber

Description

A C macro that returns a pointer to the interface structure for a given frame grabber handle. You should assume that the structure pointed to is read-only. It is your responsibility to know what type of object is represented by handle and to cast the Return Value to the correct type. Be sure the handle is valid, since this macro is not good at error detection. This macro is intended for advance users who want to write complicated device-independent code.

Example: GetInterface

```
void GenericReset(void * fgh)  
{  
    /*  
     * at this point, we don't know whether this frame grabber handle is  
     * for a PXD, a PXC, or a new type. But, we can get the correct interface  
     * from the frame grabber directly. For safety, we will assume  
     * that the only functions available are the common ones. Just to  
     * get a prototype, we will treat the function as a PXD.  
     */  
    PXD * pPxx;  
  
    pPxx = GetInterface(fgh);  
    pPxx->Reset(fgh);  
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

AllocateFG (PXD library) to allocate a frame grabber

GetIOType

```
short GetIOType (long fgh,short n);
```

Return Value

Type of I/O line *n* if successful; 0 on failure.

| | |
|-----------|---|
| IO_INPUT | The state of the line is equal to the signal value. |
| IO_OUTPUT | The line is an output line. |

Parameters

fgh
handle to a frame grabber

n
the line number

Description

Returns the type of I/O line number *n*, where $0 \leq n < 15$, and the type is one of the following:

I/O Line Types

| Name | Value | Description |
|-----------|-------|------------------------------------|
| IO_INPUT | 4 | signal can be read but not written |
| IO_OUTPUT | 5 | signal can be written and read |

The I/O lines on the PXD1000 are numbered as follows:

| Name | I/O number | Types |
|-----------|------------|-----------|
| Trigger | 0 | IO_INPUT |
| WEN | 1 | IO_INPUT |
| FIELD | 2 | IO_INPUT |
| LDV | 3 | IO_INPUT |
| FDV | 4 | IO_INPUT |
| HDrive | 5 | IO_INPUT |
| VDrive | 6 | IO_INPUT |
| TTL In 0 | 7 | IO_INPUT |
| TTL In 1 | 8 | IO_INPUT |
| Strobe 0 | 9 | IO_OUTPUT |
| Strobe 1 | 10 | IO_OUTPUT |
| Control 0 | 11 | IO_OUTPUT |
| Control 1 | 12 | IO_OUTPUT |
| Control 2 | 13 | IO_OUTPUT |
| TTL Out 0 | 14 | IO_OUTPUT |
| TTL Out 1 | 15 | IO_OUTPUT |

Example: GetIOType

```
void ShowLineTypes(PXD * pPxd, long fgh)
{
    short sCounter;
    char szBuffer[256];
    short sType;

    szBuffer[0] = '\\0';
    for (sCounter = 0; sCounter < 16; sCounter++){
        sType = pPxd->GetIOType(fgh, sCounter);
        if (sType == 0)    {
            sprintf(strlen(szBuffer), "%2d : Unknown\n", sCounter);
        } else if (sType == IO_INPUT){
            sprintf(strlen(szBuffer), "%2d : Input\n", sCounter);
        } else {
            sprintf(strlen(szBuffer), "%2d : Output\n", sCounter);
        }
    }
    MessageBox(NULL, szBuffer, "Line Types", MB_OK);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

ReadIO (PXD library) to read an IO line

WriteImmedateIO (PXD library) set the IO line

GetLastFrame

```
short GetLastFrame (long hFG,
                   FRAME **hFrame,
                   short *pLine,
                   short *pCount,
                   long *pErrors) ;
```

Return Value

non-zero if successful; 0 if unsuccessful

Parameters

hFG

a handle to a frame grabber

hFrame

the handle of the image buffer where the most recently completed frame was written

pLine

points to a long that contains the number of valid lines written to the last frame

pCount

points to a short that contains the local frame index number of the last frame (see below)

pErrors

points to a long that contains the bit-flags for any errors generated in this frame, the bit-flags are:

| | |
|-------------------|--|
| ERR_CORRUPT | the frame grabber transferred a partial frame because it could not recover from FIFO overflow. This can only happen for frames larger than ~ 1.5 Megs. |
| ERR_DROPPED_FRAME | the frame grabber dropped an entire frame to make room in the FIFO. |
| ERR_RESYNC | the image from the camera was never completed. This usually means an asynchronous reset of the camera occurred. |
| ERR_LOST_TRIGGER | a second trigger pulse was detected before a GrabTriggered was finished, and the second pulse was discarded |

| | |
|---------------|---|
| ERR_NOT_VALID | <i>hFG</i> is not a valid frame grabber handle. |
| ERR_HW_FAIL | the frame grabber has gotten into an unstable state, most likely because of bad connections or noise in the camera cable. The frame grabber will resynchronize by discarding all data in it's FIFO, marking all currently queued grabs as corrupt and turning off continuous acquire. |

Description

GetLastFrame returns the status of the most recently completed frame grabbed as part of a **Grab** or **GrabTriggered** operation.

**pCount* was the value of the local frame counter in the PXD1000 when the grab was in progress. The local frame counter is a 6-bit counter that increments on the rising edge of each frame data valid signal (FDV) from the camera.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

CheckError (PXD library) returns accumulated error status since the last **CheckError**, **Reset** or **AllocateFG**

GetActiveFrame (PXD library) returns the status of the current **Grab** or **GrabTriggered** operation.

GetLeft

```
short GetLeft (long hFG) ;
```

Return Value

the X coordinate of the pixel at the left edge of the cropping region if successful;
0 if unsuccessful

Parameters

hFG
a handle to a frame grabber

Description

Often, only a portion of the image produced by the camera contains information of interest. Capturing only the area of interest, called the cropping region, instead of the entire image can reduce system memory and PCI bus bandwidth requirements. The pixel coordinates of the left edge and top along with the width and height define the cropping region. **GetLeft** returns the X coordinate of the pixels at the left edge of the currently selected cropping region.

This function executes immediately regardless of the state of the frame grabber queue.

GetLeft will fail if the handle to the frame grabber, *hFG* is invalid.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetHeight (PXD library) returns the height of the cropping region

GetTop (PXD library) returns the top edge of the cropping region

GetWidth (PXD library) returns the width of the cropping region

GetXResolution (PXD library) returns the width of the uncropped image

GetYResolution (PXD library) returns the height of the uncropped image

SetHeight (PXD library) sets the height of the cropping region

SetLeft (PXD library) set the left edge of the cropping region

SetTop (PXD library) sets the top edge of the cropping region

SetWidth (PXD library) sets the width of the cropping region

GetModelNumber

```
short GetModelNumber(long fgh);
```

Return Value

The model number of the PXD.

Parameters

fgh
handle to the frame grabber

Description

Returns the model number of the frame grabber, or 0 if *fgh* is not a valid handle. Currently the only possible valid return is 1000, indicating the PXD1000.

Example: GetModelNumber

```
void IdentifyGrabber(PXD * pPxd, long fgh)
{
    char szBuffer[256];

    // note: the following command builds a string by using the
    // C syntax of "aaa"bbb" gets treated by the compiler as "aaabbb"
    sprintf(szBuffer, "Protection key : %d\n"
        "Serial Number : %d\n"
        "Hardware Revision: %08X\n"
        "Software Revision: %08X\n"
        "PXD Model : %d\n",
        pPxd->ReadProtection(fgh),
        pPxd->ReadSerial(fgh),
        pPxd->ReadRevision(fgh),
        pPxd->ReadRevision(0))
    pPxd->GetModelNumber(fgh);
    MessageBox(NULL, szBuffer, "Grabber Identity", MB_OK);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

- **SetInputLUT** (PXD library) sets the LUT
- **LoadConfig** (PXD library) reads the configuration file, to identify the number of channels

GetPixelType

```
short GetPixelType (CAMERA_TYPE *pCamera) ;
```

Return Value

the value stored in the *pix_type* field of the camera configuration structure pointed to by *pCamera*, pixel types are:

| | |
|-------------|---|
| PBITS_Y8 | 8-bit gray scale stored in an <i>unsigned char</i> |
| PBITS_Y16 | 16-bit gray scale stored in an <i>unsigned short</i> |
| PBITS_RGB32 | 8/8/8 Red/Green/Blue color stored in the lower 24 bits of a <i>long</i> |

Parameters

pCamera

a pointer to a camera configuration structure

Description

GetPixelType returns the value currently stored in the *pix_type* field of the camera configuration structure pointed to by *pCamera*. **GetPixelType** is one of a small set of functions that provide access to specific fields in a camera configuration structure.

The pixel type can be programmed into a camera configuration file with the PXD Configuration application.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SetExposureTime (PXD library) sets the *exposure_time* field in a camera configuration structure

GetExposureTime (PXD library) reads the *exposure_time* field from a camera configuration structure

GetFramePeriod (PXD library) reads the *frame_period* field from a camera configuration structure

SetFramePeriod (PXD library) which sets the *frame_period* field in a camera configuration structure

GetStrobePeriod

```
short GetStrobePeriod (long hFG, short iWhich, short cP,
                      float *fSegment) ;
```

Return Value

Non-zero if successful; 0 on failure

Parameters

hFG

a handle to a frame grabber

iWhich

selects which of the two strobe generators, STROBE_0 or STROBE_1 will be programmed with the values contained in the *fSegment* array, the two strobe pulse generators are:

| Name | Connector |
|----------|--|
| STROBE_0 | as a TTL signal on pin 18 of the User I/O connector or as a RS422 signal on pins 9(+) and 59(-) of the camera connector |
| STROBE_1 | as a TTL signal on pin 19 of the User I/O connector Or as a RS422 signal on pins 8(+) and 58(-) of the camera connector |

cP

a count of the number of segments in the pulse sequence

fSegment

an array of floats containing *cP* elements, each of which will be loaded with the time in seconds of a segment of the pulse sequence

Description

GetStrobePeriod is used to read the pulse sequence from strobe pulse generator *iWhich*. Since a pulse sequence consists of up to eight segments (pulses and delays) you should in general allocate an eight-element array of floats for *fSegments*. Regardless of the size of *fSegments* **GetStrobePeriod** is guaranteed to never step past *cP* elements.

If the contents of *fSegment[0]* returned by **GetStrobePeriod** is greater than 0 then it represents the width, in seconds, of the first pulse; *fSegment[1]* holds the first delay and so forth.

If *fSegment[0]* contains zero then the pulse sequence begins with a delay whose value, in seconds, is contained in *fSegment[1]*; *fSegment[2]* then contains the first pulse width and so forth.

A pulse can be set to be either active-high or active low by setting of clearing the appropriate bits in the IO Register. A 1 written to the STROBE_0 bit of the IO Register will make a pulse active-low. Likewise a 0 will make a pulse active high. **WriteImmediateIO** is used to write the bits of the IO Register.

The strobe periods can be programmed with the **SetStrobePeriods** function. The strobe periods can also be programmed into a configuration file with the PXD configuration application.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

FireStrobes (PXD library) to initiate software triggered strobes

SetStrobePeriods (PXD library) to set the current pulse sequences

TriggerStrobes (PXD library) to set up hardware triggered strobes

WriteImmediateIO (PXD library) to set the polarity of the strobe outputs

GetSwitch

```
short GetSwitch(long fgh);
```

Return Value

-1

Description

This function is for compatibility only, and always returns -1.

GetTop

```
short GetTop (long hFG) ;
```

Return Value

the Y coordinate of the pixel at the top edge of the cropping region if successful;
0 if unsuccessful

Parameters

hFG
a handle to a frame grabber

Description

Often, only a portion of the image produced by the camera contains information of interest. Capturing only the area of interest, called the cropping region, instead of the entire image can reduce system memory and PCI bus bandwidth requirements. The pixel coordinates of the left edge and top along with the width and height define the cropping region. **GetTop** returns the Y coordinate of the pixels at the top edge of the currently selected cropping region.

This function executes immediately regardless of the state of the frame grabber queue.

GetTop will fail if the handle to the frame grabber, *hFG* is invalid.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetHeight (PXD library) returns the height of the cropping region

GetLeft (PXD library) returns the left edge of the cropping region

GetWidth (PXD library) returns the width of the cropping region

GetXResolution (PXD library) returns the width of the uncropped image

GetYResolution (PXD library) returns the height of the uncropped image

SetHeight (PXD library) sets the height of the cropping region

SetLeft (PXD library) set the left edge of the cropping region

SetTop (PXD library) sets the top edge of the cropping region

SetWidth (PXD library) sets the width of the cropping region

GetWidth

```
short GetWidth (long hFG) ;
```

Return Value

the width, in pixels, of the cropping region if successful; 0 if unsuccessful

Parameters

hFG
a handle to a frame grabber

Description

Often, only a portion of the image produced by the camera contains information of interest. Capturing only the area of interest, called the cropping region, instead of the entire image can reduce system memory and PCI bus bandwidth requirements. The pixel coordinates of the left edge and top along with the width and height define the cropping region. **GetWidth** returns the width of the currently selected cropping region.

This function executes immediately regardless of the state of the frame grabber queue.

GetWidth will fail if the handle to the frame grabber, *hFG* is invalid.

Example: GetWidth

```
void CropToCenterQuarter(PXD * pPxd, long fgh)
{
    short sWidth;
    short sHeight;

    sWidth = pPxd->GetWidth(fgh);
    sHeight = pPxd->GetHeight(fgh);

    pPxd->SetLeft(fgh, sWidth/4);
    pPxd->SetTop(fgh, sHeight/4);
    pPxd->SetWidth(fgh, sWidth/2);
    pPxd->SetHeight(fgh, sHeight/2);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetHeight (PXD library) returns the height of the cropping region

GetLeft (PXD library) returns the left edge of the cropping region

GetTop (PXD library) returns the top edge of the cropping region

GetXResolution (PXD library) returns the width of the uncropped image

GetYResolution (PXD library) returns the height of the uncropped image

SetHeight (PXD library) sets the height of the cropping region

SetLeft (PXD library) set the left edge of the cropping region

SetTop (PXD library) sets the top edge of the cropping region

SetWidth (PXD library) sets the width of the cropping region

GetXResolution

```
short GetXResolution (long hFG) ;
```

Return Value

the number of pixels per row that will be delivered from the camera to the frame grabber if successful; 0 if unsuccessful

Parameters

hFG
a handle to a frame grabber

Description

GetXResolution will report the actual number of pixels delivered from the camera to the frame grabber. If a cropping region has been selected, with **SetWidth**, **SetHeight**, **SetLeft** or **SetTop**, this will not reflect the number of pixels captured into an image buffer. **GetWidth** can be used to determine the width of the captured image, in pixels.

Since **GetXResolution** simply reports the value of the image width stored in the camera configuration file, this function executes immediately regardless of the state of the frame grabber queue.

GetXResolution will fail if the handle to the frame grabber, *hFG* is invalid.

Required Compile-time Headers

`pxd.h` for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

`ilib_32.lib` for Windows NT, 98 and 95
`pxd_fw.lib` for DOS4GW

Required Run-time Libraries

`pxd_32.dll` for Windows NT, 98 and 95

Also See

GetHeight (PXD library) returns the height of the cropping region

GetLeft (PXD library) returns the left edge of the cropping image

GetTop (PXD library) returns the top edge of the cropping region

GetWidth (PXD library) returns the width of the cropping region

GetYResolution (PXD library) returns the height of the uncropped image

SetHeight (PXD library) sets the height of the cropping region

SetLeft (PXD library) set the left edge of the cropping region

SetTop (PXD library) sets the top edge of the cropping region

SetWidth (PXD library) sets the width of the cropping region

GetYResolution

```
short GetYResolution (long hFG) ;
```

Return Value

the number of lines per frame that will be delivered from the camera to the frame grabber if successful; 0 if unsuccessful

Parameters

hFG

a handle to a frame grabber

Description

GetYResolution will report the actual number of lines per frame delivered from the camera to the frame grabber. If a cropping region has been selected, with **SetWidth**, **SetHeight**, **SetLeft** or **SetTop**, this will not reflect the number of lines captured into an image buffer. **GetHeight** can be used to determine the width of the captured image.

Since **GetYResolution** simply reports the value of the image height stored in the camera configuration file, this function executes immediately regardless of the state of the frame grabber queue.

GetYResolution will fail if the handle to the frame grabber, *hFG* is invalid.

Required Compile-time Headers

`pxd.h` for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

`ilib_32.lib` for Windows NT, 98 and 95

`pxd_fw.lib` for DOS4GW

Required Run-time Libraries

`pxd_32.dll` for Windows NT, 98 and 95

Also See

GetHeight (PXD library) returns the height of the cropping region

GetLeft (PXD library) returns the left edge of the cropping image

GetTop (PXD library) returns the top edge of the cropping region

GetWidth (PXD library) returns the width of the cropping region

GetXResolution (PXD library) returns the width of the uncropped image

SetHeight (PXD library) sets the height of the cropping region

SetLeft (PXD library) set the left edge of the cropping region

SetTop (PXD library) sets the top edge of the cropping region

SetWidth (PXD library) sets the width of the cropping region

Grab

```
long Grab(long fgh, FRAME *frh, short flags);
```

Return Value

A non-zero value if successful; 0 on failure. If the operation is QUEUED, the return value is a handle to a queued operation.

Parameters

fgh

handle to a frame grabber

frh

Handle to a frame buffer, created by AllocateAddress (**not WinNT**), AllocateBuffer, AllocateBufferList, AllocateFlatFrame (**not WinNT**), AllocateMemoryFrame, AliasFrame, FrameFromPointer, or a buffer constructed by the application.

flags

Options for how to perform the grab.

The flags field can be:

0 to wait for queued grabs to complete, then capture an image, and then return.

IMMEDIATE to capture an image, and then return, unless grabs are queued. If any grabs are queued, the function will return immediately, with an error.

QUEUED to return immediately, and complete the grab in the background

Description

Captures a video image and writes it to frame buffer *frh*. Grab() fails if the image size is larger in either the horizontal or vertical dimension than the destination frame. The parameter *flags* is a set of flag bits that can specify modes of operation for this function.

The two possible flags are QUEUED, which causes the Grab function to return immediately while the actual image capture occurs in the background, and IMMEDIATE which causes Grab to return an error if there are any uncompleted queued operations. If *flags* is 0, then the Grab function waits for all currently queued operations to complete, then captures an image, then returns to the application program.

The Grab function can act on lists of FRAMEs created by AllocateBufferList. The SINGLE_FLD, FIELD_0 and FIELD_1 flags from the PXC200 are not supported. Interlaced

video are supported through config file options, but switching between interlace and single field grabs cannot be done on a Grab by Grab basis.

The PXD library provides a special feature when grabbing from a line-scan camera. If the frame has a height larger than 1, and the source of the image is a linescan camera, i.e. has a height of 1, then the grab will not complete until the tall frame is full. Lines of image will be received from the camera, with each line being placed into successive rows of the frame.

A similar behavior exists in GrabContinuous. If the source of the image is a linescan camera, and the destination frame has a height greater than 1, then successive lines will be placed into successive rows of the frame, and will restart at the top of the frame after the bottom is reached.

If a buffer list is provided, the grab will not be completed until enough frames have been captured to fill the buffer list.

Grabs are always synchronized to Vsync. The number of frames that can be captured per second is directly affected as to whether the grabs are queued or not. If the grabs are queued, a new grab can begin as soon as the old grab is completed. As such, no frames are skipped. However, if the grabs are not queued, the previous grab may have completed at the end of a Vsync, but the application will probably not be able to request a grab before the next field starts. The next grab cannot occur until the next Vsync occurs. This lost time depends upon whether the camera is providing full frames, or a frame made of two fields, as in interlaced. If the camera is providing full frames, and entire frame is lost. Otherwise, a half frame, or a field, is lost. Thus, if the grabs are queued, up to full frame rate can be grabbed. Otherwise, grabs cannot occur at more than half of the camera's frame rate, if full frames are provided, or at two-thirds of the frame rate, if frames are provided as half-fields.

To take advantage of queued grabs, the application should double-buffer the grabs. One buffer is used to grab into, while the previously grabbed image can be examined in the other buffer. Once the new grab is completed, and the processing of the old image is completed, the role of the buffers can be reversed. That is, the first buffer becomes the examine buffer, and the second buffer is queued for a grab.

If the PCI bus is overloaded, it's possible for captured data to be corrupt. Although the Grab function can't determine when data is being corrupted, `CheckError()` will return the value `ERR_CORRUPT`.

The most common reasons the function can fail are:

- The frame grabber handle or the frame buffer handle is invalid.

- The image specified by `SetWidth()` or `SetHeight()` (or the default image size) is too large in width or height for the frame buffer.

If the function executes successfully, but doesn't produce the image you expect, the most common reasons are:

- If the captured image is all black or all blue, be sure to check that your video source is attached to the frame grabber and that the iris on the video camera is open.
- If you're using a system with an Intel Pentium Pro processor, you might not be able to read valid data from a frame buffer in system memory immediately after grabbing the image. This is due to the processor caching the data, rather than writing the data immediately to memory. Try inserting a delay in your program before reading the data.
- If you get only a few lines of valid video at the top of an image you've grabbed to a frame buffer in system memory, the PCI bus is being overloaded or errors are occurring on the bus. Most Intel 486-based systems don't have a PCI bus that is fast enough for PXC200 frame grabbers. Run the VGACOPY program to check for errors on the PCI bus.
- The frame grabber can't produce the image specified by **`SetHeight()`**, **`SetWidth()`**, **`SetXResolution()`**, and **`SetYResolution()`**.

The **`Grab()`** function will automatically terminate any **`GrabContinuous()`** operation.

Example: Grab

```
// this will include some code snippets describing how to perform a queued
// grab

// open the PXD library - demonstrated with the Windows command
// for DOS, replace the command with PXD_OpenLibrary
imagination_OpenLibrary("pxd32.dll", &pxd, sizeof(PXD));

// initialize the frame grabber. We will assume that it is grabbing
// at 640 by 480 by 8 bpp
fgh = pxd.AllocateFG(-1);

// create two buffers into which we can grab
frh1 = AllocateBuffer(640, 480, PBITS_Y8);
frh2 = AllocateBuffer(640, 480, PBITS_Y8);

// queue the first two grabs. Do this to keep the queue busy. If we do not
// queue these first two grabs, the queue will not be able to stay ahead
// of us, and we will not be able to grab at maximum frame rate.
// be sure to track the queue handles
```

```

qhl = pxd.Grab(fgh, frh1, QUEUED);
qh2 = pxc.Grab(fgh, frh2, QUEUED);
// record that the next grab that we expect to complete is the first grab,
// that is the operation identified by qh1
NextGrab = 1;

/*
   do some more stuff, like setting up windows, etc.
*/

// this piece goes into the body of the program, where you can poll for
// completion of a grab.
if (NextGrab == 1){
    // check if qh1 is completed
    if (pxd.IsFinished(fgh, qh1){
        // notice that we leave qh2, and the second frame buffer alone
        // the grab is complete. Thus, we can process the grabbed Image
        Process(frh1);    // user's process function
        // we are now done with the frame buffer, so we can queue it
        // to grab another Image
        qhl = pxd.Grab(fgh, frh1, QUEUED);
        // we will let this grab complete, awaiting the completion of
        // qh2 for the next image
        NextGrab = 2;
    }
}
}else{
    // check if qh2 is completed
    if (pxd.IsFinished(fgh, qh2){
        // notice that we leave qh1, and the first frame buffer alone
        // the grab is complete. Thus, we can process the grabbed Image
        Process(frh2);    // user's process function
        // we are now done with the frame buffer, so we can queue it
        // to grab another Image
        qh2 = pxd.Grab(fgh, frh2, QUEUED);
        // we will let this grab complete, awaiting the completion of
        // qh1 for the next image
        NextGrab = 1;
    }
}
}

```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

CheckError (PXD library) to identify the cause of an error

GrabContinuous (PXD library) to grab continuously into a buffer

IsFinished (PXD library) to check if a queued operation is completed

WaitFinished (PXD library) to wait until a queued operation is completed

GrabContinuous

```
long GrabContinuous(long fgh, FRAME *frh, short state, short flags);
```

Return Value

Non-zero if successful; 0 on failure

Parameters

fgh

handle to a frame grabber

frh

Handle to a frame buffer, created by `AllocateAddress (not WinNT)`, `AllocateBuffer`, `AllocateBufferList`, `AllocateFlatFrame (not WinNT)`, `AllocateMemoryFrame`, `AliasFrame`, `FrameFromPointer`, or a buffer constructed by the application.

state

if `-1`, turn on continuous grabbing, if `0` turn off continuous grabbing

flags

Options for how to perform the grab.

The flags field can be:

`0` to wait for queued grabs to complete, then capture the first image, and then return.

`IMMEDIATE` to capture an image, and then return, unless grabs are queued. If any grabs are queued, the function will return immediately, with an error.

`QUEUED` to return immediately, and perform the grabs in the background

Description

Turns continuous acquire mode on (if *state* = -1) or off (if *state* = 0) for a given frame grabber. In continuous acquire mode, the buffer *frh* is continuously updated with new video data. **GrabContinuous()** fails if the frame is not of the correct type to hold the data.

Continuous acquire mode can be useful for software that is watching a small number of pixels in every image, or for sending video data directly to another PCI device, but also requires fast access to RAM.

Since the buffer is updated continuously, it is possible to detect a shearing effect when the camera is capturing motion. This is because the moved image is overwriting the old image. To avoid motion shear within a buffer, use the `Grab()` function with the `QUEUED` flag.

The PXD library provides a special feature when grabbing from a line-scan camera. If the frame has a height larger than 1, and the source of the image is a linescan camera, i.e. has a height of 1, then the grabs will advance to fill up the frame. Lines of image will be received from the camera, with each line being placed into successive rows of the frame. After the bottom row is filled, the grabs will continue into the top row of the frame.

A similar behavior exists in `Grab`. If the source of the image is a linescan camera, and the destination frame has a height greater than 1, then a grab will not be completed until the entire buffer is filled.

If a buffer list is provided, each successive grabbed image will be placed into a successive frame in the list. After the last frame in the list is filled, the grabs will continue starting at the top of the list.

The `Grab()` function and any operations that change the type of data produced by the frame grabber or the resolution or size of the video image automatically turn off continuous acquire mode.

The parameter *flags* can specify additional modes of operation for this function. Note that a queued `GrabContinuous` operation is considered finished as soon as continuous acquire mode starts or stops, not when the first image has been fully captured.

If *flags* is 0, the default modes will be used.

If the PCI bus is overloaded, it's possible for captured data to be corrupt. Although the `Grab` function can't determine when data is being corrupted, `CheckError()` will return the value `ERR_CORRUPT`.

The most common reasons the function can fail are:

- The frame grabber handle or the frame buffer handle is invalid.
- The image specified by `SetWidth()` or `SetHeight()` (or the default image size) is too large in width or height for the frame buffer.

If the function executes successfully, but doesn't produce the image you expect, the most common reasons are:

- If the captured image is all black or all blue, make sure your video source is attached to the frame grabber and that the iris on the video camera is open.

- If you're using a system with an Intel Pentium Pro processor, you might not be able to read valid data from a frame buffer in system memory immediately after grabbing the image. This is due to the processor caching the data, rather than writing the data immediately to memory. Try inserting a delay in your program before reading the data.
- If you get only a few lines of valid video at the top of an image you've grabbed to a frame buffer in system memory, the PCI bus is being overloaded or errors are occurring on the bus. Most Intel 486-based systems don't have a PCI bus that is fast enough for PXD1000 frame grabbers. Run the VGACOPY program to check for errors on the PCI bus.
- The frame grabber can't produce the image specified by **SetHeight()**, **SetWidth()**, **SetXResolution()**, and **SetYResolution()**.

Example: GrabContinuous

```
long StartGrabbing(PXD * pPxd, long fgh, FRAME * pfrh)
{
    pPxd->GrabContinuous(fgh, pfrh, -1, 0);
}

long StopGrabbing(PXD * pPxd, long fgh, FRAME * pfrh)
{
    pPxd->GrabContinuous(fgh, pfrh, 0, 0);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

CheckError (PXD library) to check the cause of an error condition

Grab (PXD library) to grab a single frame

GrabTriggered

```
long GrabTriggered(long fgh,FRAME *frh,short deltime,short flags);
```

Return Value

A non-zero value if successful; 0 on failure. If the operation is QUEUED, the return value is a handle to a queued operation.

Parameters

fgh

handle to a frame grabber

frh

Handle to a frame buffer, created by AllocateAddress (**not WinNT**), AllocateBuffer, AllocateBufferList, AllocateFlatFrame (**not WinNT**), AllocateMemoryFrame, AliasFrame, FrameFromPointer, or a buffer constructed by the application.

deltime

Vertical syncs (or horizontal syncs if a line-scan camera) to wait after the trigger occurs before capturing the image.

flags

Options for how to perform the grab.

The flags field can be:

0 to wait for queued grabs to complete, then capture an image, and then return.

IMMEDIATE to capture an image, and then return, unless grabs are queued. If any grabs are queued, the function will return immediately, with an error.

QUEUED to return immediately, and complete the grab in the background

Description

After the frame grabber receives a trigger signal, it waits for *deltime* vertical sync times (or horizontal sync times for a linescan camera) and then captures a video image to the specified buffer. The *deltime* parameter can be used to delay the capture to allow time for a camera to reset, for example. This function can be used to synchronize frame grabbing to an external device.

Notice that if a trigger never comes this function will never complete. This means it is often advisable to use the QUEUED flag and the TimedWaitFinished function to provide a timeout capability.

Example: GrabTriggered

```
short GrabWhenTriggered(PXD * pPxd, long fgh, FRAME * pfrh)
{
    long qh;
    // grab the next sync pulse after the trigger. Save the queue handle
    // so that we can check when the grab is complete. We will wait for
    // up to ½ second
    qh = pPxd->GrabTriggered(fgh, pfrh, 0, QUEUED);

    return pPxd->TimedWaitFinished(fgh, qh, 0.5);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

Grab (PXD library) to grab a single frame

GrabContinuous (PXD library) to grab continuously into a frame

SetTriggerSource (PXD library) to select the input line which will serve as the trigger

TimedWaitFinished (PXD library) to prevent a triggered grab from stalling forever

IsFinished

```
short IsFinished(long fgh, long qh);
```

Return Value

- > 0 if the operation is not in the queue
- 0 if the specified operation is in the queue and has not been completed
- 1 if the specified frame grabber is invalid

Parameters

- fgh*
handle to a frame grabber
- qh*
handle to a queued operation

Description

Can be used to check whether a queued operation has finished by passing the *handle* returned by the function that queued the operation. It can also check whether **all** operations queued for a particular frame grabber are finished by using *handle* = 0.

This function and WaitFinished are the functions provided to determine whether a queued operation has completed. They are called to allow the application to poll when a grab, for instance, has completed. This function differs from WaitFinished, in that it returns immediately. This way, the application can perform additional processing while the queued operation is still being processed in the background.

Example

See the example under **Grab()**

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

Grab (PXD library) to grab a frame

KillQueue (PXD library) to terminate any queued operations

WaitFinished (PXD library) to wait until a queued operation completes

KillQueue

```
void KillQueue(long fgh);
```

Return Value

none

Parameters

fgh
handle to a frame grabber

Description

Aborts any operations in progress for the specified frame grabber. Any operations in the queue when this function is called will be removed, although the operations might already have executed. For instance, if a Grab() command was in the queue, some or all of the video data might have been written into the frame by the time the queue is killed.

It is a good programming practice to kill the queue before freeing any frames, if you have any queued grabs. This will guarantee that the grab will not inadvertently grab into unused memory.

This function takes several milliseconds to execute. It is intended primarily for recovering from error conditions, and to terminate any grabs before reconfiguring the frame grabber.

Example: KillQueue

```
void ShutDownFrameGrabber(PXD * pPxd, long fgh, FRAME * pfrh)
{
    // kill the queue, so that the frames are no longer in use
    pPxd->KillQueue(fgh);

    // free the frame handle, since we are no longer grabbing into it
    pPxd->FreeFrame(pfrh);

    // we can free the frame grabber
    pPxd->FreeFG(fgh);

    // we can free the library - using Windows syntax.
    // for DOS, use PXD_CloseLibrary(pPxd)
    imagenation_CloseLibrary(pPxd);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

Reset

LoadConfig

```
CAMERA_TYPE *LoadConfig (char *szFileName) ;
```

Return Value

A pointer to a camera configuration data structure on success; NULL on failure

Parameters

szFileName

a pointer to a zero terminated string containing the name of the camera configuration file

Description

LoadConfig creates a camera configuration data structure, reads the contents of the camera configuration file *szFileName* into it and returns a pointer to the structure.

LoadConfig is the first step in preparing a PXD frame grabber to operate with a specific digital camera. It should be followed by a call to **SetCameraConfig** which uses the camera configuration data structure loaded in to memory by **LoadConfig** to program the PXD for the specified camera.

If *szFileName* is **NULL** the default camera configuration is loaded – **DEFAULT.CAM**.

LoadConfig will fail if *szFileName* is not the name of a valid camera configuration file.

Example: LoadConfig

```
/* LoadConfig Example - loads camera configuration structure into memory */
CAMERA_TYPE *configInMem          /* create a pointer to a camera
                                configuration data structure */

char configFile [] = "Kodak14.CAM" ; /* a camera configuration file */

configInMem = LoadConfig (configFile) ; /* load structure from file */

/* use the camera configuration here */

FreeConfig (configInMem) ;          /* done, release the pointer */
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SaveConfig (PXD library) saves a camera configuration structure to a file

FreeConfig (PXD library) releases a memory resident camera config data structure created with LoadConfig

SetCameraConfig (PXD library) load a PXD frame grabber with a camera configuration

LockFrame

```
short LockFrame(long fgh, FRAME *frh);
```

Return Value

A non-zero value upon success, 0 if failure.

Parameters

fgh

handle to a frame grabber

frh

Handle to a frame buffer, created by `AllocateAddress` (**not WinNT**), `AllocateBuffer`, `AllocateBufferList`, `AllocateFlatFrame` (**not WinNT**), `AllocateMemoryFrame`, `AliasFrame`, `FrameFromPointer`, or a buffer constructed by the application.

Description

A locked frame is one which is ready for DMA, and will not cause a long pause for page swapping or DMA list building when it is grabbed to. Calling `LockFrame` forces all the memory swapping overhead and such to happen immediately, instead of happening in response to a grab. This may improve the realtime response of some systems, but may also cause problems by preventing the OS from efficiently using virtual memory. Use with caution.

`LockFrame` calls can be nested. Be sure you make exactly one `UnlockFrame` call for each `LockFrame` call. Freeing a locked frame can cause problems. Calling the functions `SetCameraConfig`, `SetLeft`, `SetTop`, `SetWidth`, or `SetHeight` will cause all locked frames to be unlocked and relocked.

`LockFrame` waits for the frame grabber's queue to become empty before executing.

Required Compile-time Headers

`pxd.h` for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

`ilib_32.lib` for Windows NT, 98 and 95

`pxd_fw.lib` for DOS4GW

Required Run-time Libraries

`pxd_32.dll` for Windows NT, 98 and 95

Also See

UnlockFrame (PXD library) to unlock the frame

OpenLibrary

```
short imagenation_OpenLibrary (  
  pSTR dllName,  
  interface* pLibInterfacae,  
  short sizeof(interface) ;
```

Return Value

the number of frame grabbers available; 0 on failure

Parameters

dllName

the name of the library, the libraries are:

| | |
|--------------|--|
| pxd_32.dll | the pxd frame grabber library providing functions to manage the frame grabber and capture images |
| frame_32.dll | the frame library providing functions to allocate image memory and save images |

pLibInterface

this is the name you will use to access the selected library, it is a pointer to a library interface structure which is itself an array of pointers to the individual functions in that library. There are two library interface structure types corresponding to the two Imagenation libraries:

| | |
|----------|---------------------------------|
| PXD | an interface to a pxd library |
| FRAMELIB | an interface to a frame library |

sizeof(interface)

the size of the interface structure for the library selected

Description

Before calling any of the library functions you must explicitly initialize each library by calling the appropriate **OpenLibrary** function. To call **OpenLibrary** you must first define a pointer to a library interface structure for the library you wish to open, either PXD or FRAMELIB.

OpenLibrary then:

- scans the system for PXD frame grabbers and returns the number found
- loads the selected library if one or more PXD frame grabbers are found and if it is not already loaded
- initializes the library by filling in the interface structure with pointers to the library functions
- connects the interface name you have specified (pLibInterface) to the loaded library.

You can then call any function in that library as a member of the interface structure.

OpenLibrary will fail if no frame grabbers are detected. It may also fail under extremely low memory conditions.

Following your last call to a library, and before your program terminates, you must call the appropriate **CloseLibrary** function.

Example: Open Library

```

PXD    Pxd ;           /* a pointer to a pxd library interface structure
                        this is the name you will use to access the
                        pxd_32.dll library functions */
FRAMELIB  Framelib ; /* a pointer to a frame library interface structure
                        this is the name you will use to access the
                        frame_32.dll library functions */
long    hFG ;         /* a handle to a frame grabber */
FRAME*  pFrame ;     /* a pointer to a image buffer in memory */

/* for DOS use:
   PXD_OpenLibrary ("pxd_32.dll", &Pxd, sizeof (PXD)) ;
   PXD_OpenLibrary ("frame_32.dll", &FrameLib, sizeof (FRAMELIB)) ;
*/
/* get access to the libraries, in Windows*/
imagenation_OpenLibrary ("pxd_32.dll", &Pxd, sizeof (PXD)) ;
imagenation_OpenLibrary ("frame_32.dll", &FrameLib, sizeof (FRAMELIB)) ;

/* allocate an image buffer */


---


pFrame = pxd.AllocateBuffer (IMAGEWIDTH, IMAGEHEIGHT, PBITS Y16)) ;

/* get access to a frame grabber */
hFG = pxd.AllocateFG (-1) ;

/* capture an image and move it to the image buffer */

```

Chapter 6

```
pxd.Grab (hFG, pFrame, 0) ;

/* do your image processing here */

/* release the image buffer */
FrameLib.freeFrame (pFrame) ;

/* release the frame grabber */
pxd.FreeFG (hFG) ;

/* close the library */
/* for DOS use
   PXD_CloseLibrary(&pxd);
*/
/* for windows: */
imagenation_CloseLibrary(&pxd);
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

CloseLibrary (PXD library) to close the library when done with the grabber

QualifyGrabs

```
short QualifyGrabs(long fgh,short delay,short count);
```

Return Value

A non-zero value if success, or 0 if failure.

Parameters

fgh

handle to the frame grabber

delay

number of frames after the trigger to delay. If a linescan camera, this is the number of lines to delay.

count

the number of frames to grab

Description

Arranges things so that grabs will only occur for **count** frames (or lines for linescan cameras) starting **delay** frames (or lines) after each trigger. The images which are qualified by the trigger can be captured using normal **Grab** or **GrabContinuous** commands. This function provides higher resolution capture control than **GrabTriggered**.

Calling **QualifyGrab** with a count of 0 returns the frame grabber to normal grabbing.

If **QualifyGrabs** is active, **GrabTriggered** will always fail without doing anything.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

Grab (PXD library) to grab a single frame

GrabContinuous (PXD library) to grab continuously into a frame

GrabTriggered (PXD library) to grab after a trigger occurs

ReadConfigData

```
short ReadConfigData(long fgh, short start, short len, void
*buf) ;
```

Return Value

A 0 is returned on failure, or nonzero is returned on success.

Parameters

fgh

a handle to a frame grabber

start

The starting entry to read

len

The number of entries to read

buf

an array of bytes to receive the configuration data. It must be able to receive *len* number of bytes.

Description

ReadConfigData returns customer-specific data stored in an onboard nonvolatile memory. The PXD1000 has 128 bytes of storage available. The size of the requested data is in bytes.

Example: ReadConfigData

```
unsigned long IncrementNonVolatileCounter(PXD * pPxd, long fgh)
{
    // for this example, we will use 4 bytes starting at entry 16 to record
    // a non-volatile counter. These values are arbitrary.
    // We will also return the new count
    unsigned long        ulCounter;

    pPxd->ReadConfigData(fgh, 16, 4, &ulCounter);
    ulCounter += 1;

    pPxc->WriteConfigData(fgh, 16, 4, &ulCounter);
    return ulCounter;
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

ReadProtection (PXD library) to read the protection of the pxd

ReadRevision (PXD library) to read the revision of the pxd

ReadSerial (PXD library) to read the serial number of the pxd

WriteConfigData (PXD library) to write the configuration data

ReadIO

```
long ReadIO(long fgh);
```

Return Value

The state of the I/O lines. An I/O line will be 0 if the line is low, and 1 if the line is high.

Parameters

fgh
handle to a frame grabber

Description

Returns a set of bit flags indicating the state of the I/O lines. Bits that have no associated I/O line return zero.

Example: ReadIO

```
void ShowIO(PXD * pPxd, long fgh)
{
    short sCounter;
    long IOFlags;
    unsigned short sMask;
    char szBuffer[256];

    // read the IO bit flags
    IOFlags = pPxd->ReadIO(fgh);
    sMask = 1<<15; // set the mask to bit 15
    for (sCounter = 15; sCounter > 0; sCounter--)
    {
        sprintf(strlen(szBuffer), "Bit %2d: %s\n",
            (IOFlags & sMask ? "HI" : "LO"));
        sMask >>= 1; // mask a lower bit
    }
    MessageBox(NULL, szBuffer, "IO Lines", MB_OK);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetIOType
WriteImmediateIO

I/O Line Types

| Name | Value | Description |
|-----------|-------|------------------------------------|
| IO_INPUT | 4 | signal can be read but not written |
| IO_OUTPUT | 5 | signal can be written and read |

The I/O lines on the PXD1000 are numbered as follows:

| Name | I/O Number | Types |
|-----------|------------|-----------|
| Trigger | 0 | IO_INPUT |
| WEN | 1 | IO_INPUT |
| FIELD | 2 | IO_INPUT |
| LDV | 3 | IO_INPUT |
| FDV | 4 | IO_INPUT |
| HDrive | 5 | IO_INPUT |
| VDrive | 6 | IO_INPUT |
| TTL In 0 | 7 | IO_INPUT |
| TTL In 1 | 8 | IO_INPUT |
| Strobe 0 | 9 | IO_OUTPUT |
| Strobe 1 | 10 | IO_OUTPUT |
| Control 0 | 11 | IO_OUTPUT |
| Control 1 | 12 | IO_OUTPUT |
| Control 2 | 13 | IO_OUTPUT |
| TTL Out 0 | 14 | IO_OUTPUT |
| TTL Out 1 | 15 | IO_OUTPUT |

ReadProtection

```
short ReadProtection(long fgh);
```

Return Value

The protection key if successful; 0 on failure

Parameters

fgh
handle to a frame grabber

Description

Returns the hardware protection key of the frame grabber. The returned value will be zero unless the frame grabber has been programmed with a key to match your custom software. A return of (-1) indicates a hardware compatibility problem.

Example: ReadProtection

```
void IdentifyGrabber(PXD * pPxd, long fgh)
{
    char szBuffer[256];

    // note: the following command builds a string by using the
    // C syntax of "aaa"bbb" gets treated by the compiler as "aaabbb"
    sprintf(szBuffer, "Protection key : %d\n"
        "Serial Number : %d\n"
        "Hardware Revision: %08X\n"
        "Software Revision: %08X\n"
        "PXD Model : %d\n",
        pPxd->ReadProtection(fgh),
        pPxd->ReadSerial(fgh),
        pPxd->ReadRevision(fgh),
        pPxd->ReadRevision(0))
        pPxd->GetModelNumber(fgh);
    MessageBox(NULL, szBuffer, "Grabber Identity", MB_OK);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetModelNumber

ReadConfigData

ReadRevision

ReadSerial

ReadRevision

```
long ReadRevision(long fgh);
```

Return Value

The revision number if successful; 0 on failure

Parameters

fgh
handle to a frame grabber

Description

Returns the hardware/firmware revision number of the frame grabber. If *fgh* = 0, ReadRevision() returns the revision number of the software library.

A revision number for the frame grabber has the form:

```
0xBBbbFFff
```

where BB and bb are major and minor board revisions. FF and ff are major and minor FPGA code revisions. Note that there is an interface change from the PXC, which returns a short.

You can also get the revision numbers with the ViewPXD program: Start ViewPXD. From the View menu, select Grabber Configuration to display the revisions.

Example: ReadRevision

```
void GetRevision(PXD * pPxd, long fgh, long * plHWRev, long * plSWRev)
{
    *plHWRev = pPxd->ReadRevision(fgh);
    *plSWRev = pPxd->ReadRevision(0);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

ReadConfigData (PXD library) to read the configuration data

ReadRevision (PXD library) to read the revision of the pxd

ReadSerial (PXD library) to read the serial number of the pxd

ReadSerial

```
long ReadSerial(long fgh);
```

Return Value

The serial number of the board if successful; 0 on failure

Parameters

fgh
handle to a frame grabber

Description

Returns the serial number of the frame grabber. The value returned will be zero unless the frame grabber has been programmed with a serial number. A return of (-1) indicates a hardware compatibility problem.

Example

See example in **ReadProtection**.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

ReadConfigData (PXD library) to read the configuration data of the pxd

ReadRevision (PXD library) to read the revision of the pxd

ReadSerial (PXD library) to read the serial number of the pxd

Reset

```
void Reset(long fgh);
```

Return Value

none

Parameters

fgh
handle to a frame grabber

Description

Returns the frame grabber to a default state, and aborts any queued operations and any digital I/O operations. This function takes several milliseconds to execute.

In order for Reset to work, the file **pxd1000.pxd** must be in the library's search path. Reset will also use the file **default.cam** to initialize the camera setup if it can be found.

The library's search path is: The Windows/system32 directory (Windows NT only), the Windows/system directory (Windows only) the Windows directory (Windows only), then any directories listed in the IMAGENATION environment variable, then the directories in the PATH variable, then the current directory.

Example: Reset

```
Void ShutDown(PXD * pPxd, long fgh)
{
    // before we shut down, let's kill the queue, and reset the camera.
    // By killing the queue, we will be sure that the grabber is no longer
    // getting data from the camera.
    // By resetting the grabber, we will be sure that it is in a clean
    // state the next time we use it - just in case we would have left it
    // in a messed up state, such as with an odd LUT.
    // But, the ConfigData is not reset
    pPxd->KillQueue(fgh);
    pPxd->Reset(fgh);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

KillQueue (PXD library) to terminate any queued operations

SaveConfig

```
short SaveConfig (CAMERA_TYPE *configInMem,  
                 char *szFileName  
                 short overwrite) ;
```

Return Value

Non-zero on success; zero on failure

Parameters

configInMem

a pointer to a camera configuration structure in memory

szFileName

a pointer to a zero terminated string that will be used as the name for the camera configuration file

overwrite

- if *overwrite* is non-zero and *szFileName* already exists then the contents will be overwritten with the new configuration data;
- if *overwrite* is 0 and *szFileName* exists then the contents will not be overwritten and **SaveConfig** will return an error (a 0 return value);
- if *szFileName* does not exist then *overwrite* has no effect on the operation of **SaveConfig**

Description

SaveConfig writes a camera configuration data structure to a file pointed to by *szFileName*. When *overwrite* != 0 an existing file will be overwritten with new configuration data.

SaveConfig can fail if:

- a file of the same name as *szFileName* already exists and *overwrite* permission has not been granted (*overwrite* = 0),
- *configInMem* does not point to a valid camera configuration structure or
- a disk error occurs

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

LoadConfig (PXD library) loads a camera configuration structure from a file

FreeConfig (PXD library) releases a memory resident camera config data structure created with LoadConfig

SetCameraConfig (PXD library) which programs a PXD frame grabber with a camera configuration

SetBrightness

```
long SetBrightness(long fgh, float offset, short flags);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

SetCamera

```
short SetCamera(long fgh, short n, short flags);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

SetCameraConfig

```
short SetCameraConfig (long hFG, CAMERA_TYPE *cam) ;
```

Return Value

Non-zero on success; 0 on failure

Parameters

hFG

a handle to a PXD frame grabber

cam

a pointer to a camera configuration data structure

Description

SetCameraConfig configures the frame grabber to work with the camera described in the *cam* structure. This structure can be read from disk using **LoadConfig**, included in a program's static data, or even created on the fly by a sufficiently clever program.

SetCameraConfig makes its own internal copies of any data it needs to preserve from *cam*, so the structure can be changed or freed after the **SetCameraConfig** call without modifying the state of the frame grabber.

When a configuration file is loaded, **SetCameraConfig** will test the correctness of the configuration. Some fields are based upon a computation of other fields, others must have values that are from a list of possibilities, and others may or may not be required. If the variety of tests applied to the camera configuration fail, then the **SetCameraConfig** function will return a 0 as the return value. To get specific information about why the camera configuration failed, you can load the configuration file with the **ConfigPXD** application. This will thoroughly analyze the configuration, and provide feedback about what changes are required.

Example: SetCameraConfig

```

void SetConfigTest(PXD * pPxd, long fgh)
{
    CAMERA_TYPE *configInMem          /* create a pointer to a camera
                                     configuration data structure */
    char configFile [] = "Kodak14.CAM" ; /* a camera configuration file */
    configInMem = LoadConfig (configFile) ; /* load structure from file */
    /* use the camera configuration here */
    pPxd->SetCameraConfig(fgh, configInMem);

    FreeConfig (configInMem) ;          /* done, release the pointer */
}

```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

LoadConfig (PXD library) loads a camera configuration structure from a file

SaveConfig (PXD library) saves a camera configuration structure to a file

FreeConfig (PXD library) releases a memory resident camera configuration data structure created with **LoadConfig**

SetContrast

```
long SetContrast(long fgh, float gain, short flags);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

SetExposureTime

```
short SetExposureTime(CAMERA_TYPE *cam, float t);
```

Return Value

Nonzero if successful; 0 on failure

Parameters

cam

a camera configuration structure

t

time to set the exposure in seconds

Description

The CAMERA_TYPE structure must have a variable exposure time, or **SetExposureTime** will fail. That is, the EXP_flags field must contain the flag VARIABLE_EXPOSURE, not FIXED_EXPOSURE. The exposure time needs to be at least the value in the exposure_min field and not greater than the value in the exposure_max field.

Example: SetExposureTime

```
short ChangeExposureTime(PXD * pPxd, long fgh, float t, char *
szCamFileName, short sSave)
{
    // change the exposure time using the selected camera configuration file.
    // if sSave != 0, we will save the modified cam file
    // if we have success, we will return non-zero, otherwise we will return
    // 0
    CAMERA_TYPE * Cam;
    short sRetval;

    Cam = pPxd->LoadConfig(szCamFileName);
    if (!Cam){
        return 0;
    } else{
        pPxd->SetExposureTime(Cam, t);
        sRetval = pPxd->SetCameraConfig(fgh, Cam);
        if (sSave){
            pPxd->SaveCameraConfig(Cam, szFileName, 1/*overwrite*/);
        }
        pPxd->FreeConfig(Cam);
    }
}
```

```
    return sRetVal;  
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

LoadConfig (PXD library) to load the configuration file

SaveConfig (PXD library) to save the configuration file

SetCameraConfig (PXD library) to set the configuration to the pxd

SetFieldCount

```
short SetFieldCount(long fgh, long c);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

SetFramePeriod

```
short SetFramePeriod (CAMERA_TYPE *pCamera, float fTime)
;
```

Return Value

non-zero on success; 0 on failure

Parameters

pCamera

a pointer to a camera configuration structure

fTime

the desired frame time in seconds

Description

Some digital cameras can be set to begin to expose a new frame when they receive a trigger signal from an external source. If the trigger occur less frequently than the time necessary to send a complete frame to the frame grabber, the camera will wait for the next trigger before sending another frame. If the camera is resettable (can be interrupted in the middle of transmitting a frame) then faster triggers can produce higher frame rates as only a portion of the lines in each frame are sent to the frame grabber before the next trigger occurs.

SetFramePeriod is designed to work with those cameras that respond to an external trigger signal. It will program the strobe generators.

In other cameras the frame period is the time from the start of one frame to the start of the next. **SetFramePeriod** writes the value *fTime* into `frame_period` field of the camera configuration structure pointed to by *pCamera*. The new value will only take effect after the next call to **SetCameraConfig**. **SetFramePeriod** will be successful only if the `FIXED_PERIOD` bit-flag of the `EXP_flags` field in the camera configuration structure pointed to by *pCamera* is set to 0, indicating that the frame period is variable.

SetFramePeriod is one of a small set of functions that provide access to specific fields in a camera configuration structure.

The default frame period is programmed into the default configuration file (DEFAULT.CAM) with the PXD Configuration application.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SetExposureTime (pxd library) sets the exposure_time field in a camera configuration structure

GetExposureTime (pxd library) reads the exposure_time field from a camera configuration structure

GetPixelType (pxd library) reads the pix_type field from a camera configuration structure

GetFramePeriod (pxd library) which reads the frame_period field in a camera configuration structure

SetHeight

```
short SetHeight (long hFG, short height) ;
```

Return Value

the actual number of lines set for the cropped image if successful; 0 if unsuccessful

Parameters

hFG

a handle to a frame grabber

height

the requested number of lines in the cropped image

Description

Often, only a portion of the image produced by the camera contains information of interest. Capturing only the area of interest, called the cropping region, instead of the entire image can reduce system memory and PCI bus bandwidth requirements. The pixel coordinates of the left edge and top along with the width and height define the cropping region. **SetHeight** is used to set the height of the cropping region.

To avoid confusion about the size of the captured image, this function waits until the queue is empty before executing.

SetHeight will fail if the handle to the frame grabber, *hFG* is invalid.

If the height to be set is larger than the maximum allowed by the configuration file, the height will be set to the maximum. The height must also be a multiple of 4.

Example: SetHeight

```
FRAME * GetBuffer(PXD * pPxd, long fgh, FRAME * pfrh, short sWidth, short
sHeight, short sType)
{
    // set the grabber to the desired width and height, and then create
    // a buffer to match
    sWidth = pPxd->SetWidth(fgh, sWidth);
    sHeight = pPxd->SetHeight(fgh, sHeight);
    return pPxd->AllocateBuffer(fgh, sWidth, sHeight, sType);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetHeight (PXD library) returns the height of the cropping region

GetLeft (PXD library) returns the left edge of the cropping region

GetTop (PXD library) returns the top edge of the cropping region

GetWidth (PXD library) returns the width of the cropping region

GetXResolution (PXD library) returns the width of the uncropped image

GetYResolution (PXD library) returns the height of the uncropped image

SetLeft (PXD library) set the left edge of the cropping region

SetTop (PXD library) sets the top edge of the cropping region

SetWidth (PXD library) sets the width of the cropping region

SetInputLUT

```
short SetInputLUT(long fgh,short bits,short LUT,long
start,long len,void *data);
```

Return Value

A non-zero value on success, or 0 if failure.

Parameters

fgh

handle to a frame grabber

bits

the number of bits in the LUT entry. This is either 8 or 16.

LUT

the LUT from which to read

start

the starting entry to read

len

the number of entries to read from the LUT

data

address of the buffer to receive the LUT entries. It must be large enough to receive *len* entries with *bits* per entry.

Description

Changes values in the frame grabber's input lookup tables (LUTs). The frame grabber can act as if it has four tables, each 8 bits wide, or two tables, each 16 bits wide. The table size is specified by setting *bits* to either 8 or 16. Which table to access is selected by setting *LUT* (0..3 for 8 bit tables, 0 or 1 for 16 bit tables). All tables can be written with the same data by setting *LUT* to -1. Any subrange of the table can be changed by specifying the *start* and *length* of the range to be altered. The data to be put in the table is specified in the array *data*, which must have at least *length* entries. These entries should be unsigned char if *bits* is 8, or unsigned short if *bits* is 16. If the frame grabber is processing queued operations, SetInputLUT() waits for the operations to finish before executing.

The LUTs which are applicable, depending upon whether the camera has a single channel, two channels, or 4 channels:

| Camera has a single channel | LUT 0 | LUT 1 | LUT 2 | LUT 3 |
|------------------------------------|--------------|--------------|--------------|--------------|
| 8 bpp | No | Yes | No | Yes |
| 16 bpp | Yes | No | No | No |
| Camera has two channels | LUT 0 | LUT 1 | LUT 2 | LUT 3 |
| 8 bpp | No | Yes | No | Yes |
| 16 bpp | Yes | Yes | No | No |
| Camera has four channels | LUT 0 | LUT 1 | LUT 2 | LUT 3 |
| 8 bpp | Yes | Yes | Yes | Yes |
| 16 bpp | No | No | No | No |

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetInputLUT (PXD library) retrieve the LUT from the grabber

LoadConfig (PXD library) reads the configuration file, to identify the number of channels

SetIOType

```
short SetIOType(long fgh, short n, short type);
```

Return Value

0 (because there are no I/O lines with settable type)

Description

This function is for compatibility only, and always returns 0.

SetLeft

```
short SetLeft (long hFG, short left) ;
```

Return Value

the actual X coordinate of the left edge of the cropped image (this might be different than the value requested, see explanation below) if successful; 0 if unsuccessful

Parameters

hFG

a handle to a frame grabber

left

the X coordinate of the requested left edge of the cropped image

Description

Often, only a portion of the image produced by the camera contains information of interest. Capturing only the area of interest, called the cropping region, instead of the entire image can reduce system memory and PCI bus bandwidth requirements. The pixel coordinates of the left edge and top along with the width and height define the cropping region. **SetLeft** is used to set the X coordinate of the left edge of the cropping region. All data transfers from the frame grabber are aligned on DWORD boundaries. This means that the value of *left* will be rounded down so that the start of each line of the cropped image is DWORD aligned. **SetLeft** returns the value of the actual left edge of the cropping region set.

To avoid confusion about the size of the captured image, this function waits until the queue is empty before executing.

SetLeft will fail if the handle to the frame grabber, *hFG* is invalid.

Example: SetLeft

```
Void CropToCenterQuarter(PXD * pPxd, long fgh)
{
    short sWidth;
    short sHeight;

    sWidth = pPxd->GetWidth(fgh);
    sHeight = pPxd->GetHeight(fgh);
    pPxd->SetLeft(fgh, sWidth/4);
    pPxd->SetTop(fgh, sHeight/4);
    pPxd->SetWidth(fgh, sWidth/2);
    pPxd->SetHeight(fgh, sHeight/2);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

- GetHeight** (PXD library) returns the height of the cropping region
- GetLeft** (PXD library) returns the left edge of the cropping region
- GetTop** (PXD library) returns the top edge of the cropping region
- GetWidth** (PXD library) returns the width of the cropping region
- GetXResolution** (PXD library) returns the width of the uncropped image
- GetYResolution** (PXD library) returns the height of the uncropped image
- SetHeight** (PXD library) sets the height of the cropping region
- SetTop** (PXD library) set the top edge of the cropping region
- SetWidth** (PXD library) sets the width of the cropping region

SetStrobePeriods

```
short SetStrobePeriods (long hFG, short iWhich, short cP,
                        float *fSegment) ;
```

Return Value

Non-zero if successful; 0 on failure

Parameters

hFG

a handle to a frame grabber

iWhich

selects which of the two strobe generators, STROBE0 or STROBE1 will be programmed with the values contained in the *fSegment* array

cP

a count of the number of segments in the pulse sequence

fSegment

an array of floats containing *cP* elements, each of which is the time in seconds of a segment of the pulse sequence

Description

Each of the two strobe generators, Strobe0 and Strobe1, can be programmed to output a pulse sequence consisting of up to four pulses separated by gaps. Pulse and gap lengths can be programmed from about 1 microsecond to about three seconds, accurate to 0.5 microseconds or about 0.1%. The array *fSegment* holds *cP* time values. *fSegment[0]* is the length of the first pulse in seconds, *fSegment[1]* if the length of the first gap, and so forth. If a delayed pulse is desired, *fSegment[0]* can be set to 0, which causes the first segment to be a gap.

SetStrobePeriod executes concurrently with any queued functions. The new strobe sequence takes effect immediately.

If the strobes are firing continuously (see **ContinuousStrobes**), it is necessary that they be disabled first. This is done by calling **ContinuousStrobes** with an *iMode* of 0.

The default strobe period is programmed into the default configuration file (DEFAULT.CAM) with the PXD Configuration application.

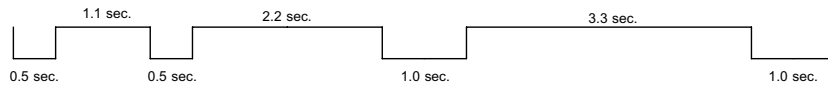
Example: SetStrobePeriods

```
/* SetStrobePeriod Example
   hFG is a valid handle to a PXD1000 frame grabber */

#define NUMSEGMENTS      8

float fSegment[NUMSEGMENTS] = { 0, 0.5, 1.1, 0.5, 2.2, 1.0, 3.3, 1.0 } ;
SetStrobePeriods (hFG, STROBE_0, NUMSEGMENTS, fSegment) ;
```

Once triggered, this code will produce the following pulse sequence from Strobe_0:



Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

- TriggerStrobes** (PXD library) to set up hardware triggered strobes
- FireStrobes** (PXD library) to initiate software triggered strobes
- WriteImmediateIO** (PXD library) to set the polarity of the strobe outputs
- ContinuousStrobes** (PXD library) to fire strobes continuously

SetTop

```
short SetTop (long hFG, short top) ;
```

Return Value

the actual Y coordinate of the top edge of the cropped image if successful; 0 if unsuccessful

Parameters

hFG

a handle to a frame grabber

top

the Y coordinate of the requested top edge of the cropped image

Description

Often, only a portion of the image produced by the camera contains information of interest. Capturing only the area of interest, called the cropping region, instead of the entire image can reduce system memory and PCI bus bandwidth requirements. The pixel coordinates of the left edge and top along with the width and height define the cropping region. **SetTop** is used to set the Y coordinate of the top edge of the cropping region.

To avoid confusion about the size of the captured image, this function waits until the queue is empty before executing.

SetTop will fail if the handle to the frame grabber, *hFG* is invalid.

Example: SetTop

```
Void CropToCenterQuarter(PXD * pPxd, long fgh)
{
    short sWidth;
    short sHeight;

    sWidth = pPxd->GetWidth(fgh);
    sHeight = pPxd->GetHeight(fgh);

    pPxd->SetLeft(fgh, sWidth/4);
    pPxd->SetTop(fgh, sHeight/4);
    pPxd->SetWidth(fgh, sWidth/2);
    pPxd->SetHeight(fgh, sHeight/2);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetHeight (PXD library) returns the height of the cropping region

GetLeft (PXD library) returns the left edge of the cropping region

GetTop (PXD library) returns the top edge of the cropping region

GetWidth (PXD library) returns the width of the cropping region

GetXResolution (PXD library) returns the width of the uncropped image

GetYResolution (PXD library) returns the height of the uncropped image

SetHeight (PXD library) sets the height of the cropping region

SetLeft (PXD library) set the left edge of the cropping region

SetWidth (PXD library) sets the width of the cropping region

SetTriggerSource

```
short SetTriggerSource(long fgh,short trig,short type);
```

Return Value

Non-zero if successful; 0 on failure

Parameters

fgh

handle to a frame grabber

trig

the input line which will serve to trigger the grab. The constants are:

| | |
|------------|---|
| TRIGGER | the trigger line, I/O line number 0 |
| WEN | I/O line number 1 |
| GPINO_MASK | the TTL Input 0 line, I/O line number 7 |

type

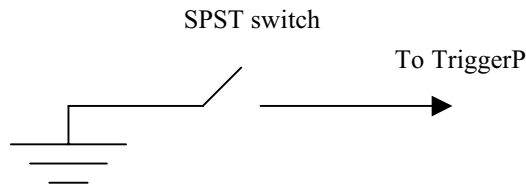
the type of trigger event that will cause the trigger. The choices are:

| | |
|---------------|---|
| LATCH_RISING | The trigger will occur when the line goes from 0 to 1. If the line is already high, it must go low first, then high again to cause a trigger. |
| LATCH_FALLING | The trigger will occur when the line goes from 1 to 0. If the line is already low, it must go high first, then low again to cause a trigger. |
| IO_INPUT_HIGH | The trigger will occur as soon as it is detected to be high. If the line is already high, the trigger will occur immediately. |
| IO_INPUT_LOW | The trigger will occur as soon as it is detected to be low. IF the line is already low, the trigger will occur immediately. |
| DEBOUNCE | This allows the 2-trigger approach to condition the trigger. |

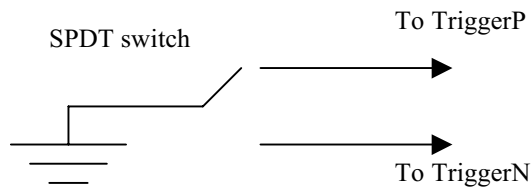
Description

Selects which input line will trigger GrabTriggered or QualifyGrabs. The possible input lines are the Trigger, the WEN pulse, or GPIInput 0. The trigger event can also be the special value STROBES_DONE, which causes a trigger whenever a strobe pulse sequence finishes. The type can be LATCH_RISING, LATCH_FALLING, IO_INPUT_HIGH, or IO_INPUT_LOW. If the trigger is STROBES_DONE, the type is ignored.

If type includes the mask DEBOUNCE, and the trig is TRIGGER, two trigger signals are required. One signal is sent to TRIGGERP, line 14 of the I/O connector. The other is sent to TRIGGERN, line 15 of the I/O connector. By sending the trigger to TRIGGERP the internal trigger state will be set to one state. Sending the trigger signal to TRIGGERN, the internal state will be reversed. This can be easily accomplished by replacing an SPST trigger switch with an SPDT. Thus, instead of just breaking the trigger line, the trigger signal will be sent to one of two outputs. These two outputs can be sent to TRIGGERN or TRIGGERP.



Example of a simple trigger using an SPST switch. The DEBOUNCE feature of the PXD1000 is not available with this configuration.



Example of a trigger using an SPDT switch. This setup can take advantage of the DEBOUNCE feature of the PXD1000.

Example: SetTriggerSource

```

short GrabWhenTriggered(PXD * pPxd, long fgh, FRAME * pfrh)
{
    long qh;
    // grab the next sync pulse after the trigger. Save the queue handle
    // so that we can check when the grab is complete. We will wait for
    // up to ½ second
    pPxd->SetTriggerSource(fgh, TRIGGER, LATCH RISING);
    qh = pPxd->GrabTriggered(fgh, pfrh, 0, QUEUED);

    return pPxd->TimedWaitFinished(fgh, qh, 0.5);
}

```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GrabTriggered (PXD library) grab a frame when the trigger occurs

TimedWaitFinished (PXD library) prevent the triggered grab from stalling forever

SetWidth

```
short SetWidth (long hFG, short width) ;
```

Return Value

the actual width of the cropped image if successful; 0 if unsuccessful

Parameters

hFG

a handle to a frame grabber

height

the requested width of the cropped image

Description

Often, only a portion of the image produced by the camera contains information of interest. Capturing only the area of interest, called the cropping region, instead of the entire image can reduce system memory and PCI bus bandwidth requirements. The pixel coordinates of the left edge and top along with the width and height define the cropping region. **SetWidth** is used to set the width cropping region.

To avoid confusion about the size of the captured image, this function waits until the queue is empty before executing.

SetWidth will fail if the handle to the frame grabber, *hFG* is invalid.

Example: SetWidth

```
Void CropToCenterQuarter(PXD * pPxd, long fgh)
{
    short sWidth;
    short sHeight;

    sWidth = pPxd->GetWidth(fgh);
    sHeight = pPxd->GetHeight(fgh);

    pPxd->SetLeft(fgh, sWidth/4);
    pPxd->SetTop(fgh, sHeight/4);
    pPxd->SetWidth(fgh, sWidth/2);
    pPxd->SetHeight(fgh, sHeight/2);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GetHeight (PXD library) returns the height of the cropping region

GetLeft (PXD library) returns the left edge of the cropping region

GetTop (PXD library) returns the top edge of the cropping region

GetWidth (PXD library) returns the width of the cropping region

GetXResolution (PXD library) returns the width of the uncropped image

GetYResolution (PXD library) returns the height of the uncropped image

SetHeight (PXD library) sets the height of the cropping region

SetLeft (PXD library) set the left edge of the cropping region

SetTop (PXD library) sets the top edge of the cropping region

SetXResolution

```
short SetXResolution(long fgh, short rez);
```

Return Value

Returns the resolution specified by the current camera config

Description

This function is for compatibility only.

SetYResolution

```
short SetYResolution(long fgh, short rez);
```

Return Value

Returns the resolution specified by the current camera config

Description

This function is for compatibility only.

SwitchCamera

```
long SwitchCamera(long fgh, short flags);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

SwitchGrab

```
long SwitchGrab(long fgh, FRAME__PX_FAR *f0,  
FRAME__PX_FAR *f1, FRAME__PX_FAR *f2, FRAME__PX_FAR *f3,  
short flags);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

TimedWaitFinished

```
short TimedWaitFinished(long fgh, long qh, float timeout);
```

Return Value

Non-zero if the operation finished; 0 if time expired

Parameters

fgh

handle to a frame grabber

qh

handle to a queued operation

timeout

the number of seconds until time expires

Description

Waits until the specified operation completes, or timeout seconds have passed, whichever comes first. If *qh* is 0, it waits for all queued operations on the frame grabber to complete.

This function is useful to prevent the GrabTriggered event from stalling forever, should a trigger never occur.

Example: TimedWaitFinished

```
short GrabWhenTriggered(PXD * pPxd, long fgh, FRAME * pfrh)
{
    long qh;
    // grab the next sync pulse after the trigger. Save the queue handle
    // so that we can check when the grab is complete. We will wait for
    // up to ½ second
    qh = pPxd->GrabTriggered(fgh, pfrh, 0, QUEUED);

    return pPxd->TimedWaitFinished(fgh, qh, 0.5);
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

GrabTriggered (PXD library) to grab a frame after the trigger occurs

TriggerStrobes

```
short TriggerStrobes (long hFG, short strobeTrig,  
                     short iType, short iField) ;
```

Return Value

Non-zero if successful; 0 on failure

Parameters

hFG

a handle to a frame grabber

strobeTrig

selects the hardware event (trigger) that will fire the strobes:

| | |
|-------|--|
| WEN | a strobe generated by the camera |
| GPINO | general purpose input 0 |
| LDV | line data valid from the camera |
| FDV | frame data valid from the camera |
| 0 | used to disable hardware-initiated strobes |

iType

selects the type of change in the selected hardware trigger will fire the strobes:

| | |
|---------------|---|
| LATCH_RISING | a low-to-high transition initiates the strobes |
| LATCH_FALLING | a high-to-low transition initiates the strobes |
| IO_INPUT_HIGH | a high level initiates the strobes, if the input selected by <i>strobeTrig</i> input remains high this mode will generate continuous repeating strobe sequences |
| IO_INPUT_LOW | a low level initiates the strobes, if the input selected by <i>strobeTrig</i> input remains low, this mode will generate continuous repeating strobe sequences |

iField

typically an input from an interlaced camera, this is a qualifier on the strobe trigger that must be satisfied before the strobes will be fired. Values for *iField* are:

| | |
|--------|---|
| FIELD0 | the strobes will fire when the FIELD input from the camera is at logical 0 and an iType strobe trigger event occurs on input strobeTrig |
| FIELD1 | the strobes will fire when the FIELD input from the camera is at logical 1 and an iType strobe trigger event occurs on input strobeTrig |
| EITHER | the strobes will fire when an iType strobe trigger event occurs on input strobeTrig regardless of the state of FIELD |

If you are using a non-interlaced camera (most digital cameras are non-interlaced) then this field must be set to EITHER.

Description

The PXD1000 contains two strobe generators, Strobe_0 and Strobe_1 each capable of stepping through a unique pulse-sequence. While the strobes can be programmed individually, they are both always fired by the same event. The effects of a call to **TriggerStrobes** continue until changed. This means that *iType* hardware events on input *strobeTrig* will cause both strobe generators to repeatedly fire.

TriggerStrobes has no effect if the frame grabber is in continuous strobe mode. This can be set with the **ContinuousStrobes** function.

You can also use the **FireStrobes** function to initiate software-triggered strobes.

Use **SetStrobePeriod** to define the pulse sequence for each strobe generator. The output polarity of each strobe is programmed with **WriteImmediateIO**.

Example

See the **TrigStro** sample in the **Samples** folder for a detailed example of triggered strobes

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

SetStrobePeriods (PXD library) to program strobe pulse-sequences

FireStrobes (PXD library) to initiate software triggered strobes

WriteImmediateIO (PXD library) to set the polarity of the strobe outputs

UnlockFrame

```
void UnlockFrame(long fgh, FRAME *frh);
```

Return Value

Non-zero if successful, or 0 if failure.

Parameters

fgh

handle to a frame grabber

frh

Handle to a frame buffer, created by `AllocateAddress` (**not WinNT**), `AllocateBuffer`, `AllocateBufferList`, `AllocateFlatFrame` (**not WinNT**), `AllocateMemoryFrame`, `AliasFrame`, `FrameFromPointer`, or a buffer constructed by the application.

Description

Releases a lock on a frame made by `LockFrame`. This gives control of the frame's buffer memory back to the operating system, possibly allowing the buffer to be swapped out or moved in physical RAM.

`UnlockFrame` waits for the frame grabber's queue to empty before executing.

Required Compile-time Headers

`pxd.h` for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

`ilib_32.lib` for Windows NT, 98 and 95

`pxd_fw.lib` for DOS4GW

Required Run-time Libraries

`pxd_32.dll` for Windows NT, 98 and 95

Also See

`LockFrame` (PXD library) to lock the frame

VideoType

```
short VideoType(long fgh);
```

Return Value

- 0 no video
- 1 NTSC video (invalid for the PxD library)
- 2 PAL/SECAM video (invalid for the PxD library)
- 3 digital
- 1 invalid *fgh*

Parameters

fgh
handle to a frame grabber

Description

Returns the type of video signal connected to the frame grabber. Returns 3 if the frame grabber detects an active pixel clock, 0 if no clock is detected, or -1 if *fgh* is not a valid frame grabber. This function is useful to identify whether a camera is connected to the frame grabber.

Example: VideoType

```
// code snippet demonstrating VideoType  
  
if (Pxd.VideoType(hFG) <= 0) {  
    MessageBox(NULL, "No Camera Attached", "Error", MB_ICONSTOP);  
}
```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Wait

```
long Wait(long fgh,short flags);
```

Return Value

Non-zero if succesful, or 0 if failure.

Parameters

fgh

handle to a frame grabber

flags

options about how to wait

Possible flags

| | |
|-----------|--|
| 0 | to wait for queued operations to finish, ignore a frame, and return. |
| IMMEDIATE | to ignore a frame, and then return, unless grabs are queued. If any grabs are queued, the function will return immediately, with an error. |
| QUEUED | to return immediately, and complete and discard the grab in the background |

Description

This function reads an image out of the frame grabber and discards it. The primary purpose of this function is to provide a delay that is equal in duration to the time to complete a grab. If the Wait() function is QUEUED, it does not pause program execution, but any QUEUED functions that are called immediately afterwards will not execute until the Wait() is finished. A useful rule for understanding the Wait() function is that it always has the same timing as a Grab() function called with the same flags; that is, a Wait() takes the same time to execute as the equivalent Grab() function (except that a Wait cannot be slowed down by limitations in PCI transfer speed), but doesn't collect any image data during that time. The parameter *flags* is a set of flag bits that can specify modes of operation for this function.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95

pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

WaitMultiple (PXD library) to discard several frames

WaitAllEvents

```
long WaitAllEvents(long fgh, long ioh, unsigned long  
mask, unsigned long state, short flags);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

WaitAnyEvent

```
long WaitAnyEvent(long fgh, long ioh, unsigned long mask,  
unsigned long state, short flags);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

WaitFinished

```
void WaitFinished(long fgh, long qh);
```

Return Value

1 if successful; 0 on failure

Parameters

fgh

handle to a frame grabber

qh

handle to a queued operation. Can be 0 to wait for all queued operations to complete

Description

Releases the processor to execute other tasks until a specific operation in the queue has finished. You identify an operation in the queue by the *qh* returned by the function that queued the operation. For *qh* = 0, `WaitFinished()` waits until all operations in the queue have finished.

This function is very useful to manage queued grabs, when it is not necessary to perform background operations, instead of using `IsFinished()`.

Windows 95 and Windows NT are multithreaded, preemptive multitasking operating systems. In such systems, using empty loops to wait for events slows the system dramatically by wasting processing time that could be used by other threads. For example, an empty loop like this might be used in a Windows 3.1 program:

```
while (!pxd.IsFinished(fgh, qh))
;
```

In Windows 95 and Windows NT, such an empty loop is not very efficient, so an alternate function, **`WaitFinished()`**, is included in the library for such applications:

```
pxd.WaitFinished(fgh, qh);
```

The `WaitFinished()` function uses system synchronization objects to prevent the current thread from executing while the wait is in progress. Since all queued operations finish executing during vertical blank, polling only once per vertical blank is just as accurate as polling more often, but significantly improves system performance.

Example: WaitFinished

```
// this will include some code snippets describing how to perform a queued
```

Chapter 6

```
// grab

// open the PXD library - demonstrated with the Windows command
// for DOS, replace the command with PXD_OpenLibrary
imagination_OpenLibrary("pxd32.dll", &pxd, sizeof(PXD));

// initialize the frame grabber. We will assume that it is grabbing
// at 640 by 480 by 8 bpp
fgh = pxd.AllocateFG(-1);

// create two buffers into which we can grab
frh1 = AllocateBuffer(640, 480, PBITS_Y8);
frh2 = AllocateBuffer(640, 480, PBITS_Y8);

// queue the first two grabs. Do this to keep the queue busy. If we do not
// queue these first two grabs, the queue will not be able to stay ahead
// of us, and we will not be able to grab at maximum frame rate.
// be sure to track the queue handles
qh1 = pxd.Grab(fgh, frh1, QUEUED);
qh2 = pxc.Grab(fgh, frh2, QUEUED);
// record that the next grab that we expect to complete is the first grab,
// that is the operation identified by qh1
NextGrab = 1;

/*
do some more stuff, like setting up windows, etc.
*/

// this piece goes into the body of the program, where you can poll for
// completion of a grab.
if (NextGrab == 1){
    // wait until qh1 is completed
    pxd.WaitFinished(fgh, qh1);
    // notice that we leave qh2, and the second frame buffer alone
    // the grab is complete. Thus, we can process the grabbed Image
    Process(frh1);    // user's process function
    // we are now done with the frame buffer, so we can queue it
    // to grab another Image
    qh1 = pxd.Grab(fgh, frh1, QUEUED);

    // we will let this grab complete, awaiting the completion of
    // qh2 for the next image
    NextGrab = 2;
}else{
    // wait until qh2 is completed
```

```

pxd.WaitFinished(fgh, qh2);
// notice that we leave qh1, and the first frame buffer alone
// the grab is complete. Thus, we can process the grabbed Image
Process(frh2); // user's process function
// we are now done with the frame buffer, so we can queue it
// to grab another Image
qh2 = pxd.Grab(fgh, frh2, QUEUED);

// we will let this grab complete, awaiting the completion of
// qh1 for the next image
NextGrab = 1;
}

```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

IsFinished (PXD library) to test whether a queued operation is completed, instead of waiting for it to complete

WaitMultiple

```
long WaitMultiple(long fgh,short count,short flags);
```

Return Value

Non-zero if successful, or 0 if failed.

Parameters

fgh

handle to a frame grabber

count

the number of frames to discard

flags

options about how to wait. Possible flags:

- | | |
|-----------|---|
| 0 | to wait for queued operations to finish, ignore a frame, and return. |
| IMMEDIATE | to ignore a frame, then return, unless grabs are queued. If any grabs are queued, the function will return immediately with an error. |
| QUEUED | to return immediately, and complete and discard the grab in the background |

Description

Discards *count* images before continuing, as if *count* waits had been called in a row. This is most useful for line scan cameras, where a Wait call will discard a single line of data.

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

Wait (PXD library) to discard a single frame

WaitVB

```
short WaitVB(long fgh);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

WriteConfigData

```
short WriteConfigData(long fgh,short start,short len,void  
*buf) ;
```

Return Value

A 0 is returned on failure, or nonzero is returned on success.

Parameters

fgh

a handle to a frame grabber

start

The starting entry to read

len

The number of entries to read

buf

an array of bytes to receive the configuration data. It must be able to receive *len* number of bytes.

Description

WriteConfigData writes customer-specific data to an onboard nonvolatile memory. The data will be copied from *buf*, which must hold the requested number of bytes. The PXD1000 has 128 bytes of storage available. The size of the requested data, and the start offset into the 128 byte memory are in bytes. Returns nonzero on success, or 0 on failure.

Example

See the example for **ReadConfigData**

Required Compile-time Headers

`pxd.h` for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

`ilib_32.lib` for Windows NT, 98 and 95

`pxd_fw.lib` for DOS4GW

Required Run-time Libraries

`pxd_32.dll` for Windows NT, 98 and 95

Also See

ReadConfigData (PXD library) to read the configuration data from the pxd

ReadProtection (PXD library) to read the protection key

ReadRevision (PXD library) to read the revision of the pxd

ReadSerial (PXD library) to read the serial number from the pxd

WriteImmediateIO

```
short WriteImmediateIO(long fgh, long mask, long state);
```

Return Value

Non-zero if successful; 0 on failure

Parameters

fgh

handle to a frame grabber

mask

mask of I/O lines to set

state

state to which the masked I/O lines should be set. The *mask* and *state* can be built from the combinations of the provided bitmasks:

TRIGGER_MASK

WEN_MASK

FIELD_MASK

LDV_MASK

FDV_MASK

HDRIVE_MASK

VDRIVE_MASK

GPINO_MASK

GPIN1_MASK

STROBE0_MASK

STROBE1_MASK

CTRL0_MASK

CTRL1_MASK

CTRL2_MASK

GPOUT0_MASK

GPOUT1_MASK

Description

Sets all I/O lines that have a 1 bit in the *mask* to the value in the associated bit of *state*. Lines with a zero bit in the mask are not affected. The function fails without doing anything if the mask has no 1 bits.

I/O Line Types

| Name | Value | Description |
|-----------|-------|------------------------------------|
| IO_INPUT | 4 | signal can be read but not written |
| IO_OUTPUT | 5 | signal can be written and read |

The I/O lines on the PXD1000 are numbered as follows:

| Name | I/O Number | Types |
|-----------|------------|-----------|
| Trigger | 0 | IO_INPUT |
| WEN | 1 | IO_INPUT |
| FIELD | 2 | IO_INPUT |
| LDV | 3 | IO_INPUT |
| FDV | 4 | IO_INPUT |
| HDrive | 5 | IO_INPUT |
| VDrive | 6 | IO_INPUT |
| TTL In 0 | 7 | IO_INPUT |
| TTL In 1 | 8 | IO_INPUT |
| Strobe 0 | 9 | IO_OUTPUT |
| Strobe 1 | 10 | IO_OUTPUT |
| Control 0 | 11 | IO_OUTPUT |
| Control 1 | 12 | IO_OUTPUT |
| Control 2 | 13 | IO_OUTPUT |
| TTL Out 0 | 14 | IO_OUTPUT |
| TTL Out 1 | 15 | IO_OUTPUT |

Example: WriteImmediateIO

```

void TestIO(PXD * pPxd, long fgh)
{
    // for this test, we will raise Strobe0, Control0, and TTLOut0.
    // then, we will drop Control 0.
    // Finally, we will drop Strobe0 and TTLOut0, but set Control0 high.

    // raise Strobe0, Control0, TTLOut0 by setting the bit in both the
    // mask and state
    pPxd->WriteImmediateIO(fgh,
        STROBE0_MASK | CTRL0_MASK | GPOUT0_MASK, // mask
        STROBE0_MASK | CTRL0_MASK | GPOUT0_MASK); // state

    // drop control 0, by including it in the mask, but leaving its bit
    // at 0 in the state
    pPxd->WriteImmediateIO(fgh,
        CTRL0_MASK, // mask
        0); // state

    // drop Strobe0 and TTLOut0, but set Control0 high, by
    // including all three in the mask, but only setting the bit
    // for Control0 in the state.
    pPxd->WriteImmediateIO(fgh,
        STROBE0_MASK | CTRL0_MASK | GPOUT0_MASK, // mask
        CTRL0_MASK); // state
}

```

Required Compile-time Headers

pxd.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

ilib_32.lib for Windows NT, 98 and 95
pxd_fw.lib for DOS4GW

Required Run-time Libraries

pxd_32.dll for Windows NT, 98 and 95

Also See

ContinuousStrobes

FireStrobes

GetIOType

GetStrobePeriods

ReadIO

SetStrobePeriod

SetIOType

TriggerStrobes

Chapter 7

Frame Library

This chapter is a complete, alphabetical function reference for the Frame libraries and DLLs. For additional information on using the functions, see Chapter 5, *Developing Applications with the PXD1000*. For reference information on the PXD Library, see Chapter 6, *PXD Library*.

This function reference is a general guide for using the functions with all operating systems and languages. The functions will work as written for C and Visual Basic with the header files provided.

If you need to construct your own header file, you will need to know the definitions of constants, the sizes of the parameters and the return values for the function calls. You can find the definitions of constants in the header files for C and Visual Basic. The following table gives the sizes of the various data types that are used by the PXD1000 library.

| Type | Size |
|--|-------------|
| Unsigned char | 8 bits |
| long, unsigned long | 32 bits |
| void *, unsigned char *, int *, char *, LPSTR | 32 bits |
| short | 16 bits |

FRAME and FRAMELIB are defined types; to see how they are defined, refer to the C language header file for the appropriate operating system. Void is a special type. When it is the type for a parameter, the function has no parameters; when it is the type for the return value, the function does not return a value.

The library and DLL interface is almost identical for all operating systems. Functions that do not apply to a particular operating system or language are noted with an icon:

AliasFrame

```
FRAME *AliasFrame(FRAME *frh, short x0,
short y0, short dx, short dy, unsigned short type);
```

Return Value

A pointer to the frame structure; NULL on failure.

Parameters

frh

handle to a frame buffer

x0

left edge of the frame *frh* that will be included in the aliased frame

y0

top edge of the frame *frh* that will be included in the aliased frame

dx

width of the region in frame *frh* that will be included in the aliased frame

dy

height of the region in the frame *frh* that will be included in the aliased frame

type

the pixel type of the aliased frame. This does not change the contents of the frame, so should typically be the same type as the frame *frh*.

| | |
|-------------|---|
| PBITS_Y8 | 8-bit gray scale stored in an <i>unsigned char</i> |
| PBITS_Y16 | 16-bit gray scale stored in an <i>unsigned short</i> |
| PBITS_RGB32 | 8/8/8 Red/Green/Blue color stored in the lower 24 bits of a <i>long</i> . |

Description

Creates a new frame structure that uses the memory from the original frame's image buffer, starting at the location of the pixel *x0,y0*. The frame *f* must not be a paged frame buffer and must not be a planar data type. The new frame treats the memory from the old frame as if it has the new data format *type*.

AliasFrame() fails if the memory required for the new frame does not fit completely inside the old frame. Freeing the old frame before freeing the alias frame can cause undefined behavior, since this frees the image buffer for the alias frame as well. Freeing the alias frame does not affect the original frame's buffer.

Note that new memory is not used for the alias frame buffer. That is, by changing the contents of the alias frame, the contents of the original frame *frh* are also changed. Likewise, if the contents of the original frame *frh* are changed within the alias region, bounded by *x0*, *y0*, *dx*, and *dy*, the contents of the alias frame will also be changed. New memory is used to track the structure of the alias frame. As such, the memory should be released with **FreeFrame()** when the alias frame is no longer needed.

This function is useful for effectively cropping an image to a smaller size, without the need to reconfigure the frame grabber, and without the need to copy the contents into another buffer.

Example

```
short SaveRegionAsPNG(FRAMELIB * pFlib, FRAME * pfrh, short x0, short y0,
short dx, short dy, char * szName)
{
    // for this example, we will save a portion of a frame in the PNG file
    // format. This will be done by aliasing a frame into an original frame,
    // and usign the WritePNG format to write the alias frame.
    FRAME * pAliasFrame;
    short sRetval = 0;

---


    pAliasFrame = pFlib->AliasFrame(pfrh, x0, y0, dx, dy);
    if (pAliasFrame != NULL){
        sRetval = pFlib->WritePNG(pAliasFrame, szFileName, TRUE /*overwrite*/);
        pFlib->FreeFrame(pAliasFrame);
    }
    return sRetval;
}
```

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

FreeFrame (frame library) to free the memory used by the alias frame

AllocateAddress

```
FRAME *AllocateAddress(unsigned long address, short dx,  
short dy, unsigned short type);  
short GetCamera(long fgh);
```

Return Value

NULL

Description

This function is for compatibility only, and always returns **NULL**.

AllocateFlatFrame

```
FRAME *AllocateFlatFrame(short dx, short dy, unsigned  
short type);
```

Return Value

NULL

Description

This function is for compatibility only, and always returns **NULL**.

AllocateMemoryFrame

```
FRAME *AllocateMemoryFrame(short dx, short dy, unsigned short type);
```

Return Value

A pointer to the frame structure; NULL on failure.

Parameters

dx

width of the region in frame *frh* that will be included in the aliased frame

dy

height of the region in the frame *frh* that will be included in the aliased frame

type

the pixel type of the aliased frame. This does not change the contents of the frame, so should typically be the same type as the frame *frh*.

| | |
|--------------|---|
| PBITS_Y8 | 8-bit gray scale stored in an unsigned char |
| PBITS_Y16 | 16-bit gray scale stored in an unsigned short |
| PBITS_Yf | the pixels are grayscale values, stored in the buffer as floats. This is a non-standard color format, so it is up to the application to interpret these values. |
| PBITS_RGB15 | 15-bit RGB, packed into 16 bits. Stored as upper bit is ignored, next 5 are Red, next 5 are Green, and lower 5 bits are Blue. |
| PBITS_RGB16 | 16-bit RGB. Stored as upper 5 bits are Red, middle 6 bits are Green, and lower 5 bits are Blue. |
| PBITS_RGB24 | 24-bit RGB. Stored as three bytes, arranged as Red/Green/Blue. |
| PBITS_RGB32 | 8/8/8 Red/Green/Blue color stored in the lower 24 bits of a long. |
| PBITS_RGBf | the pixels are color values, stored in the buffer as three floats. This is a non-standard color format, so it is up to the application to interpret these values. |
| PBITS_YUV422 | 8 bits for Y, and 8 bits for CrCb.. |

| | |
|---------------|-------------------------------|
| PBITS_YUV444 | 8 bits each for Y, Cr, and Ch |
| PBITS_YUV422P | YUV422 in planar format |
| PBITS_YUV444P | YUV444 in planar format |

Description

Creates a frame of size dx by dy , with the specified pixel *type*, from the program's memory heap. Both dx by dy must be greater than zero. The start of the image buffer will be aligned to at least a 32-bit boundary in the program's address space.

FreeFrame should be called when the frame is no longer needed.

This function differs from the **AllocateBuffer** in the pxd library, in that it does *not* guarantee that the frame will be useable by the frame grabber. **AllocateMemoryFrame** does not enforce additional alignment requirements of the pxd library, such as the width must be a multiple of 4. Also, it allows for greater flexibility in the types available. A buffer allocated with **AllocateMemoryFrame** is, however, useful for manipulating frames in memory.

Example

```
short ExampleLoadBmp(FRAMELIB * pFlib, FRAME * *ppfrh, char * szName)
{
    // This example will load a bitmap into memory, without the use of the
    // pxd library. This is useful, since the pxd does not easily support
    // RGB24, which is a common format for BMP's. Thus, we would not want
    // to use AllocateBuffer.

    // notice that the frame handle is a double pointer. That is, the calling
    // application will provide the address of a pointer to the frame buffer.

    // we are assuming that the bitmaps are 640*480*24bpp. If we did not
    // know the size of the bitmap, we would have to examine the bitmap
    // header. That is beyond the scope of this example.

    short sRetVal = 0;

---


    *ppfrh = pFlib->AllocateMemoryFrame(640, 480, PBITS_RGB24);
    

---


    if (*ppfrh){
        // we successfully allocated the frame, so we can read the bitmap
        pFlib->ReadBmp(*ppfrh, szName);
    }
    return sRetVal;
}
```

```

short TestLoadBmp()
{
    FRAMELIB Flib;
    FRAME * pfrh;

    // open the frame library, so that we can use the frame functions
    // for DOS, the syntax is:
    // FRAME_OpenLibrary(&Flib, sizeof(FRAMELIB));
    imagenation_OpenLibrary("frame_32.dll", &Flib, sizeof(FRAMELIB));

    // load the bitmap into a frame buffer. The frame buffer will
    // be created, so we can't pass a FRAME. Instead, we will pass a
    // pointer to a frame, and the function will change the value of the
    // pointer, so that we know where the frame is located in memory
    ExampleLoadBmp(&Flib, &pfrh, "TEST.BMP" /* replace with a real bmp*/);

    // do something with the bitmap here

    // free the memory used by the bitmap
    Flib.FreeBuffer(pfrh);

    // close the library
    // for DOS, the syntax is:
    // FRAME_CloseLibrary(&Flib);
    imagenation_CloseLibrary(&Flib);
}

```

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

FreeFrame (frame library) to free the memory used by the alias frame

AllocateBuffer (pxd library) to create a buffer that is guaranteed to work with the frame grabber

CloseLibrary

DOS Syntax **void FRAME_CloseLibrary(FRAMELIB *interface);**
Win C Syntax **void imagenation_CloseLibrary(FRAMELIB *inter-**
 face);
Win VB Syntax **CloseLibrary(0)**

Return Value

None.

Parameters

interface

pointer to the interface created with a call to OpenLibrary

Description

Returns to the system any resources that were allocated by OpenLibrary. CloseLibrary should be the last library function called by the program. A program that exits after calling OpenLibrary, but before calling CloseLibrary, will leave the computer in an unstable state and might crash the operating system.

Be sure to call CloseLibrary before exiting the program, but after any frame library functions need to be used.

Example

```
// very simple application to demonstrate using the frame library
//
// this would be compiled as a console app under windows
#include <stdio.h>
#include <windows.h>    // if for windows

#include "frame.h"
main()
{
    // declare a variable to hold the interface to the frame library
    FRAMELIB      FrameLib;
    // declare a variable to point to a frame
    FRAME      * pfrh;
    // declare a variable to store the width
    short      sWidth;

    // open the frame library, which gives us an interface to the
```

```

// frame library functions
// for DOS, the syntax is:
// FRAME_OpenLibrary(&FrameLib, sizeof(FRAMELIB);
imagination_OpenLibrary("frame_32.h", &FrameLib, sizeof(FRAMELIB);

// do something with the frame library, like create a frame
pfrh = FrameLib.AllocateMemoryFrame(640, 480, PBITS_Y16);

// get the width of the frame
sWidth = FrameLib.FrameWidth(pfrh);

// free the frame buffer, now that we are done. We must be sure
// not to use pfrh after this point, as the data it points to is
// no longer valid
FrameLib.FreeFrame(pfrh);

// close the library when we are done
// for DOS, the syntax is :
// FRAME_CloseLibrary(&FrameLib);
imagination_CloseLibrary(&FrameLib);
exit(0);
}

```

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

OpenLibrary (frame library) to free the memory used by the alias frame

CopyFrame

```
short CopyFrame(FRAME *source, short sourcex, short  
sourcey, FRAME *dest, short destx, short desty, short dx,  
short dy) ;
```

Return Value

Non-zero if successful; 0 on failure.

Parameters

source

the frame from which the buffer will be copied

sourcex

the column in the source buffer that will become the left edge of the copied region

sourcey

the row in the source buffer that will become the top edge of the copied region

dest

the frame into which the buffer will be copied

destx

the column in the dest buffer that will be left edge of the copied region

desty

the row in the dest buffer that will be left edge of the copied region

dx

the width of the region to be copied

dy

the height of the region to be copied

Description

Copies a rectangle of size *dx* by *dy* from the frame *source* to the frame *dest*. Copies data only between parts of rectangles that are within the boundaries of the frames.

CopyFrame fails if the specified region is entirely outside the boundaries of the frames, if the frames can't be read or written, if the frames are planar, or if the frames don't have the same pixel data type.

This method is very useful for moving data between frames. It is also useful for extracting portions of an image. When used in conjunction with FrameFromPointer(), it can be used to move data from a frame into a non-frame buffer, such as a DirectDraw surface (see the example).

Example

```

// this example shows a method of copying a frame into a DirectDraw surface.
// This is only possible under Windows.

#include <windows.h>
#include <ddraw.h>

#include "frame.h"
#include "pxd.h"

void FrameToSurface(FRAMELIB * pFrameLib,
                   FRAME * pfrh,
                   DIRECTDRAWSURFACE lpSurface)
{
    // for this example, we will assume that the frame and the surface are
    // the same bit depth. A surface created with no specified pixel
    // format will default to have the same pixel format as the display.
    // The frame grabber can be set to have the same pixel format as the
    // display by detecting the pixel format of the surface, and then
    // configuring the frame to match that pixel format.
    FRAME * pSurfaceFrame;
    short sWidth, sHeight, sType;

    // get the dimensions of the frame
    sWidth = pFrameLib->GetWidth(pfrh);
    sHeight = pFrameLib->GetHeight(pfrh);
    sType = pFrameLib->GetType(pfrh);

    // lock the surface, so that we can get a pointer to its buffer
    lpSurface->lpVtbl->Lock(lpSurface, NULL, &ddsd,
                          DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL);
    // create a frame from the pointer
    pSurfaceFrame = FrameFromPointer(ddsd.lpSurface, sWidth, sHeight, sType);

    // copy into the surface's frame
    pFrameLib->CopyFrame(pfrh, 0, 0, pSurfaceFrame, 0, 0, sWidth, sHeight);

    // release the surface frame
    pFrameLib->FreeBuffer(pSurfaceFrame);

    // unlock the surface
    lpSurface->lpVtbl->Unlock(lpSurface, ddsd.lpSurface);
}

```

```

short CreateGrabbableFrameMatchingSurface(
    PXD * pPxd,
    FRAMELIB * pFrameLib,
    DIRECTDRAWSURFACE lpSurface,
    FRAME * *ppfrh)
{
    DDSURFACEDESC ddsd;
    short sRetval = 1;
    short sType = 0;
    DWORD dwMask;

    // get the surface description, so that we can determine
    // the bit depth to which the frame should be set
    ddsd.dwSize = sizeof(DDSURFACEDESC);
    if (lpSurface->lpVtbl->GetSurfaceDesc(lpSurface, &ddsd ) != DD_OK) {
        sRetval = 0;
    } else {
        switch (ddsd.ddpfPixelFormat.dwRGBBitCount)
        {
            case 8:
                sType = PBITS_RGB8;
                break;
            case 16:
                // even though it says 16 bpp, it could really be 15 bpp.
                // if 15, we would create the surface to be with PBITS_RGB15, and
                // if 15, we would create the surface to be with PBITS_RGB16
                // check the green bit mask. This is the value that
                // determines the difference. RGB15=555, RGB16=565

                dwMask = ddsd.ddpfPixelFormat.dwGBitMask;
                while ( !(dwMask & 0x01) ){
                    dwMask >>= 1;
                }
                if(dwMask == 0x1F) {
                    sType = PBITS_RGB15;
                } else if(dwMask == 0x3F) {
                    sPixType = PBITS_RGB16;
                }
                break;
            case 24:
                sType = PBITS_RGB24;
                break;
            case 32:
                sType = PBITS_RGB32;
                break;
        }
    }
}

```

```

    }

    // create the frame at the requested frame depth. We will assume
    // that the surface is in system memory, so that the pitch in bytes
    // matches the number of bytes in the width. If no, change
    // the buffer's width to match the number of bytes in the pitch,
    // divided by the bytes per pixel - that is, treat the pitch
    // as the entire width
    *ppfrh = pPxd->AllocateBuffer(ddsd.dwWidth, ddsd.dwHeight, sType);
    if (*ppfrh == NULL){
        sRetval = 0;
    }
}
return sRetval;
}

void ExampleQuickGrabIntoSurface(LPDIRECTDRAWSURFACE lpSurface)
{
    // demonstrate the creation of a frame that matches the bit depth of a
    // DirectDraw surface, and how to copy the contents of a frame
    // into the surface

    // we will create a pointer to a frame, so that we can be told
    // where the frame was created
    FRAME * pfrh;
    FRAMELIB FrameLib;
    PXD pxd;
    long fgh;

    // create access to the frame grabber
    imagenation_OpenLibrary("pxd_32.dll", &pxd, sizeof(PXD));
    imagenation_OpenLibrary("frame_32.dll", &FrameLib, sizeof(FRAMELIB));
    CreateGrabbableFrameMatchingSurface(&pxd, &FrameLib, lpSurface, &pfrh);
    fgh = pxd.AllocateFG(-1);

    // grab an image, so that we can copy it
    Pxd.Grab(fgh, 0);

    // copy the image into a surface, using the example function
    FrameToSurface(&FrameLib, pfrh, lpSurface);
}

```

```
// shut the pxd back down
pxd.FreeFG(fgh);
pxd.FreeFrame(pfrh);
imagination_CloseLibrary(&FrameLib);
imagination_CloseLibrary(&pxd);
}
```

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

AllocateFrame (pxd library) to create a buffer which can be grabbed into
FreeFrame (frame library) to free the memory used by the alias frame

ExtractPlane

```
FRAME *ExtractPlane(FRAME *frh, short plane);
```

Return Value

Returns a frame that contains a single plane of the planar frame *frh*.

Parameters

frh

handle to a frame

plane

identifier to the frame to extract

0 – the Y component, same height as the source frame

1 – the Cr component, same height as the source frame, except for YUV422P,
it is half the height of the source frame, rounded up

2 – the Cb component, same height as the source frame, except for YUV422P,
it is half the height of the source frame, rounded up

Description

Returns a frame that contains a single plane of the planar frame *frh*. Returns NULL if *frh* is not planar. The frame returned contains Y8 data for all the planar types generated by the Frame library. The returned frame has a width and height less than or equal to that of the source frame.

For YUV planar formats, plane 0 is the Y component, plane 1 is the Cr component, and plane 2 is the Cb component. In YUV422P format, plane 0 is the same width and height as the source frame, while both planes 1 and 2 are the height of the source frame by half the width (rounded up).

The frame returned by `ExtractPlane` does not need to be freed by `FreeFrame`, and calling `FreeFrame` on a frame with a single plane will cause the function to return without doing anything. All planes extracted from a frame immediately become invalid when the original frame is freed.

This function is also used to access a single frame from within a buffer list. A buffer list is allocated with `AllocateBufferList`. The *n*th frame must be extracted from the list with a call such as:

```
frameN = frameLib. ExtractPlane(frh, N);
```

The frame returned by `ExtractPlane` does not need to be freed by `FreeFrame`, and can be

used in any of the PXD or Frame library functions that are a frame handle.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

AllocateBuffer (pxd library) to allocate a planar frame

AllocateBufferList (pxd library) to allocate a list of buffers

Grab (frame library) to grab an image into the frame

FrameAddress

```
unsigned long FrameAddress(FRAME *frh);
```

Return Value

0

Description

This function is for compatibility only, and always returns 0.

FrameBuffer

```
void *FrameBuffer(FRAME *frh);
```

Return Value

The logical address of the frame's image buffer; 0 if the frame handle is invalid.

Parameters

frh

handle to the frame buffer

Description

Returns a pointer to the start of the data buffer of the specified frame, or NULL if the data is not in the program's address space. An application can use this pointer to access a frame's image data.

For RGB, float, and Y images, the image is stored sequentially in the frame buffer, with the upper-left pixel at the start of the buffer, and the lower-right pixel at the end of the buffer. The size of the buffer is `frame_width * frame_height * bytes_per_pixel`, where the bytes per pixel can be found with the following table

| Frame Type | Bytes Per Pixel |
|-------------|-------------------|
| PBITS_Y8 | 1 |
| PBITS_Y16 | 2 |
| PBITS_Yf | sizeof(float) |
| PBITS_RGB15 | 2 |
| PBITS_RGB16 | 2 |
| PBITS_RGB24 | 3 |
| PBITS_RGB32 | 4 |
| PBITS_RGBf | 3 * sizeof(float) |

For the YUV images, the different color planes of the image can be accessed via the **ExtractPlane()** function.

One interesting aspect about the format of the frame buffer, for RGB24 and Y8 formats, is that they can be treated as the buffer to a Windows Device Independent Bitmap. This makes it possible to use the frame buffer in the DIBits functions. The following example demonstrates how to display a frame buffer in a window. The primary difference is that a DIB with a positive height is stored as bottom-up, but the frame buffer is stored as top-down. By treating the DIB to have a negative height, it will be understood by the DIBits functions to be top-down.

Example

```

// structure that is identical to the Windows bitmap header, except that
// it supports a full palette, rather than one palette entry. It can
// always be cast as a BITMAPINFO and work correctly
typedef struct {
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD          bmiColors[256];
} XBITMAPINFO;

void ShowFrameInWindow(HWND hWnd,
                      FRAME * pfrh,
                      FRAMELIB * pFrameLib)
{
    // we will treat the contents of the frame buffer as the source of
    // data for the DIBits functions.

    BYTE * pbBuffer;           // pointer to the frame buffer
    XBITMAPINFO BitmapInfo;    // description of the DIB
    short sType;               // type of the frame library
    short sWidth;              // width of frame
    short sHeight;             // height of frame
    short sCounter;            // counter when needed
    HDC hDC;                   // Device Context of the window
    RECT Rect;                 // client region of the window
    HPALETTE hPalette;         // our palette, if we are displaying on 8 Bpp
    HPALETTE hOrgPalette;     // original palette

    sType = pFrameLib->FrameType(pfrh);

    if (sType != PBITS_Y8 && sType != RGB24){
        return;
    }
    // at this point, we know that we have one of the bit depths supported
    // by DIBs - 8 or 24 bit

    // describe the bitmap info header, so that the DIBits functions
    // understand the layout of the image
    sWidth = pFrameLib->FrameWidth(pfrh);
    sHeight = pFrameLib->FrameHeight(pfrh);
    memset(&BitmapInfo, 0, sizeof(BITMAPINFO));
    BitmapInfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    BitmapInfo.bmiHeader.biWidth = sWidth;
    BitmapInfo.bmiHeader.biHeight = -sHeight; // negative to force top-down
    BitmapInfo.bmiHeader.biPlanes = 1;

```



```

}

// the fastest and cleanest method to scale the image is to
// set the stretch mode to COLORONCOLOR. This prevents aliasing
// which can cause odd color artifact.
SetStretchBltMode(hDC, COLORONCOLOR);

// we need to know the size of the window, so the function can
// correctly scale the image
GetClientRect(hWnd, &Rect);

// for this example, we will scale the image to fill the window.
// Other techniques are possible, such as zooming in and out,
// and scaling-to-fit, where the aspect ratio is kept the same,
// regardless of the shape of the window
StretchDIBits(hDC,           // handle of device context
              0,             // x coordinate of upper-left corner of dest.
              0,             // y-coordinate of upper-left corner of dest.
              Rect.right,    // width of destination rectangle
              Rect.bottom,   // height of destination rectangle
              0,             // x-coordinate of lower-left corner of source
              sHeight,       // y-coordinate of lower-left corner of source
              sWidth,        // source rectangle width
              sHeight,       // source rectangle height
              pbBuffer,      // address of array with DIB bits
              (BITMAPINFO *)&BitmapInfo,
              DIB_RGB_COLORS, // RGB or palette indices
              SRCCOPY         // raster operation code
            );

// we are done with any palette we created, so release it
if (hPalette != NULL){
    // restore the original palette, thus releasing the use of
    // our palette. Then we can destroy it
    SelectObject(hDC, hOrgPalette);
    DeleteObject(hPalette);
}
// we are now done with the DC, so we can release it
ReleaseDC(hWnd, hDC);
}

// This routine forces the palette to what we need
HPALETTE CreateOurPalette(BitmapInfo.bmiColors)
{
    // from the example "Using DirectDraw Palettes in Windowed Mode"

```

```
// in the Windows SDK
typedef struct {
    WORD          palVersion;
    WORD          palNumEntries;
    PALETTEENTRY palPalEntry[256];
} XLOGPALETTE;

DWORD dwCtr = 0;
XLOGPALETTE Palette;

Palette.palVersion = 0x300;
Palette.palNumEntries = 256;
for (dwCounter = 0; dwCounter < 256; dwCounter++)
{
    if (dwCounter < 10 || dwCounter > 245)
    {
        Palette.palPalEntry[dwCtr].peFlags = PC_EXPLICIT;
        Palette.palPalEntry[dwCtr].peRed = (unsigned char)dwCtr;
        Palette.palPalEntry[dwCtr].peGreen = 0;
        Palette.palPalEntry[dwCtr].peBlue = 0;
    } else {
        Palette.palPalEntry[dwCtr].peFlags = PC_NOCOLLAPSE;
        Palette.palPalEntry[dwCtr].peRed = lpPalette[dwCounter].rgbRed;
        Palette.palPalEntry[dwCtr].peGreen = lpPalette[dwCounter].rgbGreen;
        Palette.palPalEntry[dwCtr].peBlue = lpPalette[dwCounter].rgbBlue;
    }
}
return CreatePalette((LOGPALETTE *) &Palette);
}
```

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

ExtractPlane (frame library) to access the buffer of a planar frame
FrameType (frame library) to identify the pixel format of the frame
FrameHeight (frame library) to identify the height of the frame

FrameWidth (frame library) to identify the width of the frame

FrameFromPointer

```
FRAME *FrameFromPointer (void *ptr, short dx, short dy,  
unsigned short type);
```

Return Value

A pointer to the frame structure; NULL on failure.

Parameters

ptr

a pointer to a buffer of data that is to be made accessible to the frame library. This pointer refers to logical memory – not physical memory.

dx

width of the image in the buffer

dy

height of the image in the bufer

type

the pixel type of the aliased frame. This does not change the contents of the frame, so should typically be the same type as the frame *frh*.

| | |
|-------------|---|
| PBITS_Y8 | 8-bit gray scale stored in an unsigned char |
| PBITS_Y16 | 16-bit gray scale stored in an unsigned short |
| PBITS_Yf | the pixels are grayscale values, stored in the buffer as floats. This is a non-standard color format, so it is up to the application to interpret these values. |
| PBITS_RGB15 | 15-bit RGB, packed into 16 bits. Stored as upper bit is ignored, next 5 are Red, next 5 are Green, and lower 5 bits are Blue. |
| PBITS_RGB16 | 16-bit RGB. Stored as upper 5 bits are Red, middle 6 bits are Green, and lower 5 bits are Blue. |
| PBITS_RGB24 | 24-bit RGB. Stored as three bytes, arranged as Red/Green/Blue. |
| PBITS_RGB32 | 8/8/8 Red/Green/Blue color stored in the lower 24 bits of a long. |
| PBITS_RGBf | the pixels are color values, stored in the buffer as three floats. This is a non-standard color format, so it is up to |

| | |
|---------------|--|
| | the application to interpret these values. |
| PBITS_YUV422 | 8 bits for Y, and 8 bits for CrCb.. |
| PBITS_YUV444 | 8 bits each for Y, Cr, and Ch |
| PBITS_YUV422P | YUV422 in planar format |
| PBITS_YUV444P | YUV444 in planar format |

Description

Creates a frame of size dx by dy , with the specified pixel *type*, from the specified buffer. Both dx by dy must be greater than zero. The start of the image buffer will be aligned to at least a 32-bit boundary in the program's address space. This frame must have a width that is a multiple of 4 to be useful as a frame buffer.

This function does not create a new buffer – the frame will use the buffer referred to by *ptr*. It does create a frame header, so **FreeFrame** should be called when the frame is no longer needed.

The advantage of this function is that it allows a user-defined buffer to be treated as a frame. For example, this could be a DirectDraw surface, or a shared memory buffer.

Example

See **CopyFrame** for an example.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
 frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

FreeFrame (frame library) to free the memory used by the alias frame

AllocateBuffer (pxd library) to create a buffer that is guaranteed to work with the frame grabber

FrameHeight

```
short FrameHeight(FRAME *frh);
```

Return Value

The height of the frame in pixels; 0 if the frame handle is invalid.

Parameters

frh
handle to a frame buffer

Description

Returns the height of a frame created with any of the Allocate functions

Example

See the example in **FrameBuffer**.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

FrameBuffer (frame library) to get the frame buffer
FrameType (frame library) to identify the pixel format of the frame
FrameWidth (frame library) to identify the width of the frame

FrameType

```
short FrameType(FRAME *frh);
```

Return Value

The pixel data type of the frame; 0 if the frame handle is invalid.

Parameters

frh

handle to the frame buffer

| | |
|---------------|---|
| PBITS_Y8 | 8-bit gray scale stored in an unsigned char |
| PBITS_Y16 | 16-bit gray scale stored in an unsigned short |
| PBITS_Yf | the pixels are grayscale values, stored in the buffer as floats. This is a non-standard color format, so it is up to the application to interpret these values. |
| PBITS_RGB15 | 15-bit RGB, packed into 16 bits. Stored as upper bit is ignored, next 5 are Red, next 5 are Green, and lower 5 bits are Blue. |
| PBITS_RGB16 | 16-bit RGB. Stored as upper 5 bits are Red, middle 6 bits are Green, and lower 5 bits are Blue. |
| PBITS_RGB24 | 24-bit RGB. Stored as three bytes, arranged as Red/Green/Blue. |
| PBITS_RGB32 | 8/8/8 Red/Green/Blue color stored in the lower 24 bits of a long. |
| PBITS_RGBf | the pixels are color values, stored in the buffer as three floats. This is a non-standard color format, so it is up to the application to interpret these values. |
| PBITS_YUV422 | 8 bits for Y, and 8 bits for CrCb.. |
| PBITS_YUV444 | 8 bits each for Y, Cr, and Ch |
| PBITS_YUV422P | YUV422 in planar format |
| PBITS_YUV444P | YUV444 in planar format |

Description

Returns the pixel data type of the frame created with any of the Allocate functions.

Example

See **Framebuffer** for an example

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

Framebuffer (frame library) to get a pointer to the frame buffer

FrameHeight (frame library) to get the height of the frame buffer

FrameWidth (frame library) to get the width of the frame buffer

FrameWidth

```
short FrameWidth(FRAME *frh);
```

Return Value

The width of the frame in pixels; 0 if the frame handle is invalid.

Parameters

frh
handle to a frame buffer

Description

Returns the width of a frame created with any of the Allocate functions.

Example

See **FrameBuffer** for an example.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

FrameBuffer (frame library) to get the frame buffer

FrameHeight (frame library) to identify the height of the frame

FrameType (frame library) to identify the pixel format of the frame

FreeFrame

```
void FreeFrame(FRAME *frh);
```

Return Value

None.

Parameters

frh
handle to a frame buffer

Description

Returns memory associated with a FRAME handle to the system. You must free all frames allocated by `AllocateBuffer`, `AllocateMemoryFrame` and `FrameFromPointer`, before calling `CloseLibrary`

This function is identical to the `FreeFrame` function in the PXD Frame Grabber library. Either version of the function can free a frame allocated by either library.

Example

See `AllocateMemoryFrame` for an example.

Required Compile-time Headers

`frame.h` for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

`frame_32.lib` for Windows NT, 98 and 95
`frame_fw.lib` for DOS4GW

Required Run-time Libraries

`frame_32.dll` for Windows NT, 98 and 95

Also See

`AllocateBuffer` (pxd library) to create a buffer that is guaranteed to work with the frame grabber

`AllocateMemoryFrame` (frame library) to create a generic frame buffer

GetColumn

```
short GetColumn(FRAME *frh, void *buf, short column);
```

Return Value

Non-zero if successful; 0 on failure.

Parameters

frh

handle to a frame buffer

buf

pointer to a buffer to receive the column of pixels

column

a number indicating the column to pull the pixels from. The leftmost column is 0.

Description

Copies a column of the image stored in frame *f* into the buffer *buf*. The columns are numbered starting with 0 at the left of the frame. The buffer is assumed to be an array of the correct type to hold the column of pixels. If the entire column won't fit in the memory pointed to by *buf* undefined behavior and data corruption might result.

GetColumn will fail if the specified column is outside the boundaries of the frame, if the frame can't be read, or if the frame contains planar data.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

GetPixel (frame library) to get a single pixel from the frame

GetRectangle (frame library) to get a block of pixels from the frame

GetRow (frame library) to get a row of pixels from the frame

PutColumn (frame library) to set a column of pixels in the frame

PutPixel (frame library) to set a pixel in the frame

PutRectangle (frame library) to set a block of pixels in the frame

PutRow (frame library) to set a row of pixels in the frame

GetPixel

```
short GetPixel(FRAME *frh, void *pixel, short x, short y);
```

Return Value

Non-zero if successful; 0 on failure.

Parameters

frh
handle to a frame buffer

pixel
pointer to a pixel of the correct type

x
column of the pixel – top is 0

y
row of the pixel – left is 0

Description

Copies the pixel at (x,y) into *pixel*, where (0,0) is the upper-left corner of the frame. The parameter *pixel* is assumed to point to a variable or structure of the correct type to hold the pixel. If *pixel* doesn't point to an object of sufficient size to hold the pixel, undefined behavior and data corruption might result. If the frame is planar, *pixel* must point to an object that can hold one pixel from each plane, appended in order (example: YUV422P frames require a byte of brightness, followed by a byte of red, followed by a byte of blue, for a total of 24 bits).

If the point specified by (x,y) is outside the boundaries of the frame, or the frame can't be read, the function call fails.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

- GetColumn** (frame library) to get a column of pixels from the frame
- GetRectangle** (frame library) to get a block of pixels from the frame
- GetRow** (frame library) to get a row of pixels from the frame
- PutColumn** (frame library) to set a column of pixels in the frame
- PutPixel** (frame library) to set a pixel in the frame
- PutRectangle** (frame library) to set a block of pixels in the frame
- PutRow** (frame library) to set a row of pixels in the frame

GetRectangle

```
short GetRectangle(FRAME *frh, void *buf, short x0, short
y0, short dx, short dy);
```

Return Value

Non-zero if successful; 0 on failure.

Parameters

frh

handle to the frame buffer

buf

address of a buffer to receive the data. It must be large enough to receive the data.

x0

the left column of the rectangle within the frame. The leftmost column of the frame is 0.

y0

the top row of the rectangle within the frame. The topmost row of the frame is 0.

dx

the width of the rectangle

dy

the height of the rectangle

Description

Copies a rectangular region of the frame *f* into the buffer *buf*. The rectangle has upper left corner (*x0,y0*) in the source frame, width *dx*, and height *dy*. The buffer is assumed to be an array of the correct type to hold the row of pixels. If the entire rectangle won't fit in the memory pointed to by *buf* undefined behavior and data corruption might result. If the region is partially outside the boundaries of the frame, *GetRectangle* will copy only the parts of the rectangle that are within the frame..

GetRectangle will fail if the specified rectangle is entirely outside the boundaries of the frame, if the frame can't be read, or if the frame contains planar data.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

GetColumn (frame library) to get a column of pixels from the frame

GetPixel (frame library) to get a single pixel from the frame

GetRow (frame library) to get a row of pixels from the frame

PutColumn (frame library) to set a column of pixels in the frame

PutPixel (frame library) to set a pixel in the frame

PutRectangle (frame library) to set a block of pixels in the frame

PutRow (frame library) to set a row of pixels in the frame

GetRow

```
short GetRow(FRAME *frh, void *buf, short row);
```

Return Value

Non-zero if successful; 0 on failure.

Parameters

frh

handle to a frame buffer

buf

address of a buffer to receive the row. It must be large enough to receive the data.

row

the row number from which to get the data. The topmost row of the frame is 0.

Description

Copies a row of the image stored in frame *f* into the buffer *buf*. The rows are numbered starting with 0 at the top of the frame. The buffer is assumed to be an array of the correct type to hold the row of pixels. If the entire row won't fit in the memory pointed to by *buf* undefined behavior and data corruption might result.

GetRow will fail if the specified row is outside the boundaries of the frame, if the frame can't be read, or if the frame contains planar data.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

GetColumn (frame library) to get a column of pixels from the frame

GetPixel (frame library) to get a single pixel from the frame

GetRectangle (frame library) to get a block of pixels from the frame

PutColumn (frame library) to set a column of pixels in the frame

PutPixel (frame library) to set a pixel in the frame

PutRectangle (frame library) to set a block of pixels in the frame
PutRow (frame library) to set a row of pixels in the frame

OpenLibrary

DOS Syntax `short FRAME_OpenLibrary(FRAMELIB __PX_FAR
 *interface, short size);`

Win C Syntax `short imagenation_OpenLibrary(char * dllname,
 void * interface, short size);`

Win VB Syntax `integer OpenLibrary(0,0)`

Return Value

Non-zero if successful; 0 on failure.

Parameters

dllname

name of the DLL providing the interface. For the Frame library, this is "frame_32.dll"

interface

a pointer to the variable being filled with the interface

size

the size of the interface. For the Frame library, this is `sizeof(FRAME)`

Description

Initializes library data structures. It must be called successfully before any other library functions can be used.

Be sure to close the library before exiting the application, with `CloseLibrary()`.

Example

For an example, see `AllocateMemoryFrame`.

Required Compile-time Headers

`frame.h` for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

`frame_32.lib` for Windows NT, 98 and 95
`frame_fw.lib` for DOS4GW

Required Run-time Libraries

`frame_32.dll` for Windows NT, 98 and 95

Also See

`CloseLibrary` (frame library) to close the library

PutColumn

```
void PutColumn(void *buf, FRAME *frh, short col);
```

Return Value

Non-zero if successful; 0 on failure.

Parameters

buf

address of the buffer containing the pixels to be placed in the column.

frh

handle to the frame buffer

col

the column in the frame which will receive the pixels. The leftmost column of the frame is 0;

Description

Copies the data stored in the buffer *buf* into a column of frame *f*. The columns are numbered starting with 0 at the left of the frame. The buffer is assumed to be an array of the correct type to hold the column of pixels. If *buf* doesn't point to enough data to hold an entire column, undefined behavior and illegal memory accesses might result.

PutColumn will fail if the specified column is outside the boundaries of the frame, if the frame can't be written, or if the frame contains planar data.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

GetColumn (frame library) to get a column of pixels from the frame

GetPixel (frame library) to get a single pixel from the frame

GetRectangle (frame library) to get a block of pixels from the frame

GetRow (frame library) to get a row of pixels from the frame

PutPixel (frame library) to set a pixel in the frame

PutRectangle (frame library) to set a block of pixels in the frame

PutRow (frame library) to set a row of pixels in the frame

PutPixel

```
short PutPixel(void *pixel, FRAME *frh, short x, short y);
```

Return Value

Non-zero if successful; 0 on failure.

Parameters

pixel

address of the pixel data to be placed in the frame

frh

handle to a frame buffer

x

the column of the frame which will receive the pixel. The leftmost column is 0.

y

the row of the frame which will receive the pixel. The topmost row is 0.

Description

Copies the data pointed to by *pixel* into location (x,y) in the frame, where 0,0 is the upper-left corner of the frame. The parameter *pixel* is assumed to point to a variable or structure of the correct type to hold the pixel. If *pixel* doesn't point to an object of sufficient size to hold the pixel, unde-fined behavior and illegal memory accesses might result. If the frame is planar, *pixel* must point to an object that holds one pixel from each plane, appended in order (example: YUV422P frames require a byte of brightness, followed by a byte of red, followed by a byte of blue, for a total of 24 bits).

If the point specified by (x,y) is outside the boundaries of the frame, or the frame can't be read, the function call fails.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

GetColumn (frame library) to get a column of pixels from the frame

GetPixel (frame library) to get a single pixel from the frame

GetRectangle (frame library) to get a block of pixels from the frame

GetRow (frame library) to get a row of pixels from the frame

PutColumn (frame library) to set a column of pixels in the frame

PutRectangle (frame library) to set a block of pixels in the frame

PutRow (frame library) to set a row of pixels in the frame

PutRectangle

```
void PutRectangle(void *buf, FRAME *frh, int x0, short  
y0, short dx, short dy);
```

Return Value

Non-zero if successful; 0 on failure.

Parameters

buf

address of the buffer containing the pixels to be placed in the frame. It must hold at $dx * dy$ pixels of the correct type.

frh

handle to a frame buffer

x0

the leftmost column of the frame buffer to be filled with the pixels.

y0

the topmost column of the frame buffer to be filled with the pixels.

dx

the width of the rectangle to be filled with the pixels.

dy

the height of the rectangle to be filled with the pixels.

Description

Copies a rectangular region from buffer *buf* into the frame *f*. The rectangle goes into the frame with its upper left corner at $(x0, y0)$, width *dx*, and height *dy*. The buffer is assumed to be an array of the correct type to hold the rectangle of pixels as a series of concatenated lines. If *buf* doesn't point to enough data to hold the entire rectangle, undefined behavior and illegal memory accesses might result. If the specified rectangle is partly outside the frame boundaries, only the data within the frame boundaries is written.

PutRectangle fails if the specified rectangle is entirely outside the boundaries of the frame, if the frame can't be written, or if the frame contains planar data.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

GetColumn (frame library) to get a column of pixels from the frame

GetPixel (frame library) to get a single pixel from the frame

GetRectangle (frame library) to get a block of pixels from the frame

GetRow (frame library) to get a row of pixels from the frame

PutColumn (frame library) to set a column of pixels in the frame

PutPixel (frame library) to set a pixel in the frame

PutRow (frame library) to set a row of pixels in the frame

PutRow

```
short PutRow(void *buf, FRAME *frh, short row);
```

Return Value

Non-zero if successful; 0 on failure.

Parameters

buf

address of the buffer containing the row of pixels to be put into the frame

frh

handle to the frame buffer

row

the row number of the frame to receive the pixels.

Description

Copies the data stored in the buffer *buf* into a row of frame *f*. The rows are numbered starting with 0 at the top of the frame. The buffer is assumed to be an array of the correct type to hold the row of pixels. If *buf* doesn't point to enough data to hold an entire row, undefined behavior and illegal memory accesses might result.

PutRow will fail if the specified row is outside the boundaries of the frame, if the frame can't be written, or if the frame contains planar data.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

GetColumn (frame library) to get a column of pixels from the frame

GetPixel (frame library) to get a single pixel from the frame

GetRectangle (frame library) to get a block of pixels from the frame

GetRow (frame library) to get a row of pixels from the frame

PutColumn (frame library) to set a column of pixels in the frame

PutPixel (frame library) to set a pixel in the frame

PutRectangle (frame library) to set a block of pixels in the frame

ReadBin

```
short ReadBin(FRAME *frh, char *filename);
```

Return Value

The Return Values are:

| | |
|-----------------|---|
| SUCCESS | The file was read successfully. |
| FILE_OPEN_ERROR | The specified file could not be opened. |
| BAD_READ | An error occurred while a file was being read. |
| BAD_FILE | The file being read is not of the correct format. |
| INVALID_FRAME | The frame pointer is invalid or the frame data can't be accessed. |
| FRAME_SIZE | The frame is not large enough to hold the data being read.. |

Parameters

frh

handle to a frame buffer

filename

name of a file containing a binary image.

Description

Reads the unformatted binary file *filename* and copies it into frame buffer *f*. The function stores as much of the contents of the file in the buffer as will fit. If the type of data in the file does not match the data type of the frame, the data will be interpreted as if it were in the frame's data format. For planar frames, each plane is read from the file in order. If the data in the file is too large to fit in the frame, the function reads as much data as will fit and returns the FRAME_SIZE error. If the file doesn't contain enough data to fill the frame, the entire file is read, the remainder of the frame is set to zero, and the function returns the FRAME_SIZE error.

ReadBin opens and closes the file.

If the frame's width, height, and frame type do not match the width, height, and frame type of the image that was written into the BIN file, the image will not appear correct. It will probably appear to have the wrong colors – indicative of the wrong frame type – or will be skewed at an angle – indicative of the wrong width. If the height is wrong, the

image will appear to be chopped off, or have garbage at the bottom.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

WriteBin

ReadBMP

```
short ReadBMP(FRAME *frh, char *filename);
```

Return Value

The Return Values are:

| | |
|-----------------|---|
| SUCCESS | The file was read successfully. |
| FILE_OPEN_ERROR | The specified file could not be opened. |
| BAD_READ | An error occurred while a file was being read. |
| BAD_FILE | ReadBMP attempted to read a non-BMP-for-matted file. |
| INVALID_FRAME | The frame pointer is invalid or the frame data can't be accessed. |
| FRAME_SIZE | The frame is not large enough to hold the data being read.. |

Parameters

frh

handle to a frame buffer

filename

name of the .BMP to read

Description

Reads the image stored in the BMP file *filename* and copies it into frame buffer *f*. Y8 images are read from 8-bit-per-pixel BMP files, RGB images are read from 24-bit, true-color BMP files, with low-order bits discarded to match the RGB pixel type format as necessary. Attempting to read files with any other pixel format results in an error.

If the frame is larger than the image data in the file, the data appears in the upper-left corner of the frame with the remainder of the frame set to zero. If the frame is smaller than the image, the upper-left portion of the image is read into the frame, and the FRAME_SIZE error is returned.

ReadBMP opens and closes *filename*.

This function can read .BMP's created by other applications, so long as they are created according to Microsoft's specification of a bitmap file.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

WriteBMP (frame library) to save a bitmap file

ReadPNG

```
short ReadBMP(FRAME *frh, char *filename);
```

Return Value

The Return Values are:

| | |
|-----------------|---|
| SUCCESS | The file was read successfully. |
| FILE_OPEN_ERROR | The specified file could not be opened. |
| BAD_READ | An error occurred while a file was being read. |
| BAD_FILE | ReadPNG attempted to read a non-PNG-formatted file. |
| INVALID_FRAME | The frame pointer is invalid or the frame data can't be accessed. |
| FRAME_SIZE | The frame is not large enough to hold the data being read.. |

Parameters

frh

handle to a frame buffer

filename

name of the .PNG to read

Description

Reads the image stored in the PNG file *filename* and copies it into frame buffer *f*. Y8 images are read from 8-bit-per-pixel PNG files, Y16 images are read from 16 bit-per-pixel PNG files, RGB images are read from 24-bit, true-color PNG files, with low-order bits discarded to match the RGB pixel type format as necessary. Attempting to read files with any other pixel format results in an error.

If the frame is larger than the image data in the file, the data appears in the upper-left corner of the frame with the remainder of the frame set to zero. If the frame is smaller than the image, the upper-left portion of the image is read into the frame, and the FRAME_SIZE error is returned.

ReadPNG opens and closes *filename*.

This function can read PNG's created by other applications, so long as they are created according to the official PNG specification.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

WriteBMP (frame library) to save a bitmap file

WriteBin

```
short WriteBin(FRAME *frh, char *filename, short overwrite);
```

Return Value

The Return Values are:

| | |
|-----------------|---|
| SUCCESS | The file was written successfully. |
| FILE_EXISTS | The file already exists, but the function call did not specify that the file should be overwritten. |
| FILE_OPEN_ERROR | The file could not be opened. |
| BAD_WRITE | An error occurred while a file was being written. |
| INVALID_FRAME | The frame pointer is invalid or the frame's data can't be accessed.. |

Parameters

frh

handle to a frame buffer containing an image to be written

filename

the name of the file to be saved

overwrite

if 0, the file will not be overwritten if it already exists. If other than 0, an existing file with the same name will be overwritten.

Description

Writes the image in frame buffer *f* to the file *filename*. No information about the image (height, width, and bits per pixel) is written, only the pixel values. Data in the file exactly matches the format of the data in memory. Planar frames are written to the file plane by plane.

If *filename* already exists and *overwrite* is zero, the function returns an error; otherwise, the contents of *filename* are overwritten. WriteBin opens and closes the file.

The dimensions and frame type of the frame are not saved with the file. It will be necessary to record this information elsewhere, should it be necessary to read the file.

Other file functions, such as WriteBMP or WritePNG, do save the dimensions and frame type with the file.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95

frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

ReadBin (frame library) to read a binary file

WriteBMP

```
short WriteBMP(FRAME *frh, char *filename, short over-  
write);
```

Return Value

The Return Values are:

| | |
|-----------------|---|
| SUCCESS | The file was written successfully. |
| FILE_EXISTS | The file already exists, but the function call did not specify that the file should be overwritten. |
| FILE_OPEN_ERROR | The file could not be opened. |
| BAD_WRITE | An error occurred while a file was being written. |
| INVALID_FRAME | The frame pointer is invalid or the frame data can't be accessed. |
| WRONG_BITS | The file format does not accept data of the type contained in the frame |

Parameters

frh

handle to a frame buffer containing an image to be written

filename

the name of the file to be saved

overwrite

if 0, the file will not be overwritten if it already exists. If other than 0, an existing file with the same name will be overwritten.

Description

Writes the image stored in frame buffer *f* to the file *fname* in the BMP format. Y8 images are written as 8-bits-per-pixel BMP files with a gray-scale palette. RGB images are written as 24-bit, true-color BMP files. Any alpha channel data is ignored. Attempting to write floating-point formats, Y16, and YUV formats results in an error. To write frames of other frame types, use WriteBin or WritePNG.

If *filename* already exists and *overwrite* is zero, the function returns an error; otherwise, the contents of *filename* are overwritten. WriteBMP opens and closes the file *filename*.

The files written by WriteBMP can be read by other applications, so long as they can interpret files that meet Microsoft's definition of the BMP file format.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

ReadBMP

WritePNG

```
short WritePNG(FRAME *frh, char *filename, short over-  
write);
```

Return Value

The Return Values are:

| | |
|-----------------|---|
| SUCCESS | The file was written successfully. |
| FILE_EXISTS | The file already exists, but the function call did not specify that the file should be overwritten. |
| FILE_OPEN_ERROR | The file could not be opened. |
| BAD_WRITE | An error occurred while a file was being written. |
| INVALID_FRAME | The frame pointer is invalid or the frame data can't be accessed. |
| WRONG_BITS | The file format does not accept data of the type contained in the frame |

Parameters

frh

handle to a frame buffer containing an image to be written

filename

the name of the file to be saved

overwrite

if 0, the file will not be overwritten if it already exists. If other than 0, an existing file with the same name will be overwritten.

Description

Writes the image stored in frame buffer *f* to the file *fname* in the PNG format. Y8 images are written as 8-bits-per-pixel BMP files with a gray-scale palette. Y16 images are written with 16 bit pixels, with full grayscale information. RGB images are written with 24-bit, true-color pixels. Attempting to write floating-point formats, and YUV formats results in an error. To write frames of other frame types, use WriteBin.

If *filename* already exists and *overwrite* is zero, the function returns an error; otherwise, the contents of *filename* are overwritten. WritePNG opens and closes the file *filename*.

The files written by WritePNG can be read by other applications, so long as they can interpret files that meet the official PNG file format.

Required Compile-time Headers

frame.h for DOS4GW, Windows NT, 98 and 95

Required Link-time Libraries

frame_32.lib for Windows NT, 98 and 95
frame_fw.lib for DOS4GW

Required Run-time Libraries

frame_32.dll for Windows NT, 98 and 95

Also See

ReadPNG (frame library) to read the PNG



Glossary

Bits/Pixel

For a monochrome camera this describes the maximum number of resolvable gray levels that the camera can provide. Eight bits per pixel (256 gray levels) is quite common with 10, 12 and 14 bits/pixel available in some models. A distinct advantage of a digital camera results from having the digital-to-analog converter moved from the frame grabber card into the camera. This reduces effect of transmission line noise on the quality of the image, making the least significant bits in each pixel more meaningful.

Burst PCI Rate versus Sustained PCI Rate

A bus master can burst data across a well designed PCI bus at 132MB/second. Other users of the bus can request and gain access to the bus, lowering the sustained performance. The ability of a frame grabber to sustain data transfers without losing data is related to the ability of the grabber to buffer data while another user has control of the bus. The higher the input data rate from the camera, the more the grabber needs to buffer. The maximum transfer rate that a grabber can sustain is related to how efficient the buffering scheme on the board is and how efficient the PCI interface is.

Channels

As the resolution of the image sensor and the bits/pixel and the frame rate increase, larger and larger amounts of image data must be transferred to the frame grabber. To keep frame rates high many digital cameras deliver image data via multiple (synchronized) digital outputs, called channels. Each channel is used to transfer only a portion of the image information. For example the Dalsa CA-D4 is 1024 x 1024, 8-bit/pixel camera can operate as either a one or two channel camera. With a single channel it can deliver 25-million pixels/second, at 21 frames/second. But by switching to 2-channel mode, where each channel transmits 25-million pixels/second, the frame rate increases to 40 fps.

In order for a digital frame grabbers to be able to receive more than one pixel at a time it must first have a digital input port that is wide enough to handle the number of simultaneous data bits the camera is transmitting. In two channel mode, the Dalsa CA-D4 transmits two 8-bit pixels on each clock. A digital grabber with at least a 16-bit input port would be required. But input data port width is not necessarily enough to insure that the captured data can be delivered to the application in a ready to use form.

EIA422-B vs. EIA-644

These are “balanced” data transmission standards that require two wires per signal. The state of the signal at the receiver is determined by the potential difference between the two wires and not by the difference between the signal on a single wire and ground. Since each wire in the pair is subjected to roughly the same transmission environment, electrical noise adds equally to both wires. This “common mode” noise is subtracted at the receiver. This makes both of these standards particularly useful in noisy environments. EIA-644 operates at lower voltage differences than EIA-422 providing higher transmission bandwidths. EIA-644 transmitters and receivers also introduce less line-to-line skew meaning that signal integrity is better preserved even when the transmitter is EIA-422 and the receiver is EIA-644.

Frame Rate

Refers to the sustained rate at which a camera can generate images. It is usually the longer of the exposure time or the image transfer time.

Input Look-Up Tables (LUTs)

LUTs are useful for several pixel operations that free the processor from mundane pixel mapping. Typical uses include

- applying a gamma correction
- mapping the input pixel values to another set of values
- performing a threshold operation to produce a binary image

Pixel Clock Source

Most digital cameras provide their own pixel clock to the frame grabber but in situations where a custom frame and pixel rate are required, the frame grabber must create the pixel clock for the camera. A pixel clock source provides more flexibility for the system integrator.

Pixel Swizzling

The second aspect of multi-channel cameras that can cause problems for digital frame grabbers is the ordering of the received pixels. Figure 1 illustrates how the Dalsa CA-D4 two-channel camera transmits pixels to the frame grabber. Two pixels are received on the first pixel clock; pixel 0 from the top left edge of the image and pixel 1023 (remember there are 1024 pixels/line in the) from the top right. On each successive clock the next pixel received from each channel is from one step in toward the center line.

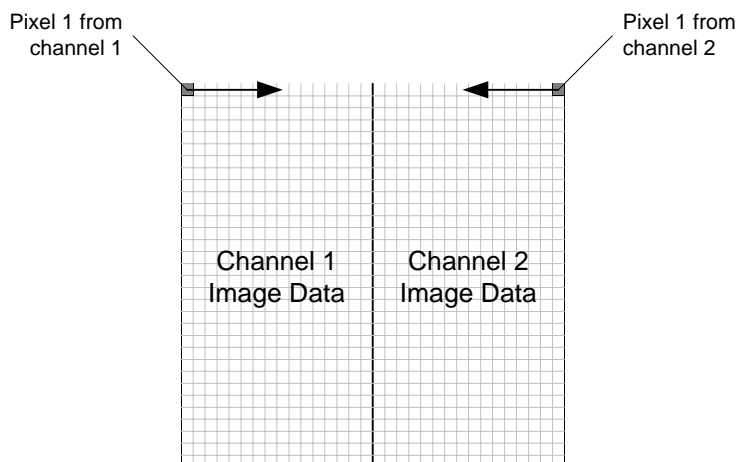


Figure 1.

Channel 1 in the Dalsa CA-D4 (in 2 channel mode) transmits image data beginning at the left edge of the top row continuing to the midpoint at which time it returns to the left edge to begin line two. It continues sending the data line by line until it reaches the bottom of the image. Channel 2 transmits right to left, stopping at the midpoint and in a similar fashion transmitting each half line from top to bottom.

If this information were simply transferred to system memory (Figure2), the end-user application would not have a coherent image but instead would need to de-scramble the image software, a time consuming task

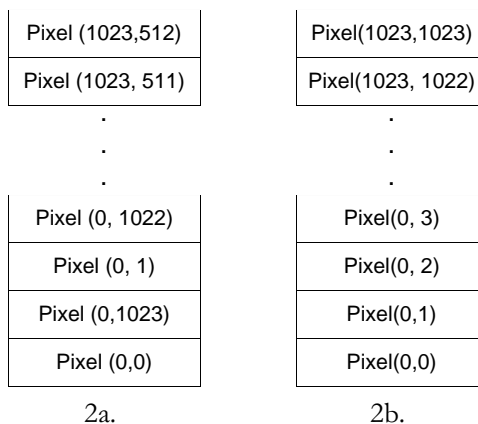
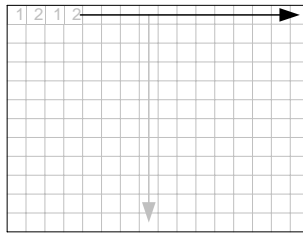


Figure 2a.

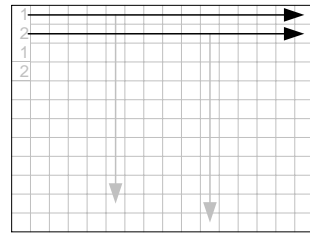
Image data as transferred directly to system memory from a Dalsa CA-D4 two-channel camera and 2b.) after hardware reordering into scan line order by a frame grabber.

To alleviate this problem, many digital frame grabbers incorporate pixel swizzling circuitry to dynamically rearrange the pixels into scan line order so that the application can immediately begin the image processing task.

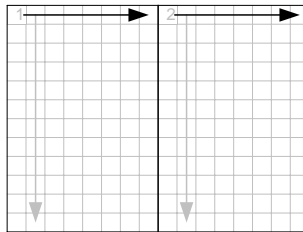
A number of different image-formatting schemes are employed by various cameras. The most common are shown in Figure 3.



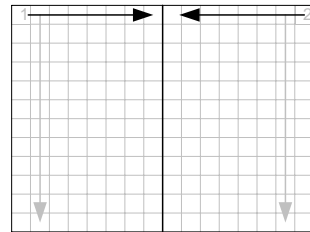
2a) 2-ch. alternate pixels



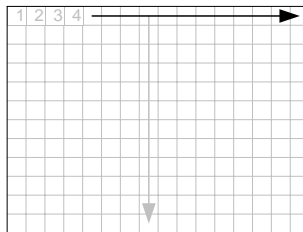
2b) 2-ch. interlaced lines



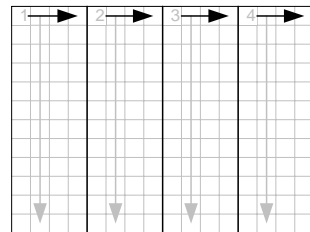
2c) 2-ch. half-lines



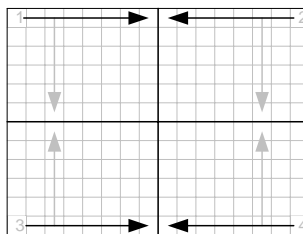
2d) 2-ch. half-lines reverse



2e) 4-ch. alternate pixels



2f) 4-ch. quarter-lines



2g) 4-ch. quadrants double reverse

Figure 3.

A modern digital frame grabber should be able to convert data from cameras with each of these formats into scan line ordered images in system memory.

Resolution

The ability of a camera to resolve details in a scene is dependent on the type of lens employed and the relation of the camera to the scene. Therefore manufacturers often describe the resolution of a camera by the stating number of horizontal and vertical picture elements contained in the image sensor. For example, the Pulnix TM1300 area-scan digital camera has 1300 picture elements in each row by 1030 rows.

Scatter/Gather

Traditionally to get a contiguous block of memory in Windows 95 you had to use a kernel level driver to capture the memory at power-on boot time. When your application was done the memory couldn't be used by other applications until the computer was rebooted without the contiguous block memory request.

On the other hand, requesting contiguous memory in an application at run-time allows the memory to be freed up to other applications when the requestor is finished. But large contiguous blocks of memory might not be available at run-time because as a computer opens and closes applications the memory gets fragmented into smaller blocks.

A frame grabber that supports scatter/gather uses small blocks of memory as if it has a large contiguous block. It does this by building a table of addresses of the small blocks and stepping through the table to fill them with the completed image, so that it appears to the application as one large block.

Index

B

box contents 12

C

cameras 11

- Dalsa CA-D4 11
- Dalsa CA-D7 11
- Hitachi KPF100 11
- Kodak Megaplug 1.4i 11
- Kodak Megaplug ES1.0 11
- Pulnix TM1001-02 11
- Pulnix TM1300 11, 24, 25
- Pulnix TM9701 11
- Reticon LD2040 11

capture an image 32-33

F

functions

- AliasFrame 275
- AllocateAddress 277
- AllocateBuffer 127
- AllocateBufferList 130
- AllocateFG 133
- AllocateFlatFrame 278
- AllocateMemoryFrame 279
- CheckError 136
- CloseLibrary 138, 282
- ContinuousStrobes 139
- CopyFrame 284

- ExtractPlane 289
- FireStrobes 141
- FrameAddress 291
- FrameBuffer 292
- FrameFromPointer 298
- FrameHeight 300
- FrameType 301
- FrameWidth 303
- FreeConfig 143
- FreeFG 145
- FreeFrame 146, 304
- GetActiveFrame 148
- GetBrightness 150
- GetCamera 151
- GetColumn 305
- GetContrast 152
- GetExposureTime 153
- GetFieldCount 154
- GetFramePeriod 155
- GetHeight 156
- GetInputLUT 158
- GetInterface 160
- GetIOType 162
- GetLastFrame 165
- GetLeft 167
- GetModelNumber 169
- GetPixel 307
- GetPixelType 171
- GetRectangle 309
- GetRow 311
- GetStrobePeriod 173
- GetSwitch 175
- GetTop 176

- GetWidth 178
 - GetXResolution 180
 - GetYResolution 182
 - Grab 184
 - GrabContinuous 189
 - GrabTriggered 192
 - IsFinished 194
 - KillQueue 196
 - LoadConfig 198
 - LockFrame 200
 - OpenLibrary 202, 313
 - PutColumn 314
 - PutPixel 316
 - PutRectangle 318
 - PutRow 320
 - QualifyGrabs 205
 - ReadBin 322
 - ReadBMP 324
 - ReadConfigData 207
 - ReadIO 209
 - ReadPNG 326
 - ReadProtection 211
 - ReadRevision 213
 - ReadSerial 215
 - Reset 216
 - SaveConfig 218
 - SetBrightness 220
 - SetCamera 221
 - SetCameraConfig 222
 - SetContrast 224
 - SetExposureTime 225
 - SetFieldCount 227
 - SetFramePeriod 228
 - SetHeight 230
 - SetInputLUT 232
 - SetIOType 234
 - SetLeft 235
 - SetStrobePeriods 237
 - SetTop 239
 - SetTriggerSource 241
 - SetWidth 244
 - SetXResolution 246
 - SetYResolution 247
 - SwitchCamera 248
 - SwitchGrab 249
 - TimedWaitFinished 250
 - TriggerStrobes 252
 - UnlockFrame 255
 - VideoType 256
 - Wait 257
 - WaitAllEvents 259
 - WaitAnyEvent 260
 - WaitFinished 261
 - WaitMultiple 264
 - WaitVB 266
 - WriteBin 328
 - WriteBMP 330
 - WriteConfigData 267
 - WriteImmediateIO 269
 - WritePNG 332
- I**
- installation 11-28
 - DOS 21
 - hardware 16
 - software 22

Windows NT, 98 or 95 19

L

look up table (LUT) 31

R

revision number, displaying 35

S

system requirements 12

T

text overlay, applying 34-35

V

ViewPXD

- capture an image 32-33

- revision number, displaying 35

- text overlay, applying 34-35

- view live video stream 30