IMAQ™ Vision for LabWindows®/CVI™

February 1997 Edition
Part Number 321424A-01

Internet Support

`support@natinst.com`
E-mail: `info@natinst.com`
FTP Site: `ftp.natinst.com`
Web Address: `http://www.natinst.com`

Bulletin Board Support

BBS United States: (512) 794-5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support

(512) 418-1111

Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678

International Offices

Australia 02 9874 4100, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,
Finland 09 527 2321, France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186,
Israel 03 5734815, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456,
Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886,
Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200,
U.K. 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway   Austin, TX 78730-5039   Tel: (512) 794-0100

Important Information

Contents

About This Manual

The *IMAQ Vision for LabWindows/CVI* user manual describes the features, functions, and operation of the IMAQ Vision for LabWindows/CVI toolkit. To use this manual effectively, you must be familiar with image processing, LabWindows/CVI, and your image capture hardware.

Organization of This Manual

This manual is designed to accompany the IMAQ Vision for LabWindows/CVI software. Read this section prior to writing C code that uses any of the IMAQ Vision functions.

The *IMAQ Vision for LabWindows/CVI* user manual is organized as follows:

- Chapter 1, *Introduction*, describes IMAQ Vision, the image processing and analysis library for LabWindows/CVI from National Instruments. IMAQ Vision is fully integrated with LabWindows/CVI making it a powerful development environment for image processing. You can use it for almost any type of scientific or industrial tasks, from medical microscopy to quality control.
- Chapter 2, *Basic Concepts*, explains the basic ideas underlying image processing with IMAQ Vision for LabWindows/CVI.
- Chapter 3, *Management Functions*, describes the IMAQ Vision management functions.
- Chapter 4, *Display and File Functions*, describes the IMAQ Vision display and file functions.
- Chapter 5, *Tools Functions*, describes the IMAQ Vision tools functions.
- Chapter 6, *Image Processing Functions*, describes the IMAQ Vision image processing functions.

- Appendix, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

bold	Bold text denotes a parameter, menu name, palette name, menu item, return value, function panel item, or dialog box button or option.
italic	Italic text denotes mathematical variables, emphasis, a cross reference, or an introduction to a key concept.
bold italic	Bold italic text denotes a note.
`monospace`	Text in this font denotes text or characters that you literally enter from the keyboard. Sections of code, programming examples, and syntax examples also appear in this font. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, variables, filenames, and extensions, and for statements and comments taken from program code.
<>	Angle brackets enclose the name of a key on the keyboard—for example, <PageDown>.
-	A hyphen between two or more key names enclosed in angle brackets denotes that you must simultaneously press the named keys—for example, <Control-Alt-Delete>.
<Control>	Key names are capitalized.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, and files, as in `C:\dir1name\dir2name\filename`.
☞	This icon to the left of bold italicized text denotes a note, which alerts you to important information.

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in the Appendix, *Customer Communication*, at the end of this manual.

Chapter 1

Introduction

Welcome to IMAQ Vision, the image processing and analysis library for LabWindows/CVI from National Instruments. IMAQ Vision is fully integrated with LabWindows/CVI making it a powerful development environment for image processing.

You can use it for almost any type of scientific or industrial task, from medical microscopy to quality control. After you have familiarized yourself with IMAQ Vision for LabWindows/CVI, you will find it easy to write applications that automatically capture, measure, and control processes based on image processing.

System Set Up and Operation

This manual presumes that you have already written C programs and are familiar with the LabWindows/CVI environment. Terminology in this document is consistent with C language and LabWindows/CVI terminology.

IMAQ Vision for LabWindows/CVI is currently available for LabWindows/CVI 4.0 for the Windows 95 and Windows NT operating systems.

System Requirements

- An IBM PC or compatible computer with an 80486 DX2 processor. However, National Instruments strongly recommends a Pentium processor.
- A SVGA display board capable of displaying 800x600 pixels in 32,768 or 16 million colors
- 16 MB of Ram
- A hard drive
- Microsoft Windows 95 or Windows NT
- LabWindows/CVI version 4.0.1

Installation

Launch the `Setup.exe` file from your IMAQ Vision distribution disk. If your LabWindows/CVI folder is not `C:\CVI401`, you must change the destination directory.

Chapter 2

Basic Concepts

This chapter explains the basic ideas underlying image processing with IMAQ Vision for LabWindows/CVI.

About Images

An image is a function of the light intensity *f*(*x*,*y*) where *x* and *y* represent the spatial coordinates of a point in an image and where *f* is the brightness of the point (*x*,*y*). In other words, an image is a two dimensional array of values which represent light intensity. These values are encoded with a range determined by the pixel depth.

Several factors influence the decision to encode an image in 8 bits, 16 bits or in a floating value. These factors include the nature of the image, the type of image processing to use, and the type of analysis to perform. For example, 8-bit encoding is sufficient if you plan to perform morphology analysis (surface, elongation factor, and so on). On the other hand, if you want to obtain a highly precise quantification of the light intensity from an image or a region of an image, 16-bit or 32-bit (floating point) encoding is required.

An image can consist of one or several planes. It is possible to acquire and process a real color image which contains three planes. Each plane represents the intensities of the primary colors: red, green, and blue. This image type is also known as *RGB Chunky*. It is encoded in 32 bits including 8 bits for the alpha channel (not used in IMAQ Vision), and 8 bits each for the red, green, and blue planes. The most common operation on this image type is the extraction of the color, light, saturation, or hue component from the image. The final result is an 8 bit image that you can process as a standard monochrome image.

Complex images are made up of two planes: the Real and the Imaginary planes; each pixel is encoded as two times a 32-bit floating value. This type of image is the result of mathematical calculations called Fourier Transforms; hence the name *complex* image. They are particularly intended for processing images in the frequency domain.

IMAQ Vision for LabWindows/CVI uses all the aforementioned image types. However, certain operations do not have a practical purpose when used on certain image types. For example, it does not make sense to apply the logic operator AND to a Complex image. IMAQ Vision cannot use certain other image types, particularly images encoded in files as 1-, 2-, or 4-bit images. In these cases, IMAQ Vision automatically transforms the images into 8-bit images (minimum pixel depth for IMAQ Vision) when you open the image file. This transformation is transparent to you and has no effect on the use of these image types.

In IMAQ Vision, the image type is defined when the IPI_Create() function creates the image object. The most common image type for the scientific and industrial fields is 8 bit (in other words, a single image plane encoded using 8 bits per pixel). However, IMAQ Vision is designed to acquire and process images encoded in 10, 12, or 16 bits as well as in floating point and true color (RGB).

The following are used to define the image type for each of the IMAQ Vision functions.

Table 2-1. IMAQ Vision for LabWindows/CVI Image Types

Value	Description
IPI_PIXEL_U8	8 bits per pixel (unsigned, standard monochrome)
IPI_PIXEL_I16	16 bits per pixel (signed)
IPI_PIXEL_SG	32 bits per pixel (floating point)
IPI_PIXEL_COMPLEX	2 times 32 bits per pixel (floating point) (native format after a FFT)
IPI_PIXEL_RGB	32 bits per pixel (RGB chunky, standard color)

An IMAQ Vision image has the following other attributes in addition to its type and size:

- Calibration
- Image border

The calibration attribute defines the physical horizontal and vertical dimensions of the pixels. With the ability to calibrate two axes independently, you can correct defaults resulting from the sensor (this is not uncommon). Only calculations based on morphological transformations (surface, perimeter, and so on) require the use of these coefficients. These coefficients have no effect on either processing or operations between images.

An image border also exists. This border physically reserves space in the image. It is completely transparent to you. Borders are necessary when you want to perform a morphological transformation, a convolution, or a particle analysis. All these processes include an operation between neighboring pixels. These operations assign a new value to a pixel in relation to the value of its neighbor. With the border, operations can treat all pixels the same way.

Overview

The IMAQ Vision installation adds three important files in your current LabWindows/CVI directory:

- `IMAQ_CVI.H` contains all constants, enumerated types, structures and prototypes related to IMAQ Vision.
- `IMAQ_CVI.FP` contains all functions front panels. These panels are similar to the other LabWindows/CVI functions front panels.
- `IMAQ_CVI.LIB` contains the code of the IMAQ Vision functions. This library is compatible with Microsoft VISUAL C.

The IMAQ Vision function tree (`IMAQ_CVI.FP`) contains separated classes corresponding to a group or a type of function. When choosing IMAQ Vision in the LabWindows/CVI **Instrument** menu, the following table appears.

Table 2-2. IMAQ Vision for LabWindows/CVI Function Types

Function Type	Description
Analysis...	Functions analyzing the contents of an image. Basic and complex particle detection. Extraction of measurements and morphological coefficients for each object in an image.
Color...	Functions for color image processing and analysis (histogram, threshold) and the manipulation of color image planes (conversions).
Complex...	Fast Fourier Transforms (FFT), inverse FFT, truncation, attenuation, addition, subtraction, multiplication, and division of complex images. Functions for the extraction and manipulation of complex planes.
Conversion...	Linear or non-linear conversions from one image type into another.
Display...	Functions covering all aspects of image visualization and image window management. You can control up to 16 image windows. Image window managers are also included so you can select various shapes for creating and manipulating a region of interest.
Files...	Functions for reading and writing images from and to disk files.
Filters...	Contains functions such as convolution and non linear filters: median, gradient, low pass, Prewitt, Sobel, Roberts, sigma.
Geometry...	Includes functions for 3D view, rotate, shift, and symmetry.
Management...	Functions initializing the IMAQ Vision subsystem, creating, listing, and disposing of image structures. Also includes error handling for all the IMAQ Vision functions.
Morphology...	Morphology functions processing binary images. Include erosion, dilation, closing, opening, edge detection, thinning, thickening, hole filling, low pass, high pass, distance mapping, and rejection of particles touching the border. Morphology functions for modifying gray scale images include erosion, dilation, closing, opening, and auto-median.

Table 2-2. IMAQ Vision for LabWindows/CVI Function Types (Continued)

Function Type	Description
Operator...	Arithmetic, logic, and comparison functions. Addition, subtraction, multiplication, division, ratio and modulo between two images or between one image and a constant. Logic operators include AND, NAND, OR, NOR, XOR, XNOR and LogDiff between two images or between one image and a constant. Clear or Set (affect) as a function of a relational operator between two images or between one image and a constant. Masking, extraction of a minimum, maximum, or average can be completed between two images or between an image and a constant.
Processing...	Threshold, label, LUT (lookup table) transformation, and so on.
Tools(diverse)...	Functions to draw shapes into an image.
Tools(Image)...	A set of diverse functions for the manipulation of images (copy, reduction, expansion, extraction, and so on). Also included is a function to get all information about the image attributes and pixel mapping.
Tools(Pixels)...	Function to transform the contents of an image from and to a user array.

Source, Destination, and Mask Images

IMAQ Vision for LabWindows/CVI uses internal tables for all images and private data structures. The only way to initialize the IMAQ Vision internal tables is by calling one of these two functions:

- `IPI_Create();`
- `IPI_InitSys();`

`IPI_Create()` implicitly calls the `IPI_InitSys()`. Do not use any other function prior to one of these two calls. You must end IMAQ Vision function calls by calling `IPI_CloseSys()`. This destroys all internal tables and sets them in the initial state. Notice that `IPI_Create()` is often the first function to be used.

Under IMAQ Vision, an image is a private structure. The only way to create images is by calling `IPI_Create()`. `IPI_Create()` returns an image reference you systematically use when calling other IMAQ Vision functions. However, with functions such as `IPI_GetImageInfo()`, you have access to everything you need to know about image structure and mapping, including the pixel address.

No limitation exists in the number or size of images you can create. The only limit is the amount of memory installed in your computer.

Depending on the function, you might need one or more image references. In some cases, you might only need one image reference. Typically, the functions that analyze an image, read an image from a file, or transform a user array into an image (`IPI_ArrayToImage()`, for example) use one image reference only. In other cases you might need to use a second image as a mask image.

If a function has a `mask_image` parameter, this indicates that the function process or analysis is dependent on the contents of another image (the `mask_image`). Each pixel in `image` is processed if the corresponding pixel in the `mask_image` has a value other than zero. This image mask must be an image type `IPI_PIXEL_U8` and its contents are binary (zero or other than zero).

If you want to apply a process or an analysis function to the entire image, insert the keyword `IPI_NOMASK` instead of a mask image reference. As an example, see the following implementation variant calling `IPI_Histogram()`:

- `IPI_Histogram(myImage, myMaskImage, ...);`

  This call performs a histogram computation using a mask image.
- `IPI_Histogram(myImage, IPI_NOMASK,...);`

  This call performs a histogram computation on the full image.

☞ **Note:** `IPI_NOMASK` ***is the default value for all function panels that use a*** `mask_image`**.**

All IMAQ Vision functions that process the contents of an image (that is, that modifies the pixel values) have `source_image` and a `dest_image` input parameters. This is the most common type of prototype in IMAQ Vision. The `source_image` receives the image to process. The `dest_image` can receive either another image or the original one, depending on your wishes. If two different images are used for the two inputs, the original image `source_image` is not affected. If the `source_image` and `dest_image` receive the same image, the processed image is placed into the original image and the original image data is lost. See the following examples applied on the `IPI_Threshold()` function:

- `IPI_Threshold(myImage, myImage, 0, 128, 1,TRUE);`

  This applies a threshold to the image using the same image for the source and destination. The content of the image changes.

- `IPI_Threshold(myImage, myBinaryImage, 0, 128, 1, TRUE);`

  This applies a threshold to the image using a destination image different from the source. The source image remains unchanged, while the destination image `myBinaryImage` contains the result.

The `dest_image` is the image that receives the process results. Depending on the function, its type can be either the same as or different from that of the `source_image`. In later chapters, this manual describes each function and the image types allowed for their `image` input. In all cases, the size of the image connected to `dest_image` is irrelevant because the function modifies it automatically to correspond to the source image size.

Other functions such as linear filters are able to process the image according to a `mask_image`. This kind of function has three image references as input parameters: `source_image`, `mask_image` and `dest_image`. See the following implementations of the `IPI_GrayEdge()` function:

- `IPI_GrayEdge(myImage, IPI_NOMASK, myImage, IPI_EDG_PREWITT, 0);`

  This function performs the process on the entire image using the same image as source and destination.

- `IPI_GrayEdge(myImage, myMaskImage, myImage, IPI_EDG_PREWITT, 0);`

  This function performs the process according a mask using the same image as source and destination.

- `IPI_GrayEdge(myImage, myMaskImage, myEdgeImage, IPI_EDG_PREWITT, 0);`

  This function performs the process according to a mask using a different image as destination.

- `IPI_GrayEdge(myImage, IPI_NOMASK, myEdgeImage, IPI_EDG_PREWITT, 0);`

  This function performs the process on the entire image using a different image as destination.

Some functions perform arithmetic or logical operations between two images. There are two source images for a destination image. You can perform an operation between two images and then store the result in another image. You can also store the result in one of the two source images if you consider the original data unnecessary. The following examples show the possible combinations using `IPI_Add()` function:

- `IPI_Add(myImageA, myImageB, myResultImage, 0);`

  This function adds two images and puts the result into a third one.

☞ **Note:** ***In this case the three images are all different.*** `myImageA` ***and*** `myImageB` ***are intact after processing and the result of this operation is stored in*** `myResultImage`**.**

- `IPI_Add(myImageA, myImageB, myImageA, 0);`

  This function adds two images and puts the result into the first one.
- `IPI_Add(myImageA, myImageB, myImageB, 0);`

  This function adds two images and puts the result into the second one.

Most operations between two images require that the images have the same size. However, arithmetic operations can be performed between two different image types (in other words, 8-bit and 16-bit).

- `IPI_Add(myImage, IPI_USECONSTANT, myResultImage, 25);`

  This function adds the image and a constant and puts the results into another image.
- `IPI_Add(myImage, IPI_USECONSTANT, myImage, 25);`

  This function adds the image and a constant and puts the result into the original image.

Processing Options

Connectivity

In some functions (primarily the morphology function group), there is a parameter with which you can specify the pixel *connectivity*. This parameter, `connectivity_8`, selects how the algorithm determines if two adjacent pixels are connected or are part of the same particle.

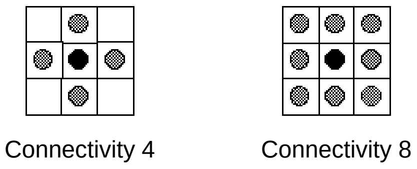

Figure 2-1. Connectivity

Example

The gray points in the original image define the particles. In the subsequent images, various shades of gray distinguish the particles. Using connectivity 4, six particles are detected. Using connectivity 8, three particles are detected.

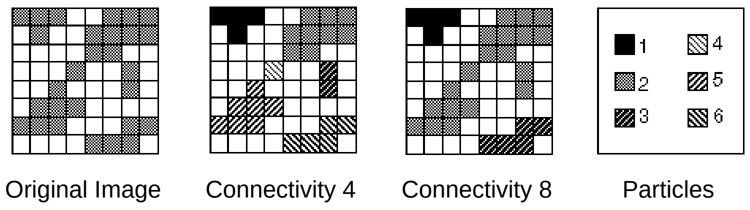

Figure 2-2. Example of Connectivity Processing

Structuring Element Descriptor

A structuring element descriptor is a specific IMAQ Vision structure defined as (see `IMAQ_CVI.H`):

```
typedef struct {
   int strucElemWidth;
   int strucElemHeight;
   int *strucElements;
   int hexaProcessing;
   } IPIMorphoDesc, * IPIMorphoDescPtr;
```

It is used specifically for morphological transformations. The first two fields `strucElemWidth` and `strucElemHeight` set the geometry and the size of the structuring element itself. The third field `strucElements` is a pointer to the structuring element values. The values contained in this structuring element are either 0 or 1. These values dictate which pixels are to be taken into account during process.

The use of structuring elements requires that the image contains a border. The application of a 3x3 structuring element requires a minimal border size of 1. In the same way, structuring elements of 5x5 and 7x7 require a minimal border size of 2 and 3 respectively. Bigger structuring elements require corresponding increases in the image border size.

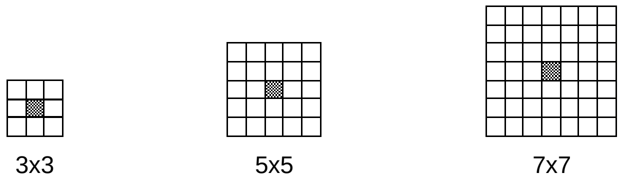

Figure 2-3. Structuring Element

The coordinates of the central pixel (the pixel being processed) are determined as a function of the structuring element. In this example the coordinates of the processed pixels are (1,1), (2,2), and (3,3). Notice that the origin is always the upper, left-hand corner pixel.

The hexaProcessing Field

Remember that a digital image is a 2D array of pixels arranged in a regular rectangular grid. In image processing, this grid can have two different pixel frames: square or hexagonal. As a result the structuring element applied during a morphological transformation can have either a *square* or *hexagonal* frame. You make the decision to use a square frame or hexagonal frame. This decision affects how the algorithm analyzes the image when you process it with functions that use this frame concept. The chosen pixel frame directly affects the output from morphological measurements (perimeter, surface, and so on). However, the frame has no effect on the availability of the pixel in memory.

By default the square frame is used in IMAQ Vision (hexaProcessing contains 0). Use a hexagonal frame to obtain highly precise results. As shown below, with a hexagonal plane, the even lines shift a half pixel to the right. Therefore, the hexagonal frame places the pixels in a configuration similar to a true circle. In those cases when the hexagonal frame is used, only the structuring element values that possess an *x* are used.

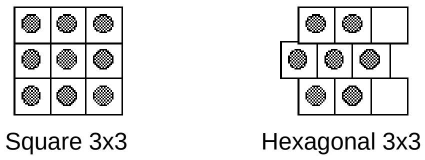

Figure 2-4. Square vs. Hexagonal Frames

It is clear that the size of the structuring element directly determines the speed of the morphological transformation. Different results occur when the contents of the structuring element are changed. National Instruments recommends that you possess a solid comprehension of morphology or spend some time learning how to use these elements before changing the standard structuring element (filled with 1s).

The structuring elements shown below each have a different result.

1	1	0
1	1	0
1	1	0

0	1	1
0	1	1
0	1	1

1	1	1
1	1	1
0	0	0

0	0	0
1	1	1
1	1	1

Figure 2-5. Structuring Element Morphological Results

The standard way to perform morphological operations is to use a structuring element containing 1, no matter what size the structuring element is. To simplify the use of morphological functions that need a `Structuring element descriptor`, IMAQ Vision has three standard `MorphoDescPtr` functions for three different sizes:

- `IPI_MO_STD3X3`—pointing to a 3x3 structuring element filled with nine 1 values.
- `IPI_MO_STD5X5`—pointing to a 5x5 structuring element filled with twenty-five 1 values.
- `IPI_MO_STD7X7`—pointing to a 7x7 structuring element filled with forty-nine 1 values.

☞ **Note:** ***The default value in the IMAQ Vision for LabWindows/CVI morphological function panel is*** `IPI_MO_STD3X3`**.**

User Pointers and IMAQ Vision for LabWindows/CVI Pointers

Several IMAQ Vision functions return data, data structure, and data array. For most of these functions, a parameter you provide determines the returned data size. Two examples are the buffers used in the functions `IPI_GetLine()` and `IPI_Histogram()`.

IPI_GetLine

```
IPI_GetLine (IPIImageRef image,Point start,Point end,
int array_format,void *array,int *nb_of_elements);
```

`IPI_GetLine` returns all the pixels located under the vector given by the `start` point and the `end` point into a user buffer in the desired format. The `array` is the address of memory buffer allocated by you. You *must* allocate a buffer big enough to receive all the pixel values. In this case, you can use one of the following three approaches to compute and allocate the necessary buffer space:

- Extract the maximum range in the `start` and the `end` points using:

  `max(abs(start.x - end.x), abs (start.y - end.y)) +1`

  This approach works if you are sure that the points are both in the image space coordinates. If the point coordinates are virtual, you must allocate an infinite buffer.

- Obtain the *x* and *y* resolution of the image using `IPI_GetImageInfo()`. Next you have to extract the maximum between the horizontal and the vertical sizes and allocate your buffer using this value. The real number of pixels copied from the image into your array is returned by the function in the `nb_of_elements` parameter.
- Determine an arbitrary maximum buffer (1024, 2048,...) and use the `nb_of_elements` returned by the function to know how much data is relevant.

IPI_Histogram()

Other cases of using a user pointer can be explained by describing an `IPI_Histogram()` call as follows:

```
IPI_Histogram(IPIImageRef image, IPIImageRef,
mask_image,int number_of_classes, float minimum_value,
float maximum_value, int histogram[], IPIHistoReport
*histogram_report);
```

The `histogram[]` parameter is an array of integers where the function returns the histogram values. The value of `number_of_classes` determines the size of this array. You have to be consistent with the size allocated for the `histogram` buffer and the value passed in `number_of_classes`.

Furthermore, it is not easy and might be impossible for you to determine the necessary buffer size for a set of functions. In this case instead of using a user pointer, the function has to return the pointer. The `IPI_Particle()` function, for example, performs a particle detection and returns parameters on detected particles. This function cannot use a user pointer because you do not know the number of particles before the function detects them.

These functions have to allocate their own pointers and return these pointers to you. The standard `realloc()` has a similar use and behavior. If you pass a pointer to a NULL pointer, IMAQ Vision allocates a new one adjusted according the required size. If you pass a pointer to a previously allocated pointer, this pointer might be changed and/or resized. In all cases, you have to free the buffer. The following lines show you a typical technique for this kind of function:

```
int particleCount;
IPIFullPreportPtr myParticleReports = NULL;
IPI_Particle(myImage, IPI_CON8, &particleCount, &myPar-
ticleReports);
.
. processing particle reports...
.
free(myParticleReports);
```

The following is another typical program processing several images:

```
void procAllImages(IPIImageRef images[], int imageCount)
{
int particleCount, i;
IPIFullPreportPtr myParticleReports = NULL;
for(i=0; i < imageCount; i++)
   {
IPI_Particle(images[i], IPI_CON8, &particleCount, &my-
ParticleReports);
   .
   . processing particle reports.
   .
   }
free(myParticleReports);
}
```

You have to implement this technique when you use a function with a parameter named `<something>Ptr *<paramname>`.

Starting with IMAQ Vision for LabWindows/CVI

The IMAQ Vision installation procedure creates a directory named `Samples`. The `Samples` directory contains two sub-directories named `Sample1` and `Sample2`. The `Sample1` directory contains an IMAQ Vision example you can read and use before starting your own IMAQ Vision based program. The files related to `Sample1` are:

- `IMAQSample1.prj`—the LabWindows project file
- `IMAQSample1.h`—the header file
- `IMAQSample1.c`—the source file
- `IMAQSample1.uir`—the user interface file

In this example, you can see how to call and chain IMAQ Vision functions such as reading an image file, displaying an image, setting a color palette, performing a threshold, computing and plotting a histogram, and finally labelling the particles.

The files related to the `Sample2` are:

- `IMAQSample2.prj`—the LabWindows project file
- `IMAQSample2.h`—the header file
- `IMAQSample2.c`—the source file
- `IMAQSample2.uir`—the user interface file
- `iron.bmp`—the image

`Sample2` contains a real-world example of detecting iron particles in iron ore and measuring the density of these particles. This example includes loading, thresholding, and labeling the particles in the image. It also illustrates the use of functions to make specific measurements (such as area, perimeter, etc.) of detected particles. `Sample2` can only be used with LabWindows/CVI 4.0.1 or later.

Chapter 3

Management Functions

This chapter describes the IMAQ Vision management functions. Management functions initialize the IMAQ Vision subsystem and create, list, and dispose of image structures. These functions also include error handling for all the IMAQ Vision functions.

IPI_InitSys

```
IPIError = IPI_InitSys(void);
```

Purpose

This function initializes the IMAQ Vision memory subsystem. It is implicitly called by `IPI_Create()`. One of these two functions must be called prior to any other IMAQ Vision function. You only have to use this function if you want to use an IMAQ Vision function before creating an image.

IPI_Create

```
IPIError = IPI_Create (IPIImageRef *image_ptr, IPIPixelType
pixel_type, int border_size);
```

Purpose

This function creates an image structure reference. It is the only way to create a IMAQ Vision image. An image reference is simply called `image` in the rest of this manual.

After the image is created, its size is NULL. If you want to fill the image pixels by yourself without using one of the IMAQ Vision functions, you have to resize the image pixel space using `IPI_SetImageSize()` and then get the pixel pointer and mapping using `IPI_GetImageInfo()`.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`, `COMPLEX`

Input

`pixel_type` indicates the data format of the pixels within the image. The only way to change the image type after creation is to call one of the conversion functions (see sections *IPI_Convert* and *IPI_Cast* in Chapter 5, *Tools Functions*). The most common image type has pixels coded using an 8-bit unsigned `char`, and is called `IPI_PIXEL_U8` in IMAQ Vision. However, you can use any of the following predefined values:

- `IPI_PIXEL_U8`—unsigned 8-bit
- `IPI_PIXEL_I16`—signed 16-bit
- `IPI_PIXEL_SGL`—single floating point (32-bit) pixels
- `IPI_PIXEL_RGB32`—32-bit color pixels
- `IPI_PIXEL_COMPLEX`—two single floating point (64-bit) pixels

`border_size` determines the width in pixels of the border created around an image. These pixels are used only by specific functions related to morphology or filtering. Unless you use 7x7 or bigger morphology or convolution process, a border size of 2 is sufficient for all IMAQ Vision functions.

Output

`image_ptr` returns the image structure reference that is supplied as input to all functions used by IMAQ Vision.

IPI_Dispose

```
IPIError = IPI_Dispose(IPIImageRef image);
```

Purpose

This function discards an image and reallocates the occupied space in memory. It must be used for each created image to free the memory allocated by `IPI_Create()`.

Image Type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`, `COMPLEX`

Input

`image` is the image to be disposed.

IPI_SetErrorMode

```
IPIError = IPI_SetErrorMode (IPIErrorMode mode);
```

Purpose

This function sets up the IMAQ Vision behavior when an error occurs.

Input

`mode` indicates your selected error mode:

- `IPI_ERRORMODE_ALERT`—alert dialog (default as startup)
- `IPI_ERRORMODE_IGNORE`—no alert dialog

IPI_GetErrorMode

```
IPIErrorMode = IPI_GetErrorMode(void);
```

Purpose

This function returns the current error mode. The error mode is programmed using `IPI_SetErrorMode`.

IPI_GetLastError

```
IPIError = IPI_GetLastError(char *procName);
```

Purpose

This function returns the last error recorded using `IPI_ProcessError()`. After reading, the error is cleared.

Input

`procName` is the name of the IMAQ Vision function where the last error has occurred.

IPI_ProcessError

```
IPIError = IPI_ProcessError (char *procName, IPIError error);
```

Purpose

This function is called internally by every IMAQ Vision function to record errors. `IPI_SetErrorMode()` determines the behavior of this function (for example, whether a warning dialog appears).

Input

`procName` is the C string containing the calling function name. This name appears in the error dialog.

`error` is the error code to record.

IPI_CloseSys

```
IPIError = IPI_CloseSys(void);
```

Purpose

This function clears the IMAQ Vision memory subsystem. It disposes of memory allocated by IMAQ Vision and returns it to the initial environment.

Chapter 4

Display and File Functions

This chapter describes the IMAQ Vision display and file functions.

Display

Controlling image visualization is of primary importance in an imagery application. Be aware that image processing and image visualization are distinct and separate elements. Image visualization deals only with the presentation of image data to you and how you work with the visualized images. Notice that a typical imagery application has many more images than image windows.

Because people have different imagery needs and skills, IMAQ Vision has a full set of functions that make it very easy to display images and to manage image windows. The novice user can easily access the basic `IPI_WindDraw()` functions while OEMs and other professional users can create imagery applications containing sophisticated display and control capabilities.

With the basic functions, you can display the images in image windows and position, open, and close them on the screen. These image windows can be resized and you can place scroll bars in these image windows. You can also program when to display the image data. Notice that these image windows are based on LabWindows panels and canvas objects that appear as a special subset. Only IMAQ Vision functions manage these windows.

The other features allow you to manage user interaction on image windows, including drawing shapes and selecting tools. These tools can be used to physically access the image data visualized in the image window. They include points, lines, rectangles, ovals, freehand, multi-segment lines, and free objects. You can convert data accessed with these tools into a region of interest or *ROI*. The functions also regulate user interaction in the IMAQ Vision image windows and the events that occur in those image windows.

With this library, you can complete the following tasks:

- manage a tools windows (WindTools)
- select a region tool for defining a region of interest (ROI)
- manage a standard palette of display tools
- get both the events generated by a user and the associated data from an image window

☞ **Note:** ***The display functions use parts of the LabWindows toolbox. You must add the*** `toolbox.obj` ***in your LabWindows project.***

Display Basics

The following functions control the basics of image display.

IPI WindDraw

```
IPIError = IPI_WindDraw(IPIImageRef image, int window_number, char
*window_title, int resize_window);
```

Purpose

This function displays an image in an image window. The image window appears automatically when the function executes.

By using an 8-bit image buffer (`Tmp`), this function displays 16-bit and floating point images. This 8-bit image buffer is calculated as a function of the dynamic range from the image source. The function automatically calculates the minimum value (`Min`), the maximum value (`Max`) and then the following formula is applied to each pixel:

$$\text{Tmp}(x, y) = [\text{Source}(x, y) - \text{Min}] * 255 / (\text{Max} - \text{Min})$$

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`, `COMPLEX`

Input

`image` indicates the image to display.

`window_number (0..15)` indicates the image window where the image is displayed. Up to 16 windows can be displayed simultaneously. Each window is labeled with a value ranging from 0 to 15. Only the image window indicated in `image` is affected by the function.

`window_title` is the image window name. If the string is not empty, the image window automatically takes that name. The default name for the image window is `Image #<Window Number>`.

`resize_window` indicates if you want to automatically resize the image window to fit the image size. With this, you do not have to know the size of a source image prior to displaying it.

IPI_WSetPalette

```
IPIError = IPI_WSetPalette(int window_number, IPIPalette palette,
int color_table[]);
```

Purpose

This function sets a color palette to an image window.

Input

`window_number (0..15)` uses a number from 0 to 15 to indicate the image window to use.

`palette` indicates one of the five predefined palettes or a user color table.

- `IPI_PLT_GRAY`—gray. Gray scale is the default palette. The color tables for the red, green, and blue planes are identical.
- `IPI_PLT_BINARY`—binary palette is designed especially for binary images
- `IPI_PLT_GRADIENT`—gradient palette
- `IPI_PLT_RAINBOW`—rainbow palette
- `IPI_PLT_TEMPERATURE`—temperature palette
- `IPI_PLT_USER`—user palette

`color_table` is the address of your color table. If used, this table contains 256 integers specifying the RGB color corresponding to the 256 possible pixel values. A specific color is the result of a value between 0 and 255 for each of the three color planes: red, green, and blue. If the three planes have the identical values, a shade of gray between (0,0,0) = black and (255,255,255) = white results.

IPI_SetWindowAttributes

```
IPIError = IPI_SetWindowAttribute(int window_number, int
window_attribute, ...);
```

Purpose

This function changes an attribute of one image window.

Input

`window_number (0..15)` indicates the image window to use. It is indicated by a number from 0 to 15.

`window_attribute` is the attribute value to set. This can be one of the standard LabWindows attributes used by IMAQ Vision for image windows or an IMAQ Vision for LabWindows/CVI specific attribute. Following are the standard attributes:

- `ATTR_VISIBLE`—visible
- `ATTR_LEFT`—left
- `ATTR_TOP`—top
- `ATTR_WIDTH`—width
- `ATTR_HEIGHT`—height
- `ATTR_TITLEBAR_VISIBLE`—title bar visible
- `ATTR_TITLE`—title
- `ATTR_SCROLL_BARS`—scroll bars
- `ATTR_HSCROLL_OFFSET`—horizontal scroll bar offset
- `ATTR_VSCROLL_OFFSET`—vertical scroll bar offset
- `ATTR_CAN_MAXIMIZE`—can maximize
- `ATTR_CAN_MINIMIZE`—can minimize
- `ATTR_CLOSE_ITEM_VISIBLE`—close item visible
- `ATTR_FLOATING`—floating window
- `ATTR_MOVABLE`—movable window
- `ATTR_SIZABLE`—sizable window

The IMAQ Vision for LabWindows/CVI specific attributes are the following:

- `IPI_ATTR_VZOOM`—Vertical zoom ratio
- `IPI_ATTR_HZOOM`—Horizontal zoom ratio

- `IPI_ATTR_VGRID`—Vertical drawing grid
- `IPI_ATTR_HGRID`—Horizontal drawing grid

`attribute_value` contains the value corresponding to the attribute to change. With `ATTR_TITLE`, this parameter is a `char*`. For all the other attributes, it is an integer value.

IPI_SetWindow2DAttributes

```
IPIError = IPI_SetWindow2DAttributes(int window_number, int
window_2D_attribute, int V_attribute_value, int H_attribute_value);
```

Purpose

This function changes the attribute of the two axes of the image window.

Input

`window_number (0..15)` indicates the image window to use by a number from 0 to 15.

`window_2D_attribute` is one of the following predefined 2D attributes:

- `IPI_ATTR_TOP_AND_LEFT`—top and left
- `IPI_ATTR_HEIGHT_AND_WIDTH`—height and width
- `IPI_ATTR_VH_SCROLL_OFFSET`—Vertical and Horizontal scroll bar offset
- `IPI_ATTR_VH_ZOOM`—Vertical and Horizontal zoom ratio
- `IPI_ATTR_VH_GRID`—Vertical and Horizontal drawing grid

`V_attribute_value` is the vertical attribute value.

`H_attribute_value` is the horizontal attribute value.

IPI_GetWindowAttribute

```
IPIError = IPI_GetWindowAttribute(int window_number, int
window_attribute, void *attribute_value);
```

Purpose

This function reads image window attributes.

Input

window_number (0..15) indicates the image window to use by a number from 0 to 15.

window_attribute is the attribute value to set. This can be one of the standard LabWindows attributes used by IMAQ Vision for LabWindows/CVI for image windows or an IMAQ Vision for LabWindows/CVI specific attribute. Following are the standard attributes:

- ATTR_VISIBLE—visible
- ATTR_LEFT—left
- ATTR_TOP—top
- ATTR_WIDTH—width
- ATTR_HEIGHT—height
- ATTR_TITLEBAR_VISIBLE—title bar visible
- ATTR_TITLE—title
- ATTR_SCROLL_BARS—scroll bars
- ATTR_HSCROLL_OFFSET—horizontal scroll bar offset
- ATTR_VSCROLL_OFFSET—vertical scroll bar offset
- ATTR_CAN_MAXIMIZE—can maximize
- ATTR_CAN_MINIMIZE—can minimize
- ATTR_CLOSE_ITEM_VISIBLE—close item visible
- ATTR_FLOATING—floating window
- ATTR_MOVABLE—movable window
- ATTR_SIZABLE—sizable window

The IMAQ Vision for LabWindows/CVI specific attributes are the following:

- IPI_ATTR_VZOOM—Vertical zoom ratio
- IPI_ATTR_HZOOM—Horizontal zoom ratio
- IPI_ATTR_VGRID—Vertical drawing grid
- IPI_ATTR_HGRID—Horizontal drawing grid

Output

attribute_value returns the value corresponding to the attribute. With the attribute ATTR_TITLE, this parameters is a char*. For all the other attributes the value is an integer.

IPI_GetWindow2DAttributes

```
IPIError = IPI_GetWindow2DAttributes (int window_number, int
window_2D_attribute, int *V_attribute_value, int
*H_attribute_value);
```

Purpose

This function reads 2D window attributes.

Input

`window_number (0..15)` indicates the image window to use. It is indicated by a number from 0 to 15.

`window_2D_attribute` is one of the following predefined 2D attributes:

- `IPI_ATTR_TOP_AND_LEFT`—top and left
- `IPI_ATTR_HEIGHT_AND_WIDTH`—height and width
- `IPI_ATTR_VH_SCROLL_OFFSET`—vertical and horizontal scroll bar offset
- `IPI_ATTR_VH_ZOOM`—vertical and horizontal zoom ratio
- `IPI_ATTR_VH_GRID`—vertical and horizontal drawing grid

Output

`V_attribute_value` points to the vertical attribute value.

`H_attribute_value` points to the horizontal attribute value.

IPI_WindClose

```
IPIError = IPI_WindClose (int window_number, int
close_all_windows);
```

Purpose

This function closes an image window. It also destroys the space reserved in memory for the image window.

Input

`window_number (0..15)` indicates the image window to close by a number from 0 to 15.

`close_all_windows` indicates that all the image windows are to be closed, if this value is set to TRUE. If this value is FALSE, only the indicated window is closed.

Display Tools

The following functions control display tools.

IPI_WindToolsSetup

```
IPIError = IPI_WindToolsSetup(int icon_per_line, IPITool
tool_list[], int tool_icon_count, int draw_coordinates);
```

Purpose

The `WindTools` palette is a floating palette where you find tools to create a ROI in the image. This function must be called prior to any other function related to the WindTools. This function controls the configuration and the appearance of the WindTools.

☞ **Note:** ***You can read the coordinates of a selected region with*** `IPI_GetLastEvent`**,** `IPI_GetLastWEvent` ***or by installing a callback procedure using*** `IPI_InstallWCallback()`**.**

The regions tools can be altered by the following keyboard keys:

- <Shift>—forces a straight line when using the *line regions tool*, a square when using the *rectangle regions tool*, a circle when using the *oval regions tool*, or reduces the zoom factor when using the *zoom tool*
- <Shift> before a <click>—adds a ROI
- <Control>—moves a region when you click on the region while pressing the <Control> key
- <Control> before a <click>—displaces a ROI

Input

`icon_per_line` selects the number of visible tool icons per line. This parameter configures the width of the tools window. The number of lines are determined by the number of remaining available icons. The value 4 is recommended.

`tool_list` is a pointer to an array specifying the tool icons to show. Use `IPI_WT_STDLIST` if you want to use the standard full tool icons list. If you want to configure the tool icons list yourself, you can create your own array putting the following values in the order you want:

- `IPI_WT_POINT`—point regions tool. You can select a pixel in the image.
- `IPI_WT_LINE`—line regions tool. You can draw a line in the image.
- `IPI_WT_RECTANGLE`—rectangle regions tool. You can draw a rectangle (or square) in the image.
- `IPI_WT_OVAL`—oval regions tool. You can draw an oval (or circle) in the image.
- `IPI_WT_POLYGON`—polygon regions tool. You can draw a polygon in the image.
- `IPI_WT_FREEHAND`—freehand regions tool. You can draw a freehand region in the image.
- `IPI_WT_ZOOM`—zoom. You can zoom in or zoom out in an image.
- `IPI_WT_BROKENLINE`—broken line
- `IPI_WT_FREE`—free hand line

`tool_icon_count` indicates the number of used icons. Use `IPI_ALL_WTOOLS` if you want to configure the WindTools with all existing tool icons.

`draw_coordinates` indicates if the active pixel coordinates and the drawing coordinates are shown.

IPI_SetWindToolsAttribute

```
IPIError = IPI_SetWindToolsAttribute(int window_attribute, ...);
```

Purpose

This function sets the tool attributes.

Input

`window_attribute` indicates one of the following predefined attributes:

- `ATTR_VISIBLE`—visible
- `ATTR_LEFT`—left
- `ATTR_TOP`—top
- `ATTR_TITLEBAR_VISIBLE`—title bar visible
- `ATTR_TITLE`—title
- `ATTR_CLOSE_ITEM_VISIBLE`—close item visible

- `ATTR_FLOATING`—floating window
- `ATTR_MOVABLE`—movable

`attribute_value` contains the value corresponding to the attribute to change. With `ATTR_TITLE`, this parameter is a `char*`. For all the other attributes, it is an integer value.

IPI_GetWindToolsAttribute

```
IPIError = IPI_GetWindToolsAttribute(int window_attribute, void
*attribute_value);
```

Purpose

This function reads all the tool attributes.

Input

`window_attribute` indicates one of the following predefined attributes:

- `ATTR_VISIBLE`—visible
- `ATTR_LEFT`—left
- `ATTR_TOP`—top
- `ATTR_TITLEBAR_VISIBLE`—title bar visible
- `ATTR_TITLE`—title
- `ATTR_CLOSE_ITEM_VISIBLE`—close item visible
- `ATTR_FLOATING`—floating window
- `ATTR_MOVABLE`—movable

Output

`attribute_value` contains the current value of the selected attribute.

IPI_SetActiveTool

```
IPIError = IPI_SetActiveTool(IPITool tool);
```

Purpose

This function selects the current active tool on the image window. If the WindTools palette is visible, the selected tool becomes the active icon in the WindTools palette.

☞ **Note:** ***Only one of the tools passed in the*** `tool_list` ***parameter when calling*** `IPI_WindToolsSetup()` ***can be chosen.***

Input

`tool` must be one of the following values:

- `IPI_WT_NOSELECTION`—no tool selected. You are unable to draw any ROI in the image.
- `IPI_WT_POINT`—point (click) regions tool. You can select a pixel in the image.
- `IPI_WT_LINE`—line regions tool. You can draw a line in the image.
- `IPI_WT_RECTANGLE`—rectangle regions tool. You can draw a rectangle (or square) in the image.
- `IPI_WT_OVAL`—oval regions tool. You can draw an oval (or circle) in the image.
- `IPI_WT_POLYGON`—polygon regions tool. You can draw a polygon in the image.
- `IPI_WT_FREEHAND`—freehand regions tool. You can draw a freehand region in the image.
- `IPI_WT_ZOOM`—zoom. You can zoom in or zoom out in an image.
- `IPI_WT_BROKENLINE`—broken line
- `IPI_WT_FREE`—free hand line

IPI_GetActiveTool

```
IPIError = IPI_GetActiveTool (IPITool *active_tool);
```

Purpose

This function returns the currently selected tool.

Output

`active_tool` contains the current active tool.

IPI_WindToolsClose

```
IPIError = IPI_WindToolsClose (void);
```

Purpose

This function closes the tool palette window.

It works in the same way as `IPI_WindClose`, which closes image windows. This function also destroys the space reserved in memory for the tool palette window.

IPI_InstallWCallback

```
IPIError = IPI_InstallWCallback(int window_number, IPIWCallbackFunc
Callback_Function,void *Callback_Data);
```

Purpose

This function connects a callback function receiving all the user and system events coming from an image window.

Input

`window_number` indicates the image window number on which you want to install a callback function. Use the predefined value `IPI_ALL` to connect a common callback function associated with all events coming from all image windows.

`Callback_Function` is the address of the callback function you want to install. You must declare the callback function using the following prototype:

```
void myWindCallBack(IPIWindEventRecord *event_record, void
*Callback_Data);
```

`Callback_Data` is a user-defined data value you retrieve in your callback function.

IPI_RemoveWCallback

```
void IPI_RemoveWCallback (int window_number);
```

Purpose

This function deletes a window callback function.

Input

`window_number` indicates the image window number on which you want to remove a callback function. The predefined value `IPI_ALL` removes the current common callback function.

IPI_GetLastEvent

```
int IPI_GetLastEvent(IPIWindEventRecord *event_record);
```

Purpose

This function retrieves the last event on all image windows.

Output

`event_record` is a structure filled with the following information:

- windowNumber—image window receiving the event
- event—event type
- usedTool—tool used to generate the event
- coordinates[4]—array containing draw coordinates values
- otherData[4]—other data values

`event_record` returns the occurred event. Table 4-1 below shows the coordinate content according to the type of event and the tool used.

Table 4-1. Event/Tool Coordinates

Event	Tool	Coordinates	Other Parameters
IPI_EVT_NOEVENT	n/a	not filled	not filled
IPI_EVT_CLICK	IPI_WT_POINT	[0,1] = click point	[0,1,2] = pixel value(*)
	IPI_WT_ZOOM	[0,1] = click point [2,3] = image center	[0] = zoom factor
IPI_EVT_DRAW	IPI_WT_LINE	[0,1] = starting point [2,3] = ending point	[0,1] = width and height [2] = vertical segment angle [3] = segment length
	IPI_WT_RECTANGLE	[0,1] = starting point [2,3] = ending point	[0,1] = width and height
	IPI_WT_OVAL	[0,1] = left/top bounding point [2,3] = right/bottom bounding point	[0,1] = width and height

Table 4-1. Event/Tool Coordinates (Continued)

Event	Tool	Coordinates	Other Parameters
IPI_EVT_DRAW (continued)	IPI_WT_POLYGON	[0,1] = left/top bounding point [2,3] = right/bottom bounding point	[0,1] = width and height
	IPI_WT_FREEHAND	[0,1] = left/top bounding point [2,3] = right/bottom bounding point	[0,1] = width and height
	IPI_WT_BROKENLINE	[0,1] = left/top bounding point [2,3] = right/bottom bounding point	[0,1] = width and height
	IPI_WT_FREE	[0,1] = left/top bounding point [2,3] = right/bottom bounding point	[0,1] = width and height
IPI_EVT_MOVE	n/a	[0,1] = position of image window	empty
IPI_EVT_SIZE	n/a	[0,1] = width and height of image window	empty
IPI_EVT_SCROLL	n/a	[0,1] = center position of image	empty
IPI_EVT_ACTIVATE	n/a	empty	empty
IPI_EVT_CLOSE	n/a	empty	empty

(*) Pixel values are stored in the first element of the array for 8-bit, 16-bit, and floating point images.

The RGB values of color images are stored in the order [0,1,2].

The real and imaginary values of a complex image are stored in the order [0,1].

IPI_GetLastWEvent

```
int IPI_GetLastWEvent (int window_number, IPIWindEventRecord
*event_record);
```

Purpose

This function reads the last event on a window.

Input

`window_number (0..15)` indicates the image window used. It is a number from 0 to 15.

Output

`event_record` is a structure filled with the following information:

- windowNumber—image window receiving the event
- event—event type
- usedTool—tool used to generate the event
- coordinates[4]—array containing draw coordinates values
- otherData[4]—other data values

`event_record` returns the occurred event. Table 4-1, *Event/Tool Coordinates*, shows the coordinate content according to the type of event and the tool used.

Regions of Interest

Regions of interest can be used to focus your processing and analysis on a part of an image. A ROI can be traced using standard contours (oval, rectangle, and so on) or freehand contours. The IMAQ Vision user has the following options:

- associate a ROI with an image window
- extract a ROI associated with an image window
- erase the current ROI from an image window
- transform a ROI into an image mask
- transform an image mask into a ROI

An image mask that is converted into a ROI must have an offset. The image mask uses the offset to associate the ROI with an image window that contains image data. In other words, the offset places a newly created ROI into the space of another image. The offset defines the upper left corner coordinates (*x*,*y*) for the bounding rectangle belonging to the ROI. The default value is (0,0).

The ROI descriptor is described as follows:

- Bounding rectangle for a ROI
- Regions list:
  - contour identifier (exterior or interior contour)
  - contour type (point, line, rectangle, oval, freehand, and so on)
  - list of points (*x*,*y*) describing the contour

IPI_SetWROI

```
IPIError = IPI_SetWROI(int window_number, IPIROIPtr ROI);
```

Purpose

This function associates a ROI with an image window.

Input

`window_number (0..15)` indicates the image window to use. It is a number from 0 to 15.

`ROI` is the address of an array describing a ROI.

IPI_GetWROI

```
IPIError = IPI_GetWROI(int window_number, IPIROIPtr *ROI);
```

Purpose

This function reads a ROI associated with an image window.

Input

`window_number (0..15)` indicates the image window to use. It is a number from 0 to 15.

Output

`ROI` returns the address of an array describing the ROI. The `ROI` structure and substructure are allocated (or reallocated) within this function. This `ROI` must be deleted using `IPI_FreeROI()`.

IPI_ClearWROI

```
IPIError = IPI_ClearWROI (int window_number);
```

Purpose

This function erases the current ROI from an image window.

☞ **Note:** ***It is also possible to erase a region of interest in an image window by pressing the backspace key when the current image window is active.***

Input

`window_number (0..15)` indicates the image window to use. It is a number from 0 to 15.

IPI_ROIToMask

```
IPIError = IPI_ROIToMask (IPIImageRef image, IPIImageRef
size_model_image, IPIROIPtr ROI, int filling_value);
```

Purpose

This function transforms a ROI into a mask.

☞ **Note:** ***There are two ways to use this function. The simplest technique is to define the*** `size_model_image`***. In this case you can use the source image, where the image ROI was drawn, as a template for the final destination image. As a result, the output image automatically acquires the size of the image and location of the region of interest as found in the original source image.***

However, you do not have to specify a `size_model_image`. In this case the ROI requires an offset which is automatically determined from the upper left corner of the bounding rectangle described by the ROI. The bounding rectangle information is a part of the `ROI` structure.

Image type: `IPI_PIXEL_U8`

Input

`image` is the destination image where the mask is created.

`size_model_image` serves as a template for the destination image where the mask is placed. `image` takes the characteristics of `size_model_image` (size and location of `ROI`) when `size_model_image` is given. However, the `size_model_image` is optional. This can be of any image type used by IMAQ Vision.

`ROI` is the address of an array describing the ROI.

`filling_value` is the pixel value of the mask. All pixels inside the ROI take this value.

IPI_MaskToROI

```
IPIError = IPI_MaskToROI (IPIImageRef image, int
external_edges_only, int max_number_of_points, IPIROIPtr *ROI, int
*too_much_points);
```

Purpose

This function transforms an image mask into a ROI.

Image type: `IPI_PIXEL_U8`

Input

`image` is the image containing the image mask that is transformed into a ROI.

`external_edges_only` indicates if only the external edges are transformed.

`max_number_of_points` is the maximum size of the definition of a ROI. This arbitrary value provides a way to limit memory usage within this function.

Output

`ROI` returns the address of the array describing the ROI. The `ROI` structure and substructure are allocated (or reallocated) within this function. This `ROI` must be disposed using `IPI_FreeROI()`.

`too_much_points` returns TRUE if there are too many points defining the contour of the ROI.

IPI_FreeROI

```
void IPI_FreeROI(IPIROIPtr ROI);
```

Purpose

This function deallocates memory space used for a ROI structure and substructures.

Input

`ROI` is the ROI previously returned by the `IPI_GetWROI()` or `IPI_MaskToROI()` functions.

Files

The following functions control reading and writing images from and to disk files.

IPI_ReadFile

```
IPIError = IPI_ReadFile(IPIImageRef image, char file_name[], int
load_color_table, int color_table[], int *nb_of_colors);
```

Purpose

This function reads an image file. The file format can be any standard format: AIPD, TIFF, or BMP. The read pixels automatically convert to the image type of the input image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `COMPLEX`, `RGB32`

Input

`image` is the image filled with pixels read from the file. The `image` is resized automatically according to the file header information.

`file_name` is the complete path name (**Drive»Directory»File**) of the file to be loaded.

`load_color_table` determines if you want to load the color table present in the file. If loaded, this table is read and made available at the `color_table` output.

Output

`color_table` contains the color table (R,G,B) read from the file if you pass the value TRUE at the `load_color_table` input. To load a color table, you have to allocate the space corresponding to 256 integers (1024 bytes) and use this pointer as `color_table`.

`nb_of_colors` is the number of colors contained in the `color_table`.

IPI_WriteFile

```
IPIError = IPI_WriteFile (IPIImageRef image, char file_name[],
IPIFileFormat format, int color_table[], int nb_of_colors);
```

Purpose

This function writes an image to a file.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `COMPLEX`, `RGB32`

Input

`image` is the image to be written.

`file_name` is the complete path name (**Drive»Directory»File**) of the file to be written.

`format` indicates the standard file format to be created.

- `IPI_FILE_AIPD` creates an AIPD file (the only file format that uses all image types).
- `IPI_FILE_BMP` creates a BMP file, that uses 8-bit or color 24 bit.
- `IPI_FILE_TIFF` creates a TIFF file, that uses 8-bit or color 24 bit.

`color_table` contains the color table to include in the file (BMP and TIFF only).

`nb_of_colors` is the number of colors contained in the `color_table`.

IPI_GetFileInfo

```
IPIError = IPI_GetFileInfo(char file_name[], IPIFileInfo
*file_info);
```

Purpose

This function gets information on the contents of a file. This information is supplied only if the file has a standard file format (AIPD, BMP, TIFF).

Input

`file_name` is the complete path name (**Drive»Directory»File**) of the file.

Output

`file_info` is a pointer to a structure that contains the following information:

- `fileFormat` indicates the file type that was read. It can be any of the following:
  - `IPI_FILE_UNKNOWN`
  - `IPI_FILE_AIPD`
  - `IPI_FILE_BMP`
  - `IPI_FILE_TIFF`
- `bitsPerPixel` indicates how many bits are used per pixel
- `nbPlanes` indicates the number of planes in the image (1 for monochrome images, 2 for complex images, 3 for color images)
- `width` is the horizontal size defined in the header of standard image file formats
- `height` is the vertical size defined in the header of standard image file formats

Chapter 5

Tools Functions

This chapter describes the IMAQ Vision tools functions.

Tools Image

The following functions are a set of diverse functions for the manipulation of images (copy, reduction, expansion, extraction, and so on). Also included are functions that get all information about the image attributes and pixel mapping, operate on individual pixels, and convert images to arrays and vice versa.

IPI_GetImageInfo

```
IPIError = IPI_GetImageInfo (IPIImageRef image, IPIImageInfo
*image_info);
```

Purpose

This function produces information about the image size, calibration, and offset.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`, `COMPLEX`

Input

`image` is the image on which you get information.

Output

`image_info` is a pointer to a structure that contains the following information:

- width—X image size
- height—Y image size
- pixelType—type of pixel in the image
- pixelSize—size of each pixel in bytes
- rawPixels—offset to next line in pixels
- rawBytes—offset to next line in bytes
- border—border size

- pixelSpace—amount of memory used for the pixels in bytes
- xOffset—X start coordinate when used as a mask
- yOffset—Y start coordinate when used as a mask
- unit—representation unit
- xCalib—X calibration ratio
- yCalib—Y calibration ratio
- firstPixelAddress—address of the pixel (0,0)

IPI_SetImageSize

```
IPIError = IPI_SetImageSize (IPIImageRef image, int width, int
height);
```

Purpose

This function modifies the resolution of an image.

☞ **Note:** ***This function reuses the memory space previously occupied by the image pixels. It does not transfer the original image into a new memory space so the original image is lost.***

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`, `COMPLEX`

Input

`image` is the image that is resized.

`width` is the new number of pixels per line.

`height` is the new number of pixels per column.

IPI_SetImageCalibration

```
IPIError = IPI_SetImageCalibration (IPIImageRef image, IPIImageUnit
unit, float x_axis_ratio, float y_axis_ratio);
```

Purpose

This function sets the calibration scale for an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`, `COMPLEX`

Input

`image` is the image that receives the new calibration.

`unit` is the measure unit associated with the image. You can select the following values:

- `IPI_UNIT_UNDEF`—undefined
- `IPI_UNIT_ANGSTROM`—angstr
- `IPI_UNIT_MICROMETER`—micrometer
- `IPI_UNIT_MILLIMETER`—millimeter
- `IPI_UNIT_CENTIMETER`—centimeter
- `IPI_UNIT_METER`—meter
- `IPI_UNIT_KILOMETER`—kilometer
- `IPI_UNIT_MICROINCH`—microinch
- `IPI_UNIT_INCH`—inch
- `IPI_UNIT_FOOT`—foot
- `IPI_UNIT_NAUTICMILE`—nautic mile
- `IPI_UNIT_GROUNDMILE`—ground mile

`x_axis_ratio` indicates the horizontal distance separating two adjacent pixels in the indicated `unit`.

`y_axis_ratio` indicates the vertical distance separating two adjacent pixels in the indicated `unit`.

IPI_SetImageOffset

```
IPIError = IPI_SetImageOffset(IPIImageRef image, int x_offset, int
y_offset);
```

Purpose

This function defines the position of an image mask in relation to the origin of the coordinate system (0,0).

Image type: `IPI_PIXEL_U8`

Input

`image` is the image for which you set the offset.

`x_offset` indicates the new horizontal offset of the image.

`y_offset` indicates the new vertical offset of the image.

IPI_Expand

```
IPIError = IPI_Expand (IPIImageRef source_image, IPIImageRef
dest_image, int x_duplication, int y_duplication, Rect rectangle);
```

Purpose

This function expands an image or part of an image by adjusting the horizontal and vertical resolution.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`

Input

`source_image` is the image to expand.

`dest_image` is the resulting image.

`x_duplication` indicates the number of pixel duplications per column. The column is copied if the default value (1) is used.

`y_duplication` indicates the number of pixel duplications per line. The row is copied if the default value (1) is used.

`rectangle` is a `Rect` structure containing the coordinates and the size of the region to expand.

IPI_Extract

```
IPIError = IPI_Extract(IPIImageRef source_image, IPIImageRef
dest_image, int x_subsample, int y_subsample, Rect rectangle);
```

Purpose

This function extracts (reduces) an image or part of an image by adjusting the horizontal and vertical resolution.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`

Input

`source_image` is the source image to extract.

`dest_image` is the resulting image.

`x_subsample` is the vertical sampling step and defines the columns to be extracted (the horizontal reduction ratio). For example, with an `x_subsample` equal to 3, one out of every three columns are extracted from the `source_image` into the `dest_image`. Each column is extracted if the default value (1) is used.

`y_subsample` is the horizontal sampling step and defines the lines to be extracted (the vertical reduction ratio). Each row is extracted if the default value (1) is used.

`rectangle` is a `Rect` structure containing the coordinates and the sizes of the region to extract.

IPI_Resample

```
IPIError = IPI_Resample(IPIImageRef source_image, IPIImageRef
dest_image, int x_new_size, int y_new_size, Rect rectangle);
```

Purpose

This function resizes the original image to a user-defined resolution. It is useful for displaying a reduced or enlarged image (that is, zoom in/zoom out).

Image type: `IPI_PIXEL_U8`, `RGB32`

Input

`source_image` is the source image to resample.

`dest_image` is the resulting image.

`x_new_size` is the final horizontal size of the image.

`y_new_size` is the final vertical size of the image.

`rectangle` is a `Rect` structure containing the coordinates and the size of the region to extract and resample.

IPI_Copy

```
IPIError = IPI_Copy(IPIImageRef source_image, IPIImageRef
dest_image);
```

Purpose

This function copies the attributes and the pixels of one image into another image of the same type. It is used for keeping an original copy of an image (that is, before processing an image).

☞ **Note:** ***The images to be copied must be of the same type. This function copies the complete definition of the source image and its pixel data to the destination image. It also modifies the border of the destination image so it will be equal to the source image.***

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`, `COMPLEX`

Input

`source_image` is the source image to copy.

`dest_image` is the resulting image.

IPI_ImageToImage

```
IPIError = IPI_ImageToImage (IPIImageRef source_image, IPIImageRef
dest_image, int destination_top, int destination_left);
```

Purpose

This function copies a small image into another larger image. It is useful for making thumbnail sketches from multiple miniature images.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`

Input

`source_image` is the source image to copy.

`dest_image` is the resulting image.

`destination_top` and `destination_left` specify the `dest_image` pixel coordinates where the `source_image` is copied to.

IPI_GetPixelValue

```
IPIError = IPI_GetPixelValue (IPIImageRef image, int x_coordinate,
int y_coordinate, float *pixel_value);
```

Purpose

This function reads or extracts a pixel value from an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image used for this operation.

`x_coordinate` is the horizontal coordinate of the pixel to read.

`y_coordinate` is the vertical coordinate of the pixel to read.

Output

`pixel_value` returns the pixel value.

IPI_SetPixelValue

```
IPIError = IPI_SetPixelValue(IPIImageRef image, int x_coordinate,
int y_coordinate, float pixel_value);
```

Purpose

This function changes the pixel value in an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image to modify.

`x_coordinate` is the horizontal coordinate of the pixel to modify.

`y_coordinate` is the vertical coordinate of the pixel to modify.

`pixel_value` contains the replacement pixel value.

IPI_GetRowCol

```
IPIError = IPI_GetRowCol (IPIImageRef image, int row_or_column, int
row_col_index, int array_format, void *array, int *nb_of_elements);
```

Purpose

This function reads either a row or a column of pixel values from an image into an array, and returns the number of elements in this array.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image used for this operation.

`row_or_column` specifies operation on a row or column. `IPI_ROW` indicates an operation on a row. `IPI_COLUMN` indicates an operation on a column.

`row_col_index` is the row or column number to be extracted.

`array_format` indicates the data type of the array using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

Output

`array` is the pointer to the pixel array allocated by you. It must be big enough to contain all elements.

`nb_of_elements` returns the number of elements copied into the array.

IPI_SetRowCol

```
IPIError = IPI_SetRowCol (IPIImageRef image, int row_or_column, int
row_col_index, int array_format, void *array, int nb_of_elements);
```

Purpose

This function changes the values of pixels in either a row or a column in an image. An array that you define contains the new values of the pixels.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image to modify.

`row_or_column` specifies operation on a row or column. `IPI_ROW` indicates an operation on a row. `IPI_COLUMN` indicates an operation on a column.

`row_col_index` is the row or column number to modify.

`array_format` indicates the data type of the array using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

`array` defines the pointer to the pixel array containing the new pixel values which are copied into the image.

`nb_of_elements` defines the number of elements in the array.

IPI_GetLine

```
IPIError = IPI_GetLine (IPIImageRef image, Point start, Point end,
int array_format, void *array, int *nb_of_elements);
```

Purpose

This function reads a line of pixels from an image into an array and returns the number of elements in this array.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image used for this operation.

`start` is the start point of the line read.

`end` is the end point of the line read.

`array_format` indicates the data type of the array using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

Output

`array` is the pointer to the pixel array allocated by you. It must be big enough to contain all copied elements.

`nb_of_elements` returns the number of copied elements in the array.

IPI_SetLine

```
IPIError = IPI_SetLine(IPIImageRef image, Point start, Point end,
int array_format, void *array, int nb_of_elements);
```

Purpose

This function writes a line of pixel in an image. An array that you define contains the new values of the pixels.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image to modify.

`start` is the start point of the line to modify.

`end` is the end point of the line to modify.

`array_format` indicates the data type of the array using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

`Array` defines the pointer to the pixel array containing the new pixel values which are copied into the image.

`nb_of_elements` defines the number of elements in the array.

IPI_ImageToArray

```
IPIError = IPI_ImageToArray (IPIImageRef image, Rect rectangle, int
array_format, void *array, int *array_x_size, int *array_y_size);
```

Purpose

This function extracts a pixel array from an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image used for this operation.

`rectangle` is a `Rect` structure containing the coordinates and the size of the rectangle to extract from the image.

`array_format` indicates the data type of the array using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

Output

`array` is the pointer to the pixel array allocated by you. It must be big enough to contain all the copied elements.

`array_x_size` returns the horizontal number of copied elements in the array.

`array_y_size` returns the vertical number of copied elements in the array.

IPI_ArrayToImage

```
IPIError = IPI_ArrayToImage (IPIImageRef image, int array_format,
void *array, int array_x_size, int array_y_size);
```

Purpose

This function sets an image from a pixel array. The resulting image is resized to `array_x_size` and `array_y_size`.

☞ **Note:** ***The resulting image is cut to*** `array_x_size` ***and*** `array_y_size`**.**

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image to modify.

`array_format` indicates the data type of the array using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

`array` defines the pointer of the pixel array containing the new pixel values which are copied into the image.

`array_x_size` is the horizontal number of elements in the array.

`array_y_size` is the vertical number of elements in the array.

IPI_GetPixelAddress

```
IPIError = IPI_GetPixelAddress (IPIImageRef image, int
x_coordinate, int y_coordinate, IPIPixelPtr *pixel_address);
```

Purpose

This function returns a pixel address from an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image used for this operation.

`x_coordinate` is the horizontal coordinate of the pixel.

`y_coordinate` is the vertical coordinate of the pixel.

Output

`pixel_address` returns the address of the indicated pixel.

Tools Diverse

The following functions draw shapes into an image.

IPI_DrawLine

```
IPIError = IPI_DrawLine (IPIImageRef source_image, IPIImageRef
dest_image, Point point_1, Point point_2, IPIDrawMode draw_mode,
float gray_level);
```

Purpose

This function draws a line in an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_image` is the source image where the line is drawn.

`dest_image` is the resulting image.

`point_1` and `point_2` are the start point and the end point of the line.

`draw_mode` defines how to draw the object. It can take the following values:

- `IPI_DRAW_FRAME`—frame. Draws the contour using the `gray_level` value.
- `IPI_DRAW_PAINT`—paint. Fills the shape using the `gray_level` value.
- `IPI_INVERT_FRAME`—invert frame. Uses the inverse of the pixel values when drawing the contour.
- `IPI_INVERT_PAINT`—invert paint. Uses the inverse of the pixel values when drawing the whole shape.

`gray_level` is the pixel value used for drawing. This value is not used in the modes `IPI_INVERT_FRAME` or `IPI_INVERT_PAINT`.

IPI_DrawRect

```
IPIError = IPI_DrawRect (IPIImageRef source_image, IPIImageRef
dest_image, Rect rectangle, IPIDrawMode draw_mode, float
gray_level);
```

Purpose

This function draws a rectangle in an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_image` is the source image where the rectangle is drawn.

`dest_image` is the resulting image.

`rectangle` is a `Rect` structure containing the coordinates and the size of the rectangle to draw.

`draw_mode` defines how to draw the object. It can take the following values:

- `IPI_DRAW_FRAME`—frame. Draws the contour using the `gray_level` value.
- `IPI_DRAW_PAINT`—paint. Fills the shape using the `gray_level` value.
- `IPI_INVERT_FRAME`—invert frame. Uses the inverse of the pixel values when drawing the contour.
- `IPI_INVERT_PAINT`—invert paint. Uses the inverse of the pixel values when drawing the whole shape.

`gray_level` is the pixel value used for drawing. This value is not used in the modes `IPI_INVERT_FRAME` or `IPI_INVERT_PAINT`.

IPI_DrawOval

```
IPIError = IPI_DrawOval  (IPIImageRef source_image, IPIImageRef
dest_image, Rect rectangle, IPIDrawMode draw_mode, float
gray_level);
```

Purpose

This function draws an oval in an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_image` is the source image where the oval is drawn.

`dest_image` is the resulting image.

`rectangle` is a `Rect` structure containing the coordinates and the size of the oval to draw.

`draw_mode` defines how to draw the object. It can take the following values:

- `IPI_DRAW_FRAME`—frame. Draws the contour using the `gray_level` value.
- `IPI_DRAW_PAINT`—paint. Fills the shape using the `gray_level` value.
- `IPI_INVERT_FRAME`—invert frame. Uses the inverse of the pixel values when drawing the contour.
- `IPI_INVERT_PAINT`—invert paint. Uses the inverse of the pixel values when drawing the whole shape.

`gray_level` is the pixel value used for drawing. This value is not used in the modes `IPI_INVERT_FRAME` or `IPI_INVERT_PAINT`.

IPI_MagicWand

```
IPIError = IPI_MagicWand (IPIImageRef source_image, IPIImageRef
dest_image, int x_coordinate, int y_coordinate, float tolerance,
int connectivity_8, float replacement_value);
```

Purpose

This function creates an image mask by extracting a region surrounding a reference pixel and using a positive and negative tolerance of intensity variation around the value of the reference pixel. The process searches for all neighboring pixels whose values are found within the tolerance of the reference value.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

`x_coordinate` and `y_coordinate` define the position of the point taken as reference.

`tolerance` is the range of intensity variation for pixels.

`connectivity_8` indicates the connectivity used to determine if a particle is selected.

The connectivity mode determines if an adjacent pixel belongs to the same particle or a different particle. The possible values are:

- TRUE—The function detects particles in connectivity mode 8.
- FALSE—The function detects particles in connectivity mode 4.

`replacement_value` is the value that is assigned to pixels with the tolerance range in the destination image.

Conversion

The following functions perform linear or nonlinear conversion from one image type to another.

IPI_Convert

```
IPIError = IPI_Convert(IPIImageRef source_image, IPIImageRef
dest_image);
```

Purpose

This function converts the image type of `source_image` into the image type of `dest_image`. The image type encoded by `dest_image` defines how the function converts the image. The conversion rules are described below.

- U8 to I16 or SGL—pixel values are copied (0 to 255)
- U8 to RGB32—pixel values are copied into each of the three color planes R, G, and B
- I16 to U8—pixel values < 0 are set to 0. Pixel values between 0 and 255 are copied. Pixel values >255 are set to 255.
- I16 to SGL—pixel values are copied (–32768 to 32767)
- I16 to RGB32—pixel values are copied into each of the three color planes R, G, and B with the same conversion rule as I16 to U8
- SGL to U8—pixel values < 0 are set to 0. Pixel values between 0 and 255 are copied. Pixel values >255 are forced to 255.
- SGL to I16—pixel values <–32768 are set to –32768. Pixel values between –32768 and 32767 are copied. Pixel values >32767 are set to 32767.
- SGL to RGB32—same rule applies as I16 to RGB32

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`, `COMPLEX`

Input

`source_image` is the image to be converted.

`dest_image` is the image resulting from the conversion.

IPI_Cast

```
IPIError = IPI_Cast(IPIImageRef image, IPIPixelType pixel_type);
```

Purpose

This function changes the type of an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`, `COMPLEX`

Input

`image` is the image to be converted.

`pixel_type` determines which type the source image is converted to.

- `IPI_PIXEL_U8`—unsigned 8-bit
- `IPI_PIXEL_I16`—signed 16-bit
- `IPI_PIXEL_SGL`—single floating point (32-bit) pixels
- `IPI_PIXEL_RGB32`—32-bit color pixels
- `IPI_PIXEL_COMPLEX`—two single floating point (64-bit) pixels

IPI_ConvertByLookup

```
IPIError = IPI_ConvertByLookup(IPIImageRef source_image,
IPIImageRef dest_image, int lookup_format, void *lookup_array, int
nb_of_lookup_elements);
```

Purpose

This function converts an image by using a lookup table which is encoded in floating point values.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_image` is the image to be converted.

`dest_image` is the image resulting from the conversion. The image type for `dest_image` can be the following:

- `IPI_PIXEL_SGL`, `IPI_PIXEL_I16` if `source_image` is of type `IPI_PIXEL_U8`
- `IPI_PIXEL_SGL` if `source_image` is of type `IPI_PIXEL_I16`

`lookup_format` indicates the data type of the lookup table using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

`lookup_array` is the reference of the array. It consists of up to 256 elements if `source_image` is of type `IPI_PIXEL_U8` or up to 65,536 elements if the `source_image` is of type `IPI_PIXEL_I16`. This array is completed with values equal to the index if it has less elements than the maximum needed by the image type in `source_image`.

`nb_of_lookup_elements` indicates the number of elements in the lookup array.

Chapter 6

Image Processing Functions

This chapter describes the IMAQ Vision image processing functions. These functions encompass arithmetic and logic operations, image processing, image filtering, morphological operations, image analysis functions (in both the space and frequency domain), and functions for geometric, complex, and color processing of images.

Arithmetic Operators

The following functions perform arithmetic operations between two images or between an image and a constant.

IPI_Add

```
IPIError = IPI_Add (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, float constant);
```

Purpose

This function adds either an image to an image or a constant to an image. If `source_B_image` is equal to `IPI_USECONSTANT`, a constant is added to an image.

The two possibilities are distinguished in the following manner:

dest(x,y) = source A(x,y) + source B(x,y)

or

dest(x,y) = source A(x,y) + constant

☞ **Note:** ***To add a constant to an image, the*** `dest_image` ***must be of the same image type as the*** `source_A_image`***. If the size of one of the two source images is NULL, the result is the copy of the other. If*** `source_A_image` ***and*** `source_B_image` ***are different, the*** `dest_image` ***must match the image type of the source image encoded with the most bits.***

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_A_image` and `source_B_image` are the two input images.

`dest_image` is the resulting image. It can be one of the source images.

`constant` is the value to add to the input `source_A_image` for an operation between an image and a constant. The constant is rounded down if the image is encoded as an integer.

IPI_Subtract

```
IPIError = IPI_Subtract (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, float constant);
```

Purpose

This function subtracts an image from an image or a constant from an image. If `source_B_image` is equal to `IPI_USECONSTANT`, a constant is subtracted from an image.

The two possibilities are distinguished in the following manner:

dest(x,y) = source A(x,y) – source B(x,y)

or

dest(x,y) = source A(x,y) – constant

☞ **Note:** ***To subtract a constant from an image, the*** `dest_image` ***must be of the same image type as the*** `source_A_image`***. If the size of one of the two source images is NULL, the result is the copy of the other. If*** `source_A_image` ***and*** `source_B_image` ***are different, the*** `dest_image` ***must match the image type of the source image encoded with the most bits.***

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_A_image` and `source_B_image` are the two input images.

`dest_image` is the resulting image. It can be one of the source images.

`constant` is the value to subtract from the input `source_A_image` for an operation between an image and a constant. The constant is rounded down if the image is encoded as an integer.

IPI_Multiply

```
IPIError = IPI_Multiply (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, float constant);
```

Purpose

This function multiplies an image by an image or an image by a constant.

If `source_B_image` is equal to `IPI_USECONSTANT`, an operation between an image and a constant is made.

The two possibilities are distinguished in the following manner:

dest(x,y) = source A(x,y) * source B(x,y)

or

dest(x,y) = source A(x,y) * constant

☞ **Note:** ***To multiply a constant and an image, the*** `dest_image` ***must of be the same image type as the*** `source_A_image`***. If the size of one of the two source images is NULL, the result is the copy of the other. If*** `source_A_image` ***and*** `source_B_image` ***are different, the*** `dest_image` ***must match the image type of the source image encoded with the most bits.***

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_A_image` and `source_B_image` are the two input images.

`dest_image` is the resulting image. It can be one of the source images.

`constant` is the value by which to multiply the `source_A_image` for an operation between an image and a constant. The constant is rounded down if the image is encoded as an integer.

IPI_Divide

```
IPIError = IPI_Divide (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, float constant);
```

Purpose

This function divides an image by an image or an image by a constant.

If `source_B_image` is equal to `IPI_USECONSTANT`, an operation between an image and a constant is made.

The two possibilities are distinguished in the following manner:

dest(x,y) = source A(x,y) / source B(x,y)

or

dest(x,y) = source A(x,y) / constant

☞ **Note:** ***To divide an image by a constant, the*** *dest_image* ***must be of the same image type as the*** *source A image****. You cannot divide an image by 0. If the constant is 0, it is automatically replaced by 1. If the size of one of the two source images is NULL, the result is the copy of the other. If*** `source_A_image` ***and*** `source_B_image` ***are different, the*** `dest_image` ***must match the image type of the source image encoded with the most bits.***

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_A_image` and `source_B_image` are the two input images.

`dest_image` is the resulting image. It can be one of the source images.

`constant` is the value of the divider of the input `source_A_image` for an operation between an image and a constant. The constant is rounded down if the image is encoded as an integer.

IPI_Modulo

```
IPIError = IPI_Modulo (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, float constant);
```

Purpose

This function modulo divides between an image and an image or an image and a constant and results in the remainder.

If `source_B_image` is equal to `IPI_USECONSTANT`, the modulo of an image and a constant is produced.

The two possibilities are distinguished in the following manner:

dest(x,y) = source A(x,y) % source B(x,y)

or

dest(x,y) = source A(x,y) % constant

If the `source_A_image` is a floating point image type, the function completes the following operation:

dest(x,y) = source A(x,y) – source B(x,y) * E(source A(x,y) / source B(x,y))

or

dest(x,y) = source A(x,y) – constant * E(source A(x,y) / constant) with E(x) = integer part of x

☞ **Note:** ***To modulo divide an image by a constant, the*** `dest_image` ***must be of the same image type as the*** `source_A_image`***. You cannot divide by zero. If the constant is 0 it is automatically replaced by 1. If the size of one of the two source images is NULL, the result is the copy of the other. If*** `source_A_image` ***and*** `source_B_image` ***are different, the type of the*** `dest_image` ***must correspond to the type of the source image which is encoded with the biggest number of bits.***

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_A_image` and `source_B_image` are the two input images.

`dest_image` is the resulting image. It can be one of the source images.

`constant` is the value to modulo divide by the input `source_A_image` for operation between an image and a constant. The constant is rounded down if the image is encoded as an integer.

IPI_MulDiv

```
IPIError = IPI_MulDiv (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, float
multiplication_constant);
```

Purpose

This function computes a ratio between two images. Each pixel in `source_A_image` is multiplied by the integer value indicated in the `multiplication_constant` before the result of this operation is divided by the equivalent pixel found in `source_B_image`. If the background is lighter than the image, this function can be used to correct the background. In a background correction, `source_A_image` is the acquired image and `source_B_image` is the light background.

dest(*x*,*y*) = (source A(*x*,*y*) * `multiplication_constant`) / source B(*x*,*y*)

The two input images must be of the same image type. If `source_B_image` is equal to `IPI_USECONSTANT`, an operation occurs between an image and a constant.

☞ **Note:** ***To complete this function with an image and a constant, the output*** `dest_image` ***must be of the same image type as the input*** `source_A_image`***. You cannot divide by 0. If the constant is 0 it is automatically replaced by 1. If one of the two source images is empty, the result is the copy of the other. If*** `source_A_image` ***and*** `source_B_image` ***are different, the type of the*** `dest_image` ***must correspond to the type of the source image which is encoded with the biggest number of bits.***

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_A_image` and `source_B_image` are the two input images.

`dest_image` is the resulting image. It can be one of the source images.

`multiplication_constant` is the value to be multiplied by each pixel in `source_A_image` prior to dividing by the equivalent pixel in `source_B_image`. The value 255 corresponds to the maximum value for a pixel encoded in an 8-bit image.

Logic Operators

The following functions perform logic operations between two images or between an image and a constant.

IPI_And

```
IPIError = IPI_And (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, int And_or_Nand, int
constant)
```

Purpose

This function computes the intersection between two images. If `source_B_image` is equal to `IPI_NOIMAGE`, an operation occurs between an image and a constant.

The function completes the following operation for each pixel (*x*,*y*) :

dest(*x*,*y*) = sourceA(*x*,*y*) AND sourceB(*x*,*y*)

dest(*x*,*y*) = sourceA(*x*,*y*) AND constant if `source_B_image` is equal to `IPI_NOIMAGE`

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`. All input images must have the same image type.

Input

`source_A_image` and `source_B_image` are the two source images.

`dest_image` is the resulting image.

`And_or_Nand` is set to TRUE if you want the logic operator NAND to occur instead of AND.

`constant` is the binary constant used for an operation between an image and a constant.

IPI_Or

```
IPIError = IPI_Or (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, int Or_or_Nor, int
constant);
```

Purpose

This function computes the union between two images. If `source_B_image` is equal to `IPI_NOIMAGE`, an operation occurs between an image and a constant.

The function completes the following operation for each pixel (*x*,*y*):

dest(*x*,*y*) = sourceA(*x*,*y*) OR sourceB(*x*,*y*)

dest(*x*,*y*) = sourceA(*x*,*y*) OR constant if `source_B_image` is equal to `IPI_NOIMAGE`

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`. All input images must have the same image type.

Input

`source_A_image` and `source_B_image` are the two input images.

`dest_image` is the resulting image.

`Or_or_Nor` is set to TRUE if you want the logic operator NOR to occur instead of OR.

`constant` is the binary constant used for an operation between an image and a constant.

IPI_Xor

```
IPIError = IPI_Xor (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, int Xor_or_Xnor, int
constant);
```

Purpose

This function selects the pixels which are lit only in one of these two images. If `source_B_image` is equal to `IPI_NOIMAGE`, an operation between an image and a constant is made.

The function completes the following operation for each pixel (*x*,*y*):

dest(*x*,*y*) = sourceA(*x*,*y*) XOR sourceB(*x*,*y*)

dest(*x*,*y*) = sourceA(*x*,*y*) XOR constant if `source_B_image` is equal to `IPI_NOIMAGE`.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`. All input images must have the same image type.

Input

`source_A_image` and `source_B_image` are the two input images.

`dest_image` is the resulting image.

`Xor_or_Xnor` is set to TRUE if you want the logic operator XNOR to occur instead of XOR.

`constant` is the binary constant used for an operation between an image and a constant.

IPI_Mask

```
IPIError = IPI_Mask (IPIImageRef source_image, IPIImageRef
mask_image, IPIImageRef dest_image);
```

Purpose

This function copies the `source_image` into the `dest_image`. If a pixel value is 0 (OFF) in the `mask_image`, the corresponding pixel in `dest_image` is set to 0.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `COMPLEX`, `RGB32`

Mask image type: `IPI_PIXEL_U8`

Input

`source_image` is the input image which is masked.

`dest_image` is the resulting image.

☞ **Note:** `dest_image` ***must be the same image type as*** `source_image`**.**

`mask_image` is the image which contains the mask applied to the source image. It is considered as a binary image. All pixel values different than zero are ON and all pixel values of 0 are OFF.

IPI_Compare

```
IPIError = IPI_Compare (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, IPICPOperator operator,
float constant);
```

Purpose

This function contains comparison operations between two images or an image and a constant.

If `source_B_image` is equal to `IPI_NOIMAGE`, an operation between an image and a constant is made.

☞ **Note:** ***If one of the two source images is empty, the result is the copy of the other.***

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`. All input images must have the same image type.

Input

`source_A_image` and `source_B_image` are the two source images.

`dest_image` is the resulting image.

`Operator` indicates the comparison operator to use. The valid operators are:

- Average—compute the average
- Min—extract the smallest value
- Max—extract the largest value
- Clear if <—if `source_A_image`(*x*,*y*) < `source_B_image`(*x*,*y*) or a `constant`, `dest_image` (*x*,*y*) = 0 otherwise `dest_image`(*x*,*y*) = `source_A_image`(*x*,*y*)
- Clear if <=—if `source_A_image`(*x*,*y*)<= `source_B_image`(*x*,*y*) or a `constant`, `dest_image` (*x*,*y*) = 0 otherwise `dest_image`(*x*,*y*) = `source_A_image`(*x*,*y*)
- Clear if =—if `source_A_image`(*x*,*y*) = `source_B_image`(*x*,*y*) or a `constant`, `dest_image` (*x*,*y*) = 0 otherwise `dest_image`(*x*,*y*) = `source_A_image`(*x*,*y*)
- Clear if >=—if `source_A_image`(*x*,*y*)>= `source_B_image`(*x*,*y*) or a `constant`, `dest_image` (*x*,*y*) = 0 otherwise `dest_image`(*x*,*y*)= `source_A_image`(*x*,*y*)
- Clear if >—if `source_A_image`(*x*,*y*) > `source_B_image`(*x*,*y*) or a `constant`, `dest_image` (*x*,*y*) = 0 otherwise `dest_image`(*x*,*y*) = `source_A_image`(*x*,*y*)

`constant` is the value used in comparison with `source_A_image` for the image/constant operations.

IPI_LogDiff

```
IPIError = IPI_LogDiff (IPIImageRef source_A_image, IPIImageRef
source_B_image, IPIImageRef dest_image, int constant);
```

Purpose

The function completes the following operation for each pixel (*x*,*y*):

dest(*x*,*y*) = sourceA(*x*,*y*) AND NOT (sourceB(*x*,*y*))

dest(*x*,*y*) = sourceA(*x*,*y*) AND NOT constant if the source B image is equal to `IPI_NOIMAGE`

If `source_B_image` is equal to `IPI_NOIMAGE`, an operation between an image and a constant is made.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`. All input images must have the same image type.

Input

`source_A_image` and `source_B_image` are the two source images.

`dest_image` is the resulting image.

`constant` is a constant value that can replace `source_B_image` for the image/constant operation.

Processing

The following functions perform image processing.

IPI_Label

```
IPIError = IPI_Label (IPIImageRef source_image, IPIImageRef
dest_image, int connectivity_8, int *labelled_particles);
```

Purpose

This function labels the particles in an image. This operation assigns a value to all pixels that compose the same group of pixels (i.e. a particle). This color level is encoded in 8 or 16 bits depending on the image type. As a result, 255 particles can be labelled in an 8-bit image and 65,535 particles in a 16-bit image. In the case where you want to label more than 255 particles in an 8-bit image, it is necessary to apply a threshold with an interval of (255,255) after processing the first 254 particles. The aim of this threshold is to eliminate the first 254 particles to visualize the next 254 particles.

☞ **Note:** ***The source and destination images must be of the same image type and their borders must be greater than or equal to 2.***

Image type: `IPI_PIXEL_U8`, `I16`

Input

`source_image` is the image to process.

`dest_image` is the resulting image.

`connectivity_8` indicates the connectivity used for particle detection. The connectivity mode directly determines if an adjacent pixel belongs to the same particle or a different particle.

Possible values are:

- TRUE—The function completes particle detection in connectivity mode 8.
- FALSE—The function completes particle detection in connectivity mode 4.

Output

`labelled_particles` contains the number of particles detected in the image.

IPI_Threshold

```
IPIError = IPI_Threshold (IPIImageRef source_image, IPIImageRef
dest_image, float min_value, float max_value, float new_value, int
do_replacement);
```

Purpose

This function applies a threshold to an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_image` is the image to process.

`dest_image` is the resulting image.

`min_value` is the lowest pixel value considered.

`max_value` is the highest pixel value considered.

`new_value` replaces all values found between this range. Values outside this range are set to 0.

`do_replacement` determines if another value is to replace the pixels existing in the range between `min_value` and `max_value`. TRUE means replacement. FALSE means no replacement.

☞ **Note:** ***Use a binary palette when you plan to visualize an image to which a threshold has been applied. The palette to use for visualization depends on the value of*** `new_value` ***and*** `do_replacement`***. For example, a threshold image can be displayed with a gray palette. However, with a high replacement value, such as 255 (white), you can actually see the displayed result.***

IPI_MultiThreshold

```
IPIError = IPI_MultiThreshold (IPIImageRef source_image,
IPIImageRef dest_image, int threshold_count, IPIThresholdData
threshold_data[]);
```

Purpose

This function applies multiple thresholds to an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_image` is the image to process.

`dest_image` is the resulting image.

`threshold_count` indicates the number of threshold ranges passed in `threshold_data`.

`threshold_data` is an array of clusters specifying the mode and threshold range. Each cluster is composed of the following elements:

- `minValue` is the lowest pixel value to be taken into account.
- `maxValue` is the highest pixel value to be taken into account.

- `newValue` is the replace value for all pixels between the two previous values.
- `doReplacement` determines if the pixels existing in the range between `minValue` and `maxValue` are to be replaced by another value. TRUE means replacement. FALSE means no replacement.

☞ **Note:** ***The threshold operations are completed in the*** `threshold_data` ***order. A pixel can only be taken into account once, even if the pixel is included in the threshold range of two different*** `threshold_data`**.**

Example

The threshold data contains the following two structures:

- `minValue` = 80, `maxValue` = 150, `newValue` = 255, `doReplacement` = TRUE
- `minValue` = 120, `maxValue` = 200, `doReplacement` = FALSE

This example shows two threshold ranges which overlap between 120 and 150. The pixels between 120 and 150 are affected only by the first threshold. The following results occur after execution of this function. The pixel values between 0 and 79 are replaced by 0, the pixel values between 80 and 150 are replaced by 255, the pixel values between 151 and 200 keep their original values, and the pixel values greater than 200 are set to 0.

IPI_AutoBThreshold

```
IPIError = IPI_AutoBThreshold (IPIImageRef image, IPIATMethod
method, int lookup_format, void *lookup_ptr, int *threshold_value);
```

Purpose

The Automatic Binary Threshold applies a threshold to an image that initially possesses 256 gray levels that divides the image into two classes. A statistical calculation is done to determine the optimal threshold.

Image type: `IPI_PIXEL_U8`

Input

`image` is the image to process.

`method` indicates the threshold method you want to use:

- `IPI_AT_CLUSTER`—clustering method
- `IPI_AT_ENTROPY`—entropy method
- `IPI_AT_METRIC`—metric method

- `IPI_AT_MOMENT`—moments method
- `IPI_AT_INTER`—inter variance method

`lookup format` indicates the data type of the lookup table using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

Output

`lookup_ptr` points to a lookup table containing 256 elements encoded in 0 and 1. If the threshold value is 160, the values between 0 and 159 become 0 and the values between 160 and 255 become 1. This array can be directly used by `IPI_UserLookup`.

`threshold_value` returns the computed threshold value.

IPI_AutoMThreshold

```
IPIError = IPI_AutoMThreshold (IPIImageRef image, int
number_of_classes, int lookup_format, void *lookup_ptr,
IPIThresholdData threshold_data[]);
```

Purpose

Automatic Multi-Threshold is a variant of the classification by clustering method.

The method is based on a reiterated measurement of a histogram. Starting from a random sort, the method determines the gray scale values. After finding the best result, it segments the histogram into *n* groups. These groups are based on the fact that each point in a group is closer to the barycenter of its own group than the barycenter of another group.

The function outputs the threshold data in the following two forms:

- A lookup table (LUT) directly usable by `IPI_UserLookup`
- An array directly usable by `IPI_MultiThreshold` (Threshold Data)

Image type: `IPI_PIXEL_U8`

Input

`image` is the image to process.

`number_of_classes` is the number of preferred phases. This algorithm uses a clustering method and can take any value between 2 and 256.

`lookup_format` indicates the data type of the lookup table using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

Output

`lookup_ptr` returns a pointer to a lookup table you can use calling `IPI_UserLookup()`.

This array contains 256 elements encoded between 0 and the `number_of_classes`.

`threshold_data` returns an array containing the `number_of_classes` compatible with `IPI_MultiThreshold`. The results are from 0 to *n* – 1 where *n* is the `number_of_classes.`

IPI_MathLookup

```
IPIError = IPI_MathLookup IPIImageRef source_image, IPIImageRef
mask_image, IPIImageRef dest_image, IPILKOperator operator, float
x_value, float minimum_value, float maximum_value);
```

Purpose

This function converts the pixel values of an image by replacing them with values from a defined lookup table.

This function modifies the dynamic range of either part of an image or the complete image, depending on the type of transformation.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Mask image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to process.

`mask_image` indicates the region in the image to use for computing the histogram. Only pixels in the original image that correspond to a non-NULL pixel in the mask are replaced by the values in the lookup table. A replacement on the complete image occurs if this input is equal to `IPI_NOMASK`.

`dest_image` is the resulting image.

`operator` indicates the mapping procedure to use. The default is 0 or linear. The table below indicates the different available possibilities. The horizontal axis represents the pixel values before processing (between `Min` and `Max`) and the vertical axis represents the pixel values (between `Dynamic_Min` and `Dynamic_Max`) after processing.

- `IPI_LK_LIN`—linear. Linear remapping.
- `IPI_LK_LOG`—logarithmic. Algorithmic remapping operation that results in extended contrast for small pixel values and less contrast for large pixel values.
- `IPI_LK_EXP`—exponential. An exponential remapping operation that results in extended contrast for large pixel values and less contrast for small pixel values.
- `IPI_LK_SQR`—square. Similar to exponential but with a more gradual effect.
- `IPI_LK_SQRT`—square root. Similar to logarithmic but with a more gradual effect.

- `IPI_LK_POWX`—power X. Causes variable effects depending on the value of *X* (default *X* = 1.5).
- `IPI_LK_POW1X`—power 1/X. Causes variable effects depending on the value of *X* (default *X* = 1.5).

`x_value` is a value used for the operators power X and power 1/X only.

`minimum_value` is the smallest value used for processing. After processing, all pixel values equal to or less than the `minimum_value` (in the original image) are set to 0 for an 8-bit image or to the smallest pixel value in the original image for 16-bit and 32-bit images.

`maximum_value` is the largest value used for processing. After processing, all pixel values equal to or less than the `maximum_value` (in the original image) are set to 255 for an 8-bit image or to the largest pixel value in the original image for 16-bit and 32-bit images.

IPI_UserLookup

```
IPIError = IPI_UserLookup (IPIImageRef source_image, IPIImageRef
mask_image, IPIImageRef dest_image, int lookup_format, void
*lookup_array, int nb_of_lookup_elements);
```

Purpose

This function remaps the pixel values in an image.

Image type: `IPI_PIXEL_U8`, `I16`

Mask image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to process.

`mask_image` indicates the region in the image to use for computing the histogram. Only pixels in the original image that correspond to a non-NULL pixel in the mask is replaced by the values in the Lookup table. The complete image is modified if this input is equal to `IPI_NOMASK`.

`dest_image` is the resulting image.

`lookup_format` indicates the data type of the lookup table using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

`lookup_array` is a pointer on a replacement color table.

`nb_of_lookup_elements` indicates the size of the lookup table. This table can contain 256 elements (8-bit) or 65,536 elements (16-bit) depending on the type of `source_image`. Individual pixels within the image are not modified in the case where the lookup is missing a corresponding value.

IPI_Equalize

```
IPIError = IPI_Equalize (IPIImageRef source_image, IPIImageRef
mask_image, IPIImageRef dest_image, int histogram[], int
number_of_classes, IPIHistoReportPtr histogram_report, float
minimum_value, float maximum_value);
```

Purpose

This function creates a histogram equalization of an image. This function redistributes the pixel values of an image to create a linear accumulated histogram. It is necessary to execute `IPI_Histogram` prior to this function to supply `histogram_report` as input for `IPI_Equalize`. The precision of the function depends on the histogram precision, which in turn depends on the number of classes used in the histogram.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Mask image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to process.

`mask_image` indicates the region in the image to use for computing the histogram. Only pixels in the original image that correspond to a non-NULL pixel in the mask is replaced. All pixels not corresponding to this criteria keep their original value. The complete image is modified if this input is equal to `IPI_NOMASK`.

`dest_image` is the resulting image.

`histogram` contains the histogram values in an array. The elements found in this array are the number of pixels per class. The *n* classes contain all pixel values belonging to the interval.

[starting value + (*n* + 1) * interval width, starting value + *n* * interval width – 1].

`number_of_classes` is the number of preferred phases. This algorithm uses a clustering method and can take any value between 2 and 256.

`histogram_report` is the histogram of the source image and is supplied as an output of the `IPI_Histogram` function. No processing occurs if this input is NULL. It is necessary to input the same image to both `IPI_Histogram` and this function.

`minimum_value` is the smallest value used for processing. After processing, all pixel values that were equal to or less than the `minimum_value` (in the original image) are set to 0 for an 8-bit image or to the smallest pixel value found in the original image for 16-bit and 32-bit images.

`maximum_value` (default 0) is the largest value used for processing. After processing, all pixel values that were equal to or less than the `maximum_value` (in the original image) are set to 255 for an 8-bit image or to the largest pixel value found in the original image for 16-bit and 32-bit images.

Filters

Filters are divided into two types: linear (or convolutions) and non-linear.

A linear filter or convolution is a special algorithm that calculates the value of a pixel based on its own pixel value as well as the pixel values of its neighbors. The sum of this calculation is divided by the sum of the elements in the matrix to obtain a new pixel value. The size of the convolution matrix (or kernel) does not have a theoretical limit and can be either square or rectangular (3x3, 5x5, 5x7, 9x3, 127x127, and so on). Filters belong to one of four families: gradient, laplacian, smoothing, and gaussian. This organization is determined by the convolution matrix contents or the weight of each pixel as designated by its geographical location in relation to the central matrix pixel.

IMAQ Vision supplies you with a set of standard convolution kernels for each family and for the usual sizes (3x3, 5x5 and 7x7). They are accessible from the `IPI_GetConvolutionMatrix` function. You can also create your own kernels. You choose the contents of these user-defined kernels. Their size is virtually unlimited. With this capability, you can create special effect filters.

The aim of the non-linear filters is either to extract the contours (edge detection) or to remove isolated pixels. The function `IPI_GrayEdge` has six different methods for contour extraction (differentiation, gradient, Prewitt, Roberts, sigma, Sobel). Two functions, `IPI_NthOrder` and `IPI_LowPass`, can complete the harmonization of pixel values. These functions require that a kernel size and either an order number (`NthOrder`) or a percentage (`LowPass`) be indicated on input.

IPI_GetConvolutionMatrix

```
IPIConvoDescPtr = IPI_GetConvolutionMatrix (IPIConvFamily
convolution_family, int matrix_size, int kernel_number);
```

Purpose

This function returns a pointer to a predefined convolution matrix.

The function `IPI_Convolute()` needs a `convolution_matrix_descriptor`. A *convolution matrix descriptor* is a structure described in the IMAQ Vision header file (`IMAQ_CVI.H`) as:

```
typedef struct {
int matrixWidth;
int matrixHeight;
float *matrixElements;
float divider;
} IPIConvoDesc, *IPIConvoDescPtr;`
```

The first two elements of this structure are used within the `IPI_Convolute()` function to learn the structure of the data array given by `matrixElements`.

The `divider` is a normalization factor that is applied to the sum of the obtained products. Under normal conditions, the divider is the sum of all the matrix elements values.

You can construct your own convolution matrix descriptor; but, in most cases, you can use `IPI_GetConvolutionMatrix()` to obtain directly one of the predefined convolution matrix descriptors.

☞ **Note:** ***A matrix is a 2D array which contains the convolution to be applied to the image. The size of the convolution is fixed by the size of this array.***

A convolution matrix must have odd sized dimensions to contain a central pixel. The function does not take into account the boundary if one of the matrix dimensions is even. For example, if the input matrix is 6x4 ($X = 6$ and $Y = 4$), the actual convolution is 5x3. Both the 6th line and the 4th row are ignored.

The processing speed is correlated with the size of the matrix. A 3x3 convolution processes nine pixels while a 5x5 convolution processes 25 pixels.

Input

`convolution_family` determines the basic family of the convolution matrix. It can be one of the following predefined values:

- `IPI_COFAM_GRADIENT`—gradient
- `IPI_COFAM_LAPLACIAN`—laplacian
- `IPI_COFAM_SMOOTHING`—smoothing
- `IPI_COFAM_GAUSSIAN`—gaussian

`matrix_size` determines the horizontal and vertical matrix size. The values are:

- *3*—corresponding to the convolution with a 3x3 kernel
- *5*—corresponding to the convolution with a 5x5 kernel
- *7*—corresponding to the convolution with a 7x7 kernel

The `convolution_family` and the `matrix_size` determine the matrix type.

`kernel_number` is the number of the selected matrix belonging to this matrix type.

The following table shows the available predefined convolution matrixes:

Table 6-1. Gradient 3x3

matrix # and content	**matrix # and content**	**matrix # and content**	**matrix # and content**
#0 –1 0 1 –1 0 1 –1 0 1	#1 –1 0 1 –1 1 1 –1 0 1	#2 0 1 1 –1 0 1 –1 –1 0	#3 0 1 1 –1 1 1 –1 –1 0
#4 1 1 1 0 0 0 –1 –1 –1	#5 1 1 1 0 1 0 –1 –1 –1	#6 1 1 0 1 0 –1 0 –1 –1	#7 1 1 0 1 1 –1 –0 –1 –1
#8 1 0 –1 1 0 –1 1 0 –1	#9 1 0 –1 1 1 –1 1 0 –1	#10 0 –1 –1 1 0 –1 1 1 0	#11 0 –1 –1 1 1 –1 1 1 0
#12 –1 –1 –1 0 0 0 1 1 1	#13 –1 –1 –1 0 1 0 1 1 1	#14 –1 –1 0 –1 0 1 0 1 1	#15 –1 –1 0 –1 1 1 0 1 1
#16 –1 0 1 –2 0 2 –1 0 1	#17 –1 0 1 –2 1 2 –1 0 1	#18 0 1 2 –1 0 1 –2 –1 0	#19 0 1 2 –1 1 1 –2 –1 0
#20 1 2 1 0 0 0 –1 –2 –1	#21 1 2 1 0 1 0 –1 –2 –1	#22 2 1 0 1 0 –1 0 –1 –2	#23 2 1 0 1 1 –1 0 –1 –2
#24 1 0 –1 2 0 –2 1 0 –1	#25 1 0 –1 2 1 –2 1 0 –1	#26 0 –1 –2 1 0 –1 2 1 0	#27 0 –1 –2 1 1 –1 2 1 0
#28 –1 –2 –1 0 0 0 1 2 1	#29 –1 –2 –1 0 1 0 1 2 1	#30 –2 –1 0 –1 0 1 0 1 2	#31 –2 –1 0 –1 1 1 0 1 2

Table 6-2. Gradient 5x5

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 $\begin{matrix} 0 & -1 & 0 & 1 & 0 \\ -1 & -2 & 0 & 2 & 1 \\ -1 & -2 & 0 & 2 & 1 \\ -1 & -2 & 0 & 2 & 1 \\ 0 & -1 & 0 & 1 & 0 \end{matrix}$	#1 $\begin{matrix} 0 & -1 & 0 & 1 & 0 \\ -1 & -2 & 0 & 2 & 1 \\ -1 & -2 & 1 & 2 & 1 \\ -1 & -2 & 0 & 2 & 1 \\ 0 & -1 & 0 & 1 & 0 \end{matrix}$	#2 $\begin{matrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 & 1 \\ -1 & -2 & 0 & 2 & 1 \\ -1 & -2 & -2 & 0 & 0 \\ -1 & -1 & -1 & 0 & 0 \end{matrix}$	#3 $\begin{matrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 & 1 \\ -1 & -2 & 1 & 2 & 1 \\ -1 & -2 & -2 & 0 & 0 \\ -1 & -1 & -1 & 0 & 0 \end{matrix}$
#4 $\begin{matrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 2 & 2 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & -2 & -2 & -2 & -1 \\ 0 & -1 & -1 & -1 & 0 \end{matrix}$	#5 $\begin{matrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 2 & 2 & 2 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ -1 & -2 & -2 & -2 & -1 \\ 0 & -1 & -1 & -1 & 0 \end{matrix}$	#6 $\begin{matrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 2 & 2 & 0 & 0 \\ 1 & 2 & 0 & -2 & -1 \\ 0 & 0 & -2 & -2 & -1 \\ 0 & 0 & -1 & -1 & -1 \end{matrix}$	#7 $\begin{matrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 2 & 2 & 0 & 0 \\ 1 & 2 & 1 & -2 & -1 \\ 0 & 0 & -2 & -2 & -1 \\ 0 & 0 & -1 & -1 & -1 \end{matrix}$
#8 $\begin{matrix} 0 & 1 & 0 & -1 & 0 \\ 1 & 2 & 0 & -2 & -1 \\ 1 & 2 & 0 & -2 & -1 \\ 1 & 2 & 0 & -2 & -1 \\ 0 & 1 & 0 & -1 & 0 \end{matrix}$	#9 $\begin{matrix} 0 & 1 & 0 & -1 & 0 \\ 1 & 2 & 0 & -2 & -1 \\ 1 & 2 & 1 & -2 & -1 \\ 1 & 2 & 0 & -2 & -1 \\ 0 & 1 & 0 & -1 & 0 \end{matrix}$	#10 $\begin{matrix} 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & -2 & -2 & -1 \\ 1 & 2 & 0 & -2 & -1 \\ 1 & 2 & 2 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{matrix}$	#11 $\begin{matrix} 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & -2 & -2 & -1 \\ 1 & 2 & 1 & -2 & -1 \\ 1 & 2 & 2 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{matrix}$
#12 $\begin{matrix} 0 & -1 & -1 & -1 & 0 \\ -1 & -2 & -2 & -2 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 2 & 2 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{matrix}$	#13 $\begin{matrix} 0 & -1 & -1 & -1 & 0 \\ -1 & -2 & -2 & -2 & -1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 2 & 2 & 2 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{matrix}$	#14 $\begin{matrix} -1 & -1 & -1 & 0 & 0 \\ -1 & -2 & -2 & 0 & 0 \\ -1 & -2 & 0 & 2 & 1 \\ 0 & 0 & 2 & 2 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{matrix}$	#15 $\begin{matrix} -1 & -1 & -1 & 0 & 0 \\ -1 & -2 & -2 & 0 & 0 \\ -1 & -2 & 1 & 2 & 1 \\ 0 & 0 & 2 & 2 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{matrix}$

Table 6-3. Gradient 7x7

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 0 –1 –1 0 1 1 0 –1 –2 –2 0 2 2 1 –1 –2 –3 0 3 2 1 –1 –2 –3 0 3 2 1 –1 –2 –3 0 3 2 1 –1 –2 –2 0 2 2 1 0 –1 –1 0 1 1 0	#1 0 –1 –1 0 1 1 0 –1 –2 –2 0 2 2 1 –1 –2 –3 0 3 2 1 –1 –2 –3 1 3 2 1 –1 –2 –3 0 3 2 1 –1 –2 –2 0 2 2 1 0 –1 –1 0 1 1 0	#2 0 1 1 1 1 1 0 1 2 2 2 2 2 1 1 2 3 3 3 2 1 0 0 0 0 0 0 0 –1 –2 –3 –3 –3 –2 –1 –1 –2 –2 –2 –2 –2 –1 0 –1 –1 –1 –1 –1 0	#3 0 1 1 1 1 1 0 1 2 2 2 2 2 1 1 2 3 3 3 2 1 0 0 0 1 0 0 0 –1 –2 –3 –3 –3 –2 –1 –1 –2 –2 –2 –2 –2 –1 0 –1 –1 –1 –1 –1 0
#4 0 1 1 0 –1 –1 0 1 2 2 0 –2 –2 –1 1 2 3 0 –3 –2 –1 1 2 3 0 –3 –2 –1 1 2 3 0 –3 –2 –1 1 2 2 0 –2 –2 –1 0 1 1 0 –1 –1 0	#5 0 1 1 0 –1 –1 0 1 2 2 0 –2 –2 –1 1 2 3 0 –3 –2 –1 1 2 3 1 –3 –2 –1 1 2 3 0 –3 –2 –1 1 2 2 0 –2 –2 –1 0 1 1 0 –1 –1 0	#6 0 –1 –1 –1 –1 –1 0 –1 –2 –2 –2 –2 –2 –1 –1 –2 –3 –3 –3 –2 –1 0 0 0 0 0 0 0 1 2 3 3 3 2 1 1 2 2 2 2 2 1 0 1 1 1 1 1 0	#7 0 –1 –1 –1 –1 –1 0 –1 –2 –2 –2 –2 –2 –1 –1 –2 –3 –3 –3 –2 –1 0 0 0 1 0 0 0 1 2 3 3 3 2 1 1 2 2 2 2 2 1 0 1 1 1 1 1 0

Table 6-4. Laplacian 3x3

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 0 –1 0 –1 4 –1 0 –1 0	#1 0 –1 0 –1 5 –1 0 –1 0	#2 0 –1 0 –1 6 –1 0 –1 0	#3 –1 –1 –1 –1 8 –1 –1 –1 –1
#4 –1 –1 –1 –1 9 –1 –1 –1 –1	#5 –1 –1 –1 –1 10 –1 –1 –1 –1	#6 –1 –2 –1 –2 12 –2 –1 –2 –1	#7 –1 –2 –1 –2 13 –2 –1 –2 –1

Table 6-5. Laplacian 5x5

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 24 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1	#1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 25 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1		

Table 6-6. Laplacian 7x7

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 48 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1	#1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 49 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1 –1		

Table 6-7. Smoothing 3x3

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 0 1 0 1 0 1 0 1 0	#1 0 1 0 1 1 1 0 1 0	#2 0 2 0 2 1 2 0 2 0	#3 0 4 0 4 1 4 0 4 0
#4 1 1 1 1 0 1 1 1 1	#5 1 1 1 1 1 1 1 1 1	#6 2 2 2 2 1 2 2 2 2	#7 4 4 4 4 1 4 4 4 4

Table 6-8. Smoothing 5x5

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 $\begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{matrix}$	#1 $\begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{matrix}$		

Table 6-9. Smoothing 7x7

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 $\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$	#1 $\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$		

Table 6-10. Gaussian 3x3

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 $\begin{matrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{matrix}$	#1 $\begin{matrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{matrix}$	#2 $\begin{matrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{matrix}$	#3 $\begin{matrix} 1 & 1 & 1 \\ 1 & 4 & 1 \\ 1 & 1 & 1 \end{matrix}$
#4 $\begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix}$	#5 $\begin{matrix} 1 & 4 & 1 \\ 4 & 16 & 4 \\ 1 & 4 & 1 \end{matrix}$		

Table 6-11. Gaussian 5x5

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 $\begin{matrix}1&2&4&2&1\\2&4&8&4&2\\4&8&16&8&4\\2&4&8&4&2\\1&2&4&2&1\end{matrix}$			

Table 6-12. Gaussian 7x7

matrix # and content	matrix # and content	matrix # and content	matrix # and content
#0 $\begin{matrix}1&1&2&2&2&1&1\\1&2&2&4&2&2&1\\2&2&4&8&4&2&2\\2&4&8&16&8&4&2\\2&2&4&8&4&2&2\\1&2&2&4&2&2&1\\1&1&2&2&2&1&1\end{matrix}$			

IPI_Convolute

```
IPIError = IPI_Convolute (IPIImageRef source_image, IPIImageRef
mask_image, IPIImageRef dest_image, IPIConvoDescPtr
convolution_matrix_descriptor, IPIBorderMethod border_method);
```

Purpose

This function filters an image using a linear filter. The calculations are completed either with integers or floating point values depending on the image type. The source and the destination image must be of the same type.

The source image must have been created with a border capable of using the size of the convolution matrix. A 3x3 matrix must have a minimum border of 1, a 5x5 matrix must have a minimum border of 2, and so on. The border size of the destination image is not important.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Mask image type: `IPI_PIXEL_U8`

Input

`source_image` is the image on which the convolution is made.

`mask_image` is the image that contains the mask applied to the source image.

It indicates the region of the image where the convolution is applied. Only pixels in the original image that correspond to a non-NULL pixel in the mask are processed. A convolution on the complete image occurs if this input is equal to `IPI_NOMASK`.

`dest_image` is the resulting image.

`convolution_matrix_descriptor` is a pointer to a structure containing the following information (see the section *IPI_GetConvolutionMatrix* in this chapter):

- convolution matrix width
- convolution matrix height
- pointer to convolution matrix elements
- divider (0.0 means matrix sum)

`border_method` indicates the method used to fill the border of the image before processing it.

- `IPI_BO_MIRROR`—repeat the last line of pixels by symmetry
- `IPI_BO_COPY`—copy the last line of pixels
- `IPI_BO_CLEAR`—all the border pixels are set to 0

IPI_GrayEdge

```
IPIError = IPI_GrayEdge (IPIImageRef source_image, IPIImageRef
mask_image, IPIImageRef dest_image, IPIEdgeMethod method, float
threshold);
```

Purpose

This function extracts the contours (edge detection) in gray level values.

The source and the destination images must be of the same type.

The source image must be created with a border of at least 1. The border size of the destination image is not important.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Mask image type: `IPI_PIXEL_U8`

Input

`source_image` is the image on which the edge detection is made.

`mask_image` is the image that contains the mask applied to the source image. It indicates the region of the image where the convolution is applied. Only pixels in the original image that correspond to a non-NULL pixel in the mask are used. An edge detection on the complete image occurs if this input is equal to `IPI_NOMASK`.

`dest_image` is the resulting image.

`method` indicates the type of edge detection filter to use.

- `IPI_EDG_DIFFER`—differentiation. Processing with a 2x2 matrix.
- `IPI_EDG_GRADIENT`—processing with a 2x2 matrix
- `IPI_EDG_PREWITT`—processing with a 3x3 matrix
- `IPI_EDG_ROBERTS`—processing with a 2x2 matrix
- `IPI_EDG_SIGNMA`—processing with a 3x3 matrix
- `IPI_EDG_SOBEL`—processing with a 3x3 matrix

`threshold` is the minimum pixel value to appear in the resulting image. It is rare to use a value greater than 0 for this type of processing because the results of the process are usually very dark and possess a low dynamic range.

IPI_LowPass

```
IPIError = IPI_LowPass (IPIImageRef source_image, IPIImageRef
mask_image, IPIImageRef dest_image, int x_filter_size, int
y_filter_size, float tolerance);
```

Purpose

This function computes the inter-pixel variation between the pixel being processed and those surrounding it. If the pixel being processed has a variation greater than a specific percentage, it is set to the average pixel value as calculated from the neighboring pixels. The source and the destination image must be of the same type.

The source image has to be created with a border capable of using the size of the convolution matrix. A 3x3 matrix must have a minimum border of 1, a 5x5 matrix must have a minimum border of 2, and so on. The border size of the destination image is not important.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Mask image type: `IPI_PIXEL_U8`

Input

`source_image` is the image on which the low pass processing is made.

`mask_image` is the image that contains the mask applied to the source image. It indicates the region of the image where the convolution is applied. Only pixels in the original image that correspond to a non-NULL pixel in the mask are used. A low pass on the complete image occurs if this input is equal to `IPI_NOMASK`.

`dest_image` is the resulting image.

`x_filter_size` is the size of the horizontal matrix axis.

`y_filter_size` is the size of the vertical matrix axis.

`tolerance` is the percentage of the maximum variation authorized.

IPI_NthOrder

```
IPIError = IPI_NthOrder (IPIImageRef source_image, IPIImageRef
mask_image, IPIImageRef dest_image, int x_filter_size, int
y_filter_size, int order_number);
```

Purpose

This function filters an image using a non-linear filter. This algorithm orders (or classifies) the pixel values surrounding the pixel being processed. The pixel being processed is set to the *N*th pixel value which is `order number`. The source and the destination image must be of the same type. The source image must be created with a border capable of using the size of the convolution matrix. A 3x3 matrix must have a minimum border of 1, a 5x5 matrix must have a minimum border of 2, and so on. The border size of the destination image is not important.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Mask image type: `IPI_PIXEL_U8`

Input

`source_image` is the image on which the processing is made.

`mask_image` is the image that contains the mask applied to the source image.

It indicates the region of the image where the convolution is applied. Only pixels in the original image that correspond to a non-NULL pixel in the mask are used. Filtering on the complete image occurs if this input is equal to `IPI_NOMASK`.

`dest_image` is the resulting image.

`x_filter_size` is the size of the horizontal matrix axis.

`y_filter_size` is the size of the vertical matrix axis.

`order_number` is the order number chosen after classifying the values.

Lighter images result when using a higher order number (such as 7 in a 3x3 matrix).

Darker images result when using a lower order number (such as 1 in a 3x3 matrix).

A median (center pixel) operation has the advantage of standardizing the gray level values without significantly modifying the shape of the objects or the overall brightness in the image.

If the input order value is 0, the obtained image is representative of the local minimum of the source image. If the order value is equal to (`x_filter_size` * `y_filter_size`) – 1, the obtained image is representative of the local maximum of the source image.

Morphology

The morphological transformations are divided into two groups: binary morphology and gray level morphology. In *binary morphology*, the pixels are either ON (with a pixel value different than 0) or OFF (a pixel value equal to 0). However, with *gray level morphology* the aim is to compare a pixel with those surrounding it, and to keep the pixel value which is the smallest (erosion) or the largest (dilation). Functions responsible for binary morphological transformations only accept an 8-bit image type, while the functions for gray level morphological transformations (`IPI_GrayMorphology`) can accept 8-bit, 16-bit, or floating images.

An image is considered to be binary after it has undergone a threshold function (`IPI_Threshold`, `IPI_AutoBThreshold`, and so on). In IMAQ Vision, binary morphology is divided into two groups. The primary operations operate a single function, `IPI_Morphology`. These might be erosions, dilations, openings, closings, or contour extractions. The advanced functions complete multiple operations. These functions include the separation of particles, removing either small or large particles, filling holes in particles, removing particles that touch the border of the image, creating the skeleton of particles, and so on.

Morphological transformations depend on an object known as the structuring element. With this structuring element, you have control of the effect of the functions on the shape and the boundary of objects. In IMAQ Vision, the *structuring element descriptor* is a structure that controls which pixels are to be processed and which pixels are to be left alone. A structuring element must have a center pixel and therefore must have odd-sized axes. The contents of the structuring element are also considered to be binary: zero or not zero. The most common structuring element is a 3x3 matrix that contains only ones (1). This is usually the default model for binary and gray level morphological transformations. You are advised to have some knowledge of structuring elements before experimenting with user-chosen sizes and contents. Most advanced morphology functions do not even possess an input for structuring elements because they are only the standard 3x3 matrix.

The connected source image for a morphological transformation must have been created with a border capable of using the size of the structuring element. A 3x3 structuring element requires a minimum border of 1, a 5x5 structuring element requires a minimum border of 3, and so on.

The input `connectivity_4/8` is used for the advanced morphology functions: `IPI_LowHighPass`, `IPI_RejectBorder`, and `IPI_FillHole`. These functions use this parameter to dictate whether a neighboring pixel is considered to be part of the same particle.

IPI_Morphology

```
IPIError = IPI_Morphology (IPIImageRef source_image, IPIImageRef
dest_image, IPIMOOperator operator, IPIMorphoDescPtr
structuring_element_descriptor);
```

Purpose

This function completes primary morphological transformations. The source image for a morphological transformation must be created with a border capable of using the size of the structuring element. A 3x3 structuring element requires a minimum border of 1, a 5x5 structuring element requires a minimum border of 2, and so on. The border size of the destination image is not important.

Image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

`operator` indicates the type of morphological transformation procedure to use.

- `IPI_MO_AUTOM`—auto median
- `IPI_MO_CLOSE`—closing. Dilation followed by an erosion.
- `IPI_MO_DILATE`—dilation. The opposite of an erosion (see below).
- `IPI_MO_ERODE`—erosion. Eliminates pixels in a source image.
- `IPI_MO_GRADIENT`—int & ext edges. Extraction of internal and external contours of a particle.
- `IPI_MO_GRADIOUT`—ext edge. Extraction of external contours of a particle.
- `IPI_MO_GRADIN`—int edge. Extraction of internal contours of a particle.

- `IPI_MO_HITM`—hit miss. Erases all pixels that do not have the same pattern as found in the structuring element.
- `IPI_MO_OPEN`—opening. Erosion followed by a dilation.
- `IPI_MO_PCLOSE`—proper closing. A succession of seven closings and openings.
- `IPI_MO_POPEN`—proper opening. A succession of seven openings and closings.
- `IPI_MO_THICK`—thick. Turn ON all pixels matching the pattern in the structuring element.
- `IPI_MO_THIN`—thin. Turn OFF all pixels matching the pattern in the structuring element.

`structuring_element_descriptor` is a pointer to a structure that describes the structuring element to be applied to the image. A structuring element of 3x3 is used if this input is equal to `IPI_MO_STD3X3`.

`structuring_element_descriptor` must have odd sized dimensions to contain a central pixel. The function does not take into account the even boundary, furthest out on the matrix, if one of the dimensions for the structuring element is even. For example, if the input structuring element is 6x4 ($X = 6$ and $Y = 4$), the actual processing is completed at 5x3. Both the 6th line and the 4th row are ignored.

IPI_GrayMorphology

```
IPIError = IPI_GrayMorphology (IPIImageRef source_image,
IPIImageRef dest_image, IPIMOOperator operator, IPIMorphoDescPtr
structuring_element_descriptor);
```

Purpose

This function makes morphological transformations that can be applied directly to gray level images. Source and destination image types must be of the same type. The source image for a morphological transformation must have been created with a border capable of using the size of the structuring element. A 3x3 structuring element requires a minimum border of 1, a 5x5 structuring element requires a minimum border of 2, and so on. The border size of the destination image is not important.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

`operator` indicates the type of morphological transformation procedure to use.

- `IPI_MO_AUTOM`—auto median
- `IPI_MO_CLOSE`—closing. Dilation followed by an erosion.
- `IPI_MO_DILATE`—dilation. The opposite of an erosion (see below).
- `IPI_MO_ERODE`—erosion. Eliminates pixels in a source image.
- `IPI_MO_OPEN`—opening. Erosion followed by a dilation.
- `IPI_MO_PCLOSE`—proper closing. A succession of seven closings and openings.
- `IPI_MO_POPEN`—proper opening. A succession of seven openings and closings.

`structuring_element_descriptor` is a pointer to a structure that describes the structuring element to be applied to the image. A structuring element of 3x3 is used if this input is equal to `IPI_MO_STD3X3`.

`structuring_element_descriptor` must have odd sized dimensions to contain a central pixel. The function does not take into account the even boundary, furthest out on the matrix, if one of the dimensions for the structuring element is even. For example, if the input structuring element is 6x4 (X = 6 and Y = 4), the actual processing is completed at 5x3. Both the 6th line and the 4th row are ignored.

IPI_Circles

```
IPIError = IPI_Circles (IPIImageRef source_image, IPIImageRef
dest_image, float min_radius, float max_radius, IPICirclesReportPtr
*circles_report_array_ptr, int *nb_of_detected_circles);
```

Purpose

This function separates overlapping circular objects and classifies them depending on their radius, surface, and perimeter. Starting from a binary image, it finds the radius and center of the circular objects even when multiple circular objects are overlapping. In addition, this function draws the circles in the destination image. It constructs and uses a Danielsson distance map to determine the radius of each object.

Image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

`min_radius` is the smallest radius (in pixels) to be detected. Undetected circles do not appear in the destination image and have a negative radius value in the `circles_report array_ptr` output.

`max_radius` is the largest radius (in pixels) to be detected. Circles possessing a radius larger than this value do not appear in the destination image and have a negative radius value in the `circles_report_array_ptr`.

Output

`circles_report_array_ptr` contains the pointer to the report containing the measurements for all detected circles. Each element of the report has a structure with the following elements:

- `xCenter` is the horizontal position (in pixels) of the center of the circle
- `yCenter` is the vertical position (in pixels) of the center of the circle
- `radius` is the radius of the circle in pixels
- `area` is the surface area (in pixels) of the nucleus of the circle in the Danielsson distance map

☞ **Note:** ***Circles with a radius outside the limits of*** `min_radius` ***or*** `max_radius` ***receive a negative radius value.***

`nb_of_detected_circles` contains the number of detected circles in the image. Circles with a radius outside the limits of `min_radius` or `max_radius` also are included in this number.

IPI_Convex

```
IPIError = IPI_Convex (IPIImageRef source_image, IPIImageRef
dest_image);
```

Purpose

This function computes a convex envelope for labelled particles in an image.

Image type: `IPI_PIXEL_U8`, `I16`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

☞ **Note:** ***If the image contains more than one object, it is necessary to execute*** `IPI_Label()` ***prior to this function to label the objects in the image.***

IPI_Danielsson

```
IPIError = IPI_Danielsson (IPIImageRef source_image, IPIImageRef
dest_image);
```

Purpose

This function completes the distance map based on the Danielsson algorithm. The Danielsson distance map produces an image and data similar to the function `IPI_Distance`, but it is much more accurate.

Image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

IPI_Distance

```
IPIError = IPI_Distance (IPIImageRef source_image, IPIImageRef
dest_image, IPIMorphoDescPtr structuring_element_descriptor);
```

Purpose

This function encodes the pixel value of a particle as a function of the distance of the pixel from the particle border. The source image must have been created with a border size of at least 1 and must be an 8-bit image.

Image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

`structuring_element_descriptor` is a pointer to a structure that describes the structuring element to be applied to the image. A structuring element of 3x3 is used if this input is equal to `IPI_MO_STD3X3`.

It must have odd sized dimensions to contain a central pixel. The function does not take into account the even boundary furthest out on the matrix, if one of the dimensions for the structuring element is even. For example, if the input structuring element is 6x4 (*X* = 6 and *Y* = 4), the actual processing is done at 5x3. Both the 6th line and the 4th row are ignored.

IPI_Separation

```
IPIError = IPI_Separation (IPIImageRef source_image, IPIImageRef
dest_image, int number_of_erosion, IPIMorphoDescPtr
structuring_element_descriptor);
```

Purpose

This function separates touching particles. It operates particularly on small isthmuses found between particles. It completes `number_of_erosions`. It then reconstructs the final image based on the results of the erosion. If, during the erosion process, an existing isthmus is broken or removed, the particles are reconstructed without the isthmus. The reconstructed particles, however, have the same size as the initial particles except that they are separated.

During the erosion process, if no isthmus is broken, the particles are reconstructed as they were initially found (that is, no changes are made). The source image must be an 8-bit or binary image and it must have a border greater than or equal to 1.

Image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

`number_of_erosion` indicates the number of erosions applied to separate the particles.

`structuring_element_descriptor` is a pointer to a structure that describes the structuring element to be applied to the image. A structuring element of 3x3 is used if this input is equal to `IPI_MO_STD3X3`.

It must have odd sized dimensions to contain a central pixel. The function does not take into account the even boundary furthest out on the matrix, if one of the dimensions for the structuring element is even. For example, if the input structuring element is 6x4 (*X* = 6 and *Y* = 4), the actual processing completes at 5x3. Both the 6th line and the 4th row are ignored.

IPI_FillHole

```
IPIError = IPI_FillHole (IPIImageRef source_image, IPIImageRef
dest_image, int connectivity_8);
```

Purpose

This function fills the holes found in a particle. The holes are filled with a pixel value of 1. The source image must be an 8-bit or binary image. This operation creates a temporary memory space equal to the size of the source image.

Image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

`connectivity_8` indicates the connectivity used to determine if a hole is to be filled. The connectivity mode determines if an adjacent pixel belongs to the same particle or a different particle. Possible values are:

- TRUE—The function completes particle detection in connectivity mode 8.
- FALSE—The function completes particle detection in connectivity mode 4.

Note: ***The holes found in contact with the image border are never filled because it is not possible to know whether these holes are part of a particle.***

IPI_LowHighPass

```
IPIError = IPI_LowHighPass (IPIImageRef source_image, IPIImageRef
dest_image, int connectivity_8, int high_pass, int
number_of_erosion, IPIMorphoDescPtr
structuring_element_descriptor);
```

Purpose

This function filters the particles according to their size. It eliminates or keeps particles present after a specific number of 3x3 erosions. This function creates a temporary memory space twice the size of the source image.

Image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

`connectivity_8` indicates the connectivity used for particle detection.

The connectivity mode directly determines if an adjacent pixel belongs to the same particle or a different particle. Possible values are:

- TRUE—The function completes particle detection in connectivity mode 8.
- FALSE—The function completes particle detection in connectivity mode 4.

`high_pass` controls if the particles present after the specified number of erosions are to be discarded (FALSE) or kept (TRUE).

`number_of_erosion` indicates the number of erosions applied to separate the particles.

`structuring_element_descriptor` is a pointer to a structure that describes the structuring element to be applied to the image. A structuring element of 3x3 is used if this input is equal to `IPI_MO_STD3X3`.

It must have odd sized dimensions to contain a central pixel. The function does not take into account the even boundary furthest out on the matrix, if one of the dimensions for the structuring element is even. For example, if the input structuring element is 6x4 (X = 6 and Y = 4), the actual processing completes at 5x3. Both the 6th line and the 4th row are ignored.

IPI_RejectBorder

```
IPIError = IPI_RejectBorder (IPIImageRef source_image, IPIImageRef
dest_image, int connectivity_8);
```

Purpose

This function eliminates particles touching the border of an image. This operation requires the creation of a temporary memory space equal to the size of the source image.

Image type: `IPI_PIXEL_U8`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

`connectivity_8` indicates the connectivity used to determine if a particle touching the border is to be eliminated. The connectivity mode determines if an adjacent pixel belongs to the same particle or a different particle. Possible values are:

- TRUE—The function completes particle detection in connectivity mode 8.
- FALSE—The function completes particle detection in connectivity mode 4.

IPI_Segmentation

```
IPIError = IPI_Segmentation (IPIImageRef source_image, IPIImageRef
dest_image);
```

Purpose

Starting from a labelled image, this function calculates the zone of influence of each particle. Each labelled particle grows until the particles reach their neighbors at which time this growth is stopped. The source image must have a border greater than or equal to 1.

Image type: `IPI_PIXEL_U8`, `I16`

Input

`source_image` is the image to transform.

`dest_image` is the resulting image.

IPI_Skeleton

```
IPIError = IPI_Skeleton (IPIImageRef source_image, IPIImageRef
dest_image, IPISkeletonMethod method);
```

Purpose

Starting from a binary image, this function calculates the skeleton of particles inside an image or in other words the lines separating the zones of influence (skeleton of an inverse image). The source image must have a border greater or equal to 1.

Image type: `IPI_PIXEL_U8`

Input

source_image is the image to transform.

dest_image is the resulting image.

method indicates the type of skeleton to use.

- IPI_MO_SKL—uses the Skeleton L structuring element
- IPI_MO_SKM—uses the Skeleton M structuring element
- IPI_MO_SKIZ—uses an inverse skeleton (Skeleton L on an inverse image)

Analysis

The following functions analyze the contents of an image. Some functions allow you to perform basic and complex particle detection. With others, you can extract measurements and morphological coefficients for each object in an image.

IPI_Histogram

```
IPIError = IPI_Histogram (IPIImageRef image, IPIImageRef
mask_image, int number_of_classes, float minimum_value, float
maximum_value, int histogram_array[], IPIHistoReport
*histogram_report);
```

Purpose

This function computes the histogram of an image. A histogram indicates the quantitative distribution of the pixels of an image per gray level value.

Image type: IPI_PIXEL_U8, I16, SGL

Mask image type: IPI_PIXEL_U8

Input

image is the image used to compute the histogram.

mask_image indicates the region in the image used for computing the histogram. In other words, only pixels in the original image that correspond to a non-NULL pixel in the mask are used to compute the histogram. A histogram on the complete image is computed if this input is equal to IPI_NOMASK.

number_of_classes indicates the number of classes in the histograms (that is, the number of elements of the histogram array). The number of calculated classes differ from the indicated value if the minimum and maximum boundaries are overshot. It is advised to specify an even number of classes (that is, 2,4,8) for 8- or 16-bit images. The value 256 causes a uniform class distribution and one class for each pixel value in an 8-bit image. Only pixels whose values fall in the range of minimum_value and maximum_value are taken into account in the histogram calculation.

minimum_value is the lower limit of the range. Passing the value 0 in minimum_value and maximum_value ensures that the lowest value is taken from the source image. The minimum default value depends on the image type.

- IPI_PIXEL_U8—the minimum default value is 0
- IPI_PIXEL_I16—the minimum default value is the minimum pixel value found in the image
- IPI_PIXEL_SGL—the minimum default value is the minimum pixel value found in the image

maximum_value is the higher limit of the range. Passing the value 0 in minimum_value and maximum_value ensures that the highest value is taken from the source image. The maximum default value depends on the image type.

- IPI_PIXEL_U8—the maximum default value used is 255
- IPI_PIXEL_I16—the maximum default value used is the maximum pixel value found in the image
- IPI_PIXEL_SGL—the maximum default value used is the maximum pixel value found in the image

Output

histogram_array is filled with the histogram values. The elements found in this array are the number of pixels per class. The *n* class contains all pixel values belonging to the interval.

[Starting Value + (*n* – 1) * Interval Width, Starting Value + *n* * Interval Width – 1].

It can be a NULL pointer if you do not need this histogram.

histogram_report is a structure filled the statistical values related to the histogram computation. It can be a NULL pointer if you do not need this report.

This structure contains the following elements:

- minValue—lowest pixel value found in the calculated area
- maxValue—highest pixel value found in the calculated area

- `startValue`—lowest pixel value in the first class of the histogram. It can be equal to the `minimum value`, or the smallest value found from the input image.
- `interval`—width of each class
- `mean`—mean value of the pixels
- `stdDeviation`—standard deviation of the considered class in the histogram. The higher this value is, the better the distribution of the values in the histogram and the corresponding image.
- `area`—number of pixels used in the histogram calculation. This is influenced by the values indicated by `minimum_value` and `maximum_value`, and the contents of `mask_image`.

IPI_Quantify

```
IPIError = IPI_Quantify (IPIImageRef image, IPIImageRef
labelled_image, IPIQuantifyElem *global_report, IPIQuantifyElemPtr
*region_report_array_ptr, int *nb_of_region_reports);
```

Purpose

This function quantifies the contents of an image or the regions within an image. The function uses a labelled mask image to define the regions. Each region (or particle) of the mask image possesses a single unique value.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Labelled image type: `IPI_PIXEL_U8`

Input

`image` is the image to quantify.

`labelled_image` indicates the mask image that contains the labelled regions quantified in the image. Only pixels in the original image that correspond to the equivalent pixel in the mask are used for the quantification. Each pixel in this labelled image indicates by its value, to which region the corresponding pixel in `image` belongs. From `image`, 255 different regions can be quantified directly. This function performs a quantification completed on the complete image if `labelled_image` is equal to `IPI_NOIMAGE`.

Output

`global_report` is a structure containing the following elements:

- `mean`—mean gray value in the particle
- `stdDeviation`—standard deviation of the pixel values. Pixel values are better distributed as the standard deviation increases.
- `minValue`—lowest gray value in the particle
- `maxValue`—highest gray value in the particle
- `surface`—analyzed surface in user-defined units
- `area`—analyzed surface in pixels
- `percent`—percentage of the analyzed surface in relation to the complete image

`region_report_array_ptr` is a pointer to a structure that contains the quantification data relative to all the regions within an image. This structure is allocated or reallocated within this function. You must deallocate it using `free()`.

`nb_of_region_reports` is the number of elements in the `region_report_array_ptr`.

IPI_Centroid

```
IPIError = IPI_Centroid (IPIImageRef image, IPIImageRef mask_image,
float *x_centroid, float *y_centroid);
```

Purpose

This function computes the centroid (the center of the pixel energy) in an image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Mask image type: `IPI_PIXEL_U8`

Input

`image` is the image used to compute the centroid coordinates.

`mask_image` indicates the region in the image to use for computing the centroid. Only pixels in the original image that correspond to a non-NULL pixel in the mask are used to compute the centroid. A computation on the complete image occurs if this parameter is equal to `IPI_NOMASK`.

Output

`x_centroid` returns the value of the centroid X coordinate.

`y_centroid` returns the value of the centroid Y coordinate.

IPI_LineProfile

```
IPIError = IPI_LineProfile (IPIImageRef image, Point start, Point
end, int profile_format, void *profile_array, int *nb_of_elements,
IPIProfReport *profile_report);
```

Purpose

This function computes the profile of a line of pixels. This function is similar to the histogram, working only on a line vector in the image.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`

Input

`image` is the image used to compute the line profile.

`start` defines the (*x*,*y*) coordinates of the start point of the line profile.

`end` defines the (*x*,*y*) coordinates of the end point of the line profile.

`profile_format` indicates the data type you want to get in the `profile_array` using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

Output

`profile_array` points on the first pixel of the line profile you allocated in the image.

`nb_of_elements` returns the number of pixels filled in the `profile_array`.

`profile_report` points to a structure which contains relevant information on the pixels found in the indicated line. It returns the following elements:

- `start`—corrected start point
- `end`—corrected end point
- `minValue`—lowest value of the line
- `maxValue`—highest value of the line
- `meanValue`—mean gray value of the line
- `stdDeviation`—standard deviation of the line profile
- `count`—number of pixels on the line

IPI_BasicParticle

```
IPIError = IPI_BasicParticle(IPIImageRef image, int connectivity_8,
IPIBasicPReportPtr *particles_report_array_ptr, int
*nb_of_detected_particles);
```

Purpose

This function detects and returns the area and position of particles in a binary image.

Image type: `IPI_PIXEL_U8`

Input

`image` is the image scanned to detect the particles. The image must be binary (a threshold `IPI_PIXEL_U8` image). A particle consists of pixels that are not set to 0. This source image must have been created with a border size of at least 2.

`connectivity_8` indicates the connectivity used for particle detection.

The connectivity mode determines if an adjacent pixel belongs to the same particle or to a different particle. Possible values are:

- TRUE—The function completes particle detection in connectivity mode 8.
- FALSE—The function completes particle detection in connectivity mode 4.

Output

`particles_report_array_ptr` returns a pointer to a set of measurements on the detected particles. Each report contains the following elements:

- `area`—surface of the particle in number of pixels
- `surface`—surface of the particle in user-defined units
- `particleRect`—bounding rectangle of the particle. It is a standard rectangle that contains the coordinates of a bounding rectangle for the particle.

☞ **Note:** ***This structure is allocated or reallocated within this function. You must deallocate it using*** `free()`**.**

`nb_of_detected_particles` returns the number of particles detected in the image. This value indicates the number of reports allocated in the buffer returned in `particles_report_array_ptr`.

IPI_Particle

```
IPIError = IPI_Particle (IPIImageRef image, int connectivity_8,
IPIFullPReportPtr *particles_report_array_ptr, int
*nb_of_detected_particles);
```

Purpose

This function detects and returns all parameters of particles in a binary image.

Image type: `IPI_PIXEL_U8`

Input

`image` is the image scanned to detect the particles. The image must be binary (a threshold `IPI_PIXEL_U8` image). A particle consists of pixels that are not set to 0. This image must have a border size of at least 2.

`connectivity_8` indicates the connectivity used for particle detection.

The connectivity mode determines if an adjacent pixel belongs to the same particle or to a different particle. Possible values are:

- TRUE—The function completes particle detection in connectivity mode 8.
- FALSE—The function completes particle detection in connectivity mode 4.

Output

`particles_report_array_ptr` returns a pointer to a set of measurements on the detected particles. Each report contains the following elements:

- `area`—surface of the particle in number of pixels
- `surface`—surface of the particle in user-defined units
- `perimeter`—perimeter size of the particle in user-defined units
- `holeNumber`—number of holes in the particle
- `holeArea`—total surface area of all the holes in the particle in user-defined units
- `holePerimeter`—total perimeter size calculated from every hole in the particle in user units
- `includeRect`—bounding rectangle of the particle. It is a standard rectangle that contains the coordinates of a bounding rectangle for the particle.
- `sigmaX`—sum of X-axis for each pixel of the particle
- `sigmaY`—sum of Y-axis for each pixel of the particle
- `sigmaXX`—sum of X-axis squared for each pixel of the particle
- `sigmaYY`—sum of Y-axis squared for each pixel of the particle
- `sigmaXY`—sum of the X-axis and Y-axis for each pixel of the particle
- `segmentMax`—longest segment of the particle
- `segmentMaxPt`—left-most pixel in the `segmentMax` of the particle
- `projectionX`—half the sum of the horizontal segments in a particle which do not overlap another adjacent horizontal segment
- `projectionY`—half the sum of the vertical segments in a particle which do not overlap another adjacent vertical segment

Note: ***This structure is allocated or reallocated within this function. You must deallocate it using*** **`free()`.**

`nb_of_detected_particles` returns the number of particles detected in the image. This value indicates the number of reports allocated in the buffer returned by `particles_report_array_ptr`.

IPI_ParticleCoeffs

```
IPIError = IPI_ParticleCoeffs (IPIImageRef image, int
parameters_array[], int nb_of_parameters, IPIFullPReport
particles_report_array[], int nb_of_particle_reports, float
particles_coefficients_array[]);
```

Purpose

Using reports coming from the `IPI_Particle()`, this function computes and returns a set of measurements.

Image type: `IPI_PIXEL_U8`

Input

`Image` is the image previously used in `IPI_Particle()`. This function needs this image to get the calibration values.

`parameter_array` is an array containing the parameter list you want to extract. This parameter list has to contain elements taken from the following predefined values:

- `IPI_PP_Area`— surface of a particle in number of pixels
- `IPI_PP_AreaCalibrated`—surface of a particle in user-defined units
- `IPI_PP_HoleNumber`—number of hole in the particle
- `IPI_PP_HoleArea`—total surface of every holes in a particle in user-defined units
- `IPI_PP_AreaTotal`—full surface occupied by the particles and its holes in user-defined units
- `IPI_PP_AreaScanned`—surface of the image in user-defined units
- `IPI_PP_RatioAreaTotal`—ratio of the surface of the particle to the total surface of particles
- `IPI_PP_RatioAreaScanned`—ratio of the surface of the particle to the surface of the image
- `IPI_PP_CenterMassX`—abscissa of the center of mass of the particle
- `IPI_PP_CenterMassY`—ordinate of the center of mass of the particle
- `IPI_PP_RectLeft`—abscissa of the bounding rectangle
- `IPI_PP_RectTop`—upper ordinate of the bounding rectangle
- `IPI_PP_RectRight`—right abscissa of the bounding rectangle
- `IPI_PP_RectBottom`—lower ordinate of the bounding rectangle
- `IPI_PP_RectWidth`—width of the bounding rectangle in user-defined units

- `IPI_PP_RectHeight`—height of the bounding rectangle in user-defined units
- `IPI_PP_MaxSegment`—longest horizontal segment of the particle
- `IPI_PP_MaxSegmentX`—left abscissa of the longest segment of the particle
- `IPI_PP_MaxSegmentY`—ordinate of the longest segment of the particle
- `IPI_PP_Perimeter`—perimeter size of a particle in user-defined units
- `IPI_PP_HolePerimeter`—total perimeter size calculated from every hole in a particle in user-defined units
- `IPI_PP_SigmaX`—sum of X-axis for each pixel of the particle
- `IPI_PP_SigmaY`—sum of Y-axis for each pixel of the particle
- `IPI_PP_SigmaXX`—sum of X-axis squared for each pixel of the particle
- `IPI_PP_SigmaYY`—sum of Y-axis squared for each pixel of the particle
- `IPI_PP_SigmaXY`—sum of the X-axis and Y-axis for each pixel of the particle
- `IPI_PP_ProjectionX`—half the sum of the horizontal segments in a particle that do not overlap another adjacent horizontal segment
- `IPI_PP_ProjectionY`—half the sum of the vertical segments in a particle that do not overlap another adjacent vertical segment
- `IPI_PP_IXX`—coefficients of the X-axis squared inertia matrix
- `IPI_PP_IYY`—coefficients of the Y-axis squared inertia matrix
- `IPI_PP_IXY`—coefficients of the XY-axis inertia matrix
- `IPI_PP_MeanChordX`—mean length of horizontal segments
- `IPI_PP_MeanChordY`—mean length of vertical segments
- `IPI_PP_MaxIntercept`—longest segment of the particle
- `IPI_PP_MeanIntercept`—mean length segment of the particle
- `IPI_PP_Orientation`—orientation of the longest segment of the particle
- `IPI_PP_EquEllipseMinor`—minor axis of the equivalent ellipse
- `IPI_PP_EllipseMajor`—major axis of the equivalent ellipse equivalent to the particle surface
- `IPI_PP_EllipseMinor`—minor axis of the equivalent ellipse of equal surface to the particle
- `IPI_PP_RatioEquEllipse`—ratio of equivalent ellipse axes
- `IPI_PP_RectBigSide`—longest segment of the rectangle of equal surface to the particle in user-defined units
- `IPI_PP_RectSmallSide`—smallest segment of the rectangle of equal surface to the particle in user-defined units

- `IPI_PP_RatioRect`—ratio of the axes of the particle rectangle
- `IPI_PP_Elongation`—elongation factor
- `IPI_PP_Compactness`—compactness factor
- `IPI_PP_Heywood`—Heywood factor
- `IPI_PP_TypeFactor`—complex factor between surface and inertia matrix
- `IPI_PP_HydraulicRadius`—hydraulic radius in user-defined units
- `IPI_PP_WaddelDisk`—Waddle disk factor
- `IPI_PP_Diagonal`—equivalent rectangle diagonal in user-defined units

`nb_of_parameters` indicates the number of elements of the `parameter_array`.

`particles_report_array` is an array containing the particle reports.

`nb_of_particle_reports` is the number of particles of the particles report array.

Output

`particles_coefficient_array` is an array filled with the computed particle coefficients. You must allocate an array big enough to receive `nb_of_parameters * nb_of_particle_reports` single floating values. The computed coefficients are returned particle by particle. That means it can be considered as a 2D array in which a row contains `nb_of_parameters` coefficients extracted from a particle and each column contains the same coefficients extracted from each particle measurement.

IPI_ParticleDiscrim

```
IPIError = IPI_ParticleDiscrim (IPIImageRef image, IPIPartDiscrim
discrimination_array[], int nb_of_discriminations, IPIFullPReport
particle_report_array[], int nb_of_particle_report, int
*nb_of_remaining_particles);
```

Purpose

This function filters particle reports returned from `IPI_Particle()` according to a set of user-defined coefficient ranges.

Note: ***The image is not modified. This function works only on the particle report array, removing particle reports that do not match the selection criteria.***

Image type: `IPI_PIXEL_U8`

Input

`image` is the image previously used in `IPI_Particle()`. This function needs this image to get the calibration values.

`discrimination_array` describes the criteria for the particle you want to keep. Each element of this array has to contain a structure giving the following information:

- `parameter`—coefficient predefined value (see the section *IPI_ParticleCoeffs* in this chapter)
- `minValue`—coefficient lowest value
- `maxValue`—coefficient highest value

Only the particles with a parameter matching the range given by `minValue` and `maxValue` are not removed from the `particles_report_array`.

`nb_of_discriminations` is the number of discrimination structures contained in `discrimination_array`.

`particle_report_array` is an array containing the particle reports coming from `IPI_Particle()`. The array is processed and returned with only the matching particle reports.

`nb_of_particle_report` indicates the number of particles in the `particle_report_array` before the discrimination process.

Output

`nb_of_remaining_particles` returns the number of particles contained in the `particles_report_array` after the discrimination.

Geometry

The following section includes descriptions for 3D view, rotate, shift, and symmetry functions.

IPI_3DView

```
IPIError = IPI_3DView (IPIImageRef source_image, IPIImageRef
dest_image, int subsample, int maximum_height, IPI3DDirection
direction, IPI3DOptionsPtr options);
```

Purpose

This function displays an image using an isometric view. Each pixel from the image source appears as a column of pixels in the 3D view. The pixel value corresponds to the altitude.

Image type: `IPI_PIXEL_U8`, `COMPLEX`

Input

`source_image` is the input image.

`dest_image` is the resulting image. It must be an 8-bit image.

`subsample` is a factor applied to the source image to calculate the final dimensions of the 3D view image. This factor is a divider that is applied to the source image when determining the final height and width of the 3D view image. A factor of 1 uses all of the source image pixels when determining the 3D view image. A factor of 2 results in using every other line and every other column of the source image pixels to determine the 3D view image.

`maximum_height` defines the maximum height of a pixel from the image source that is drawn in 3D. This value is mapped from a maximum of 255 (from the source image) in relation to the baseline in the 3D view. A value of 255 therefore assigns a one to one correspondence between the intensity value in the source image and the display in 3D view. The default value of 64 results in a 4x4 reduction between the original intensity value of the pixel in the source image and the final displayed 3D image.

`direction` defines the 3D orientation. The following four viewing angles are possible:

- `IPI_3D_NW`—northwest view
- `IPI_3D_SW`—southwest view

- `IPI_3D_SE`—southeast view
- `IPI_3D_NE`—northeast view

`Options` is a cluster containing the following elements:

- `alpha` defines the angle between the horizontal and the baseline. The value can be set between 15° and 45°. The default value is 30°.
- `beta` defines the angle between the horizontal and the second baseline. The value can be set between 15° and 45°. The default value is 30°.
- `border` defines the border size in the 3D view. The default value is 20.
- `background` defines the background color for the 3D view. The default value is 85.
- `plane` indicates the view to show if the image is complex. For complex images the default plane shown is the magnitude plane. The four possible planes can be visualized from a complex image, as follows:
  - 0—real
  - 1—imaginary
  - 2—magnitude
  - 3—phase

IPI_Rotate

```
IPIError = IPI_Rotate (IPIImageRef source_image, IPIImageRef
dest_image, float angle, float fill_value);
```

Purpose

This function rotates an image counterclockwise.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`

Input

`source_image` is the input image.

`dest_image` is the resulting image.

`angle` indicates the rotation angle (in degrees).

`fill_value` defines the new pixel value due to the rotation.

IPI_Shift

```
IPIError = IPI_Shift (IPIImageRef source_image, IPIImageRef
dest_image, int x_shift, int y_shift, float fill_value);
```

Purpose

This function translates an image based on a horizontal and vertical offset.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`

Input

`source_image` is the input image.

`dest_image` is the resulting image.

`x_shift` is the horizontal offset to add to the image.

`y_shift` is the vertical offset to add to an image.

`fill_value` defines the new pixel value due to the translation.

IPI_Symmetry

```
IPIError = IPI_Symmetry (IPIImageRef source_image, IPIImageRef
dest_image, IPISymOperator symmetry_type);
```

Purpose

This function transforms an image around an axis or point of symmetry.

Image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `RGB32`

Input

`source_image` is the source image.

`dest_image` is the resulting image.

`symmetry_type` indicates the symmetry to use.

- `IPI_SY_HOR`—(default) horizontal. Based on the horizontal axis of the image.
- `IPI_SY_VER`—vertical. Based on the vertical axis of the image.
- `IPI_SY_CEN`—central. Based on the center of the image.

- `IPI_SY_DG1`—1st diagonal. Based on the first diagonal of the image. The image must be square.
- `IPI_SY_DG2`—2nd diagonal. Based on the second diagonal of the image. The image must be square.

Complex

Frequency processing is another way to extract information from an image. Instead of using the location and direction of light intensity variations, you can manipulate the frequency of occurrence in the spatial domain. This new component is called the *spatial frequency* and is the frequency with which the light intensity in an image varies as a function of its spatial coordinates.

Spatial frequencies of an image are computed with the Fast Fourier Transform (FFT). The FFT results in a complex image where high frequencies are grouped at the center, while low frequencies are located at the edges. The FFT is calculated in two steps: 1) a one-dimensional transform of the rows followed by 2) a one-dimensional transform of the columns of the results of step one. The complex numbers which compose the FFT plane are encoded in 64-bit floating point values: 32 bits for the real part and 32 bits for the imaginary part. IMAQ Vision can read and write complex images through `IPI_ReadFile` and `IPI_WriteFile` using the AIPD format only.

In an image, details and sharp edges are associated with high spatial frequencies because they introduce significant gray level variations over short distances. Gradually varying patterns are associated with low spatial frequencies. Filtering spatial frequencies is a means to remove, attenuate, or highlight the spatial components to which they relate.

A `low-pass_frequency_filter` can be used to attenuate frequencies present in the FFT plane. It suppresses information related to fast variations of light intensities in the spatial image. An inverse FFT after a low-pass frequency filter produces an image with reduced noise, details, texture and sharp edges (for example, `IPI_ComplexAttenuate` or `IPI_ComplexTruncate`).

A `high-pass_frequency_filter` can be used to attenuate or remove (truncate) low frequencies present in the FFT plane. It suppresses information related to slow variations of light intensities in the spatial image. In this case, an inverse FFT after a high-pass frequency filter produces an image with sharpened overall patterns and emphasized details (for example, `IPI_ComplexAttenuate` or `IPI_ComplexTruncate`).

A `mask_frequency_filter` removes frequencies contained in a mask indicated by you (`IPI_Mask`).

`IPI_WindDraw` handles the display of complex images. This function displays an image by flipping the high and low frequencies and then dividing their values by a size factor.

This size factor *m* is calculated using the following formula.

$m = f(w + h) = f(32.2^n) = 2.4^n$,

where *w* is the width of the image and *h* is its height.

IPI_FFT

```
IPIError = IPI_FFT (IPIImageRef source_image, IPIImageRef
complex_dest_image);
```

Purpose

This function computes the FFT of an image.

Source image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `COMPLEX`

Dest image type: `IPI_PIXEL_COMPLEX`

Input

`source_image` is the source image. The image must have a resolution of 2^n x 2^m.

`complex_dest_image` is the complex image that contains the resulting FFT image. The calculated FFT is not normalized. (You can use `IPI_ComplexDivide` to normalize the complex image.) The complex image is resized to the source image.

IPI_InverseFFT

```
IPIError = IPI_InverseFFT (IPIImageRef complex_source_image,
IPIImageRef dest_image);
```

Purpose

This function computes the inverse FFT of a complex image.

☞ **Note:** ***This function uses a buffer equal to the size of the complex image. An 8-bit image with a resolution of 256x256 pixels uses 64 KB of memory. The FFT associated with this image requires eight times the memory or 512 KB. The calculation of the inverse FFT also requires a temporary buffer of 512 KB. Therefore the total memory necessary for this operation is 1080 KB.***

Source image type: `IPI_PIXEL_COMPLEX`

Dest image type: `IPI_PIXEL_U8`, `I16`, `SGL`, `COMPLEX`

Input

`complex_source_image` is the source image.

`dest_image` contains the resulting FFT image.

IPI_ComplexConjugate

```
IPIError = IPI_ComplexConjugate (IPIImageRef complex_source_image,
IPIImageRef complex_dest_image);
```

Purpose

This function computes the conjugate of a complex image. It converts the complex pixel data $z = a + ib$ of an FFT image into $z = a - ib$.

Image type: `IPI_PIXEL_COMPLEX`

Input

`source_image` is the source image.

`complex_dest_image` contains the resulting image.

IPI_ComplexFlipFrequency

```
IPIError = IPI_ComplexFlipFrequency (IPIImageRef
complex_source_image, IPIImageRef complex_dest_image);
```

Purpose

This function transposes the complex components of an FFT image.

☞ **Note:** ***The high and low frequency components of an FFT image are flipped to produce a central symmetric representation of the spatial frequencies.***

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_source_image` is the source image.

`complex_dest_image` is the image that contains the flipped frequencies image.

IPI_ComplexAttenuate

```
IPIError = IPI_ComplexAttenuate (IPIImageRef complex_source_image,
IPIImageRef complex_dest_image, int high_pass);
```

Purpose

This function attenuates the frequencies of a complex image.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_source_image` is the source image.

`complex_dest_image` contains the resulting image.

`high_pass` determines which frequencies are attenuated. Choose TRUE to attenuate the high frequencies or FALSE to attenuate the low frequencies.

IPI_ComplexTruncate

```
IPIError = IPI_ComplexTruncate (IPIImageRef complex_source_image,
IPIImageRef complex_dest_image, int high_pass, float
truncation_frequency);
```

Purpose

This function truncates the frequencies of a complex image.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_source_image` is the source image.

`complex_dest_image` contains the resulting image.

`high_pass` determines which frequencies are truncated. Choose TRUE to remove the high frequencies or FALSE to remove the low frequencies.

`truncation_frequency` is the percentage of the frequencies that are retained within a Fourier transformed image. The default value is 10%. This percentage works in conjunction with the length of the diagonal of the FFT image and the Boolean `high_pass`. For example, a FALSE value or `low_pass` and `10%` results in retaining 10% of the frequencies starting from the center (low frequencies.) A TRUE value or `High_pass` and `10%` results in retaining 10% of the frequencies starting from the outer periphery.

IPI_ComplexAdd

```
IPIError = IPI_ComplexAdd (IPIImageRef complex_source_A_image,
IPIImageRef complex_source_B_image, IPIImageRef
complex_dest_image, IPIPixComplex constant);
```

Purpose

This function adds two images where the first is a complex image. An operation between an image and a constant occurs when the input `complex_source_B_image` is equal to `IPI_USECONSTANT`.

The two possibilities are distinguished in the following manner:

dest(x,y) = source A(x,y) + source B(x,y)

or

dest(x,y) = source A(x,y) + constant

The result of this operation follows:

- The resulting real part is the sum of the real parts of the input images, and
- the resulting imaginary part is the sum of the imaginary parts of the input images.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_source_A_image` is the first source image and must be a complex image.

`complex_source_B_image` is the second source image.

`complex_dest_image` is the image that contains the result of the operation.

`complex_constant` is the constant added to the input `complex_source_A_image` for an operation between an image and a constant.

IPI_ComplexSubtract

```
IPIError = IPI_ComplexSubtract (IPIImageRef complex_source_A_image,
IPIImageRef complex_source_B_image, IPIImageRef
complex_dest_image, IPIPixComplex constant);
```

Purpose

This function subtracts two images where the first is a complex image. An operation between an image and a constant occurs when the input `complex_source_B_image` is equal to `IPI_USECONSTANT`.

The two possibilities are distinguished in the following manner:

dest(x,y) = source A(x,y) – source B(x,y)

or

dest(x,y) = source A(x,y) – constant

The result of this operation follows:

- The resulting real part is the subtraction between the real parts of the input images, and
- the resulting imaginary part is the subtraction between the imaginary parts of the input images.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_source_A_image` is the first source image and must be a complex image.

`complex_source_B_image` is the second source image.

`complex_dest_image` contains the resulting image.

`complex_constant` is the constant subtracted from the input `complex_source_A_image` for the image/constant operation.

IPI_ComplexMultiply

```
IPIError = IPI_ComplexMultiply (IPIImageRef complex_source_A_image,
IPIImageRef complex_source_B_image, IPIImageRef
complex_dest_image, IPIPixComplex constant);
```

Purpose

This function multiplies two images where the first is a complex image. An operation between an image and a constant occurs when the input `complex_source_B_image` is equal to `IPI_USECONSTANT`.

The two possibilities are distinguished in the following manner:

dest(x,y) = source A(x,y) * source B(x,y)

or

dest(x,y) = source A(x,y) * constant

The result of this operation follows:

- The resulting real part is the multiplication of the real parts of the input images, and
- the resulting imaginary part is the multiplication of the imaginary parts of the input images.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_source_A_image` is the first source image.

`complex_source_B_image` is the second source image.

`complex_dest_image` contains the resulting image.

`complex_constant` is the constant multiplier of the `complex_source_A_image` for the image/constant operation.

IPI_ComplexDivide

```
IPIError = IPI_ComplexDivide (IPIImageRef complex_source_A_image,
IPIImageRef complex_source_B_image, IPIImageRef
complex_dest_image, IPIPixComplex constant);
```

Purpose

This function divides two images where the first is a complex image. An operation between an image and a constant occurs when the input `complex_source_B_image` is equal to `IPI_USECONSTANT`.

The two possibilities are distinguished in the following manner:

dest(x,y) = source A(x,y) / source B(x,y)

or

dest(x,y) = source A(x,y) / constant

The result of this operation follows:

- The resulting real part is the division between the real parts of the input images, and
- the resulting imaginary part is the division between the imaginary parts of the input images.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_source_A_image` is the first source image.

`complex_source_B_image` is the second source image.

`complex_complex_dest_image` contains the resulting image.

`complex_constant` is the constant divider of the `complex_source_A_image` for the image/constant operation.

IPI_ComplexImageToArray

```
IPIError IPI_ComplexImageToArray (IPIImageRef complex_image, Rect
rectangle, IPIPixComplex complex_array[], int *array_x_size, int
*array_y_size);
```

Purpose

This function reads the pixels from a complex image into a 2D single complex array.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_image` is the image used for this operation.

`rectangle` is a `Rect` structure containing the coordinates and the size of the rectangle.

Output

`complex_array` is the pointer of the complex pixel array allocated by you. It must be big enough to contain all elements.

`array_x_size` returns the horizontal number of copied elements in the array.

`array_y_size` returns the vertical number of copied elements in the array.

IPI_ArrayToComplexImage

```
IPIError IPI_ArrayToComplexImage (IPIImageRef complex_image,
IPIPixComplex complex_array[], int array_x_size, int array_y_size);
```

Purpose

This function creates a complex image, starting from an array of complex values. The resulting image is resized to `array_x_size` and `array_y_size`.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_image` is the image to modify.

`complex_array` defines the pointer to the pixel array containing the new pixel values which are copied into the image.

`array_x_size` is the horizontal number of elements in the array.

`array_y_size` is the vertical number of elements in the array.

IPI_ComplexPlaneToArray

```
IPIError IPI_ComplexPlaneToArray (IPIImageRef complex_image, int
plane, Rect rectangle, float float_array[], int *array_x_size, int
*array_y_size);
```

Purpose

This function extracts the pixels from the real, imaginary, magnitude, or phase plane of a complex image into a floating point 2D array.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_image` is the image used for this operation.

`plane` selects the plane to extract.

- 0—real
- 1—imaginary
- 2—magnitude
- 3—phase

`rectangle` is a `Rect` structure containing the coordinates and the size of the rectangle.

Output

`float_array` is the pointer of an array allocated by you. It must be big enough to contain all the copied elements.

`array_x_size` returns the horizontal number of copied elements in the array.

`array_y_size` returns the vertical number of copied elements in the array.

IPI_ArrayToComplexPlane

```
IPIError IPI_ArrayToComplexPlane (IPIImageRef complex_image, int
plane, float float_array[], int array_x_size, int array_y_size);
```

Purpose

Starting from a 2D array of floating point values, this function replaces the real or imaginary plane of a complex image.

Image type: `IPI_PIXEL_COMPLEX`

Input

`complex_image` is the image used for this operation.

`plane` selects the plane to extract.

- 0—real
- 1—imaginary

`float_array` defines the pointer of the pixel array containing the new pixel values which are copied into the image plane.

`array_x_size` is the horizontal number of elements in the array.

`array_y_size` is the vertical number of elements in the array.

IPI_ExtractComplexPlane

```
IPIError = IPI_ExtractComplexPlane. (IPIImageRef
complex_source_image, IPIImageRef dest_image, int plane);
```

Purpose

This function extracts the real or imaginary plane of a complex image.

Image type: `IPI_PIXEL_COMPLEX`, `SGL`

Input

`complex_source_image` is the source image used for this operation.

`dest_image` is the resulting spatial image that contains the extracted plane.

`plane` indicates the plane to be extracted.

- 0—real
- 1—imaginary
- 2—magnitude
- 3—phase

IPI_ReplaceComplexPlane

```
IPIError = IPI_ReplaceComplexPlane (IPIImageRef source_image,
IPIImageRef complex_dest_image, int plane);
```

Purpose

This function replaces the real or imaginary plane of a complex image.

Image type: `IPI_PIXEL_COMPLEX`

Input

`source_image` is the source image used for this operation.

`complex_dest_image` is the image that contains the result.

`plane` selects the plane to replace.

- 0—real
- 1—imaginary

Color

Typical color images are coded using three planes: red, green, and blue. In reality, pixels are encoded in 32 bits (four channels).

- bits 31 to 24—the alpha channel (not used)
- bits 23 to 16—the red channel
- bits 15 to 8—the green channel
- bits 7 to 0—the blue channel

However this representation is not valid on every 32-bit micro-processor type. From the memory point of view, the order depends on the computer.

- On Intel processor-based computers, the memory byte order is called little endian. The bytes inside a 32-bit word are organized in the following order: blue, green, red, alpha.
- On Apple or Sun computers, the memory byte order is big endian: alpha, red, green, blue.

IMAQ Vision for LabWindows/CVI manages this difference internally and has functions that make this aspect completely transparent to you.

A color image always is encoded in memory in the RGB form. However, there are a number of other coding models such as Hue, Saturation, and Lightness (HSL) and Hue, Saturation, and Value (HSV).

To compute the values for hue, saturation, lightness, or value, a measurement is made from the red, green, and blue components. Notice that these measurements take time depending on the values to extract. These extractions are also not completely objective. A color converted from a color model to another one (i.e. RGB to HSL) and then converted back to the original model does not have exactly the same value as the original image. This is primarily due to the fact that the image planes are encoded on 8 bits, which causes some data loss.

The main operations on color images are as follows:

- extract or replace a color image plane (R,G,B,H,S,L,V)
- apply a threshold to a color image based on one of the three color models (RGB, HSL, or HSV)
- perform a histogram on a color image based on one of the three color models (RGB, HSL, or HSV)

The other functions serve as auxiliary functions.

IPI_ExtractColorPlanes

```
IPIError = IPI_ExtractColorPlanes (IPIImageRef color_source_image,
IPIImageRef red_hue_plane, IPIImageRef green_sat_plane, IPIImageRef
blue_light_val_plane, IPIColorMode color_mode);
```

Purpose

This function extracts planes from a color image in RGB, HSV, or HSL mode. It is possible to extract only the selected planes.

Image type: `IPI_PIXEL_RGB32`

Image plane type: `IPI_PIXEL_U8`

Input

`color_source_image` is the color image from which the color planes are extracted.

`red_hue_plane` is the first destination image. It contains either the red plane (`color_mode` RGB) or the hue plane (`color_mode` HSL or 2). The corresponding color plane is not extracted if this parameter is equal to `IPI_NOIMAGE`.

`green_sat_plane` is the second destination image. It contains either the green plane (`color mode` RGB) or the saturation plane (`color mode` HSL or 2). The color plane is not extracted if this parameter is equal to `IPI_NOIMAGE`.

`blue_light_val_plane` is the third destination image. It contains either the blue plane (`color_mode` RGB), the light plane (`color_mode` HSL), or the value plane (`color_mode` HSV). The color plane is not extracted if this parameter is equal to `IPI_NOIMAGE`.

`color_mode` defines the color mode used for this operation. It can take one of the following predefined values:

- `IPI_RGB`—processing in red, green, and blue
- `IPI_HSL`—processing in hue, saturation, and light
- `IPI_HSV`—processing in hue, saturation, and value

IPI_ReplaceColorPlanes

```
IPIError = IPI_ReplaceColorPlanes (IPIImageRef color_source_image,
IPIImageRef color_dest_image, IPIImageRef red_hue_plane,
IPIImageRef green_sat_plane, IPIImageRef blue_light_val_plane,
IPIColorMode color_mode);
```

Purpose

This function replaces one or more planes in a color image, in RGB, HSL, or HSV mode.

- If the three plane images are defined, the `color_source_image` is not necessary, only the `color_dest_image` is used.
- If one or two plane images are defined, the `color_source_image` is necessary.

Note: ***All source images must have the same size.***

Image type: `IPI_PIXEL_RGB32`, `IPI_PIXEL_U8`

Input

`color_source_image` is the color image, where one or more planes are replaced.

`color_dest_image` is the resulting color image. This image is not necessary if the three planes are defined and the `color_source_image` is not defined.

`red_hue_plane` is the first source image. It contains either the red plane (`color_mode` RGB) or the hue plane (`color_mode` HSL or HSV). The red or hue plane is not replaced if this parameter is equal to `IPI_NOIMAGE`.

`green_sat_plane` is the second source image. It contains either the green plane (`color_mode` RGB) or the saturation plane (`color_mode` HSL or HSV). The green or saturation plane is not replaced if this parameter is equal to `IPI_NOIMAGE`.

`blue_light_val_plane` is the third source image. It contains either the blue plane (`color_mode` RGB), the light plane (`color_mode` HSL), or the value plane (`color_mode` HSV). The blue, light, or value plane is not replaced if this parameter is equal to `IPI_NOIMAGE`.

`color_mode` defines the color mode used for the operation. It can take one of the following predefined values:

- `IPI_RGB`—processing in red, green and blue
- `IPI_HSL`—processing in hue, saturation, and light
- `IPI_HSV`—processing in hue, saturation, and value

IPI_ColorEqualize

```
IPIError = IPI_ColorEqualize (IPIImageRef color_source_image,
IPIImageRef color_dest_image, int color_equalization);
```

Purpose

This function equalizes a color image. It works by equalizing either the lightness plane or the three color planes (red, green, and blue).

Image type: `IPI_PIXEL_RGB3HSV`

Input

`color_source_image` is the color image to equalize.

`color_dest_image` is the resulting color image.

`color_equalization` equalizes each separate plane if TRUE. It only equalizes the light plane if FALSE.

IPI_ColorHistogram

```
IPIError = IPI_ColorHistogram (IPIImageRef color_image, IPIImageRef
mask_image, int number_of_classes, IPIColorMode color_mode, int
red_hue_histogram[], int green_sat_histogram[], int
blue_light_val_histogram[], IPIHistoReport *red_hue_report,
IPIHistoReport *green_sat_report, IPIHistoReport
*blue_light_val_report);
```

Purpose

This function computes and displays the histograms extracted from the three planes of an image. It works in RGB, HSL, or HSV mode.

The three histogram reports contain the following elements:

- `minimalValue`—lower pixel value found in the calculated area
- `maximalValue`—higher pixel value found in the calculated area
- `startingValue`—always equal to 0
- `interval`—width of each class
- `meanValue`—mean value of the pixel values in the calculated area

- `stdDeviation`—standard deviation of the pixel values in the calculated area. Values are distributed better in the histogram and the corresponding image as `stdDeviation` increases.
- `area`—number of pixels used in the histogram calculation. This is influenced by the contents of `mask_image`.

Image type: `IPI_PIXEL_RGB32`

Mask type: `IPI_PIXEL_U8`

Input

`color_image` is the color image used to compute the histogram.

`mask_image` indicates the region to use for computing the histogram. Only pixels in the original image that correspond to a non-NULL pixel in the mask are used to compute the histogram. A histogram on the complete image occurs if `mask_image` is equal to `IPI_NOMASK`.

`number_of_classes` indicates the number of classes of the histograms (i.e. the number of elements of each histogram array).

`color_mode` defines the color mode used for the operation. It can take one of the following predefined values:

- `IPI_RGB`—processing in red, green, and blue
- `IPI_HSL`—processing in hue, saturation, and light
- `IPI_HSV`—processing in hue, saturation, and value

Output

`red_hue_histogram` returns the red (`color_mode` RGB) or the hue (`color_mode` HSL) histogram values in an array. It can be a NULL pointer if you do not need this histogram.

`green_sat_histogram` returns the green (`color_mode` RGB) or the saturation (`color_mode` HSL) histogram values in an array. It can be a NULL pointer if you do not need this histogram.

`blue_light_val_histogram` returns the blue (color mode RGB), the lightness (color mode HSL), or the value (color mode HSV) histogram values in an array. It can be a NULL pointer if you do not need this histogram.

`red_hue_report` is a structure filled with detailed statistics from the histogram calculated on the red or the hue plane (depending on the `color_mode`). It can be a NULL pointer if you do not need this report.

`green_sat_report` is a structure filled with detailed statistics from the histogram calculated on the green or the saturation plane (depending on the `color_mode`). It can be a NULL pointer if you do not need this report.

`blue_light_val_report` is a structure filled with detailed statistics from a histogram calculated on the blue, the lightness, or the value plane (depending on the `color_mode`). It can be a NULL pointer if you do not need this report.

IPI_ColorThreshold

```
IPIError = IPI_ColorThreshold (IPIImageRef color_source_image,
IPIImageRef dest_image, IPIColorMode color_mode, float new_value,
float red_hue_min_value, float red_hue_max_value, float
green_sat_min_value, float green_sat_max_value, float
blue_light_val_min_value, float blue_light_val_max_value);
```

Purpose

This function applies a threshold to the three planes of an image and places the result into an 8-bit image.

The function tests each range (`red_hue` range, `green_sat` range, and `blue_light_val` range, as defined by `min_value` and `max_value`) to determine if the corresponding pixel from the `color_source_image` is set to the value in `new_value`. If a pixel from the `color_source_image` does not have a value indicated in all three ranges, the corresponding pixel is set to 0 in `dest_image` .

Image type: `IPI_PIXEL_RGB32`, `IPI_PIXEL_U8`

Input

`color_source_image` is the color image to apply a threshold to.

`dest_image` is the resulting image.

`color_mode` defines the color mode used for the operation. It can take one of the following predefined values:

- `IPI_RGB`—processing in red, green, and blue
- `IPI_HSL`—processing in hue, saturation, and light
- `IPI_HSV`—processing in hue, saturation, and value

`new_value` indicates the value assigned to pixels in `dest_image` when the corresponding pixels from the `color_source_image` have values within all three ranges.

Any pixel values not included in the defined range are reset to zero in the destination image.

The pixel values included in these ranges are set to the value indicated in `new_value`.

`red_hue_min_value` is the lower limit of the range in the red or the hue plane.

`red_hue_max_value` is the higher limit of the range in the red or the hue plane.

`green_sat_min_value` is the lower limit of the range in the green or the saturation plane.

`green_sat_max_value` is the higher limit of the range in the green or the saturation plane.

`blue_light_val_min_value` is the lower limit of the range in the blue or the light or the value plane.

`blue_light_val_max_value` is the higher limit of the range in the blue or the light or the value plane.

IPI_ColorUserLookup

```
IPIError = IPI_ColorUserLookup (IPIImageRef color_source_image,
IPIImageRef mask_image, IPIImageRef color_dest_image, PIColorMode
color_mode, int lookup_format, void *red_hue_lookup, void
*green_sat_lookup, void *blue_light_val_lookup);
```

Purpose

This function applies a lookup table (LUT) to each color plane.

Image type: `IPI_PIXEL_RGB32`

Mask type: `IPI_PIXEL_U8`

Input

`color_source_image` is the color image on which the LUT is applied.

`mask_image` indicates the region in the image where the LUT is applied. Only pixels in the original image that correspond to the equivalent pixel in the mask are processed (if the value in the mask is not 0). A LUT on the complete image occurs if `mask_image` is equal to `IPI_NOMASK`.

`color_dest_image` is the resulting color image.

`color_mode` defines the color mode used for the operation. It can be one of the following predefined values:

- `IPI_RGB`—processing in red, green, and blue
- `IPI_HSL`—processing in hue, saturation, and light
- `IPI_HSV`—processing in hue, saturation, and value

`lookup_format` indicates the data type of the lookup table using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

`red_hue_lookup` is the LUT to apply to the first color plane (depending on the `color_mode`). This array can contain up to 256 elements. It is filled automatically if less than 256 elements are indicated.

`green_sat_lookup` is the LUT to apply to the second color plane (depending on the `color_mode`). This array can contain up to 256 elements. It is filled automatically if less than 256 elements are indicated.

`blue_light_val_lookup` is the LUT to apply to the third color plane (depending on the `color_mode`). This array can contain up to 256 elements. It is filled automatically if less than 256 elements are indicated.

The automatic LUT filling leaves all pixels with their original values. If the lookup is equal to NULL, this array is empty and no replacement occurs on the plane.

IPI_GetColorPixel

```
IPIError = IPI_GetColorPixel (IPIImageRef color_image, int
x_coordinate, int y_coordinate, int *rgb_color_value);
```

Purpose

This function reads the pixel value from a color image.

Image type: `IPI_PIXEL_RGB32`

Input

`color_image` is the color image from which you can get color pixel values.

`x_coordinate` is the horizontal position of the pixel.

`y_coordinate` is the vertical position of the pixel.

Output

`rgb_color_value` returns the color pixel value.

IPI_SetColorPixel

```
IPIError = IPI_SetColorPixel (IPIImageRef color_image, int
x_coordinate, int y_coordinate, int rgb_color_value);
```

Purpose

This function changes the pixel value of a color image.

Image type: `IPI_PIXEL_RGB32`

Input

`color_image` is the color image where the new color pixel value is written.

`x_coordinate` is the horizontal position of the pixel.

`y_coordinate` is the vertical position of the pixel.

`rgb_color_value` indicates the new color pixel value.

IPI_GetColorLine

```
IPIError = IPI_GetColorLine (IPIImageRef color_image, Point start,
Point end, int color_array[], int *nb_of_elements);
```

Purpose

This function reads a line of pixels from a color image into an array and returns the number of elements in this array. The line is defined by a start point and an end point.

Image type: `IPI_PIXEL_RGB32`

Input

`color_image` is the color image where the line is read.

`start` is the start point of the line of pixels.

`end` is the end point of the line of pixels.

Output

`color_array` is an array allocated by you. It must be big enough to contain all elements of the line.

`nb_of_elements` returns the number of elements in the array.

IPI_SetColorLine

```
IPIError = IPI_SetColorLine (IPIImageRef color_image, Point start,
Point end, int color_array[], int nb_of_elements);
```

Purpose

This function writes a line of pixels in a color image. The new values of the pixels are contained in a color array defined by you.

Image type: `IPI_PIXEL_RGB32`

Input

`color_image` is the color image where the line is written.

`start` is the start point of the line of pixels.

`end` is the end point of the line of pixels.

`color_array` is the allocated array containing the new color pixel values. They are copied to the image.

`nb_of_elements` indicates the number of pixels of the array.

☞ **Note:** ***If the line defined by*** `start` ***and*** `end` ***is longer than*** `nb_of_elements`***, it is shortened. That is, only the pixels defined by*** `nb_of_elements` ***are copied. If the line is shorter, the array is not entirely written to the image.***

IPI_ColorImageToArray

```
IPIError = IPI_ColorImageToArray (IPIImageRef color_image, Rect
rectangle, int color_array[], int *array_x_size, int
*array_y_size);
```

Purpose

This function extracts a pixel array from a color image and returns the size of this array.

Image type: `IPI_PIXEL_RGB32`

Input

`color_image` is the source color image.

`rectangle` is a `Rect` structure containing the coordinates and the size of the rectangle to extract from the image. The operation is applied to the entire image if this parameter is equal to `IPI_FULL_RECT`. The rectangle is defined by a buffer of integers allocated by you.

Output

`color_array` is an array allocated by you. It must be big enough to contain all copied elements of the rectangle.

`array_x_size` returns the horizontal number of elements in the color array.

`array_y_size` returns the vertical number of elements in the color array.

IPI_ArrayToColorImage

```
IPIError = IPI_ArrayToColorImage (IPIImageRef color_image, int
color_array[], int array_x_size, int array_y_size);
```

Purpose

This function replaces the pixels of a color image with pixels defined in a color array allocated by you.

Image type: `IPI_PIXEL_RGB32`

Input

`color_image` is the color image to modify. The resulting image is resized to `array_x_size` and `array_y_size`.

`color_array` is the allocated array.

`array_x_size` indicates the horizontal size of the pixels array.

`array_y_size` indicates the vertical size of the pixels array.

IPI_ColorConversion

```
IPIError = IPI_ColorConversion (IPIColorMode src_color_mode,
unsigned char src_red_hue, unsigned char src_grn_sat, unsigned char
src_blu_light_val, IPIColorMode dst_color_mode, unsigned char
*dst_red_hue, unsigned char *dst_grn_sat, unsigned char
*dst_blu_light_val);
```

Purpose

This function converts the color pixel values from one color mode to another.

Input

`src_color_mode` defines the source color mode used for the operation. It can take any one of the following predefined values:

- `IPI_RGB`—processing in red, green, and blue
- `IPI_HSL`—processing in hue, saturation, and light
- `IPI_HSV`—processing in hue, saturation, and value

`src_red_hue` is the red or hue source value of the pixel.

`src_grn_sat` is the green or sat source value of the pixel.

`src_blu_light_val` is the blue or light or val source value of the pixel.

`dst_color_mode` defines the destination color mode used for the operation. It can be one of the following predefined values:

- `IPI_RGB`—processing in red, green, and blue
- `IPI_HSL`—processing in hue, saturation, and light
- `IPI_HSV`—processing in hue, saturation, and value

Output

`dst_red_hue` points to the value of the red or hue of the color pixel.

`dst_grn_sat` points to the value of the green or sat of the color pixel.

`dst_blu_light_val` points to the value of the blue, light, or val of the color pixel.

IPI_IntegerToColor

```
IPIError = IPI_IntegerToColor(int source_integer_array[], int
nb_of_elements, IPIColorMode dst_color_mode, int
dests_array_format, void *dst_red_hue_array, void
*dst_grn_sat_array, void *dst_blu_light_val_array);
```

Purpose

This function converts an array of integers into a color array.

Input

`source_integer_array` is the array of integers.

`nb_of_elements` indicates the number of pixels of the color array.

`dst_color_mode` defines the source color mode used for the operation. It can take any one of the following predefined values:

- `IPI_RGB`—processing in red, green, and blue
- `IPI_HSL`—processing in hue, saturation, and light
- `IPI_HSV`—processing in hue, saturation, and value

`dests_array_format` indicates the data type of the output arrays using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

`dst_red_hue_array` points to the red or hue color array.

`dst_grn_sat_array` points to the green or saturation color array.

`dst_blu_light_val_array` points to the blue, light, or value color array.

IPI_ColorToInteger

```
IPIError = IPI_ColorToInteger (void *source_red_hue_array, void
*source_grn_sat_array, void *source_blu_light_val_array, int
nb_of_elements, IPIColorMode source_color_mode, int
sources_array_format, int dest_integer_array[]);
```

Purpose

This function converts a pixel color array into an array of integers.

Input

`source_red_hue_array` is the red or hue source color array.

`source_grn_sat_array` is the green or saturation source color array.

`source_blu_light_val_array` is the blue, light, or value source color array.

`nb_of_elements` indicates the number of pixels in the array of integers.

`source_color_mode` defines the source color mode used for the operation. It can take any one of the following predefined values:

- `IPI_RGB`—processing in red, green, and blue
- `IPI_HSL`—processing in hue, saturation, and light
- `IPI_HSV`—processing in hue, saturation, and value

`sources_array_format` indicates the data type of the arrays using one of the following LabWindows standard values:

- `VAL_CHAR`—character
- `VAL_SHORT_INTEGER`—short integer
- `VAL_INTEGER`—integer
- `VAL_FLOAT`—floating point
- `VAL_DOUBLE`—double-precision
- `VAL_UNSIGNED_SHORT_INTEGER`—unsigned short integer
- `VAL_UNSIGNED_INTEGER`—unsigned integer
- `VAL_UNSIGNED_CHAR`—unsigned character

`dest_integer_array` is an array of integers allocated by you. It must be big enough to contain all elements of the source array.

Appendix

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-On-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at (512) 418-1111.

E-Mail Support (currently U.S. only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

	Telephone	Fax
Australia	02 9874 4100	02 9874 4455
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 527 2321	09 502 2930
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 5734815	03 5734816
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ____

Company ____

Address ____

____

Fax (___)___________________ Phone (___) ____________________________

Computer brand ________________ Model ________________ Processor___________________

Operating system (include version number) _____________________________________

Clock speed ______MHz RAM _____MB Display adapter __________________________

Mouse ___yes ___no Other adapters installed _________________________________

Hard disk capacity _____MB Brand ____

Instruments used ____

____

National Instruments hardware product model__________ Revision _______________________

Configuration ____

National Instruments software product___________________________Version ____________

Configuration ____

The problem is: ____

____

____

____

____

List any error messages: ____

____

____

The following steps reproduce the problem:_______________________________________

____

____

____

____

____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *IMAQ™ Vision for LabWindows®/CVI™*

Edition Date: February 1997

Part Number: 321424A-01

Please comment on the completeness, clarity, and organization of the manual.

If you find errors in the manual, please record the page numbers and describe the errors.

Thank you for your help.

Name ______________________________

Title ______________________________

Company ______________________________

Address ______________________________

____

Phone (___)______________________ Fax (___) ______________________

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

Index