Intel Image Processing Library

Reference Manual

Order Number 663791B-002

How to Use This Online Manual

Click to hide or show subtopics when the bookmarks are shown.

Double-click to jump to a topic when the bookmarks are shown.

Click to display bookmarks.

Click to display thumbnails.

Click to close bookmark or thumbnail view.

Click and use on the page to drag the page in vertical direction.

Click and drag to the page to magnify the view.

Click and drag to the page to reduce the view.

Click and drag the selection cursor to the page.

Click to go to the first page of the manual.

Click to go to the previous page.

Click to go to the next page.

Click to go to the last page.

Click to return back to the previous view. Use this button when you need to go back after using the jump button (see below).

Click to go forward from the previous view.

Click to set 100% of the page view.

Click to display the entire page within the window.

Click to fill the width of the window.

Click to open a dialog to search for a word or multiple words.

Click jump button on manual pages to jump to the related subjects. Use the return back icon above to go back.

Printing an Online File. Select **Print** from the **File** menu to print an online file. The dialog that opens allows you to print full text, range of pages, or selection.
Viewing Multiple Online Manuals. Select **Open** from the **File** menu, and open a .PDF file you need. Select **Cascade** from the **Window** menu to view multiple files.
Resizing the Bookmark Area. Drag the double-headed arrow that appears on the area's border as you pass over it.
Jumping to Topics. Throughout the text of this manual, you can jump to different topics by clicking on keywords printed in green color, underlined style or on page numbers in a box.

To return to the page from which you jumped, use the ⏪ icon in the tool bar. Try this example:

This software is briefly described in the Overview; see page 1-1.

If you click on the phrase printed in green color, underlined style, or on the page number, the Overview opens.

Intel Image Processing Library Reference Manual

Order Number: 663791B-002

Revision	Revision History	Date
B-001	Documents release 1.0 Beta of the library.	01/20/97
B-002	Documents release 1.0 Beta 2 of the library. The iplBitonalToGray, iplCreateColorTwist, iplDeleteColorTwist, iplLShiftS, iplMultiplyScale, iplMultiplySScale, and iplNot functions have been added. Miscellaneous edits have been made.	04/04/97

Contents

Figures

Tables

Examples

Overview

1

This manual describes the structure, operation and functions of the Intel Image Processing Library (IPL). This library supports many functions whose performance can be significantly enhanced on the Intel Architecture (IA), particularly the MMX^{TM} technology.

The manual describes the architecture of the IPL data and execution and provides detailed descriptions of the functions included in the Intel Image Processing Library.

This chapter introduces the Intel Image Processing Library and explains the organization of this manual.

About This Software

The Intel Image Processing Library focuses on taking advantage of the parallelism of the SIMD (single-instruction, multiple-data) instructions that comprise the MMX technology. This technology improves the performance of computationally intensive image processing functions. Thus this library includes a set of functions whose performance significantly improves when used with the Intel Architecture MMX technology. The library does not support the reading and writing of a wide variety of image file formats or the display of images.

Hardware and Software Requirements

The Intel Image Processing Library runs on personal computers that are based on Intel Architecture processors and running Microsoft* Windows*, Windows 95*, or Windows NT*. The library integrates into the customer's application or library written in C or C++.

About This Manual

This manual provides a background of the image and execution architecture of the Intel Image Processing Library as well as detailed descriptions of the IPL functions. The IPL functions are combined in groups by their functionality. Each group of functions is described in a separate chapter (chapters 3 through 11).

Manual Organization

This manual contains eleven chapters:

Chapter 1	"Overview." Introduces the Image Processing Library software, provides information on manual organization, and explains notational conventions.
Chapter 2	"Image Architecture." Describes the data formats supported, the execution architecture, and image tiling. The data formats include color models, data types, data order, and coordinate systems. The execution architecture discussion focuses on overflow and underflow issues and in-place and out-of-place operations.
Chapter 3	"Error Handling." Provides information on the error-handling functions included with the library. User-defined error handler is also described.
Chapter 4	"Image Creation and Access." Describes the functions used to: create, set, and access image attributes; set image border and tiling; and allocate the memory for different data types. The chapter also describes the functions that facilitate operations in the window environment.

Chapter 5	“Image Arithmetic and Logical Operations.” Describes image processing operations that modify pixel values using simple arithmetic or logical operations. These operations include monadic operations (single input image) and dyadic operations (two input images).
Chapter 6	“Image Filtering.” Describes linear and non-linear filtering operations that can be applied to images.
Chapter 7	“Linear Image Transforms.” Describes the fast Fourier transform (FFT) and Discrete Cosine Transform (DCT) implemented in the IPL.
Chapter 8	“Morphological Operations.” Describes the morphological operations supported in the library: simple Erosion, Dilation, Opening and Closing
Chapter 9	“Color Space Conversion.” Describes the color space conversions supported in the library; for example, color reduction from high resolution color to low resolution color; conversion from Palette to Absolute color and vice versa; conversion to different color models.
Chapter 10	“Histogram and Thresholding Functions.” Describes functions that treat an image on a pixel-by-pixel basis: operations that alter the histogram of the image; contrast stretching, histogram computation, histogram equalization and thresholding.
Chapter 11	“Image Geometric Transforms.” Describes geometric transforms: Zoom, Decimate, Rotate, and Mirror.

The manual also includes Appendix that lists supported image attributes and operation modes, Glossary of terms, Bibliography, and Index.

Function Descriptions

In Chapters 3 through 11, each function is introduced by name (without the `ipl` prefix) and a brief description of its purpose. This is followed by the function call sequence, more detailed description of the function’s purpose, and definitions of its arguments. The following sections are included in each function description:

Arguments	Describes all the function arguments.
Discussion	Defines the function and describes the operation performed by the function. Often, code examples and the equations the function implements are included.
Return Value	If present, describes a value indicating the result of the function execution.
Application Notes	If present, describe any special information which application programmers or other users of the function need to know.
See Also	If present, lists the names of functions which perform related tasks.

Audience for This Manual

The manual is intended for the developers of image processing applications and image processing libraries. Both parts of the audience are expected to be experienced in using C and to have a working knowledge of the vocabulary and principles of image processing. The developers of image processing software can use the Intel IPL capabilities to improve performance on IA with MMX technology.

Online Version

This manual is available in an online hypertext format. To obtain a hard copy of the manual, print the online file using the printing capability of Adobe* Acrobat, the tool used for the online presentation of the document.

Related Publications

For more information about computer graphics concepts and objects, refer to the books and materials listed in the Bibliography.

Notational Conventions

In this manual, notational conventions include:

- Fonts used for distinction between the text and the code
- Naming conventions
- Function name conventions

Font Conventions

The following font conventions are used:

`UPPERCASE COURIER`	Used in the text for IPL constant identifiers; for example, `IPL_DEPTH_1U`.
`lowercase courier`	Mixed with the uppercase in function names as in `SetExecutionMode`; also used for key words in code examples; for example, in the function call statement `void iplSquare()`.
`lowercase mixed with UpperCase Courier italic`	Variables in arguments and parameters discussion; for example, *`mode`*, *`dstImage`*.

Naming Conventions

The following data type conventions are used by the IPL library:

- Constant identifiers are in uppercase; for example, `IPL_SIDE_LEFT`.
- All constant identifiers have the `IPL` prefix.
- All of the function names have the `ipl` prefix. In code examples, you can distinguish the IPL interface functions from the application functions by this prefix.

NOTE. *In this manual, the `ipl` prefix in function names is always used in the code examples. In the text, this prefix is sometimes omitted.*

- All image header structures have the `Ipl` prefix; for example, `IplImage`, `IplROI`.
- Each new part of a function name starts with an uppercase character, without underscore; for example, `iplAlphaComposite`.

Function Name Conventions

The function names in the IPL library typically begin with the `ipl` prefix and have the following general format:

```
ipl < action > < target > < mod >()
```

where

`action`	indicates the core functionality; for example, `-Set-`, `-Create-`, or `-Convert-`.
`target`	indicates the area where image processing is being enacted; for example, `-ConvKernel` or `-FromDIB`.
	In a number of cases, the target consists of two or more words; for example, `-ConvKernel` in the function `CreateConvKernel`. Some function names consist of an *`action`* or *`target`* only; for example, the functions `Multiply` or `RealFft2D`, respectively.
`mod`	The *`mod`* field is optional and indicates a modification to the core functionality of a function. For example, in the name `iplAlphaCompositeC()`, `C` indicates that this function is using constant alpha values.

Image Architecture

2

This chapter describes the data architecture and the execution architecture of the Intel Image Processing Library (IPL). The data formats supported by IPL define the image architecture as explained in the sections that follow.

Data Architecture

An IPL image has a header that describes the image as a list of attributes and pointers to the data associated with the image. Library functions use the image header to get the format and characteristics of the image(s) passed to the functions. Based on the information obtained from the header, the functions make appropriate calls to set the data structures. Images can have different organization of data. IPL supports numerous data formats that use different color models, data types, data order, and coordinate systems.

Color Models

The IPL image format supports the following color models:

- Monochrome or gray scale image (one color channel)
- Color image (3 or 4 color channels)
- Multi-spectral image (any number of channels).

Color models are defined by the number of channels and the colors they contain. Examples of three-channel models are RGB, HSV, CMY, and YCC. Examples of four-channel color models are CMYK and RGBA.

Image processing operations can be performed on one or all channels in the image. The operations are performed without specific identification of the colors, unless it is a certain color conversion operation where color identification is required.

The multi-spectral image (MSI) model is used for general purpose images. It is used for any kind of multi-spectral data and any kind of image. For example, the Fourier transform operation writes transform coefficients of color or monochrome images to this model—one channel for each channel in the input. The result can be viewed as an MSI image. An MSI image can contain any number of color channels; they may even correspond to invisible parts of the spectrum. The library functions do not need to identify any specific MSI image channels.

Data Types and Palettes

The parameter that determines the image data type is the pixel depth in bits. The data could be signed or unsigned. The following data types are supported for various color models (s = signed, u = unsigned):

Gray scale	1, 8s, 8u, 16s, and 16u bits per pixel
Color (three-channel)	8u and 16u bits per channel
Four-channel and MSI	8s, 8u, 16s, 16u, 32s bits per channel

IPL supports only absolute color images in which each pixel is represented by the channel intensities. For example, in an absolute color 24-bit RGB image, three bytes (24 bits) per pixel represent the three channel intensities. LUT (lookup table) images, that is, palette color images are not supported. You must convert palette images to absolute color images for further processing by the IPL functions. The library provides functions to convert DIB palette images to IPL absolute color images.

Color images with 8, 16, or 32 bits per channel simply pack each channel, respectively, into a byte, word, or doubleword. All channels within a given image have the same data type.

Signed data (8s, 16s, or 32s) are used for storing the output of some image processing operations; for example, this is the case for transforms such as FFT. Unless specified otherwise, signed data cannot be used as input to image processing operations.

The Sequence and Order of Color Channels

Channel sequence corresponds to the order of the color channels in absolute color images. For example, in an RGB image the channels could be stored in the sequence RGB or in the sequence BGR. The sequence of the channels is not important to the image processing functions which do not identify the channels.

NOTE. *For the color conversion operations, the information about channel sequence is required and therefore must be provided.*

For images with pixel-oriented data, the channel sequence corresponds to the color data order for each pixel. Data ordering corresponds to the way the color data is arranged: by planes or by pixels. Table 2-1 lists the orderings that are supported for planes and for pixels.

Table 2-1 Data Ordering

Data Ordering	Description	RGB Example (channel ordering = RGB)
Pixel-oriented	All channels for each pixel are clustered.	RGBRGBRGB (line 1) RGBRGBRGB (line 2) RGBRGBRGB (line 3)
Plane-oriented	All image data for each channel is contiguous followed by the next channel.	RRRRRRRRR (line 1) RRRRRRRRR (line 2) R plane RRRRRRRRR (line 3) GGGGGGGGG (line 1) GGGGGGGGG (line 2) G plane GGGGGGGGG (line 3) ...

Coordinate Systems

Two coordinate systems are supported by the IPL image format.

- The origin of the image is in the top left corner, the x values increase from left to right, and y values increase from top to bottom.
- The origin of the image is in the bottom left corner, the x values increase from left to right, and y values increase from the bottom to the top.

Image Regions

A very important concept in the IPL library architecture is an image region of interest (ROI). All image processing functions can operate not only on entire images but also on image regions.

Depending on the processing needs, the following image regions can be specified:

- A channel of interest (COI). A COI can be one or all channels of the image. By default, unless the COI is changed by the `SetROI()` function, processing will be carried out on all channels in the image.
- A rectangular region of interest (rectangular ROI). A rectangular ROI is a portion of the image or, possibly, the entire image. By default, unless changed by the `SetROI()` function, the entire image is the rectangular region of interest.

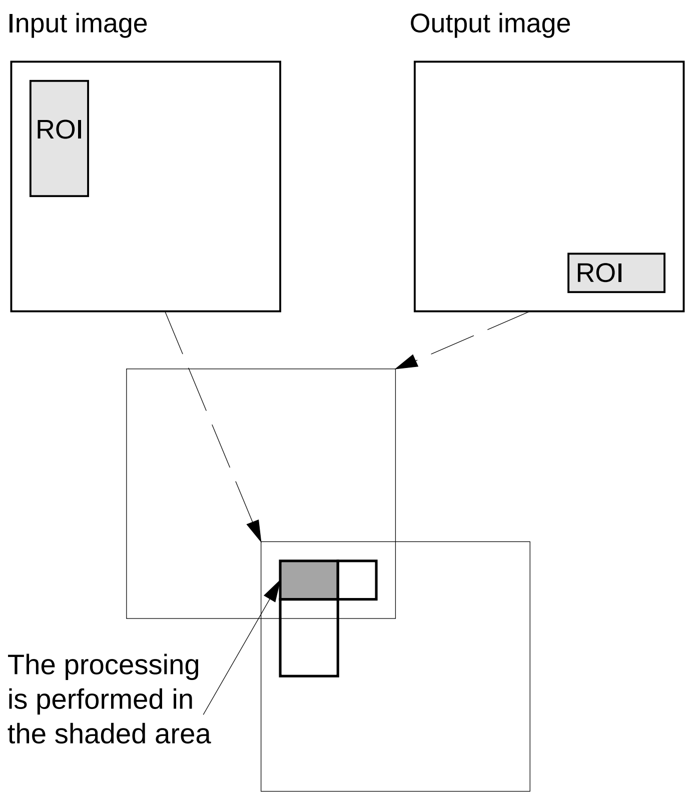
An IPL image can simultaneously have a rectangular ROI and a channel of interest. If this is the case, operations will be performed on the "intersection" of these two ROIs.

Thus an image region specifies some part of an image or the entire image. Once set, the region information of the IPL image remains the same until changed by the function `SetROI()`.

Setting an ROI for Multi-Image Operations

Figure 2-1 illustrates image processing operations that take one or more input images and store the results onto an output image.

Figure 2-1 Setting an ROI for Multi-Image Operations



All images (input and output) in Figure 2-1 have rectangular ROIs that specify either the entire image or specific regions set by the `SetROI()` function. The first step is to align the rectangular ROIs of all the images so that their top left corners coincide. The operation is, then, performed in the rectangular region where all the images overlap. This scheme gives much flexibility, effectively enabling translation of image data (even for equal-size images) from one region of an input image to another region of an output image.

To successfully perform an image processing operation, one of the following conditions must be met for the channel of interest (COI):

- Each image (input and output) has one channel in the channel COI,
- Each image (input and output) has all channels included in the ROI (COI = 0) and all images (input and output) have the same number of channels (one or more).

If one image (input or output) has one channel in its COI and another image (input or output) has more than one channel included in its COI, an error will occur.

Alpha (Opacity) Channel

In addition to the color channels, an IPL image can have one alpha channel, also known as an opacity channel, which is mainly used for image compositing operations (see "Image Compositing Based on Opacity" in Chapter 5).

The alpha channel is treated like any other channel in the IPL format; you are allowed not to identify it when not required. However, alpha channels must be explicitly specified for the functions that require them (see, for example, the `iplAlphaComposite` function).

Scanline Alignment

Image row data (scanline) can be aligned on doubleword (32-bit) or quadword (64-bit) boundaries. Each row is padded with zeros if required. For maximum performance with MMX™ technology, it is important to have the image data aligned on quadword boundaries.

Image Dimensions

There is no practical limit of the image size. An unsigned long integer is used for the height and width of the image. This allows you to create images of size up to 2^{31} by 2^{31} pixels, which is much beyond the hardware and OS constraints of today's PCs or workstations.

Execution Architecture

This section describes the execution time issues such as overflow/underflow handling and in-place and out-of-place operations. Cache optimization and deferred execution mode will be discussed in the next release.

Handling Overflow and Underflow

Overflow and underflow are handled in each image processing function. The default mode of operation is saturation which is a mode that prevents from potential overflow or underflow of the values. In saturation mode, when an overflow of a value is about to happen, this value is clamped to the maximum permissible value (for example, 255 for an unsigned byte). Similarly, when underflow of a value is about to happen, it is clamped to the minimum permissible value, which is always zero for the case of unsigned bytes.

In-Place and Out-of-Place Operations

All image processing operations in the library can be in-place or out-of-place operations, unless it is explicitly specified that a particular operation must belong to one of these categories only. With an in-place operation, the output image is one of the input images modified (that is, the pointer to the output image is the same as the pointer to the input one). With an out-of-place operation, the output image is a new image, not the same as any of the input images.

Image Tiling

Tiling is a method of image representation in which, for reasons of efficiency, the image is broken up into smaller images, or tiles. The whole image is reconstructed by arranging the individual tiles in a grid. Usually, the tiles are of a uniform size; special techniques can be used to accommodate an overall image height or width that is not an even multiple of the tile size. Image processing applications frequently use square tiles with sizes that are multiples of two; for example, a 64-by-64 tile is typical.

In most IPL functions, tiled images may be used in the same way as non-tiled images, subject to some restrictions. The effect of functions on tiled images is always the same as that of the same function on an image of the same size and content that is not tiled. The behavior varies as stated below, particularly in the call-back requirement.

This section gives a short overview of image tiling in the IPL. In Chapter 4 you can find more information about tiling, namely, the descriptions of the `TileInfo` structure, the `imageID` parameter, and the functions `CreateTileInfo`, `SetTileInfo`, and `DeleteTileInfo`.

Tile Size

In the IPL, all tiles must be of the same size, including those on the edge of an image. The tiles on the edge of an image must contain valid data up to the border of the image; beyond that, the pixels are ignored, and the border mode is used instead.

The size of the image tiles is contained within the `IplTileInfo` structure. It is restricted to being an even multiple of 8 in each dimension.

For functions that take more than one source image, either all source images must be tiled with equally-sized tiles or they must all be non-tiled. The source and destination images tiling and tile sizes need not be the same.

IplCoord Structure

The arguments of every function with both an input and output image include a pointer to the following structure:

```
iplFunction(..., IplCoord*);
```

```
typedef struct _IplCoord {
 int xDst, yDst;
} IplCoord
```

The `xDst` and `yDst` fields are the offsets (in pixels) of the origin of the destination image from the origin of the source image.

If there are multiple source images, these images are assumed to have their ROI origins aligned with each other.

The `IplCoord` structure is not specific to tiling in that it is supported whether the source/destination images are tiled or not. Its intent is to support for tiling schemes **other** than the IPL tiling scheme by providing a method of identifying the location of the destination with respect to the source. Such a method is important if the argument images are, unbeknownst to the IPL functions, actually tiles in a user-implemented tiling scheme. The `IplCoord` parameter is ignored if `NULL`.

Call-backs

Since the `IplImage` structure does not contain any image data, functions operating on tiled images must acquire data tile-by-tile. To do this, the IPL uses a system of call-backs, in which the IPL function requests pointers to individual tiles based on need.

The call-back system is implemented (by the library user) as a single function, the prototype and behavior of which are specified below. When called **by the library**, this function provides or releases one tile's worth of data. The function is specified to the library in the `callBack` field of the `IplTileInfo` structure. The prototype is as follows:

```
void (*IplCallBack)  (const IplImage* img, int xIndex,
                      int yIndex, int mode);
```

where

`img` is the header provided for the parent image;
`xIndex` and `yIndex` are the indices of the requested tile; they refer to the tile number, not pixel number, and count from the origin at (0,0);
`mode` is one of the following:

`IPL_GET_TILE_TO_READ`	get a tile for reading; the tile data is returned in `img->tileInfo->tileData` and must not be changed;
`IPL_GET_TILE_TO_WRITE`	get a tile for writing; the tile data is returned in `img->tileInfo->tileData` and may be changed; changes will be reflected in the image;
`IPL_RELEASE_TILE`	release tile; commit writes.

Memory pointers provided by a get function will not be used after the corresponding release function has been called.

ROI and Tiling

The meaning and behavior of ROI for tiled images are identical to those for a non-tiled image.

In-Place Operations and Tiling

Functions that are called with identical source and destination images (header pointers equal) are handled correctly by the library, even with tiling. If the source and destination image pointers are not equal, no support for source and destination overlap is provided.

Note that the presence of `IplROI` and/or `IplCoord` structures does not affect this restriction.

Error Handling

3

This chapter describes the error handling facility of the Image Processing Library. The IPL functions report a variety of errors including bad arguments and out-of-memory conditions. When a function detects an error, instead of returning a status code, the function signals an error by calling `iplSetErrStatus()`. This allows the error handling mechanism to work separately from the normal flow of the image processing code. Thus, the image processing code is cleaner and more compact as shown in this example:

```
ColorTwist = iplSetColorTwist(data, scalingValue);

if(iplGetErrStatus()<0)    // check for errors
```

The error handling system is hidden within the function `iplSetColorTwist()`. As a result, this statement is uncluttered by error handling code and closely resembles a mathematical formula.

Your application should assume that every library function call may result in some error condition. The Image Processing Library performs extensive error checks (for example, `NULL` pointers, out-of-range parameters, corrupted states) for every library function.

Error macros are provided to simplify the coding for error checking and reporting. You can modify the way your application handles errors by calling `iplRedirectError()` with a pointer to your own error handling function. For more information, see "Adding Your Own Error Handler" later in this chapter. For even more flexibility, you can replace the whole error handling facility with your own code. The source code of the default error handling facility is provided.

The Image Processing Library does not process numerical exceptions (for example, overflow, underflow, and division by zero). The underlying floating point library or processor has the responsibility for catching and reporting these exceptions. A floating-point library is needed if a processor that handles floating-point is not present. You can attach an exception handler using an underlying floating-point library for your application, if your system supports such a library.

Error-handling Functions

The following sections describe the error functions in the Image Processing Library.

Error

Performs basic error handling.

```
void iplError(IPLStatus status, const char *func,
              const char *context);
```

`status`	Code that indicates the type of error (see Table 3-1, "iplError() Status Codes".)
`func`	Name of the function where the error occurred.
`context`	Additional information about the context in which the error occurred. If the value of `context` is `NULL` or empty, this string will not appear in the error message.

Discussion

The `iplError()` function must be called whenever any of the IPL functions encounters an error. The actual error reporting is handled differently, depending on whether the program is running in Windows mode or in console mode. Within each invocation mode, you can set the error mode flag to alter the behavior of the `iplError()` function. For more information on the defined error modes, see "SetErrMode" section.

To simplify the coding for error checking and reporting, the error handling system supplied by the IPL Library supports a set of error macros. See "Error Macros" for a detailed description of the error handling macros.

The `iplError()` function calls the default error reporting function. You can change the default error reporting function by calling `iplRedirectError()`. For more information, see "`RedirectError`" (for `iplRedirectError()`).

GetErrStatus SetErrStatus

Gets and sets the error codes that describe the type of error being reported.

```
typedef int IPLStatus;

IPLStatus iplGetErrStatus();

void iplSetErrStatus(IPLStatus status);
```

`status`	Code that indicates the type of error (see Table 3-1, "iplError() Status Codes").

Discussion

The `iplGetErrStatus()` and `iplSetErrStatus()` functions get and set the error status codes that describe the type of error being reported. See "Status Codes" for descriptions of each of the error status codes.

GetErrMode
SetErrMode

Gets and sets the error modes that describe how an error is processed.

```
#define IPL_ErrModeLeaf    0
#define IPL_ErrModeParent  1
#define IPL_ErrModeSilent  2
int iplGetErrMode();
void iplSetErrMode(int  errMode);
```

errMode	Indicates how errors will be processed. The possible values for *errMode* are `IPL_ErrModeLeaf`, `IPL_ErrModeParent`, or `IPL_ErrModeSilent`.

Discussion

NOTE. *This section describes how the default error handler handles errors for applications which run in console mode. If your application has a custom error handler, errors will be processed differently than described below*

The `iplSetErrMode()` function sets the error modes that describe how errors are processed. The defined error modes are `IPL_ErrModeLeaf`, `IPL_ErrModeParent`, and `IPL_ErrModeSilent`.

If you specify `IPL_ErrModeLeaf`, errors are processed in the “leaves” of the function call tree. The `iplError()` function (in console mode) prints an error message describing *status*, *func*, and *context*. It then terminates the program.

If you specify `IPL_ErrModeParent`, errors are processed in the "parents" of the function call tree. When `iplError()` is called as the result of detecting an error, an error message will print, but the program will not terminate. Each time a function calls another function, it must check to see if an error has occurred. When an error occurs, the function should call `iplError()` specifying `IPL_StsBackTrace`, and then return. The macro `IPL_ERRCHK()` may be used to perform both the error check and back-trace call. This passes the error "up" the function call tree until eventually some parent function (possibly `main()`) detects the error and terminates the program.

`IPL_ErrModeSilent` is similar to `IPL_ErrModeParent`, except that error messages are not printed.

`IPL_ErrModeLeaf` is the default, and is the simplest method of processing errors. `IPL_ErrModeParent` requires more programming effort, but provides more detailed information about where and why an error occurred. All of the functions in the library support both options (that is, they use IPL_ERRCHK() after function calls). If an application uses the IPL_ErrModeParent option, it is essential that it check for errors after all library functions that it calls.

The status code of the last detected error is stored into the global variable `IplLastStatus` and can be returned by calling `iplGetErrStatus()`. The value of this variable may be used by the application during the back-trace process to determine what type of error initiated the back trace.

ErrorStr

Translates an error or status code into a textual description.

```
const char* iplErrorStr(IPLStatus status);
```

`status`	Code that indicates the type of error (see Table 3-1, "iplError() Status Codes").

Discussion

The function `iplErrorStr()` returns a short string describing *`status`*. Use this function to produce error messages for users. The returned pointer is a pointer to an internal static buffer that may be overwritten on the next call to `iplErrorStr()`.

RedirectError

Assigns a new error handler to call when an error occurs.

```
IPLErrCallBack iplRedirectError(IPLErrCallBack  func);
```

`func`	Pointer to the function that will be called when an error occurs.

Discussion

The `iplRedirectError()` function assigns a new function to be called when an error occurs in the IPL Library. If *`func`* is `NULL`, `iplRedirectError()` installs the IPL Library's default error handler.

The return value of `iplRedirectError()` is a pointer to the previously assigned error handling function.

For the definition of the function typedef `IPLErrCallBack`, see the include file `iplerror.h`. See "Adding Your Own Error Handler" for more information on the `iplRedirectError()` function.

Error Macros

The error macros associated with the `iplError()` function are described below.

```
#define IPL_ERROR(status, func, context) \
     iplError((status),(func),(context);

#define IPL_ERRCHK(func, context)\
     ( (iplGetErrStatus()>=0) ? IPL_StsOk \
            : IPL_ERROR(IPL_StsBackTrace,(func),(context)) )

#define IPL_ASSERT(expr, func, context)\
     ( ( expr) ? IPL_StsOk\
            : IPL_ERROR(IPL_StsInternal,(func),(context)) )

#define IPL_RSTERR()      (iplSetErrStatus(IPL_StsOk))
```

`context`	Provides additional information about the context in which the error has occurred. If the value of *`context`* is `NULL` or empty, this string does not appear in the error message.
`expr`	An expression that checks for an error condition and returns `FALSE` if an error has occurred.
`func`	Name of the function where the error occurred.
`status`	Code that indicates the type of error (see Table 3-1, "`iplError()` Status Codes.")

Discussion

The `IPL_ASSERT()` macro checks for the error condition *`expr`* and sets the error status `IPL_StsInternal` if the error occurred.

The `IPL_ERRCHK()` macro checks to see if an error has occurred by checking the error status. If an error has occurred, `IPL_ERRCHK()` creates an error back trace message and returns a non-zero value. This macro should normally be used after any call to a function that might have signaled an error.

The `IPL_ERROR()` macro simply calls the `iplError()` function by default. This macro is used by other error macros. By changing `IPL_ERROR()` you can modify the error reporting behavior without changing a single line of source code.

The `IPL_RSTERR()` macro resets the error status to `IPL_StsOk`, thus clearing any error condition. This macro should be used by an application when it decides to ignore an error condition.

Status Codes

The status codes used by the IPL Library are described in Table 3-1. Status codes are integers, not an enumerated type. This allows an application to extend the set of status codes beyond those used by the library itself. Negative codes indicate errors, while non-negative codes indicate success.

Table 3-1 iplError() Status Codes

Status Code	Value	Description
`IPL_StsOk`	0	No error. The `iplError()` function does nothing if called with this status code.
IPL_StsBackTrace	-1	Implements a back-trace of the function calls that lead to an error. If `IPL_ERRCHK()` detects that a function call resulted in an error, it calls `IPL_ERROR()` with this status code to provide further context information for the user.
`IPL_StsError`	-2	An error of unknown origin, or of an origin not correctly described by the other error codes.
`IPL_StsInternal`	-3	An internal "consistency" error, often the result of a corrupted state structure. These errors are typically the result of a failed assertion.

continued ☞

Table 3-1 iplError() Status Codes (continued)

Status Code	Value	Description
`IPL_StsNoMem`	-4	A function attempted to allocate memory using `malloc()` or a related function and was unsuccessful. The message *`context`* indicates the intended use of the memory.
`IPL_StsBadArg`	-5	One of the arguments passed to the function is invalid. The message *`context`* indicates which argument and why.
`IPL_StsBadFunc`	-6	The function is not supported by the implementation, or the particular operation implied by the given arguments is not supported.
`IPL_StsNoConv`	-7	An iterative convergence algorithm failed to converge within a reasonable number of iterations.

Application Notes

The global variable `IplLastStatus` records the status of the last error reported. Its value is initially `IPL_StsOk`. The value of `IplLastStatus` is not explicitly set by the library function detecting an error. Instead, it is set by `iplSetErrStatus()`.

If the application decides to ignore an error, it should reset `IplLastStatus` back to `IPL_StsOk` (see `IPL_RSTERR()` under "Error Macros"). An application-supplied error-handling function must update `IplLastStatus` correctly; otherwise the Image Processing Library might fail. This is because the macro `IPL_ERRCHK()`, which is used internally to the library, refers to the value of this variable.

Error Handling Example

The following example describes the default error handling for a console application. In the example program, `test.c`, assume that the function `libFuncB()` represents a library function such as `ipl?TranslateDIB()`, and the function `libFuncD()` represents a function that is called internally to the library. In this scenario, `main()` and `appFuncA()` represent application code.

The value of the error mode is set to `IPL_ErrModeParent`. The `IPL_ErrModeParent` option produces a more detailed account of the error conditions.

Example 3-1 Error Functions

```
/* application main function */
main() {
  iplSetErrMode(IPL_ErrModeParent);
  appFuncA(5, 45, 1.0);
  if (IPL_ERRCHK("main","compute something")) exit(1);
  return 0;
}
/* application subroutine */
void appFuncA(int order1, int order2, double a) {
  libFuncB(a, order1);
  if (IPL_ERRCHK("appFuncA","compute using order1")) return;
  libFuncB(a, order2);
  if (IPL_ERRCHK("appFuncA","compute using order2")) return;
}
  /* do some more work  */
```

continued ☞

Example 3-1 Error Functions (continued)

```
/* library function */
void libFuncB(double a, int order) {
  float *vec;
  if (order > 31) {
    IPL_ERROR(IPL_StsBadArg, "libFuncB",
    "order must be less than or  equal to 31");
    return;
  }
  if ((vec = libFuncD(a, order)) == NULL) {
    IPL_ERRCHK("libFuncB", "compute using a");
    return;
  }
/* code to do some real work goes here */
  free(vec);
}            // next: library function called internally
double *libFuncD(double a, int order) {
  double *vec;
  if ((vec=(double*)malloc(order*sizeof(double))) == NULL) {
    IPL_ERROR(IPL_StsNoMem, "libFuncD",
    "allocating a vector of doubles");
    return NULL;
  }
  /* do something with vec */
return vec;
}
```

When the program is run, it produces the output illustrated in Example 3-2.

Example 3-2 Output for the Error Function Program (IPL_ErrModeParent)

```
IPL Library Error: Invalid argument in function libFuncB: order must
be less than or equal to 31

      called from function appFuncA: compute using order2

      called from function main: compute something
```

If the program runs with the `IPL_ErrModeLeaf` option instead of `IPL_ErrModeParent`, only the first line of the above output is produced before the program terminated.

If the program in Example 3-1 runs out of heap memory while using the `IPL_ErrModeParent` option, then the output illustrated in Example 3-3 is produced.

Example 3-3 Output for the Error Function Program (IPL_ErrModeParent)

```
IPL Library Error: Out of memory in function libFuncD: allocating a
vector of doubles

      called from function libFuncB: compute using a

      called from function appFuncA: compute using order1

      called from function main[]: compute something
```

Again, if the program is run with the `IPL_ErrModeLeaf` option instead of `IPL_ErrModeParent`, only the first line of the output is produced.

Adding Your Own Error Handler

The Image Processing Library allows you to define your own error handler. User-defined error handlers are useful if you want your application to send error messages to a destination other than the standard error output stream. For example, you can choose to send error messages to a dialog box if your application is running under a Windows system or you can choose to send error messages to a special log file.

There are two methods of adding your own error handler. In the first method, you can replace the `iplError()` function or the complete error handling library with your own code. Note that this method can only be used at link time.

In the second method, you can use the `iplRedirectError()` function to replace the error handler at run time. The steps below describe how to create your own error handler and how to use the `iplRedirectError()` function to redirect error reporting.

1. Define a function with the function prototype, `IPLErrCallBack`, as defined by the IPL.
2. Your application should then call the `iplRedirectError()` function to redirect error reporting for your own function. All subsequent calls to `iplError()` will call your own error handler.
3. To redirect the error handling back to the default handler, simply call `iplRedirectError()` with a `NULL` pointer.

Example 3-4 illustrates a user-defined error handler function, `ownError()`, which simply prints an error message constructed from its arguments and exits.

Example 3-4   A Simple Error Handler

```
IPLStatus ownError(IPLStatus status, const char *func,
 const char *context, const char *file, int line);
{
  fprintf(stderr, "IPL Library error: %s, ", iplErrorStr(status));
  fprintf(stderr, "function %s, ", func ? func : "<unknown>");
  if (line > 0) fprintf(stderr, "line %d, ", line);
  if (file != NULL) fprintf(stderr, "file %s, ", file);
  if (context) fprintf(stderr, "context %s\n", context);
  IplSetErrStatus(status);
  exit(1);
}
main () {
  extern IPLErrCallBack ownError;
/* Redirect errors to your own error handler */
  iplRedirectError( ownError);
/* Redirect errors back to the default error handler */
  iplRedirectError(NULL);
}
```

Image Creation and Access

4

This chapter describes the functions that provide the following functionalities:

- Creating and accessing attributes of images (both tiled and non-tiled)
- Allocating memory for data of required type (see also the functions CreateConvKernel in Chapter 6 and CreateColorTwist in Chapter 9)
- Manipulating the image
- Working in the Windows* DIB (device-independent bitmap) environment.

Table 4-1 Image Creation, Data Exchange and Windows DIB Environment Functions

Group	Function Name	Description
Creating Images	`iplCreateImageHeader`	Creates an image header according to the specified attributes.
	`iplAllocateImage`	Allocates memory for image data.
	`iplDeAllocateImage`	Deallocates or frees memory for image data pointed to in the image header.
	`iplCreateROI`	Creates a region of interest (ROI) header with specified attributes.
	`iplDeallocate`	Deallocates header attributes or image data or ROI or all of the above.
	`iplSetROI`	Sets a region of interest for an image.
	`iplSetBorderMode`	Sets the mode for handling the border pixels.
	`iplCreateTileInfo`	Creates the `IplTileInfo` structure.
	`iplSetTileInfo`	Sets the tiling information.
	`iplDeleteTileInfo`	Deletes the `IplTileInfo` structure.

continued ☞

Table 4-1 Image Creation, Data Exchange and Windows DIB Environment Functions (continued**)**

Group	Function Name	Description
Memory Allocation	`iplMalloc`	Allocates memory aligned to 8 bytes boundary.
	`iplwMalloc`	Allocates memory aligned to 8 bytes boundary for 16-bit words.
	`ipliMalloc`	Allocates memory aligned to 8 bytes boundary 32-bit double words.
	`iplsMalloc`	Allocates memory aligned to 8 bytes boundary for single float elements.
	`ipldMalloc`	Allocates memory aligned to 8 bytes boundary for double float elements.
	`iplFree`	Frees memory allocated by the `ipl?Malloc` functions.
Data Exchange	`iplSet`	Sets a value for the image pixel data.
	`iplCopy`	Copies image data from one IPL image to another.
	`iplExchange`	Exchanges image data between two IPL images.
	`iplConvert`	Converts an IPL image based on the input and output image requirements.
Windows DIB	`iplTranslateDIB`	Translates a DIB image into an IPL image.
	`iplConvertFromDIB`	Converts a DIB image to an IPL image with specified attributes.
	`iplConvertToDIB`	Converts an IPL image to a DIB image with specified attributes.

Image Header and Attributes

The IPL library functions operate on a single format for images in memory henceforth called the IPL image format. The IPL image format consists of a header of type `IPLImage` containing the information for all the attributes of the image. The header finally contains a pointer to the image data. (See the attributes description in Chapter 2, section "Data Architecture.") The values that these attributes can assume are listed in Table 4-2.

Table 4-2 IPL Image Header Attributes

Description	Variable or Value	Corresponding DIB Attribute
Size of the IPL image header (for internal use)	`nSize` in bytes	
IPL Image Header Revision ID (internal use)	ID number	
Number of Channels	1 to `N` (including alpha channel, if any)	1 (Gray) 3 (RGB) 4 (RGBA)
Alpha channel number	0 (if not present) n	4 (RGBA)
Bits per channel		
Gray only	`IPL_DEPTH_1U` (1-bit)	Supported
All images: color, gray, and multi-spectral	`IPL_DEPTH_8U` (8-bit unsigned)	Supported (RGB, RGBA)
(The signed data is used only as output for some image output operations.)	`IPL_DEPTH_8S` (8-bit signed) `IPL_DEPTH_16U` (16-bit unsign.) `IPL_DEPTH_16S` (16-bit signed) `IPL_DEPTH_32S` (32-bit signed)	Not supported Not supported Not supported Not supported
Color model	4 character string: "Gray", "RGB," "RGBA", "CMYK," etc.	Not supported. Implicitly, RGB color model.

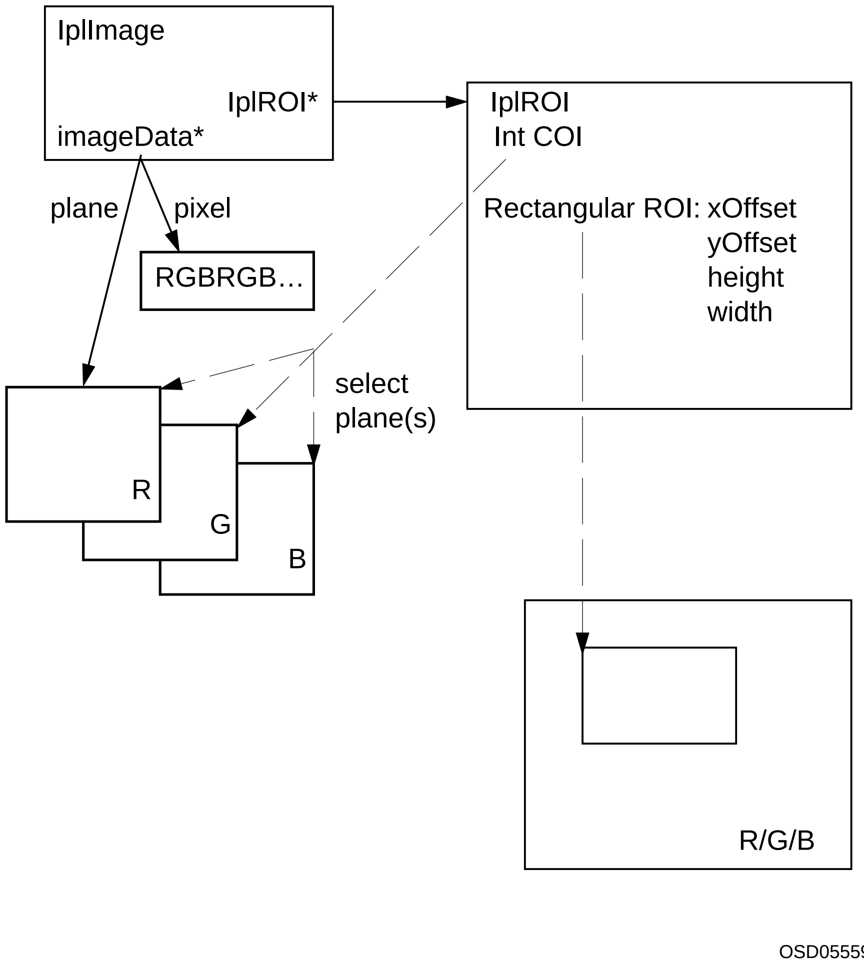
continued ☞

Table 4-2 IPL Image Header Attributes (continued**)**

Description	Value	Corresponding DIB Attribute
Channel sequence	4-character string: “Gray”, “RGB,” “RGBA”, “CMYK,” etc.	Not supported. Always implicitly BGR for RGB images.
Data Ordering	`IPL_DATA_ORDER_PIXEL` `IPL_DATA_ORDER_PLANE`	Supported Not supported
Origin	`IPL_ORIGIN_TL` (top left corner) `IPL_ORIGIN_BL` (bottom left corner)	Supported Supported
Scanline alignment	`IPL_ALIGN_DWORD` `IPL_ALIGN_QWORD`	Supported Not Supported
Image dimensions		
Image Height	m	m
Image Width	n	n
Region of interest (ROI)	Pointer to structure	Not supported
Image size (bytes)	Integer	
Image data pointer	Pointer to data	
Aligned width	Width (row length) in bytes of aligned image	
Border mode of the top, bottom, left, and right sides of the image.	BorderMode [4]	
Border constant on the top, bottom, left, and right side of the image.	BorderConst [4]	
Original Image	Pointer to original image data	
Image ID for tiling	For application use. Ignored by IPL	
Tiling information	Describes tiles for IPL	

Figure 4-1 presents a graphical depiction of an RGB IPL image with a rectangular ROI and a COI.

Figure 4-1 RGB Image with a Rectangular ROI and a COI



Example 4-1 presents a C language definition for the `IPLImage` structure.

Example 4-1 IPLImage Definition

```
typedef struct _IplImage {
                                IPL.H
   int       nSize              /* size of iplImage struct   */
   int       ID                 /* image header version      */
   int       nChannels;
   int       alphaChannel;
   int       depth;             /* pixel depth in bits       */
   char      colorModel[4];
   char      channelSeq[4];
   int       dataOrder;
   int       origin;
   int       align;             /* 4- or 8-byte align        */
   int       height;
   int       width;
   struct    _IplROI *roi;      /* pointer to ROI if any     */
   void      *imageId;          /* use of the application   */
   struct    _IplTileInfo *tileInfo;
                           /* contains information on tiling */
   int       imageSize;         /* useful size in bytes      */
   char      *imageData;        /* pointer to aligned
                                  image                      */
   int       widthStep;          /* size of aligned line in
                                 bytes                       */
   int       BorderMode[4];     /*                           */
   int       BorderConst[4];    /*                           */
   char      *imageDataOrigin; /* ptr to full, nonaligned
                                image                        */
} IplImage;
```

Tiling Fields in the IplImage Structure

Image tiling in the IPL was briefly described in Section 2.

The following fields serve for tiling purposes in the `IplImage` structure:

```
struct IplImage {
   ...
   void* imageId;
   IplTileInfo *tileInfo;
   ...
}
```

The `imageId` field can be used by the application, and is ignored by the library. The `tileInfo` field contains information on tiling. It is described in the next section.

The library expects either the `tileInfo` pointer or the `imageData` pointer to be `NULL`. If the former is `NULL`, the image is not tiled; if the latter is `NULL`, the image is tiled. It is an error condition if both or neither of the two are `NULL`.

IplTileInfo Structure

This structure provides information for image tiling:

```
typedef struct _IplTileInfo
 {
   IplCallBack callBack;
   void *id;
   char* tileData
   int  width, height;
 } IplTileInfo;
```

Here `callBack` is the call-back function (see "Call-backs" in Chapter 2); `id` is an additional identification field; `width` and `height` are the tile sizes for the image.

Creating Images

The following are the ways to create an IPL image:

- Construct an IPL image header by setting the attributes to appropriate values, then call the function `iplAllocateImage()` to allocate memory for the image or set the image data pointer to image data (in a compatible format) that already exists.
- Call `iplCreateImageHeader()` to create an IPL image header, then call the function `iplAllocateImage()` to allocate memory for the image or set the image data pointer to image data (in a compatible format) that already exists.
- Convert a DIB image to an IPL image using the functions `iplTranslateDIB()` or `iplConvertFromDIB()`. See the section "Working in the Windows DIB Environment."

CreateImageHeader

Creates an IPL image header according to the specified attributes.

```
IplImage* iplCreateImageHeader(int nChannels,
int alphaChannel, int depth, char* colorModel,
char* channelSeq, int dataOrder, int origin, int align,
int height, int width, IplROI* roi);
```

nChannels	Number of channels in the image.
alphaChannel	Alpha channel number (0 if no alpha channel in the image).
depth	Bit depth of pixels. Can be one of `IPL_DEPTH_1U`, `IPL_DEPTH_8U`, `IPL_DEPTH_8S`, `IPL_DEPTH_16U`, `IPL_DEPTH_16S`, or `IPL_DEPTH_32S`. See Table 4-2.

`colorModel`	A four-character string describing the color model: “RGB”, “GRAY”, “MSI” etc.
`channelSeq`	The sequence of channels in the image; for example, “BGR” for an RGB image.
`dataOrder`	`IPL_DATA_ORDER_PIXEL` or `IPL_DATA_ORDER_PLANE`.
`origin`	The origin of the image. Can be `IPL_ORIGIN_TL` or `IPL_ORIGIN_BL`.
`align`	Alignment of image data. Can be `IPL_ALIGN_DWORD` or `IPL_ALIGN_QWORD`.
`height`	Height of the image in pixels.
`width`	Width of the image in pixels.
`roi`	Pointer to an ROI (region of interest) structure. This argument can be `NULL`, which implies that a region of interest comprises all channels and the entire image area.

Discussion

The function `iplCreateImageHeader()` creates an IPL image header according to the specified attributes. The image data pointer is set to `NULL`; no memory for image data is allocated. To allocate memory for image data, call the function `iplAllocateImage()`. The image size attribute (set by the `iplAllocateImage()` function) in the header is set to zero.

Return Value

The newly constructed IPL image header.

AllocateImage

Allocates memory for image data according to the specified header.

```
void iplAllocateImage(IplImage* image, int fillValue)
    /* */
```

image	An IPL image header with a `NULL` image data pointer. The pointer will be set to newly allocated image data memory after calling this function.
fillValue	The initial value to use for pixel data. Use a value of xFFFFFFFF (hexadecimal) not to initialize the pixel data.

Discussion

The function `iplAllocateImage()` is used to allocate image data on the basis of a specified image header. The header must be properly constructed before calling this function. Memory is allocated for the image data according to the attributes specified in the image header (see Example 4-1).

The image data pointer will then point to the allocated memory. It is highly preferable, for efficiency considerations, that the scanline alignment attribute (argument *align*) in the image header be set to `IPL_ALIGN_QWORD`. This will force the image data to be aligned on a quadword (64-bit) memory boundary.

This function sets the image size attribute in the header to the number of bytes allocated for the image.

DeallocateImage

Deallocates (frees) memory for image data pointed to in the image header.

```
void iplDeallocateImage(IplImage* image)
```

image	An IPL image header with a pointer to the allocated image data memory. The image data pointer will be set to `NULL` after this function executes.

Discussion

The function `iplDeallocateImage()` is used to free image data memory pointed to by the *imageData* member of the image header. The respective pointer to image data or ROI data is set to `NULL` after the memory is freed up.

Deallocate

Deallocates or frees memory for image header or data or region of interest or all three.

```
void iplDeallocate (IplImage* image, int flag)
```

image	An IPL image header with a pointer to allocated image data memory. The image data pointer will be set to `NULL` after this function executes.
flag	Flag indicating what memory area to free:
	`IPL_IMAGE_HEADER` Free header structure.

`IPL_IMAGE_IMAGE`	Free image data, set pointer to `NULL`.
`IPL_IMAGE_ROI`	Free image ROI, set pointer to `NULL`.
`IPL_IMAGE_ALL`	Free header, image data, and ROI.

Discussion

The function `iplDeallocate()` is used to free or destroy memory allocated for header structure, image data, ROI data, or all three. The respective pointer is set to `NULL` after the memory is freed up.

Setting Regions of Interest

To set a region of interest, the function `iplSetROI()` uses a ROI structure `IplROI` presented in Example 4-2. The `IplROI` member of the IPL image header must point to this `IplROI` structure to be effective. This can be done by a simple assignment. The application may choose to construct the ROI structure explicitly without the use of the function.

Example 4-2 IplROI Definition

```
typedef struct _IplROI {
  unsigned int coi;
                 //Channel to effect in original image
  int xOffset;
  int yOffset;
  int height;
  int width;
 } IplROI;
```

The members in the above `IplROI` structure define:

`coi`	The channel of interest number. This parameter indicates which channel in the original image will be affected by processing taking place in the region of interest; *`coi`* equal to 0 indicates that all channels will be affected.
`xOffset` and *`yOffset`*	The offset from the origin of the rectangular ROI. (See section "Image Regions" in Chapter 2 for the description of image regions.)
`height` and *`width`*	The size of the rectangular ROI.

CreateROI

Allocates and sets the region of interest (ROI) structure.

```
IplROI* iplCreateROI(int coi, int xOffset, int yOffset,
int height, int width);
```

`coi`	The channel of interest. It can be set to 0 (for all channels) or to a specific channel number.
`xOffset, yOffset`	The offsets from the origin of the rectangular region.
`height, width`	The size of the rectangular region.

Discussion

The function `iplCreateROI()` allocates a new ROI structure with the specified attributes and returns a pointer to this structure. If the IPL image pointer is `NULL`, then only a rectangular ROI is defined.

Return Value

A pointer to the newly constructed ROI structure.

SetROI

Sets the region of interest (ROI) structure.

```
void iplSetROI(IplROI* roi, int coi, int xOffset, int
yOffset, int height, int width);
```

`roi`	The pointer to the ROI structure to modify in the original image.
`coi`	The channel of interest in the original image. It can be set to 0 (for all channels) or to a specific channel number.
`xOffset, yOffset`	The offset from the origin of the rectangular region.
`height, width`	The size of the rectangular region.

Discussion

The function `iplSetROI()` sets the channel of interest and the rectangular region of interest in the structure `roi`.

The argument `coi` defines the number of the channel of interest. The arguments `xOffset` and `yOffset` define the offset from the origin of the rectangular ROI. The members `height` and `width` define the size of the rectangular ROI.

Image Borders and Image Tiling

Many neighborhood operators need intensity values for pixels that lie outside the image, that is, outside the borders of the image. For example, a 3 by 3 filter, when operating on the first row of an image, needs to assume pixel values of the preceding (non-existent) row. A larger filter will require more rows from the border. These border issues therefore exist at the top and bottom, left and right sides, and the four corners of the image. The library provides a function `iplSetBorderMode` that the application can use to set the border mode within the image. This function specifies the behavior for handling border pixels.

For tiled images, the border mode is handled in the same way as for non-tiled images, except that in the outer tiles there might be extra data which is ignored.

SetBorderMode

Sets the mode for handling the border pixels.

```
void iplSetBorderMode(IplImage *src, int mode,
                      int border, int constVal)
```

src	The image where the border mode is to be set.
mode	The following modes are supported:

`IPL_BORDER_CONSTANT`	The value `constVal` is used for all pixels.
`IPL_BORDER_REPLICATE`	The last row or column is replicated for the border.
`IPL_BORDER_REFLECT`	The last *n* rows or columns are reflected in reverse order to create the border.

`IPL_BORDER_WRAP`	The required border rows or columns are taken from the opposite side of the image.

`border`	The side that this function is called for. Can be an OR of one or more of the following four sides of an image:

`IPL_SIDE_TOP`	Top side.
`IPL_SIDE_BOTTTOM`	Bottom side.
`IPL_SIDE_LEFT`	Left side.
`IPL_SIDE_RIGHT`	Right side.

If no mode has been set for a side, the default `IPL_BORDER_CONSTANT` is assumed with a value of 0 for `constVal`. The top side is also used to define all border pixels in the top left and right corners. Similarly, the bottom side is used to define the border pixels in the bottom left and right corners.

`constVal`	The value to use for the border when the mode is set to `IPL_BORDER_CONSTANT`.

Discussion

The function `iplSetBorderMode()` is used to set the border handling mode of one or more of the four sides of an image. If the mode is not set for any side, then a constant value of 0 is used for all border pixels on that side. Intensity values for the border pixels are assumed or created based on the mode.

CreateTileInfo

Creates the IplTileInfo structure.

```
IplTileInfo* iplCreateTileInfo(IplCallBack callBack,
void* id, int width, int height);
```

callBack	The call-back function.
id	The image ID (for application use).
width, height	The tile sizes.

Discussion

The function `iplCreateTileInfo()` allocates a new `IplTileInfo` structure with the specified attributes and returns a pointer to this structure.

Return Value

The pointer to the created `IplTileInfo` structure.

SetTileInfo

Sets the IplTileInfo structure fields.

```
void iplSetTileInfo(IplTileInfo* tileInfo, IplCallBack
callBack, int width, int height);
```

tileInfo	The pointer to the `IplTileInfo` structure.
callBack	The call-back function.
id	The image ID (for application use).
width, height	The tile sizes.

Discussion

This function sets attributes for an existing `IplTileInfo` structure.

DeleteTileInfo

Deletes the IplTileInfo structure.

```
void iplDeleteTileInfo(IplTileInfo* tileInfo);
```

tileInfo	The pointer to the `IplTileInfo` structure.

Discussion

This function deletes the `IplTileInfo` structure previously created by the CreateTileInfo function.

Memory Allocation Functions

Functions of the `ipl?Malloc()` group allocate aligned memory blocks for IPL image data. The size of allocated memory is specified by the *`size`* parameter. The “`?`” in `ipl?Malloc()` stands for `w`, `i`, `s`, or `d`; these letters indicate the data type in the function names as follows:

`iplMalloc()`	byte
`iplwMalloc()`	16-bit word
`ipliMalloc()`	32-bit double word
`iplsMalloc()`	4-byte single floating-point element
`ipldMalloc()`	8-byte double floating-point element

NOTE. *The only function to free the memory allocated by any of these functions is* `iplFree()`.

Malloc

Allocates memory aligned to 8 bytes boundary.

```
void* iplMalloc(int size);
```

`size`	Size (in bytes) of memory block to allocate.

Discussion

The `iplMalloc()` function allocates memory block aligned to 8 bytes boundary.

Return Value

The return value of `iplMalloc()` is a pointer to aligned memory block. To free this block, only the function `iplFree()` must be used. If no memory is available in the system, then the `NULL` value is returned.

wMalloc

Allocates memory aligned to 8 bytes boundary for 16-bit words.

```
short* iplwMalloc(int size);
```

`size`	Size in words (16 bits) of memory block to allocate.

Discussion

The `iplwMalloc()` function allocates memory block aligned to 8 bytes boundary for 16-bit words.

Return Value

The return value of `iplwMalloc()` is a pointer to aligned memory block. To free this block only the function `iplFree()` must be used. If no memory is available in the system, then the `NULL` value is returned.

iMalloc

Allocates memory aligned to 8 bytes boundary for 32-bit double words.

```
int* ipliMalloc(int size);
```

size	Size in double words (32 bits) of memory block to allocate.

Discussion

The `ipliMalloc()` function allocates memory block aligned to 8 bytes boundary for 32-bit double words.

Return Value

The return value of `iplMalloc()` is a pointer to aligned memory block. To free this block only the function `iplFree()` must be used. If no memory available in the system, then the `NULL` value is returned.

sMalloc

Allocates memory aligned to 8 bytes boundary for floating-point elements.

```
float * iplsMalloc(int size);
```

size	Size in float elements (4 bytes) of memory block to allocate.

Discussion

The `iplsMalloc()` function allocates memory block aligned to 8 bytes boundary for floating-point elements.

Return Value

The return value of `iplsMalloc()` is a pointer to aligned memory block. To free this block only the function `iplFree()` must be used. If no memory is available in the system, then the `NULL` value is returned.

dMalloc

Allocates memory aligned to 8 bytes boundary for double floating-point elements.

```
double* ipldMalloc(int size);
```

size	Size in double elements (8 bytes) of memory block to allocate.

Discussion

The `ipldMalloc()` function allocates memory block aligned to 8 bytes boundary for double floating-point elements.

Return Value

The return value of `ipldMalloc()` is a pointer to aligned memory block. To free this block only the function `iplFree()` must be used. If no memory is available in the system, then the `NULL` value is returned.

iplFree

Frees memory allocated by one of the `ipl?Malloc` functions.

```
void  iplMalloc(void * ptr);
```

`ptr`	Pointer to memory block to free.

Discussion

The `iplFree()` function frees the aligned memory block allocated by one of the functions `iplMalloc()`, `iplwMalloc()`, `ipliMalloc()`, `iplsMalloc()`, or `ipldMalloc()`.

NOTE. *The function* `iplFree()` *can't be used to free memory allocated by standard functions like* `malloc()` or `calloc()`.

Image Data Exchange

The functions described in this section provide image manipulation capabilities, such as setting the image pixel data, copying data from one image to another, exchanging the data between the images, and converting one IPL image to another according to the attributes defined in the source and resultant image headers.

Set

Sets a value for an IPL image pixel data.

```
void iplSet(IplImage* image, int fillValue,
IplCoord* map);
```

image	An IPL image header with allocated image data.
fillValue	The value to set the pixel data.
map	The structure specifying offsets for tiling purposes.

Discussion

The function `iplSet()` sets an IPL image pixel data. Before calling this function, the IPL image header must be properly constructed and image data must be allocated. For images with the bit depth lower than the *fillVallue*, the *fillValue* is truncated when assigned to pixel. If an ROI is specified, only that ROI is filled.

Copy

Copies image data from one IPL image to another.

```
void iplCopy(IplImage* srcImage, IplImage* dstImage,
              IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplCopy()` copies image data from a source image to a resultant image. Before calling this function, the source and resultant headers must be properly constructed and image data for both images must be allocated. The following constraints apply to the copying:

- The bit depth per channel of the source image should be equal to that of the resultant image.
- The number of channels of interest in the source image should be equal to the number of channels of interest in the resultant image; that is, either the source `coi` = the resultant `coi` = 0 or both cois are nonzero.
- The data ordering (by pixel or by plane) of the source image should be the same as that of the resultant image.

The *`origin`*, *`align`*, *`height`*, and *`width`* field values (see Table 4-2) may differ in source and resultant images. Copying applies to the areas that intersect between the source ROI and the destination ROI.

Exchange

Exchanges image data between two IPL images.

```
void iplExchange(IplImage* ImageA, IplImage* ImageB,
                 IplCoord* map);
```

`ImageA`	The first image.
`ImageB`	The second image.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplExchange()` exchanges image data between two images, the first and the second. The image headers must be properly constructed before calling this function, and image data for both images must be allocated. The following constraints apply to the data exchanging:

- The bit depths per channel of both images should be equal.
- The numbers of channels of interest in both images should be equal.
- The data ordering of both images should be the same (either pixel- or plane-oriented) .

The `origin`, `align`, `height`, and `width` field values (see Table 4-2) may differ in the first and the second images. The data are exchanged at the areas of intersection between the ROI of the first image and the ROI of the second image.

Convert

Converts source IPL image data to resultant IPL image according to the source and resultant image headers.

```
void iplConvert(IplImage* srcImage, IplImage* dstImage,
int convertMode, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`convertMode`	A flag indicating how to perform the image conversion by reducing the bit depth. The flag values are: `IPL_BITS_HIGH` Reduce by scaling; `IPL_BITS_LOW` Reduce by using the lower part of the pixel values
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplConvert()` converts image data from the source IPL image to the resultant IPL image according to the attributes defined in the source and resultant image headers. The images that can be converted may have the following different characteristics:

- Bit depth per channel
- Data ordering
- Origins

For the above data description, see Table 4-2.

The following constraints apply to the conversion:

- If the source image has a bit depth per channel equal to 1, the resultant image should also have the bit depth equal to 1.
- The number of channels in the source image should be equal to the number of channels in the resultant image.
- The height and width of the source image should be equal to that of the resultant image.

All ROIs are ignored; `IplCoord` must be `NULL`.

Working in the Windows DIB Environment

The IPL library provides functions to convert a DIB (device-independent bitmap) image to an IPL image and vice versa. Table 4-2 shows that the DIB image format is a subset of the IPL image format. Not included in this subset are the DIB palette images and DIB 8-bit- and 16-bit-per-pixel absolute color images because they have no equivalent IPL images.

The DIB palette images must be first converted to IPL absolute color images. DIB 8-bit- and 16-bit-per-pixel images have to be unpacked into IPL 8-bit, 16-bit- or 32-bit-per-channel images.

However, any DIB 24-bit absolute color image can be directly converted to an IPL image. You just need to create an IPL image header corresponding to the DIB attributes. The DIB image data can be pointed to by the IPL header or it can be duplicated.

The IPL functions can perform conversion from a DIB image to an IPL image and vice versa with additional useful capabilities:

`iplTranslateDIB()`	Performs a simple translation of a DIB image to an IPL image as described above. Also converts a DIB palette image to an IPL absolute color image. While this is the most efficient way of converting a DIB image, it is not the most efficient format for the IPL functions to manipulate because the DIB image data is doubleword-aligned, and not quadword-aligned.
`iplConvertFromDIB()`	Provides more control of the conversion and can convert a DIB image to an IPL image with a prepared IPL image header. The IPL image header must be then set to the desired attributes. The bit depth of the channels in the IPL image header must be equal to or greater than that in the DIB header.
`iplConvertToDIB()`	Performs conversion in the opposite direction: an IPL image to a DIB image. This function performs dithering if the bit depth of the DIB is less than that of the IPL image. It can also be used to create a DIB palette image from an IPL absolute color image. The function can optionally create a new palette.

TranslateDIB

Translates a DIB image into the corresponding IPL image.

```
iplImage* iplTranslateDIB(BITMAPINFOHEADER* dib,
BOOL cloneData)
```

`dib`	The DIB image.
`cloneData`	An output flag (Boolean): if false, indicates that the image data pointer in the IPL image will point to the DIB image data; if true, indicates that the data was copied.

Discussion

The function `iplTranslateDIB()` translates a DIB image into an IPL image. The IPL image attributes corresponding to the DIB image are automatically chosen (see Table 4-2), so no explicit control of the conversion is provided. A DIB palette image will be converted to an IPL absolute color image with a bit depth of 8 bits per channel, and the image data will be copied, returning *`cloneData`* = true.

A 24-bit-per-pixel DIB RGB image will be converted to an 8-bit-per-channel RGB IPL image.

A 32-bit-per-pixel DIB RGBA image will be converted to an 8-bit-per-channel RGBA IPL image with an alpha channel.

An 8-bit-per-pixel or 16-bit-per-pixel DIB absolute color RGB image will be converted (by unpacking) into an 8-bit-per-channel RGB IPL image. The image data will be copied, returning *`cloneData`* = true.

A 1-bit-per-pixel or 8-bit-per-pixel DIB gray scale image with a standard gray palette will be converted to a 1-bit-per-channel or 8-bit-per-channel IPL gray-scale image, respectively.

A 4-bit-per-pixel DIB gray-scale image with a standard gray palette will be converted into an 8-bit-per-pixel IPL gray-scale image and the image data will be copied, returning `cloneData` = true.

Note that in the cases above where the image data is not copied, it will result in inefficient access of the image by the IPL image processing functions. This is because DIB image data is aligned on doubleword (32-bit) boundaries. Alternatively, when `cloneData` is true, the DIB image data is replicated into newly allocated image data memory and automatically aligned to quadword boundaries which results in a better memory access.

Return Value

A constructed IPL image.

ConvertFromDIB

Converts a DIB image to an IPL image with specified attributes.

```
void iplConvertFromDIB(BITMAPINFOHEADER*  dib,
                       IplImage*  image)
```

`dib`	The input DIB image.
`image`	The IPL image header with specified attributes. If the data pointer is `NULL`, image data memory will be allocated and the pointer set to it.

Discussion

The function `iplConvertFromDIB()` converts a DIB image into an IPL image according to the attributes set in the IPL image header. Explicit control of the conversion is therefore provided. The following constraints apply to the conversion:

- The bit depth per channel of the IPL image should be greater than or equal to that of the DIB image.
- The number of channels (not including the alpha channel) in the IPL image should be greater than or equal to the number of channels in the DIB image (not including the alpha channel if present).
- The dimensions of the IPL image should be greater than or equal to that of the DIB image. When the IPL image is larger than the DIB image, the origins of the IPL and DIB images are made coincident for the purposes of copying.
- When converting a DIB RGBA image, the IPL image should also contain an alpha channel.

ConvertToDIB

Converts an IPL image to a DIB image with specified attributes.

```
void iplConvertToDIB(iplImage* image, BITMAPINFOHEADER*
                 dib, int dither, int paletteConversion)
```

image	The input IPL image.
dib	The output DIB image.
dither	The dithering algorithm to use if applicable. Dithering will be done if the bit depth in the DIB is less than that of the IPL image. The following algorithms are supported corresponding to these *dither* identifiers:

`IPL_DITHER_STUCKEY`	The Stucki dithering algorithm is used.
`IPL_DITHER_NONE`	No dithering is done. The most significant bits in the IPL image pixel data are retained.

paletteConversion	Applicable when the DIB is a palette image. Specifies the palette algorithm to use when converting the IPL absolute color image. The following options are supported:

`IPL_PALCONV_NONE`	The existing palette in the DIB is used.
`IPL_PALCONV_POPULATE`	The popularity palette conversion algorithm is used.
`IPL_PALCONV_MEDCUT`	The median cut algorithm for palette conversion is used.

Discussion

The function `iplConvertToDIB()` converts an IPL image to a DIB image. The conversion takes place according to the IPL image and DIB image attributes. While IPL images are always in absolute color, DIB images can be in absolute or palette color. When the DIB is a palette image, the absolute color IPL image is converted to a palette image according to the palette conversion option specified. When the bit depth of an absolute color DIB image is less than that of the IPL image, then dithering according to the specified option is performed.

The following constraints and considerations apply when using this function:

- The number of channels (not including the alpha and ROI channels) in the IPL image should be equal to the number of channels in the DIB image.
- The alpha channel in an IPL image will be passed on only when the DIB is an RGBA image.

Image Arithmetic and Logical Operations

5

This chapter describes image processing functions that modify pixel values using simple arithmetic or logical operations. It also includes the library functions that perform image compositing based on opacity (alpha-blending). All these operations can be broken into two categories: monadic operations, which use single input images, and dyadic operations, which use two input images. Table 5-1 lists the functions that perform arithmetic and logical operations.

Table 5-1 Image Arithmetic and Logical Operations

Group	Function Name	Description
Arithmetic operations	`iplAddS`	Adds a constant to the image pixel values.
	`iplSubtractS`	Subtracts a constant from the pixel values or the values from a constant.
	`iplMultiplyS`	Multiplies pixel values by a constant.
	`iplMultiplySScale`	Multiplies pixel values by a constant and scales the product.
	`iplSquare`	Squares the pixel values of an image.
	`iplAdd`	Adds pixel values of two images.
	`iplSubtract`	Subtracts pixel values of one image from those of another image.
	`iplMultiply`	Multiplies pixel values of two images.
	`iplMultiplyScale`	Multiplies pixel values of two images and scales the product.

Continued ☞

Table 5-1 Image Arithmetic and Logical Operations (continued)

Group	Function Name	Description
Logical operations	`iplAndS`	Performs a bitwise AND operation on each pixel with a constant.
	`iplOrS`	Performs a bitwise OR operation on each pixel with a constant.
	`iplXorS`	Performs a bitwise XOR operation on each pixel with a constant.
	`iplNot`	Performs a bitwise NOT operation on each pixel
	`iplLShiftS`	Multiplyes pixel values by a constant power of 2 by shifting bits to the left.
	`iplRShiftS`	Divides pixel values by a constant power of 2 by shifting bits to the right.
	`iplAnd`	Combines corresponding pixels of two images by a bitwise AND operation.
	`iplOr`	Combines corresponding pixels of two images by a bitwise OR operation.
	`iplXor`	Combines corresponding pixels of two images by a bitwise XOR operation.
Alpha-blending	`iplPreMultiplyAlpha`	Pre-multiplies pixel values of an image by alpha values.
	`iplAlphaComposite`	Composites two images using alpha (opacity) values.
	`iplAlphaCompositeC`	Composites two images using constant alpha (opacity) values.

The functions `iplSquare()`, `iplNot()`, and `iplPreMultiplyAlpha()` as well as all functions with names containing an additional `S` use single input images (perform monadic operations). All other functions in the above table use two input images (perform dyadic operations).

Monadic Arithmetic Operations

The sections that follow describe the IPL functions that perform monadic arithmetic operations (note that the `iplPreMultiplyAlpha` function is described in the "Image Compositing Based on Opacity" section of this chapter). All these functions use a single input image to create an output image.

AddS

Adds a constant to pixel values of the source image.

```
void iplAddS(IplImage* srcImage, IplImage* dstImage, int
value, IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.
value	The value to be added to the pixel values.
map	The structure specifying offsets for tiling purposes.

Discussion

The function `iplAddS()` changes the image intensity by adding the *value* to pixel values. A positive *value* brightens the image (increases the intensity); a negative *value* darkens the image (decreases the intensity).

SubtractS

Subtracts a constant from pixel values, or pixel values from a constant.

```
void iplSubtractS(IplImage* srcImage, IplImage* dstImage,
int value, BOOL flip, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`value`	The value to be subtracted from the pixel values.
`flip`	A Boolean used to change the order of subtraction.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplSubtractS()` changes the intensity of an image as follows:

If `flip` is false, the `value` is subtracted from the image pixel values. If `flip` is true, the image pixel values are subtracted from the `value`.

MultiplyS

Multiplies pixel values by a constant.

```
void iplMultiplyS(IplImage* srcImage, IplImage* dstImage,
unsigned int value, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.

`value`	A positive value by which to multiply the pixel values.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplMultiplyS()` increases the intensity of an image by multiplying each pixel by a positive constant *`value`*.

MultiplySScale

Multiplies pixel values by a constant and scales the products.

```
void iplMultiplySScale(IplImage* srcImage, IplImage*
dstImage, int value, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`value`	A positive value by which to multiply the pixel values.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplMultiplySScale()` multiplies the input image pixel values by *`value`* and scales the products using the following formula:

`dst_pixel` = *`src_pixel`* * *`value`* / *`max_val`*

where *`src_pixel`* is a pixel value of the source images, *`dst_pixel`* is the resultant pixel value, and *`max_val`* is the maximum presentable pixel value. The source and resultant images must have the same pixel depth.

Square

Squares the pixel values of the image.

```
void iplSquare(IplImage* srcImage, IplImage* dstImage,
IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplSquare()` increases the intensity of an image by squaring each pixel value.

Dyadic Arithmetic Operations

The sections that follow describe the IPL functions that perform dyadic arithmetic operations. These functions use two input images to create an output image.

Add

Combines corresponding pixels of two images by addition.

```
void iplAdd(IplImage* srcImageA, IplImage* srcImageB,
IplImage* dstImage, IplCoord* map);
```

srcImageA	The first source image.
srcImageB	The second source image.
dstImage	The resultant image obtained as *dstImage* = *srcImageA* + *srcImageB*.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplAdd()` adds corresponding pixels of two input images to produce an output image.

Subtract

Combines corresponding pixels of two images by subtraction.

```
void iplSubtract(IplImage* srcImageA, IplImage*
srcImageB, IplImage* dstImage, BOOL flip, IplCoord* map);
```

`srcImageA`	The first source image.
`srcImageB`	The second source image.
`dstImage`	If `flip` (see below) is false, the resultant image is `dstImage = srcImageA - srcImageB`, otherwise it is `dstImage = srcImageB - srcImageA`.
`flip`	A Boolean flag to indicate the order in which the input images are subtracted. See `dstImage` above.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplSubtract()` subtracts corresponding pixels of two input images to produce an output image.

Multiply

Combines corresponding pixels of two images by multiplication.

```
void iplMultiply(IplImage* srcImageA, IplImage*
srcImageB, IplImage* dstImage, IplCoord* map);
```

srcImageA	The first source image.
srcImageB	The second source image.
dstImage	The resultant image.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplMultiply()` multiplies corresponding pixels of two input images to produce an output image.

MultiplyScale

Multiplies pixel values of two images and scales the products.

```
void iplMultiplyScale(IplImage* srcImageA, IplImage*
srcImageB, IplImage* dstImage, IplCoord* map);
```

`srcImageA`	The first source image.
`srcImageB`	The second source image.
`dstImage`	The resultant image.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplMultiplyScale()` multiplies corresponding pixels of two input images and scales the products using the following formula:

`dst_pixel` = *`srcA_pixel`* * *`srcB_pixel`* / *`max_val`*

where *`srcA_pixel`* and *`srcB_pixel`* are pixel values of the source images, *`dst_pixel`* is the resultant pixel value, and *`max_val`* is the maximum presentable pixel value. Both source images and the resultant image must have the same pixel depth.

Monadic Logical Operations

The sections that follow describe the IPL functions that perform monadic logical operations. All these functions use a single input image to create an output image.

LShiftS

Shifts pixel values' bits to the left.

```
void iplLShiftS(IplImage* srcImage, IplImage* dstImage,
unsigned int nShift, IplCoord* map);
```

`srcImage`	Thesource image.
`dstImage`	The resultant image.
`nShift`	The number of bits by which to shift each pixel value to the left.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplLShiftS()` changes the intensity of the source image by shifting the bits in each pixel value by *`nShift`* bits to the left. The positions vacated after shifting the bits are filled with zeros.

RShiftS

Divides pixel values by a constant power of 2 by shifting bits to the right.

```
void iplRShiftS(IplImage* srcImage, IplImage* dstImage,
unsigned int nShift, IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.
nShift	The number of bits by which to shift each pixel value to the right.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplRShiftS()` decreases the intensity of the source image by shifting the bits in each pixel value by *nShift* bits. The positions vacated after shifting the bits are filled with zeros.

Not

Performs a bitwise NOT operation on each pixel.

```
void iplNot(IplImage* srcImage, IplImage* dstImage,
IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.

`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplNot()` performs a bitwise NOT operation on each pixel value.

AndS

Performs a bitwise AND operation of each pixel with a constant.

```
void iplAndS(IplImage* srcImage, IplImage* dstImage,
unsigned int value, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`value`	The bit sequence used to perform the bitwise AND operation on each pixel.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplAndS()` performs a bitwise AND operation between each pixel value and *`value`*. The least significant bit(s) of the *`value`* are used.

OrS

Performs a bitwise OR operation of each pixel with a constant.

```
void iplOrS(IplImage* srcImage, IplImage* dstImage,
unsigned int value, IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.
value	The bit sequence used to perform the bitwise OR operation on each pixel.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplOrS()` performs a bitwise OR between each pixel value and *value*. The least significant bit(s) of the *value* are used.

XorS

Performs a bitwise XOR operation of each pixel with a constant.

```
void iplXorS(IplImage* srcImage, IplImage* dstImage,
unsigned int value, IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.
value	The bit sequence used to perform the bitwise XOR operation on each pixel.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplXorS()` performs a bitwise XOR between each pixel value and *value*. The least significant bit(s) of the *value* are used.

Dyadic Logical Operations

This section describes the IPL functions that perform dyadic logical operations. These functions use two input images to create an output image.

And

Combines corresponding pixels of two images by a bitwise AND operation.

```
void iplAnd(IplImage* srcImageA, IplImage* srcImageB,
IplImage* dstImage, IplCoord* map);
```

srcImageA	The first source image.
srcImageB	The second source image.
dstImage	The image resulting from the bitwise operation between input images *srcImageA* and *srcImageB*.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplAnd()` performs a bitwise AND operation between the values of corresponding pixels of two input images.

Or

Combines corresponding pixels of two images by a bitwise OR operation.

```
void iplOr(IplImage* srcImageA, IplImage* srcImageB,
IplImage* dstImage, IplCoord* map);
```

srcImageA	The first source image.
srcImageB	The second source image.

dstImage	The image resulting from the bitwise operation between input images *srcImageA* and *srcImageB*.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplOR()` performs a bitwise OR operation between the values of corresponding pixels of two input images.

Xor

Combines corresponding pixels of two images by a bitwise XOR operation.

```
void iplXor(IplImage* srcImageA, IplImage* srcImageB,
IplImage* dstImage, IplCoord* map);
```

srcImageA	The first source image.
srcImageB	The second source image.
dstImage	The image resulting from the bitwise operation between input images *srcImageA* and *srcImageB*.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplXor()` performs a bitwise XOR operation between the values of corresponding pixels of two input images.

Image Compositing Based on Opacity

The IPL library provides functions to composite two images using either the opacity (alpha) channel in the images or a provided alpha value. Alpha values range from 0 (100% translucent, 0% coverage) to full range (0% translucent, 100% coverage). Coverage is the percentage of the pixel's own intensity that is visible.

Using the opacity channel for image compositing provides the capability of overlaying the arbitrarily shaped and transparent images in arbitrary positions. It also reduces aliasing effects along the edges of the combined regions by allowing some of the bottom image's color to show through.

Let us consider the example of RGBA images. Here each pixel is a quadruple (r, g, b, α) where r, g, b, and α are the red, green, blue and alpha channels, respectively. In the formulas that follow, the Greek letter α with subscripts always denotes the normalized (scaled) alpha value in the range 0 to 1, no matter what the full range of the alpha channel value.

There are many ways of combining images using alpha values. In all compositing operations a resultant pixel $(r_C, g_C, b_C, \alpha_C)$ in image C is created by overlaying a pixel $(r_A, g_A, b_A, \alpha_A)$ from the foreground image A over a pixel $(r_B, g_B, b_B, \alpha_B)$ from the background image B. The resulting pixel values for an OVER operation (A OVER B) are computed as shown below.

$$r_C = \alpha_A * r_A + (1 - \alpha_A) * \alpha_B * r_B$$

$$g_C = \alpha_A * g_A + (1 - \alpha_A) * \alpha_B * g_B$$

$$b_C = \alpha_A * b_A + (1 - \alpha_A) * \alpha_B * b_B$$

The above three expressions can be condensed into one as follows:

$$C = \alpha_A * A + (1 - \alpha_A) * \alpha_B * B$$

In this example, the color of the background image B influences the color of the resultant image through the second term $(1 - \alpha_A) * \alpha_B * B$. The resulting alpha value is computed as

$$a_C = \alpha_A + (1 - \alpha_A) * \alpha_B$$

Using Pre-multiplied Alpha Values

In many cases it is computationally more efficient to store the color channels pre-multiplied by the alpha values. In the RGBA example, the pixel (r, g, b, α) would actually be stored as (r*α, g*α, b*α, α). This storage format reduces the number of multiplications required in the compositing operations. In interactive environments, when an image is composited many times, this capability is especially efficient.

One known disadvantage of the pre-multiplication is that once a pixel is marked as transparent, its color value is gone because the pixel's color channels are multiplied by 0.

The function `iplPreMultiplyAlpha()` implements various alpha compositing operations between two images. One of them is converting the pixel values to pre-multiplied form.

The color channels in images with the alpha channel can be optionally pre-multiplied with the alpha value. This saves a significant amount of computation for some of the alpha compositing operations. For example, in an RGBA color model image, if (r, g, b, α) are the channel values for a pixel, then upon pre-multiplication they are stored as (r*α, g*α, b*α, α).

AlphaComposite AlphaCompositeC

Composite two images using alpha (opacity) values.

```
void iplAlphaComposite(IplImage* srcImageA, IplImage*
srcImageB, IplImage* dstImage, int compositeType,
IplImage* alphaImageA, IplImage* alphaImageB, IplImage*
alphaImageDst, BOOL premulAlpha, BOOL divideMode,
IplCoord* map);
```

```
void iplAlphaCompositeC(IplImage* srcImageA, IplImage*
srcImageB, IplImage* dstImage, int compositeType, int aA,
int aB, BOOL premulAlpha, BOOL divideMode, IplCoord* map);
```

`srcImageA`	The foreground input image.
`srcImageB`	The background input image.
`dstImage`	The resultant output image.
`compositeType`	The composition type to perform. See Table 5-2 for the type value and description.
`aA`	The constant alpha value to use for the source image `srcImageA`. Should be a positive number.
`aB`	The constant alpha value to use for the source image `srcImageB`. Should be a positive number.
`alphaImageA`	The image to use as the alpha channel for `srcImageA`. If the image contains an alpha channel, that channel is used. Otherwise channel 1 in the image is used as the alpha channel. If this is not suitable for the application, then the alpha channel number in the IPL header for the image should be set appropriately before calling this function. If the argument `alphaImageA` is `NULL`, then the internal alpha channel of `srcImageA` is used. If `srcImageA` does not contain an alpha channel, an error message is issued.
`alphaImageB`	The image to use as the alpha channel for `srcImageB`. If the image already contains an alpha channel, that channel is used. Otherwise channel 1 in the image is used as the alpha channel. If this is not suitable for the application, then the alpha channel number in the IPL header for the image should be set appropriately before calling this function. If the argument `alphaImageB` is `NULL`, then the internal alpha channel of `srcImageB` is used.

	If `srcImageB` does not contain an alpha channel, then the value $(1 - \alpha_A)$ is used for the alpha, where α_A is a scaled alpha value of `srcImageA` in the range 0 to 1.
`alphaImageDst`	The image to use as the alpha channel for `dstImage`. If the image already contains an alpha channel, that channel is used. Otherwise channel 1 in the image is used as the alpha channel. If this is not suitable for the application, then the alpha channel number in the IPL header for the image should be set appropriately before calling this function. This argument can be `NULL`, in which case the resultant alpha values are not saved.
`premulAlpha`	A Boolean flag indicating whether or not the input images contain pre-multiplied alpha values. If true, they contain these values.
`divideMode`	A Boolean flag set to false by default. When true, the resultant pixel color (see Table 5-2) is further divided by the resultant alpha value to get the final resultant pixel color.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplAlphaComposite()` performs an image compositing operation by overlaying the foreground image `srcImageA` with the background image `srcImageB` to produce the resultant image `dstImage`.

The function `iplAlphaComposite()` executes under one of the following conditions for the alpha channels:

- If `alphaImageA` and `alphaImageB` are both `NULL`, then the internal alpha channels of the two input images specified by their respective IPL image headers are used. The application has to ensure that these are set to the proper channel number prior to calling this function. If `srcImageB` does not have an alpha channel, then its alpha value is set to $(1 - \alpha_A)$ where α_A is the scaled alpha value of image `srcImageA` in the range 0 to 1.
- If both alpha images `alphaImageA` and `alphaImageB` are not `NULL`, then they are used as the alpha values for the two input images. If `alphaImageB` is `NULL`, then its alpha value is set to $(1 - \alpha_A)$ where α_A is the scaled alpha value of image `alphaImageA` in the range 0 to 1.

It is an error if none of the above conditions is satisfied.

If `alphaImageDst` is not `NULL`, then the resultant alpha values are written to it. If it is `NULL` and the output image `imageDst` contains an alpha channel (specified by its IPL image header), then it is set to the resulting alpha values. Otherwise (that is, alpha channel number is zero), the output pixel values are multiplied by the resulting alpha values before final storage in the output image occurs.

The function `iplAlphaCompositeC()` is used to specify constant alpha values α_A and α_B to be used for the two input images (usually α_B is set to the value $1 - \alpha_A$). The output pixel values are multiplied by the resultant alpha values before final storage in the output image. The resultant alpha values (also constant) are not saved.

The type of compositing is specified by the argument `compositeType` which can assume the values shown in Table 5-2.

Table 5-2 Types of Image Compositing Operations

Type	Output Pixel (see Note)	Output Pixel (pre-mult. α)	Resultant Alpha	Description
OVER	α_A*A+ (1- α_A)*α_B*B	A+(1-α_A)*B	α_A+ (1- α_A)* α_B	A occludes B
IN	α_A*A* α_B	A*α_B	α_A* α_B	A within B. A acts as a matte for B. A shows only where B is visible.
OUT	α_A*A*(1- α_B)	A*(1- α_B)	α_A *(1- α_B)	A outside B. NOT-B acts as a matte for A. A shows only where B is not visible.
ATOP	α_A*A* α_B+ (1- α_A)*α_B*B	A* α_B+ (1- α_A)*B	α_A* α_B+ (1- α_A)* α_B	Combination of (A IN B) and (B OUT A). B is both back-ground and matte for A.
XOR	α_A*A*(1-α_B)+ (1- α_A)* α_B*B	A*(1- α_B)+ (1- α_A)*B	α_A*(1- α_B)+ (1- α_A)* α_B	Combination of (A OUT B) and (B OUT A). A and B mutually exclude each other.
PLUS	α_A *A + α_B*B	A + B	α_A + α_B	Blend without precedence

NOTE. *In Table 5-2, the resultant pixel value is divided by the resultant alpha when* `divideMode` *is set to true (see the argument descriptions for the* `iplAlphaComposite()` *function). The Greek letter* α *here and below denotes normalized (scaled) alpha values in the range 0 to 1.*

For example, for the OVER operation, the output C for each pixel in the inputs A and B is determined as

$$C = \alpha_A * A + (1 - \alpha_A) * \alpha_B * B$$

The above operation is done for each color channel in A, B, and C. When the images A and B contain pre-multiplied alpha values, C is determined as

$$C = A + (1 - \alpha_A) * B$$

The resultant alpha value `aC` (alpha in the resultant image C) is computed as (both pre-multiplied and not pre-multiplied alpha cases) from `aA` (alpha in the source image A) and `aB` (alpha in the source image B):

$$\alpha_C = \alpha_A + (1 - \alpha_A) * \alpha_B$$

Thus, to perform an OVER operation, use the `IPL_COMPOSITE_OVER` identifier for the argument `compositeType`. For all other types, use `IPL_COMPOSITE_IN`, `IPL_COMPOSITE_OUT`, `IPL_COMPOSITE_ATOP`, `IPL_COMPOSITE_XOR`, and `IPL_COMPOSITE_PLUS`, respectively.

The argument `divideMode` is typically set to false to give adequate results as shown in the above example for an OVER operation and in Table 5-2. When `divideMode` is set to true, the resultant pixel color is divided by the resultant alpha value. This gives an accurate result pixel value, but the division operation is expensive. In terms of the OVER example without pre-multiplication, the final value of the pixel C is computed as

$$C = (\alpha_A * A + (1 - \alpha_A) * \alpha_B * B)/\alpha_C$$

There is no change in the value of α_C, and it is computed as shown above. When both A and B are 100% transparent (that is, α_A is zero and α_B is zero), α_C is also zero and the result cannot be determined. In many cases, the value of α_C is 1, so the division has no effect.

PreMultiplyAlpha

Pre-multiplies alpha values of an image.

```
void iplPreMultiplyAlpha (IplImage* image,
int alphaValue, IplCoord* map );
```

`image`	The image for which the alpha pre-multiplication is performed.
`alphaValue`	The global alpha value to use in the range 0 to 256. If this value is negative (for example, -1), the internal alpha channel of the image is used. It is an error condition if an alpha channel does not exist.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplPreMultiplyAlpha()` converts an image to the pre-multiplied alpha form. If (R, G, B, A) are the red, green, blue, and alpha values of a pixel, then the pixel is stored as (R*α, G*α, B*α, A) after execution of this function. Here α is the pixel's normalized alpha value in the range 0 to 1.

Optionally, a global alpha value `alphaValue` can be used for the entire image. Then the pixels are stored as (R*α, G*α, B*α, `alphaValue`) if the image has an alpha channel or (R*α, G*α, B*α) if the image does not have an alpha channel. Here α is the normalized `alphaValue` in the range 0 to 1.

Image Filtering 6

This chapter describes filtering operations that can be applied to images. IPL uses linear and non-linear filters. The linear filters include a subgroup of 2D convolution filters. Table 6-1 lists IPL image filtering functions.

Table 6-1 Image Filtering Functions

Group	Function Name	Description
Linear Filters	`iplBlur`	Applies a simple neighborhood averaging filter to blur the image.
2D Convolution Linear Filters	`iplCreateConvKernel`	Creates a convolution kernel.
	`iplGetConvKernel`	Reads the attributes of a convolution kernel.
	`iplDeleteConvKernel`	Deallocates a convolution kernel.
	`iplConvolve2D`	Convolves an image with one or more convolution kernels.
	`iplConvolveSep2D`	Convolves an image with a separable convolution kernel.
Non-linear	`iplMedianFilter`	Applies a median filter to the image.
Filters	`iplMaxFilter`	Applies a maximum filter to the image.
	`iplMinFilter`	Applies a minimum filter to the image.

Linear Filters

Linear filtering includes simple neighborhood averaging filter to blur the image and 2D convolution operations.

Blur

Applies simple neighborhood averaging filter to blur the image.

```
void iplBlur(IplImage* srcImage, IplImage* dstImage,
int nRows, int nCols, int anchorX, int anchorY, IplCoord*
map);
```

srcImage	The source image.
dstImage	The resultant image.
nRows	Number of rows in the neighborhood to use.
nCols	Number of columns in the neighborhood to use.
anchorX, anchorY	The [x, y] coordinates of the anchor cell in the neighborhood. In this coordinate system, the top left corner would be [0, 0] and the bottom right corner would be [*nRows*-1, *nCols*-1]. For a 3 by 3 neighborhood, the coordinates of the geometric center would be [1, 1]. This specification allows the neighborhood to be skewed with respect to its geometric center.
map	The structure specifying offsets for tiling purposes.

Discussion

The function `iplBlur()` sets each pixel in the output image as the average of all the input image pixels in the neighborhood of size *nRows* by *nCols* with the anchor cell at that pixel. This has the effect of smoothing or blurring the input image. The linear averaging filter of an image is also called a box filter.

2D Convolution

The 2D convolution is a versatile image processing primitive which can be used in a variety of image processing operations; for example, edge detection, blurring, noise removal, and feature detection. It is also known as mask convolution or spatial convolution.

NOTE. *In some literature sources, the 2D convolution is referred to as box filtering, which is an incorrect use of the term. A box filter is a linear averaging filter (see function* `iplBlur` *above). Technically, a box filter can be effectively (although less efficiently) implemented by 2D convolution using a kernel with unit or constant values.*

For 2D convolution, a rectangular kernel is used. The kernel is a matrix of signed integers which are, actually, the signed bytes of the C-language “signed char” type. The value range of these bytes is –128 to 127. The kernel could be a single row (a row filter) or a single column (a column filter) or composed of many rows and columns. There is a cell in the kernel called the “anchor,” which is usually a geometric center of the kernel, but can be skewed with respect to the geometric center.

For each input pixel, the kernel is placed on the image such that the anchor coincides with the input pixel. The output pixel value is computed as the matrix dot product of the image matrix (the portion of the input image on which the kernel is overlaid) and the kernel matrix. Optionally, the output pixel value may be scaled.

The convolution function can be used in two ways. The first way uses a single kernel for convolution. The second way uses multiple kernels and allows the specification of a method to combine the results of convolution with each kernel. This enables efficient implementation of multiple kernels which eliminates the need of storing the intermediate results when using each kernel. One IPL function, `iplConvolve2D()`, can implement both ways.

In addition, `iplConvolveSep2D()`, a convolution function that uses separable kernels is also provided. The convolution kernel is separable into the x and y components.

The 2D convolution functions, `iplCreateConvKernel()` and `iplGetConvKernel()`, allow you to create and access kernels, upon which convolving the image with one or more convolution kernels or with a separable kernel can be performed.

CreateConvKernel

Creates a convolution kernel.

```
IplConvKernel* iplCreateConvKernel(int nRows, int nCols,
int anchorX, int anchorY, char* values, int nShiftR);
```

nRows	The number of rows in the convolution kernel.
nCols	The number of columns in the convolution kernel.
anchorX, anchorY	The [x, y] coordinates of the anchor cell in the kernel. In this coordinate system, the top left corner would be [0, 0] and the bottom right corner would be [*nRows*-1, *nCols*-1]. For a 3 by 3 kernel, the coordinates of the geometric center would be [1, 1]. This specification allows the kernel to be skewed with respect to its geometric center.
values	A pointer to an array of values to be used for the kernel matrix. The values are read in row-major form starting with the top left corner. There should be exactly *nRows***nCols* entries in this array. For example, the array [1, 2, 3, 4, 5, 6, 7, 8, 9] would represent the kernel matrix

1 2 3
4 5 6
7 8 9

`nShiftR`	Scale the resulting output pixel by shifting it to the right *`nShiftR`* times.

Discussion

The function `iplCreateConvKernel()` can be used to create a convolution kernel of arbitrary size and arbitrary anchor point.

Return Value

A pointer to the convolution kernel structure `IplConvKernel`.

GetConvKernel

Reads the attributes of a convolution kernel.

```
void iplGetConvKernel(IplConvKernel* kernel, int* nRows,
int* nCols, int* anchorX, int* anchorY, char** values,
int* nShiftR);
```

`kernel`	The kernel to get the attributes for. The attributes are returned in the remaining arguments.
`nRows`	A pointer to the number of rows in the convolution kernel. Set by the function.
`nCols`	A pointer to the number of columns in the convolution kernel. Set by the function.

`anchorX, anchorY`	Pointers to the [x, y] coordinates of the anchor cell in the kernel. (See iplCreateConvKernel above.) Set by the function.
`values`	A pointer to an array of values to be used for the kernel matrix. The values are read in row-major form starting with the top left corner. There will be exactly `nRows`*`nCols` entries in this array. For example, the array [1, 2, 3, 4, 5, 6, 7, 8, 9] would represent the kernel matrix 1 2 3 4 5 6 7 8 9
`nShiftR`	A pointer to the number of bits to shift (to the right) the resulting output pixel of each convolution. Set by the function.

Discussion

The function `iplGetConvKernel()` can be used to read the attributes of a convolution kernel.

DeleteConvKernel

Deletes a convolution kernel.

```
void iplDeleteConvKernel(IplConvKernel* kernel);
```

`kernel`	The kernel to delete.

Discussion

The function `iplGetConvKernel()` must be used to delete a convolution kernel which was created by the `iplCreateConvKernel()` function.

Convolve2D

Convolves an image with one or more convolution kernels.

```
void iplConvolve2D(IplImage* srcImage, IplImage*
dstImage, IplConvKernel** kernel, int nKernels,
int combineMethod, IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.
kernel	A pointer to an array of pointers to convolution kernels. The length of the array is *nKernels*.
nKernels	The number of kernels in the array *kernel*. The value of *nKernels* can be 1 or more.
combineMethod	The way in which the results of applying each kernel should be combined. This argument is ignored when a single kernel is used. The following combinations are supported:

IPL_SUM	Sums the results.
IPL_SUMSQ	Sums the squares of the results.
IPL_SUMSQROOT	Sums the squares of the results and then takes the square root.
IPL_MAX	Takes the maximum of the results.
IPL_MIN	Takes the minimum of the results.

map	The structure specifying offsets for tiling purposes.

Discussion

The function `iplConvolve2D()` is used to convolve an image with a set of convolution kernels. The results of using each kernel are then combined using the *`combineMethod`* argument.

ConvolveSep2D

Convolves an image with a separable convolution kernel.

```
void iplConvolveSep2D(IplImgreg* srcImage, IplImage*
dstImage, IplConvKernel* xKernel, IplConvKernel* yKernel,
IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`xKernel`	The x or row kernel. Must contain only one row.
`ykernel`	The y or column kernel. Must contain only one column.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplConvolveSep2D()` is used to convolve the input image *`srcImage`* with the separable kernel specified by the row kernel *`xkernel`* and column kernel *`ykernel`*. The resulting output image is *`dstImage`*.

Non-linear Filters

Non-linear filtering involves performing non-linear operations on some neighborhood of the image. Most common are the minimum, maximum and median filters.

MedianFilter

Apply a median filter to the image.

```
void iplMedianFilter(IplImage* srcImage, IplImage*
dstImage, int nRows, int nCols, int anchorX,
int anchorY, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`nRows`	Number of rows in the neighborhood to use.
`nCols`	Number of columns in the neighborhood to use.
`anchorX, anchorY`	The [x, y] coordinates of the anchor cell in the neighborhood. In this coordinate system, the top left corner would be [0, 0] and the bottom right corner would be [`nRows-1`, `nCols-1`]. For a 3 by 3 neighborhood, the coordinates of the geometric center would be [1, 1]. This specification allows the neighborhood to be skewed with respect to its geometric center.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplMedianFilter()` sets each pixel in the output image as the median value of all the input image pixel values in the neighborhood of size *`nRows`* by *`nCols`* with the anchor cell at that pixel. This has the effect of removing the noise in the image.

MaxFilter

Apply a max filter to the image.

```
void iplMaxFilter(IplImage* srcImage, IplImage* dstImage,
int nRows, int nCols, int anchorX, int anchorY, IplCoord*
map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`nRows`	Number of rows in the neighborhood to use.
`nCols`	Number of columns in the neighborhood to use.
`anchorX, anchorY`	The [x, y] coordinates of the anchor cell in the neighborhood. In this coordinate system, the top left corner would be [0, 0] and the bottom right corner would be [*`nRows`*`-1`, *`nCols`*`-1`]. For a 3 by 3 neighborhood, the coordinates of the geometric center would be [1, 1]. This specification allows the neighborhood to be skewed with respect to its geometric center.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplMaxFilter()` sets each pixel in the output image as the maximum value of all the input image pixel values in the neighborhood of size *`nRows`* by *`nCols`* with the anchor cell at that pixel. This has the effect of increasing the contrast in the image.

MinFilter

Apply a min filter to the image.

```
void iplMinFilter(IplImage* srcImage, IplImage* dstImage,
int nRows, int nCols, int anchorX, int anchorY, IplCoord*
map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`nRows`	Number of rows in the neighborhood to use.
`nCols`	Number of columns in the neighborhood to use.
`anchorX, anchorY`	The [x, y] coordinates of the anchor cell in the neighborhood. (In this coordinate system, the top left corner would be [0, 0] and the bottom right corner would be [*`nRows`*`-1`, *`nCols`*`-1`]. For a 3 by 3 neighborhood the coordinates of the geometric center would be [1, 1]). This specification allows the neighborhood to be skewed with respect to its geometric center.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplMinFilter()` sets each pixel in the output image as the minimum value of all the input image pixel values in the neighborhood of size *`nRows`* by *`nCols`* with the anchor cell at that pixel. This has the effect of decreasing the contrast in the image.

Linear Image Transforms

7

This chapter describes the linear image transforms implemented in the library: Fast Fourier Transform (FFT) and Discrete Cosine Transform (DCT). Table 7-1 lists IPL functions performing linear image transform operations.

Table 7-1 Linear Image Transform Functions

Group	Function Name	Description
Fast Fourier Transform (FFT)	`iplRealFft2D`	Computes the forward or inverse 2D FFT of an image.
	`iplCcsFft2D`	Computes the forward or inverse 2D FFT of an image in a complex-conjugate format.
Discrete Cosine Transform (DCT)	`iplDCT2D`	Computes the forward or inverse 2D DCT of an image.

Fast Fourier Transform

This section describes the functions that implement the forward and inverse Fast Fourier Transform (FFT) on the 2-dimensional (2D) image data.

Real-Complex Packed (RCPack2D) Format

The FFT of any real 2D signal, in particular, the FFT of an image is conjugate-symmetric. Therefore, it can be fully specified by storing only half the output data. A special format called `RCPack2D` is provided for this purpose.

The function `iplRealFft2D()` transforms a 2D image and produces the Fourier coefficients in the `RCPack2D` format. To complement this, function `iplCcsFft2D()` is provided that uses its input in `RCPack2D` format, performs the Fourier transform, and produces its output as a real 2D image. The functions `iplRealFft2D()` and `iplCcsFft2D()` together can be used to perform frequency domain filtering of images.

`RCPack2D` format is defined based on the following equations:

$$A_{s,j} = \sum_{l=0}^{L-1} \sum_{k=0}^{K-1} f_{k,l} \exp\left(-\frac{2\pi ijl}{L}\right) \exp\left(-\frac{2\pi iks}{K}\right)$$

$$f_{k,l} = \frac{1}{LK} \sum_{j=0}^{L-1} \sum_{s=0}^{K-1} A_{s,j} \exp\left(\frac{2\pi ijl}{L}\right) \exp\left(\frac{2\pi iks}{K}\right)$$

where $i = \sqrt{-1}$, $f_{k,l} = f(x_k, y_l)$, $x_k = k/K$, $y_l = l/L$.

Note that the Fourier coefficients have the following relationship:

$$\tilde{A}_{s,j} = A_{K-s,L-j}$$

The symbol “~” denotes complex-conjugate. Hence, the $L*K$ real values can be used to reconstruct the $L*K$ complex coefficients $A_{k,l}$. Thus it is enough to store only $L*K$ real values.

Using the DFT (real or complex) function $a_s(y)$, we have the following Fourier coefficients:

$$A_{s,j} = \sum_{l=0}^{L-1} a_s(l) \exp\left(-\frac{2\pi ilj}{L}\right) = \sum_{l=0}^{L-1} \sum_{k=0}^{K-1} f_{k,l} \exp\left(-\frac{2\pi ijl}{L}\right) \exp\left(-\frac{2\pi iks}{K}\right)$$

where

$s = 0, \ldots K/2$	for even K
$s = 0, \ldots (K-1)/2$	for odd K
$j = 0, \ldots (L-1)$	

Other Fourier coefficients can be found using complex-conjugate relations. Fourier coefficients $A_{0,j}$ for real function a_0 (Fourier coefficients $A_{K/2,j}$ for even K) can be stored in the `RCPack` format, other coefficients are standard. The `RCPack` format is a convenient compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that if it is not the natural format used by the real FFT algorithms ("natural" in the sense that bit-reversed order is natural for rad ix-2 complex FFTs).

In the `RCPack2D` format, the output samples of the FFT are arranged as shown in Table 7-2 where Re corresponds to Real and Im corresponds to Imaginary.

Table 7-2 Arrangement of Output Samples in RCPack2D Format

$A_{0,0}$	$ReA_{0,1}$ $ImA_{0,1}$	$ReA_{0,2}$ $ImA_{0,2}$	. . . . . .	$ReA_{0,j}$ $ImA_{0,j}$
$ReA_{1,0}$ $ImA_{1,0}$	$ReA_{1,1}$ $ImA_{1,1}$	$ReA_{1,2}$ $ImA_{1,2}$	. . . . . .	$ReA_{1,j}$ $ImA_{1,j}$
. . .	. . .	. . .	. . .	. . .
$ReA_{K/2,0}$ $ImA_{K/2,0}$	$ReA_{K/2,1}$ $ImA_{K/2,1}$	$ReA_{K/2,2}$ $ImA_{K/2,2}$	. . . . . .	$ReA_{K/2,j}$ $ImA_{K/2,j}$

RealFft2D

Computes the forward or inverse 2D FFT of an image.

```
void iplRealFft2D(IplImage* srcImage, IplImage* dstImage,
                  int flags, IplCoord* map);
```

srcImage	The source image.

`dstImage`	The resultant image in `RCPack2D` format containing the Fourier coefficients. This image must be a multi-channel image containing the same number of channels as *`srcImage`*. The data type for the image must be 8, 16 or 32 bits. This image cannot be the same as the input image *`srcImage`* (that is, an in-place operation is not allowed).
`flags`	Specifies how to perform FFT . This is an integer in which every bit can be assigned the following values using logical `OR`:

`IPL_FFT_Forw`	Do forward transform
`IPL_FFT_Inv`	Do inverse transofrm
`IPL_FFT_NoScale`	Do inverse transform without scaling
`IPL_FFT_DoAlpha`	Transform alpha channel (if *`alphaChannel`* is not 0)
`IPL_FFT_UseInt`	Use only integer core
`IPL_FFT_UseFloat`	Use only float core
`IPL_FFT_OnlyOffsetROIandCalc`	Take only offset ROI and calc
`IPL_FFT_Free`	Only free all working arrays and exit
`IPL_FFT_Save`	Save all working arrays on exit

`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplRealFft2D()` performs an FFT on each channel in the specified rectangular ROI of the input image *`srcImage`* and writes the Fourier coefficients in `RCPack2D` format into the corresponding channel of the output image *`dstImage`*.

Note that the output data will be clamped (saturated) to the limits 0 and `Max`, where `Max` is determined by the data type of the output image. The 32-bit data type will produce the best results, so at least 16-bit data is recommended.

CcsFft2D

Computes the forward or inverse 2D FFT of an image in complex-conjugate format.

```
void iplCcsFft2D(IplImage* srcImage, IplImage* dstImage,
                 int flags, IplCoord* map);
```

`srcImage`	The source image in `RCPack2D` format.
`dstImage`	The resultant image. This image must be a multi-channel image containing the same number of channels as *`srcImage`*. This image cannot be the same as the input image *`srcImage`* (that is, an in-place operation is not allowed).
`flags`	Specifies how to perform FFT . This is an integer in which every bit can be assigned the following values using logical OR:

`IPL_FFT_Forw`	Do forward transform
`IPL_FFT_Inv`	Do inverse transofrm

`IPL_FFT_NoScale`	Do inverse transform without scaling
`IPL_FFT_DoAlpha`	Transform alpha channel (if *`alphaChannel`* is not zero)
`IPL_FFT_UseInt`	Use only integer core
`IPL_FFT_UseFloat`	Use only float core
`IPL_FFT_OnlyOffsetROIandCalc`	Take only offset ROI and calc
`IPL_FFT_Free`	Only free all working arrays and exit
`IPL_FFT_Save`	Save all working arrays on exit

`map` The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplRealFft2D()` performs an FFT on each channel in the specified rectangle ROI of the input image *`srcImage`* and writes the output in `RCPack2D` format to the image *`dstImage`*.

Note that the output data will be clamped (saturated) to the limits 0 and `Max`, where `Max` is determined by the data type of the output image.

Discrete Cosine Transform

This section describes the functions that implement the forward and inverse Discrete Cosine Transform (DCT) on the 2D image data. The output of the DCT for real input data is real. Therefore, unlike FFT, no special format for the transform output is needed.

DCT2D

Computes the forward or inverse 2D DCT of an image.

```
void iplDCT2D(IplImage* srcImage, IplImage* dstImage,
int flags, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image containing the DCT coefficients. This image must be a multi-channel image containing the same number of channels as `srcImage`. The data type for the image must be 8, 16 or 32 bits. This image cannot be the same as the input image `srcImage` (that is, an in-place operation is not allowed).
`flags`	Specifies how to perform FFT. This is an integer whose every bit can be assigned the following values using logical `OR`:

`IPL_DCT_Forward`	Do forward transform
`IPL_DCT_Inverse`	Do inverse transofrm
`IPL_DCT_DoAlpha`	Transform alpha channel (if `alphaChannel` is not 0)
`IPL_DCT_Free`	Only free all working arrays and exit
`IPL_DCT_UseInpBuf`	Use the input image array for the intermediate calculations. The performance of DCT increases, but the input image is destroyed. This value is a default.

`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplDCT2D()` performs a DCT on each channel in the specified rectangular ROI of the input image *`srcImage`* and writes the DCT coefficients into the corresponding channel of the output image *`dstImage`*.

Note that the output data will be clamped (saturated) to the limits `Min` and `Max`, where `Min` and `Max` are determined by the data type of the output image. The 32-bit data type will produce the best results, so at least 16-bit data type is recommended.

Morphological Operations

8

The morphological operations of the Image Processing Library are simple erosion and dilation of an image. A specified number of erosions and dilations are performed as part of image opening or closing operations in order to (respectively) eliminate or fill small and thin holes in objects, break objects at thin points or connect nearby objects, and generally smooth the boundaries of objects without significantly changing their area.

Table 8-1 lists the functions that perform these operations.

Table 8-1 Morphological Operation Functions

Group	Function Name	Description
Erode, Dilate	`iplErode`	Erodes the image an indicated number of times.
	`iplDilate`	Dilates the image an indicated number of times.
Open, Close	`iplOpen`	Opens the image while smoothing the boundaries of large objects.
	`iplClose`	Closes the image while smoothing the boundaries of large objects.

Erode

Erodes the image.

```
void iplErode(IplImage* srcImage, IplImage* dstImage,
                 int nIterations, IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.
nIterations	The number of times to erode the image.
map	The structure specifying offsets for tiling purposes.

Discussion

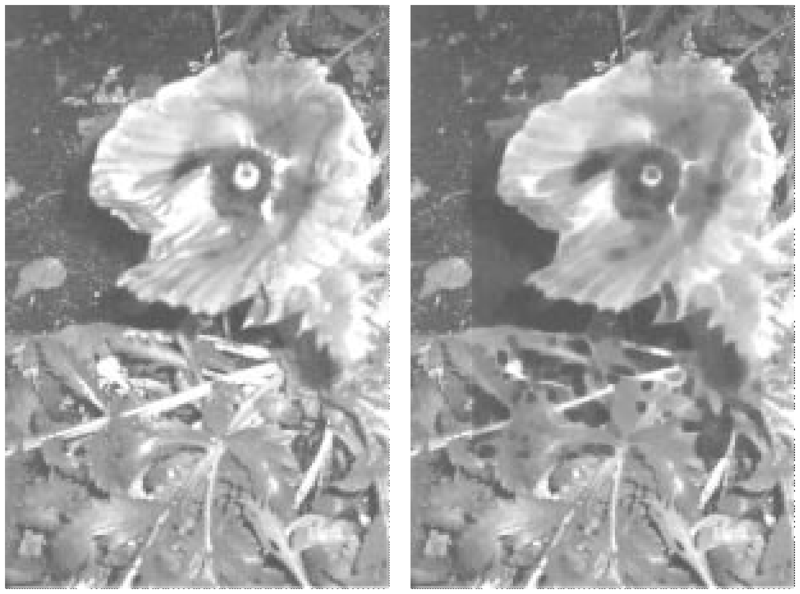
The function `iplErode()` performs an erosion of the image *nIterations* times. The way the image is eroded depends on whether it is a binary image or not.

- For a binary input image, the output pixel is set to zero if the corresponding input pixel or any of its 8 neighboring pixels is a zero.
- For a gray scale or color image, the output pixel is set to the minimum of the corresponding input pixel and its 8 neighboring pixels.

The effect of erosion is to remove spurious pixels (such as noise) and to thin boundaries of objects on a dark background (whose pixel values are less than those of the objects).

Figure 8-1 shows an example of 8-bit gray scale image before erosion (left) and the same image after erosion of a rectangular ROI (right).

Figure 8-1 Erosion in a Rectangular ROI: the Source (left) and Result (right)



The following code (Example 8-1) performs erosion of the image inside the selected rectangular ROI.

Example 8-1 Code Used to Produce Erosion in a Rectangular ROI

```
/* Create output image header with attributes pointed to
by srcImg */

dstImg = iplCreateImageHeader(
       src->nChannels, src->alphaChannel, src->depth,
       "", "", src->dataOrder, src->align,
       src->height, src->width, 0);

       /* Allocate output image */
 iplAllocateImage(dstImage, NOFILL);

       /* Copy source image into output image */
 memcpy(      dstImg->imageData, srcImg->imageData,
         src->imageSize);

       /* Set ROI  attributes */
 roi.coi = 0;        /* channels: all    */
                     /* region:          */
 roi.xOffset = 10;   /* position 10,155  */
 roi.yOffset = 155;
 roi.width = 192;    /* size: 192x86     */
 roi.height = 86;

       /* Set ROI into srcImg and dstImg */
 srcImg->roi = &roi;
 dstImg->roi = &roi;

       /* Erosion */
 IplErode(srcImg,dstImg,2);
```

NOTE. *All source image attributes are defined in the image header pointed to by* `srcImage`.

Dilate

Dilates the image.

```
void iplDilate(IplImage* srcImage, IplImage* dstImage,
int nIterations, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`nIterations`	The number of times to dilate the image.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplDilate()` performs a dilation of the image `nIterations` times. The way the image is dilated depends on whether the image is a binary image or not.

- For a binary input image, the output pixel is set to 1 if the corresponding input pixel is 1 or any of 8 neighboring input pixels is 1.
- For a gray scale or color image, the output pixel is set to the maximum of the corresponding input pixel and its 8 neighboring pixels.

The effect of dilation is to fill up holes and to thicken boundaries of objects on a dark background (whose pixel values are less than those of the objects).

Open

Opens the image by performing erosions followed by dilations.

```
void iplOpen(IplImage* srcImage, IplImage* dstImage,
                int nIterations, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`nIterations`	The number of times to erode and dilate the image.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplOpen()` performs *`nIterations`* of erosion followed by *`nIterations`* of dilation performed by `iplErode()` and `iplDilate()`, respectively.

The process of opening has the effect of eliminating small and thin objects, breaking objects at thin points, and generally smoothing the boundaries of larger objects without significantly changing their area.

See Also

Erode

Dilate

Close

Closes the image by performing dilations followed by erosions.

```
void iplClose(IplImage* srcImage, IplImage* dstImage,
int nIterations, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`nIterations`	The number of times to dilate and erode the image.
`map`	The structure specifying offsets for tiling purposes.

Discussion

The function `iplClose()` performs `nIterations` of dilation followed by `nIterations` of erosion performed by `iplDilate()` and `iplErode()`, respectively.

The process of closing has the effect of filling small and thin holes in objects, connecting nearby objects, and generally smoothing the boundaries of objects without significantly changing their area.

See Also

Erode

Dilate

Color Space Conversion 9

This chapter describes the Image Processing Library functions that perform color space conversion. The following color space conversions are supported in the library:

- Reduction from high bit resolution color to low bit resolution color
- Conversion of absolute color images to and from palette color images
- Color model conversion
- Conversion from color to gray scale and vice versa

Table 9-1 lists color space conversion functions. For information on the absolute-to-palette and palette-to-absolute color conversion, see "Working in the Windows DIB Environment" in Chapter 4.

Table 9-1 Color Space Conversion Functions

Conversion Type	Function Name	Description
Reducing Bit Resolution	`iplReduceBits`	Reduces the number of bits per channel in an image.
Bitonal to gray scale	`iplBitonalToGray`	Converts bitonal images to 8- and 16-bit gray scale images.
Color to gray scale and vice versa	`iplColorToGray` `iplGrayToColor`	Convert color images to and from gray scale images.
Color Models Conversion	`iplRGB2HSV`, `iplHSV2RGB`	Convert RGB images to and from HSV color model.
	`iplRGB2HLS`, `iplHLS2RGB`	Convert RGB images to and from HLS color model.

continued ☞

Table 9-1 Color Space Conversion Functions (continued)

Conversion Type	Function Name	Description
Color Models Conversion (continued)	`iplApplyColorTwist`	Applies a color-twist matrix to an image.
	`iplCreateColorTwist`	Allocates memory for color-twist matrix data structure.
	`iplDeleteColorTwist`	Deletes the color-twist matrix data structure.
	`iplSetColorTwist`	Sets a color-twist matrix data structure.

Reducing the Image Bit Resolution

This section describes functions that reduce the bit resolution of absolute color and gray scale images.

ReduceBits

Reduces the bits per channel in an image.

```
void iplReduceBits(IplImage* srcImage, IplImage*
dstImage, int ditherType, int jitterType, int levels,
IplCoord* map);
```

srcImage	The source image of a higher bit resolution. Refer to the discussion below for a list of valid source and resultant image combinations.

dstImage	The resultant image of a lower bit resolution. Refer to the discussion below for a list of valid source and resultant image combinations.
jitterType	The number specifying the noise added; should be in the range 0 to 8.
ditherType	The type of dithering to be used. The following types are allowed:

`IPL_DITHER_NONE`	No dithering is done
`IPL_DITHER_FS`	The Floid-Steinberg dithering algorithm is used.
`IPL_DITHER_JJH`	The Jarvice-Judice-Hinke dithering algorithm is used.
`IPL_DITHER_STUCKEY`	The Stucki dithering algorithm is used
`IPL_DITHER_BAYER`	The Bayer dithering algorithm is used

levels	Number of levels for dithering; should be a power of 2.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplReduceBits()` reduces a higher bit resolution of the absolute color or gray scale source image *srcImage* to a lower resolution of the resultant absolute color or gray scale image *dstImage*. All combinations of jittering and dithering values are valid. If *jitterType* is greater than 0, some random noise is added to all pixels before the reduction, which eliminates the problem of visible color stepping; see [Bragg]. The resultant image can be used as input to a color quantization method for further reduction in the number of colors; see [Thomas] and [Schumacher].

Table 9-2 lists the valid combinations of the source and resultant image bit data types for reducing the bit resolution.

Table 9-2 Source and Resultant Image Data Types for Reducing the Bit Resolution

Source Image	Resultant Image
32 bit per channel	1 (for gray image), 8 or 16 bit per channel
16 bit per channel	8 or 1 (for gray image) bit per channel
8 bit per channel	1 bit per channel (for gray image)

Bit reducing uses the equation `dst` = `src`*(((1<<`n`) -1)/((1<<`m`) - 1)), where `m` is the bit depth of the source and `n` is the bit depth of the destination. To reduce a gray scale image to a bitonal (1-bit) image, see the discussion under the thresholding function `iplThreshold` in Chapter 10.

Conversion from Bitonal to Gray Scale Images

This section describes the function that performs the conversion of bitonal images to gray scale.

BitonalToGray

Converts a bitonal image to gray scale.

```
void iplBitonalToGray(IplImage* srcImage, IplImage*
dstImage, int ZeroScale, int OneScale, IplCoord* map);
```

`srcImage`	The bitonal source image.

dstImage	The resultant gray scale image. (See the discussion below.)
ZeroScale	The value that zero pixels of the source image should have in the resultant image.
OneScale	The value given to a resultant pixel if the corresponding input pixel is 1.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplBitonalToGray()` converts the input 1-bit bitonal image *srcImage* to an 8s, 8u, 16s or16u gray scale image *dstImage*.

If an input pixel is 0, the corresponding output pixel is set to *ZeroScale*. If an input pixel is 1, the corresponding output pixel is set to *OneScale*.

Conversion of Absolute Colors to and from Palette Colors

Since the IPL image format supports only absolute color images, this functionality is provided only within the context of converting an IPL absolute color image to and from a palette color DIB image. See the section "Working in the Windows DIB Environment" in Chapter 4.

Conversion from Color to Gray Scale

This section describes the function that performs the conversion of absolute color images to gray scale.

ColorToGray

Converts a color image to gray scale.

```
void iplColorToGray(IplImage* srcImage,
IplImage* dstImage, IplCoord* map);
```

srcImage	The source image. See the discussion below for a list of valid source and resultant image combinations.
dstImage	The resultant image. See the discussion below for a list of valid source and resultant image combinations.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplColorToGray()` converts a color source image *srcImage* to a gray scale resultant image *dstImage*.
Table 9-3 lists the valid combinations of source and resultant image bit data types for conversion from color to gray scale.

Table 9-3 Source and Resultant Image Data Types for Conversion from Color to Gray Scale

Source Image (data type)	Result image (data type)
32 bit per channel	Gray scale (1, 8, or 16 bit)
16 bit per channel	Gray scale (1, 8, or 16 bit)
8 bit per channel	Gray scale (1, 8, or 16 bit)

The weights to compute true luminance from linear red, green and blue are these:

Y = 0.212671 * R + 0.715160 * G + 0.072169 * B.

Conversion from Gray Scale to Color (Pseudo-color)

This section describes the conversion of gray scale image to pseudo color.

GrayToColor

Converts a gray scale to color image.

```
void iplGrayToColor (IplImage* srcImage,
IplImage* dstImage, float FractR,float FractG,float
FractB, IplCoord* map);
```

srcImage	The source image. See the discussion below for a list of valid source and resultant image combinations.
dstImage	The resultant image. See the discussion below for a list of valid source and resultant image combinations.
FractR, FractG, FractB	The red, green and blue intensities for image reconstruction. See the discussion below for a list of valid *FractR*, *FractG*, and *FractB* values.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplGrayToColor()` converts a gray scale source image *srcImage* to a resultant pseudo-color image *dstImage*. Table 9-4 lists the valid combinations of source and resultant image bit data types for conversion from gray scale to color.

Table 9-4 Source and Resultant Image Data Types for Conversion from Gray Scale to Color

Source Image (data type)	Result image (data type)
Gray scale 1 bit	8 bit per channel
Gray scale 8 bit	8 bit per channel
Gray scale 16 bit	16 bit per channel
Gray scale 32 bit	32 bit per channel

The equation for chrominance in RGB from luminance `Y` is:

`R` = *`FractR`* * `Y;`	0 <= *`FractR`* <= 1
`G` = *`FractG`* * `Y;`	0 <= *`FractG`* <= 1
`B` = *`FractB`* * `Y;`	0 <= *`FractB`* <= 1.

If *`FractR`*==0 && *`FractG`*==0 && *`FractB`* == 0, then the default values are used in above equation so that:

`R` = 0.212671 * `Y`, `G` = 0.715160 * `Y`, `B` = 0.072169 * `Y`.

Conversion of Color Models

To convert one color model to another, a color twist matrix can be used. See the “Color Twist Matrices” section, which describes this method and presents examples of various color model conversions.

Described in this section are conversions of color models when the color twist matrix cannot be used.

Note that conversion of the RGB to the CMY models can be performed by a simple subtraction. The function `iplSubtractS` can be used to accomplish this conversion for two 8-bit per channel images. For example, with maximum pixel value of 255, the `iplSubtractS()` function is used as follows:

```
    iplSubtractS(rgbImage, cmyImage, 256, TRUE)
```

This call converts the RGB image `rgbImage` to the CMY image `cmyImage` by setting each channel in the CMY image as follows:

```
C = 255 - R
M = 255 - G
Y = 255 - B
```

The conversion from CMY to RGB is similar: just switch the RGB and CMY images.

RGB2HSV

Converts from the RGB color model to the HSV color model.

```
void iplRGB2HSV(IplImage* rgbImage, IplImage* hsvImage,
IplCoord* map);
```

`rgbImage`	The source RGB image.
`hsvImage`	The resultant HSV.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplRGB2HSV()` converts the RGB image `rgbImage` to the HSV image `hsvImage`. The function checks that the input image is an RGB image. The channel sequence and color model of the output image are set to HSV.

HSV2RGB

Converts from the HSV color model to the RGB color model.

```
void iplHSV2RGB(IplImage* hsvImage, IplImage* rgbImage,
IplCoord* map);
```

hsvImage	The source HSV image.
rgbImage	The resultant RGB.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplHSV2RGB()` converts the HSV image *hsvImage* to the RGB image *rgbImage*. The function checks that the input image is an HSV image. The channel sequence and color model of the output image are set to RGB.

RGB2HLS

Converts from the RGB color model to the HLS color model.

```
void iplRGB2HLS(IplImage* rgbImage, IplImage* hlsImage,
IplCoord* map);
```

rgbImage	The source RGB image.
hlsImage	The resultant HLS.

`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplRGB2HLS()` converts the RGB image *`rgbImage`* to the HLS image *`hlsImage`*. The function checks that the input image is an RGB image. The function sets the channel sequence and color model of the output image to HLS.

HLS2RGB

Converts from the HLS color model to the RGB color model.

```
void iplHLS2RGB(IplImage* hlsImage, IplImage* rgbImage,
IplCoord* map);
```

`hlsImage`	The source HLS image.
`rgbImage`	The resultant RGB.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplHLS2RGB()` converts the HLS image *`hlsImage`* to the RGB image *`rgbImage`*; see [Rogers85]. The function checks that the input image is an HLS image. The channel sequence and color model of the output image are set to RGB.

Using Color-Twist Matrices

One of the methods of color model conversion is using a color-twist matrix. The color-twist matrix is a generalized 4 by 4 matrix [$t_{i,j}$] that converts the three channels (a, b, c) into (d, e, f) according to the following matrix multiplication by a color-twist matrix (the superscript `T` is used to indicate the transpose of the matrix).

$$[d, e, f, 1]^T = \begin{bmatrix} t11 & t12 & t13 & t14 \\ t21 & t22 & t23 & t24 \\ t31 & t32 & t33 & t34 \\ 0 & 0 & 0 & t44 \end{bmatrix} * [a, b, c, 1]^T$$

To apply a color-twist matrix to an IPL image, use the function `iplApplyColorTwist()`. But first call the `iplCreateColorTwist()` and `iplSetColorTwist()` functions to create the data structure `IplColorTwist`. This data structure contains the color-twist matrix and allows you to store the data internally in a form that is efficient for computation.

The function descriptions that follow provide examples of using the color-twist matrices for color model conversion.

CreateColorTwist

Creates a color-twist matrix data structure.

```
IplColorTwist* iplCreateColorTwist(int data[16],
int scalingValue);
```

`data`	An array containing the sixteen values that constitute the color-twist matrix. The values are in row-wise order. Color-twist values that are in the range -1 to 1 should be scaled up to be in the

	range -2^{31} to 2^{31}. (Simply multiply the floating point number in the -1 to 1 range by 2^{31}.)
`scalingValue`	The scaling value: an exponent of a power of 2 that was used to convert to integer values; for example, if 2^{31} was used to multiply the values, the `scalingValue` is 31. This value is used for normalization.

Discussion

The function `iplCreateColorTwist()` allocates memory for the data structure `IplColorTwist` and creates te color-twist matrix that can subsequently be used by the function `iplApplyColorTwist()`.

Return Value

A pointer to the `IplColorTwist` data structure containing the color-twist matrix in the form suitable for efficient computation by the function `iplApplyColorTwist()`.

SetColorTwist

Sets a color-twist matrix data structure.

```
void iplSetColorTwist(IplColorTwist* cTwist, int
data[16],
int scalingValue);
```

`data`	An array containing the sixteen values that constitute the color-twist matrix. The values are in row-wise order. Color-twist values that are in the range -1 to 1 should be scaled up to be in the

	range -2^{31} to 2^{31}. (Simply multiply the floating point number in the -1 to 1 range by 2^{31}.)
`scalingValue`	The scaling value: an exponent of a power of 2 that was used to convert to integer values; for example, if 2^{31} was used to multiply the values, the `scalingValue` is 31. This value is used for normalization.

Discussion

The function `iplSetColorTwist()` is used to set the vaules of the color-twist matrix in the data structure `IplColorTwist` that can subsequently be used by the function `iplApplyColorTwist()`.

Return Value

A pointer to the `IplColorTwist` data structure containing the color-twist matrix in the form suitable for efficient computation by the function `iplApplyColorTwist()`.

ApplyColorTwist

Applies a color-twist matrix to an image.

```
void iplApplyColorTwist(IplImage* srcImage,
IplImage* dstImage, IplColorTwist* cTwist, int offset,
IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.

cTwist	The color-twist matrix data structure that was prepared by a call to the function `iplSetColorTwist()`.
offset	An offset value that will be added to each pixel channel after multiplication by the color-twist matrix.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplApplyColorTwist()` applies the color-twist matrix to each of the first three color channels in the input image to obtain the resulting data for the three channels.

For example, the matrix below can be used to convert normalized `PhotoYCC` to normalized `PhotoRGB` (both with an opacity channel) when the channels are in the order YCC and RGB, respectively:

$$\begin{matrix} 2^{31} & 0 & 2^{31} & 0 \\ 2^{31} & X & Y & 0 \\ 2^{31} & 2^{31} & 0 & 0 \\ 0 & 0 & 0 & 2^{31} \end{matrix}$$

where $X = -416611827$ (that is, $-0.194 * 2^{31}$) and
$Y = -1093069176$ (that is, $-0.509 * 2^{31}$).

Color-twist matrices may also be used to perform many other color conversions and operations such as

- Lightening an image
- Color saturation
- Color balance
- R, G, and B color adjustments
- Contrast Adjustment

DeleteColorTwist

Frees memory used for a color-twist matrix.

```
void iplDeleteColorTwist(IplColorTwist* cTwist);
```

cTwist	The color-twist matrix data structure that was prepared by a call to the function `iplCreateColorTwist()`.

Discussion

The function `iplDeleteColorTwist()` frees memory used for the color-twist matrix structure referred to by *cTwist*.

Histogram and Thresholding Functions 10

This chapter describes functions that operate on an image on a pixel-by-pixel basis, in particular, the operations that alter the histogram of the image. In addition, the use of color-twist matrices for color model conversions is described. Table 10-1 lists histogram and thresholding functions in the IPL.

Table 10-1 Histogram and Thresholding Functions

Group	Function Name	Description
Thresholding	`iplThreshold`	Performs a simple thresholding of an image.
Lookup Table and Histogram	`iplContrastStretch`	Stretches the contrast of an image using intensity transformation.
	`iplComputeHisto`	Computes the intensity histogram of an image.
	`iplHistoEqualize`	Enhances an image by flattening its intensity histogram.

Thresholding

The thresholding operation changes pixel values depending on whether they are less than, equal to, or greater than the specified *threshold*. If an input pixel value is less than the *threshold*, the corresponding output pixel is set to the minimum presentable value. Otherwise, it is set to the maximum presentable value.

Threshold

Performs a simple thresholding of an image.

```
void iplThreshold(IplImage* srcImage, IplImage* dstImage,
                  int threshold, IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.
threshold	The threshold value to use for each pixel. The pixel value in the output is set to the maximum presentable value if it is greater than or equal to the threshold value (for each channel). Otherwise the pixel value in the output is set to the minimum presentable value.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplThreshold()` thresholds the source image *srcImage* using the value *threshold* to create the resultant image *dstImage*. The pixel value in the output is set to the maximum presentable value (for example, 255 for an 8-bit-per-channel image) if it is greater than or equal to the threshold value. Otherwise it is set to the minimum presentable value (for example, 0 for an 8-bit-per-channel image). This is done for each channel in the input image.

To convert an image to bi-tonal, do the following:

- use `iplColorToGray()` to convert a color image to gray scale
- use `iplThreshold()` to threshold the image to max and zero values
- use `iplConvert()` with `IPL_BITS_LOW` flag set to convert to `1U` depth.

Lookup Table (LUT) and Histogram Operations

A LUT can be used to specify an intensity transformation. Given an input intensity, LUT can be used to look up an output intensity. Usually a LUT is provided for each channel in the image, although sometimes the same LUT can be shared by many channels.

The IplLUT Structure

Example 10-1 presents a C language definition for the `IplLUT` structure to set a LUT.

Example 10-1 IplLUT Definition

```
typedef struct _IplLUT {
    int         num;   /* number of keys or values */
    int*        key;
    int*        value;
    int*        factor;
    int         interpolateType;
 } IplLUT;
```

The *key* array has the length *num*; the *value* and *factor* are arrays of the same length *num*-1. The *interpolateType* can be either `IPL_LUT_LOOKUP` or `IPL_LUT_INTER`.
Consider the following example of *num* = 4:

key	*value*	*factor*
k1	v1	f1
k2	v2	f2
k3	v3	f3
k4		

If *interpolateType* is `LOOKUP`, then any input intensity `D` in the range `k1 ≤ D < k2` will result in the value `v1`, in the range `k2 ≤ D < k3` will result in the value `v2` and so on. If *interpolateType* is `INTER`, then an intensity `D` in the range `k1 ≤ D < k2` will result in the linearly interpolated value

```
v1 + [(v2 - v1)/(k2 - k1)] * (D - k1)
```

The *value* `(v2-v1)/(k2-k1)` is pre-computed and stored in the array *factor* in the `IplLUT` data structure.

The data structure described above can be used to specify a piece-wise linear transformation that is ideal for the purpose of contrast stretching.

The histogram is a data structure that shows how the intensities in the image are distributed. The same data structure `IplLUT` is used for a histogram except that *interpolateType* is always `IPL_LUT_LOOKUP` and *factor* is a `NULL` pointer for a histogram. However, unlike the LUT, the *value* array represents counts of pixels falling in the specified ranges in the *key* array.

The sections that follow describe the functions that use the above data structure.

ConstrastStretch

Stretches the contrast of an image using an intensity transformation.

```
void iplContrastStretch(IplImage* srcImage,
IplImage* dstImage, IplLUT** lut, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`lut`	An array of pointers to LUTs, one pointer for each channel. Each lookup table should have the `key`, `value` and `factor` arrays fully initialized (see "The `IplLUT` Structure"). One or more channels may share the same LUT. Specifies an intensity transformation.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplContrastStretch()` stretches the contrast in a color source image `srcImage` by applying intensity transformations specified by LUTs in `lut` to produce an output image `dstImage`. Fully specified LUTs should be provided to this function.

ComputeHisto

Computes the intensity histogram of an image.

```
void iplComputeHisto(IplImage* srcImage, IplLUT** lut,
IplCoord* map);
```

srcImage	The source image for which the histogram will be computed.
lut	An array of pointers to LUTs, one pointer for each channel. Each lookup table should have the `key` array fully initialized. The `value` array will be filled by this function. (For the `key` and `value` arrays, see "The `IplLUT` Structure" above.) The same LUT can be shared by one or more channels.
map	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplComputeHisto()` computes the intensity histogram of an image. The histograms (one per channel in the image) are stored in the array `lut` containing all the LUTs. The `key` array in each LUT should be initialized before calling this function. The `value` array containing the histogram information will be filled in by this function. (For the `key` and `value` arrays, see "The `IplLUT` Structure" above.)

HistoEqualize

Enhances an image by flattening its intensity histogram.

```
void iplHistoEqualize(IplImage* srcImage,
IPLImage* dstImage, IplLUT** lut, IplCoord* map);
```

`srcImage`	The source image for which the histogram will be computed.
`dstImage`	The resultant image after equalizing.
`lut`	The histogram of the image is represented as an array of pointers to LUTs, one pointer for each channel. Each lookup table should have the `key` and `value` arrays fully initialized. (For the `key` and `value` arrays, see "The `IplLUT` Structure" above.) These LUTs will contain flattened histograms after this function is executed. In other words, the call of `iplHistoEqualize()` is destructive with respect to the LUTs.
`map`	The structure specifying offsets for tiling purposes; see IplCoord Structure in Chapter 2.

Discussion

The function `iplHistoEqualize()` enhances the source image `srcImage` by flattening its histogram represented by `lut` and places the enhanced image in the output image `dstImage`. After execution, `lut` points to the flattened histogram of the output image.

Image Geometric Transforms 11

This chapter describes the IPL functions that perform geometric transforms to resize the image or change its orientation. The geometric transforms are performed by resampling (“Zoom,” “Decimate,” and “Rotate”) or flipping the axis of the image (“Mirror”). Table 11-1 lists image geometric transform functions.

Table 11-1 Image Geometric Transform Functions

Group	Function Name	Description
Resizing	`iplZoom`	Zooms or expands an image.
	`iplDecimate`	Decimates or shrinks an image.
Changing Orientation	`iplMirror`	Mirrors an image about a horizontal or vertical axis.
	`iplRotate`	Rotates an image.

Changing Image Size

The functions that expand or shrink an image perform image resampling by using various kinds of interpolation: nearest neighbor, linear, or cubic convolution.

Zoom

Zooms or expands an image.

```
void iplZoom(IplImage* srcImage, IplImage* dstImage,
int xDst, int xSrc, int yDst, int ySrc, int interpolate,
IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`xSrc, ySrc`	X and Y dimensions of the source image.
`xDst, yDst`	X and Y dimensions of the destination image.
	These four integers must be positive, meeting the conditions of `xDst` ≥ `xSrc` and `yDst` ≥ `ySrc` to specify the fractions `xDst/xSrc` and `yDst/ySrc`. These fractions indicate the value to magnify the image in the X and Y directions. For example, `xDst` = 2, `xSrc` = 1, `yDst` = 2, `ySrc` = 1 doubles the image size in each dimension to give an image 4 times larger in area.
`interpolate`	The type of interpolation to perform for resampling. The following are currently supported:

`IPL_INTER_NN`	Nearest neighbor interpolation.
`IPL_INTER_LINEAR`	Linear interpolation.
`IPL_INTER_CUBIC`	Cubic convolution interpolation.

map	The structure specifying offsets for tiling purposes. See IplCoord Structure in Chapter 2.

Discussion

The function `iplZoom()` zooms or expands the source image *srcImage* by *xDst/xSrc* in the X direction and *yDst/ySrc* in the Y direction. The interpolation specified by *interpolate* is used during resampling the input image.

Decimate

Decimates or shrinks an image.

```
void iplDecimate(IplImage* srcImage, IplImage* dstImage,
int xDst, int xSrc, int yDst, int ySrc, int interpolate,
IplCoord* map);
```

srcImage	The source image.
dstImage	The resultant image.
xSrc, ySrc	X and Y dimensions of the source image.
xDst, yDst	X and Y dimensions of the destination image.

These four integers should be positive, meeting the conditions *xDst* ≤ *xSrc* and *yDst* ≤ *ySrc* to specify the fractions *xDst/xSrc* and *yDst/ySrc*. These fractions indicate the value to shrink the image in the X and Y directions. For example, *xDst* = 1, *xSrc* = 2, *yDst* = 1, *ySrc* = 2 halves the image size in each dimension to give an image 1/4 times smaller in area.

`interpolate`	The type of interpolation to perform for resampling. The following are currently supported:
	`IPL_INTER_NN` Nearest neighbor interpolation.
	`IPL_INTER_LINEAR` Linear interpolation.
	`IPL_INTER_CUBIC` Cubic convolution interpolation.
`map`	The structure specifying offsets for tiling purposes. See IplCoord Structure in Chapter 2.

Discussion

The function `iplDecimate()` decimates or shrinks the source image *`srcImage`* by *`xDst/xSrc`* in the X direction and *`yDst/ySrc`* in the Y direction. The interpolation specified by *`interpolate`* is used during resampling the input image.

Changing Image Orientation

The functions described in this section change image orientation by rotating or mirroring the source image. Rotation involves image sampling by using various kinds of interpolation: nearest neighbor, linear, or cubic convolution. Mirroring is performed by flipping the image axis in horizontal or vertical direction.

Rotate

Rotates an image.

```
void iplRotate(IplImage* srcImage, IplImage* dstImage,
int angle, int centerX, int centerY, int interpolate,
IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`angle`	The angle in hundredths of degree to rotate the image (for example, 6000 for 60 degrees). The image is rotated about the center specified as `centerX` and `centerY` coordinates.
`centerX, centerY`	The coordinates of the rotation center.
`interpolate`	The type of interpolation to perform for resampling. The following are currently supported:

`IPL_INTER_NN`	Nearest neighbor interpolation.
`IPL_INTER_LINEAR`	Linear interpolation.
`IPL_INTER_CUBIC`	Cubic convolution interpolation.

`map`	The structure specifying offsets for tiling purposes. See IplCoord Structure in Chapter 2.

Discussion

The function `iplRotate()` rotates the source image *`srcImage`* by *`angle`* degrees around the origin defined by the coordinates *`centerX`* and *`centerY`*. The interpolation specified by *`interpolate`* is used during resampling the input image.

Mirror

Mirrors an image about a horizontal or vertical axis.

```
void iplMirror(IplImage* srcImage, IplImage* dstImage,
int flipAxis, IplCoord* map);
```

`srcImage`	The source image.
`dstImage`	The resultant image.
`flipAxis`	Specifies the axis to mirror the image. Use 0 for the horizontal axis, 1 for a vertical axis and -1 for both horizontal and vertical axes.
`map`	The structure specifying offsets for tiling purposes. See IplCoord Structure in Chapter 2.

Discussion

The function `iplMirror()` mirrors or flips the source image `srcImage` about a horizontal or vertical axis or both.

Supported Image Attributes and Operation Modes

A

This appendix contains tables that list the supported image attributes and operation modes for all IPL functions that have input and/or output images. The `ipl` prefixes in the function names are omitted.

Table A-1 Image Attributes and Modes of Data Exchange Functions

Function	Depths	Input and output images must have the same				Rect. ROI	Border Mode	In-place	Tiling
		depth	order	origin	COI	supported (x)			
`Set`	all†	always in-place				x		x	x
`Copy`	all	x	x	x	x	x		x	x
`Exchange`	all	x	x	x	x	x		x	x
`Convert`	all								x

† all = 1u, 8s, 8u, 16s, 16u, and 32s bits per channel

Table A-2 Windows* DIB Conversion Functions

Function	Depths		Input & output images have the same			Remarks
	input	output	order	origin	# of channels	
`ConvertFromDIB`	all‡	1u,8u,16u	x	x		Rectangular ROI,
`ConvertToDIB`	1u,8u,16u	all‡	x	x	x	border mode and
`TranslateDIB`	1bpp	1u	x		x	tiling are not
	other‡	8u	x		x	supported

‡ all = 1, 4, 8, 16, 24, and 32 bits per pixel (DIB images).
other = 4, 8, 16, 24, and 32 bits per pixel (DIB images).

For `iplConvertFromDIB`, the number of channels, bit depth per channel and the dimensions of the IPL image should be greater than or equal to those of the DIB image. When converting a DIB RGBA image, the IPL image should also contain an alpha channel.

Table A-3 Image Attributes and Modes of Arithmetic and Logical Functions

Function	Depths	Input and output images must have the same				Rect. ROI	Border Mode	In-place	Tiling
		depth	order	origin	COI	supported (x)			
AddS	all[†]	x	x	x	x	x		x	x
SubtractS	all	x	x	x	x	x		x	x
MultiplyS	all	x	x	x	x	x		x	x
MultiplySScale	8u,16u	x	x	x	x	x			x
Square	all	x	x	x	x	x		x	x
Add	all	x	x	x	x	x		x	x
Subtract	all	x	x	x	x	x		x	x
Multiply	all	x	x	x	x	x		x	x
MultiplyScale	8u,16u	x	x	x	x	x			x
LShiftS	all	x	x	x	x	x		x	x
RShiftS	all	x	x	x	x	x		x	x
Not	all	x	x	x	x	x		x	x
AndS	all	x	x	x	x	x		x	x
OrS	all	x	x	x	x	x		x	x
XorS	all	x	x	x	x	x		x	x
And	all	x	x	x	x	x		x	x
Or	all	x	x	x	x	x		x	x
Xor	all	x	x	x	x	x		x	x

[†] all = 1u, 8s, 8u, 16s, 16u, and 32s bits per channel

Table A-4 Image Attributes and Modes of Alpha-Blending Functions

Function	Depths	Input and output images must have the same				Rect. ROI	Border Mode	In-place	Tiling
		depth	order	origin	COI	supported (x)			
PreMultiplyAlpha	8u,16u	x	x	x	x	x			x
AlphaComposite	8u,16u	x	x	x	x	x			x
AlphaCompositeC	8u,16u	x	x	x	x	x			x

Table A-5 Image Attributes and Modes of Filtering Functions

Function	Depths	Input and output images must have the same				Rect. ROI	Border Mode	In-place	Tiling
		depth	order	origin	COI	supported (x)			
Blur	>1bpc	x	x	x	x	x	x	x	x
Convolve2D	all†	x	x	x	x	x	x	x	x
ConvolveSep2D	all	x	x	x	x	x	x		x
MaxFilter	all	x	x	x	x	x	x		x
MinFilter	all	x	x	x	x	x	x		x
MedianFilter	all	x	x	x	x	x	x		x

† all = 1u, 8s, 8u, 16s, 16u, and 32s bits per channel

Table A-6 Image Attributes and Modes of Fourier and DCT Functions

Function	Depths		Input & output images have the same			Rect. ROI	Border Mode	In-place	Tiling
	input	output	order	origin	COI	supported (x)			
DCT2D	>1bpc	16s,32s	x	x		x			
RealFft2D	8u,16u	16s,32s	x	x	x	x			
CcsFft2D	16s,32s	8u,16u	x		x	x			

Table A-7 Image Attributes and Modes of Morphological Operations

Function	Depths	Input and output images must have the same				Rect. ROI	Border Mode	In-place	Tiling
		depth	order	origin	COI	supported (x)			
Erode	1u,8u,16u	x	x	x	x	x	x		x
Dilate	1u,8u,16u	x	x	x	x	x	x		x
Open	1u,8u,16u	x	x	x	x	x	x		x
Close	1u,8u,16u	x	x	x	x	x	x		x

Table A-8 Image Attributes and Modes of Color Space Conversion Functions

Function	Depths		Input & output images have the same				Rect. ROI	Bord. Mode	In-place	Tiling
	input	output	depth	order	origin	COI	supported (x)			
ReduceBits	32s	8u,16u		x	x	x				
	16u	8u		x	x	x				
BitonalToGray	1u	>1bpc								x
RGB2HSV	1u,16u,32s		x	x	x	x				x
HSV2RGB	1u,16u,32s		x	x	x	x				x
RGB2HLS	1u,16u,32s		x	x	x	x				x
HLS2RGB	1u, 16u,32s		x	x	x	x				x
ApplyColorTwist	16u,32s		x	x	x	x	x			x

Table A-9 Image Attributes and Modes of Histogram and Thresholding Functions

Function	Depths	Input and output images must have the same				Rect. ROI	Border Mode	In-place	Tiling
		depth	order	origin	COI	supported (x)			
`Threshold`	8u,8s,16u, 16s, 32s	x	x	x	x	x	x		x
`ComputeHisto`	1u,8u,16u	no output image				x	x		x
`HistoEqualize`	8u,16u	x	x	x	x	x	x		x
`ContrastStretch`	8u,16u	x	x	x	x	x	x		x

Table A-10 Image Attributes and Modes of Geometric Transform Functions

Function	Depths	Input and output images must have the same				Rect. ROI	Border Mode	In-place	Tiling
		depth	order	origin	COI	supported (x)			
`Mirror`	1u,8u,16u	x	x	x	x	x			x
`Rotate`	1u,8u,16u	x	x	x	x	x			x
`Zoom`	1u,8u,16u	x	x	x	x	x			x
`Decimate`	1u,8u,16u	x	x	x	x	x			x

Bibliography

This bibliography provides a list of publications that might be useful to the Image Processing Library users. This list is not complete; it serves only as a starting point. The books [Rogers85], [Rogers90], and [Foley90] are good resources of information on image processing and computer graphics, with mathematical formulas and code examples.

The Image Processing Library is part of Intel Performance Libraries Suite. The manuals [RPL96] and [SPL96] describe Intel Recognition Primitives Library and Intel Signal Processing Library, which are other parts of the Performance Libraries Suite.

[Bragg]	Dennis Bragg. A simple color reduction filter, *Graphic Gems III*: 20–22.
[Foley90]	James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice,* Second Edition. Addison Wesley, 1990.
[Rogers85]	David Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill, 1985.
[Rogers90]	David Rogers and J.Alan Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, 1990.
[RPL96]	*Intel Recognition Primitives Library Reference Manual.* Intel Corp. Order number 637785, Rev.4, 1996.
[SPL96]	*Intel Signal Processing Library Reference Manual.* Intel Corp. Order number 630508, Rev.6, 1996.
[Schumacher]	Dale A. Schumacher. A comparison of digital halftoning techniques, *Graphic Gems III*: 57–71.
[Thomas]	Spencer W. Thomas and Rod G. Bogart. Color dithering, *Graphic Gems II*: 72–77.

Glossary

absolute colors	Colors specified by each pixel's coordinates in a color space. IPL functions use images with absolute colors. *See* palette colors.
alpha channel	A color channel, also known as the opacity channel, that can be used in color models; for example, the RGBA model.
arithmetic operation	An operation that adds, subtracts, multiplies, shifts, or squares the image pixel values.
CCS	*See* complex conjugate-symmetric.
channel of interest	The color channel on which an IPL operation acts (or processing occurs). Channel of interest (COI) can be considered as a separate case of region of interest (ROI).
CMY	Cyan-magenta-yellow. A three-channel color model that uses cyan, magenta, and yellow color channels.
CMYK	Cyan-magenta-yellow-black. A four-channel color model that uses cyan, magenta, yellow, and black color channels.
COI	*See* channel of interest.
color-twist matrix	A matrix used to multiply the pixel coordinates in one color space for determining the coordinates in another color space.
complex conjugate-symmetric	A kind of symmetry that arises in the Fourier transform of real signals. A complex conjugate-symmetric signal has the property that $x(-n) = x(n)^*$, where * denotes conjugation.

conjugate	The conjugate of a complex number $a+b$j is $a-b$j.
conjugate-symmetric	*See* complex conjugate-symmetric.
DCT	Acronym for the discrete cosine transform. *See* "Discrete Cosine Transform" in Chapter 7.
decimation	An IPL geometric transform operation that shrinks the source image.
DIB	Device-independent bitmap, an image format used by IPL in Windows* environment.
dilation	An IPL morphological operation whose effect is to fill up holes and thicken object boundaries.
dyadic operation	An operation that has two input images. It can have other input parameters as well.
erosion	An IPL morphological operation that results in less noise and thinner object boundaries.
FFT	Acronym for the fast Fourier transform. *See* "Fast Fourier Transform" in Chapter 7.
four-channel model	A color model that uses four color channels; for example, the RGBA color model.
geometric transform functions	IPL functions that perform geometric transformations of images: zoom, decimation, rotation, and mirror functions.
gray scale image	An image characterized by a single intensity channel so that each intensity value corresponds to a certain shade of gray.
HLS	Hue-lightness-saturation. A three-channel color model that uses hue, lightness, and saturation channels. The HLS and HSV models differ in the way of scaling the image luminance. *See* HSV.

HSV	Hue-saturation-value. A three-channel color model that uses hue, saturation, and value channels. HSV is often used as a synonym for the HSB (hue-saturation-brightness) and HSI (hue-saturation-intensity) models. *See* HLS.
hue	A color channel in several color models that measures the "angular" distance (in degrees) from red to the particular color: 60 corresponds to yellow, 120 to green, 180 to cyan, 240 to blue, and 300 to magenta. Hue is undefined for shades of gray.
in-place operation	An operation whose output image is one of the input images. *See* out-of-place operation.
linear filtering	In IPL, either neighborhood averaging (blur) or 2D convolution operations.
linear image transforms	In IPL, either the fast Fourier transform (FFT) or the discrete cosine transform (DCT).
luminance	A measure of image intensity, as perceived by a "standard observer". Since human eyes are more sensitive to green and less to red or blue, different colors of equal physical intensity make different contribution to luminance. *See* `ColorToGray` in Chapter 9.
LUT	Acronym for lookup table (palette).
MMX™ technology	A major enhancement to the Intel Architecture aimed at better performance in multimedia and communications applications. The technology uses four new data types, eight 64-bit MMX registers, and 57 new instructions implementing the SIMD (single instruction, multiple data) technique.
monadic operation	An operation that has a single input image. It can have other input parameters as well.
morphological operation	In IPL, simple erosion or dilation of an image.

MSI	Acronym for multi-spectral image. An MSI can use any number of channels and colors.
non-linear filtering	In IPL, minimum, maximum, or median filtering operation.
opacity channel	*See* alpha channel.
out-of-place operation	An operation whose output is an image other than the input image(s). *See* in-place operation.
palette colors	Colors specified by a palette, or lookup table. IPL uses palette colors only in operations of image conversion to and from absolute colors. *See* absolute colors.
PhotoYCC*	A Kodak* proprietary color encoding and image compression scheme. *See* YCC.
pixel depth	The number of bits determining a single pixel in the image.
pixel-oriented ordering	Storing the image information in such an order that the values of all color channels for each pixel are clustered; for example, RGBRGB... . *See* "Channel Sequence" in Chapter 2.
plane-oriented ordering	Storing the image information so that all data of one color channel follow all data of another channel, thus forming a separate "plane" for each channel; for example, RRRRRGGGGG...
point operation	An operation performed on a pixel-by-pixel basis. IPL point operations include applying a color-twist matrix, computing and altering the image histogram, contrast stretching, histogram equalization and thresholding.
region of interest	An image region on which an IPL operation acts (or processing occurs).
RGB	Red-green-blue. A three-channel color model that uses red, green, and blue color channels.
RGBA	Red-green-blue-alpha. A four-channel color model that uses red, green, blue, and alpha (or

	opacity) channels.
ROI	*See* region of interest.
saturation	A quantity used for measuring the purity of colors. The maximum saturation corresponds to the highest degree of color purity; the minimum (zero) saturation corresponds to shades of gray.
scanline	All image data for one row of pixels.
standard gray palette	A complete palette of a DIB image whose red, green, and blue values are equal for each entry and monotonically increasing from entry to entry.
three-channel model	A color model that uses three color channels; for example, the CMY color model.
YCC	A three-channel color model that uses one luminance channel (Y) and two chroma channels (usually denoted by C_R and C_B). The term is sometimes used as a synonym for the entire PhotoYCC encoding scheme. *See* PhotoYCC.
zoom	A geometric transform function that magnifies the source image.

Index