

**Imagenation
PX Family of Precision
Frame Grabbers**

User's Guide for the PX510 and PX610

Copyright © 1995-1997, Imagination Corporation. All rights reserved.
Imagination Corporation
P.O. Box 276
Beaverton, OR 97075-0276

September 1997

P/N MN-510-02

FCC Notice

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case the user will be required to correct the interference at his own expense.

It is the responsibility of the user to use cables and equipment that comply with FCC regulations. For questions regarding cabling and equipment please contact an Imagenation customer support representative.

Changes or modifications not expressly approved by Imagenation could void the user's authority to operate the equipment.

Contents

1. Introduction	1
Precision Capture Hardware	3
Video Inputs and Formats	4
Video Capture Modes and Resolution	5
Image Capture Modes	5
Capture Resolution	6
Real-Time Image Data Transfer	6
PCI Bus Master Design	6
Selectable Destination for Image Captures	7
Processing Video Input	7
Offset and Gain Adjustments	8
Input Lookup Table (LUT)	8
Input/Output	8
Trigger	8
Strobes	9
Output Sync Signals	9
Optional Video Cache RAM	9
Programming Libraries and DLLs	9
The PCIVU Program	10

Utility Programs.....	11
FILEIT	11
PXREV.....	11
VGACOPY	11
PXCLEAR	11
Next Steps.....	12
2. Installing Your Frame Grabber	13
Do You Need a Cable?	13
Standard PCI-Bus Cables.....	13
PC/104-Plus Cables.....	13
Installing Your Board	14
Installing the Software	16
DOS, DOS/4GW, and Windows 3.1 Software Installation	16
Windows 95 Software Installation.....	21
Windows NT Software Installation.....	23
PX Software Directories	26
Troubleshooting	26
Error Loading DLL	27
Error Loading VxD	27
Problems Running PCIVU or PXREV	27
Slow Video Display Performance.....	28
Windows Hangs or Crashes on Boot	28
Technical Support.....	29
3. The PCIVU Application	31
Setting Up PCIVU	31
Starting PCIVU.....	32
Running PCIVU with More Than One Frame Grabber.....	32
Using PCIVU	33
4. Programming PX Frame Grabbers	35
General Library Characteristics	36

Operating System and Language Specifics	37
DOS Programming.....	37
Watcom DOS/4GW Programming	38
Windows 3.1 Programming	38
Windows 95 Programming	39
Windows NT Programming	40
Programming in a Multithreaded, Multitasking Environment....	41
Visual Basic Programming	42
Initializing and Exiting the Library	44
Allocating and Freeing Frame Grabbers.....	46
The PXCLEAR Utility	47
Grabbing Images	47
Frames	48
Functions for Grabbing to Frames	48
Sending Images Directly to Another PCI Device	50
Sending Images to the Onboard Video Cache RAM	51
Grabbing Images with Non-Standard Video Formats.....	52
Accessing Frame Data	54
Setting Video Offset and Gain.....	55
Video Offset.....	55
Video Gain	56
Selecting Camera Inputs	57
Input/Output.....	58
Trigger.....	58
Strobes.....	59
Synchronization Drive Signals.....	60
Using the Input Lookup Table (LUT).....	60
Getting Information about Incoming Video	61
Checking Pixel Values	61
Video Format	62
Counting Fields	62
Reading Frame Grabber Information.....	63
Video Cache RAM.....	63
Board Revision Number.....	63
Hardware Protection Key	64
Board Configuration	64

Timing the Execution of Functions	64
Queued Functions	65
Synchronizing Program Execution to Video	67
Purging the Queue.....	67
Queue Structure under Windows NT.....	67
Immediate Functions.....	68
Function Timing Summary	68
Using Flags with Function Calls.....	70
Specifying Image Capture Resolution	71
Scaling Images	71
Cropping Images	73
Grayscale Resolution	75
Frame and File Input/Output.....	75
BMP Files	75
Binary Files	75
Using the Video Display DLL	76
Developing a Menu-Based User Interface for DOS Applications...	77
Frame Grabbing and PCI Bus Performance	78
5. Function Reference	79
6. VESAMENU Library	121
Initializing and Exiting the Library	122
VGA Text and Image Display	122
Menu Creation, Configuration, and Display.....	123
Menu Structures and Types	124
Function Reference	127
VESA and VGA Functions	127
VESA Text Functions	128
Menu Functions.....	133
Graphics Functions	137
Editing Functions	138

7. The FILEIT File Conversion Program	139
Syntax	140
Examples	141
Return Values	142
Batch File Processing	144
Notes on Format Conversions	146
A. PX500 Compatibility	149
New Features in the PX510 and PX610	149
Non-Interlaced, Progressive-Scan Video Support	149
Horizontal and Vertical Cropping and Scaling	150
Horizontal and Vertical Sync Drive Signals	150
Programmable Strobe Lines	150
Full-Size (768x576) CCIR/PAL Images	150
+12V Power Line	151
26-Pin D Connector	151
Changes in the PX Libraries	151
New Strobe Functions	151
New Sync Drive Signal Functions	152
New Video Format Functions	152
Other New Functions	153
Changes to Existing Version 1.x Functions	153
B. Cables and Connectors	155
Standard PCI Bus and CompactPCI Bus Cables	155
26-pin D Connector	156
Connecting the +12V Output	157
26-Pin to 15-Pin Adapter for PX500 Cables	157
PC/104-Plus Cables	157
C. Hardware Specifications	159
D. Block Diagram	161

E. PCI Bus System Performance	163
VGACOPY Measurements	163
VGACOPY Tests	165
Configurations Tested	165
Test Results	165
Index	167

Introduction

1

The Imagination PX family of frame grabbers features precision video capture hardware for applications that require high grayscale accuracy. Features of the precision hardware design include:

- High grayscale accuracy with low pixel jitter
- Support for resettable cameras
- Accurate sync timing to the first field of the incoming data
- Image capture at 30 frames per second in NTSC mode, 25 frames per second in CCIR/PAL mode
- Horizontal and vertical cropping and scaling
- Captures of single fields or frames, or continuous captures
- Captures starting at field 0, field 1, or the next field that occurs
- Four multiplexed video inputs
- Real-time transfer of image data to main memory via Direct Memory Access (DMA), directly to the display or other PCI device, or to the optional onboard video cache RAM
- Adjustable offset and gain
- Input lookup table
- Digital TTL-level trigger input
- Two programmable TTL-level strobe lines
- Horizontal and vertical sync output
- +12V output for powering cameras or other devices

The PX family includes these frame grabbers:

PX510. All of the features listed above configured with a PCI bus for real-time performance. The PCI bus master design frees the main CPU for other processing and lets you capture images directly to main memory or other PCI devices.

PX610. All of the features of the PX510, plus support for progressive-scan cameras with non-interlaced video output.

The PX510 and PX610 are replacements for the PX500 frame grabber, and incorporate several new features. For more information, see [Appendix A, *PX500 Compatibility*](#), on page 149. Both the PX510 and PX610 are available in configurations for the standard PCI bus, for the PC/104-Plus bus, and for the CompactPCI bus. The CompactPCI configuration is a single height, 3U board with a operating voltage of 5V only. The CompactPCI and PC/104-Plus configurations are designed for embedded-systems applications.

To make it easy to tap these hardware features, the PX family 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
- VESAMENU DOS library for building a menu-based user interface
- Sample DOS and Windows source code
- PCIVU—a DOS image capture application
- FILEIT—a DOS file conversion program that supports many standard graphics file formats

This chapter will give you an introduction to these features. More detailed technical information on features is included in [Chapter 4, *Programming PX Frame Grabbers*](#), on page 35.

Precision Capture Hardware

The design of the PX video capture hardware produces high grayscale accuracy, low pixel jitter, and precise sync timing:

Grayscale noise—0.8 LSB maximum

Pixel jitter— ± 3 ns maximum

Sync Timing—syncs to the first field of the incoming signal

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

Working with Resettable Cameras

PX frame grabbers tolerate erratic video sources and work well with resettable cameras. In industrial applications, it is often desirable to abandon the normal video timing to control when exposure will occur, such as when a moving object is in position.

Resettable cameras accept a reset pulse at arbitrary times to begin a new exposure and to restart the video output. This means the video fields may be out of order, and vertical and horizontal sync may occur at any time. When the camera receives a reset pulse, it can respond by sending either a vertical sync or a window enable (WEN) signal.

By default, a PX frame grabber will vertically re-synchronize whenever it sees a vertical sync waveform in its video input or detects a WEN signal on the trigger input. The line number and the video field are both re-established with each vertical sync or WEN signal.

A PX frame grabber will horizontally re-synchronize within one horizontal period. Video inputs with a missing horizontal sync pulse will cause a line of video to be skipped and the internal line count to be off by one. VCRs and camcorders often drop horizontal sync pulses at the end of the

video field. This is of no consequence because the line count is re-established during the subsequent vertical sync prior to the next field.

Video Inputs and Formats

The PX frame grabbers handle multiple camera inputs and video formats:

Connect up to Four Cameras. Switch between camera inputs in software. On the PX510 and PX610 standard PCI-bus versions, video input 0 is provided through a BNC connector, and all four inputs are available through the 26-pin D connector. On the PC/104-Plus versions, all four video inputs are available through the 20-pin connector.

A PX frame grabber automatically synchronizes to the selected video source.

Use Either NTSC or CCIR/PAL Video Formats. PX frame grabbers support both the 60 Hz North American NTSC format and 50 Hz European CCIR/PAL format.

Capture from Progressive-Scan Cameras with Non-Interlaced Output (PX610 only). Standard cameras expose the CCD array in alternating fashion: on one cycle the odd rows are exposed; on the other, the even rows. This corresponds to the odd and even fields of the interlaced data output format of standard NTSC or CCIR/PAL video. Unfortunately, this exposure scheme can produce an image artifact called *field flicker* or *interlace flicker*, caused by an object in an image moving far enough in 1/60 second (1/50 second for CCIR/PAL video) for the object to be in two obviously different places when each field is exposed.

Progressive-scan cameras expose the entire CCD array simultaneously, thus eliminating field flicker. Some progressive-scan cameras can output the standard NTSC- or CCIR/PAL-compatible alternating-fields format. Others output the data in row-order, or non-interlaced,

fashion. The PX610 can handle both interlaced and non-interlaced signals.

Use with color cameras. PX frame grabbers include a color filter that eliminates the color burst from video signals so it doesn't appear in captured images.

Video Capture Modes and Resolution

When you capture images with a PX 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 PX 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

PX frame grabbers sample at a horizontal resolution of either 640 or 768 pixels per scan line and a vertical resolution of one pixel per scan line. On a typical display monitor with a 4 x 3 aspect ratio, the 640-pixel horizontal resolution results in approximately square pixels for images in NTSC video mode; the 768-pixel horizontal resolution results in square pixels for images in CCIR/PAL video mode.

If you don't need to work with all of the image data, you can scale the image horizontally and vertically. Scaling is accomplished by decimation (discarding pixels). You can also crop the image horizontally and vertically. 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.

Grayscale resolution is eight bits, providing 256 shades of gray.

Real-Time Image Data Transfer

The PCI bus master design of the PX frame grabbers 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 PX frame grabbers 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 video image data.

Selectable Destination for Image Captures

You can specify one of three destinations for the image capture data:

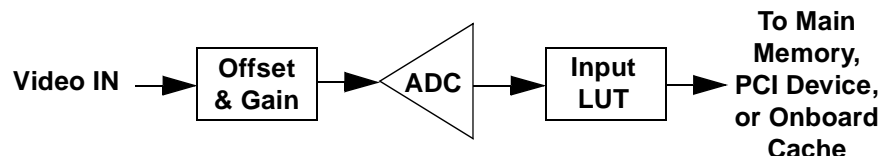
A buffer in main memory. The data is transferred via direct memory access (DMA) to a buffer in the computer's main memory. The transfer is fast, and the data is available in memory for further processing.

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.

Onboard video cache RAM. The data is placed in the optional video cache RAM on the frame grabber. Putting data directly into the onboard cache RAM doesn't use any bus bandwidth, but to process the data, you must read the data from the cache into main memory.

Processing Video Input

The PX frame grabber hardware supports offset and gain adjustments, and an input lookup table (LUT). Video signals are processed by the board as shown in this block diagram:



Offset and Gain Adjustments

The offset adjusts the D.C. video level up or down in 256 steps. This allows a bright video peak to be brought down to digital zero, or a dark video level to be boosted up to digital 255.

You can adjust the video gain from 1/2 to 8, in 1,024 steps.

Input Lookup Table (LUT)

The PX frame grabbers have an input lookup table (LUT). Video passes through the input LUT. The LUT is a 256-byte RAM that can perform a variety of grayscale pixel translations. You can use the LUT to adjust contrast, or to perform gamma corrections.

Input/Output

PX frame grabbers include I/O features that let you synchronize the frame grabber with other devices in the system.

Trigger

PX frame grabbers have an external TTL-level 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.

Mechanical switches used as the trigger input can bounce, creating spurious edges, when opening or closing. Through software, you can select a debounce delay to help avoid this problem.

Strobes

PX frame grabbers have two software-controlled TTL-level strobe lines that can be used to send signals to external hardware devices, such as resettable cameras.

Output Sync Signals

As in most frame grabbers, the video input signal is used to supply horizontal and vertical synchronization information to a PX frame grabber. In addition, PX frame grabbers can supply sync information to a camera. The frame grabber can supply this sync to the same camera that serves as the video source or to another camera, which lets you synchronize two or more cameras.

Optional Video Cache RAM

PX frame grabbers offer optional video cache RAM that can hold a single image frame of up to 768 x 576 pixels. Since you can grab images directly to the computer's RAM, you won't need a video cache on the frame grabber for many applications. However, the PCI bus can be overloaded if several PCI devices, such as frame grabbers or VGA cards, all try to transfer large blocks of data at the same time. If your application tends to overload the bus, consider adding video cache RAM to the board.

Programming Libraries and DLLs

For custom applications, the PX 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 either Borland or Microsoft C compilers, or write 32-bit DOS programs using the Watcom DOS/4GW library.

DLLs for Windows. Write 16-bit or 32-bit Windows programs for Windows 3.1, Windows 95, and Windows NT with C compilers from Borland and Microsoft, or with Visual Basic. The PX 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.

VESAMENU Library for DOS. Use the VESAMENU 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 PX Frame Grabbers](#), on page 35, describes the main features of the PX hardware and software and how to use them to build applications. For reference information on all PX library functions, see *Chapter 5, [Function Reference](#)*, on page 79. The VESAMENU library and its functions are described in *Chapter 6, [VESAMENU Library](#)*, on page 121.

The PCIVU Program

The PX software includes a DOS frame grabber application called PCIVU. Using PCIVU, you can capture images, save images to disk, and adjust many of the image capture features of a PX frame grabber—all without writing a single line of code. For more information, see *Chapter 3, [The PCIVU Application](#)*, on page 31.

Utility Programs

The PX software also includes several utility programs.

FILEIT

If you save images in binary files, you can use the FILEIT program to convert the files to standard graphics file formats, such as TIFF, BMP, and GIF. The FILEIT program is described in [Chapter 7, *The FILEIT File Conversion Program*](#), on page 139.

PXREV

If you need to contact Imagenation Technical Support, you'll be asked for your board's revision number. PXREV 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 PXGDI1, PXGDI2, and PXGDI3.

PXCLEAR

The PXCLEAR utility for Windows 3.1 and Windows 95 frees frame grabbers when a program terminates unexpectedly and does not call the required exit procedures. PXCLEAR tells you which frame grabbers are currently in use, and gives you the option of freeing all of them. It cannot be used to free individual frame grabbers; it frees all frame grabbers in

the system or none of them. For more information, see *The PXCLEAR Utility*, on page 47.

Next Steps...

For...	See...
Installing your PX frame grabber	Chapter 2, <i>Installing Your Frame Grabber</i> , on page 13
More in-depth information on PX features	Chapter 4, <i>Programming PX Frame Grabbers</i> , on page 35
Writing your own frame grabber applications	Chapter 4, <i>Programming PX Frame Grabbers</i> , on page 35 Chapter 5, <i>Function Reference</i> , on page 79 Chapter 6, <i>VESAMENU Library</i> , on page 121
Connector and cabling specifications	Appendix B, <i>Cables and Connectors</i> , on page 155
A PX board block diagram	Appendix D, <i>Block Diagram</i> , on page 161

Installing Your Frame Grabber

2

Do You Need a Cable?

Standard PCI-Bus Cables

The two BNC connectors on the standard PCI-bus configurations of the PX510 and PX610 boards let you attach one video source and an optional trigger. Additional video sources (you can connect a total of four), two strobe outputs, two synchronization drive signal outputs, and a +12V power source are also available by using the 26-pin D connector. To use the 26-pin connector, you'll need a cable with the correct mating connector and pinouts. For information on making or purchasing cables, see [Appendix B, *Cables and Connectors*](#), on page 155.

PC/104-Plus Cables

You'll need a cable to attach to the 20-pin IDC connector on frame grabbers with the PC/104-Plus configuration. For information on making cables, see [Appendix B, *Cables and Connectors*](#), on page 155.

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 PX board. Before you remove the board from its antistatic pouch, ground yourself by touching the computer's metal back panel.

- 2 If you'll be using the frame grabber with a color camera, you must enable the color filter so that the color burst doesn't appear in captured images. To enable the color filter, set the color filter switch, SW7, on the board to **OFF** (setting the switch to **ON** *disables* the filter).
- 3 Install the PX board as follows:

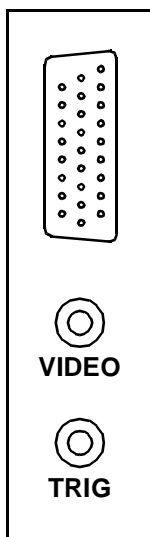
For a standard PCI-bus board:

- 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 PX board into the slot and seat it firmly.
- d Secure the board's cover plate using the screw you saved.

For a PC/104-Plus board:

- a** Set the four-position rotary switch on the PX board to an unused number. Each PC/104-Plus plug-in module must be set to a unique number.
- b** Insert the PX board into the connector and seat it firmly.

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



For a standard PCI-bus board:

BNC connectors. Connect your video source to the connector closest to the 26-pin D connector (see diagram at left). If you're using an external trigger, connect it to the BNC connector farthest from the 26-pin D connector.

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 B, Cables and Connectors](#), on page 155.

For a PC/104-Plus board:

Attach your cable to the 20-pin IDC connector on the PX board. For information on making cables, see [Appendix B, Cables and Connectors](#), on page 155.

- 5** Replace the cover on the computer, plug it in, and turn on the power.
- 6 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** PX 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 *PX500 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.
- 7** That completes the hardware installation. Next, you'll install the PX software.

Installing the Software

The PX family of 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, 0xD200-0xD2FF or 0xEA00-0xEAFF.

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 This step applies only to Watcom DOS/4GW;** if you're not using DOS/4GW, skip to the next step. The PX library for Watcom DOS/4GW requires the special memory manager FLATMEM.COM. If you want to use the Watcom PX library, you must add the following line to your CONFIG.SYS file **before** the lines for HIMEM.SYS and EMM386.EXE:

```
device=c:\px5\bin\flatmem.com 1024
```

The *1024* parameter specifies the amount of memory in KB reserved for frame allocation. You can allocate as much memory as you want, up to 1 MB less than the total amount of RAM in your system, but

keep in mind that the memory you allocate can't be used for any other purpose than storing frames.

Note

This method of reserving memory is only guaranteed to work for systems with no more than 64 MB of RAM. If you have more than 64 MB, contact Imagenation technical support for assistance.

- 3 Insert the **DOS/Windows 3.1** installation diskette in the floppy drive.
- 4 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 PCIVU program to verify that your frame grabber is correctly installed. For instructions on running PCIVU, see [Chapter 3, The PCIVU Application](#), on page 31. If an error message appears when you try to start PCIVU, see *Troubleshooting*, on page 26.

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 *PX*.

- d After restarting Windows, you can run one of the PXGDI sample programs to verify that your frame grabber is correctly installed. The sample programs are in the `c:\px5\bin` directory. If you have problems running the sample programs, see *Troubleshooting*, on page 26.

Changes to System Files for DOS 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 and Windows 3.1

```
REM Imagenation's Modifications
set path=c:\px5\bin;%path%
set imagenation=c:\px5
REM Imagenation's Modifications End
```

Adding `c:\px5\bin` to your PATH makes the samples and utilities easier to execute. The IMAGENATION environment variable specifies the location of files required by the PCIVU application. PCIVU 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:\px5\bin\px500.vxd
px500_size=256
; Imagenation's Modifications End
```

The PX Windows Virtual Device Driver (VxD), *PX500.VXD*, is added to the [386Enh] section. The VxD will be loaded only when you start Windows. The PX DLL, *WPX5.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.

The memory allocation variable, *px500_size*, is set to the amount of memory you specified during Setup. This variable specifies how much physical memory, in 4 KB blocks, the VxD should reserve for frame buffers.

Caution

If you allocate more memory for frame buffer storage than Windows 3.1 can spare, Windows 3.1 will abort during boot, returning to DOS with the message "PX???.VXD cannot allocate the requested amount of memory." If this happens, you must exit Windows and reduce the frame buffer allocation by editing the SYSTEM.INI file. You should leave at least 4 MB available to Windows; for example, if you have 8 MB of RAM in your computer, don't allocate more than 4 MB to frame buffers.

Changes to System Files for Watcom DOS/4GW

The installation programs don't make any changes to your CONFIG.SYS or AUTOEXEC.BAT files for Watcom DOS/4GW. If you want to use the Watcom DOS/4GW environment, you must make the changes listed below yourself.

CONFIG.SYS Changes for Watcom DOS/4GW

The PX library for Watcom DOS/4GW requires the special memory manager FLATMEM.COM. If you want to use the Watcom PX library, you must make a change to your CONFIG.SYS file to load the memory manager. For the specific change needed, see [Step 2](#) on page 17.

AUTOEXEC.BAT Changes for Watcom DOS/4GW

If you want to run the sample programs for Watcom DOS/4GW, the c:\px5\bin directory must be in your path. You can add the directory to your path by adding the following line to the end of your AUTOEXEC.BAT file:

```
set path=c:\px5\bin;%path%
```

Windows 95 Software Installation

- 1 If you previously installed the Windows 3.1 PX driver, you must edit the [386Enh] section of the SYSTEM.INI file to remove the lines that load the 16-bit VxD, *PX500.VXD*. For more information, see *SYSTEM.INI Changes for Windows 3.1*, on page 20.
- 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 *PX*.

When you have completed 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 PXGDI sample programs to verify that your frame grabber is correctly installed. The sample programs are in the c:\px5\bin directory. If you have problems running the sample programs, see *Troubleshooting*, on page 26.

SYSTEM.INI Changes for Windows 95

Frame buffer storage for 16-bit programs is allocated by adding a line to the [386Enh] section of the SYSTEM.INI file:

```
[ 386Enh ]  
px500_size=256
```

The memory allocation variable, *px500_size*, is set to the amount of physical memory, in 4 KB blocks, the VxD should reserve for frame buffers.

Frame buffer storage for 32-bit programs is allocated by using the Windows 95 Registry, as described in the following section.

Windows 95 Registry Changes

If you need to uninstall the PX driver, you must edit the Windows 95 Registry by using the REGEDIT.EXE program in your Windows 95 directory. If you need to change the amount of memory that is available for 32-bit PX programs, you can either edit the Windows 95 registry or rerun the installation program.

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

HKEY_LOCAL_MACHINE\System\Services\VxD\PX5_95

The values assigned to this key are:

StaticVxD. A string key that contains the complete path of the VxD file, such as *c:\px5\bin\px5_95.vxd*.

memory_size. A 32-bit value that specifies the number of bytes of contiguous physical storage to reserve for frame buffers. For example, 0x00200000 reserves 2 MB of storage. If this value is not present or is zero, no memory is allocated for 32-bit frames, and the *WPX5_95.DLL* will refuse to load.

Caution

If you allocate more memory for frame buffer storage than Windows 95 can spare, Windows 95 may hang on initialization. If this happens, you must reboot Windows 95 in safe mode, and then edit the Windows 95 Registry or rerun the installation program. You should leave at least 4 MB of RAM available to Windows 95. For example, if you have a system with 16 MB of RAM, don't allocate more than 12 MB to frame buffers.

Windows NT Software Installation

- 1 Put the **Windows NT** installation disk in the floppy drive.
- 2 Follow the appropriate instructions for Windows NT version 3.51 or version 4.0:

Version 3.51

- a From the Program Manager, 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 NT.

Version 4.0

- a** Click the Start button and click Run.
- b** For the name of the program, type **a:\setup** and click OK.
- c** When the Setup program has completed, restart Windows NT.

Setup creates a new program group called *PX*.

The installation program places a driver file, *PX500.SYS*, in your Windows NT *system32\drivers* directory. All other files are placed in the directory that you specify during the installation.

When the software installation is complete, you can use the PCIVU program to verify that your frame grabber is correctly installed. For instructions on running PCIVU, see [Chapter 3, *The PCIVU Application*](#), on page 31. If an error message appears when you try to start PCIVU, see [Troubleshooting](#), on page 26.

Windows NT Registry Changes

If you need to uninstall the PX500.SYS driver, you must edit the Windows NT Registry by using the Registry Editor (REGEDIT.EXE) program in your Windows NT directory. If you need to change the amount of memory that is available for frames, you can either edit the Windows NT registry or rerun the installation program.

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

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\  
Services\PX500
```

This key should not be modified, but if you need to uninstall the driver you must remove it. To disable the driver without having to edit the registry, remove PX500.SYS from your Windows NT system32\drivers directory; the next time you boot Windows NT, NT will report that it failed to load a driver on initialization.

The installation program also modifies another key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\  
Session Manager\Memory Management
```

The value **NonPagedPoolSize** specifies how much memory is available to NT for various kernel-level processes (including the PX500.SYS driver) that require non-paged memory. In order for the driver to be able to allocate image buffers, the size of this pool must be increased. The minimum size for NonPagedPoolSize to enable the driver to allocate one 640x480 buffer is about 2 MB (0x00200000 bytes). You can reserve more memory for frame buffers based on the size and number of buffers that you need to allocate; however, you should leave at least 8 MB available to Windows NT. For example, if you have a system with 32 MB of RAM, don't increase NonPagedPoolSize to more than 24 MB.

Caution

If you allocate more memory for frame buffer storage than Windows NT can spare, Windows NT will crash when it tries to boot. If this happens, you must reboot Windows NT using the last-known good configuration, and then edit the Windows NT Registry or rerun the installation program.

PX Software Directories

The installation programs create the LIB, INCLUDE, and BIN directories, and directories for the appropriate operating systems:

Directory	Contents
c:\px5\lib	Libraries
c:\px5\include	Header files
c:\px5\bin	Executable programs, drivers, and DLLs
c:\px5\dos	DOS and Watcom DOS/4GW sample source code.
c:\px5\win31	Windows 3.1 source code.
c:\px5\win95	Windows 95 source code.
c:\px5\winnt	Windows NT 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 VxD). The sample programs are PXGDI1, PXGDI2, and PXGDI3.

Troubleshooting

This section contains troubleshooting information for the following:

- Error loading DLLs
- Error loading VxDs
- Running PCIVU or PXREV
- Slow video display performance
- Windows hangs or crashes on reboot

Error Loading DLL

The system can't locate the PX DLL. Either edit your PATH environment variable to include the path to the PX DLL (see *PX Software Directories*, on page 26) or move the DLL to the \WINDOWS\SYSTEM directory.

Error Loading VxD

When booting Windows 3.1, you might see the error "PX500.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 PX500.VXD that came with your PX510 or PX610. If you're using an older version, upgrade to the latest version. If you still get this error message with the latest version of PX500.VXD, you'll need to upgrade your BIOS; contact the manufacturer of your system for an upgrade.

Problems Running PCIVU or PXREV

PCIVU and PXREV are DOS programs. You can't run these programs in a DOS window in Windows. If your system hangs when you run PCIVU 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 29).

Make sure that you are excluding a 4 KB block of upper memory in your CONFIG.SYS file (see [Step 1](#) on page 16 of the installation instructions).

If you see the message *This graphics card is not VESA compatible* when you run PCIVU, 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.

If you see only a few lines of video at the top of the picture in PCIVU, 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 PX frame grabbers. Run the VGACOPY program to check for errors on the PCI bus.

If you haven't set the IMAGENATION environment variable, PCIVU will display an error and won't run. For information on the IMAGENATION environment variable, see *AUTOEXEC.BAT Changes for DOS and Windows 3.1*, on page 19.

Slow Video Display Performance

When you're displaying video on the screen, the amount of memory on the VGA display controller card affects the performance. Generally, adding memory to your display controller will improve the performance.

Windows Hangs or Crashes on Boot

This can be caused by an interrupt conflict or by trying to allocate too much memory for frame buffers. Try decreasing the amount of memory you're allocating for frame buffers. You can change this allocation as follows:

Windows 3.1—Edit the SYSTEM.INI file and change the value for *px500_size* to the amount of physical memory, in 4 KB blocks, the VxD should reserve for frame buffers. Then, restart Windows.

Windows 95—Restart Windows 95 in safe mode. Then, either run the Setup program again, specifying a smaller value for the frame buffer allocation, or edit the registry to change the *memory_size* value (see *Windows 95 Registry Changes*, on page 22).

Windows NT—When Windows NT reboots, you'll see the message "Press spacebar NOW to invoke Hardware Profile/Last Known Good menu." Press the spacebar and pick the most recent configuration. This should back out the change to the Windows NT Registry. Then run the Setup program again and specify a smaller value for the frame buffer allocation.

If decreasing the amount of memory allocated for frame buffers doesn't help, 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.

Technical Support

Imagination offers free technical support to customers. If the PX 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 PXREV program in DOS or any of the PXGDI programs in Windows. You must run the PXREV program from DOS, not from a DOS window in Windows.
- The operating system you're running: DOS, DOS/4GW, Windows 3.1, Windows 95 (16-bit or 32-bit), 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).

Imagination

Voice: 503-641-7408

Toll free: 800-366-9131

Fax: 503-643-2458

CompuServe: 75211,2640

Internet:

support@Imagination.com

www.imagination.com

The 24-hour BBS and the Imagination World Wide Web site (www.imagination.com) always have the latest versions of the Imagination software. Check anytime for software updates.

The PCIVU Application

3

This chapter describes the PCIVU application program for DOS. PCIVU is a basic frame grabber application that lets you control the features of your PX frame grabber without writing your own application program. You can use PCIVU to capture frames or fields, write frames to disk files, set the gain and offset, load the input LUT, and change the video source.

Setting Up PCIVU

To run PCIVU, you must have the IMAGENATION environment variable set to point to the directory containing PCIVU.HLP and PCIVU.INI. PCIVU.HLP contains the text of the help screens you can access from PCIVU. PCIVU.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 PCIVU. If not, see *AUTOEXEC.BAT Changes for DOS and Windows 3.1*, on page 19, for the required settings.

Starting PCIVU

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

To run PCIVU, execute the following at the DOS command line (do **not** run PCIVU in a DOS window in Windows 3.1):

```
c:\px5\bin\pcivu
```

If you see a display like that shown on page 33, the PCIVU program has started correctly. Otherwise, see *Troubleshooting*, on page 26.

Running PCIVU with More Than One Frame Grabber

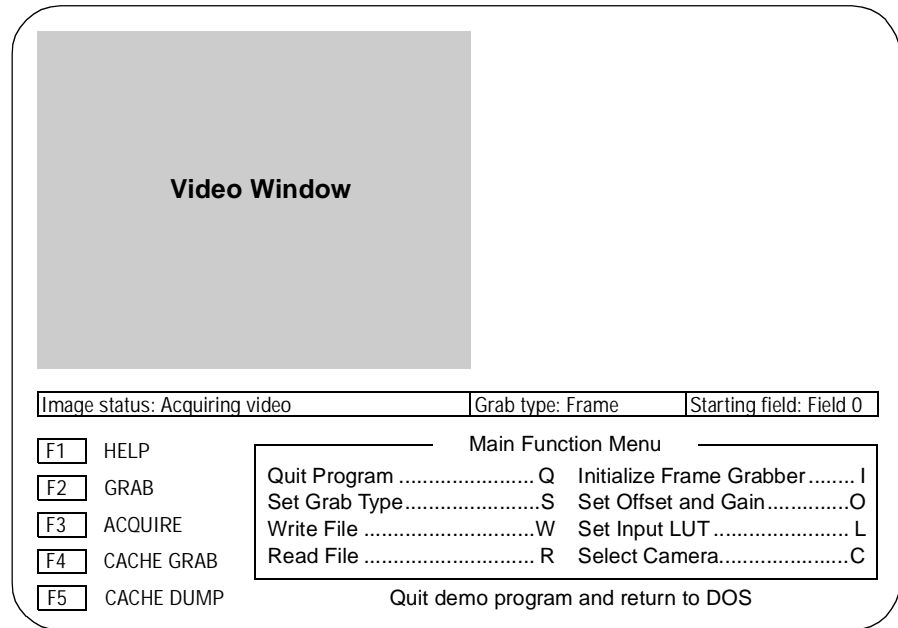
If you have more than one frame grabber installed in your system, PCIVU will use the first frame grabber that it finds. To specify a specific frame grabber, follow the command with the number of the frame grabber:

```
c:\px5\bin\pcivu 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 PCIVU application with different values for n and observe the video displayed to identify the source.

Using PCIVU

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



If you have an active video source when you start PCIVU, 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 CACHE GRAB.** Press F4 to capture a frame to the cache.
- **F5 CACHE DUMP.** Press F5 to dump the contents of the cache to system memory.

The keys F4 and F5 appear only if your PX board has the optional cache.

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 PX Frame Grabbers*](#), on page 35.

4

Programming PX Frame Grabbers

This chapter describes key features of the PX hardware and software. You'll find this information useful for using the board with the PCIVU application, and essential for developing your own PX applications. The features described include:

- General library characteristics
- Operating system and language specifics
- Initializing and exiting the library
- Allocating and freeing frame grabbers
- Grabbing images
- Setting video gain and offset
- Selecting camera inputs
- Digital input/output
- Using the input lookup table (LUT)
- Getting information on incoming video
- Reading frame grabber information
- Timing the execution of functions
- Using flags with function calls
- Specifying image capture resolution
- Frame and file input/output
- Using the Video Display DLL
- Developing a menu-based user interface for DOS applications
- Frame grabbing and PCI bus performance

General Library Characteristics

The PX frame grabber library has the following general characteristics:

- **Functions are interrupt-driven.** An interrupt handler must be installed before a process can use any of the library functions, and must be uninstalled after the library functions have been called and before the process is terminated. For more information, see *Initializing and Exiting the Library*, on page 44.
- **Frame grabbers are controlled through handles.** Software communicates with a frame grabber using a handle to the frame grabber. Only one handle to a given frame grabber may exist at one time. For more information, see *Allocating and Freeing Frame Grabbers*, on page 46.
- **Images are stored in frames.** Images are stored in data structures called frames. (This use of the term “frame” should not be confused with the video term “frame”, which refers to a video image consisting of two fields.) A frame holds an image and some basic information about it. For more information, see *Frames*, on page 48.
- **Function timing can be controlled.** Some functions can be declared to be queued, immediate, both, or neither. Queued functions return as soon as the function data has been entered into the queue and execute concurrently with subsequent non-queued functions. Immediate functions fail if they can’t execute without delay. For more information, see *Timing the Execution of Functions*, on page 64.
- **Flags control common features.** The behavior of some functions is specified by a collection of flags, which are ORed together into a single function parameter. These flags control whether the function operates on video fields or video frames, which field digitization is to begin on, and whether or not the function is either queued or immediate. For more information, see *Using Flags with Function Calls*, on page 70.

Operating System and Language Specifics

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

You can put `c:\px5\lib` and `c:\px5\include` in your environment variables for Microsoft, or in your `TURBOC.CFG` file for Borland, or in your integrated development environment (IDE) search list.

Note

*All variables declared as **int** are 16 bits long in DOS and Windows 3.1 and 32 bits long in DOS/4GW, Windows_95, and Windows_NT. All pointers in the 16-bit operating systems DOS and Windows 3.1 must be **huge**.*

DOS Programming

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

Header File	Library	Runtime, Memory, and Installation Requirements
PX5.H	Borland: PX5_LB.LIB Microsoft 7+: PX5_LM.LIB Microsoft 6-: PX5_L6.LIB	For required changes to AUTOEXEC.BAT, see <i>Changes to System Files for DOS and Windows 3.1</i> , on page 19.

Watcom DOS/4GW Programming

The following table summarizes operating system specifics for compiling, linking, and running C programs under Watcom DOS/4GW:

Header File	Library	Runtime, Memory, and Installation Requirements
PX5.H	PX5_FW.LIB	FLATMEM.COM needed for runtime. For required changes to system files, see <i>Changes to System Files for Watcom DOS/4GW</i> , on page 20.

Windows 3.1 Programming

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

Header File	Library	Runtime, Memory, and Installation Requirements
WPX5.H	WPX5.LIB	PX500.VXD and WPX5.DLL needed for runtime. For memory requirements and VxD installation, see <i>DOS, DOS/4GW, and Windows 3.1 Software Installation</i> , on page 16.

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 File	Library	Runtime, Memory, and Installation Requirements
WPX5.H	WPX5.LIB	PX5_95.VXD and WPX5.DLL needed for runtime. For memory requirements and VxD installation, see <i>SYSTEM.INI Changes for Windows 95</i> , on page 22 and <i>Windows 95 Registry Changes</i> , on page 22.

Windows 95 32-bit programs

Header File	Library	Runtime, Memory, and Installation Requirements
WPX5_95.H	Borland v5.0*: WPX5_95B.LIB All other: WPX5_95.LIB	PX5_95.VXD and WPX5_95.DLL needed for runtime. For memory requirements and VxD installation, see <i>Windows 95 Registry Changes</i> , on page 22.

* For Borland, set the 32-bit linker option for *Allow Import By Ordinal*. Versions of Borland prior to 5.0 are not supported.

Read the discussion in *Programming in a Multithreaded, Multitasking Environment*, on page 41, which applies to both Windows 95 and Windows NT.

The Windows 95 VxD, *PX5_95.VXD*, is compatible with 16-bit programs written for Windows 3.1 and with 32-bit programs written for Windows 95.

Note

You must use the Windows 95 VxD (PX5_95.VXD) in Windows 95 for both 16-bit and 32-bit programs.

Any PX DLLs that 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:

Header File	Library	Runtime, Memory, and Installation Requirements
WPX5_NT.H	Borland v5.0*: WPX5_NTB.LIB All other: WPX5_NT.LIB	PX500.SYS and WPX5_NT.DLL needed for runtime. For memory requirements and driver installation, see <i>Windows NT Registry Changes</i> , on page 24.

* For Borland, set the 32-bit linker option for *Allow Import By Ordinal*. Versions of Borland prior to 5.0 are not supported.

Read the discussion in *Programming in a Multithreaded, Multitasking Environment*, on page 41, which applies to both Windows 95 and Windows NT.

The driver structure for Windows NT differs from that of Windows 3.1 and Windows 95, so you must build separate versions of your programs for use in each environment.

DLL Interface Differences in Windows NT

While the DLL interface for Windows NT is almost identical to that of Windows 3.1 and Windows 95, there are a few differences:

AllocateAddress() and **FrameAddress()** do nothing and return zero in Windows NT. Windows NT doesn't allow direct access to the hardware in this fashion, for reasons of security and stability.

SetCurrentWindow() does nothing and returns zero in Windows NT; in its place, use the Windows NT-specific function **WaitFinished()**.

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 (!IsFinished(fgh,qh))  
    ;
```

In Windows 95 or Windows NT, such an empty loop is much less efficient than this alternate version:

```
while (!IsFinished(fgh,qh))  
    WaitVB(fgh);
```

In the Windows NT environment, you can replace the **while** loop above with the function **WaitFinished(qh)**. **WaitFinished(qh)** is equivalent and is somewhat more efficient.

The **WaitVB()** and **WaitFinished()** functions use 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. In general, polling loops should be written to use the Windows message system or have delays like the one above added where appropriate.

Scheduling multiple threads to handle complicated image processing tasks might make programming significantly easier, and the PX 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 such as `SetImageSize()` 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.

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 `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 Files. You must include the appropriate .BAS file in all projects you build using the PX DLL functions. The .BAS files include all the declarations you'll need to work with the DLLs. Use WPX5VB.BAS for all Windows 3.1 programs and for 16-bit programs under Windows 95. Use WPX_95.BAS for 32-bit Windows 95 programs and WPX_NT.BAS for Windows NT programs.

Buffers: Strings vs. Integers in Visual Basic 3.0

A Visual Basic 3.0 application can pass buffers to functions as a string by value (**ByVal buf As String**) or as an integer array by reference (**buf As Integer**). If you pass a buffer as a string, it's easy to put values into the buffer or take them out. To insert an element into a string, use the Chr\$() function on that element, and insert the result in the string with the Mid\$() function. The disadvantage to this method is that Visual Basic string manipulation is fairly slow.

If you pass a buffer as the first element of an integer array, you must pack two pixel values into each integer as you insert the values into the array, and unpack them when you remove elements from the array. This is faster, but somewhat more complicated.

The interfaces of the following functions have been defined in WPX5VB.BAS using strings.

GetColumn()	PutColumn()
GetRectangle()	PutRectangle()
GetRow()	PutRow()

If you want to change the interface, you should edit the WPX5VB.BAS file and replace occurrences of **ByVal buf As String** with **buf As Integer**.

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 WPX5_95.BAS and WPX5_NT.BAS files use **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 must use the **PXCLEAR** utility to free any frame grabbers that your program allocated.*

Displaying Video in Visual Basic Applications

The PX 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 76.

Initializing and Exiting the Library

Before calling any other library functions, you must explicitly initialize the library by calling **InitLibrary()**. Following your last call to the library, before your program terminates, you must call **ExitLibrary()**.

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

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, and the kernel driver in Windows NT. The low-level device driver is loaded when you start Windows, and is uninstalled when you exit Windows.

Check the return value from `InitLibrary()` to make sure the function was successful (non-zero = success). `InitLibrary()` 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 `InitLibrary()` will try to find such a block. For more information, see, *DOS, DOS/4GW, and Windows 3.1 Software Installation*, [Step 1](#) on page 16.
- There is insufficient conventional memory. `InitLibrary()` allocates a small amount of storage for internal data structures.
- There are no Imagenation frame grabbers in your computer, or they are malfunctioning.

Some of these error conditions can be detected by calling the `CheckError()` function (see *CheckError*, on page 83).

Allocating and Freeing 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.

Any process with a valid frame grabber handle can communicate with that frame grabber. One process can get a handle to the frame grabber using `AllocateFG()`, and other processes can use the same handle. Keep in mind that any process can change the state of the frame grabber, so a given process can't assume the state of the frame grabber will remain as that process last left it. When more than one process has simultaneous access to the same frame grabber, you must coordinate the processes accordingly.

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 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 `PCIVU` 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 Windows 3.1 and Windows 95, you can use the `PXCLEAR` program (described below) to

free all frame grabbers. For other operating systems, you might need to reboot your system.

The PXCLEAR Utility

PXCLEAR is a utility that frees frame grabbers. It is available for both Windows 3.1 and Windows 95. If a program terminates unexpectedly and does not call its exit procedures, any frame grabbers that it had allocated will still be allocated, preventing any other programs from using them. PXCLEAR tells you which frame grabbers are currently in use, and gives you the option of freeing all of them. It can't be used to free individual frame grabbers; it frees all frame grabbers in the system or none of them.

You should not use PXCLEAR as a general tool for freeing frame grabbers in preference to freeing them in your program's exit procedures. You also should not use PXCLEAR while any program that is using a frame grabber is still running.

Note

*The Visual Basic development environment **End** button terminates a running program immediately, without executing the `Form_Unload` function (or any other). If you use the **End** button to exit a program, you must use the PXCLEAR utility to free any frame grabbers that your program allocated.*

Grabbing Images

The library functions for grabbing images let you specify how you want to initiate the capture and where you want the frame grabber to send the captured image data.

Frames

Library functions can send the grabbed image data to *frames* or to the onboard video cache RAM, if it exists. 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 in two ways: with `AllocateBuffer()` or with `AllocateAddress()`. **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 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. You might need to increase the amount of memory you're allocating for frame buffers. This memory allocation is set at the time you install the PX software. For information on changing the memory allocation, see the appropriate section for your operating system in [Chapter 2, Installing Your Frame Grabber](#), on page 13.

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()`.

Functions for Grabbing to Frames

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

Grab() and **GrabTriggered()** both digitize video and copy the data to the specified frame. For both functions, you can specify which video field they should start on, whether to digitize one field or both, and when to execute (see *Timing the Execution of Functions*, on page 64). By using the CACHE flag, you can make **Grab()** and **GrabTriggered()** copy the data to the cache at the same time the data is copied to the specified frame.

Grab() starts digitizing as soon as the command is processed by the frame grabber, while **GrabTriggered()** does not start digitizing until the first trigger pulse received after the command is sent to the frame grabber. For more information on using the trigger, see *Trigger*, on page 58.

GrabContinuous() continuously digitizes and transfers video to the specified frame. By using the CACHE flag, you can make **GrabContinuous()** copy the data to the cache at the same time the data is copied to the specified frame.

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

When you want to access the data for image processing, use **FrameBuffer()** to get a logical address (a pointer) to the data.

The most common reasons the grab functions fail are:

- The frame grabber handle or the frame buffer handle is invalid.
- The image specified by **SetImageSize()** (or the default image size) is too large in width or height for the frame buffer.
- For triggered grabs, an incorrect trigger type is specified or the **GrabTriggered()** function is called with the `DEBOUNCE` flag, but without specifying a trigger type.

If the captured image is all black, 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 the PX frame grabbers. Run the VGACOPY program to check for errors on the PCI bus.

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.

Note

AllocateAddress() returns zero and has no effect in Windows NT programs; Windows NT doesn't allow applications access to the hardware in this fashion for reasons of security and stability.

Use the **Grab()**, **GrabTriggered()**, and **GrabContinuous()** functions to capture images to frames allocated with `AllocateAddress()`.

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. Also, 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.

Sending Images to the Onboard Video Cache RAM

In systems with two or more PCI devices, it's possible to overload the PCI bus, resulting in lost or corrupted data. For these situations, you can purchase PX frame grabbers with either 512 or 1,024 lines of onboard video cache RAM. Grabbing images to the video cache doesn't use the PCI bus. For example, using the video cache, you could capture images simultaneously with two or more frame grabbers and be sure of not losing any data.

When grabbing to the video cache, you use the **GrabToCache()** and **CacheTriggered()** functions. **GrabToCache()** and **CacheTriggered()** are identical to **Grab()** and **GrabTriggered()**, except that the image data is transferred to the optional onboard video cache RAM instead of to a frame in main memory (see *Video Cache RAM*, on page 63). By using the **CACHE** flag, you can also make **Grab()**, **GrabTriggered()**, and **GrabContinuous()** copy the data to the cache at the same time the data is copied to the specified frame.

You can't directly access the image data in the video cache. Use the **ReadCache()** function to transfer the information from the cache to a specified frame to examine and process the data.

The **HaveCache()** function returns the number of lines of cache that are installed on the board.

Grabbing Images with Non-Standard Video Formats

With the standard video formats NTSC and CCIR/PAL, the frame grabber can automatically synchronize to vertical and horizontal sync signals in the incoming video and capture the correct number of lines in the individual fields or frames of interlaced video. For cases where you aren't working with the standard sync signals or field lengths, PX frame grabbers include versatile capabilities for synchronizing the frame grabber with the video source and for adjusting to variations in video field length.

PX510 and PX610 frame grabbers support these synchronization modes, which you can set using the **SetVideoFormat()** function:

Automatic. The frame grabber automatically detects and synchronizes to the NTSC and CCIR/PAL video formats. Automatic synchronization is the default mode and is the simplest way to capture standard NTSC and CCIR/PAL video signals. (Automatic synchronization is the only mode the PX500 frame grabber supports.)

Internal. The frame grabber generates its own internal synchronization information. Internal synchronization is useful when you want to synchronize the camera to the frame grabber using the frame grabber's synchronization drive signals (see *Synchronization Drive Signals*, on page 60).

WEN. The frame grabber gets horizontal sync information from the incoming video signal, but vertical sync occurs when the frame grabber detects a window enable (WEN) pulse on the trigger line. WEN synchronization is intended for use with a particular type of resettable camera. Most resettable cameras generate a vertical sync pulse when they are reset; other resettable cameras don't generate a vertical sync signal, but instead generate a WEN signal on a separate line. In order to use the frame grabber with this type of camera, you must connect the WEN output of the camera to the trigger input of the frame grabber. When you select WEN synchronization, you must specify the length, in horizontal periods, of the video field and of the vertical blank (see the owner's manual for your camera for this information).

User. User synchronization works like Automatic synchronization, with the frame grabber synchronizing to the vertical and horizontal synchronization signals of the incoming video signal, but User synchronization mode lets you specify the length of the video field and of the vertical blank. User synchronization automatically adjusts for odd fields having a vertical blank that is one period longer than the vertical blank for even fields, which is standard for NTSC and CCIR/PAL video. User synchronization is useful for capturing images from cameras that output standard sync signals but non-standard field lengths.

In addition to the three modes listed above, the PX610 supports one additional synchronization mode:

Single-Field. Single-Field synchronization assumes that every video frame consists of a single field, rather than two fields separated by a vertical blank. Single-Field synchronization is useful for capturing from non-interlaced video sources, typical of many progressive-scan

cameras. For example, a progressive-scan camera might output a non-interlaced form of NTSC video with 486 lines of valid video and 39 lines of vertical blank for every frame. You can specify Single-Field sync in combination with User, WEN, or Internal to handle non-interlaced video sources using different synchronization methods. You may not use Single-Field sync alone.

The `SetVideoFormat()` function lets you specify any of the synchronization modes, the length of the video field, and the length of the vertical blank period. Standard values for interlaced signals and typical values for non-interlaced signals are given in the following table. For non-interlaced cameras, refer to the manual that came with your camera for actual values.

Video Format	Field Length	Vertical Blank
NTSC interlaced	243	19
CCIR/PAL interlaced	288	24
NTSC non-interlaced	486	39
CCIR/PAL non-interlaced	576	49

For the Automatic mode, the field length and vertical blank length parameters are ignored.

Accessing Frame Data

You can access image data stored in a frame in main memory allocated by `AllocateBuffer()` 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.
- Use the **GetRectangle()**, **PutRectangle()**, **GetRow()**, **PutRow()**, **GetColumn()**, and **PutColumn()** functions to access the frame buffer data. 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 fail if the buffer you are copying to isn't large enough to hold the data. For buffers >64 KB in size, be sure to declare the type of the buffer variable as *huge*.

You can also get the height, width, and number of bits per pixel of the frame by calling **FrameHeight()**, **FrameWidth()**, and **FrameBits()**.

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.

Note

FrameAddress() returns zero and has no effect in Windows NT programs. Windows NT doesn't allow applications access to the hardware in this fashion for reasons of security and stability.

Setting Video Offset and Gain

Video Offset

The offset adjusts the D.C. video level up or down in 256 steps, by approximately 100 percent in either direction. This allows a bright video peak to be brought down to digital zero, or a dark video level to be boosted up to digital 255. The **SetOffset()** function accepts values ranging from -128 to +127.

GetOffset() returns the current offset value.

At power-up, offset = -8. This offset value tends to compensate for the video pedestal (the built-in video bias).

Video Gain

PX frame grabbers sample the incoming video signal and assign grayscale values from zero to 255 to amplitude values that are within sampling range. If the amplitude of the input signal is greater than the sampling range, all samples above the range will be assigned the same grayscale value, essentially attenuating the signal. If you don't want this to happen, you can decrease the video gain so that all of the signal is within the sampling range.

If the range of amplitude in the incoming signal is much narrower than the sampling range, you'll be using only a portion of the full grayscale, and amplitude values that are close together will be assigned the identical grayscale value. You can increase the gain to effectively magnify the amplitude of the incoming signal so that it uses more of the grayscale.

PX frame grabbers let you adjust signal gain by selecting *fine* grain adjustments within four gain *ranges*. The gain ranges are 1/2 to 1, 1 to 2, 2 to 4, and 4 to 8. These gain ranges overlap by a few percent. The range 1/2 to 1 allows you to bring down video that is too bright. You set the gain range using the function **SetGainRange()** with a value of 0, 1, 2, or 3 to specify the range. The function **GetGainRange()** returns the currently selected range.

Within each gain range, the fine gain can be varied in 256 steps by using the **SetFineGain()** function with a value from 0 to 255. The fine gain setting deviates from linearity by a maximum of 11% in the middle of the range. The **GetFineGain()** function returns the current value set by **SetFineGain()**.

You can calculate the total gain using the following formula:

$$\text{total gain} = \frac{256}{512 - (\text{fine gain})} \times 2^{(\text{gain range})}$$

where *fine gain* is the value (0-255) of the parameter for the `SetFineGain()` function and *gain range* is the value (0-3) of the parameter for the `SetGainRange()` function.

At power-up, the gain range is 1 to 2, and the fine gain is 0, for a total gain of 1.

Selecting Camera Inputs

Each frame grabber can have up to four cameras 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, PX frame grabbers automatically detect the video format (NTSC or CCIR/PAL) on the active camera input. If you need to determine the video type 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 gen-locked or not, and brightness levels. If the cameras are all the same video type, there should be a delay of no more than one field time before re-synchronization occurs; if they are also gen-locked, there will be no appreciable delay. (Cameras of different video types can't be gen-locked.) If the cameras are not of the same video type, there may be a delay of as much as four field times 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.

Input/Output

PX frame grabbers include digital I/O features that let you synchronize the frame grabber with other devices in the system.

Trigger

PX frame grabbers have an external TTL-level trigger input that can be used to trigger an image capture using the **GrabTriggered()** or **CacheTriggered()** functions, or to trigger a strobe sequence (see *Strobes*, on page 59). A simple push button switch attached to this input can be used like a camera shutter button.

The **SetTriggerType()** function lets you specify how the frame grabber should treat the trigger signal:

- **Level-sensitive.** The trigger input can be programmed to respond to either a logic high (HIGH) or low (LOW). You should use the level-sensitive trigger mode whenever the trigger input pulse has a width of less than one field time (16 ms).
- **Edge-sensitive.** The trigger can be programmed to respond to a transition from high to low (FALLING) or from low to high (RISING). In applications where the trigger input pulse lasts longer than one field period, use an edge-sensitive trigger.
- **Debounce compensation.** Mechanical switches used as the trigger input can *bounce* (create spurious edges) when opening or closing. This causes problems in the edge-triggered mode. For example, a switch to ground will cause a falling edge when it closes, but will also cause more falling edges when it reopens due to the microscopic bounce of the switch contacts. In this case, a negative-edge trigger mode will experience unexpected triggers.

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 the edge-triggered mode in combination with debounce mode can't be used to grab two consecutive fields. In the switch example above, the switch would need to be open for at least one field time before closing again.

Strobes

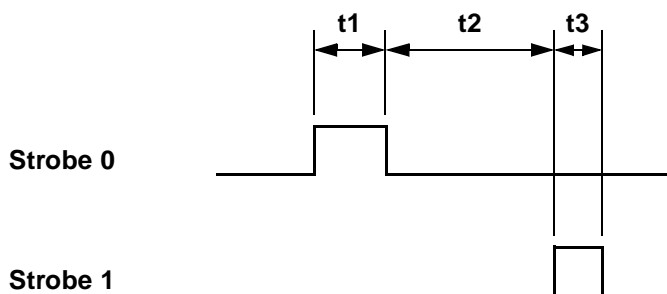
PX frame grabbers have two strobe lines that you can use to output TTL-level pulses. Using the **SetStrobeType()** function, you can set the strobe activity to one of these modes:

Off—Strobe lines are disabled (default).

Normal—Outputs TTL-level pulses on the strobe lines when initiated by the **FireStrobe()** function.

Triggered—Outputs TTL-level pulses on the strobe lines when initiated by an incoming trigger signal.

When you select either a normal or triggered TTL-level strobe, the strobe lines output a pulse on strobe 0, followed by a gap, followed by a pulse on strobe 1, as shown in the following figure:



You control the lengths of the pulses (t_1 and t_3) and the gap (t_2) with the **SetStrobePeriods()** function. Pulse lengths are set in multiples of the horizontal scan frequency (63.5 microseconds for NTSC video, 64 microseconds for CCIR/PAL). You control the strobe polarity, whether a strobe line is high or low when active, by using the **SetStrobePolarity()** function. You can control strobe polarity individually for each strobe line.

When the strobos are in triggered mode, the frame grabber initiates a strobe sequence whenever the board receives a trigger signal of the type specified by **SetTriggerType()**. To initiate a normal strobe, you must call the **FireStrobe()** function and specify one of the strobe commands. The strobe commands let you fire part or all of a strobe sequence, abort a strobe sequence in progress, or stop all strobe activity.

Synchronization Drive Signals

PX frame grabbers have two sync lines that you can use to output TTL-level, horizontal and vertical synchronization signals. You can control the polarity of the sync signals using the **SetDrivePolarity()** function.

You can avoid a possible feedback loop when the frame grabber is providing sync signals to the camera that is generating the video input. Before enabling the sync outputs, use the **SetVideoFormat()** function to set the board to *internal* synchronization mode, so the board doesn't try to synchronize to the incoming video signal. When you want the board to synchronize to the incoming video signal, use **SetVideoFormat()** to set the mode to *automatic* synchronization.

Using the Input Lookup Table (LUT)

A lookup table, or LUT, is a table that is used to change the value of a pixel based on its current value. The current grayscale value of a pixel is used as an index into a LUT, and each entry in the LUT is itself a gray-

scale value. The process of applying a LUT to an image involves examining each pixel to determine its current grayscale value, using this value to “look up” the new value in the LUT, and assigning the new value to the pixel.

PX frame grabbers have a 256-entry LUT that is applied to all images as they are captured. On power-up, the LUT is loaded with ascending values from 0 to 255, so that it leaves any captured image unchanged. You can use the **SetLUT()** function to change the values in the LUT, and the **GetLUT()** function to read the current values.

Reading or writing the LUT takes a noticeable fraction of a second; you can’t load the entire LUT during vertical blank.

Getting Information about Incoming Video

Application programs can compare the digitized pixel values of the incoming video to a reference value, read and write the built-in field counter, and determine the video format.

Checking Pixel Values

You can query the frame grabber to see if any pixels in the last video frame were equal to, or greater than, a specific value. The value is determined after the video is digitized and has passed through the input LUT. The **SetCompare()** function lets you set the comparison value, and the **CheckEqual()** and **CheckGreater()** functions return the results of the comparison.

These functions are useful for automatically checking and adjusting the video gain. If you find pixel values at the extreme (255), you might be losing useful information in the video image. You can repeatedly adjust the gain and re-check until all pixel values are within range (<255).

Video Format

PX frame grabbers support several video formats, including NTSC and CCIR/PAL. Both of these formats are typically interlaced, alternately sending the odd lines of the image as one field and the even lines of the image as another field. In addition to the standard interlaced formats, the PX610 supports non-interlaced video signals, like those generated by many progressive-scan cameras. In non-interlaced format, all lines of the image are sent in order.

On power-up and when switching between video inputs, the frame grabber automatically detects the video format within approximately four fields. You can use the **VideoType()** function to determine if the active video source is in NTSC format, CCIR/PAL format, or some other format. If the frame grabber detects a field length that is not consistent with either NTSC or CCIR/PAL, **VideoType()** reports the type as *other*.

For both interlaced formats, a video image consists of a *video frame* containing two *fields*. The period between fields is called *vertical blank*.

In NTSC video mode, the board begins and ends the sampling of the horizontal video exactly where video should begin and end, according to the NTSC RS-170 video standard. Not all cameras adhere exactly to the video standards, so don't be surprised if several columns on the extreme left or right edge of your image contain invalid information. Also, some cameras generate video artifacts on the extreme left or right edge of the image. In many video applications, these anomalies would not be a problem because they would be off the edges of the display. Your software may have to compensate by not performing analysis on these columns. For information on cropping images, see *Cropping Images*, on page 73.

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.

Reading Frame Grabber Information

Video Cache RAM

Video cache RAM is an image buffer located on the frame grabber. You can grab an image and store it in the video cache without using the PCI bus.

Video cache RAM is available only if you have purchased that option. An application can use the **HaveCache()** function to determine how many lines of video cache RAM are available.

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 PXREV program.

Any version of Windows—Run any of the PXGDI 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.

Board Configuration

The `ReadConfiguration()` function returns flags for features that the board supports:

- Video cache RAM
- Type of bus design (PCI, PC/104 Plus, Compact PCI)
- Horizontal and vertical image cropping and scaling
- Non-interlaced video sources
- Strobe signal output
- Sync drive signal output
- WEN signal trigger for vertical sync
- Board model (PX500 or PX510/PX610)

Timing the Execution of Functions

The PX 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 (;;)
{
    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 PX 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.

PX 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 70). 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:

```
int grab1, grab2;
grab1 = Grab(fgh, fbuf1, QUEUED);
grab2 = Grab(fgh, fbuf2, QUEUED);
while (!IsFinished(fgh, grab1))
    WaitVB(fgh); /* wait until grab 1 has completed */
for (;;)
{
    ProcessImage(fbuf1);
    grab1 = Grab(fgh, fbuf1, QUEUED);
    while (!IsFinished(fgh, grab2))
        WaitVB(fgh); /* wait until grab 2 has completed */
    ProcessImage(fbuf2);
    grab2 = Grab(fgh, fbuf2, QUEUED);
    while (!IsFinished(fgh, grab1))
        WaitVB(fgh); /* wait until grab 1 has completed */
}
```

The **IsFinished()** function is used to determine whether a function has completed. In the example above, once **IsFinished()** indicates that the first **Grab()** is complete, the program starts processing the first image. **IsFinished()** 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 **IsFinished()** checks the appropriate queue based on the handle *fgh* that you specify.

Note

*There is an important difference in the behavior of the queue in Windows NT and the behavior of the queue in other operating systems. For more information, see [Queue Structure](#) under *Windows NT*, on page 67.*

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.

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.

Queue Structure under Windows NT

There is a subtle but important difference in behavior between the queueing structure that the DOS, Windows 3.1, and Windows 95 drivers use, and the queueing structure that the Windows NT driver uses. The DOS/Windows 3.1/Windows 95 queue is fixed in size; if the queue is full and an application attempts to queue another function, the function will fail without effect.

The Windows NT queue, on the other hand, is variable in size, but has only a limited number (128) of queue handles available to the PX500 driver. If there are 128 operations in the queue, and an application queues up another operation, the handle of the 129th queued operation will be the same as the 1st. This will not affect the proper processing of all elements in the queue. However, if you then call `WaitFinished()` (for example) with that handle, it will not return until the 129th operation has completed, rather than returning when the 1st operation has completed. This is not likely to be a problem unless you have a processor that is capable of queueing up 128 items in less than the time that it takes the longest queued operation to complete.

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 these functions:

Grab()	ReadCache()	SetOffset()
GrabToCache()	SetCamera()	Wait()
GrabTriggered()	SetFineGain()	
CacheTriggered()	SetGainRange()	

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

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

GetCamera()	GetOffset()	SetLUT()
GetFineGain()	GetTriggerType()	SetTriggerType()
GetGainRange()	SetFieldSize()	SetVideoFormat()
GetLUT()	SetImageSize()	

GrabContinuous() always acts as if it were declared *immediate*.

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
CACHE	Captured data is sent to the onboard cache.
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 SetOffset() and SetFineGain(), ignore

the FIELD choice flags and always operate as if the EITHER flag was specified.

Specifying Image Capture Resolution

PX frame grabbers sample lines of the incoming video signal at a horizontal resolution of either 640 pixels per scan line or 768 pixels per scan line. Vertical resolution is always one pixel per scan line. Using the **SetImageSize()** function you can choose how much of this image data you want to work with. By selecting only a subset of the image, you save memory and bandwidth on the bus, leaving more of these resources available to other parts of your application and for other applications.

When you're capturing fields, rather than entire frames, use the **SetFieldSize()** function rather than **SetImageSize()**. You'll be capturing fields rather than frames when working with non-interlaced video sources, such as progressive-scan cameras.

For cameras that don't produce continuous vertical sync signals, which the frame grabber needs to determine the start and end of each frame, you can use the **SetVideoFormat()** function to tell the frame grabber what field length to expect. When you need to change the video format and the image size for a subsequent capture, always call **SetVideoFormat()** first, and then call **SetImageSize()** (or **SetFieldSize()**) before capturing the image.

Scaling Images

PX frame grabbers can scale the video image by discarding pixels along both the horizontal and vertical axes, a technique called decimation. To scale an image horizontally, you simply specify the number of pixels you want per horizontal scan line, using the *resx* parameter of **SetImageSize()**. Valid values for *resx* are: $resx = 640/m$ for NTSC or

$768/m$ for CCIR/PAL, where m is an integer, $1 \leq m \leq 64$. Round all values up to the next larger integer.

To scale an image vertically, you don't specify the number of pixels you want along the vertical axis. Instead, you specify vertical resolution based on a value of 256 equaling one pixel per line of video: $resy = 256/n$, where n is an integer, $1 \leq n \leq 256$. Round all values up to the next larger integer. For example, specifying a vertical resolution of $resy = 256$ gives one pixel per line of video; specifying $resy = 128$ gives one pixel for every two lines of video; and so on. You specify vertical resolution independent of the video mode; specifying $resy = 128$ gives you one pixel for every two lines of video in either NTSC or CCIR/PAL.

The default horizontal resolution is 640 pixels per scan line, and the default vertical resolution is 256 (one pixel per scan line) for all video formats. On a typical display monitor with a 4 x 3 aspect ratio, the 640-pixel horizontal resolution gives approximately square pixels for images in NTSC video mode; the 768-pixel horizontal resolution gives approximately square pixels for images in CCIR/PAL video mode, but you can use either resolution with either video mode.

For image scale factors between one and 1/20, valid values for horizontal and vertical resolution are listed in the following table.

Scale Factor	resx NTSC	resx CCIR	resy	Scale Factor	resx NTSC	resx CCIR	resy
1	640	768	256	1/11	59	70	24
1/2	320	384	128	1/12	54	64	22
1/3	214	256	86	1/13	50	60	20
1/4	160	192	64	1/14	46	55	19
1/5	128	154	52	1/15	43	52	18
1/6	107	128	43	1/16	40	48	16
1/7	92	110	37	1/17	38	46	16

Scale Factor	resx NTSC	resx CCIR	resy	Scale Factor	resx NTSC	resx CCIR	resy
1/8	80	96	32	1/18	36	43	15
1/9	72	86	29	1/19	34	41	14
1/10	64	77	26	1/20	32	39	13

Cropping Images

In addition to scaling images, `SetImageSize()` lets you crop images vertically and horizontally. You crop an image by specifying the starting row and number of rows to keep, and the starting column and number of columns to keep. If you are also scaling the image, you specify the cropping parameters based on the size of the image after scaling has been applied.

Starting row—any integer in the range $0 \leq y0 \leq 1,023$. While you can specify any integer up to 1,023, for standard video signals your practical upper limit will be the last line of valid video (line 485 for NTSC; line 575 for CCIR/PAL).

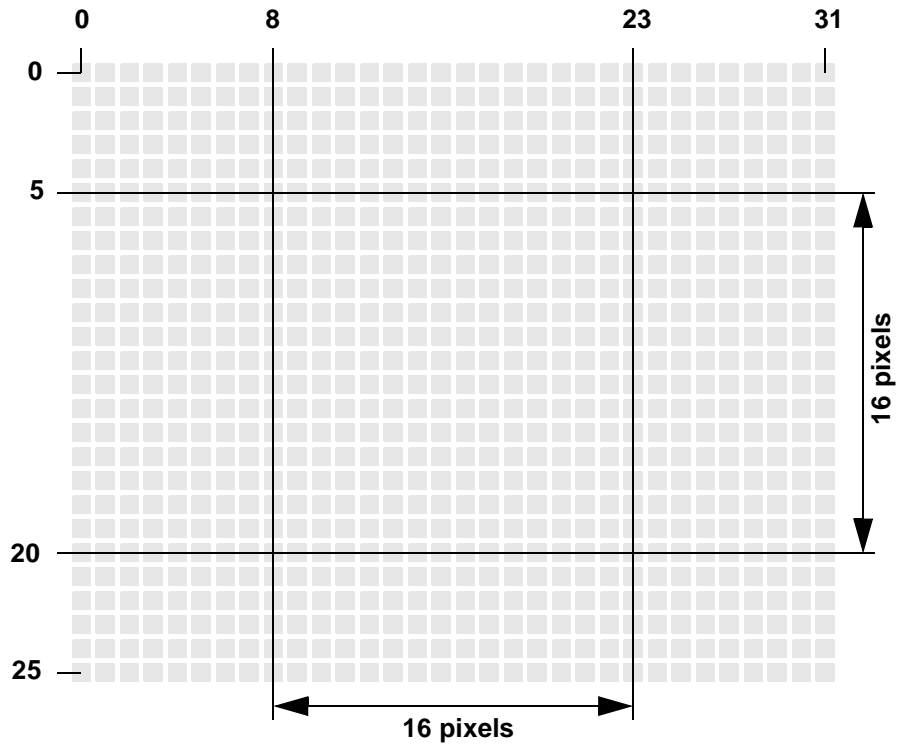
Number of rows—any integer in the range $1 \leq dy \leq (1,024 - y0)$. For standard video signals, your practical upper limit will normally be lower, depending on the type of video: $dy \leq (486 - y0)$ for NTSC; $dy \leq (576 - y0)$ for CCIR/PAL.

Starting column—any value in the range $0 \leq x0 \leq (resx - 1)$.

Number of columns—any value in the range $1 \leq dx \leq resx$.

The figure below shows an example of an NTSC image that has been scaled to 1/20 of full size, or 32 pixels by 26 pixels. From the table of scale factors on page 72, you would specify the scaling parameters for this image as $resx = 32$ and $resy = 13$. 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 $x0 = 8$, $dx = 16$, $y0 = 5$, and $dy = 16$.



For all video formats, the default starting row is row is $y0 = 4$, and the default number of rows is $dy = 480$. For PX frame grabbers, row zero of the video image is the first row of valid video.

Note

NTSC and CCIR/PAL 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 CCIR/PAL) 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.

Calling the `SetImageSize()` function after grabbing an image to the cache and before calling `ReadCache()` can cause `ReadCache()` to work incorrectly. If you're using `ReadCache()` to access image data, and you want to change the image size parameters for subsequent captures, always read the cache first and then change the image parameters for the next capture.

Grayscale Resolution

Grayscale resolution is always eight bits per pixel, providing 256 shades of gray.

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 8-bit (256-color) image files. 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 files. Unformatted files contain no information on an image's height, width, or number of bits per pixel, 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 (PXDV) 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. PXDV 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, FRAMEHANDLE 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 PX500 DLL. This means that the library must be initialized properly and a frame must be allocated before PXDV can be used.

PXDV 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:

Windows 3.1	Windows 95	Windows NT
PXDV.LIB	PXDV95.LIB	PXDVNT.LIB
PXDV.DLL	PXDV95.DLL	PXDVNT.DLL
PXDV.BAS	PXDV95.BAS	PXDVNT.BAS

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 76) in your program's source or header files; these prototypes do not appear in the main header files.

On the distribution disk, there is a sample Visual Basic program called ZOOM which uses the Video Display DLL to display video from a PX frame grabber in a window with zoom and pan controls. This program demonstrates several useful techniques, including the proper initialization of the necessary libraries and the use of a timer to cause the image to be continuously updated.

Developing a Menu-Based User Interface for DOS Applications

The VESAMENU library is a DOS-based VGA display and menu builder for both 16-bit and Watcom 32-bit DOS applications. The library makes it easy to create and display a graphics menu-based interface for a program. Imagination used this library to create the interface for PCIVU and for most of the DOS sample programs on the PX distribution disks.

The library is described in detail in [Chapter 6, *VESAMENU Library*](#), on page 121.

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 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. For more information, including results for several common configurations, see [Appendix E, *PCI Bus System Performance*](#), on page 163.

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`.

5

Function Reference

The chapter is a complete, alphabetical function reference for the PX libraries and DLLs. For additional information on using the functions, see [Chapter 4, *Programming PX Frame Grabbers*](#), on page 35.

The 16-bit Windows 3.1 PX DLLs use the Pascal calling convention. The 32-bit Windows 95 and Windows NT PX DLLs use the `_stdcall` calling convention. All variables declared as *int* are 16 bits long in DOS and Windows 3.1 and 32 bits long in DOS/4GW, Windows_95, and Windows_NT.

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

Imagination


header files for C and Visual BASIC. The following table gives the sizes of the various data types that are used by the PX library.

Type	Size
unsigned char	8 bits
long, unsigned long	32 bits
void *, unsigned char *, int *, char *, LPSTR	32 bits
int	16 bits in DOS and Windows 3.x; 32 bits in DOS/4GW, Windows 95, and Windows NT

FGHANDLE and FRAMEHANDLE 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 are noted with an icon:

 Does not apply to Windows NT

 Does not apply to DOS

AllocateAddress

Syntax FRAMEHANDLE AllocateAddress(unsigned long address, int dx, int dy, int bits);

Return Value A handle for the allocated frame structure.
0 on failure.

- Description** Creates a frame of size dx by dy , with the specified number of *bits* per pixel, from the memory at the specified physical *address*. It does no checking to determine whether the given address is valid, or whether the memory at the given address is being used for other purposes. For PX frame grabbers, dx must be a multiple of four, and *bits* is always 8.
- 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** **AllocateBuffer, FreeFrame**

AllocateBuffer

- Syntax** FRAMEHANDLE AllocateBuffer(int dx, int dy, int bits);
- Return Value** A handle to the allocated FRAME structure.
0 on failure.
- Description** Reserves memory for an image buffer of size dx by dy , with the specified number of *bits* per pixel. For the buffer to be usable by the frame grabber, dx and dy must be at least as large as the image being grabbed. For PX frame grabbers, dx must be a multiple of four, and *bits* is always 8. FreeFrame() should be used to release the frame when it is no longer needed.
- See Also** **FreeFrame**

AllocateFG

- Syntax** FGHANDLE AllocateFG(int n);
- Return Value** A handle for the requested frame grabber.
0 on failure.

Imagenation

Description	<p>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, <i>n</i>. Due to the design of the PCI bus, bus slot <i>0</i> doesn't necessarily correspond to frame grabber <i>0</i>, 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 PCIVU program (or your own program) to switch between frame grabbers. To request any available frame grabber, specify $n < 0$.</p> <p>If the frame grabber is available, AllocateFG() returns a handle that must be used in other library functions that refer to the frame grabber.</p> <p>The program should call FreeFG() on the frame grabber when it is no longer needed.</p>
See Also	FreeFG

CacheTriggered

Syntax	<code>int CacheTriggered(FGHANDLE fgh, int deltime, int flags);</code>
Return Value	Non-zero if successful. 0 on failure.
Description	<p>After the frame grabber receives a trigger signal, it waits <i>deltime</i> field times and then captures a video image to the onboard cache memory. The <i>deltime</i> parameter can be used to delay the capture to allow time for a camera to reset, for example. This function can be used to synchronize frame grabbing to an external device.</p> <p>The parameter <i>flags</i> is a set of flag bits that can specify modes of operation for this function. If <i>flags</i> is 0, the default modes will be used. See <i>Using Flags with Function Calls</i>, on page 70.</p> <p>For more information on the video cache RAM, see <i>Sending Images to the Onboard Video Cache RAM</i>, on page 51.</p>
See Also	GrabTriggered, SetTriggerType, ReadCache

CheckEqual

Syntax	<code>int CheckEqual(FGHANDLE fgh);</code>
Return Value	Non-zero if equal. 0 if not equal.
Description	Compares the grayscale value of each pixel in the previously digitized field against the value specified by <code>SetCompare()</code> . If any pixel value is equal to the set value, <code>CheckEqual()</code> returns a non-zero value.
See Also	SetCompare, CheckGreater

CheckError

Syntax	<code>int CheckError(FGHANDLE fgh);</code>
Return Value	<code>CheckError()</code> returns the following values:

Error Returned	Description
<code>ERR_BAD_IRQ*†</code>	An interrupt line hasn't been properly assigned to one or more of the frame grabbers. On some systems, you might need to configure the BIOS to assign these interrupts properly.
<code>ERR_CORRUPT</code>	Captured image data is corrupt.
<code>ERR_NONE</code>	No error detected.
<code>ERR_NOT_VALID</code>	<i>fgh</i> is not a valid frame grabber handle.
<code>ERR_NO_ADDRESS*†</code>	The library couldn't locate enough free address space in upper memory to enable communication with the frame grabber.
<code>ERR_NO_DEVICES</code>	No PX frame grabbers were detected.

Imagenation

Error Returned	Description
ERR_NO_DOS	The DOS library tried to load while Windows was running.
ERR_NO_PCI*†	No PCI BIOS was detected. Your BIOS ROM might not support PCI.

* Initialization error.

† If the VxD detects one of these errors, it will not allow Windows to load. These errors can only be reported by a DOS application.

Description

If *fgh* is NULL, CheckError() checks to determine whether any of a known set of errors occurred during initialization. These initialization errors are shown above marked with an asterisk.

If *fgh* is not NULL, CheckError() queries the frame grabber to determine whether any of a known set of errors occurred. These are the errors shown above without asterisks. Any of these errors are automatically cleared when CheckError() returns.

CheckGreater

Syntax int CheckGreater(FGHANDLE fgh);

Return Value Non-zero if greater.
0 if not greater.

Description Compares the grayscale value of each pixel in the previously digitized field against the value specified by SetCompare(). If any pixel value is greater than the set value, CheckGreater() returns a non-zero value.

See Also **SetCompare, CheckEqual**

ExitLibrary

Syntax	<code>void ExitLibrary(void);</code>
Return Value	None.
Description	Returns to the system any resources that were allocated by <code>InitLibrary()</code> . <code>ExitLibrary()</code> should be the last library function called by the program. A program that exits after calling <code>InitLibrary()</code> , but before calling <code>ExitLibrary()</code> , will leave the computer in an unstable state and might crash the operating system.
See Also	InitLibrary

FireStrobe

Syntax	<code>int FireStrobe(FGHANDLE fgh, int command);</code>
Return Value	Non-zero if successful. 0 on failure.
Description	Controls the strobe activity. The <i>command</i> parameter can accept the following values:

Value	Description
STROBE_STOP	Turns off all strobe activity immediately for both normal and triggered strobe modes. Does not affect the strobe type.
STROBE_0	Fires a complete strobe sequence (strobe 0, gap, strobe 1).
STROBE_GAP	Fires the gap, followed by strobe 1.
STROBE 1	Fires strobe 1 only.

Imagination

If a strobe sequence is in progress, calling `FireStrobe()` immediately cancels the current strobe sequence and starts a new one. For more information, see *Strobes*, on page 59.

See Also **GetStrobeState, SetStrobePeriods, SetStrobePolarity, SetStrobeType**

FrameAddress

Syntax `unsigned long FrameAddress(FRAMEHANDLE frh);`

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. Frames whose image buffers are not at a fixed physical address cannot be accessed by the frame grabber.

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**

FrameBits

Syntax `int FrameBits(FRAMEHANDLE frh);`

Return Value Number of bits per pixel.
0 if the frame handle is invalid.

Description Returns the number of bits per pixel in the specified frame. Currently the frame grabber can only write to 8-bit frames.

FrameBuffer

Syntax	<code>void *FrameBuffer(FRAMEHANDLE frh);</code>
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	<code>int FrameHeight(FRAMEHANDLE frh);</code>
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 <code>AllocateAddress()</code> or with <code>AllocateBuffer()</code> .
See Also	FrameWidth

FrameWidth

Syntax	<code>int FrameWidth(FRAMEHANDLE frh);</code>
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 <code>AllocateAddress()</code> or with <code>AllocateBuffer()</code> .
See Also	FrameHeight

Imagination

FreeFG

Syntax	<code>void FreeFG(FGHANDLE fgh);</code>
Return Value	None.
Description	Releases control of a frame grabber (previously allocated with the <code>AllocateFG()</code> function) after the program is finished using the frame grabber.
See Also	AllocateFG

FreeFrame

Syntax	<code>void FreeFrame(FRAMEHANDLE frh);</code>
Return Value	None.
Description	Returns memory associated with a <code>FRAMEHANDLE</code> to the system.
See Also	AllocateBuffer

GetCamera

Syntax	<code>int GetCamera(FGHANDLE fgh);</code>
Return Value	The current camera input. -1 on failure.
Description	Returns the active camera input of the specified frame grabber. Use <code>SetCamera()</code> to specify the active camera input. If the frame grabber is processing queued operations, <code>GetCamera()</code> waits for the operations to finish before executing.
See Also	SetCamera

GetColumn

- Syntax** `void GetColumn(FRAMEHANDLE frh, unsigned char *buf, int col);`
- Return Value** None.
- Description** Copies a column of the image stored in frame *frh* into the buffer *buf*. The columns are numbered starting with 0 at the left of the frame. The buffer *buf* must be at least as large as `FrameHeight(frh)`. At present `GetColumn()` only works on frames with 8-bit pixels.
- See Also** **GetRow, PutColumn, PutRow**
-

GetFieldCount

- Syntax** `long GetFieldCount(FGHANDLE fgh);`
- Return Value** Returns the field count.
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. For more information, see *Counting Fields*, on page 62.
- See Also** **SetFieldCount**
-

GetFieldLength

- Syntax** `int GetFieldLength(FGHANDLE fgh);`
- Return Value** The field length of the last video field, as measured by the frame grabber.
-1 on error.
- Description** Returns the field length of the last video field, including vertical blank, for the specified frame grabber.
- See Also** **GetFieldLength**

Imagenation

GetFineGain

Syntax int GetFineGain(FGHANDLE fgh);

Return Value The current fine gain setting.
-1 on failure.

Description Returns the fine gain setting of the specified frame grabber. Use SetFineGain() to specify the gain.

If the frame grabber is processing queued operations, GetFineGain() waits for the operations to finish before executing.

See Also **SetFineGain**

GetGainRange

Syntax int GetGainRange(FGHANDLE fgh);

Return Value The current gain range.
-1 on failure.

Description Returns the gain range of the specified frame grabber. Use SetGainRange() to specify the gain range.

If the frame grabber is processing queued operations, GetGainRange() waits for the operations to finish before executing.

See Also **SetGainRange**

GetLUT

Syntax int GetLUT(FGHANDLE fgh, int first_address, int length, int *buf);

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

Description Reads the frame grabber's input lookup table, or a section of it, into the array *buf*. The section to be read is specified by *first_address* and the number of entries (*length*) desired. The buffer must have at least *length* entries.

If the frame grabber is processing queued operations, GetLUT() waits for the operations to finish before executing.

See Also **SetLUT**

GetOffset

Syntax int GetOffset(FGHANDLE fgh);

Return Value The current offset.
255 on failure.

Description Returns the offset of the specified frame grabber. Use SetOffset() to specify the offset.

If the frame grabber is processing queued operations, GetOffset() waits for the operations to finish before executing.

See Also **SetOffset**

GetRectangle

Syntax void GetRectangle(FRAMEHANDLE frh, unsigned char *buf, int x1, int y1, int dx, int dy);

Return Value None.

Description Copies a rectangular region of the frame *frh* into the buffer *buf*. The rectangle has upper left corner $(x1, y1)$ in the source frame, width *dx*, and height *dy*. The buffer *buf* must be at least as large as $dx * dy$. At present GetRectangle() only works on frames with 8-bit pixels.

See Also **PutRectangle**

GetRow

Syntax void GetRow(FRAMEHANDLE frh, unsigned char *buf, int row);

Return Value None.

Imagination

Description Copies a row of the image stored in frame *frh* into the buffer *buf*. The rows are numbered starting with 0 at the top of the frame. The buffer *buf* must be at least as large as `FrameWidth(frh)`. At present `GetRow()` only works on frames with 8-bit pixels.

See Also **GetColumn, PutColumn, PutRow**

GetStrobeState

Syntax `int GetStrobeState(FGHANDLE fgh);`

Return Value Strobe state information if successful.
-1 on failure.

Description Returns the current state of the strobe lines in a collection of flags. The flags are:

Return Value	Description
STROBE_0	Strobe 0 is high (+5 V). Absence of this flag indicates strobe 0 is low (0 V).
STROBE_1	Strobe 1 is high (+5 V). Absence of this flag indicates strobe 1 is low (0 V).
STROBING	A strobe sequence is in progress.
STROBE_OFF	Strobes are disabled (tri-stated). This is the default.

While a strobe sequence is in progress (`STROBING` is true) or strobes are disabled (`STROBE_OFF` is true), the flags `STROBE_0` and `STROBE_1` won't necessarily return up-to-date values.

While strobes are disabled (`STROBE_OFF` is true), an external device can drive the strobe lines. This allows the strobe lines to be used as general-purpose input lines, using `GetStrobeState()` to read the lines.

See Also **FireStrobe, SetStrobeType**

GetSyncType

Syntax	<code>int GetSyncType(FGHANDLE fgh);</code>
Return Value	Current synchronization mode. -1 on failure.
Description	Returns the currently selected synchronization mode in the same format as used by <code>SetVideoFormat()</code> .
See Also	SetVideoFormat

GetTriggerType

Syntax	<code>int GetTriggerType(FGHANDLE fgh);</code>
Return Value	Trigger type. -1 on failure.
Description	Returns the current trigger type in the same format used by <code>SetTriggerType()</code> . <code>GetTriggerType()</code> waits for the frame grabber to finish processing any queued operations before executing.
See Also	SetTriggerType, GrabTriggered

Grab

Syntax	<code>int Grab(FGHANDLE fgh, FRAMEHANDLE frh, int flags);</code>
Return Value	Non-zero if successful. 0 on failure.
Description	<p>Captures a video image and writes it to frame buffer <i>frh</i>. <code>Grab()</code> fails if the image size, as specified by <code>SetImageSize()</code>, is larger in either the horizontal or vertical dimension than the destination frame.</p> <p>The parameter <i>flags</i> is a set of flag bits that can specify modes of operation for this function. If <i>flags</i> is 0, the default modes will be used. See <i>Using Flags with Function Calls</i>, on page 70. <code>Grab()</code> can also use the special CACHE flag. When you specify the CACHE flag, the frame grab-</p>

ber copies the captured data to both the onboard cache and the specified frame, if *frh* is valid; if *frh* is zero or invalid, the CACHE flag causes the data to be captured only to the onboard cache.

See Also **AllocateFG, AllocateBuffer**

GrabContinuous

Syntax `int GrabContinuous(FGHANDLE fgh, FRAMEHANDLE frh, int flags);`

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

Description Turns continuous acquire mode on (if the *flags* parameter is GRAB_ON) or off (if *flags* is GRAB_OFF) for a given frame grabber. In continuous acquire mode, the buffer *frh* is continuously updated with new video data. GrabContinuous() fails if the image size, as specified by SetImageSize(), is larger in either the horizontal or vertical dimension than the destination frame.

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.

In addition to setting continuous acquire mode, the parameter *flags* can specify additional modes of operation for this function, except that GrabContinuous() always behaves as if it were declared IMMEDIATE; you can't use the QUEUED flag with GrabContinuous(). If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 70. GrabContinuous() can also use the special CACHE flag. When you specify the CACHE flag, the frame grabber copies the captured data to both the onboard cache and the specified frame, if *frh* is

zero or invalid, the CACHE flag causes the data to be captured only to the onboard cache.

If you have set up GrabContinuous() to capture the data to both the onboard cache and a frame, you can't turn it off selectively. You must turn off both capture modes.

GrabToCache

Syntax int GrabToCache(FGHANDLE fgh, int flags);

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

Description Captures a video image to the frame grabber's internal cache memory. 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 70. Using the SINGLE_FLD flag to grab a specific field is possible, but does not prevent the field not grabbed from being overwritten in the cache, so single-field grabs can't be reliably used to store two images in the cache simultaneously.

For more information on the video cache RAM, see *Sending Images to the Onboard Video Cache RAM*, on page 51.

See Also ReadCache, HaveCache

GrabTriggered

Syntax int GrabTriggered(FGHANDLE fgh, FRAMEHANDLE frh, int deltime, int flags);

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

Description After the frame grabber receives a trigger signal, it waits *deltime* field times and then captures a video image to the specified buffer. The *deltime* parameter can be used to delay the capture to allow time for a camera to

Imagenation

reset, for example. This function can be used to synchronize frame grabbing to an external device.

GrabTriggered() fails if the image size, as specified by SetImageSize(), is larger in either the horizontal or vertical dimension than the destination frame.

The parameter *flags* is a set of flag bits that can specify modes of operation for this function. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 70. GrabTriggered() can also use the special CACHE flag. When you specify the CACHE flag, the frame grabber copies the captured data to both the onboard cache and the specified frame, if *frh* is valid. If *frh* is zero or invalid, the CACHE flag causes the data to be captured only to the onboard cache.

See Also **Grab, SetTriggerType**

HaveCache

Syntax int HaveCache(FGHANDLE fgh);

Return Value The number of lines of cache on the board.

Description Returns the amount of video cache installed on the board in number of lines of video that can be stored.

For more information on the video cache RAM, see *Sending Images to the Onboard Video Cache RAM*, on page 51.

See Also **ReadCache, GrabToCache**

InitLibrary

Syntax int InitLibrary(void);

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.

InitLibrary() will usually fail only if no frame grabbers are detected, but may also fail under conditions of extremely low memory. When InitLibrary() fails, use CheckError() to get the error.

For more information on using InitLibrary(), see *Initializing and Exiting the Library*, on page 44.

See Also **ExitLibrary**

IsFinished

Syntax int IsFinished(FGHANDLE fgh, int handle);

Return Value Non-zero if the operation is finished.
0 if the specified operation 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 64.

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 70.

See Also **WaitFinished**

KillQueue

Syntax void KillQueue(FGHANDLE fgh);

Return Value None.

Imagenation

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.

PutColumn

Syntax void PutColumn(unsigned char *buf, FRAMEHANDLE frh, int col);

Return Value None.

Description Copies a column stored in the buffer *buf* into frame *frh*. The columns are numbered starting with 0 at the left of the frame. The buffer *buf* must be at least as large as FrameHeight(*frh*). At present PutColumn() only works on frames with 8-bit pixels.

See Also **Get Column, GetRow, PutRow**

PutRectangle

Syntax void PutRectangle(unsigned char *buf, FRAMEHANDLE frh, int x1, int y1, int dx, int dy);

Return Value None.

Description Copies a rectangular region from buffer *buf* into the frame *frh*. The rectangle goes into *frh* with its upper left corner at (*x1*,*y1*), width *dx*, and height *dy*. The buffer *buf* must be at least as large as *dx*dy*. At present PutRectangle() only works on frames with 8-bit pixels.

See Also **GetRectangle**

PutRow**Syntax** void PutRow(unsigned char *buf, FRAMEHANDLE frh, int row);**Return Value** None.**Description** Copies a row from the buffer *buf* into frame *frh*. The rows are numbered starting with 0 at the top of the frame. The buffer *buf* must be at least as large as FrameWidth(*frh*). At present PutRow() only works on frames with 8-bit pixels.**See Also** **GetColumn, GetRow, PutColumn**

ReadBin**Syntax** int ReadBin(FRAMEHANDLE frh, char *fname);**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.
INVALID_FRAME	The frame handle is invalid, or the frame's image data has no logical address.

Description Reads the unformatted binary file *fname* and copies it into frame buffer *frh*. The function stores as much of the contents of *fname* in the buffer as will fit. ReadBin() opens and closes the file *fname*.When calling this function from a Windows program, the type of *fname* should be LPSTR rather than char *.**See Also** **WriteBin**

Imagenation

ReadBMP

Syntax int ReadBMP(FRAMEHANDLE frh, char *fname);

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.
WRONG_BITS	ReadBMP() attempted to read a file that did not have 8 bits per pixel. The BMP routines read and write only 8-bit image files.
BAD_FILE	ReadBMP() attempted to read a non-BMP-formatted file.
INVALID_FRAME	The frame handle is invalid, the frame's image data has no logical address, or the number of bits in the frame is not 8.

Description

Reads the image stored in the BMP file *fname* and copies it into frame buffer *frh*. The function copies each row of the BMP image to a row in the frame buffer. If the BMP image is smaller than the frame buffer, the image is padded on the right and bottom with zeros. If the BMP image is larger than the frame buffer, the image is clipped on the right and bottom to fit the frame buffer.

ReadBMP() opens and closes *fname*.

When calling this function from a Windows program, the type of *fname* should be LPSTR rather than char *.

See Also

WriteBMP

ReadCache**Syntax** int ReadCache(FGHANDLE fgh, FRAMEHANDLE frh, int flags);**Return Value** Non-zero if successful.
0 on failure.**Description** Copies a video image from the frame grabber's internal cache memory to the specified frame buffer. ReadCache() fails if the image size, as specified by SetImageSize(), is larger in either the horizontal or vertical dimension than the destination frame.

The parameter *flags* is a set of flag bits that can specify modes of operation for this function. If *flags* is 0, the default modes will be used. See *Using Flags with Function Calls*, on page 70.

For more information on the video cache RAM, see *Sending Images to the Onboard Video Cache RAM*, on page 51.

See Also **GrabToCache, HaveCache**

ReadConfiguration**Syntax** long ReadConfiguration(FGHANDLE fgh);**Return Value** Board configuration information if successful.
-1 on failure.**Description** Returns a collection of flags that specify the configuration of the board:

Return Value	Description
PXC_BUS	A multi-bit flag that specifies the bus design the board is connected to. Flag values are PXC_PCI, PXC_104_PLUS, and PXC_COMPACT_PCI.
PXC_CACHE	The board has the optional cache RAM. HaveCache() will return the specific amount of cache.

Imagenation

Return Value	Description
PXC_CUSTOM_HW	A multi-bit flag. Zero indicates the board is a PX500; one indicates the board is a PX510 or PX610.
PXC_H_CROP	The board can crop the image to less than full width.
PXC_H_SCALE	The board can scale the image horizontally.
PXC_NONINTERLACE	The board supports non-interlaced video sources.
PXC_STROBES	The board has strobe outputs.
PXC_V_CROP	The board can crop the image to less than full height.
PXC_V_SCALE	The board can scale the image vertically.
PXC_VIDEO_DRIVE	The board can generate horizontal and vertical sync drive signals.
PXC_WEN_SYNC	The board can use a window enable (WEN) signal on the trigger input as a vertical sync signal.

See Also **HaveCache, ReadProtection, ReadRevision**

ReadProtection

Syntax `int ReadProtection(FGHANDLE fgh);`

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

Description Returns the hardware protection key of the frame grabber. This will be 0x55 unless the frame grabber has been programmed with a key to match your custom software.

ReadRevision

- Syntax** `int ReadRevision(FGHANDLE fgh);`
- Return Value** The revision number if successful.
0 on failure.
- Description** Returns the hardware/firmware revision number of the frame grabber.
- You can also get the revision number using the PXREV utility program in DOS or any of the PXGDI sample programs in Windows. The PXGDI programs display the revision number in the title bar.

ResetFG

- Syntax** `void ResetFG(FGHANDLE fgh);`
- Return Value** None.
- Description** Returns the frame grabber to a default state, and aborts any queued operations. This function takes several milliseconds to execute.
- See Also** **KillQueue**

SetCamera

- Syntax** `int SetCamera(FGHANDLE fgh, int camera, int flags);`
- Return Value** Non-zero 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 70.
- See Also** **GetCamera**

Imagination

SetCompare

Syntax	int SetCompare(FGHANDLE fgh, int value);
Return Value	Non-zero if successful. 0 on failure.
Description	Sets the grayscale <i>value</i> used by the CheckEqual() and CheckGreater() functions for comparing pixel values in the previously digitized video field.
See Also	CheckEqual, CheckGreater

SetCurrentWindow

Syntax	int SetCurrentWindow(FGHANDLE fgh, HWND window);
Return Value	1 if successful. 0 on failure.
Description	SetCurrentWindow() is a Windows-only function. Following a call to SetCurrentWindow(), every queued function sends a message when finished to the window whose handle is <i>HWND</i> . If <i>HWND</i> is zero, no messages are sent. When the library is initialized, message posting is turned off by default.

The format for messages is:

Parameter	Value
message number	Q_MESSAGE
wParam	The <i>handle</i> returned by the function that queued the operation. This is the same <i>handle</i> used by IsFinished().
lParam	0L.

You can use this function to schedule queued events from the frame grabber in the same way you schedule events for other Windows I/O, instead of using `IsFinished()`.

In the example below, the window procedure for the window identified by *hwnd* will eventually receive a message with *wParam* == *h1* to indicate that the frame grab to *frh* is complete:

```
// fgh should be a valid frame grabber handle, as
// returned by AllocateFG
// hwnd should be a valid window handle.
// frh should be a valid frame handle, as returned by
AllocateBuffer
SetCurrentWindow(fgh,hwnd);
h1=Grab(fgh,frh,QUEUED);
```

Only functions with the QUEUED flag set will generate messages. Messages always go to the window specified as the current window at the time the function is **called**. For example, the following code causes one message to be sent to each window.

```
SetCurrentWindow(fgh,window1);
Grab(fgh,frh,QUEUED);
SetCurrentWindow(fgh,window2);
Grab(fgh,frh,QUEUED);
```

Calling `SetCurrentWindow()` with *hwnd* = 0 turns off message posting for functions called after that time. However, functions already in the queue will continue to post messages, so programs should be prepared to process all the messages they have requested. Calling `KillQueue()` or `ResetFG()` will prevent messages from being sent by functions that are successfully cancelled, but functions that were in the queue when `KillQueue()` was called may have just finished execution and already posted their messages.

`ResetFG()` turns off message posting, but `KillQueue()` does not.

See Also

IsFinished

Imagination

SetDrivePolarity

Syntax	<code>int SetDrivePolarity(FGHANDLE fgh, int vdrive, int hdrive);</code>
Return Value	Non-zero if successful. 0 on failure.
Description	Sets the polarity of the horizontal and vertical synchronization drive signals: 0 = active high. 1 = active low.
See Also	SetDriveType

SetDriveType

Syntax	<code>int SetDriveType(FGHANDLE fgh, int mode);</code>						
Return Value	Non-zero on success. 0 on failure.						
Description	Controls the mode of action for the synchronization drive lines. The possible values of <i>mode</i> are: <table border="1"><thead><tr><th>Parameter</th><th>Value</th></tr></thead><tbody><tr><td>SYNC_NORMAL</td><td>Both horizontal and vertical sync outputs are active.</td></tr><tr><td>SYNC_OFF</td><td>Both horizontal and vertical sync outputs are disabled (tri-stated). This is the default.</td></tr></tbody></table>	Parameter	Value	SYNC_NORMAL	Both horizontal and vertical sync outputs are active.	SYNC_OFF	Both horizontal and vertical sync outputs are disabled (tri-stated). This is the default.
Parameter	Value						
SYNC_NORMAL	Both horizontal and vertical sync outputs are active.						
SYNC_OFF	Both horizontal and vertical sync outputs are disabled (tri-stated). This is the default.						
	For more information, see <i>Synchronization Drive Signals</i> , on page 60.						
See Also	SetDrivePolarity, SetVideoFormat						

SetFieldCount

Syntax	<code>int SetFieldCount(FGHANDLE fgh, long count);</code>
Return Value	Non-zero on success. 0 on failure.
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 <code>GetFieldCount()</code> function. For more information, see <i>Counting Fields</i> , on page 62.
See Also	GetFieldCount

SetFieldSize

Syntax	<code>int SetFieldSize(FGHANDLE fgh, int resx, int resy, int x0, int y0, int dx, int dy, int bits);</code>
Return Value	Non-zero if successful. 0 on failure.
Description	Specifies the scaling and cropping of captured images for single-field captures.

Parameter	Description
<code>fgh</code>	A handle to the frame grabber.
<code>resx</code>	The horizontal resolution in pixels per scan line; $resx = 640/m$ for NTSC or $768/m$ for CCIR/PAL, where m is an integer, $1 \leq m \leq 64$. Round all values up to the next larger integer. Default is 640.
<code>resy</code>	The vertical resolution in lines per field; $resy = 256/n$, where n is an integer, $1 \leq n \leq 256$. Round all values up to the next larger integer. Default is 256.

Parameter	Description
<code>x0</code> , <code>y0</code> <code>dx</code> , <code>dy</code>	The cropping rectangle for the image to be captured is defined by the pixel column, <code>x0</code> , representing the left edge; the row, <code>y0</code> , representing the top edge; the width of the rectangle, <code>dx</code> , in pixels; and the height of the rectangle, <code>dy</code> , in pixels. Values must be in the ranges $0 \leq x0 \leq (resx - 1)$; $0 \leq y0 \leq 1,023$; $1 \leq dx \leq resx$; and $1 \leq dy \leq (1,024 - y0)$. Also, <code>dy</code> should be no larger than the total number of valid lines of video in the image to be captured, as specified by <code>SetVideoFormat()</code> . Defaults are <code>x0 = 0</code> , <code>y0 = 4</code> , <code>dx = 640</code> , and <code>dy = 480</code> .
<code>bits</code>	Number of bits per pixel. Must be 8 for PX frame grabbers.

When you need to change the video format and the image size for a subsequent capture, always call `SetVideoFormat()` first, and then call `SetFieldSize()` before capturing the image.

`SetFieldSize()` waits for the frame grabber to finish processing any queued operations before executing. For full-frame captures, use `SetImageSize()` rather than `SetFieldSize()`.

For more information on specifying the image size and resolution for captures, see *Specifying Image Capture Resolution*, on page 71.

See Also

`SetImageSize`

`SetFineGain`

Syntax `int SetFineGain(FGHANDLE fgh, int gain, int flags);`

Input Values An integer from 0 to 255.

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

Description Sets the fine video *gain* for the specified frame grabber. The overall gain is determined from the combination of the fine gain and the gain range. For more information, see *Setting Video Offset and Gain*, on page 55.

At power-up, the fine *gain* is set to 0.

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 70.

See Also **SetGainRange, GetFineGain**

SetGainRange

Syntax int SetGainRange(FGHANDLE fgh, int range, int flags);

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

Description Sets the *range* for the video gain for the specified frame grabber. Gain ranges are:

<i>range</i> parameter	Gain range
0	0.5 - 1
1	1 - 2
2	2 - 4
3	4 - 8

The default at power-up is *range* = 1, for a gain range of 1 - 2.

The overall gain is determined from the combination of the fine gain and the gain range. For more information, see *Setting Video Offset and Gain*, on page 55.

Imagenation

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 70.

See Also **SetFineGain, GetGainRange**

SetImageSize

Syntax `int SetImageSize(FGHANDLE fgh, int resx, int resy, int x0, int y0, int dx, int dy, int bits);`

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

Description Specifies the scaling and cropping of images for full-frame captures.

Parameter	Description
<code>fgh</code>	A handle to the frame grabber.
<code>resx</code>	The horizontal resolution in pixels per scan line; $resx = 640/m$ for NTSC or $768/m$ for CCIR/PAL, where m is an integer, $1 \leq m \leq 64$. Round all values up to the next larger integer. Default is 640.
<code>resy</code>	The vertical resolution in lines per field; $resy = 256/n$, where n is an integer, $1 \leq n \leq 256$. Round all values up to the next larger integer. Default is 256.

Parameter	Description
<code>x0</code> , <code>y0</code> <code>dx</code> , <code>dy</code>	The cropping rectangle for the image to be captured is defined by the pixel column, <code>x0</code> , representing the left edge; the row, <code>y0</code> , representing the top edge; the width of the rectangle, <code>dx</code> , in pixels; and the height of the rectangle, <code>dy</code> , in pixels. Values must be in the ranges $0 \leq x0 \leq (resx - 1)$; $0 \leq y0 \leq 1,023$; $1 \leq dx \leq resx$; and $1 \leq dy \leq (1,024 - y0)$. Also, <code>dy</code> should be no larger than the total number of valid lines of video in the image to be captured, as specified by <code>SetVideoFormat()</code> . Defaults are <code>x0 = 0</code> , <code>y0 = 4</code> , <code>dx = 640</code> , and <code>dy = 480</code> .
bits	Number of bits per pixel. Must be 8 for PX frame grabbers.

When you need to change the video format and the image size for a subsequent capture, always call `SetVideoFormat()` first, and then call `SetImageSize()` before capturing the image.

`SetImageSize()` waits for the frame grabber to finish processing any queued operations before executing. When you are capturing single fields, use `SetFieldSize()` rather than `SetImageSize()`.

For more information on specifying the image size and resolution for captures, see *Specifying Image Capture Resolution*, on page 71.

See Also**SetFieldSize****SetLUT****Syntax**

```
int SetLUT(FGHANDLE fgh, int first_address, int length, int *buf);
```

Return Value

Non-zero if successful.
0 on failure.

Description

Changes values in the frame grabber's input lookup table (LUT). Any subrange of the table can be changed by specifying the *first_address* and

Imagination

length of the range to be altered. The data to be put in the table is specified in the integer array *buf*, which must have at least *length* entries.

See Also **GetLUT**

SetOffset

Syntax `int SetOffset(FGHANDLE fgh, int offset, int flags);`

Input Values An integer from -128 to 127.

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

Description Sets the video offset for the specified frame grabber. For more information, see *Setting Video Offset and Gain*, on page 55.

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 70.

SetStrobePeriods

Syntax `int SetStrobePeriods(FGHANDLE fgh, int t1, int t2, int t3);`

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

Description Sets the length of the pulses and the length of the gap between pulses for the two strobe lines. All times are in multiples of the horizontal scan frequency (63.5 microseconds for NTSC and 64 microseconds for CCIR/PAL). *t1* is the length of the strobe 0 pulse; *t2* is the length of the gap between pulses; *t3* is the length of the strobe 1 pulse. The maximum value for each of the three parameters is 65,535, even though *int* variables are 32 bits long in DOS/4GW, Windows_95, and Windows_NT

For more information, see *Strobes*, on page 59.

See Also **FireStrobe, GetStrobeState, SetStrobePolarity, SetStrobeType**

SetStrobePolarity**Syntax** int SetStrobePolarity(FGHANDLE fgh, int strobe, int polarity);**Return Value** Non-zero if successful.
0 on failure.**Description** Sets the polarity for the specified strobe line.:

0 = active high.

1 = active low.

For more information, see *Strobes*, on page 59.**See Also** **FireStrobe, SetStrobePeriods, SetStrobeType**

SetStrobeType**Syntax** int SetStrobeType(FGHANDLE fgh, int mode);**Return Value** Non-zero if successful.
0 on failure.**Description** Controls the mode of action for the strobe lines. The mode parameter can accept the following values:

Value	Description
STROBE_NORMAL	The FireStrobe() function controls the strobes.
STROBE_OFF	Disables (tri-states) strobe lines. This is the default.
STROBE_TRIG	A complete strobe sequence (strobe 0, gap, strobe 1) is initiated each time a trigger is detected.

For more information, see *Strobes*, on page 59.**See Also** **FireStrobe, GetStrobeState, SetStrobePolarity, SetStrobeType**

Imagination

SetTriggerType

Syntax `int SetTriggerType(FGHANDLE fgh, int type);`

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

Description Specifies the TTL-level trigger signal the frame grabber's trigger detection hardware is expecting. The possible values of *type* are:

Value	Description
LOW	Trigger occurs whenever the trigger input is near 0 volts.
HIGH	Trigger occurs whenever the trigger input is near 5 volts.
RISING	Trigger occurs whenever the input goes from low to high.
FALLING	Trigger occurs whenever the input goes from high to low.
DEBOUNCE	Input must return to the inactive state (low, for rising edge, high for falling edge) for at least one video field before another trigger will be detected.

The RISING and FALLING options can be combined with the DEBOUNCE flag using the bitwise OR operator; for example, RISING | DEBOUNCE. If the DEBOUNCE flag is set, the input must return to the inactive state (low, for rising edge, high for falling edge) for at least one video field before another trigger will be detected. This is useful for preventing switch bounce and other noise sources from being detected as trigger signals. You can't use the DEBOUNCE flag alone; it must be used in combination with RISING or FALLING.

SetTriggerType() waits for the frame grabber to finish processing any queued operations before executing.

See Also **GrabTriggered, GetTriggerType**

SetVideoFormat

Syntax `int SetVideoFormat(FGHANDLE fgh, int field_length, int blank_length, int flags);`

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

Description Specifies the synchronization mode and tells the frame grabber to expect a specific number of lines of vertical blank (*blank_length*), followed by a specific number of lines of valid video (*field_length*).

The *flags* parameter specifies the synchronization mode. The possible values of *flags* are:

Value	Description
AUTOMATIC_SYNC	Default. The board automatically detects the video format (NTSC or CCIR/PAL) and synchronizes to the incoming video signal. <i>field_length</i> and <i>blank_length</i> are ignored.
INTERNAL_SYNC	The board generates its own video timing and ignores sync information in the incoming video signal. With this mode, you can use the board's sync drive signals to synchronize the video source with the board.
USER_SYNC	The video timing is entirely specified by the <i>video_length</i> and <i>blank_length</i> . Even fields have vertical blanks of length <i>blank_length</i> ; odd fields have vertical blanks of length <i>blank_length</i> + 1.
WEN_SYNC	The board synchronizes vertically to the WEN signal received on the trigger input. The board still uses the horizontal sync in the incoming video signal.

Value	Description
SINGLE_FIELD_SYNC	(PX610 only) The board treats every field of video received as if it were field 0. You'll typically use this mode with non-interlaced video. This flag must be combined with one of the flags WEN_SYNC, INTERNAL_SYNC, or USER_SYNC using the bitwise OR operator. When combined with the INTERNAL_SYNC flag, the frame grabber generates timing internally as if each field is field 0, and this timing is reflected in the behavior of the sync drive signals.

Valid values for *blank_length* depend on the sync mode:

Sync Mode	Range for <i>blank_length</i>
USER_SYNC and INTERNAL_SYNC	$14 \leq \textit{blank_length} \leq 256$
WEN_SYNC	$1 \leq \textit{blank_length} \leq 256$

Valid values for *field_length* are:

Frame Grabber	Range for <i>field_length</i>
PX510	$1 \leq \textit{field_length} \leq 288$, (288 is the length of a standard CCIR/PAL video field and is longer than a standard NTSC video field.)
PX610	$1 \leq \textit{field_length} \leq 32,767$

When you need to change the video format and the image size for a subsequent capture, always call `SetVideoFormat()` first, and then call `SetImageSize()` (or `SetFieldSize()`) before capturing the image.

SetVideoFormat() waits for the frame grabber to finish processing any queued operations before executing.

For more information, see *Grabbing Images with Non-Standard Video Formats*, on page 52.

See Also **GetFieldLength, GetSyncType**

VideoType

Syntax int VideoType(FGHANDLE fgh);

Return Value 0 No video.
 1 NTSC video.
 2 CCIR/PAL video.
 3 Other.

Description Returns the type of video signal connected to the frame grabber: North American NTSC format, European CCIR/PAL format, or other.

Wait

Syntax int Wait(FGHANDLE fgh, int flags);

Return Value Non-zero 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.

Imagination

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 70.

See Also **WaitVB**

WaitFinished **Windows NT only**

Syntax int WaitFinished(int 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 more information, see *Programming in a Multithreaded, Multitasking Environment*, on page 41 and *Queue Structure under Windows NT*, on page 67.

See Also **IsFinished**

WaitVB

Syntax int WaitVB(FGHANDLE fgh);

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

Description Waits until the end of 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 afterwards to capture the field that has just begun.

See Also **Wait**

WriteBin**Syntax** `int WriteBin(FRAMEHANDLE frh, char *fname, int 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 handle is invalid, or the frame's image data has no logical address.

Description Writes the image in frame buffer *frh* to the file *fname*. No information about the image (height, width, and bits per pixel) is written, only the pixel values. If *fname* already exists and *overwrite* is zero, the function returns an error; otherwise, the contents of *fname* are overwritten. WriteBin() opens and closes the file *fname*.

When calling this function from a Windows program, the type of *fname* should be LPSTR rather than char *.

See Also **ReadBin**

Imagination

WriteBMP

Syntax int WriteBMP(FRAMEHANDLE frh, char *fname, int 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 handle is invalid, or the frame's image data has no logical address.
WRONG_BITS	The number of bits in the specified frame is not 8.

Description Writes the image stored in frame buffer *frh* to the file *fname* in the BMP format. If *fname* already exists and *overwrite* is zero, the function returns an error; otherwise, the contents of *fname* are overwritten. WriteBMP() opens and closes the file *fname*.

When calling this function from a Windows program, the type of *fname* should be LPSTR rather than char *.

See Also **ReadBMP**

VESAMENU Library

6

The VESAMENU 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. Imagination used this library to create the interface for PCIVU and for most of the DOS sample programs.

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

vmenu_lb.lib. Turbo, version 3.0 and later and Borland, version 3.1 and later.

vmenu_lm.lib. Microsoft, version 6.0 and later.

vmenu_fw.lib. Watcom DOS/4GW version 10.6 and later.

The library provides functions for the following purposes:

- Entering, configuring, and exiting graphics mode
- Text display and configuration
- Menu creation, configuration, display, and manipulation
- Bit-mapped image display
- Editing text strings

In order to use this VESAMENU library, your video card and monitor must be VESA-compatible and capable of supporting VESA mode 105 hex, which is a 1024x768, 8-bit (256-color) mode.

Initializing and Exiting the Library

Before you call any other VESAMENU functions, you must call **vg_init_graph()**. This function saves the current display mode, sets the display to VESA mode 105 hex, and initializes some global data structures.

vg_exit_graph() resets the display mode to the mode that was active before the call to **vg_init_graph()**. A program must not call **vg_exit_graph()** until after all other VESAMENU functions have been called.

If your program calls **vg_init_graph()** more than once before calling **vg_exit_graph()**, **vg_exit_graph()** will not be able to restore the display mode that was active before the first call of **vg_init_graph()**.

VGA Text and Image Display

The basic functions this library provides for displaying text are **vg_gotoxy()**, which places an invisible cursor at the specified screen location, and **vg_print()**, which displays a string at the cursor location. The text font, foreground color, background color, and size are all configurable. All text and images use the same display palette, as defined by **vga_set_palette()**.

The library provides four different graphics operations:

- **draw_image()**—display a bitmap
- **draw_scaled_image()**—display a scaled bitmap
- **draw_rectangle()**—draw a rectangle
- **fill_rectangle()**—draw a filled rectangle

The bitmaps are assumed to have one byte per pixel, and to be no wider than 1,024 pixels and no taller than 768 pixels.

In normal operation, a VGA card is addressed through a 64 KB window located at memory segment 0xA000. To access different parts of the VGA memory, pages must be swapped in and out of that window. (Some VGA cards have a special *flat* addressing mode in which the entire memory is addressed without paging. However, each card does this differently, so the library uses the normal addressing mode.) The **vg_gotoxy()** function automatically sets the page to that portion of the VGA memory that is closest to the specified location. The **vg_print()** function and the graphics functions automatically increment the page when they approach a page boundary.

Menu Creation, Configuration, and Display

A menu is a data structure whose contents can be manipulated and displayed using the **menu_select()** and **menu_display()** functions. All menus must be successfully initialized by the **menu_generate()** 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 **menu_generate()** is called. For more information, see *Menu Structure*, on page 125 and *menu_generate*, on page 135.

The **menu_select()** 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

Colors Structure

```
struct colors
extern struct tagcolors
{
    unsigned char standard, standardbk;
    unsigned char high, highbk;
    unsigned char menu, menubk;
    unsigned char help, helpbk;
} colors;
```

This structure defines the foreground and background colors (VGA palette indices) used by the menu functions to display the different types of text:

- *menu* and *menubk* colors are used to display non-highlighted menu options.
- *high* and *highbk* colors are used to display the highlighted menu options.
- *help* and *helpbk* colors are used to display the single-line help messages at the bottom of the screen.
- *standard* and *standardbk* colors are used to display all other menu functions.

Menu Structure

```

struct menu
typedef struct tagmenu
{
    int xmin, ymin, dx, dy; /* Screen location and size */
    int rows, cols;        /* Display configuration of
                               items */

    int numitems;          /* Number of items */
    char *title;           /* Title (displayed at top of
                               menu)*/

    int highlight;         /* Currently selected item */
    menuitem *data;        /* Pointer to menu items */
} menu;

```

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

- (*xmin*, *ymin*) defines the upper left-hand corner on the screen where the menu will be drawn
- (*dx*, *dy*) defines its height and width.
- *rows* and *cols* define the number of rows and columns in which the menu items will be organized and displayed; these values are set by the `menu_generate()` function.
- *numitems* defines the number of items in the menu.
- **title* points to the title, if any, of the menu; a menu that doesn't have a title must initialize this pointer to `NULL`.
- *highlight* defines which of the menu items is currently selected; if this value is not initialized, it will be set to 0 (the first item) by `menu_generate()`.
- **data* points to the **menuitem** structures below, and is usually set to point to an array.

Menuitem Structure

```
struct menuitem
typedef struct tagmenuitem
{
    int xoff, yoff; /* Screen, location, relative to menu */
    int i, j;      /* Position in menu (row, column) */
    char *text;    /* Menu text string */
    int hotkey;    /* Optional specifier for a hotkey;
                    0 indicates none */
    char *help;    /* Help message string */
} menuitem;
```

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

- *(xoff, yoff)* defines the item's display coordinates relative to the menu's upper left-hand corner; these values are set by `menu_generate()`.
- *(i, j)* defines the item's (row, column) coordinates in the menu display; these values are set by `menu_generate()`.
- **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 0.
- **help* defines the text string that will be displayed at the bottom of the screen when this item is selected; it should describe the functioning of this item.

Function Reference

VESA and VGA Functions

vg_exit_graph

Syntax `void vg_exit_graph(void);`

Return Value None.

Description Resets the VESA display to the mode it was in just before `vg_init_graph()` was called. A program must not call `vg_exit_graph()` until after all other VESAMENU functions have been called.

See Also **vg_init_graph**

vg_init_graph

Syntax `char *vg_init_graph(void);`

Return Value A pointer to an error message.
NULL if graphics mode was enabled successfully.

Description This function does the following:

- Saves the current display mode for later restoration
- Sets the display mode to VESA mode 105 hex (1024x768, 256-colors)
- Sets up a 64 gray-scale VGA palette
- Initializes the font type and size to an 8x16 font, standard size
- Initializes the *colors* structure

It must be called before any other VESAMENU library function.

See Also **vg_exit_graph, vga_set_palette**

Imagenation

vga_set_palette

Syntax	<code>void vga_set_palette(unsigned char far palette[256*3]);</code>
Input Values	An array containing 256 byte triplets in the format: red value, green value, blue value.
Return Value	None.
Description	When the frame grabber displays an image on the VGA monitor, each of the 256 grayscale values is mapped to a VGA palette value. This function allows the application to specify the VGA palette. The default palette, specified by <code>vg_init_graph()</code> , has 64 grayscale values, rather than 256, because there can be no more than 64 values of any given color. Changing the VGA palette shouldn't normally be necessary, but can be done to create false-color displays.
See Also	<code>vg_init_graph</code>

vga_wait_vb

Syntax	<code>void vga_wait_vb(void);</code>
Return Value	None.
Description	Waits until the beginning of the VGA display's vertical blank period. This function is useful for forcing graphics displays to synchronize to the VGA monitor.

VESA Text Functions

vg_getbkcolor

Syntax	<code>int vg_getbkcolor(void);</code>
Return Value	0-255 The current VGA text display background color. -1 Background is transparent.

Description Returns the palette index of the color `vg_print()` uses for the text background. A value of -1 indicates that `vg_print()` will not draw a background around the text.

See Also `vg_setbkcolor`, `vg_print`

`vg_getcolor`

Syntax `int vg_getcolor(void);`

Return Value 0-255 The current VGA text display foreground color.

Description Returns the palette index of the color `vg_print()` uses to draw text.

See Also `vg_setcolor`, `vg_print`

`vg_gotoxy`

Syntax `unsigned char far *vg_gotoxy(int x, int y);`

Return Value A pointer to the screen location specified by (x, y) .

Description Places the invisible cursor at location (x,y) as specified in pixels. It sets the VGA page such that the specified position is as close as possible to the top of the page without being above it, subject to the granularity with which the page can be set. x and y are clipped to the ranges $[0, \text{vg_maxx}())$ and $[0, \text{vg_maxy}())$ respectively.

Description The location of the cursor specifies where `vg_print()` will print.

See Also `vg_maxx`, `vg_maxy`, `vg_print`

`vg_maxx`

Syntax `int vg_maxx(void);`

Return Value The maximum x value in the currently active graphics mode.

Imagination

Description This implementation of the VESAMENU library hardwires the graphics mode to VESA mode 105 hex (a 1024x768, 256-color mode). Therefore, the maximum x value is 1023.

See Also `vg_maxy`, `vg_gotoxy`

`vg_maxy`

Syntax `int vg_maxy(void);`

Return Value The maximum y value in the currently active graphics mode.

Description This implementation of the VESAMENU library hardwires the graphics mode to VESA mode 105 hex (a 1024x768, 256-color mode). Therefore, the maximum y value is 767.

See Also `vg_maxx`, `vg_gotoxy`

`vg_print`

Syntax `void vg_print(char *st);`

Return Value None.

Description Writes text to the screen at the cursor location specified by the most recent call to `vg_gotoxy()` and in the colors specified by `vg_setcolor()` and `vg_setbkcolor()`. This function does no string formatting; use the standard library function `sprintf()` if you need to format the string.

When `vg_print()` finishes, the x cursor location is changed to the first column after the end of the displayed text, and the y cursor location remains the same.

Caution

This function does not check to make sure that text won't run off the right or bottom edges of the screen. If you tell `vg_print()` to print text that is too wide or placed too low, the resulting display

will probably be in unexpected places, and the final cursor location will not be accurate.

See Also `vg_gotoxy`, `vg_setcolor`, `vg_setbkcolor`

vg_resizefont

Syntax `void vg_resizefont(int size);`

Return Value None.

Description Sets the scale factor for text drawn by `vg_print()`. Text can be scaled by any integer ≥ 1 . The parameter *size* sets the scale factor, which is 1 by default.

See Also `vg_setfont`, `vg_print`

vg_setbkcolor

Syntax `void vg_setbkcolor(int color);`

Return Value None.

Description Sets the palette index of the color `vg_print()` uses for the text background. If color is -1, `vg_print()` will not draw a background around the text.

See Also `vg_getbkcolor`, `vg_setcolor`, `vg_print`

vg_setcolor

Syntax `void vg_setcolor(int color);`

Return Value None.

Description Sets the palette index of the color `vg_print()` uses to draw text.

See Also `vg_getcolor`, `vg_setbkcolor`, `vg_print`

Imagination

vg_setfont

Syntax `int vg_setfont(int font);`

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

Description Sets the font used by `vg_print()`. There are three fonts available: 8x8, 8x14, and 8x16. The fonts are numbered 1, 2, and 3 respectively. The default (set by `vg_init_graph()`) is the 8x16 font.

`vg_setfont()` doesn't affect the font scale factor set by `vg_resizefont()`. This function will fail if it is passed any number other than 1, 2, or 3.

See Also `vg_print`, `vg_init_graph`, `vg_resizefont`

vg_sizex

Syntax `int vg_sizex(void);`

Return Value The width of a text character as printed by `vg_print()`.

Description Returns the width in pixels of a text character, taking into account the font scale factor as set by `vg_resizefont()`.

See Also `vg_sizey`

vg_sizey

Syntax `int vg_sizey(void);`

Return Value The height of a text character as printed by `vg_print()`.

Description Returns the height in pixels of a text character, taking into account the font scale factor as set by `vg_resizefont()`.

See Also `vg_sizex`

vg_wherex

Syntax	<code>int vg_wherex(void);</code>
Return Value	The horizontal position of the cursor used by <code>vg_print()</code> .
Description	Returns the horizontal position, in pixels, of the cursor as set by <code>vg_gotoxy()</code> .
See Also	<code>vg_gotoxy</code>, <code>vg_print</code>

vg_wherey

Syntax	<code>int vg_wherey(void);</code>
Return Value	The vertical position of the cursor used by <code>vg_print()</code> .
Description	Returns the vertical position, in pixels, of the cursor as set by <code>vg_gotoxy()</code> .
See Also	<code>vg_gotoxy</code>, <code>vg_print</code>

Menu Functions

get_key

Syntax	<code>int get_key(void);</code>
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 listed in VMENU.H. The <code>menu_select()</code> function uses some of these special keys, so it should take its input from <code>get_key()</code> .
See Also	<code>menu_select</code>

Imagination

menu_calc_dx

Syntax	<code>int menu_calc_dx(menu *m, int columns);</code>
Return Value	The calculated menu width.
Description	Calculates the width in pixels that the menu <i>m</i> should be if its items are arranged in a number of columns equal to <i>columns</i> . This calculation is based on the width of each menu item and the width in pixels of the text (as defined by <code>vg_setfont()</code> and <code>vg_resizefont()</code>).
See Also	menu_calc_dy, menu_generate

menu_calc_dy

Syntax	<code>int menu_calc_dy(menu *m, int columns);</code>
Return Value	The calculated menu height.
Description	Calculates the height in pixels that the menu <i>m</i> should be if its items are arranged in a number of columns equal to <i>columns</i> . This calculation is based on the number of items and the height in pixels of the text (as defined by <code>vg_setfont()</code> and <code>vg_resizefont()</code>).
See Also	menu_calc_dx, menu_generate, struct menu

menu_display

Syntax	<code>void menu_display(menu *m);</code>
Return Value	None.
Description	Displays menu <i>m</i> on the VGA screen at the location specified by the <i>x</i> and <i>y</i> 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.
See Also	menu_erase, struct menu

menu_erase**Syntax** void menu_erase(menu *m);**Return Value** None.**Description** Erases the menu *m* from the VGA display by calling fill_rectangle(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.**See Also** **menu_display, struct colors, struct menu**

menu_generate**Syntax** int menu_generate(menu *m);**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 it to function properly, several items in the menu structure must be initialized before menu_generate() 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 menu_generate() function assumes that all menu item names have the same number of characters. It calculates the number of rows in the menu display 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 the number of items. The `menu_generate()` function will fail under the following circumstances:

- The menu coordinates are off-screen.
- The menu is too wide to fit on the screen with the given origin.
- 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 `menu_generate()` should always be checked for errors before menu *m* is used with any other VESAMENU function.

See Also **struct menu**

menu_select

Syntax `int menu_select(menu *m, int key);`

Return Value Return values are:

Return Value	Description
-1	No selection made.
0 to <code>m->numitems - 1</code>	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 `menu_select()`:

- 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.

See Also `get_key`

Graphics Functions

fill_rectangle

Syntax `void fill_rectangle(int xmin, int ymin, int dx, int dy, int color);`

Return Value None.

Description Draws a *color* filled rectangle on the VGA display at location (*xmin*, *ymin*). The rectangle is *dx* pixels wide and *dy* pixels tall.

See Also `draw_rectangle`

draw_rectangle

Syntax `void draw_rectangle(int xmin, int ymin, int dx, int dy, int color);`

Return Value None.

Description Draws an unfilled *color* rectangle on the VGA display at location (*xmin*, *ymin*). The rectangle is *dx* pixels wide and *dy* pixels tall.

See Also `fill_rectangle`

draw_image

Syntax `void draw_image(unsigned char huge *buf, int xmin, int ymin, int dx, int dy);`

Return Value None.

Description Copies the information in *buf* to the VGA display at location (*xmin*, *ymin*), assuming that *buf* contains an image that is *dx* pixels wide by *dy* pixels tall.

See Also `draw_scaled_image`

Imagenation

draw_scaled_image

Syntax	<code>void draw_scaled_image(unsigned char huge *buf, int dx1, int dy1, int xmin, int ymin, int dx2, int dy2);</code>
Return Value	None.
Description	Copies the information in <i>buf</i> to the VGA display at location (<i>xmin</i> , <i>ymin</i>), assuming that <i>buf</i> contains an image that is <i>dx1</i> pixels wide by <i>dy1</i> pixels tall. The image will be scaled in the x direction by a factor of <i>dx2/dx1</i> , and in the y direction by a factor of <i>dy2/dy1</i> , so the resultant image will be <i>dx2</i> pixels wide and <i>dy2</i> pixels tall.
See Also	draw_image

Editing Functions

edit

Syntax	<code>int edit(int inserting, int len, char *prompt, char *start);</code>
Return Value	RET Changes to *start saved. ESC Changes to *start not saved.
Description	<p>Provides an interface for editing a string at a <i>prompt</i>. It displays <i>prompt</i> at the location of the cursor as placed by <code>vg_gotoxy()</code>, and modifies the string according to keyboard input until RETURN or ESC is pressed.</p> <p>If <i>inserting</i> is non-zero, <code>edit()</code> starts in insert mode; otherwise, <code>edit()</code> starts in overstrike mode. After <code>edit()</code> is called, the INSERT key toggles the mode.</p> <p><i>len</i> defines the maximum length of the editing buffer.</p> <p><i>*prompt</i> is an application-defined prompt string such as "File name: ".</p> <p><i>*start</i> is the string to be edited, which must be allocated by the application. It may have initial information in it, such as a default file extension.</p> <p>The keys defined in <code>edit()</code> are BACKSPACE, DELETE, the left and right arrow keys, HOME, END, and the alphanumeric keys.</p>

The FILEIT File Conversion Program

7

FILEIT is a stand-alone DOS program that converts Imagenation binary image files to formatted files, or formatted files to binary image files. FILEIT can be executed as a DOS command on the command line, from a batch file, or in a DOS window in Microsoft Windows.

The routines in FILEIT that perform the graphics conversions were adapted from the book *Supercharged Bitmapped Graphics* by Steve Rimmer. The source is included on a disk when you purchase the book.

The PX library can write either unformatted binary image files, typically identified by a BIN extension, or Microsoft BMP formatted files.

FILEIT can convert between BIN files and files in any of the following graphics formats:

File Extension	Description
BMP	Microsoft Windows 3.x Paintbrush bitmap format.
GIF	CompuServe graphics interchange format.
PCX	ZSoft format.
PIC	PC Paint/Pictor format.
TGA	Truevision Targa format.

File Extension	Description
TIF	Aldus tagged image file Format.
WPG	WordPerfect metafile format.

FILEIT expects to have a binary image file (BIN) as either the source or target file type or override, otherwise the type of conversion is ambiguous. If the format of a file is properly identified by its file extension, then an override parameter is not necessary.

The command line parser is not case-sensitive. Upper and lower case have the same meaning.

Syntax

```
fileit /option source.fil /ovrd target.fil /ovrd /w=n
```

/option can have the following values:

- ?** Display syntax.
- f** Force overwrite of existing target file.
- /w=n** Width of the image, where n is the number of pixels. Use this switch when converting source files in binary image format, which don't store information about the width of the image. If the source file is not a BIN file, this switch is not needed.

/ovrd is a file type override and can be any of the three-character file extensions listed in the table above.

Examples

To convert a binary image file (f1) to a TIFF file (f2):

```
fileit f1 /bin f2 /tif
```

To convert a binary image file (f1) to a TIFF file (f2.tif):

```
fileit f1 /bin f2.tif
```

To convert a binary image file (f1.bin) to a TIFF file (f2.tif):

```
fileit f1.bin f2.tif
```

To convert a binary image file (f1.bin) to a TIFF file (f1.tif):

```
fileit f1.bin *.tif
```

To convert all binary image files (*.bin) to TIFF files (*.tif):

```
fileit *.bin *.tif
```

To convert a TIFF file (f2.tif) to a binary image (f1.bin), reverse the file names in any of the above examples:

```
fileit *.tif *.bin
```

If a target file exists, you will be asked whether you want it overwritten. Target files will not be automatically destroyed. However, if you want the target files overwritten in all cases without any prompting, use the `/f` (force overwrite) option, like this:

```
fileit /f *.tif *.bin
```

Source and Target file specifications can include drives, directories, and any wild cards that DOS recognizes in file specifications. Directory specifications can be either relative or absolute. Directories and Drives cannot contain wild cards.

Absolute directory specification:

```
c:\dir\file.ext
```

Relative directory specification:

```
..\dir\file.ext
```

Return Values

FILEIT returns an integer to the calling process, usually DOS, upon termination. Non-zero return values indicate a fatal error.

All errors that occur during command line parsing are fatal. Most errors that occur during file processing are not fatal. Errors that occur during file conversion are fatal to the current conversion, but should not affect subsequent conversions if multiple files have been specified with wild cards. Non-fatal errors produce error messages, but don't stop the process.

The following errors are fatal and cause FILEIT to terminate:

Error Number	Description
201	Too many or misplaced options or overrides.
202	Too many parameters on command line.
203	Invalid command option.
204	Invalid source override.

Error Number	Description
205	Invalid target override.
206	Source file specification not found. No such file.
207	Drive and Directory specifications cannot contain wild cards.
208	Missing source file specification. Source file must be specified.
209	Missing target file specification or target file type override.
210	Type of conversion is ambiguous. Source file type not specified.
211	Type of conversion is ambiguous. Target file type not specified.
212	Type of conversion is ambiguous. Target and source file types are equal.
213	Type of conversion is ambiguous. Either target or source must be BIN.
214	Cannot generate full path names. Check for existence of directories.

The following errors are fatal to the current conversion only, and don't cause the process to terminate. These errors don't return values since they don't cause FILEIT to terminate:

- Error writing to output file. Wrong number of bytes written.
- Error reading from input file. Wrong number of bytes read.
- Cannot allocate enough memory. Remove some TSRs and try again.
- Error writing palette structure to output file.
- Wrong number of bits. Bits per pixel should be 8.
- Unexpected End of File marker. File is too short.

- Bits per pixel illegal for LZW compression. Cannot be greater than 9.
- Cannot match source file size. May not be an Imagenation binary file.
- Target and source file names are identical. Cannot do conversion.
- Cannot open target file.
- Cannot open source file.
- Unrecognized file type.
- File structure does not match command line specification.
- Conversion type is ambiguous.

Batch File Processing

When you run FILEIT from a DOS batch file, you should be aware of the following:

- To save the output of the run in a log file, the log file specification must be part of the batch file, not just a parameter on the command line that executes the batch file.
- FILEIT asks questions about overwriting files. If the answers are not provided to the batch file, the program will hang, and you'll have to reboot your computer.

In order to use batch files successfully, you'll need to use the DOS redirection operators:

Operator	Description
>	Erases a file before writing to it
>>	Appends to a file
<	Reads from a file

Examples

The examples below illustrate the use of the redirection operators.

Example 1: Converts Z1 to Z1.BMP and directs the output to a log file named LOG. If the file named LOG contains any data prior to executing this batch file, it is erased.

```
fileit z1 /bin z1.bmp >log
```

Example 2: Converts Z1 to a BMP file and converts Y1 to a PCX file. On the first line, the output is directed to a log file named LOG. On the second line the output is appended to the same log file.

```
fileit z1 /bin z1.bmp >log
fileit y1 /bin y1.pcx >>log
```

In both examples 1 and 2, if FILEIT asks for permission to overwrite an existing file, it will hang, and you'll need to reboot your computer. You can solve this problem in one of two ways.

- Use the /f option to force overwriting and avoid the question.

```
fileit /f z1 /bin z1.bmp >log
fileit /f y1 /bin y1.pcx >>log
```

- Supply the answers in two separate files. Create a file called YES that contains the letter “Y”, and create a file called NO that contains the letter “N”. Then rewrite the above batch file as follows:

```
fileit z1 /bin z1.bmp <yes >log
fileit y1 /bin y1.pcx <no >>log
```

When FILEIT asks for permission to overwrite an existing file, it does not wait for a response from the terminal, instead it takes the answer from the first line of the file name that follows the “<” operator. All terminal

output including the “yes” and “no” answers will be directed to the file named LOG.

If the above command line included wild cards, then you would have to include a “yes” and “no” answer for each file. Suppose you have 10 files beginning with the letter “Z”, and you want to convert all of them to BMP files using a batch file. You also want any existing BMP files beginning with a “Z” to be overwritten. Your batch file would contain the following line:

```
fileit z* /bin z*.bmp <yes >log
```

The file named YES would have to contain 10 Ys each on a separate line. Each time FILEIT needs to ask for permission to overwrite a file, it reads the next line of YES. The file named YES can always contain more Ys than are required, but no fewer Ys. If you know you’re going to be converting between 50 and 100 files at any given time, you could create a YES file with 200 Ys in it, and it would always be sufficient.

Notes on Format Conversions

FILEIT was not designed to read and write all variations of formatted image files. Formatted files, such as PCX, BMP, and WPG, have many variations, and new ones are being created all the time. FILEIT reads and writes only a subset. The way FILEIT handles each of the formats is described in the following sections.

BMP Files

BMP files are becoming a standard among Windows 3.x applications. The Windows implementation of ZSoft’s PC Paintbrush uses BMP files. If you need to import an image file into a Windows application, try using FILEIT to convert the file to BMP format. The only problem you might encounter is that some Windows applications expect to operate on 16-color BMP files, like the ones distributed with Windows as wallpaper. PX

image files are 8-bit grey scale which is equivalent to 256-color, so your application needs to be able to handle the 256-color palette. Conversions from BMP to binary only work for BMP files for images whose width is divisible by four. If BMP files do not work well with your application, try TIFF. Many word processors are capable of reading TIFF files.

GIF Files

FILEIT creates GIF 87a files. There is a newer, more complex format, GIF 89a, that allows multiple images within a single file. However, GIF 89a is less compatible with existing applications. An application capable of reading GIF 89a format should be capable of reading GIF 87a format.

PCX Files

The PCX format was developed by ZSoft as the native format for PC Paintbrush for Windows 3.x. Because of the popularity of PC paintbrush, PCX files have become standard among PC applications, including desktop publishing, drawing applications, and many others. The PCX files produced by FILEIT are 8-bit files, and are compressed using runlength encoding.

PIC Files

There are many applications that use PIC as a file extension. Lotus 1-2-3 uses it to denote graph files. Lotus PIC files are vector drawings rather than bitmapped graphics. FILEIT will not convert a Lotus PIC file back to a binary file; it will give you an error message.

The PIC files created by FILEIT are PC Paint/Pictor format files. The PIC format used by FILEIT was originally the proprietary format of PC Paint 2.0. This format is not the same as ZSoft's PC Paintbrush. The Windows 3.x implementation of ZSoft's PC Paintbrush uses BMP files.

TGA Files

The Targa format was originally created for the Truevision Targa board, a high-end video board for PCs. The format has since migrated to many other applications. Although the Targa format was originally created to handle 24-bit color, the file that FILEIT produces is 8-bit, with a palette and no image compression.

TIFF Files

The TIFF (Tagged Image File Format) file was designed to be very flexible so it's quite complicated. There is more than a good chance that a given TIFF reader will not be able to read a given TIFF file. For that reason, FILEIT produces a fairly simple TIFF file: an 8-bit file with no image compression. The lack of compression seems to make the file compatible with most applications. Many word processors are able to read TIFF files.

WPG Files

WPG files are used by WordPerfect. They were intended for vector images not bitmapped images, but they can also contain bitmapped images. PX binary image files use 8-bit grey scale. The WPG format allows 8-bit images, however, at present WordPerfect only reads images of four or fewer bits. If you use FILEIT to create a WPG file, the file will be created properly, but WordPerfect won't be able to read it. If you need to import an image into a WordPerfect document, use the BMP format.

PX500 Compatibility

A

This appendix describes features of the PX510 and PX610 that were not available on the PX500, and changes in this version (2.0) of the PX libraries for supporting these new features. You'll find this information useful if you want to upgrade a PX500-based system to use either the new PX510/PX610 hardware, or the new version of the PX software libraries, or both.

New Features in the PX510 and PX610

Non-Interlaced, Progressive-Scan Video Support

The PX500 and PX510 can capture images from video sources in two interlaced video formats: NTSC and CCIR/PAL. In addition to supporting these interlaced formats, the PX610 can capture images from video sources that output *non*-interlaced video, which is the most common video format used by progressive-scan cameras. For more information on non-interlaced, progressive-scan video, see *Grabbing Images with Non-Standard Video Formats*, on page 52, and *Video Format*, on page 62.

Horizontal and Vertical Cropping and Scaling

The PX500 can scale images horizontally and crop images vertically. The new PX frame grabbers can crop and scale images both horizontally and vertically, and the scaling feature is more versatile. For more information on cropping and scaling, see *Scaling Images*, on page 71 and *Cropping Images*, on page 73.

Horizontal and Vertical Sync Drive Signals

The new PX frame grabbers have two sync lines that you can use to output TTL-level synchronization signals. The PX500 did not have the ability to output sync signals. For more information, see *Synchronization Drive Signals*, on page 60.

Programmable Strobe Lines

The PX500 has two output strobe lines, and you can control the polarity of the output signals. The new PX frame grabbers also have two strobe lines with polarity control, but on the new frame grabbers you can program the duration of the strobe periods, start the strobe output on command or on receipt of a trigger signal, and fire all or only part of the strobe cycle. For more information, see *Strobes*, on page 59.

Full-Size (768x576) CCIR/PAL Images

Resolution on the PX500 was limited to 640 x 512. The new frame grabbers have a maximum resolution of 768 x 576, allowing you to capture the entire vertical range of a CCIR/PAL image with an aspect ratio that results in square pixels in the captured image.

+12V Power Line

The new frame grabbers include a +12V output on the 26-pin connector. The PX500 did not offer a +12V output.

26-Pin D Connector

The PX500 used a 15-pin D connector to provide connections for the four video inputs, the strobe lines, and the trigger line. To accommodate the extra sync drive signals, the +12V output, and the accompanying ground signals, the new frame grabbers use a 26-pin connector. Pinouts for the connector are listed in *26-pin D Connector*, on page 156.

If you already have a 15-pin cable for use with the PX500, you can get an adapter that allows you to use your 15-pin cable with the 26-pin connectors on the PX510 and PX610. For more information, contact Imagenation Technical Support (see *Technical Support*, on page 29).

Changes in the PX Libraries

This section describes changes in the PX libraries from version 1.x that shipped with the PX500, to version 2.x that ships with the PX510 and PX610.

New Strobe Functions

In version 1.x of the PX libraries, strobe control was limited to changing the polarity of the strobe lines with the `SetStrobePolarity()` function. Version 2.0 includes the four additional functions listed below for use with the PX510 and PX610. None of these functions has any effect when used with the PX500.

FireStrobe()—Initiates part or all of the strobe sequence, or turns off all strobe activity.

GetStrobeState()—Tells whether the strobes are disabled, whether a strobe sequence is in progress, and the current state of each strobe line.

SetStrobePeriods()—Controls the length of the two strobe pulses and the gap that separates the pulses.

SetStrobeType()—Sets the strobe to be initiated by either the `FireStrobe()` function or by an incoming trigger signal. Can also be used to disable the strobe lines. The strobe lines must be enabled on the PX510 and PX610 before the polarities can be changed.

For more information on using these strobe functions, see *Strobes*, on page 59, and the function reference pages in [Chapter 4, *Programming PX Frame Grabbers*](#), on page 35.

New Sync Drive Signal Functions

Version 2.0 of the PX library includes two new functions for control of the sync drive signal lines that were added on the PX510 and PX610. Calls to these functions will fail when used with the PX500.

SetDrivePolarity()—Controls the signal polarity of the sync signal lines.

SetDriveType()—Enables or disables both sync signal lines.

New Video Format Functions

Version 2.0 of the PX library includes three new functions for working with non-standard video signals.

GetFieldLength()—Returns the field length of the last video field, including vertical blank. `GetFieldLength()` works with both the PX500 and the PX510/PX610.

GetSyncType()—Returns the currently selected synchronization mode, as set by `SetVideoFormat()`. On the PX500, `GetSyncType()` always returns `AUTOMATIC_SYNC` as the mode.

SetVideoFormat()—Specifies the synchronization mode for the frame grabber and the length of the video field and vertical blank. On the PX500, `SetVideoFormat()` can set the synchronization mode only to `AUTOMATIC_SYNC`.

Other New Functions

Version 2.0 of the PX libraries also contains these two additional functions, both of which work with either the PX500 or the PX510/PX610.

ReadConfiguration()—Returns information about the configuration of the board, including whether video cache RAM is installed, whether the form factor is PC/104 Plus, whether the board supports non-interlaced video, and whether the board is a PX500 or a PX510/PX610.

SetFieldSize()—Like `SetImageSize()`, `SetFieldSize()` specifies the scaling and cropping of the captured image, but `SetFieldSize()` is specifically designed for single-field captures.

Changes to Existing Version 1.x Functions

Version 2.0 of the libraries contains several changes in the behavior of functions that were implemented in the Version 1.x libraries:

Grab(), GrabContinuous(), and GrabTriggered()—In Version 1.x of the PX libraries, you could only capture an image to the onboard

video cache RAM by using the `GrabToCache()` and `CacheTriggered()` functions. In addition, the Version 2.0 libraries let you use the new `CACHE` flag with the `Grab()`, `GrabContinuous()`, and `GrabTriggered()` functions to simultaneously capture an image to both a frame and to the onboard cache.

HaveCache()—In Version 1.x of the PX libraries, the `HaveCache()` function simply indicated whether video cache RAM was present on the board. In the Version 2.0 libraries, `HaveCache()` returns the amount of cache installed, in terms of the number of lines of video that can be stored.

VideoType()—In Version 2.0 of the PX libraries, `VideoType()` can return “other” for video type, in addition to the two video types supported in Version 1.x: NTSC and CCIR/PAL.

Cables and Connectors

B

This chapter includes information on purchasing and making cables for the PX family of frame grabbers.

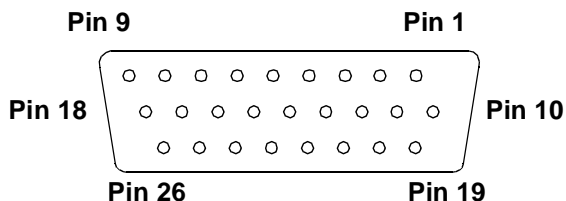
Standard PCI Bus and CompactPCI Bus Cables

Imagination offers pre-wired cables for the standard PCI-bus and CompactPCI-bus configurations of the PX510 and PX610. The cables bring the 26-pin D connector out to nine BNC connectors for the four video inputs, the sync drive outputs, the strobe outputs, and the trigger input, plus a tenth wire for the +12 Volts DC.

You can also make your own cables using the pinout information in the next section.

26-pin D Connector

Pinouts for the 26-pin D connector on the PX510 and PX610 are shown below:



Pin	Description	Pin	Description
1	Video 0	14	Ground*
2	Video 1	15	Trigger/WEN
3	Video 2	16	Reserved
4	Video 3	17	Reserved
5	Reserved	18	Reserved
6	Horizontal Drive	19	Reserved
7	Vertical Drive	20	Reserved
8	Ground*	21	Reserved
9	+12 V DC	22	Reserved
10	Ground (Video 0)	23	Strobe 0
11	Ground (Video 1)	24	Strobe 1
12	Ground (Video 2)	25	Reserved
13	Ground (Video 30)	26	Reserved

* Grounds on pins 8 and 14 are for all digital signals: horizontal and vertical drive, strobes 0 and 1, and trigger/WEN.

Connecting the +12V Output

To activate the +12V output on pin 9, 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.

26-Pin to 15-Pin Adapter for PX500 Cables

If you already have a 15-pin cable for use with the PX500, you can get an adapter that allows you to use your 15-pin cable with the 26-pin connectors on the PX510 and PX610. For more information, contact Imagenation Technical Support (see *Technical Support*, on page 29).

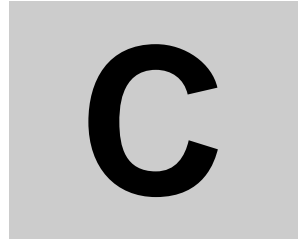
PC/104-Plus Cables

Connector J5 on the PC/104-Plus configuration is a 20-pin IDC male connector with the following pinouts:

Pin	Description	Pin	Description
1	Ground	2	Video In 0
3	Ground	4	Video In 1
5	Ground	6	Video In 2
7	Ground	8	Video In 3
9	Ground	10	Vertical Drive
11	Ground	12	Horizontal Drive
13	Ground	14	Strobe 0
15	Ground	16	Strobe 1
17	Ground	18	Trigger/WEN
19	Ground	20	+12 Volts

The ground pin for each signal is shown on the same line in the table.

Hardware Specifications



This appendix lists specifications for the PX510 and PX610 hardware.

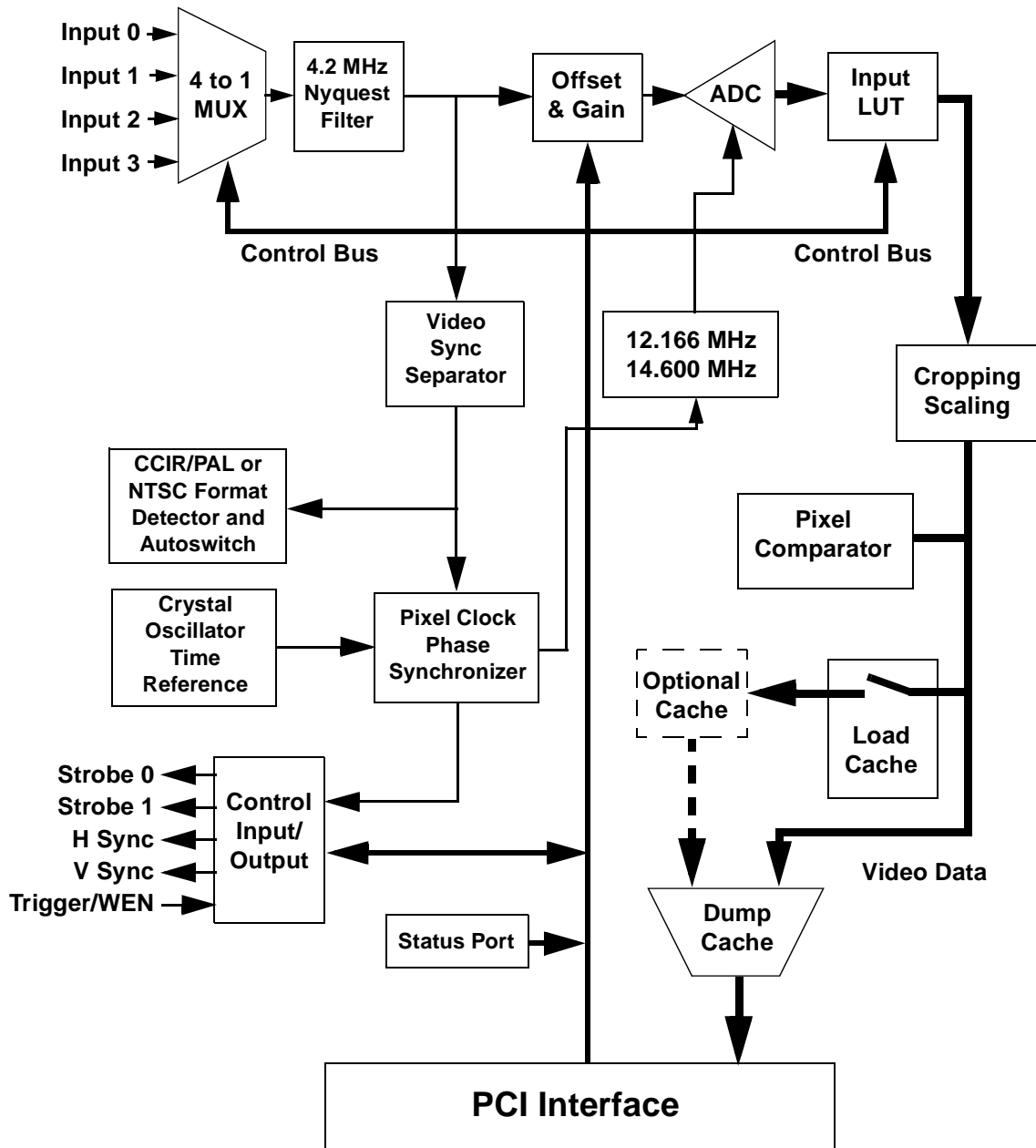
Input composite video format	Monochrome, RS-170 (NTSC) or CCIR/PAL with auto-detect. Non-interlaced formats typical of progressive-scan cameras (PX610 only).
Input video signal	1 V peak-to-peak, 75 Ohm. Diode clamped to ± 1.2 V.
Resolution	RS-170: 640 x 480 pixels (maximum: 768 x 486 pixels). CCIR/PAL: 768 x 576 pixels. 256 gray levels (8 bits).
Sampling jitter	Maximum of ± 3 ns relative to horizontal synchronization.
Capture time	Real-time video capture. RS-170 (NTSC): 1/30 second per frame. CCIR/PAL: 1/25 second per frame.
Look-up tables (LUTs)	Software-programmable, 256-byte input LUT.

External trigger	Input pulled up by 10 KOhm to 5 V. Trigger requires a TTL pulse of 100 ns minimum. Software programmable edge or level sensitivity and polarity.
Strobe output	Two TTL outputs with independently-programmable pulse widths and polarities to control resettable cameras, exposure time, strobe lights, etc.
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 module: 100 x 160 mm 3.94 x 6.3 in.
Video noise	≤ 0.7 LSB (least significant bit).
Power	+5 VDC, 650 mA.
Camera power	+12 VDC output, 1.5 A maximum, fused.
Video multiplexer	Four video inputs which can be a mix of RS-170 (NTSC) and CCIR/PAL.
Camera genlocking	Horizontal and vertical drive outputs to genlock camera with frame grabber.
File formats	Binary conversion program allows creation of BMP, TIFF, GIF, PIC, PCX, TGA, and WPG files.
Operating temperature	0° C to 60° C.
Warranty	One-year limited parts and labor.

Block Diagram



A block diagram of the PX board is shown on the following page.



PCI Bus System Performance



A PCI frame grabber should, in theory, be able to deliver real-time video, 30 frames per second with an NTSC input and 25 frames per second with a CCIR input, to the computer's main memory. In practice, the actual rate of transfer is dependent on the motherboard and the chip set used on the motherboard. Apparently, not all PCI buses are equal.

There are two considerations here:

- How many video frames can be transferred from the frame grabber to main memory per second.
- How many video frames can be displayed on the VGA per second.

The VGACOPY.EXE program included with the PX software measures both of these activities.

VGACOPY Measurements

VGACOPY is a DOS program and must be run under DOS, not in a DOS window within Windows.

VGACOPY provides the following measurements:

- 1** It determines whether or not you can achieve a rate of 30 frames per second from the frame grabber to main memory.
- 2** It measures the speed of copying a static memory buffer to the VGA card.
- 3** It measures the number of frames per second that can be displayed on your VGA display when there is only one frame being copied across the PCI bus at a time. This measurement grabs a frame to main memory and then copies it to the VGA card.
- 4** It measures the number of frames per second that can be displayed on your VGA display when there are multiple frames being copied across the PCI bus at the same time. This measurement puts the frame grabber into continuous acquire mode and copies each frame to the VGA as soon as it is acquired. This process fails completely on some systems. The failure is related to the motherboard, the chip set on the motherboard, and the VGA card. The failure might be caused by saturating the PCI bus, and as a result no data moves in either direction.

The measurements above are more closely related to the type of motherboard than to the speed of the processor. For example, the Intel Zappa motherboard running at 75 MHz produced better results than an Intel Neptune motherboard running at 90 MHz. Measurement 4, above, fails completely on a Neptune 90 system with a Diamond Stealth 64 VGA card, but achieves 30 frames per second on a Zappa 75 system with the same Diamond Stealth 64 card.

VGACOPY Tests

Configurations Tested

We have tested the PX frame grabbers with three different Intel Pentium systems and four different PCI VGA cards.

Intel Pentium systems

- Neptune 90 MHz motherboard, Triton chip set.
- Zappa 75 MHz motherboard, Triton chip set.
- Zappa 100 MHz motherboard, Triton chip set.

PCI VGA cards

- Diamond SpeedStar, 1 MB of DRAM.
- Diamond Stealth 64, 2 MB of DRAM.
- Orchid Kelvin, 2 MB of DRAM.
- ATI Mach 64, 2 MB of DRAM.

Test Results

The performance of the Zappa motherboards is dependent on the brand of VGA card when VGA transfers are involved. When VGA transfers are not involved, the Zappa seems to be able to transfer 30 frames per second, using an NTSC camera, from the frame grabber to main memory regardless of processor speed. The processor speed of the Pentium chips on the Zappa motherboards does affect transfers to the VGA card. We were able to achieve higher transfer rates with the Zappa 100.

The performance of the Neptune motherboard is also dependent on the brand of VGA card when VGA transfers are involved. However, the Nep-

Imagenation

tune does not perform as well as the Zappa in any case. We were not able to achieve a transfer rate of 30 frames per second, using an NTSC camera, from the frame grabber to main memory with the Neptune.

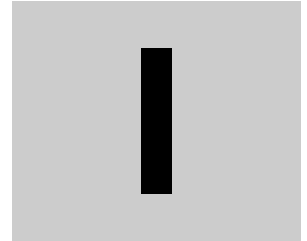
Here are some of our test results using VGACOPY and an NTSC camera (all values are in frames per second):

Configuration	Grab Frames and Copy to VGA Simultaneously	Grab one Frame and Copy to VGA
Zappa 100 & ATI Mach 64	30	15
Zappa 100 & Diamond Stealth 64	30	15
Zappa 100 & Diamond SpeedStar 64	20	10
Zappa 100 & Orchid Kelvin	30	15
Zappa 75 & ATI Mach 64	20	10
Zappa 75 & Diamond Stealth 64	30	15
Zappa 75 & Diamond SpeedStar 64	17	10
Zappa 75 & Orchid Kelvin	20	10
Neptune 90 & ATI Mach 64	20	10
Neptune 90 & Diamond Stealth 64	fails	15
Neptune 90 & Diamond SpeedStar 64	15	10
Neptune 90 & Orchid Kelvin	20	10

15 frames per second is a theoretical maximum for NTSC video when we grab one frame and then copy it to the VGA card because at least one field time is lost during the copy.

The measurements above apply only to DOS. You will see lower transfer rates under Windows because of the increased overhead and because Windows is a multitasking OS.

Index



Numerics

26-pin D connector [156](#)
386MAX [17](#)

A

adapter, 15-pin to 26-pin cables [157](#)
addresses
 logical [49, 54](#)
 physical [50, 55](#)
allocating frame grabbers [46](#)
 multiple frame grabbers [46, 82](#)
AUTOEXEC.BAT file [19, 21](#)
automatic synchronization [52](#)

B

binary files [75](#)
block diagram [161–162](#)
BMP files [75, 146](#)
board configuration [64](#)
board diagram [161–162](#)
board revision numbers [11, 63](#)
buffers, Visual Basic [43](#)

C

cables [13, 155–157](#)
 15-pin to 26-pin adapter [157](#)
CACHE flag [49, 52, 70, 93, 94, 96](#)
cache RAM [9, 51, 63](#)
camera inputs [57](#)
cameras
 progressive-scan [2, 4, 52, 62](#)
 resettable [3](#)
capture resolution [71–75](#)
capturing images [47–55](#)
CCIR/PAL [57, 62, 74](#)
color filter [5, 14](#)
CompactPCI bus [2](#)
 cables [155](#)
compatibility with PX500 [149–154](#)
compiling programs [37–44](#)
CompuServe address [30](#)
CONFIG.SYS file [17, 21](#)
configuration, hardware [64](#)
connectors [13, 155–157](#)
continuous acquire mode [94](#)
converting file formats [139–148](#)
corrupt image data [49](#)

- counting fields 62
- cropping images 73
- customer support 29–30

D

- debounce compensation 58
- decimation 6, 71
- digital input/output 8, 58
- direct memory access 50, 51
- directories 26
- display controllers 165
- DLLs
 - error loading 27
 - PXDV 76
 - PXDV.DLL 77
 - PXDV95.DLL 77
 - PXDVNT.DLL 77
 - Video Display 76
 - Windows 3.1 38, 39
 - Windows 3.1 Video Display DLL 77
 - Windows 95 39
 - Windows 95 Video Display DLL 77
 - Windows NT 40
 - Windows NT Video Display DLL 77
 - WPX5.DLL 38, 39
 - WPX5_95.DLL 39
 - WPX5_NT.DLL 40
- DMA 50, 51
- DOS Install program 18
- drivers
 - PX500.SYS 24
 - Windows NT 40

E

- EITHER flag 70
- EMM386 17
- environment variables 19, 21, 28, 31
- errors
 - error loading DLL 27
 - error loading VxD 27

- FILEIT program 142
- execution timing 64–70
- exiting libraries 44, 122
- external triggers 8, 58
 - debounce 58

F

- features, hardware 64
- field counter 62
- field length 71
- FIELD0 flag 70
- FIELD1 flag 70
- fields, video 62
- FILEIT program 139–148
 - batch processing 144
 - errors 142
- files
 - AUTOEXEC.BAT 19, 21
 - BIN format 75
 - binary 75
 - BMP format 75, 146
 - CONFIG.SYS 17, 21
 - GIF format 147
 - graphics formats 139
 - PCIVU.HLP 31
 - PCIVU.INI 31
 - PCX format 147
 - PIC format 147
 - PXDV.BAS 77
 - PXDV95.BAS 77
 - PXDVNT.BAS 77
 - reading and writing 75
 - SYSTEM.INI 20, 21, 22
 - TGA format 148
 - TIFF format 148
 - WPG format 148
 - WPX_95.BAS 43
 - WPX_NT.BAS 43
 - WPX5.H 39
 - WPX5.LIB 39
 - WPX5_95.H 39

- WPX5_95.LIB 39
- WPX5_95B.LIB 39
- WPX5_NT.H 40
- WPX5_NT.LIB 40
- WPX5_NT.B.LIB 40
- WPX5VB.BAS 43
- WPXVB.BAS 43
- flags 65, 68, 69, 70
 - CACHE 49, 52, 93, 94, 96
- FLATMEM.COM 17, 21
- frame buffers
 - error trying to allocate 48
 - memory allocation 20, 22
 - PX500_SIZE variable 20, 22
- frame grabber handles 46
- frames 48
 - video 62
- freeing frame grabbers 46
 - PXCLEAR program 47
- freeing memory 48
- function flags 70
- function reference 79–120
- function timing 64–70
- functions
 - AllocateAddress() 50, 80
 - AllocateBuffer() 48, 81
 - AllocateFG() 46, 81
 - CacheTriggered() 52, 58, 82
 - CheckEqual() 61, 83
 - CheckError() 83
 - CheckGreater() 61, 84
 - draw_image() 123, 137
 - draw_rectangle() 123, 137
 - draw_scaled_image() 123, 138
 - edit() 138
 - ExitLibrary() 44, 85
 - fill_rectangle() 123, 137
 - FireStrobe() 60, 85
 - FrameAddress() 55, 86
 - FrameBits() 55, 86
 - FrameBuffer() 49, 54, 87
 - FrameHeight() 55, 87
 - FrameWidth() 55, 87
 - FreeFG() 46, 88
 - FreeFrame() 48, 88
 - get_key() 133
 - GetCamera() 57, 88
 - GetColumn() 89
 - GetFieldCount() 62, 89
 - GetFieldLength() 89
 - GetFineGain() 56, 90
 - GetGainRange() 56, 90
 - GetLUT() 61, 90
 - GetOffset() 55, 91
 - GetRectangle() 91
 - GetRow() 91
 - GetStrobeState() 92
 - GetSyncType() 93
 - GetTriggerType() 93
 - Grab() 49, 93
 - GrabContinuous() 49, 94
 - GrabToCache() 52, 95
 - GrabTriggered() 49, 58, 95
 - HaveCache() 63, 96
 - immediate 68
 - InitLibrary() 44, 96
 - IsFinished() 66, 97
 - KillQueue() 67, 97
 - menu_calc_dx() 134
 - menu_calc_dy() 134
 - menu_display() 123
 - menu_display() 134
 - menu_erase() 135
 - menu_generate() 123, 135
 - menu_select() 123, 136
 - PutColumn() 98
 - PutRectangle() 98
 - PutRow() 99
 - pxPaintDisplay() 76
 - pxSetWindowSize() 76
 - queued 65, 67
 - ReadBin() 75, 99
 - ReadBMP() 75, 100
 - ReadCache() 52, 101

ReadConfiguration() 101
ReadProtection() 64, 102
ReadRevision() 63, 103
ResetFG() 103
SetCamera() 57, 103
SetCompare() 61, 104
SetCurrentWindow() 104
SetDrivePolarity() 60, 106
SetDriveType() 106
SetFieldCount() 63, 107
SetFieldSize() 107
SetFineGain() 56, 108
SetGainRange() 56, 109
SetImageSize() 71, 110
SetLUT() 61, 111
SetOffset() 55, 112
SetStrobePeriods() 60, 112
SetStrobePolarity() 60, 113
SetStrobeType() 59, 113
SetTriggerType() 58, 114
SetVideoFormat() 52, 60, 71
vg_exit_graph() 122, 127
vg_getbkcolor() 128
vg_getcolor() 129
vg_gotoxy() 122, 123, 129
vg_init_graph() 122, 127
vg_maxx() 129
vg_maxy() 130
vg_print() 122, 123, 130
vg_resizefont() 131
vg_setcolor() 131
vg_setfont() 132
vg_sizex() 132
vg_sizey() 132
vg_wherex() 133
vg_wherey() 133
vga_set_palette() 122, 128
vga_wait_vb() 128
VideoType() 57, 62, 117
Wait() 67, 117
WaitFinished() 118
WaitVB() 67, 118

WriteBin() 75, 119
WriteBMP() 75, 120

G

gain 56, 61
GIF files 147
grabbing images 47–55
 incomplete image captures 50
 invalid data in buffer 50
graphics file formats 139
grayscale noise 3
grayscale resolution 6, 75

H

handles 46
hardware configuration 64
hardware installation 14–16
hardware protection key 64
hardware specifications 159–160
header files 26
 DOS 37, 38
 PX5.H 37, 38
 Windows 95 39
 Windows NT 40
 WPX5.H 38, 39
 WPX5_95.H 39
 WPX5_NT.H 40

I

image cropping 73
image resolution 71–75
image scaling 71
IMAGENATION variable 19, 28, 31
IMMEDIATE flag 68, 69, 70
immediate functions 68
initializing libraries 44, 122
input/output 8, 58
inputs, video 57
INSTALL program 18

installation 13–30
 installing the PX board 14–16
 installing the PX software 16–26
 internal synchronization 53
 Internet address 30
 interrupt handlers 45
 interrupts 45
 IRQ conflicts 27, 28, 45

L

languages, programming 37–44
 libraries

- Borland, DOS 37
- compatibility 149–154
- compiling and linking 37–44
- error when initializing 45
- exiting 44, 122
- function reference 79–120
- general characteristics 36
- initializing 44, 122
- Microsoft, DOS 37
- PX5_FW.LIB 38
- PX5_L6.LIB 37
- PX5_LB.LIB 37
- PX5_LM.LIB 37
- PXD.V.LIB 77
- PXD.V95.LIB 77
- PXD.VNT.LIB 77
- troubleshooting 45
- VESAMENU 121–138
- Watcom DOS/4GW 38
- Windows 3.1 38
- Windows 3.1 Video Display DLL 77
- Windows 95 39
- Windows 95 Video Display DLL 77
- Windows NT 40
- Windows NT Video Display DLL 77
- WPX5.LIB 38, 39
- WPX5_95.LIB 39
- WPX5_95B.LIB 39
- WPX5_NT.LIB 40

WPX5_NT.B.LIB 40
 linking programs 37–44
 logical addresses 49, 54
 lookup table 60
 LUT, see lookup table

M

memory

- allocation variable 20, 22
- freeing 48
- managers 17
- requirements 16, 23, 25, 45

 memory_size registry key 23
 menus 121–138
 motherboards 165
 MSD program 17
 multitasking and multithreaded operating systems 41

N

non-interlaced video 4, 62
 NonPagedPoolSize registry key 25
 non-standard video formats 52
 NTSC 57, 62, 74

O

offset 55
 operating systems 37–44

- multitasking and multithreaded 41
- Windows 95 39
- Windows NT 40

P

PAL 57, 62
 PATH variable 19, 21
 PC/104-Plus bus 2

- cables 157

 PCI BIOS 45

- PCI bus [6](#), [78](#), [163–166](#)
 - cables [155](#)
 - PCIVU program [31–34](#)
 - troubleshooting [27](#)
 - PCIVU.HLP file [31](#)
 - PCIVU.INI file [31](#)
 - PCX files [147](#)
 - performance [11](#), [78](#), [163–166](#)
 - physical addresses [50](#), [55](#)
 - PIC files [147](#)
 - pixel decimation [6](#), [71](#)
 - pixel jitter [3](#)
 - pixel values
 - comparing [61](#)
 - lookup table [60](#)
 - pointers [42](#), [49](#), [54](#)
 - programming [35–78](#)
 - programming languages [37–44](#)
 - programs
 - compiling and linking [37–44](#)
 - directory location [26](#)
 - FILEIT [139–148](#)
 - INSTALL [18](#)
 - MSD [17](#)
 - PCIVU [27](#), [31–34](#)
 - PXCLEAR [11](#), [47](#)
 - PXGDI1 [11](#)
 - PXGDI2 [11](#)
 - PXGDI3 [11](#)
 - PXREV [11](#), [27](#)
 - SETUP [18](#)
 - VGACOPY [11](#), [163](#)
 - progressive-scan cameras [2](#), [4](#), [52](#), [62](#)
 - protection key, hardware [64](#)
 - purging the function queue [67](#)
 - PX5 directory [26](#)
 - PX5.H file [37](#), [38](#)
 - PX5_95.VXD virtual device driver [39](#), [40](#)
 - PX5_FW.LIB library [38](#)
 - PX5_L6.LIB library [37](#)
 - PX5_LB.LIB library [37](#)
 - PX5_LM.LIB library [37](#)
 - PX500 frame grabber
 - 15-pin to 26-pin adapter [157](#)
 - compatibility with PX510 [149–154](#)
 - PX500.SYS driver [24](#), [40](#)
 - PX500.VXD virtual device driver [20](#), [38](#)
 - PX500_SIZE variable [20](#), [22](#)
 - PX510 frame grabber
 - cables [155](#)
 - features [2](#)
 - PX610 frame grabber
 - cables [155](#)
 - features [2](#)
 - progressive-scan feature [4](#)
 - PXCLEAR program [11](#), [47](#)
 - PXDV.BAS file [77](#)
 - PXDV.DLL [77](#)
 - PXDV.LIB library [77](#)
 - PXDV95.BAS file [77](#)
 - PXDV95.DLL [77](#)
 - PXDV95.LIB library [77](#)
 - PXDVNT.BAS file [77](#)
 - PXDVNT.DLL [77](#)
 - PXDVNT.LIB library [77](#)
 - PXGDI1 program [11](#)
 - PXGDI2 program [11](#)
 - PXGDI3 program [11](#)
 - PXREV program [11](#)
 - troubleshooting [27](#)
- ## Q
- QEMM [17](#)
 - queue structure under Windows NT [67](#)
 - QUEUED flag [65](#), [69](#), [70](#)
 - queued functions [65](#), [67](#)
- ## R
- registry, Windows 95 [22](#)
 - registry, Windows NT [24](#)
 - resettable cameras [3](#)
 - resolution [71–75](#)

return values, FILEIT program 142
 revision numbers 11, 63
 RS-170 standard 62

S

sample programs, see programs
 sampling range 56
 scaling images 71
 security 64
 SETUP program 18
 SINGLE_FLD flag 70
 single-field synchronization 53
 software
 directories 26
 installation 16–26
 security 64
 updates 30
 source code directory location 26
 specifications 159–160
 StaticVxD registry key 23
 strobes 8, 59, 112, 113
 structures
 colors 124
 menu 123, 125
 menuitem 123, 126
 support 29–30
 sync signals, output 60
 synchronization modes 52
 synchronization timing 3, 60
 synchronizing program execution to video 67
 system files 19, 20
 SYSTEM.INI file 20, 21, 22

T

technical support 29–30
 TGA files 148
 TIFF files 148
 timing, function execution 64–70

triggers 8, 58
 debounce 58
 troubleshooting
 AllocateBuffer() 48
 AllocateFG() 46
 Borland 32-bit programs 39, 40
 can't allocate a frame grabber 46
 can't allocate frames 48
 corrupt image data 49
 error loading DLL 27
 error loading VxD 27
 freeing frame grabbers 47
 GetColumn(), GetRectangle(),
 GetRow() 55
 grab functions fail 49
 grabbing images 50
 image is all black 49
 incomplete image 50
 InitLibrary() 45
 invalid data in buffer 50
 IRQ conflicts 27, 28, 45
 library fails to initialize 45
 partial image 28
 PCIVU program 27
 PutColumn(), PutRectangle(),
 PutRow() 55
 PXREV program 27
 slow video display performance 28
 Windows 28

U

updates, software 30
 user interface 121–138
 user synchronization 53
 utility programs, see programs

V

vertical resolution 72
 VESA display drivers 27

VESAMENU library [121–138](#)

VGA cards [165](#)

VGACOPY program [11](#), [163](#)

video

 cache RAM [9](#), [51](#), [63](#)

 field length [71](#)

 fields [62](#)

 formats [57](#), [62](#)

 frames [62](#)

 gain [56](#), [61](#)

 inputs [57](#)

 non-standard formats [52](#)

 offset [55](#)

 sampling [62](#)

 synchronization [52](#)

Video Display DLL [76](#)

virtual device drivers [20](#), [38](#), [39](#), [40](#), [45](#)

Visual Basic

 buffers [43](#)

 declarations [43](#)

 End button [44](#)

 programming tips [42](#)

 Video Display DLL [76](#)

VxD [20](#), [38](#), [39](#), [40](#), [45](#)

 error loading [27](#)

W

WEN signals [3](#), [53](#)

window enable signal [3](#)

Windows

 troubleshooting [28](#)

Windows 95 [39](#)

 programming tips [39](#)

 registry changes [22](#)

 virtual device driver [40](#)

Windows NT

 AllocateAddress() [51](#)

 DLL function differences [41](#)

 FrameAddress() [55](#)

 programming tips [40](#)

 PX500.SYS driver [40](#)

 queue structure [67](#)

 registry changes [24](#)

Windows Setup program [18](#)

WPG files [148](#)

WPX_95.BAS file [43](#)

WPX_NT.BAS file [43](#)

WPX5.DLL [38](#), [39](#)

WPX5.H file [38](#), [39](#)

WPX5.LIB file [39](#)

WPX5.LIB library [38](#)

WPX5_95.DLL [39](#)

WPX5_95.H file [39](#)

WPX5_95.LIB file [39](#)

WPX5_95B.LIB file [39](#)

WPX5_NT.DLL [40](#)

WPX5_NT.H file [40](#)

WPX5_NT.LIB file [40](#)

WPX5_NT.B.LIB file [40](#)

WPX5VB.BAS file [43](#)

WPXVB.BAS file [43](#)