PXC200 Precision Color Frame Grabber

Imagenation Corporation
P.O. Box 276
Beaverton, OR 97075-0276

December 1997

P/N MN-200-02

Contents

1 Introduction

The Imagenation PXC200 frame grabber features precision video capture hardware for applications that require high color accuracy. Features of the precision hardware design include:

- High color accuracy with low pixel jitter
- PCI bus master design for real-time image capture to system memory or directly to the VGA display
- Image capture resolution up to full-size: 640 x 480 (NTSC) and 768 x 576 (PAL and SECAM)
- Horizontal and vertical cropping and scaling of captured images to minimize system memory and bus bandwidth requirements
- Common color output formats, including YCrCb, RGB, and Y8 (grayscale)
- Continuous, software-initiated, and triggered image captures
- Four multiplexed composite video inputs (one input can be S-video) with automatic video format detection of NTSC and PAL/SECAM formats
- Digital trigger input
- +12V output for powering cameras or other devices

The optional Control Package for the PXC200 extends the input/output capabilities to include the following:

- Four general-purpose TTL-level input lines and four general-purpose TTL-level output lines
- Vertical and horizontal sync outputs for genlocking a video source
- Strobe inhibit during CCD transfer time for reliable image capture with strobes
- All four video inputs can accept S-video or composite video
- Faster switching between multiplexed video sources with faster video format detection
- DC restore on all four video inputs for instant switching between genlocked video sources

The PXC200 is available in three hardware configurations:

- PCI bus, short card—for typical desktop PC systems
- PC/104-Plus bus—for embedded-systems applications based on the PC/104-Plus format
- CompactPCI—for industrial applications based on the CompactPCI format, which combines the standard PCI electrical specifications with a Eurocard physical format

To make it easy to tap these hardware features, the PXC200 includes an elegant software interface that supports developing applications for 16-bit DOS, Watcom 32-bit DOS/4GW, Windows 3.1, Windows 95 and Windows NT:

- C libraries for building DOS applications
- DLLs for building Windows applications
- DOS VGA Video Display library for building a menu-based user interface
- Sample DOS and Windows source code
- PXCVU—a DOS image capture application

This chapter will give you an introduction to these features. More detailed technical information on features is included in Chapter 4, *Programming the PXC200*, on page 33.

Precision Capture Hardware

The design of the PXC200 video capture hardware produces high color accuracy and low pixel jitter:

Grayscale noise—1.0 LSB RMS maximum

Pixel jitter—±4 ns maximum

This accuracy makes PXC200 frame grabbers ideal for demanding scientific and industrial applications.

Video Inputs and Formats

The PXC200 frame grabber handles multiple camera inputs and video formats:

Connect up to Four Cameras. Switch between camera inputs in software. All four inputs can accept composite video signals, and video input 1 can be used for S-video; with the optional Control Package, all four video inputs can be used for S-video.

A PXC200 frame grabber automatically synchronizes to the selected video source. For very high-speed switching between camera inputs, the optional Control Package provides DC voltage restoration for all video inputs and horizontal and vertical sync output signals for genlocking cameras. Even for non-genlocked video sources, the Control Package cuts the time for synchronizing to a new source from about 2.5 seconds to less than 0.5 seconds.

Use NTSC, PAL, or SECAM Video Formats. PXC200 frame grabbers support the 60 Hz North American NTSC color and RS-170 monochrome formats, and 50 Hz European PAL and SECAM color and monochrome formats.

Video Capture Modes and Resolution

When you capture images with a PXC200 frame grabber, you can specify how you want to start the capture process, and whether you want to work with all or with just a subset of the total image data.

Image Capture Modes

There are three ways to capture images with a PXC200 frame grabber:

Software-initiated grab. On a command from an application program, the board grabs a single frame or field.

Triggered grab. The board waits for an external trigger and then grabs the frame.

Continuous acquire. In this mode, the board grabs one image after another. Continuous acquire is useful for applications that need to watch for changes between successive images, and for sending video data directly to other PCI devices.

With any of these modes, you can start the capture at the next field in the incoming video signal, or you can specify that the capture will start with field 0 or field 1.

Capture Resolution

PXC200 frame grabbers use a crystal-controlled pixel clock to sample horizontal lines of video at 14.32 MHz for NTSC or 17.73 MHz for PAL/SECAM. At these frequencies the frame grabber acquires more pixels per line than are required for the standard video formats and then uses interpolation to reduce the number of pixels to the specified value. On a typical display monitor with a 4 x 3 aspect ratio, a 640-pixel horizontal resolution results in approximately square pixels for images in NTSC video mode; a 768-pixel horizontal resolution results in square pixels for images in PAL and SECAM video modes; and a 720-pixel horizontal resolution supports the rectangular video pixels of conventional video displays.

If you don't need to work with all of the image data, you can further scale the image horizontally and vertically. You can also crop the image horizontally and vertically, retaining just a rectangular subset of the image. By transferring only a subset of the image, you save memory and bandwidth on the bus, leaving more of both resources available to other parts of your application and to other applications.

Common color formats are supported for output, including YCrCb, RGB, and Y8 (8-bit grayscale).

Real-Time Image Data Transfer

The PCI bus master design of the PXC200 frame grabber lets you achieve real-time performance for captures to main memory or directly to the display.

PCI Bus Master Design

The bus master design of the PXC200 frame grabber lets the frame grabber directly control the transfer of image data to main memory or to

another PCI device, such as a display controller. While the frame grabber is transferring data, the main CPU is free to run other parts of your application or other applications.

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 full-size, 8-bit-per-pixel video image data. At 16 or 24 bits per pixel, you might not be able to achieve real-time display of full-size images, depending on the design of the system.

Selectable Destination for Image Captures

You can choose the destination for the image capture data:

A buffer in main memory. The data is transferred via direct memory access (DMA) to a buffer in the computer's main memory. The transfer is fast, and the data is available in memory for further processing.

Another memory-mapped device. The data is transferred via DMA directly to another PCI device. For example, some PCI VGA cards support such transfers, which can be used to display live video.

I/O Features

Trigger Input

PXC200 frame grabbers have an external trigger input that can be used to trigger an image capture. A simple push button switch attached to this input can be used like a camera shutter button. The trigger input can be programmed to respond to either low or high logic levels, or to rising or falling edges.

Optional I/O

The optional Control Package expands the I/O capabilities of the PXC200 with the following:

Digital I/O—Four general-purpose input lines, software programmable as separate triggers, plus four general-purpose output lines, which can be used as triggered or software-programmable strobes. These eight I/O lines replace the single trigger input on the standard PXC200.

Sync Signals—Vertical and horizontal sync outputs, which can be used to genlock a video source. The PXC200 can resynchronize much faster when switching between genlocked video sources.

Strobe Inhibit—You can specify a holdoff period for firing strobes to prevent the strobes from firing during a camera's inter-line transfer. Inhibiting the strobes during the CCD transfer time gives you more reliable image captures.

S-Video—All four video inputs can accept S-video or composite video; on the standard PXC200, only one of the video inputs can accept S-video.

Programming Libraries and DLLs

For custom applications, the PXC200 software includes support for writing your own frame grabber programs. The library and DLL functions take care of the details of low-level hardware control for you, letting you concentrate on getting your application working.

C Libraries for DOS—Write 16-bit DOS programs using the 16-bit library with Borland, Microsoft, or Watcom C compilers, or write 32-bit DOS programs using the Watcom DOS/4GW library.

DLLs for Windows—Write programs for Windows 3.1, Windows 95, and Windows NT with C compilers from Borland and Microsoft, or with Visual Basic. The PXC200 DLLs are standard Windows DLLs, and you should be able to use them with most Windows development tools that can make calls to Windows DLLs.

DOS VGA Video Display Library—Use the Video Display library to create a menu-based user interface for your 16-bit DOS and 32-bit DOS/4GW applications that allows you to simultaneously display graphics and text.

Sample source code—Sample source code is provided, for both DOS and Windows, to show you how to use various features of the libraries and DLLs.

Chapter 4, *Programming the PXC200*, on page 33, describes the main features of the PXC200 hardware and software and how to use them to build applications. For reference information on all PXC200 library functions, see Chapter 5, *PXC200 Library Reference*, on page 81, and Chapter 6, *Frame Library Reference*, on page 121. The DOS VGA Video Display library and its functions are described in Chapter 7, *The VGA Video Display Library*, on page 139.

The PXCVU Program

The PXC200 software includes a DOS frame grabber application called PXCVU. Using PXCVU, you can capture images, save images to disk, and adjust many of the image capture features of a PXC200 frame grabber—all without writing a single line of code. For more information, see Chapter 3, *The PXCVU Application*, on page 29.

Utility Programs

The PXC200 software also includes several utility programs.

PXCREV

If you need to contact Imagenation Technical Support, you'll be asked for your board's revision number. PXCREV is a DOS program that displays the revision number for any frame grabbers it finds in your system. You must run this program from DOS, not from a DOS window in Windows.

VGACOPY

VGACOPY is a test program that lets you evaluate the performance of your computer for grabbing images and copying the data to the VGA display in DOS. For similar tests in Windows, see the Windows sample programs PXCDRAW1 and PXCDRAW2.

Next Steps...

For...	See...
Installing your PXC200 frame grabber	Chapter 2, *Installing Your Frame Grabber*, on page 11
Operating your PXC200 with the PXCVU program	Chapter 3, *The PXCVU Application*, on page 29
Writing your own frame grabber applications	Chapter 4, *Programming the PXC200*, on page 33
Connector and cabling specifications	Appendix A, *Cables and Connectors*, on page 157

Installing Your Frame Grabber

2

Do You Need a Cable?

Standard PCI and CompactPCI Cables

The BNC composite video connector and the S-video connector on the standard PCI and CompactPCI configurations of the PXC200 board let you attach up to two video sources. Additional video sources (you can connect a total of four), a trigger input, and a +12V power source are also available by using the 26-pin D connector. If you have the optional Control Package, the 26-pin D connector also gives you access to the additional digital I/O lines and sync signals. To use the 26-pin connector, you'll need a cable with the correct mating connector and pinouts. For information on making cables, see Appendix A, *Cables and Connectors*, on page 157.

PC/104-Plus Cables

You'll need cables to attach to the connectors on frame grabbers with the PC/104-Plus configuration. For information on making cables, see Appendix A, *Cables and Connectors*, on page 157.

Installing Your Board

Follow the instructions below to install your board:

1 Turn off and unplug your computer, then remove its cover.

> **Caution**
> *Static electricity can damage the electronic components on the PXC200 board. Before you remove the board from its antistatic pouch, ground yourself by touching the computer's metal back panel.*

2 Install the PXC200 board as follows:

For a standard PCI-bus board:

a Locate an unused PCI expansion slot that is enabled for bus mastering. On some systems, you must enable a PCI slot for bus mastering by using a switch or jumper on the system board, or by changing the BIOS settings. Refer to the manual that came with your computer for more information.

b Remove the cover plate. Save the screw.

c Insert the PXC200 board into the slot and seat it firmly.

d Secure the board's cover plate using the screw you saved.

e If you want to use the +12V output on the 26-pin connector, attach a power connector from your PC's power supply cable to the J4 connector on the board. The J4 connector accepts the same type of power supply connector used for floppy disk drives. If you don't want to use the +12V output, you can skip this step.

For a CompactPCI board:

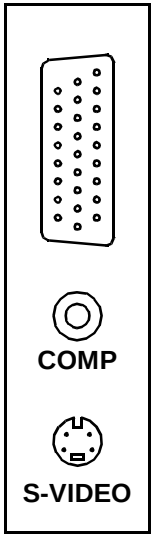
a Locate an unused slot in your chassis that supports DMA bus mastering. The first two or three slots next to the CPU slot typically support bus mastering.

b Insert the PXC200 board into the slot, seat it firmly, and tighten the holding screw.

For a PC/104-Plus board:

a Set the four-position rotary switch on the PXC200 board to an unused number. Each PC/104-Plus plug-in module must be set to a unique number. On some PC/104-Plus systems, the board closest to the CPU must be set to zero, the next board must be set to one, and so on.

b Insert the PXC200 board into the connector and seat it firmly.



3 Following the instructions below, connect your board to the video input and, optionally, to other I/O:

For a standard PCI-bus or CompactPCI-bus board:

BNC and S-video connectors. Connect your video source to the S-video connector or to the composite video BNC connector (see diagram at left). The composite connector is video input 0, and the S-video connector is video input 1.

26-pin D connector. If you're using the 26-pin D connector, connect your cable to that connector. If you need to purchase or make a cable, see Appendix A, *Cables and Connectors*, on page 157.

For a PC/104-Plus board:

Attach your cable to the connector on the PXC200 board. For information on making cables, see Appendix A, *Cables and Connectors*, on page 157.

4 Replace the cover on the computer, plug it in, and turn on the power.

5 **This step applies to Windows 95 only.** When you restart your system, you might see the message "Found new multimedia PCI device," and the *Add New Hardware Wizard* is displayed. If this happens, follow the steps below:

a Insert the **Windows 95** PXC200 software installation disk in the drive.

b In the wizard, click the Have Disk button.

c In the *Install from Disk* dialog, specify the drive letter for the floppy disk drive and click OK.

You should see a single option, *PX Precision Frame Grabber*, listed in the wizard.

d Select *PX Precision Frame Grabber* and click Next.

e Click Next again to let Plug and Play complete the installation.

You should see a message that Windows hasn't finished installing the necessary software. You'll install the software in the next section.

f Click Finish.

6 That completes the hardware installation. Next, you'll install the PXC200 software.

Installing the Software

PXC200 frame grabbers can be used with DOS, DOS/4GW, Windows 3.1, Windows 95, and Windows NT. Refer to the appropriate section below for the operating system you are running.

DOS, DOS/4GW, and Windows 3.1 Software Installation

1 **This step applies only to DOS**; if you're not using DOS, skip to the next step. The frame grabber needs a vacant 4 KB block of system memory in segment 0xD000 or in segment 0xE000. The 4 KB block of memory must be aligned on a 4 KB boundary; that is, it must be of the form 0xD**?**00-0xD**?**FF or 0xE**?**00-0xE**?**FF, where **?** is the same hexadecimal digit in both the beginning and ending numbers of the range. For example, 0xD**2**00-0xD**2**FF or 0xE**A**00-0xE**A**FF.

   To make a memory block available for the frame grabber:

   a Make sure the block is not used by any other hardware devices. You can use the Microsoft diagnostics program MSD to display memory usage. (MSD comes with DOS and Windows.)

   b Modify the entry in CONFIG.SYS for your memory manager to prevent it from using the block. For example, if you are using EMM386, and you want to use 0xE000-0xE0FF for the frame grabber, add **`x=e000-e0ff`** to the end of the EMM386.EXE entry in your CONFIG.SYS:

```
device=c:\dos\emm386.exe noems x=e000-e0ff
```

   If you're using another memory manager, like QEMM or 386MAX, consult your manual.

2 Insert the **DOS/Windows 3.1** installation diskette in the floppy drive.

3 The diskette includes two installation programs, one for DOS and another for Windows. The DOS *INSTALL.EXE* program installs **only** the DOS and DOS/4GW software, not the Windows software; the Windows *SETUP.EXE* program installs all three. Decide which installation program you want to use, and follow the appropriate instructions below:

DOS and DOS/4GW only

a At the DOS prompt, type (substitute the appropriate drive letter for "a") **`a:\install`** and press Enter.

b When the INSTALL program has completed, reboot your computer.

c After rebooting your system, you can use the PXCVU program to verify that your frame grabber is correctly installed. For instructions on running PXCVU, see Chapter 3, *The PXCVU Application*, on page 29. If an error message appears when you try to start PXCVU, see *Troubleshooting*, on page 22.

Windows, DOS, and DOS/4GW

a From the Program Manager in Windows, choose the File menu and select Run.

b In the Command Line box, type **`a:\setup`**, and click OK.

c When the SETUP program has completed, restart Windows.

Setup creates a new program group called *PXC*.

d After restarting Windows, you can run one of the PXCDRAW sample programs to verify that your frame grabber is correctly installed. The sample programs are in the c:\pxc2\samples\win16

directory. If you have problems running the sample programs, see *Troubleshooting*, on page 22.

Changes to System Files for DOS, DOS/4GW, and Windows 3.1

The installation programs will, at your option, modify your AUTOEXEC.BAT and SYSTEM.INI (SETUP only) files. The changes are listed below so that you can make your own modifications, if you prefer. The installation programs do not look for their own modifications; if you run the installation programs more than once, don't let them modify your system files unless you have removed the previous modifications.

AUTOEXEC.BAT Changes for DOS, DOS/4GW, and Windows 3.1

```
REM Imagenation's Modifications
set path=c:\pxc2\bin;%path%
set imagenation=c:\pxc2
REM Imagenation's Modifications End
```

Adding c:\pxc2\bin to your PATH makes the samples and utilities easier to execute. The IMAGENATION environment variable specifies the location of files required by the PXCVU application. PXCVU won't run unless this variable is correctly defined.

After your AUTOEXEC.BAT file is modified, you must reboot your computer for the changes to take effect.

SYSTEM.INI Changes for Windows 3.1

```
[386Enh]
; Imagenation's Modifications
device=c:\pxc2\bin\pxc2.vxd
; Imagenation's Modifications End
```

The PXC200 Windows Virtual Device Driver (VxD), *PXC2.VXD*, is added to the [386Enh] section. The VxD will be loaded only when you

start Windows. The PXC200 DLL, *PXC2_16.DLL*, requires this VxD; the DLL will not run unless the VxD is installed. After running Setup, you must restart Windows to load the VxD.

Windows 95 Software Installation

1. If you previously installed the Windows 3.1 software, you must edit the [386Enh] section of the SYSTEM.INI file to remove the lines that load the VxD, *PXC2.VXD*. Otherwise, when you run Windows 95, the system will try to load the VxD twice. For more information, see *SYSTEM.INI Changes for Windows 3.1*, on page 17.

2. Put the **Windows 95** installation disk in the floppy drive.

3. Click the Start button and click Run.

4. For the name of the program, type **`a:\setup`** and click OK.

5. Follow the instructions in the Install wizard to complete the installation.

   Setup creates a new program group called *PXC*.

   When you have competed installing the software, you must reboot Windows 95 before the drivers that you have installed will be accessible.

6. Click the Start button and click Shut Down.

7. In the Shut Down Windows dialog, click **Restart the computer** and click **Yes** to restart Windows 95.

   After restarting Windows, you can run one of the PXCDRAW sample programs to verify that your frame grabber is correctly installed. The

sample programs are in the c:\pxc2\bin directory. If you have problems running the sample programs, see *Troubleshooting*, on page 22.

Windows 95 Registry Changes

If you let the Setup program create a registry entry for the PXC200 driver, and you later need to uninstall the driver, you must edit the Windows 95 Registry by using the REGEDIT.EXE program in your Windows 95 directory.

The installation program adds the following key to the Windows Registry:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet
                              \Services\VxD\PXC2
```

The value assigned to this key is:

StaticVxD. A string key that contains the complete path of the VxD file, such as *c:\pxc2\bin\pxc2.vxd*.

When you remove the registry entry, Windows will no longer automatically load the VxD. When you run a frame grabber program, the DLL will attempt to locate the VxD and load it dynamically. If your programs have problems locating the VxD, copy the VxD to the Windows SYSTEM directory.

Although loading the VxD dynamically adds about 1/2 second to program startup time, it ensures that the VxD gets unloaded when the program terminates, de-allocating all frame grabbers. This can be particularly useful during program development when programs might crash and leave a frame grabber allocated.

Windows NT Software Installation

1. To install the driver, you must be logged in as Administrator or have equivalent access.

2. Put the **Windows NT** installation disk in the floppy drive.

3. Click the Start button and click Run.

4. For the name of the program, type **`a:\setup`** and click OK.

5. Follow the instructions in the Install wizard to complete the installation.

   The drivers can be configured to start automatically at boot time or to be manually controlled. The installation program will default to loading the driver automatically at boot time. If you choose to manually load and unload the driver, the driver operation can be controlled through the Devices icon in the Windows NT Control Panel. You can also manually start the driver by entering the following command at the command prompt:

   ```
   net start pxc200
   ```

   To stop the PXC200 driver, enter this command at the command prompt:

   ```
   net stop pxc200
   ```

   Setup creates a new program group called *PXC*.

   When you have competed installing the software, you must reboot Windows NT before the registry entries you have made will take effect.

6 Click the Start button and click Shut Down.

7 In the Shut Down Windows dialog, click **Restart the computer** and click **Yes** to restart Windows NT.

After restarting Windows, you can run one of the PXCDRAW sample programs to verify that your frame grabber is correctly installed. The sample programs are in the c:\pxc2\bin directory. If you chose to control the driver manually, you must start the driver before running the sample programs. If you have problems running the sample programs, see *Troubleshooting*, on page 22.

Windows NT Registry Changes

If you let the Setup program create a registry entry for the PXC200 driver, and you later need to uninstall the driver, you must edit the Windows NT Registry by using the REGEDIT.EXE program in your Windows NT directory.

The installation program adds the following keys to the Windows Registry:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet
                                \Services\PXC200
HKEY_LOCAL_MACHINE\System\CurrentControlSet
                \Services\EventLog\System\PXC200
```

When you remove these registry entries, Windows NT will no longer automatically load the driver, and you will be unable to use the PXC200 or any of its programs until you re-install the driver.

The number of PXC200 boards in a system is limited to four. If you need to have more than four boards in a system, contact Imagenation Technical Support for assistance (see *Technical Support*, on page 26).

PXC200 Software Directories

The installation programs create the LIB, BIN, and INCLUDE directories, and directories for the sample source code:

Directory	Contents
c:\pxc2\lib	DOS and Windows libraries.
c:\pxc2\bin	Executable sample programs, DLLs, and drivers.
c:\pxc2\include	Header files.
c:\pxc2\samples\dos	DOS and Watcom DOS/4GW sample source code.
c:\pxc2\samples\win16	Windows 16-bit sample source code.
c:\pxc2\samples\win32	Windows 32-bit sample source code.

These directories are structured to make program execution, compiling, and linking convenient.

You can run the Windows sample programs to control the frame grabber, write BMP files, and run the timing tests (don't forget to first restart Windows to load the driver). The sample programs are PXCDRAW1 and PXCDRAW2.

Troubleshooting

This section contains troubleshooting information for the following:

- Error loading DLLs
- Error loading VxDs
- Running PXCVU or PXCREV
- Slow video display performance
- Windows hangs or crashes on reboot
- Windows NT-specific problems

Error Loading DLL

The system can't locate the PXC200 DLL. Either edit your PATH environment variable to include the path to the PXC200 DLL (see *PXC200 Software Directories*, on page 22) or move the DLL to the \WINDOWS\SYSTEM directory for Windows 3.1 and Windows 95, or to the \WINDOWS\SYSTEM32 directory for Windows NT.

Error Loading VxD

When booting Windows 3.1 or Windows 95, you might see the error "PXC2.VXD Requires a PCI compatible BIOS." This means your BIOS lacks the BIOS32 Service Directory feature of the PCI BIOS Specification, Revision 2.0.

First, make sure you are using the version of the PXC2.VXD that came with your PXC200. If you're using an older version, upgrade to the latest version. If you still get this error message with the latest version of PXC2.VXD, you'll need to upgrade your BIOS; contact the manufacturer of your system for an upgrade.

Problems Running PXCVU or PXCREV

PXCVU and PXREV are DOS programs. You can't run these programs in a DOS window in Windows. If your system hangs when you run PXCVU or PXREV, this is the most likely cause.

If the program hangs when you start it, you might have an IRQ conflict or a compatibility problem with the PCI chip set in your PC. Check for possible IRQ conflicts first. For the latest compatibility information, contact Imagenation Technical Support (see *Technical Support*, on page 26).

Make sure that you are excluding a 4 KB block of upper memory in your CONFIG.SYS file (see **Step 1** on page 15 of the installation instructions).

If you see the message *This graphics card is not VESA compatible* when you run PXCVU, you aren't using a VESA-compatible display driver. Check the documentation for your display controller board to see if a VESA-compatible driver is available.

In PXCVU, if you see broken lines in the video (like *snow* in a TV picture) the PCI bus is being overloaded or errors are occurring. Most Intel 486-based systems don't have a PCI bus that is fast enough for the PXC200 frame grabber. Run the VGACOPY program to check for errors on the PCI bus.

If you haven't set the IMAGENATION environment variable, PXCVU will display an error and won't run. For information on the IMAGENATION environment variable, see *AUTOEXEC.BAT Changes for DOS, DOS/4GW, and Windows 3.1*, on page 17.

PXCVU will fail to run if the file DOS4GW.EXE is not accessible through your PATH environment variable.

If you see the message *AllocateVGA failed. Your video card may not support this mode.*, PXCVU might be using a video mode that your video card doesn't support under DOS. This is usually caused by the pixel-depth setting. Try changing the setting, as follows:

a In a text editor, open the file c:\pxc2\pxcvu.ini.

b Locate the following line:

```
BitsPerPixel=16
```

c Change the number of bits per pixel from 16 to 8 or 24.

d Try running PXCVU again.

Slow Video Display Performance

When you're displaying video on the screen, the amount of memory on the VGA display controller card can affect the performance. With some display controllers, adding memory to the display controller will improve the performance.

Windows Hangs or Crashes on Boot

This can be caused by an interrupt conflict. Check to make sure you have an IRQ available and that no ISA device is trying to use the same IRQ that any PCI device is trying to use.

Windows NT-Specific Problems

If you have trouble under Windows NT, it is most likely related to the PXC200 device driver not loading. The following items are some common problems that can cause the driver not to load:

- Make sure you are logged in as Administrator, or have Administrator-level access when you install the driver. If you installed the driver without having this level of access, you must log in as Administrator and re-install the driver.

- If during the installation you selected to manually load the driver, you must start the driver every time you reboot Windows NT. To start the driver, in the Windows NT Control Panel, click the Devices icon; then click the PXC200 Driver; and then click the Start button.

  Alternatively, you can start the driver from the command prompt by executing this command:

  ```
  net start pxc200
  ```

To stop the PXC200 driver, enter this command from the command prompt:

```
net stop pxc200
```

- Check that the PXC200 driver is present and has been started. You can do this by clicking the Devices icon in the Control panel. If the driver is not present, check the Event Viewer to see if there was an error when the driver attempted to load. There might be a conflict with another device that is reported in the Event Viewer.

If Windows NT refuses to boot after the PXC200 drivers have been installed, use the following method to recover:

1. When Windows NT reboots, you'll see the message “Press spacebar NOW to invoke Hardware Profile/Last Known Good menu.”

2. Press the spacebar and pick the most recent configuration. This should reverse the change to the Windows NT Registry, allowing you to boot Windows NT and troubleshoot further.

Technical Support

Imagenation offers free technical support to customers. If the PXC200 board appears to be malfunctioning, or you’re having problems getting the library functions to work, please read the appropriate sections in this manual. If you still have questions, contact us, and we’ll be happy to help you.

When you contact us, please make sure that you have the following information available:

- The revision number of your board. You can get this number by using the PXCREV program in DOS or either of the PXCDRAW programs

in Windows. You must run the PXCREV program from DOS, not from a DOS window in Windows.

- The operating system you're running: DOS, DOS/4GW, Windows 3.1, Windows 95, or Windows NT.
- The compiler you're using, including the name of the manufacturer and the version number (for example, Borland C version 5.0).

Voice: 503-641-7408 **Toll free:** 800-366-9131

Fax: 503-643-2458 **CompuServe:** 75211,2640

Internet: support@Imagenation.com
www.imagenation.com

The Imagenation World Wide Web site (www.imagenation.com) always has the latest versions of the Imagenation software. Check anytime for software updates.

The PXCVU Application

3

This chapter describes the PXCVU application program for DOS. PXCVU is a basic frame grabber application that lets you control the features of your PXC200 frame grabber without writing your own application program. You can use PXCVU to capture frames or fields, write frames to disk files, change the video source, and to set the brightness, contrast, hue, and saturation.

Setting Up PXCVU

To run PXCVU, you must have the IMAGENATION environment variable set to point to the directory containing PXCVU.HLP and PXCVU.INI. PXCVU.HLP contains the text of the help screens you can access from PXCVU. PXCVU.INI is an optional file that contains initialization values for the application.

If you let the DOS Install or Windows Setup programs copy the files from the diskette and make the required changes to your system files, you're ready to run PXCVU. If not, see *AUTOEXEC.BAT Changes for DOS, DOS/4GW, and Windows 3.1*, on page 17, for the required settings.

Starting PXCVU

Make sure you have a video source connected to your PXC200 board before starting the PXCVU program.

To run PXCVU, execute the following at the DOS command line (do **not** run PXCVU in a DOS window in Windows):

```
c:\pxc2\bin\pxcvu
```

If you see a display like that shown on page 31, the PXCVU program has started correctly. Otherwise, see *Troubleshooting*, on page 22.

Running PXCVU with More Than One Frame Grabber

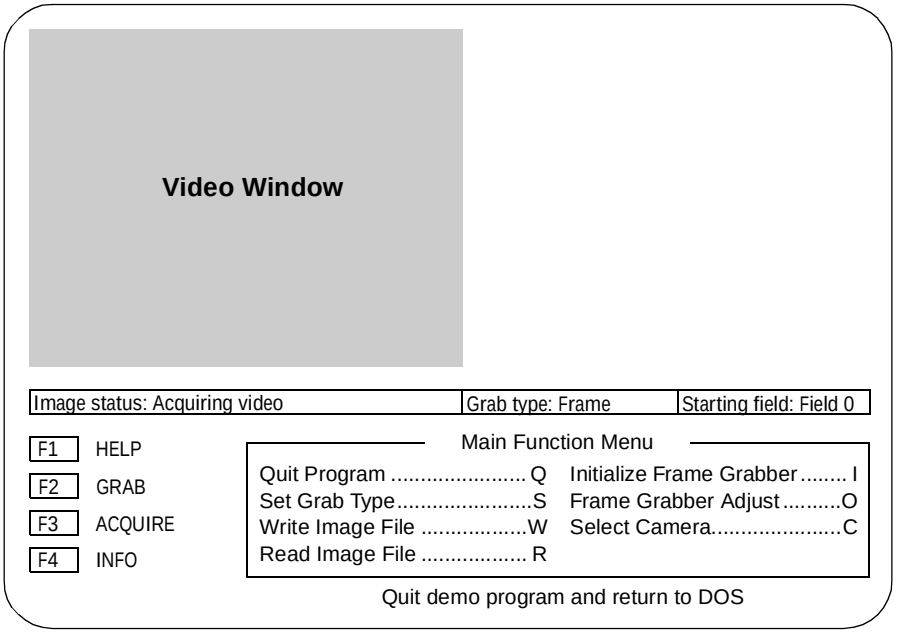
If you have more than one frame grabber installed in your system, PXCVU will use the first frame grabber that it finds. To specify a particular frame grabber, follow the command with the number of the frame grabber:

```
c:\pxc2\bin\pxcvu n
```

Frame grabbers are numbered sequentially starting with $n = 0$. Due to the nature of the PCI bus, the number of the frame grabber won't necessarily correspond to the PCI bus slot in which the frame grabber is installed. To determine the correct number, *n*, of each frame grabber, you'll just have to try the PXCVU application with different values for *n* and observe the video displayed to identify the source.

Using PXCVU

The screen for the PXCVU application looks similar to the picture below:



If you have an active video source when you start PXCVU, the video should appear in the **Video Window** as soon as you start the program.

The **Status Line** below the video window shows you the current selections for the image displayed in the Video Window, the type of grab, and the starting field.

Definitions for functions keys are shown in the lower left corner:

- **F1 HELP—**Press F1 to get help on the currently-selected menu item.
- **F2 GRAB—**Press F2 to grab a frame using the current grab mode.

- **F3 ACQUIRE—**Press F3 to turn continuous acquire mode on or off.

- **F4 INFO—**Press F4 to display the hardware revision number and serial number for the board, the image size, and the screen size.

The **Main Function Menu** gives you more detailed control of the board. A short explanation of the currently-highlighted menu item is shown at the bottom of the screen. For help on a menu item, move the highlight to the item using the arrow keys, and press F1 for Help. The features listed in the menu are also explained in more detail in Chapter 4, *Programming the PXC200*, on page 33.

Programming the PXC200

4

This chapter describes how to write your own software programs for the PXC200 using the functions provided in the PXC200 software libraries. The chapter begins with an overview of how the libraries are organized, followed by information about programming for specific operating systems, and about using specific programming languages. The remainder of the chapter describes how to use the functions in the libraries to perform the basic steps required to capture images and access the image data, plus optional features you can use.

Library Organization

The PXC200 software is implemented as a set of libraries:

PXC200 Frame Grabber Library—Includes the functions you'll use to control the frame grabber, including capturing images, setting image resolution, switching video inputs, and setting image contrast, brightness, hue, and saturation. Chapter 5, *PXC200 Library Reference*, on page 81, describes the syntax and other details for each function.

Frame Library—Includes the functions you'll use to access captured image data and to read and write image files. Chapter 6, *Frame Library Reference*, on page 121, describes the syntax and other details for each function.

DOS VGA Video Display Library—A DOS-only library that includes functions for controlling the VGA display, creating a menu-style user interface, and drawing basic graphic primitives. This library is not included in the current chapter, but is described in Chapter 7, *The VGA Video Display Library*, on page 139.

Operating System Specifics

Follow the guidelines in this section for compiling, linking, and running PXC200 programs.

You can put c:\pxc2\lib and c:\pxc2\include in your environment variables for Microsoft, or in your TURBOC.CFG file for Borland, or in your integrated development environment (IDE) search list.

DOS Programming

The following table summarizes operating system specifics for compiling, linking, and running C programs under DOS. The DOS 16-bit library supports the large memory model only.

DOS 16-bit Programs		
Header Files	**Libraries**	**Runtime, Memory, and Installation Requirements**
PXC200.H FRAME.H VIDEO.H*	Borland: PXC2_LB.LIB FRAME_LB.LIB VIDEO_LB.LIB* Microsoft Ver. 6: PXC2_L6.LIB FRAME_L6.LIB VIDEO_L6.LIB* Microsoft Ver. 7+: PXC2_LM.LIB FRAME_LM.LIB VIDEO_LM.LIB*	For required changes to AUTOEXEC.BAT, see *Changes to System Files for DOS, DOS/4GW, and Windows 3.1*, on page 17.

* The VIDEO files are described in Chapter 7, *The VGA Video Display Library*, on page 139.

Watcom DOS and DOS/4GW Programs		
Header Files	**Libraries**	**Runtime, Memory, and Installation Requirements**
PXC200.H FRAME.H VIDEO.H*	16-bit: PXC2_LW.LIB FRAME_LW.LIB VIDEO_LW.LIB* 32-bit: PXC2_FW.LIB FRAME_FW.LIB VIDEO_FW.LIB*	For required changes to system files, see *Changes to System Files for DOS, DOS/4GW, and Windows 3.1*, on page 17.

* The VIDEO files are described in Chapter 7, *The VGA Video Display Library*, on page 139.

Windows 3.1 Programming

The following table summarizes operating system specifics for compiling, linking, and running C programs under Windows 3.1:

Header Files	Libraries†	Runtime, Memory, and Installation Requirements
PXC200.H FRAME.H VIDEO_16.H*	Borland: ILIB_LB.LIB ILIB_MB.LIB ILIB_SB.LIB VIDEO_16.LIB* Microsoft: ILIB_LM.LIB ILIB_MM.LIB ILIB_SM.LIB VIDEO_16.LIB*	PXC2.VXD, PXC2_16.DLL, FRAME_16.DLL, and VIDEO_16.DLL needed for runtime. For VxD installation, see *DOS, DOS/4GW, and Windows 3.1 Software Installation*, on page 15.

† The three libraries for each compiler are for different memory models: large (ILIB_L), medium (ILIB_M), and small (ILIB_S).
* The VIDEO files are described in *Using the Video Display DLL*, on page 78.

Windows 95 Programming

The following tables summarize operating system specifics for compiling, linking, and running C programs under Windows 95:

Windows 95 16-bit programs		
Header Files	**Libraries†**	**Runtime, Memory, and Installation Requirements**
PXC200.H FRAME.H VIDEO_16.H*	Borland: ILIB_LB.LIB ILIB_MB.LIB ILIB_SB.LIB VIDEO_16.LIB* Microsoft: ILIB_LM.LIB ILIB_MM.LIB ILIB_SM.LIB VIDEO_16.LIB*	PXC2.VXD, PXC2_16.DLL, FRAME_16.DLL, and VIDEO_16.DLL needed for runtime. For VxD installation, see *Windows 95 Registry Changes*, on page 19.

† The three libraries for each compiler are for different memory models: large (ILIB_L), medium (ILIB_M), and small (ILIB_S).
* The VIDEO files are described in *Using the Video Display DLL*, on page 78.

Windows 95 32-bit programs		
Header Files	**Libraries**	**Runtime, Memory, and Installation Requirements**
PXC200.H FRAME.H VIDEO_32.H*	Borland: ILIB_32B.LIB VIDEO32B.LIB* Microsoft: ILIB_32.LIB VIDEO_32.LIB*	PXC2.VXD, PXC2_95.DLL, FRAME_32.DLL, and VIDEO_32.DLL needed for run-time. For VxD installation, see *Windows 95 Registry Changes*, on page 19.

* The VIDEO files are described in *Using the Video Display DLL*, on page 78.

Any DLLs your application uses should be in the Windows SYSTEM directory or in your path.

Windows NT Programming

The following table summarizes operating system specifics for compiling, linking, and running C programs under Windows NT:

Windows NT 32-bit programs		
Header Files	**Libraries**	**Runtime, Memory, and Installation Requirements**
PXC200.H FRAME.H VIDEO_32.H*	Borland: ILIB_32B.LIB VIDEO32B.LIB* Microsoft: ILIB_32.LIB VIDEO_32.LIB*	PXC200.SYS, PXC2_NT.DLL, FRAME_32.DLL, and VIDEO_32.DLL needed for runtime. For driver installation, see *Windows NT Registry Changes*, on page 21.

* The VIDEO files are described in *Using the Video Display DLL*, on page 78.

Any DLLs your application uses should be in the Windows SYSTEM32 directory or in your path.

Programming in a Multithreaded, Multitasking Environment

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 (!pxc.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:

```
pxc.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. **WaitVB()** can be used to add delays to polling loops to improve system performance.

Scheduling multiple threads to handle complicated image processing tasks might make programming significantly easier, and the PXC200 library does allow multithreading with one important exception. A program should **not** allow two different threads of execution to access the same frame grabber at the same time. Doing so could put the frame grabber into an unpredictable state, and possibly cause DMA transfers to be misdirected. This limitation can't be fixed by simply wrapping each frame grabber control function in a mutual exclusion object, since many functions permanently change the state of the frame grabber. In general, you should make sure that only one thread is responsible for each frame grabber. Functions that do not directly access the frame grabber, such as the file I/O functions and the buffer manipulating functions, are safe to multithread as long as the usual care is taken to be sure that the data they access does not become invalid.

Programming Language Specifics

This section discusses specific information about writing programs in C and in Visual Basic.

Programming in C

If you're using third-party libraries or multiple frame grabber libraries in developing your programs, the same function name might exist in more than one library, causing a symbol collision. The PXC200 software libraries are designed to help you avoid symbol conflicts.

When you initialize a library, you can specify a unique library name that you'll use for calling all functions in that library. When you make function calls to that library, you call a function as a member of a structure. The name of the structure is the library name you used to initialize the library. The following example shows how you might initialize the PXC200 frame grabber library using the library name *pxc* and then call the AllocateFG() function, which is used to get a handle for a frame grabber:

```
imagenation_OpenLibrary("pxc2_95.dll", &pxc,
                                          sizeof(pxc));
handle = pxc.AllocateFG(-1);
```

The first line initializes the frame grabber library. The second parameter, *pxc*, is the library name you have chosen. The second line calls the AllocateFG() function as a member of a structure called *pxc*.

The same technique works with the Frame library and the DOS VGA Video Display library. Just be sure to choose unique library names for each library you initialize.

Visual Basic Programming

The Windows DLLs were designed to make the function calls as uniform as possible, whether you're programming in C or in Visual Basic. Since the syntax and keywords in Visual Basic differ from those of C, before you start programming in Visual Basic, you should look at the Visual Basic function definitions in the .BAS file.

There are a few things you should keep in mind when using Visual Basic with the DLL functions:

Accessing frame data—In C, you can use the pointer returned by FrameBuffer() to access the image data in the frame. Visual Basic doesn't use pointers, so you must use the functions GetPixel(), GetColumn(), GetRectangle(), and GetRow() to access the data in a frame. The FrameBuffer() function exists in Visual Basic for situations where you need to get a pointer to pass to other Windows API functions that are designed to work with pointers.

.BAS File—You must include the appropriate .BAS file in all projects you build using the PXC200 DLL functions. The .BAS file includes all the declarations you'll need to work with the DLLs. For 16-bit or 32-bit programs, include PXC2_V4.BAS and FRAME_V4.BAS; for 32-bit programs only, include VIDEO_32.BAS.

Buffers in Visual Basic 4.0

Visual Basic 4.0 includes a **`Byte`** type, which is equivalent to the **`unsigned char`** type that the DLLs expect for buffers. Thus, the VIDEO_32.BAS file uses **`buf As Byte`** in the function definitions. To pass a buffer to the DLL, just pass the first element of your declared **`Byte`** array.

Using the Visual Basic Development Environment

Caution

*Do not use the **End** button in the Visual Basic development environment to terminate your application. The **End** button terminates a program immediately, without executing the Form_Unload function or any other functions. If you use the **End** button to exit a program, you might need to restart Windows to free any frame grabbers that your program allocated.*

Displaying Video in Visual Basic Applications

The PXC200 software includes a Video Display DLL that makes displaying captured images in a window quite simple. For more information, see *Using the Video Display DLL*, on page 78.

Typical Program Flow

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

1. Initialize the libraries.
2. Request access to the frame grabber.
3. Set up the destination for the captured image data.
4. Capture the image.
5. Release the frame grabber.
6. Exit the library.

In addition, a program might include:

- Selecting a video source, if you have more than one.
- Adjusting attributes of the image, such as hue and saturation.
- Specifying scaling and cropping for the image.
- 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.

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 PXC200 Frame Grabber library under Windows 95 (32-bit programs) are:

```
imagenation_OpenLibrary("pxc2_95.dll", &pxc,
                                              sizeof(pxc))
imagenation_CloseLibrary(&pxc)
```

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

```
imagenation_OpenLibrary("frame_32.dll", &frm,
                                              sizeof(frm))
imagenation_CloseLibrary(&frm)
```

Where *pxc* 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. For more information on this calling convention, see *Programming in C*, on page 41.

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 PXC200 Frame Grabber library and the Frame library for C programs under DOS are:

```
PXC200_OpenLibrary(&pxc, sizeof(pxc))
PXC200_CloseLibary(&pxc)

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

Where *pxc* and *frm* are the names you will use for the structures for calling library functions. For more information on this calling convention, see *Programming in C*, on page 41.

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.

Visual Basic and Windows Programs

The OpenLibrary() and CloseLibrary() functions for the PXC200 Frame Grabber library for Visual Basic programs under Windows 3.1 and Windows 95 are declared and called as:

```
declare function OpenLibrary lib “pxc2_95.dll” (ByVal
            pxc as Long, ByVal count as Long) as Integer
declare sub CloseLibrary lib “pxc2_95.dll” (ByVal
                                        pxc as Long)

OpenLibrary(0,0)
CloseLibrary(0)
```

For the Frame library, substitute “frame_32.dll” for “pxc2_95.dll” in the declarations and replace pxc.

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 PXC200 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. For more information, see, *DOS, DOS/4GW, and Windows 3.1 Software Installation*, **Step 1** on page 15.
- 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.

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.

FreeFG() frees the specified frame grabber, so it can be allocated by other processes.

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 PXCVU program (or your own program) to switch between frame grabbers.

When the AllocateFG() function fails, it is often because another process is using the frame grabber, or because a program terminated unexpectedly, leaving a frame grabber allocated. In the latter case, to free all frame grabbers, you might need to reboot your system.

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

You can create a frame for capturing images in two ways: with AllocateBuffer() or with AllocateAddress(). The Frame library (see Chapter 6, *Frame Library Reference*, on page 121) includes two additional functions for allocating frames for uses other than grabbing frames: AllocateFlatFrame(), and AllocateMemoryFrame().

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. AllocateAddress() is discussed in *Sending Images Directly to Another PCI Device*, below.

When you allocate storage for a frame you specify the type of pixel data that will be stored in the frame using one of the types listed below.

Pixel Data Type	Description
PBITS_Y8	8-bit grayscale.
PBITS_Y16*	16-bit grayscale.
PBITS_Yf*	Floating point grayscale.
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.
PBITS_RGBf*	A floating point number each for red, green, and blue.

* These types aren't supported by the PXC200 frame grabber and can't be allocated with AllocateBuffer(). However, they can be useful in image processing. For more information, see *Accessing Captured Image Data*, on page 76.

Pixel Data Type	Description
PBITS_YUV422	8 bits for Y and 8 bits for CrCb.
PBITS_YUV444*	8 bits each for Y, Cr, and Cb.
PBITS_YUV422P	YUV422 in planar format.
PBITS_YUV444P*	YUV444 in planar format.

* These types aren't supported by the PXC200 frame grabber and can't be allocated with AllocateBuffer(). However, they can be useful in image processing. For more information, see *Accessing Captured Image Data*, on page 76.

Captured video is digitized by the frame grabber in YCrCb 4:2:2 format and then converted to the specified pixel type before being transferred to the frame.

For most pixel data types, the buffer is organized as an array of pixels, where each pixel is represented by the data structure described above. (See the PXC200.H file for the actual structure declarations.) The YUV422P and YUV444P are both planar types. In these planar types, the data is organized in three planes: plane 0 for the Y component, plane 1 for the Cr component, and plane 2 for the Cb component.

When the AllocateBuffer() function fails, it means that you don't have enough memory allocated for frame buffers. Try freeing any frame buffers that you don't need. If calls to AllocateBuffer() still fail, try rebooting your system.

When you want to free memory previously allocated by AllocateBuffer() or AllocateAddress(), use the **FreeFrame()** function. Do not try to free a buffer when data is being transferred to it by queued functions or by GrabContinuous().

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.

Use **AllocateAddress()** to create a frame for a specified *physical address*, where the frame grabber will copy the image data. AllocateAddress() does not allocate any storage for an image buffer, since the data will be sent directly to the physical address.

> **Caution**
> *Use transfers to PCI devices only if you are familiar with DMA data transfers. DMA transfers bypass the operating system, so there is no opportunity to check for an incorrect address, and no protection faults are issued. An incorrect address could cause the operating system to crash. Since you are bypassing the window management routines of Windows, you can also corrupt the windows of other programs.*

AllocateAddress() doesn't allocate any storage for an image buffer, so the FreeFrame() function frees only the memory used by the frame structure.

Grabbing Images

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

Grab() digitizes video and copies the data to the specified frame. You can specify which video field the capture should start on, whether to digitize one field or both, and when to execute (see *Using Flags with Function Calls*, on page 66).

Grab() starts digitizing as soon as the command is processed by the frame grabber.

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

The frame grabber automatically changes to the correct pixel format for the destination frame whenever a Grab(), GrabContinuous(), or SwitchGrab() function is executed. Switching to a different pixel format takes about one field time. When the change occurs because of a Grab, this delay becomes part of the latency for the Grab. You can use the **SetPixelFormat()** function to preset the expected pixel format and minimize the latency in the Grab function.

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 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() (see *Scaling and Cropping Images*, on page 58).

Selecting Video Inputs

Each frame grabber can have up to four video sources connected directly to it. The **SetCamera()** function selects one of the four video inputs to be digitized. The **GetCamera()** function returns the currently selected input.

By default, PXC200 frame grabbers automatically detect the video format (NTSC or PAL/SECAM) on the active camera input. If you need to determine the video format of the current video source for use in your program, you can use the **VideoType()** function.

When you switch from one video input to another, there may be a delay before the frame grabber can synchronize to the new video input. Three factors determine the time that it takes to synchronize to a video input once you've switched to it: input video type, whether the cameras are genlocked or not, and brightness levels. If the cameras are all of the same video type, there should be a delay of no more than eight field times before re-synchronization occurs; if they are also genlocked, there will be no appreciable delay. (Cameras of different video types can't be genlocked.) If the cameras are not of the same video type, there may be a delay of as much as 2.5 seconds before re-synchronization occurs. If the brightness level differs between two cameras of the same video type, there may be some additional delay when switching. The optional Control Package reduces the synchronization delay to less than two field times for non-genlocked video sources of the same type (less than 0.5 seconds for non-genlocked video sources of different types), provides DC voltage restoration on all video inputs, and provides horizontal and vertical sync outputs and for genlocking video sources to make switching between sources instantaneous.

If the delay in detecting a video format change is too long, you can set the video type directly by using the **SetVideoDetect()** function to specify the type of video the frame grabber should expect. This forces the frame grabber to digitize the incoming video based on the video format you specify. You can specify the video format from a list of optional formats for NTSC, PAL, and SECAM. The **GetVideoDetect()** function returns the currently set video format.

SetVideoDetect() is also useful when you are genlocking the PXC200 to a video source using the horizontal and vertical sync drive signals available with the optional Control Package. In this case, the PXC200 might not be able to reliably detect the format of the incoming video, but you can use SetVideoDetect() to specify the format.

Counting Fields

You can use the **GetFieldCount()** function to count the number of fields the frame grabber has received. The counter normally reports the number of fields that have elapsed since the last reset of the frame grabber, but you can set the counter to start counting from any value by using the **SetFieldCount()** function.

If the frame grabber is not connected to a video source, it will produce an internal video sync pulse, so the field count will continue to increase even in the absence of video input. Since the field counter counts vertical sync pulses on the active input, switching input sources can cause irregular field counts, depending on the relative phase of the video inputs.

Adjusting the Video Image

The PXC200 provides a variety of adjustments you can make to the video signal to change the way the signal is processed and the appearance of the resulting captured image.

Setting Contrast and Brightness

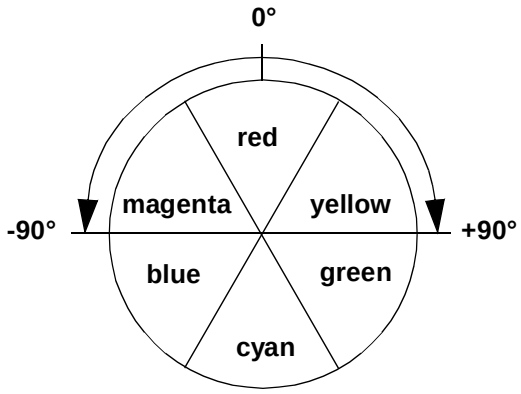
The contrast adjustment lets you lighten or darken the image. It's like a gain control on the monochrome part of the video signal. Contrast can be adjusted from 0.0 to 2.0. A contrast value of 1.0 leaves the signal unchanged. You set the contrast adjustment using the **SetContrast()** function. The **GetContrast()** function returns the current contrast adjustment.

The brightness adjustment acts as an offset for the monochrome part of the video signal. The brightness can be adjusted from -0.5 to +0.5. A value of +0.5 increases the digitized value of black to medium gray, and a value of -0.5 brings the digitized value of white to medium gray. A value

of 0.0 leaves the digitized value unchanged. You set the brightness adjustment using the **SetBrightness()** function. The **GetBrightness()** function returns the current brightness adjustment.

Setting Hue and Saturation

The hue adjustment lets you shift the colors in the image. Adjusting the hue is like rotating the color wheel, shown below. Positive values for the hue adjustment shift colors displayed as red toward yellow and green; negative values shift reds toward magenta and blue.

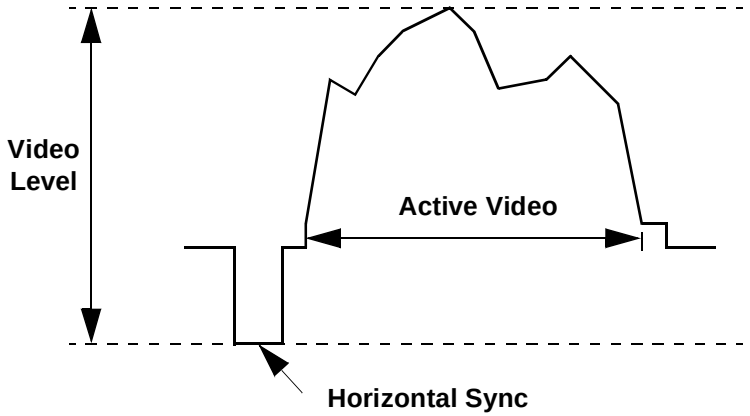


You set the hue adjustment using the **SetHue()** function. The **GetHue()** function returns the current hue adjustment. For NTSC video, you can adjust the hue from -90° to +90°. Because of the nature of PAL/SECAM signals, hue adjustments can't be made.

The saturation adjustment lets you change the intensity of the colors in the image. It's like a gain control on the color part of the video signal. Saturation can be adjusted from 0.0 to 2.0, with a value of 1.0 being normal. A saturation value of zero removes all color, leaving a monochrome image. You set the saturation adjustment using the **SetSaturation()** function. The **GetSaturation()** function returns the current contrast adjustment.

Setting the Video Level

The video level adjustment lets you set the expected amplitude range of the video signal from the bottom of the video sync portion of the signal to bright white. (See the drawing, below, of a video signal for a single horizontal line of video.) This value is normally 1.3 V, but can be set to any value in the range zero to 2.5 V for video sources that don't produce signals at the normal value. You set the video level using the **SetVideoLevel()** function. The **GetVideoLevel()** function returns the current video level adjustment.



Setting Luma Controls

The term *luma* refers to the monochrome part of the video signal. The luma control lets you specify several features the frame grabber can apply to processing the monochrome part of the video signal:

Low Filter—A low-pass filter that reduces high-frequency information in the video signal.

Core Function—Causes all video below a specified level to be digitized to black. Coring can improve the apparent contrast of some dark images.

Gamma Correction—Provides gamma correction for RGB video output. For NTSC, a gamma value of 2.2 is used; for PAL, the gamma value is 2.8.

Comb Filter—Activates a comb filter to reduce artifacts in the monochrome signal caused by crosstalk from the color signal.

Peak Filter—Activates a filter that amplifies high frequencies. This filter can sharpen edges in a blurry image, but might also cause artifacts on edges that are already sharp.

You set the luma control features using the **SetLumaControl()** function. The **GetLumaControl()** function returns the current setting for each luma control feature.

Setting Chroma Controls

The term *chroma* refers to the color part of the video signal. The chroma control lets you specify several features the frame grabber can apply to processing the color part of the video signal:

S-Video—Tells the frame grabber that the video signal is an S-video signal with separate color and monochrome channels, rather than a composite video signal. This causes the frame grabber to extract the color information from the separate video signal rather than from the composite signal. With the optional Control Package, all four video inputs support S-video; without the Control Package, only video input 1 supports S-video.

Notch Filter—Activates a filter to remove the color burst signal from the video signal before the signal is digitized. This prevents color arti-

facts from appearing in composite video, while still allowing the color information to be digitized.

Automatic Gain Control—Activates automatic gain control (AGC) for color saturation to compensate for non-standard color signals.

Monochrome Detect—Sets the color signal to zero when the board detects a missing or weak color burst signal.

Comb Filter—Activates a comb filter to reduce color artifacts.

You set the chroma control features using the **SetChromaControl()** function. The **GetChromaControl()** function returns the current setting for each chroma control feature.

Scaling and Cropping Images

The resolution of full-size digitized images depends on the video format and the aspect ratio of your screen and pixels. Typical computer monitors have an aspect ratio of 4 x 3 and use square pixels. Conventional television monitors use rectangular pixels. Typical resolutions for several common formats are given below:

Video Format	Image Resolution
NTSC square pixels	640 x 480
NTSC rectangular pixels	720 x 480
PAL/SECAM square pixels	768 x 576
PAL/SECAM rectangular pixels	720 x 576

You can digitize images at these resolutions, or you can scale and crop the images, which saves memory and bandwidth for transferring and processing images.

Scaling Images

PXC200 frame grabbers can scale the video image by interpolating pixel values along both the horizontal and vertical axes. To scale an image, you simply specify the number of pixels you want along the horizontal and vertical axes using the **SetXResolution()** and **SetYResolution()** functions. The **GetXResolution()** and **GetYResolution()** functions return the current values. You can scale images down to approximately 1/16 size.

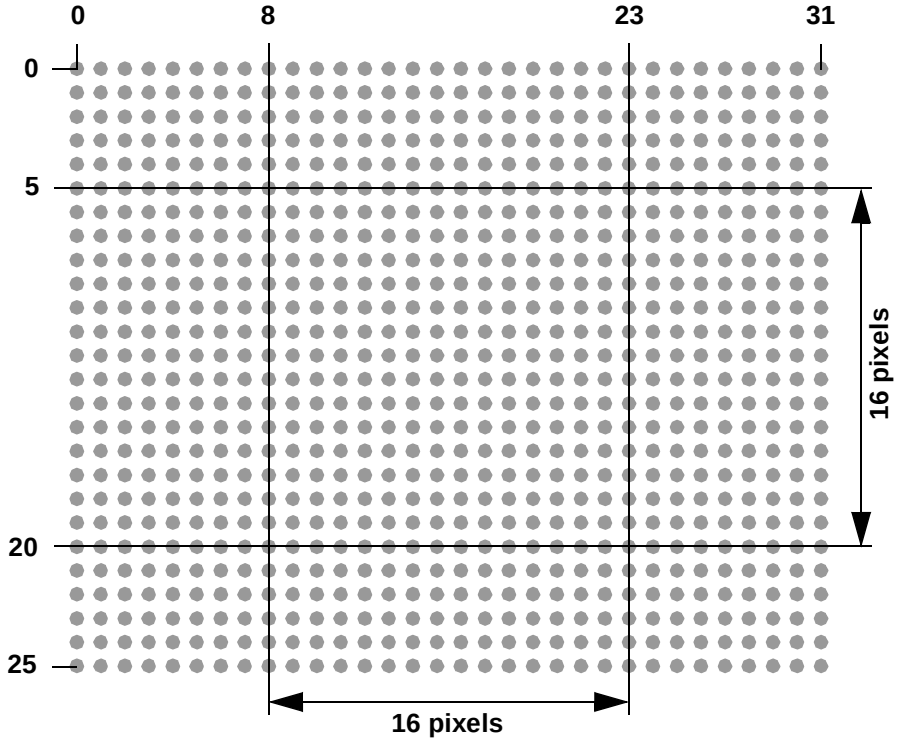
> **Note**
> *When working with small values for Y resolution, you can often get better image quality by specifying twice the desired Y resolution and using the SINGLE_FLD flag with the Grab() function. This eliminates field blur and other problems related to interlacing.*

Cropping Images

In addition to scaling images, you can crop images vertically and horizontally. 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()**.

The figure on page 60 shows an example of an NTSC image that has been scaled to 32 pixels by 26 pixels. If you want to crop the image to get a rectangular image 16 pixels by 16 pixels from the center of the scaled image, you would specify the cropping parameters as *left* = 8, *width* = 16, *top* = 5, and *height* = 16.

For all video formats, the default starting row is row four, and the default number of rows is 480. For PXC200 frame grabbers, row zero of the video image is the first row of valid video.



Note

NTSC and PAL/SECAM video signals have only a half row of valid video on the first and last rows of each frame. The first line (row zero for both formats) contains valid video for only the last half of the row. The last line (row 485 for NTSC, row 575 for PAL/SECAM) contains valid video for only the first half. If you include either of these rows in your image data, the entire row will be sampled.

Timing the Execution of Functions

The PXC200 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:

```
for (;;)
{
    pxc.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 PXC200 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.

PXC200 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*, on page 66). 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 = pxc.Grab(fgh, fbuf1, QUEUED);
grab2 = pxc.Grab(fgh, fbuf2, QUEUED);
pxc.WaitFinished(fgh, grab1)
    ;  /* wait until grab 1 has completed */
for (;;)
{
    ProcessImage(fbuf1);
    grab1 = pxc.Grab(fgh, fbuf1, QUEUED);
    pxc.WaitFinished(fgh, grab2)
        ;/* wait until grab 2 has completed */
    ProcessImage(fbuf2);
    grab2 = pxc.Grab(fgh, fbuf2, QUEUED);
    pxc.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 *fgh* that you specify.

Synchronizing Program Execution to Video

The library has two functions, Wait() and WaitVB(), that can be used to synchronize program execution to incoming video:

WaitVB() pauses until the end of the next vertical blank before returning. This is the most efficient way to synchronize program execution to video for non-queued functions.

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*, on page 66).

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.

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.
FIELD0	Operation will start on an even video field.
FIELD1	Operation will start on an odd video field.
SINGLE_FLD	Operation will only apply to one 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 PXC200. The PXC200 includes a single digital input (line 0) that lets you synchro-

nize 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*, on page 71, applies to your board, and the functions in *Controlling the Input Lines*, below, can only be used with line 0.

Controlling the Input Lines

You can use the input lines to read information from an external device and to initiate actions in your program. For example, you could use an input line to trigger the frame grabber to capture an image on a signal from a camera or other external device.

Setting Up and Reading the Input Lines

You use the **SetIOType()** function to set up the input lines. You can set up an input line so that the state of the line will be set for any of the following conditions:

Rising signal—signal changed from low to high.

Falling signal—signal changed from high to low.

Input signal—signal is high (the default).

On the standard board (without the optional Control Package) rising and falling triggers on the single trigger input are detected immediately. With the optional Control Package, rising and falling triggers on any of the inputs are always detected at vertical blank, so only one transition per field will be detected.

The **GetIOType()** function returns the current type of an I/O line, as set by the SetIOType() function.

The **ReadIO()** function returns the current state of all I/O lines. Bits 0-3 represent the input lines, and bits 4-7 represent the output lines.

Dealing with Trigger Bounce on Input Lines

Mechanical switches used as the trigger input can bounce (create spurious edges) when opening or closing. This can cause problems when you set the input to watch for signal edges or transitions. For example, a switch to ground will cause a falling edge when the switch closes, but will also cause more falling edges when the switch reopens, due to the microscopic bounce of the switch contacts. This would cause the PXC200 to detect multiple triggers where only one real trigger event has occurred.

With debounce compensation, after a trigger and the software acknowledge, new triggers are locked out until the trigger input has returned to the untriggered state during at least one vertical blank. This means that when you use debounce compensation with an edge-triggered input, you won't be able to grab two consecutive fields. In the switch example given above, the switch would need to be open for at least one field time before closing again.

The **SetDebounce()** function lets you set the length of the debounce delay and whether to debounce both the latched edge and the inactive edge of the signal for each input line. **GetDebounce()** returns the currently set values for an input line. The debounce functions apply only to boards that have the optional Control Package.

Using an Input Line as a Trigger

Using an input line to initiate some action typically involves the following steps in a program:

1 Set up the line to change state when the signal on the line changes.

2 Queue a WaitAnyEvent() or WaitAllEvents() function to wait for the state of the line to change.

3 Queue a follow-on action to take place when the event has been detected.

You can program the **WaitAnyEvent()** function to watch the state of one or more input lines. When WaitAnyEvent() reaches the top of the queue, processing of the queue pauses until at least one of the watched lines is in the specified state; then, the next function in the queue is processed. For example, you can set up WaitAnyEvent() to watch for lines 0 and 3 to be set, and the program will pause as long as the states of both lines are clear. As soon as the state of either (or both) lines is set, the program will resume processing the queue.

A common use for a trigger input on the PXC200 is to initiate a capture when the trigger signal is detected. You can accomplish this with these two lines of code:

```
pxc.WaitAnyEvent(fgh, fgh, 1, 0, QUEUED);
pxc.Grab(fgh, frh, flags);
```

The **WaitAllEvents()** function pauses processing of the queue until **all** of the watched lines are at the specified state. For example, if you set up WaitAllEvents() to watch for lines 0 and 3 to be clear, the state of both lines must be clear before processing the queue resumes. As long as the state of at least one of the lines is set, processing the queue remains suspended.

You designate which lines WaitAnyEvent() and WaitAllEvents() should watch, and which state to watch for, by setting a *state* parameter and a

mask parameter in the function call. Both functions read the I/O lines, as ReadIO() would, and evaluate the expression:

(ReadIO() ^ !*state*) & *mask*

where "^" is the bitwise exclusive OR operator, and "&" is the bitwise AND operator. This lets you designate the state (0 or 1) to watch for on each line and limits the lines watched to those with a value of 1 in the mask. Bits 0-3 in both *state* and *mask* represent the input lines 0-3 on the PXC200.

The WaitAnyEvent() and WaitAllEvents() functions also let you use the I/O lines on one frame grabber to trigger events on another frame grabber by specifying the handles for the two frame grabbers in the function call.

When processing continues, WaitAnyEvent() and WaitAllEvents() set a switch and clear the state of the input line. The *switch* is set to the number of the highest line that had a state of 1. If the state of more than one of the watched lines is a 1, WaitAnyEvent() clears only the state of the highest-numbered line, while WaitAllEvents() clears all lines. For example, if both lines 0 and 3 have a state of 1, the switch will be set to 3; WaitAnyEvent() will clear only the state of line 3, while WaitAllEvents() will clear both lines 0 and 3.

The follow-on operation in the queue can be a Grab() or any other function that can be queued (see the list on page 65).

Several functions are specifically designed to work with the switch value set by the WaitAnyEvent() and WaitAllEvents() functions:

GetSwitch()—Returns the current value of the switch. You can use the value returned to control the flow of your program.

SwitchGrab()—Performs a Grab() to capture an image, but sends the image to one of four possible frames depending on the value of the switch.

SwitchCamera()—Performs a SetCamera(), selecting one of the four possible video input sources based on the value of the switch.

For example, the following code causes the camera input to switch to input 0 if trigger 0 is received, to input 1 if trigger 1 is received, and so on:

```
pxc.WaitAnyEvent(fgh, fgh, 0x0F, 0x0F, QUEUED);
pxc.SwitchCamera(fgh);
```

The switch value is cleared when the frame grabber is reset by calling the Reset() function.

Controlling the Output Lines

You can use the output lines (available with the optional Control Package) to send timing signals or other information to an external device. For example, you could use output lines to send a programmed sequence of strobe pulses to a camera or other device. You can control the output lines either by writing the state of the lines directly or by using the automatic strobe functions.

Writing to the Output Lines

You can set the state of the output lines using the **WriteImmediateIO()** function. You designate which lines to set, and which state to set for each, by setting a *state* parameter and a *mask* parameter in the function call. Any line for which the *mask* bit is set to 1 will have its state set to the value of the corresponding bit in *state*. The function will fail if all mask bits are zero.

On boards with latched input lines, you can use the WriteImmediateIO() function to clear the input line after you read the line.

The **ReadIO()** function returns the current state of all I/O lines. You can use the SetIOType() function with output lines 4-7, but the only valid type for these lines is IO_OUTPUT.

Using the Automatic Strobe Functions

While you can control the output lines of the optional Control Package with the WriteImmediateIO() function, the frame grabber library includes dedicated strobe functions that simplify creating strobe pulses on the output lines.

> **Note**
> *The automatic strobe functions assume a stable incoming video signal. If the video signal is absent or unstable, strobe pulse timing will be inaccurate.*

The functions for firing the strobes are:

FireStrobe()—Fires a strobe pulse once on the specified output lines.

SyncStrobe()—Fires a strobe pulse at the specified line of the incoming video field on the specified output lines. SyncStrobe() continues to fire for each field until you disable it.

TriggerStrobe()—Fires a strobe pulse once on the specified output lines in response to a trigger on an input line.

> **Note**
> *If the input line is set to be edge-sensitive, the strobe will trigger immediately when the board sees the edge, even though software functions such as ReadIO(), WaitAnyEvent(), and WaitAllEvents() won't register the trigger input until the next vertical blank.*

Several functions are also provided for setting up various features of the strobe pulses:

SetStrobePeriod()—Sets the duration of the strobe pulse for each line.

SetDoubleStrobe()—Sets up output line 7 to output two strobe pulses and specifies the gap that separates the pulses. The width of both pulses is determined by SetStrobePeriod(). This function works only on line 7.

SetHoldoffMask(), **SetHoldoffStart()**, and **SetHoldoffWidth()**—Sets up a *holdoff period* for the strobes. The holdoff period is defined by specifying a starting line number in an incoming video field and a number of lines for the duration of the holdoff period. When the FireStrobe() or TriggerStrobe() functions execute during the holdoff period, the strobes are delayed, firing at the end of the holdoff period rather than immediately. The holdoff period is ignored for the SyncStrobe() function.

Each of these functions has a corresponding function for determining the current setting: **GetStrobePeriod()**, **GetDoubleStrobe()**, **GetHoldoffMask()**, **GetHoldoffStart()**, and **GetHoldoffWidth()**.

All of the output lines are initialized with SyncStrobe() disabled, double pulses disabled, a strobe length of 1.088 ms (17 scan lines), and a holdoff period of one line starting at line 9 for field 0 and at line 8 for field 1.

Horizontal and Vertical Sync Drive Signals

The horizontal and vertical sync drive signals available with the optional Control Package are always enabled, so there is no software interface for these signals.

Error Handling

The **CheckError()** function returns a flag if any of the following errors have occurred:

Invalid frame grabber handle—CheckError() was called using an invalid handle for the frame grabber.

Corrupt data—A captured image was transferred incorrectly and might contain bad data.

Overflow—The incoming video signal exceeds the range of the digitizer.

Error flags get cleared every time CheckError(), AllocateFG(), or Reset() are called.

You can use the **Reset()** function to restore the frame grabber to its default state. Reset() aborts any operations pending in the queue and the digital I/O.

Reading Frame Grabber Information

Board Revision Number

The frame grabber has a revision number encoded in it, which can be read using the **ReadRevision()** function. In most cases you won't need this function. If you need your revision number for calling Imagenation Technical Support, use one of these easy methods:

DOS or DOS/4GW—Run the PXCREV program.

Any version of Windows—Run either of the PXCDRAW sample programs. The revision number appears in the title bar.

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.

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.

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 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 program; 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 and Windows BMP 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.

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.

Using the Video Display DLL

The Video Display DLL is a simple tool for displaying video images in a window. Since it is a standard DLL, it can be used with Visual Basic, C, and other languages that can call DLLs. The Video Display DLL supports only one operation: copying an arbitrary rectangle of an image frame onto an arbitrary rectangle of a window's client area. There are two functions that are needed for this purpose:

void pxSetWindowSize(int x, int y, int dx, int dy) This function specifies the position and size of the rectangle where the image will be drawn, in units of pixels relative to the client area of the window where the drawing takes place. If pxSetWindowSize() is never called, the default values are $x = 0$, $y = 0$, $dx = 640$, and $dy = 512$.

void pxPaintDisplay(HDC hdc, FRAME __PX_FAR *frh, int x, int y, int dx, int dy) This function takes the rectangular area specified by *x*, *y*, *dx*, and *dy* from the frame *frh*, stretches it to fit the rectangle set by pxSetWindowSize(), and draws it into the device context *hdc*, which should be a valid device context for the window in which the image is to appear.

The frame pointer used by pxPaintDisplay() must reference a valid frame created by a call to the Frame DLL. This means that the library must be initialized properly and a frame must be allocated before the Video Display DLL can be used.

The Video Display DLL doesn't necessarily use the most efficient techniques to pipe the video information to a window. It is intended to be a tool to make video display as easy as possible, and may not be the best solution if you are concerned primarily with performance.

To incorporate the Video Display DLL into your programs, you will need these files:

16-bit Windows Programs	**32-bit Windows Programs**
VIDEO_16.H	VIDEO_32.H
VIDEO_16.LIB	VIDEO_32.LIB or VIDEO32B.LIB*
VIDEO_16.DLL	VIDEO_32.DLL
VIDEO_16.BAS	VIDEO_32.BAS

* Use VIDEO_32.LIB for Microsoft and VIDEO32B.LIB for Borland.

To link to the DLL, you must include the .BAS files in a Visual Basic program. If you want to use this DLL with a C program, you must put the prototypes of the functions (as they appear on page 78) in your program's source or header files; these prototypes do not appear in the main header files.

PXC200 Library Reference

5

The chapter is a complete, alphabetical function reference for the PXC200 Frame Grabber libraries and DLLs. For additional information on using the functions, see Chapter 4, *Programming the PXC200*, on page 33. For reference information on the Frame library, see Chapter 6, *Frame Library Reference*, on page 121.

The 16-bit Windows 3.1, *PXC2_16.DLL*, uses the Pascal calling convention. The 32-bit Windows 95, *PXC2_95.DLL*, uses the _stdcall calling convention.

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 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 and Visual BASIC. The following table gives the sizes of the various data types that are used by the PXC200 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:

Does not apply to Visual Basic.

AllocateBuffer()

Syntax FRAME __PX_FAR *AllocateBuffer(short dx, short dy, short type);

Return Value A handle to the allocated FRAME structure.
NULL on failure.

Description Reserves memory for an image buffer of size *dx* by *dy*, with the specified pixel data *type*. 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 frame when it is no longer needed.

For more information and a list of pixel data types, see *Allocating and Freeing Frames*, on page 48.

See Also **FreeFrame()**

AllocateFG()

Syntax long AllocateFG(short n);

Return Value A handle for the requested frame grabber.
0 on failure.

Description AllocateFG() attempts to find a frame grabber and give the program access to it. The program can request a specific frame grabber in a system that has more than one by specifying a number, *n*. Due to the design of the PCI bus, bus slot *0* doesn't necessarily correspond to frame grabber *0*, 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 PCXVU program (or your own program) to switch between frame grabbers. To request any available frame grabber, specify *n* = -1.

If the frame grabber is available, AllocateFG() returns a handle that must be used in other library functions that refer to the frame grabber.

The program should call FreeFG() on the frame grabber when it is no longer needed.

For more information, see *Requesting Access to Frame Grabbers*, on page 47.

See Also **FreeFG()**

CheckError()

Syntax long CheckError(long fgh);

Return Value 0 if no errors have occurred.
1 if the handle *fgh* is invalid.
One or more of these flags if an error has occurred:

Error Returned	Description
ERR_CORRUPT	A captured image was transferred incorrectly and might contain bad data.
ERR_IO_FAIL	The state of the digital I/O lines does not match the state the software set them to.
ERR_NOT_VALID	*fgh* is not a valid frame grabber handle.
WARN_OVERFLOW	The video signal exceeds the range of the digitizer.

Description CheckError() queries the frame grabber to determine whether any of a known set of errors occurred. These errors are automatically cleared when CheckError() returns and by successful calls to AllocateFG() and Reset().

CloseLibrary()

DOS Syntax void PXC200_CloseLibrary(FGLIB __PX_FAR *interface);

Win C Syntax void imagenation_CloseLibrary(FGLIB __PX_FAR *interface);

Win VB Syntax CloseLibrary(0)

Return Value None.

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.

For more information, see *Initializing and Exiting Libraries*, on page 44.

See Also **OpenLibrary()**

FireStrobe()

Syntax short FireStrobe(long fgh, long mask);

Return Value Non-zero if successful.
0 on failure.

Description Causes any output lines specified by 1 bits in *mask* to immediately begin a strobe pulse. FireStrobe() will fail if any bits other than 4-7 are set. This function executes concurrently with any queued functions. For more information see *Using the Automatic Strobe Functions*, on page 72.

See Also **SetDoubleStrobe(), SetHoldoffMask(), SetHoldoffStart(), SetHoldoffWidth(), SetStrobePeriod(),**

FreeFG()

Syntax void FreeFG(long fgh);

Return Value None.

Description Releases control of a frame grabber (previously allocated with the AllocateFG() function) after the program is finished using the frame grabber.

See Also **AllocateFG()**

FreeFrame()

Syntax void FreeFrame(FRAME __PX_FAR *f);

Return Value None.

Description Returns memory associated with a FRAME handle to the system. You must free all frames allocated by AllocateBuffer() before calling CloseLibrary()

This function is identical to the FreeFrame() function in the Frame library. Either version of the function can free a frame allocated by either library.

See Also **AllocateBuffer()**

GetBrightness()

Syntax float GetBrightness(long fgh);

Return Value The current brightness setting.
0 on failure.

Description Returns the current brightness (monochrome offset) setting for the frame grabber. This function executes concurrently with any queued functions. If a SetBrightness() function is queued when GetBrightness() is called, either function might execute first, affecting the result returned by GetBrightness().

See Also **SetBrightness()**, **SetContrast()**

GetCamera()

Syntax short GetCamera(long fgh);

Return Value The currently active video input.
-1 on failure.

Description Returns the active video input of the specified frame grabber. Use SetCamera() to specify the active video input. If a SetCamera() function is queued when GetCamera() is called, either function might execute first, affecting the result returned by GetCamera().

See Also **SetCamera()**

GetChromaControl()

Syntax short GetChromaControl(long fgh);

Return Value A set of flags if successful.
-1 on failure.

Description Returns a set of flags for the optional features for processing the color portion of the video signal. The flag values are listed for the function *SetChromaControl()*, on page 101. For more information, see *Setting Chroma Controls*, on page 57.

See Also **SetChromaControl()**

GetContrast()

Syntax float GetContrast(long fgh);

Return Value The current contrast setting.
0 on failure.

Description Returns the current contrast (monochrome gain) setting for the frame grabber. This function executes concurrently with any queued functions. If a SetContrast() function is queued when GetContrast() is called, either function might execute first, affecting the result returned by GetContrast().

See Also **SetBrightness()**, **SetContrast()**

GetDebounce()

Syntax short GetDebounce(long fgh, short n);

Return Value The currently set debounce mode if successful.
-1 if fgh is invalid.

Description Returns the currently set debounce mode.

See Also **SetDebounce()**

GetDoubleStrobe()

Syntax float GetDoubleStrobe(long fgh, short n);

Return Value The currently set value if successful.
0.0 if the double strobe is disabled.
-1.0 on failure.

Description Returns the currently set gap, in seconds, between double strobes on output line *n*.

See Also **SetDoubleStrobe()**

GetFieldCount()

Syntax long GetFieldCount(long fgh);

Return Value The field count if successful.
0 if fgh is not a valid handle.

Description Returns the number of fields the frame grabber has received since the last reset of the board. You can set the starting count by using the SetFieldCount() function.

See Also **SetFieldCount()**

GetHeight()

Syntax short GetHeight(long fgh);

Return Value The currently set height if successful.
0 if fgh is invalid.

Description Returns the height in pixels of the cropped image, as set by SetHeight(). The top-most pixel in the cropped image is set with SetTop().

This function waits until the frame grabber queue is empty before executing.

See Also **SetHeight()**, **SetTop()**

GetHoldoffMask()

Syntax long GetHoldoffMask(long fgh);

Return Value The currently set holdoff mask if successful.
-1 on failure.

Description Returns the currently set mask that defines which output lines are affected by the holdoff period.

See Also **SetHoldoffMask()**

GetHoldoffStart()

Syntax short GetHoldoffStart(long fgh, short field);

Return Value The currently set starting line of the holdoff period if successful.
-1 on failure.

Description Returns the currently set starting line of the incoming video signal that defines the beginning of the holdoff period. Valid values for *field* are FIELD0 and FIELD1.

See Also **SetHoldoffStart()**

GetHoldoffWidth()

Syntax short GetHoldoffWidth(long fgh, short field);

Return Value The currently set duration of the holdoff period if successful.
0 on failure.

Description Returns the currently set number of lines of the incoming video signal that defines the duration of the holdoff period. Valid values for *field* are FIELD0 and FIELD1.

See Also **SetHoldoffWidth()**

GetHue()

Syntax float GetHue(long fgh);

Return Value The current hue setting if successful.
0 on failure.

Description Returns the current hue setting for the frame grabber. This function executes concurrently with any queued functions. If the SetHue() function is queued when GetHue() is called, either function might execute first, affecting the result returned by GetHue(). For more information, see *Setting Hue and Saturation*, on page 55.

See Also **SetHue()**

GetInterface()

Syntax const void __PX_FAR *GetInterface(long handle);

Return Value None.

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.

See Also **OpenLibrary()**

GetIOType()

Syntax short GetIOType(long fgh, short n);

Return Value Type of I/O line n if successful.
0 on failure.

Description Returns the type of I/O line number *n*, where $0 \le n \le 7$, and the type is one of the following:

Return Value	Description
LATCH_RISING	The state of the line will be set to 1 if the signal changes from low to high.
LATCH_FALLING	The state of the line will be set to 1 if the signal changes from high to low.
IO_INPUT	The state of the line is equal to the signal value.
IO_OUTPUT	The line is an output line.

Lines 0-3 are input lines. Lines 4-7 are output lines and always return IO_OUTPUT. For more information, see *Digital I/O*, on page 66.

See Also **SetIOType()**

GetLeft()

Syntax short GetLeft(long fgh);

Return Value The currently set left edge if successful.
0 if fgh is invalid.

Description Returns the left-most pixel of the cropped image, as set by SetLeft(). The width of the cropped image is set with SetWidth().

This function waits until the frame grabber queue is empty before executing.

See Also **SetLeft()**, **SetWidth()**

GetLumaControl()

Syntax short GetLumaControl(long fgh);

Return Value A set of flags if successful.
-1 on failure.

Description Returns a set of flags for the optional features for processing the monochrome portion of the video signal. The flag values are listed for the function *SetLumaControl()*, on page 108. For more information, see *Setting Luma Controls*, on page 56.

See Also **SetLumaControl()**

GetSaturation()

Syntax float GetSaturation(long fgh);

Return Value The current saturation setting if successful.
0 on failure.

Description Returns the current saturation adjustment. This function executes concurrently with any queued functions. If the SetSaturation() function is queued when GetSaturation() is called, either function might execute first, affecting the result returned by GetSaturation(). For more information, see *Setting Hue and Saturation*, on page 55.

See Also **SetSaturation()**

GetStrobePeriod()

Syntax float GetStrobePeriod(long fgh, short n);

Return Value The currently set strobe period in seconds if successful.
-1 on failure.

Description Returns the strobe period, in seconds, currently set for line *n*, where *n* is one of the four output lines 4-7.

See Also **SetStrobePeriod()**

GetSwitch()

Syntax short GetSwitch(long fgh);

Return Value The number of the I/O line.
0 if neither of the Wait functions has completed.
-1 on failure.

Description When a WaitAllEvents() or WaitAnyEvent() function completes, the function sets the switch value to the number of the highest I/O line that satisfied the wait condition. GetSwitch() returns the line number, or zero if the function hasn't yet completed. For more information, see *Controlling the Input Lines*, on page 67.

This function executes concurrently with any queued functions. If another WaitAllEvents() or WaitAnyEvent() function is queued when GetSwitch() is called, either function might execute first, affecting the result returned by GetSwitch().

See Also **WaitAllEvents()**, **WaitAnyEvent()**

GetTop()

Syntax short GetTop(long fgh);

Return Value The currently set top edge if successful.
0 if fgh is invalid.

Description Returns the top-most pixel of the cropped image, as set by SetTop(). The height of the cropped image is set with SetHeight().

This function waits until the frame grabber queue is empty before executing.

See Also **SetHeight()**, **SetTop()**

GetVideoDetect()

Syntax short GetVideoDetect(long fgh);

Return Value The currently-set video type if successful.
-1 on failure.

Description Returns the video type expected by the frame grabber, as set by SetVideoDetect().

See Also **SetVideoDetect()**

GetVideoLevel()

Syntax float GetVideoLevel(long fgh);

Return Value The current video level if successful.
0 on failure.

Description Returns the voltage difference between the bottom of video sync and bright white, as set by SetVideoLevel(). For more information, see *Setting the Video Level*, on page 56.

See Also **SetVideoLevel()**

GetWidth()

Syntax short GetWidth(long fgh);

Return Value The currently set width if successful.
0 if fgh is invalid.

Description Returns the width in pixels of the cropped image, as set by SetWidth(). The left-most pixel in the cropped image is set with SetLeft().

This function waits until the frame grabber queue is empty before executing.

See Also **SetLeft()**, **SetWidth()**

GetXResolution()

Syntax short GetXResolution(long fgh);

Return Value The current X resolution if successful.
0 if fgh is invalid.

Description Returns the number of pixels the frame grabber will digitize per row of video, as set by SetXResolution(). The captured image might be fewer pixels in width if the image has been cropped with SetLeft() and SetWidth().

See Also **SetLeft()**, **SetWidth()**, **SetXResolution()**

GetYResolution()

Syntax short GetYResolution(long fgh);

Return Value The current Y resolution if successful.
0 if fgh is invalid.

Description Returns the number of pixels the frame grabber will digitize vertically per frame of video, as set by SetYResolution(). The captured image might be fewer pixels in height if the image has been cropped with SetTop() and SetHeight().

See Also **SetHeight()**, **SetTop()**, **SetYResolution()**

Grab()

Syntax long Grab(long fgh, FRAME __PX_FAR *frh, short flags);

Return Value A queued operation handle if successful.
0 on failure.

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. For more information, see *Grabbing Images*, on page 51.

The parameter *flags* is a set of flag bits that can specify modes of operation for this function. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 66.

See Also **AllocateFG()**, **AllocateBuffer()**, **GrabContinuous()**, **SwitchGrab()**

GrabContinuous()

Syntax long GrabContinuous(long fgh, FRAME __PX_FAR *frh, short state, short flags);

Return Value Non-zero if successful.
0 on failure.

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. Using continuous acquire mode while other memory accesses or PCI accesses are occurring might require more data to be transferred than is possible on some computers, resulting in corrupt video data. The Grab functions can’t determine when data corruption is occurring, but CheckError() will return ERR_CORRUPT.

The Grab() and SwitchGrab() functions 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.

For more information, see *Grabbing Images*, on page 51.

The parameter *flags* can specify additional modes of operation for this function. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 66.

See Also **Grab()**, **SwitchGrab()**

IsFinished()

Syntax short IsFinished(long fgh, int handle);

Return Value >0 if the operation is not in the queue.
0 if the specified operation is in the queue and has not completed.
-1 if the specified frame grabber is invalid.

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. For more information on queued functions, see *Timing the Execution of Functions*, on page 60.

Many frame grabber control functions can queue operations if they are passed the appropriate flags. For more information, see *Using Flags with Function Calls*, on page 66.

See Also **WaitFinished()**

KillQueue()

Syntax void KillQueue(long fgh);

Return Value None.

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.

This function takes several milliseconds to execute. It is intended primarily for recovering from error conditions.

See Also **Reset()**

OpenLibrary()

DOS Syntax short PXC200_OpenLibrary(FGLIB __PX_FAR *interface, short sizeof(interface));

Win C Syntax short imagenation_OpenLibrary(LPSTR dllname, __PX_FAR *interface, short sizeof(interface));

Win VB Syntax integer OpenLibrary(0,0)

Return Value Number of available frame grabbers.
0 on failure.

Description Initializes library data structures and locates all available frame grabbers. It must be called successfully before any other library functions can be used.

OpenLibrary() will usually fail only if no frame grabbers are detected, but may also fail under conditions of extremely low memory.

For more information on using OpenLibrary(), see *Initializing and Exiting Libraries*, on page 44.

See Also **CloseLibrary()**

ReadIO()

Syntax unsigned long ReadIO(long fgh);

Return Value The state of the I/O lines if successful.
0 on failure.

Description Returns a set of bit flags indicating the state of the I/O lines. Bits 0 - 3 correspond to input lines 0 - 3. Bits 4 - 7 correspond to output lines 4 - 7. Bits that have no associated I/O line return zero.

See Also **SetIOType()**, **WriteImmediateIO()**

ReadProtection()

Syntax short ReadProtection(long fgh);

Return Value The protection key if successful.
0 on failure.

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.

ReadRevision()

Syntax short ReadRevision(long fgh);

Return Value The revision number if successful.
0 on failure.

Description Returns the hardware/firmware revision number of the frame grabber. If *fgh* = 0, ReadRevision() returns the revision number of the software library.

You can also get the revision number using the PXCREV utility program in DOS or any of the sample programs in Windows; the sample programs display the revision number in the title bar.

ReadSerial()

Syntax long ReadSerial(long fgh);

Return Value The serial number of the board if successful.
0 on failure.

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.

Reset()

Syntax void Reset(long fgh);

Return Value None.

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.

See Also **KillQueue()**

SetBrightness()

Syntax long SetBrightness(long fgh, float offset, short flags);

Return Value A queued operation handle if successful.
0 on failure.

Description Sets the *offset* value for the monochrome signal, where $-0.5 \leq offset \leq +0.5$. A value of +0.5 increases the digitized value of black to medium gray, and a value of -0.5 brings the digitized value of white to medium gray. For more information, see *Setting Contrast and Brightness*, on page 54.

The parameter *flags* is a set of flag bits that can specify modes of operation for this function. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 66.

See Also **GetBrightness()**, **SetContrast()**

SetCamera()

Syntax short SetCamera(long fgh, short n, short flags);

Return Value A queued operation handle if successful.
0 on failure.

Description Selects one of the video inputs (0-3) on the frame grabber to be active. The camera attached to the selected input is the source for all subsequent video input to the frame grabber.

The parameter *flags* is a set of flag bits that can specify modes of operation for this function. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 66.

See Also **GetCamera()**

SetChromaControl()

Syntax short SetChromaControl(long fgh, short cf);

Return Value Non-zero if successful.
0 on success.

Description Selects features for processing the color portion of the video signal. The parameter *cf* is a set of flags that can be combined with the OR operator to select specific features:

Flag	Description
SVIDEO	Color information is digitized from the separate chroma channel of the S-video input. If this flag is not set, color information is extracted from the composite video signal. With the optional Control Package, this flag affects all four video inputs simultaneously; without the Control Package, this flag affects only video input 1.
NOTCH_FILTER	Activates an analog filter to remove the color burst signal from the luminance channel before brightness information is digitized
AGC	Activates the automatic gain control for color saturation. If this flag is enabled, the board attempts to compensate for non-standard color burst amplitudes.

Flag	Description
BW_DETECT	Forces the board to output only monochrome video when the board detects weak or missing color burst signals.
COMB_FILTER	Activates digital filtering of the color data to reduce artifacts.

For more information, see *Setting Chroma Controls*, on page 57.

This function waits for the queue to empty before executing.

See Also **GetChromaControl()**

SetContrast()

Syntax long SetContrast(long fgh, float gain, short flags);

Return Value A queued operation handle if successful.
0 on failure.

Description Sets the monochrome *gain* for the frame grabber, where $0.0 \leq gain \leq 2.0$. The amplitude of the input signal is multiplied by the *gain*, so the contrast of the input signal is unchanged for *gain* = 1. For more information, see *Setting Contrast and Brightness*, on page 54.

The parameter *flags* is a set of flag bits that can specify modes of operation for this function. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 66.

See Also **GetContrast()**, **SetBrightness()**

SetDebounce()

Syntax short SetDebounce(long fgh, short n, short db);

Return Value Non-zero on success.
0 on failure.

Description

Sets the debounce mode for line *n*, where *n* is one of the four input lines 0-3. The debounce mode is set to the value defined by *db*, where *db* is a logical expression of zero or more of the following flags:

DEBOUNCE_LONG	Causes a delay of at least two vertical blanks before a new edge can be detected. If this flag is absent, the delay is one vertical blank.
DEBOUNCE_BOTH	Applies the debounce delay to both the latched edge and the returning edge. If this flag is absent, the inactive edge is not debounced.

The default value for the debounce mode is zero. For more information, see *Dealing with Trigger Bounce on Input Lines*, on page 68.

This function applies only to boards with the optional Control Package; boards without the control package always behave as if *db* = 0. This function executes concurrently with any queued functions.

See Also **GetDebounce()**, **SetIOType()**, **TriggerStrobe()**

SetDecisionPoint()

Syntax short SetDecisionPoint(long fgh, short field, short line);

Return Value Non-zero if successful.
0 on failure.

Description Sets the point during vertical blank where the queue is serviced, and the point where edge-sensitive inputs are latched, to a specific *line* for the specified *field* of the incoming video signal. This setting determines, among other things, when the frame grabber decides whether to grab the next field for a pending Grab() operation. Valid values for *field* are FIELD0, FIELD1, and EITHER. Valid values for *line* are $1 \leq line \leq 256$. If the decision point is set too late, the frame grabber won't be able to capture the next field. Recommended ranges are: $3 \leq line \leq 17$ for NTSC

and 3 ≤ *line* ≤ 22 for PAL. Default values are 9 for field 0 and 8 for field 1.

This function executes concurrently with any queued functions.

SetDoubleStrobe()

Syntax float SetDoubleStrobe(long fgh, short n, float gap);

Return Value The actual value set if successful.
0.0 if the double strobe is disabled.
-1.0 on failure.

Description Sets strobes on output line *n* to fire two pulses separated by *gap* seconds of delay. This function is valid only for *n* = 7. The duration of both pulses is set by SetStrobePeriod(). Setting *gap* ≤ 0.0 disables double strobes. For more information, see *Using the Automatic Strobe Functions*, on page 72.

This function executes concurrently with any queued functions.

See Also **FireStrobe()**, **SetStrobePeriod()**, **SyncStrobe()**, **TriggerStrobe()**

SetFieldCount()

Syntax short SetFieldCount(long fgh, long c);

Return Value Non-zero if successful.
0 if fgh is not a valid handle.

Description Sets the starting value for counting incoming video fields. You can get the number of fields that have elapsed since the field count was last set, or since the board was last reset, by using the GetFieldCount() function.

See Also **GetFieldCount()**

SetHeight()

Syntax short SetHeight(long fgh, short dy);

Return Value The actual height set if successful.
0 if fgh is invalid.

Description The height in pixels of the cropped image. The top-most pixel in the cropped image is set with SetTop(). The frame grabber sets the height to the closest value less than or equal to *dy* it is capable of and returns the actual value set. For more information, see *Scaling and Cropping Images*, on page 58.

This function waits until the frame grabber queue is empty before executing.

See Also **SetLeft()**, **SetTop()**, **SetWidth()**

SetHoldoffMask()

Syntax short SetHoldoffMask(long fgh, long mask);

Return Value Non-zero if successful.
0 on failure.

Description Determines which output lines, specified by 1 bits in *mask* (bits 4-7), are affected by the holdoff period set with SetHoldoffStart() and SetHoldoffWidth(). If FireStrobe() or TriggerStrobe() execute within the specified holdoff period, the actual firing of the strobe is delayed until the end of the holdoff period. By default, holdoff is enabled for all output lines, 4-7. For more information, see *Using the Automatic Strobe Functions*, on page 72.

This function executes concurrently with any queued functions.

See Also **FireStrobe()**, **SetHoldoffStart()**, **SetHoldoffWidth()**, **TriggerStrobe()**

SetHoldoffStart()

Syntax short SetHoldoffStart(long fgh, short field, short start);

Return Value Non-zero if successful.
0 on failure.

Description Specifies the line, *start*, of the incoming video signal that defines the first line of the holdoff period. If FireStrobe() or TriggerStrobe() execute within the specified holdoff period, the actual firing of the strobe is delayed until the end of the holdoff period. Separate holdoff periods can be defined for each *field* of the incoming video. Valid values for the *field* parameter are FIELD0, FIELD1, or EITHER. The default values for *start* are line 9 for field 0 and line 8 for field 1. For more information, see *Using the Automatic Strobe Functions*, on page 72.

This function executes concurrently with any queued functions.

See Also **FireStrobe()**, **SetHoldoffMask()**, **SetHoldoffWidth()**, **TriggerStrobe()**

SetHoldoffWidth()

Syntax short SetHoldoffWidth(long fgh, short field, short width);

Return Value Non-zero if successful.
0 on failure.

Description Specifies the number of lines, *width*, of the incoming video signal that defines the duration of the holdoff period. If FireStrobe() or TriggerStrobe() execute within the specified holdoff period, the actual firing of the strobe is delayed until the end of the holdoff period. Separate holdoff periods can be defined for each *field* of the incoming video. Valid values for the *field* parameter are FIELD0, FIELD1, or EITHER. The default value for *width* is one line. For more information, see *Using the Automatic Strobe Functions*, on page 72.

This function executes concurrently with any queued functions.

See Also **FireStrobe()**, **SetHoldoffMask()**, **SetHoldoffStart()**, **TriggerStrobe()**

SetHue()

Syntax long SetHue(long fgh, float h, short flags);

Return Value A queued operation handle if successful.
0 on failure.

Description Sets the hue adjustment to *h*, or the closest value the frame grabber is capable of, where -90 ≤ *h* ≤ +90. SetHue() is ignored for PAL/SECAM video signals.

For more information, see *Setting Hue and Saturation*, on page 55.

See Also **GetHue()**, **SetSaturation()**

SetIOType()

Syntax short SetIOType(long fgh, short n, short type);

Return Value Non-zero if successful.
0 on failure.

Description Sets the type of I/O line number *n*, where 0 ≤ *n* ≤ 7, and the type is one of the following:

Return Value	Description
LATCH_RISING	The state of the line will be set to 1 if the signal changes from low to high.
LATCH_FALLING	The state of the line will be set to 1 if the signal changes from high to low.
IO_INPUT	The state of the line is equal to the signal value. This is the default type for input lines 0-3.
IO_OUTPUT	The default type for output lines 4-7. This is the only valid type you can use with SetIOType() for the output lines.

Without the optional Control Package, only *n* = 0 is valid.

SetIOType() executes only after all functions in the queue have completed. For more information, see *Digital I/O*, on page 66.

See Also **GetIOType()**

SetLeft()

Syntax short SetLeft(long fgh, short x0);

Return Value The actual pixel position set if successful.
0 if fgh is invalid.

Description The left-most pixel of the cropped image. The width of the cropped image is set with SetWidth(). The frame grabber sets the left-most pixel to the closest value to *x0* it is capable of and returns the actual value set. For more information, see *Scaling and Cropping Images*, on page 58.

This function waits until the frame grabber queue is empty before executing.

See Also **SetHeight()**, **SetTop()**, **SetWidth()**

SetLumaControl()

Syntax short SetLumaControl(long fgh, short lf);

Return Value Non-zero if successful.
0 on failure.

Description

Selects features for processing the monochrome portion of the video signal. The parameter *lf* is a set of flags that can be combined with the OR operator to select specific features:

Flag	Description
LOW_FILT_AUTO LOW_FILT_1 LOW_FILT_2 LOW_FILT_3	Activates a low-pass filter that reduces high-frequency information in the video. LOW_FILT_3 has the highest level of filtering. LOW_FILT_AUTO automatically sets the filtering based on the resolution. Set, at most, one of these flags, or omit all for no filtering.
CORE_8 CORE_16 CORE_32	Forces any video with a brightness value less than n/256 (where n is 8, 16, or 32) to be digitized as black. Set, at most, one of these flags, or omit all for no coring.
GAMMA_CORRECT	Activates a filter to gamma correct RGB video. For NTSC, gamma = 2.2; for PAL/SECAM, gamma = 2.8. YCrCb video is never gamma corrected.
COMB_FILTER	Activates digital filtering of the brightness data to reduce artifacts.
PEAK_FILT_0 PEAK_FILT_1 PEAK_FILT_2 PEAK_FILT_3	Activates a filter that amplifies high-frequency information in the video. PEAK_FILT_0 has the highest gain. These filters will sharpen edges in a blurry video image, but might cause artifacts on edges that are already sharp. Set, at most, one of these flags, or omit all for no filtering.

For more information, see *Setting Luma Controls*, on page 56.

This function waits for the queue to empty before executing.

See Also

GetLumaControl()

SetPixelFormat()

Syntax short SetPixelFormat(long fgh, short type);

Return Value Non-zero if successful.
0 on failure.

Description Sets the pixel format that the frame grabber expects to digitize. Pixel types are listed in the table on page 48.

The frame grabber automatically changes to the correct format for the destination frame when a Grab(), GrabContinuous(), or SwitchGrab() function is executed, so using SetPixelFormat() explicitly is often not necessary. The frame grabber requires one field time of delay before it can digitize in a new format, whether the format change occurs due to calling SetPixelFormat() or due to the frame type for a Grab function. When the change occurs because of a Grab, this delay becomes part of the latency for the Grab. Using SetPixelFormat() to preset the expected pixel format minimizes the latency in the Grab function. For more information, see *Allocating and Freeing Frames*, on page 48

This function waits for the queue to empty before executing.

SetSaturation()

Syntax long SetSaturation(long fgh, float s, short flags);

Return Value A queued operation handle if successful.
0 on failure.

Description Sets the saturation adjustment to *s*, or the closest value the frame grabber is capable of, where $0.0 \leq s \leq 2.0$. For more information, see *Setting Hue and Saturation*, on page 55.

See Also **GetSaturation()**, **SetHue()**

SetStrobePeriod()

Syntax float SetStrobePeriod(long fgh, short n, float p);

Return Value The value of the period actually set if successful.
-1 on failure.

Description Sets line *n*, where *n* is one of the four output lines 4-7, to strobe for a period of *p* seconds when fired. The value of *p* is between approximately 64 microseconds and 4.2 seconds (1 to 0xFFFF scan lines). The value actually set is rounded to the closest available value. The default value is 1.088 ms (17 scan lines). For more information, see *Using the Automatic Strobe Functions*, on page 72.

SetStrobePeriod() assumes that the horizontal scan rate of incoming video is 64 µs per line. This function executes concurrently with any queued functions.

See Also **FireStrobe()**, **GetStrobePeriod()**, **SyncStrobe()**, **TriggerStrobe()**

SetTop()

Syntax short SetTop(long fgh, short y0);

Return Value The actual pixel position set if successful.
0 if fgh is invalid.

Description The top-most pixel of the cropped image. The height of the cropped image is set with SetHeight(). The frame grabber sets the top-most pixel to the closest value to *y0* it is capable of and returns the actual value set. For more information, see *Scaling and Cropping Images*, on page 58.

This function waits until the frame grabber queue is empty before executing.

See Also **SetHeight()**, **SetLeft()**, **SetWidth()**

SetVideoDetect()

Syntax short SetVideoDetect(long fgh, short type);

Return Value Non-zero if successful.
0 on failure.

Description Sets the video format the frame grabber should expect to *type*. Calling this function may cause the X resolution and Y resolution to change if the frame grabber can't support the current resolution in the new video format. Possible values for *type* are:

Value	Description
AUTO_FORMAT	The frame grabber will measure the field length and adjust to either NTSC or PAL video. Detecting a format change will take about 2.5 seconds for the standard board, or 0.5 seconds with the optional Control Package.
NTSC_FORMAT	The frame grabber expects NTSC video.
NTSCJ_FORMAT	The frame grabber expects NTSC with no pedestal voltage.
PAL_FORMAT	The frame grabber expects PAL B,D,G,H, or I video.
PALM_FORMAT	The frame grabber expects PAL M video.
PALN_FORMAT	The frame grabber expects PAL N video.
SECAM_FORMAT	The frame grabber expects SECAM video.

For more information, see *Selecting Video Inputs*, on page 52.

This function waits for the video queue to empty before executing.

See Also **VideoType()**

SetVideoLevel()

Syntax float SetVideoLevel(long fgh, float white);

Return Value The video level actually set if successful.
0 on failure.

Description Sets the expected voltage difference between the bottom of video sync and bright white, where $0.0 \leq white \leq 2.5$. The nominal level is 1.3 V. The function sets the video level to the closest value the frame grabber is capable of and returns the value actually set. For more information, see *Setting the Video Level*, on page 56.

This function waits for the queue to empty before executing.

See Also **GetVideoLevel()**

SetWidth()

Syntax short SetWidth(long fgh, short dx);

Return Value The actual width set if successful.
0 if fgh is invalid.

Description The width in pixels of the cropped image. The left-most pixel in the cropped image is set with SetLeft(). The frame grabber sets the width to the closest value less than or equal to *dx* it is capable of and returns the actual value set. For more information, see *Scaling and Cropping Images*, on page 58.

This function waits until the frame grabber queue is empty before executing.

See Also **SetHeight()**, **SetLeft()**, **SetTop()**

SetXResolution()

Syntax short SetXResolution(long fgh, short rez);

Return Value The actual resolution set if successful.
0 if fgh is invalid.

Description Sets the number of pixels the frame grabber will digitize per row of video. The frame grabber sets the resolution to the closest value to *rez* it is capable of and returns the actual value set. For more information, see *Scaling and Cropping Images*, on page 58.

This function waits until the frame grabber queue is empty before executing.

See Also **SetLeft()**, **SetWidth()**, **SetYResolution()**

SetYResolution()

Syntax short SetYResolution(long fgh, short rez);

Return Value The actual resolution set if successful.
0 if fgh is invalid.

Description Sets the number of pixels the frame grabber will digitize vertically per frame of video. The frame grabber sets the resolution to the closest value to *rez* it is capable of and returns the actual value set. For more information, see *Scaling and Cropping Images*, on page 58.

This function waits until the frame grabber queue is empty before executing.

See Also **SetHeight()**, **SetTop()**, **SetXResolution()**

SwitchCamera()

Syntax long SwitchCamera(long fgh, short flags);

Return Value A queued operation handle if successful.
0 on failure.

Description Sets the active video input to the switch value set by the last complete WaitAnyEvent() or WaitAllEvents() function. If the value of the switch is larger than the number of valid video inputs, the function does nothing. For more information, see *Controlling the Input Lines*, on page 67.

See Also **WaitAllEvents()**, **WaitAnyEvent()**

SwitchGrab()

Syntax long SwitchGrab(long fgh, FRAME __PX_FAR *f0, FRAME __PX_FAR *f1, FRAME __PX_FAR *f2, FRAME __PX_FAR *f3, short flags);

Return Value A queued operation handle if successful.
0 on failure.

Description This function behaves just like Grab(), except that the image data is written to one of four frames depending on the last WaitAnyEvent() or WaitAllEvents() function that completed. Some, but not all, of the frame pointers can be NULL; if a NULL frame pointer is selected, the function completes, but does nothing. For more information, see *Controlling the Input Lines*, on page 67.

This function fails if all frame pointers are NULL or if any of the frames don't have the correct width and height.

See Also **WaitAllEvents()**, **WaitAnyEvent()**

SyncStrobe()

Syntax short SyncStrobe(long fgh, short n, short field, short line);

Return Value Non-zero if successful.
0 on failure.

Description Fires a strobe pulse on output line *n* at the specified *line* of the specified *field* of the incoming video signal. Valid values for the *field* parameter are FIELD0, FIELD1, EITHER, or zero. The strobe fires repeatedly with each incoming field until you disable synchronous strobing by calling the

function again with *field* = 0. Valid values for *line* are 1 ≤ *line* ≤ 512, though currently, fields longer than the PAL standard of 288 lines are not supported. For more information, see *Using the Automatic Strobe Functions*, on page 72.

This function executes concurrently with any queued functions.

See Also **SetStrobePeriod()**

TriggerStrobe()

Syntax short TriggerStrobe(long fgh, short trig, long mask);

Return Value Non-zero if successful.
0 on failure.

Description Fires strobes on output lines with a 1 bit set in *mask* when a trigger is detected on input line *trig*. If the input line is set to LATCH_RISING or LATCH_FALLING, the strobe fires on the rising or falling edge of the trigger, respectively. If the input line is set to IO_INPUT, the strobe fires after the rising edge of the trigger. The strobe fires immediately after the trigger edge, even though software functions such as ReadIO(), WaitAllEvents(), and WaitAnyEvent() will not detect the trigger until the next vertical blank. For more information, see *Using the Automatic Strobe Functions*, on page 72.

This function executes concurrently with any queued functions.

See Also **SetDebounce()**, **SetIOType()**, **SetStrobePeriod()**

VideoType()

Syntax short VideoType(long fgh);

Return Value
0 No video.
1 NTSC video.
2 PAL/SECAM video.
3 Other.
-1 Invalid *fgh*.

Description Returns the type of video signal connected to the frame grabber: NTSC format, PAL/SECAM format, or other. The video source is determined by counting the number of lines per video frame. When the video line count doesn't match either NTSC or PAL/SECAM, or the frame grabber is not auto-detecting, the function returns *Other*.

See Also **SetVideoDetect()**

Wait()

Syntax long Wait(long fgh, short flags);

Return Value A queued operation handle if successful.
0 on failure.

Description Waits for the end of the next field, the end of the next frame (two complete fields), or the end of a specific field, depending on the *flags* you specify. The default behavior when *flags* = 0 is to wait for two complete fields.

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, 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. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 66.

See Also **WaitVB()**

WaitAllEvents()

Syntax

long WaitAllEvents(long fgh, long ioh, unsigned long mask, unsigned long state, short flags);

Return Value

A queued operation handle if successful.
0 on failure.

Description

Pauses processing of the queue until an I/O event occurs. WaitAllEvents() examines the I/O lines as if by the expression ((ReadIO(*ioh*) ^ !*state*) & *mask*). While the expression is not equal to *mask*, the queue is paused. If the expression is equal to *mask*, the state for the highest I/O line that was set is cleared, the switch is set to that I/O line number, and processing of the queue resumes. For more information, see *Controlling the Input Lines*, on page 67.

This function will fail when *mask* = 0 or when *mask* has any bits set that represent invalid I/O lines or lines that are output-only.

The parameter *flags* is a set of flag bits that can specify modes of operation for this function. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 66.

See Also

GetSwitch(), SetIOType(), SwitchCamera(), SwitchGrab(), WaitAnyEvent()

WaitAnyEvent()

Syntax

long WaitAnyEvent(long fgh, long ioh, unsigned long mask, unsigned long state, short flags);

Return Value

A queued operation handle if successful.
0 on failure.

Description

Pauses processing of the queue until an I/O event occurs. WaitAnyEvent() examines the I/O lines as if by the expression ((ReadIO(*ioh*) ^ !*state*) & *mask*). While the expression is zero, the queue is paused. If the expression is non-zero, the state for the highest I/O line that was set is cleared, the switch is set to that I/O line number, and pro-

cessing of the queue resumes. For more information, see *Controlling the Input Lines*, on page 67.

This function will fail when *mask* = 0 or when *mask* has any bits set that represent invalid I/O lines or lines that are output-only.

The parameter *flags* is a set of flag bits that can specify modes of operation for this function. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 66.

See Also **GetSwitch(), SetIOType(), SwitchCamera(), SwitchGrab(), WaitAllEvents()**

WaitFinished()

Syntax void WaitFinished(long fgh, long handle);

Return Value 1 if successful.
0 on failure.

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 *handle* returned by the function that queued the operation. For *handle* = 0, WaitFinished() waits until all operations in the queue have finished. For more information, see *Programming in a Multithreaded, Multitasking Environment*, on page 39.

See Also **IsFinished()**

WaitVB()

Syntax short WaitVB(long fgh);

Return Value Non-zero if successful.
0 on failure.

Description Waits until the next vertical blank. WaitVB() returns when the interrupt routine has completed; this is usually close to the beginning of vertical blank, but can be at any time during vertical blank depending on system

loading. WaitVB() returns too late for frame grabbing functions called immediately afterward to capture the field that has just begun.

See Also **Wait()**

WriteImmediateIO()

Syntax short WriteImmediateIO(long fgh, unsigned long mask, unsigned long state);

Return Value Non-zero on success.
0 on failure.

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.

On boards with latched input lines, you can use the WriteImmediateIO() function to clear the input line after reading the line.

For more information, see *Controlling the Output Lines*, on page 71.

See Also **ReadIO()**

Frame Library Reference

6

The chapter is a complete, alphabetical function reference for the Frame libraries and DLLs. For additional information on using the functions, see Chapter 4, *Programming the PXC200*, on page 33. For reference information on the PXC200 Frame Grabber library, see Chapter 5, *PXC200 Library Reference*, on page 81.

The 16-bit Windows 3.1, FRAME_16.DLL, uses the Pascal calling convention. The 32-bit Windows 95, FRAME_95.DLL, uses the _stdcall calling convention.

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 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 and Visual BASIC. The following table gives the sizes of the various data types that are used by the PXC200 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:

Does not apply to Visual Basic.

Does not apply to Windows NT.

AliasFrame()

Syntax FRAME __PX_FAR *AliasFrame(FRAME __PX_FAR *f, short x0, short y0, short dx, short dy, unsigned short type);

Return Value A pointer to the frame structure.
NULL on failure.

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.

AllocateAddress() NT

Syntax FRAME __PX_FAR *AllocateAddress(unsigned long address, short dx, short dy, unsigned short type);

Return Value A pointer to the frame structure.
NULL on failure.

Description Creates a frame of size *dx* by *dy*, with the specified pixel *type*, from the memory at the specified physical *address*. Both *dx* by *dy* must be greater than zero. AllocateAddress() can allocate any of the types listed on page 48, except the planar types. This function does not attempt to exclusively allocate the physical address space or to verify that writable memory actually exists there.

This function lets you program specialized operations, like peer-to-peer transfers between the frame grabber and another PCI device. It should not be used with linear addresses unless you know the processor's paging mode is disabled.

FreeFrame() should be called when the frame is no longer needed. This will de-allocate memory associated with the FRAME structure, but will not attempt to free any resources associated with the given buffer address.

See Also **FreeFrame()**

AllocateFlatFrame() NT

Syntax FRAME __PX_FAR *AllocateFlatFrame(short dx, short dy, unsigned short type);

Return Value A pointer to the frame structure.
NULL on failure.

Description Creates a frame of size *dx* by *dy*, with the specified pixel *type*, from unpaged, contiguous physical memory. 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. AllocateFlatFrame() can allocate any of the types listed on page 48, including the planar types. For planar types, the memory for each plane will be contiguous, but the three planes won't necessarily be in one contiguous block. Also, the frame structure itself is not necessarily in contiguous memory, only the image buffer.

AllocateFlatFrame() can fail if the system is not configured to allow contiguous buffers. The PXC200 doesn't need flat frames; this function is provided for compatibility with other products.

For more information and a list of pixel types, see *Allocating and Freeing Frames*, on page 48.

FreeFrame() should be called when the frame is no longer needed.

See Also **AllocateMemoryFrame()**, **FreeFrame()**

AllocateMemoryFrame()

Syntax FRAME __PX_FAR *AllocateMemoryFrame(short dx, short dy, unsigned short type);

Return Value A pointer to the frame structure.
NULL on failure.

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. AllocateMemoryFrame() can allocate any of the types listed on page 48.

For more information and a list of pixel types, see *Allocating and Freeing Frames*, on page 48.

FreeFrame() should be called when the frame is no longer needed.

See Also **AllocateFlatFrame()**, **FreeFrame()**

CloseLibrary()

DOS Syntax void FRAME_CloseLibrary(FRAMELIB __PX_FAR *interface);

Win C Syntax void imagenation_CloseLibrary(FRAMELIB __PX_FAR *interface);

Win VB Syntax CloseLibrary(0)

Return Value None.

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.

For more information, see *Initializing and Exiting Libraries*, on page 44.

See Also **OpenLibrary()**

CopyFrame()

Syntax short CopyFrame(FRAME __PX_FAR *source, short sourcex, short sourcey, FRAME __PX_FAR *dest, short destx, short desty, short dx, short dy);

Return Value Non-zero if successful.
0 on failure.

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 frame don't have the same pixel data type. For more information, see *Accessing Captured Image Data*, on page 76.

ExtractPlane()

Syntax FRAME __PX_FAR *ExtractPlane(FRAME __PX_FAR *f, short plane);

Return Value

Description Returns a frame that contains a single plane of the planar frame *f*. Returns NULL if *f* 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.

For more information, see *Accessing Captured Image Data*, on page 76.

FrameAddress() NT

Syntax unsigned long FrameAddress(FRAME __PX_FAR *f);

Return Value The physical address of the frame's image buffer.
0 on failure.

Description Returns the physical address of the specified frame's image buffer. If the frame's image buffer doesn't have a fixed physical address, the function fails.

The physical address can not, in general, be converted to a C-style pointer because of segmentation and paging of the processor's address space. In order to get a logical address (a pointer) to this buffer, use FrameBuffer().

This function is useful for writing low-level code, such as device drivers or memory managers, that need to interact with the frame grabber libraries.

See Also **FrameBuffer()**

FrameBuffer()

Syntax void __PX_HW *FrameBuffer(FRAME __PX_FAR *f);

Return Value The logical address of the frame's image buffer.
0 if the frame handle is invalid.

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.

See Also **FrameAddress()**

FrameHeight()

Syntax short FrameHeight(FRAME __PX_FAR *f);

Return Value The height of the frame in pixels.
0 if the frame handle is invalid.

Description Returns the height of a frame created with any of the Allocate functions. For more information, see *Accessing Captured Image Data*, on page 76.

See Also **FrameWidth()**

FrameWidth()

Syntax short FrameWidth(FRAME __PX_FAR *f);

Return Value The width of the frame in pixels.
0 if the frame handle is invalid.

Description Returns the width of a frame created with any of the Allocate functions. For more information, see *Accessing Captured Image Data*, on page 76.

See Also **FrameHeight()**

FrameType()

Syntax short FrameType(FRAME __PX_FAR *f);

Return Value The pixel data type of the frame.
0 if the frame handle is invalid.

Description Returns the pixel data type of the frame created with any of the Allocate functions. For more information and a list of the pixel data types, see *Allocating and Freeing Frames*, on page 48 and *Accessing Captured Image Data*, on page 76.

See Also **FrameHeight()**, **FrameWidth()**

FreeFrame()

Syntax void FreeFrame(FRAME __PX_FAR *f);

Return Value None.

Description Returns memory associated with a FRAME handle to the system. You must free all frames allocated by AllocateAddress(), AllocateFlatFrame(), and AllocateMemoryFrame() before calling CloseLibrary()

This function is identical to the FreeFrame() function in the PXC200 Frame Grabber library. Either version of the function can free a frame allocated by either library.

For more information and a list of the pixel data types, see *Allocating and Freeing Frames*, on page 48 and *Accessing Captured Image Data*, on page 76.

See Also **AllocateAddress(), AllocateFlatFrame(), AllocateMemoryFrame()**

GetColumn()

Syntax short GetColumn(FRAME __PX_FAR *f, void __PX_HUGE *buf, short column);

Return Value Non-zero if successful.
0 on failure.

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.

See Also **GetRow(), PutColumn(), PutRow()**

GetPixel()

Syntax short GetPixel(FRAME __PX_FAR *f, void __PX_HUGE *pixel, short x, short y);

Return Value Non-zero if successful.
0 on failure.

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.

See Also **PutPixel()**

GetRectangle()

Syntax short GetRectangle(FRAME __PX_FAR *f, void __PX_HUGE *buf, short x0, short y0, short dx, short dy);

Return Value Non-zero if successful.
0 on failure.

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.

See Also **PutRectangle()**

GetRow()

Syntax short GetRow(FRAME __PX_FAR *f, void __PX_HUGE *buf, short row);

Return Value Non-zero if successful.
0 on failure.

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.

See Also **GetColumn()**, **PutColumn()**, **PutRow()**

OpenLibrary()

DOS Syntax short FRAME_OpenLibrary(FRAMELIB __PX_FAR *interface, short sizeof(interface));

Win C Syntax short imagenation_OpenLibrary(LPSTR dllname, void __PX_FAR* interface, short sizeof(interface));

Win VB Syntax integer OpenLibrary(0,0)

Return Value Non-zero if successful.
0 on failure.

Description Initializes library data structures. It must be called successfully before any other library functions can be used.

For more information on using OpenLibrary(), see *Initializing and Exiting Libraries*, on page 44.

See Also **CloseLibrary()**

PutColumn()

Syntax void PutColumn(void __PX_HUGE *buf, FRAME __PX_FAR *f, short col);

Return Value Non-zero if successful.
0 on failure.

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.

See Also **GetColumn()**, **GetRow()**, **PutRow()**

PutPixel()

Syntax short PutPixel(void __PX_HUGE *pixel, FRAME __PX_FAR *f, short x, short y);

Return Value Non-zero if successful.
0 on failure.

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, undefined 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.

See Also **GetPixel()**

PutRectangle()

Syntax void PutRectangle(void __PX_HUGE *buf, FRAME __PX_FAR *f, int x0, short y0, short dx, short dy);

Return Value Non-zero if successful.
0 on failure.

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.

See Also **GetRectangle()**

PutRow()

Syntax short PutRow(void __PX_HUGE *buf, FRAME __PX_FAR *f, short row);

Return Value Non-zero if successful.
0 on failure.

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.

See Also **GetColumn(), GetRow(), PutColumn()**

ReadBin()

Syntax short ReadBin(FRAME __PX_FAR *f, char __PX_FAR *filename);

Return Value The return values are:

Return Value	Description
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.

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 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.

See Also **WriteBin()**

ReadBMP()

Syntax short ReadBMP(FRAME __PX_FAR *f, char __PX_FAR *filename);

Return Value The return values are:

Return Value	Description
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-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.

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*.

See Also **WriteBMP()**

WriteBin()

Syntax

```
short WriteBin(FRAME __PX_FAR *f, char *filename,
short overwrite);
```

Return Value The return values are:

Return Value	Description
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.

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.

See Also **ReadBin()**

WriteBMP()

Syntax

short WriteBMP(FRAME __PX_FAR *f, char __PX_FAR *filename, short overwrite);

Return Value

The return values are:

Return Value	Description
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

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.

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*.

See Also

ReadBMP()

The VGA Video Display Library

7

The VGA Video Display library is a DOS-based VGA display and menu builder. The library makes it easy to create and display a graphics menu-based interface for a program. Imagenation used this library to create the interface for PXCVU and for most of the DOS sample programs.

This library is written in C and comes in several versions:

VIDEO_LB.LIB—Turbo, version 3.0 and later and Borland, version 3.1 and later.

VIDEO_L6.LIB—Microsoft, version 6.0.

VIDEO_LM.LIB—Microsoft, version 7.0 and later.

VIDEO_LW.LIB—Watcom 16-bit compiler version 10.6 and later.

VIDEO_FW.LIB—Watcom DOS/4GW version 10.6 and later.

The library provides functions for the following purposes:

- Entering, configuring, and exiting graphics mode
- Selecting fonts and displaying text strings
- Drawing lines and rectangles
- Creating and displaying menus

In order to use this VGA Video Display library, your video card and monitor must be VESA-compatible.

Initializing and Exiting the Library

Before you call any other VGA Video Display functions, you must call **VGALIB_OpenLibrary()**. The VGALIB_OpenLibrary() function initializes the library and sets up the interface for calling functions (for more information on function calling conventions, see *Programming in C*, on page 41.)

After making the last VGA Video Display function call and before exiting your program, you must call **VGALIB_CloseLibrary()**. VGALIB_CloseLibrary() frees any resources allocated when the library was initialized.

Entering and Exiting VGA Graphics Mode

After initializing the VGA Video Display library, but before calling any other VGA Video Display functions, you must call **AllocateVGA()**. The AllocateVGA() function saves the current display mode, sets the display to the specified graphics mode, initializes some global data structures, and returns a pointer to a frame. You can use the frame pointer returned by AllocateVGA() to operate on the VGA display with functions from both the VGA Video Display library and the Frame library. You specify the graphics mode by specifying a resolution, (dx,dy), and a pixel data type. The valid pixel data types are PBITS_Y8, PBITS_RGB15, PBITS_RGB16, PBITS_RGB24, and PBITS_RGB32. (For more information on pixel data types, see *Allocating and Freeing Frames*, on page 48.)

After making the last VGA Video Display function call, but before calling VGALIB_CloseLibrary(), you must call **FreeFrame()**. FreeFrame()

resets the display mode to the mode that was active before the call to AllocateVGA().

Displaying VGA Text and Graphics

The color for both text and graphics can be controlled using the following library functions:

SetColor()—Sets the current foreground color to the RGB values specified.

GetColor()—Returns the R, G, or B value of the currently selected color.

The basic functions this library provides for displaying text are:

DrawTextString()—Draws a string of text in the current color, beginning at a specified location (x, y).

SetFontSize()—Selects one of the three fonts: 8x8, 8x14, or 8x16.

GetFontSize()—Returns the currently selected font.

The library provides the following graphics operations:

DrawLine()—Draws a line in the current color. You specify the two endpoints of the line.

DrawRectangle()—Draws a rectangle in the current color. You specify the coordinates of the upper-left corner and the width and height.

FillRectangle()—Draws a filled rectangle in the current color. You specify the coordinates of the upper-left corner and the width and height.

The library provides the following functions for locating the current cursor position following a text or drawing operation:

WhereX()—The current horizontal position of the cursor.

WhereY()—The current vertical position of the cursor.

VGA Memory Addressing

Addressing the display memory on a VGA controller often requires swapping pages of memory. The library functions for the VGA Video Display library and the Frame library automatically handle any page swapping. This means that you can't treat the VGA frame as if it were a single, contiguous block of memory. You can't use the FrameBuffer() function to get a pointer to that memory and then operate directly on the memory using that pointer. Similarly, the AliasFrame() and FrameAddress() functions can't be used with frames allocated by AllocateVGA().

Menu Creation, Configuration, and Display

A menu is a data structure whose contents can be manipulated and displayed using the **MenuSelect()** and **MenuDisplay()** functions. All menus must be successfully initialized by the **MenuGenerate()** function before they are referenced by any other function; however, some fields in the **menu** and **menuitem** structures must be initialized by the application before MenuGenerate() is called. For more information, see *Menu Structure*, on page 143 and *MenuGenerate()*, on page 152.

The MenuSelect() function is used to change the currently highlighted menu option. Its return value indicates which (if any) menu option has been selected. This return value can be used, for example, to select which of a variety of functions should be executed.

Menu Structures and Types

Menu Structure

```
struct menu
typedef struct tagmenu
{
   short xmin, ymin, dx, dy;
   short rows, cols;
   short numitems;
   char *title;
   short highlight;
   PIX_RGB32 standardc, standardcbk;
   PIX_RGB32 highc, highcbk;
   PIX_RGB32 menuc, menucbk;
   PIX_RGB32 helpc, helpcbk;
   menuitem *data;
```

This structure defines a menu. All of these values must be initialized before MenuGenerate() is called unless otherwise specified:

xmin, ymin—Define the upper left-hand corner on the screen where the menu will be drawn.

dx, dy—Define the height and width of the menu.

rows, cols—Define the number of rows and columns in which the menu items will be organized and displayed; these values are set by the MenuGenerate() function.

numitems—Defines the number of items in the menu.

***title**—Points to the title, if any, of the menu. The title appears in the menu title bar. A menu that doesn't have a title must initialize this pointer to NULL.

highlight—Defines which of the menu items is currently selected.

standardc, standardcbk—Colors used to display all menu features except menu items and help.

highc, highcbk—Colors used to display the highlighted menu items.

menuc, menucbk—Colors used to display non-highlighted menu items.

helpc, helpcbk—Colors used to display single-line help messages for highlighted menu items at the bottom of the screen.

***data**—Points to the **menuitem** structures and is usually set to point to an array.

Menuitem Structure

```
struct menuitem
typedef struct tagmenuitem
{
   short xoff, yoff;
   short i, j;
   char *text;
   short hotkey;
   char *help;
}menuitem;
```

This structure defines a menu item. All of these values must be initialized before calling MenuGenerate() on the associated menu, unless otherwise specified:

xoff, yoff—Define the item's display coordinates relative to the menu's upper left-hand corner; these values are set by MenuGenerate().

i, j—Define the item's (row, column) coordinates in the menu display; these values are set by MenuGenerate().

***text**—Points to the text string in the menu that describes this item.

hotkey—Defines a hotkey that can be used to select this menu item. If no hotkey is desired, set this field to zero.

***help**—Defines the text string that will be displayed at the bottom of the screen when this item is selected. The string should describe the function of this menu item.

Function Reference

AllocateVGA()

Syntax

FRAME __PX_FAR *AllocateVGA(short dx, short dy,
unsigned short type);

Return Value

A pointer to a frame if successful.
NULL if unsuccessful.

Description

Puts the VGA display into the graphics mode with a resolution of *dx* x *dy* and a pixel type of *type*. Valid pixel types are PBITS_Y8, PBITS_RGB15, PBITS_RGB16, PBITS_RGB24, and PBITS_RGB32. (For more information on pixel data types, see *Allocating and Freeing Frames*, on page 48.)

If the VGA display doesn't support the requested mode, the function returns NULL.

You can use the frame pointer returned by AllocateVGA() to operate on the VGA display with functions from both the VGA Video Display library and the Frame library. This means that you can use Frame library functions, such as PutRectangle() to draw to the VGA screen.

Programs must call VGALIB_OpenLibrary() and AllocateVGA(), in that order, before calling any other VGA Video Display library function.

Note:
It is also possible to use the graphics functions from the VGA Video Display library on a frame allocated with AllocateBuffer(). In that case, you must call VGALIB_OpenLibrary(), but not AllocateVGA().

See Also **FreeFrame()**, **VGALIB_OpenLibrary()**, **ChangeResolution()**

ChangeResolution()

Syntax FRAME __PX_FAR *ChangeResolution(FRAME __PX_FAR *f, short dx, short dy, unsigned short type);

Return Value Non-zero if successful.
0 on failure.

Description Changes the VGA display to the mode with a resolution of *dx* x *dy* and a pixel type of *type*. After setting the original display mode with AllocateVGA(), you can change the display mode by calling ChangeResolution() with the frame pointer *f* returned by AllocateVGA(). If the resolution is changed successfully, the frame *f* is no longer valid; you must use the new frame returned by this function for all subsequent operations. Valid pixel types are PBITS_Y8, PBITS_RGB15, PBITS_RGB16, PBITS_RGB24, and PBITS_RGB32. (For more information on pixel data types, see *Allocating and Freeing Frames*, on page 48.)

If the VGA display doesn't support the requested mode, the function returns NULL, and the display mode is unchanged.

See Also **AllocateVGA()**

DisplayMsg()

Syntax void DisplayMsg(menu *m, FRAME __PX_FAR *f, char *msg);

Return Value None.

Description Displays the text string pointed to by *msg* at the bottom of the display.

See Also **DrawTextString()**

DrawLine()

Syntax short DrawLine(FRAME __PX_FAR *f, short x0, short y0, short x1, short y1);

Return Value The length of the line if successful.
NULL if the specified location is outside the boundaries of the screen.

Description Draws a line on the frame *f* from *(x0, y0)* to *(x1, y1)* in the current color.

See Also **SetColor()**

DrawRectangle()

Syntax short DrawRectangle(FRAME __PX_FAR *f, short x0, short y0, short dx, short dy);

Return Value Non-zero if successful.
0 on failure.

Description Draws an unfilled rectangle on the frame *f* with upper-left corner at *(x0, y0)* in the current color. The rectangle is *dx* pixels wide and *dy* pixels tall.

See Also **FillRectangle()**, **SetColor()**

DrawTextString()

Syntax short DrawTextString(FRAME __PX_FAR *f, short x0, short y0, char *string);

Return Value Non-zero if successful.
NULL if the total length of the string is outside the boundaries of the screen.

Description Draws a string of text on the frame *f* starting at location *(x0, y0)* in the current color.

See Also **SetColor()**, **SetFontSize()**

FillRectangle()

Syntax short vgalib.FillRectangle(FRAME __PX_FAR *f, short x0, short y0, short dx, short dy);

Return Value Non-zero if successful.
0 on failure.

Description Draws a filled rectangle on the frame *f* with upper-left corner at *(x0, y0)* in the current color. The rectangle is *dx* pixels wide and *dy* pixels tall.

See Also **DrawRectangle()**, **SetColor()**

FreeFrame()

Syntax void FreeFrame(FRAME __PX_FAR *f);

Return Value None.

Description Resets the display to the mode it was in just before AllocateVGA() was called. Programs must call FreeFrame() after all other VGA Video Display functions have been called, but before calling VGALIB_CloseLibrary().

See Also **AllocateVGA()**, **VGALIB_CloseLibrary()**

GetBkColor()

Syntax short GetBkColor(FRAME __PX_FAR *f, short color);

Return Value The current background color if successful.
NULL if color is not supported.

Description Returns the current value for *color* for the background, where color is one of RED, GREEN, BLUE, or ALPHA. Values can range from zero to 255.

See Also **SetBkColor()**

GetColor()

Syntax short GetColor(FRAME __PX_FAR *f, short color);

Return Value The current foreground color if successful.
NULL if color is not supported.

Description Returns the current value for *color* for the foreground, where color is one of RED, GREEN, BLUE, or ALPHA. Values can range from zero to 255.

See Also **SetColor()**

GetFontSize()

Syntax short GetFontSize(void);

Return Value The currently selected font number on success.
NULL if the specified font is not supported.

Description Returns the font number of the currently selected font. There are three fonts available: 8x8, 8x14, and 8x16, numbered 1, 2, and 3 respectively.

See Also **DrawTextString()**, **SetFontSize()**

GetKey()

Syntax short GetKey(void);

Return Value The scan code of the key hit.

Description Waits for a key to be depressed, and then returns the scan code for the key. This library has definitions for the following non-standard ASCII keys and key combinations: the arrow keys, page up, page down, insert, delete, home, end, the function keys, and CONTROL + the arrow keys. The definitions are in the file VIDEO.H. The MenuSelect() function uses some of these special keys, so it should take its input from GetKey().

See Also **MenuSelect()**

MenuCalcDx()

Syntax short MenuCalcDx(menu *m, FRAME __PX_FAR *f, short columns);

Return Value The calculated menu width.

Description Calculates the width in pixels that the menu *m* should be if its items are arranged in a number of columns equal to *columns*. This calculation is based on the width of each menu item and the width in pixels of the text (as defined by SetFontSize()).

For more information, see *Menu Structure*, on page 143, and *Menuitem Structure*, on page 144.

See Also **MenuCalcDy()**, **MenuGenerate()**, **SetFontSize()**

MenuCalcDy()

Syntax short MenuCalcDy(menu *m, FRAME __PX_FAR *f, short columns);

Return Value The calculated menu height.

Description Calculates the height in pixels that the menu *m* should be if its items are arranged in a number of columns equal to *columns*. This calculation is

based on the number of items and the height in pixels of the text (as defined by SetFontSize()).

For more information, see *Menu Structure*, on page 143, and *Menuitem Structure*, on page 144.

See Also **MenuCalcDx()**, **MenuGenerate()**, **SetFontSize()**

MenuDisplay()

Syntax short MenuDisplay(menu *m, FRAME __PX_FAR *f);

Return Value Non-zero if successful.
0 on failure.

Description Displays menu *m* on the VGA screen at the location specified by the x and y values in the menu structure. It erases the area where the menu is to be drawn, draws a rectangle to frame the menu, displays the menu options and title, displays (at the bottom of the screen) the help text for the currently-selected menu option, and highlights the currently selected menu option.

For more information, see *Menu Structure*, on page 143, and *Menuitem Structure*, on page 144.

See Also **MenuErase()**

MenuErase()

Syntax void MenuErase(menu *m, FRAME __PX_FAR *f);

Return Value None.

Description Erases the menu *m* from the VGA display by calling FillRectangle(menu->xmin, menu->ymin, menu->dx, menu->dy, colors.standardbk). It does not check, before erasing this area, to see whether the menu was actually displayed on the VGA monitor.

For more information, see *Menu Structure*, on page 143, and *Menuitem Structure*, on page 144.

See Also **MenuDisplay()**

MenuGenerate()

Syntax short MenuGenerate(menu *m, FRAME __PX_FAR *f);

Return Value Return values are:

Return Value	Description
0	Menu successfully initialized.
MENU_BOUNDS_ERR	Menu screen coordinates off screen or otherwise invalid.
MENU_WIDTH_ERR	Menu not wide enough to hold a menu item.
MENU_HEIGHT_ERR	Menu not tall enough for specified width and number of menu items.

Description Sets up some internal data in menu *m* required by the menu functions. In order for MenuGenerate() to function properly, several items in the menu structure must be initialized before MenuGenerate() is called: xmin, ymin, dx, dy, numitems, *data, and *title. (*title may be initialized to NULL if you don't want your menu to have a title, but it can’t be left uninitialized.)

The MenuGenerate() function assumes that all menu item names have the same number of characters. The function calculates the number of rows for the displayed menu based on the height of the menu and of the individual characters, and then calculates the number of columns based on the number of rows and number of items. The MenuGenerate() function will fail under the following circumstances:

- The menu coordinates are off-screen.
- With the given origin, the menu is too wide to fit on the screen.

- The menu is not wide enough, based on the width of each menu item name and the number of columns.
- The menu is not tall enough, based on the width in pixels of the menu and the number of menu items.

The return value of MenuGenerate() should always be checked for errors before menu *m* is used with any other VGA Video Display function.

For more information, see *Menu Structure*, on page 143, and *Menuitem Structure*, on page 144.

See Also **MenuCalcDx(), MenuCalcDy(), MenuDisplay()**

MenuSelect()

Syntax short MenuSelect(menu *m, FRAME __PX_FAR *f, short key);

Return Value Return values are:

Return Value	Description
-1	No selection made.
0 to m->numitems - 1	Index of selected menu item.

Description Changes the highlighted menu option depending on the key that is input, or returns the index of the highlighted menu item if the key is RETURN or a defined hotkey for that menu item. The following keys have special meaning to MenuSelect():

Left and Right Arrows—Move selection left or right by one column.

Up and Down Arrows—Move selection up or down by one row.

PAGE UP and PAGE DOWN—Move selection to top or bottom of current column.

HOME and END—Move selection to first or last menu item.

For more information, see *Menu Structure*, on page 143, and *Menuitem Structure*, on page 144.

SetBkColor()

Syntax short SetBkColor(FRAME __PX_FAR *f,
PIX_RGB32 __PX_FAR *color);

Return Value Non-zero if successful.
NULL if color is not supported.

Description Sets the current background color to the RGB values specified. For each color component, values can range from zero to 255.

See Also **GetBkColor()**

SetColor()

Syntax short SetColor(FRAME __PX_FAR *f,
PIX_RGB32 __PX_FAR *color);

Return Value Non-zero if successful.
NULL if color is not supported.

Description Sets the current foreground color to the RGB values specified. For each color component, values can range from zero to 255.

See Also **GetColor()**

SetFontSize()

Syntax short SetFontSize(short font_number);

Return Value Non-zero if successful.
NULL if the specified font is not supported.

Description Sets the font used by DrawTextString() to *font_number*. There are three fonts available: 8x8, 8x14, and 8x16, with *font_number* 1, 2, and 3 respectively. The default (set by AllocateVGA()) is the 8x16 font.

See Also **AllocateVGA(), DrawTextString()**

VGALIB_CloseLibrary()

Syntax void VGALIB_CloseLibrary(VGALIB __PX_FAR *interface);

Return Value None.

Description Releases any resources allocated by VGALIB_OpenLibrary(). Programs must call VGALIB_CloseLibrary() before exiting.

See Also **VGALIB_OpenLibrary()**

VGALIB_OpenLibrary()

Syntax short VGALIB_OpenLibrary(VGALIB __PX_FAR *interface,
short sizeof(interface));

Return Value Non-zero if successful.
0 on failure.

Description Initializes the library and fills in the *interface* structure, where *interface* is the name you will use for calling other library functions (for more information on calling conventions, see *Programming in C*, on page 41).

See Also **VGALIB_CloseLibrary()**

WhereX()

Syntax short WhereX(void);

Return Value The horizontal position of the cursor.
-1 on failure.

Description Returns the horizontal position, in pixels, of the cursor following a DrawLine(), DrawRectangle(), or DrawTextString() function call.

See Also **WhereY()**

WhereY()

Syntax short WhereY(void);

Return Value The vertical position of the cursor.
-1 on failure.

Description Returns the vertical position, in pixels, of the cursor following a DrawLine(), DrawRectangle(), or DrawTextString() function call.

See Also **WhereX()**

Cables and Connectors

A

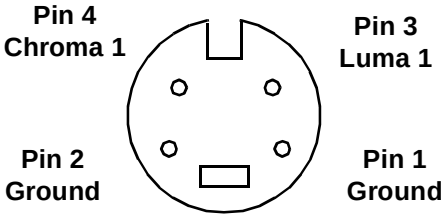
This chapter includes information on making cables for the PXC200 frame grabber.

Standard PCI and CompactPCI Cables

The versions of the PXC200 for the standard PCI bus and for the CompactPCI bus both use 26-pin D and S-video connectors. You can use commercially-available S-video cables with the S-video connector or you can make your own cables. Pinout information for both connectors follows.

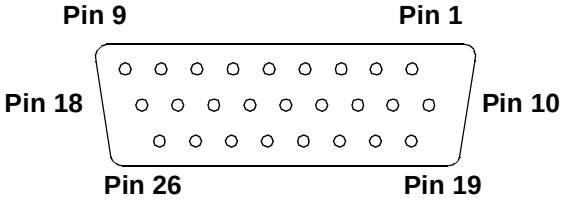
S-Video Connector

Pinouts for the S-video connector are shown below:



26-pin D Connector

Pinouts for the 26-pin D connector on the PXC200 are shown below, as viewed from the end of the board:



Pin	Description	Pin	Description
1	Y0	14	Digital Ground
2	Y1	15	Trigger 0
3	Y2	16	Trigger 1*
4	Y3	17	Trigger 2*
5	Reserved	18	Trigger 3*
6	Horizontal Sync Drive*	19	C0*
7	Vertical Sync Drive*	20	C1
8	Digital Ground	21	C2*
9	+12 V DC Out	22	C3*
10	Analog Ground 0	23	Strobe 0*
11	Analog Ground 1	24	Strobe 1*
12	Analog Ground 2	25	Strobe 2*
13	Analog Ground 3	26	Strobe 3*

* These signals are available only on versions of the PXC200 with the optional Control Package.

Connecting the +12V Output

To activate the +12V output on standard PCI bus versions of the PXC200, you must connect the board to the computer's power supply. You make this connection using the same type of connectors used to power the disk drives.

PC/104-Plus Cables

The PC/104-Plus configuration of the standard PXC200 uses a 20-pin male connector. The version of the PXC200 that includes the optional Control Package has an additional 24-pin male connector. Both connec-

tors are IDC-compatible. The pinouts for these connectors are given in the following sections.

20-Pin Connector

Connector J5 is a 20-pin IDC-compatible connector:

Pin	Description	Pin	Description
1	Ground	2	Y0
3	Ground	4	Y1
5	Ground	6	Y2
7	Ground	8	Y3
9	Ground	10	C0
11	Ground	12	C1
13	Ground	14	C2
15	Ground	16	C3
17	Ground	18	Trigger 0
19	12 V Ground	20	+12 V DC Out

For each of the even-numbered pins, the corresponding ground pin is shown on the same line of the table.

24-Pin Connector

On versions of the PXC200 with the optional Control Package, connector J10 is a 24-pin IDC-compatible connector with the following pinouts:

Pin	Description	Pin	Description
1	Ground	2	Trigger 0
3	Ground	4	Trigger 1
5	Ground	6	Trigger 2
7	Ground	8	Trigger 3
9	Ground	10	Strobe 0
11	Ground	12	Strobe 1
13	Ground	14	Strobe 2
15	Ground	16	Strobe 3
17	Ground	18	Horizontal Sync Drive
19	Ground	20	Vertical Sync Drive
21	Ground	22	Reserved
23	Ground	24	Reserved

For each of the even-numbered pins, the corresponding ground pin is shown on the same line of the table.

Hardware Specifications

B

This appendix lists specifications for the PXC200 hardware. The board is available with a standard set of features and with an optional Control Package.

Standard Features

Input video formats	NTSC, PAL, SECAM, S-video.
Input video signal	1 V peak-to-peak, 75 Ω.
Resolution	NTSC:640 x 480 pixels PAL/SECAM:768 x 576 pixels.
Sampling jitter	Maximum of ±4 ns relative to horizontal synchronization (for a stable source).
Output formats	Color: YCrCb 4:2:2; RGB 32, 24, 16, and 15. Monochrome: Y8
External trigger	Software programmable edge or level sensitivity and polarity.

Over-voltage protection	All inputs and outputs are diode protected.
Form factor	PCI short card: 174.6 x 106.7 mm 6.875 x 4.2 in. PC/104 Plus module: 91.4 x 96.5 mm 3.4 x 3.6 in. CompactPCI 3U card: 100 x 160 mm 3.94 x 6.4 in.
Video noise	≤ 1 LSB (least significant bit) RMS.
Power	+5 VDC.
Camera power	+12 VDC output.
Video multiplexer	Four video inputs, only one of which can be S-video; all four can be composite video.
Operating temperature	0° C to 60° C.
Warranty	One-year limited parts and labor.

Optional Control Package

The optional Control Package adds the following features to the standard board.

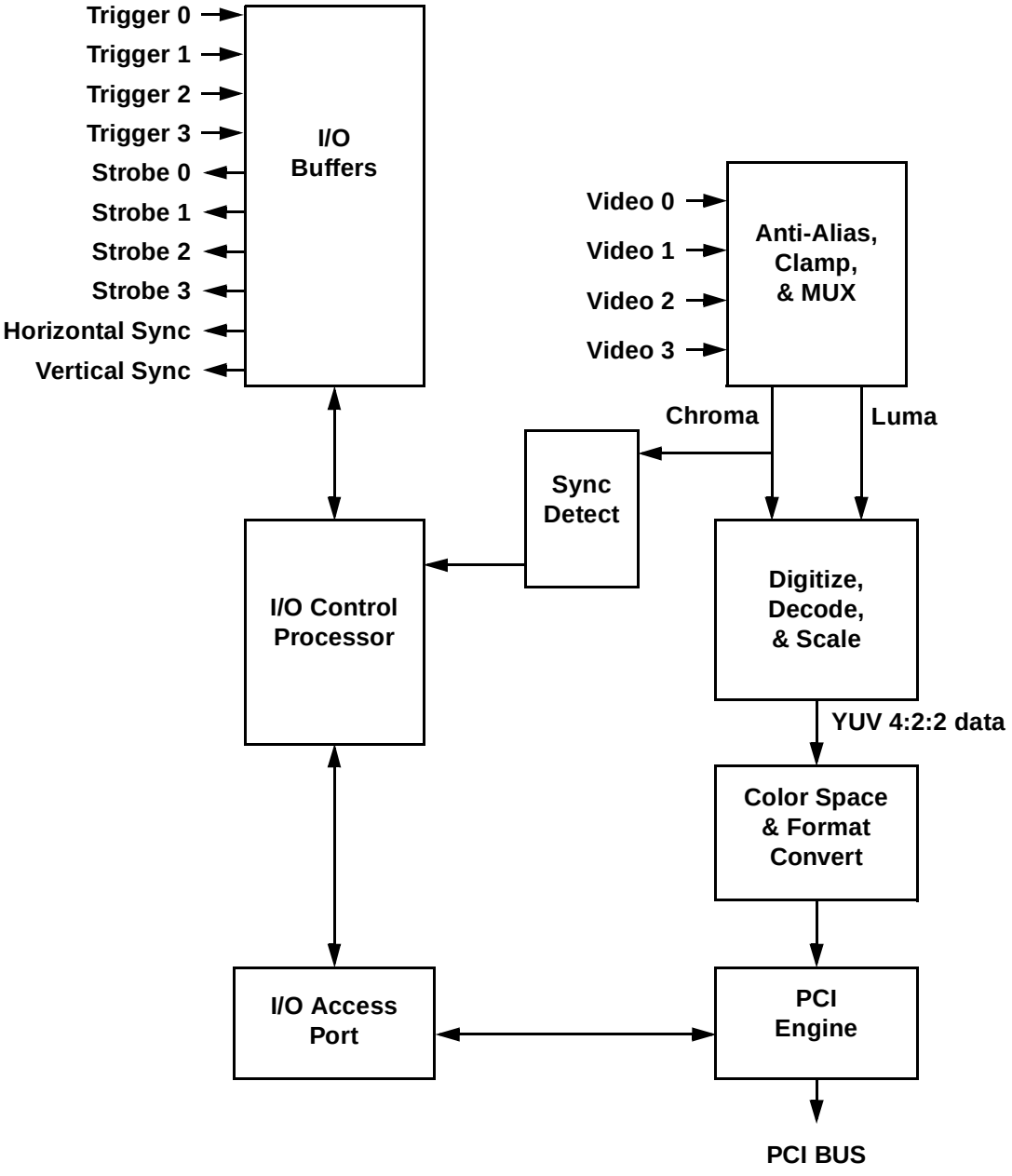
Digital I/O	Four general-purpose TTL-level input lines and four general-purpose TTL-level output lines replace the single trigger on the standard product. All lines are software programmable. Input lines are pulled up to 5 V and can compensate for trigger bounce.

Sync drive signals	Vertical and horizontal sync drive outputs. Signals are 5 V, active low.
Strobe inhibit	Output lines programmed to fire strobe pulses can be inhibited during CCD transfer time by setting a programmable holdoff period.
DC restore	All four video inputs have DC restoration.
Video multiplexer	All four video inputs can accept either composite video or S-video.

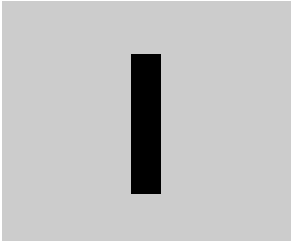
C

Block Diagram

A block diagram of the PXC200 board is shown on the following page.


Trigger 0
Trigger 1
Trigger 2
Trigger 3
Strobe 0
Strobe 1
Strobe 2
Strobe 3
Horizontal Sync
Vertical Sync
I/O Buffers
Video 0
Video 1
Video 2
Video 3
Anti-Alias, Clamp, & MUX
Chroma
Luma
Sync Detect
I/O Control Processor
Digitize, Decode, & Scale
YUV 4:2:2 data
Color Space & Format Convert
I/O Access Port
PCI Engine
PCI BUS

Index



D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W