

Matrox Genesis Native Library

version 2.1

Command Reference

Manual no. 10495-701-0210

May 12, 2000

Matrox® is a registered trademark of Matrox Electronic Systems Ltd.

Microsoft®, MS-DOS®, Windows®, and Windows NT® are registered trademarks of Microsoft Corporation.

Intel®, Pentium®, and Pentium II® are registered trademarks of Intel Corporation.

Texas Instruments is a trademark of Texas Instruments Incorporated.

RAMDAC™ is a trademark of Booktree.

All other nationally and internationally recognized trademarks and tradenames are hereby acknowledged.

© Copyright Matrox Electronic Systems Ltd., 2000. All rights reserved.

Disclaimer: Matrox Electronic Systems Ltd. reserves the right to make changes in specifications at any time and without notice. The information provided by this document is believed to be accurate and reliable. However, no responsibility is assumed by Matrox Electronic Systems Ltd. for its use; nor for any infringements of patents or other rights of third parties resulting from its use. No license is granted under any patents or patent rights of Matrox Electronic Systems Ltd.

PRINTED IN CANADA

Contents

<i>Chapter 1: Programming with the Genesis Native Library.</i>	<i>5</i>
--	----------

Overview.	6
-----------	---

A quick command reference	10
---------------------------	----

<i>Chapter 2: The command descriptions.</i>	<i>23</i>
---	-----------

Command description notes	24
---------------------------	----

<i>Appendix A: Glossary</i>	<i>393</i>
-----------------------------	------------

<i>Appendix B: Examples.</i>	<i>413</i>
------------------------------	------------

blob.c	414
--------	-----

first.c	419
---------	-----

grab.c	421
--------	-----

jpeg.c	423
--------	-----

pat.c	426
-------	-----

process.c	430
-----------	-----

tfilter.c	444
-----------	-----

Index

Product Support

Chapter 1: Programming with the Genesis Native Library

Overview

The Genesis Native Library is a board-specific library that consists of an extensive set of functions for image processing and specialized operations such as the scheduling and synchronization of parallel operations. It provides explicit control over grabbing, processing, transferring to the Host, and displaying. The library was designed for the efficient use of the Genesis board, as well as for fast application development.

User Guide

This manual describes each function of the Genesis Native Library. Note that it is a command reference of the Native Library, rather than a user manual. In many cases, it assumes you are familiar with concepts explained in the *Genesis Native Library User Guide*.

Hardware overview

The Matrox Genesis main board is a single-slot, PCI board with on-board processing and optional grab and display sections. Processing is performed by the Texas Instruments TMS320C80 (also known as the 'C80) running at 50 MHz. This single-chip, digital signal multiprocessor contains a RISC master processor and four parallel processors.

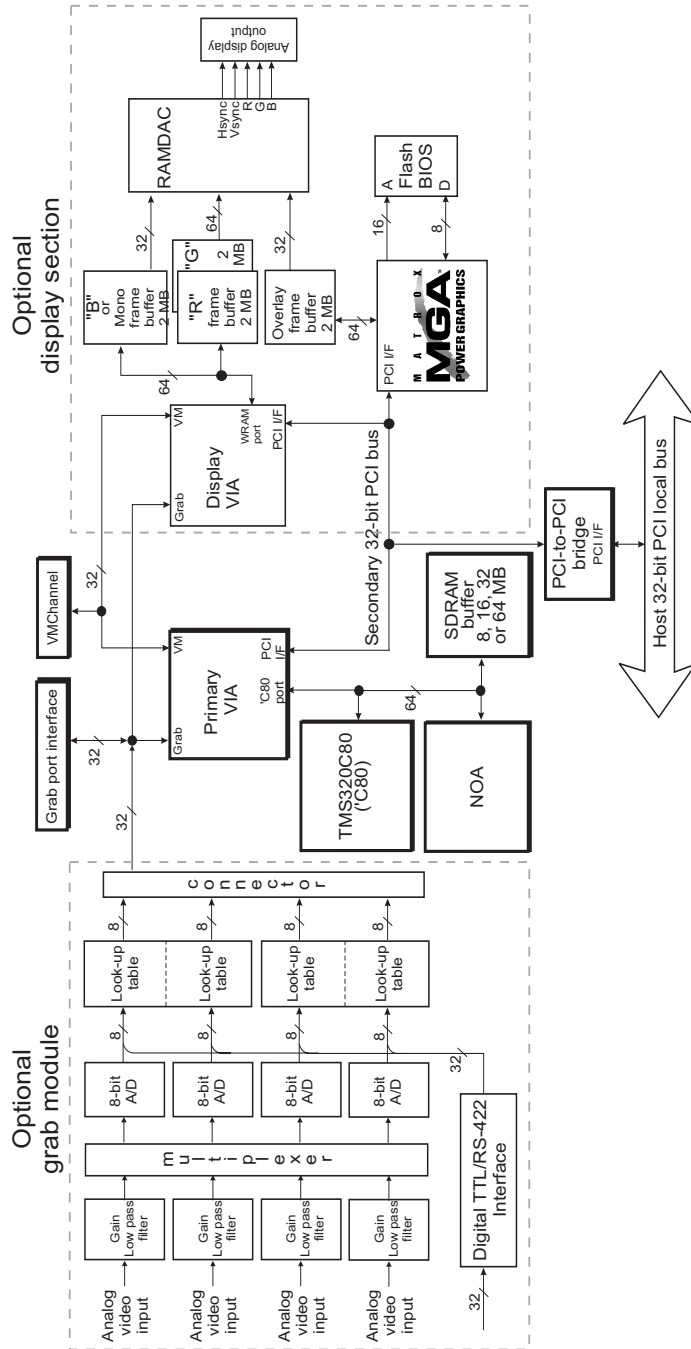
The Video Interface ASIC (VIA) connects all data buses to one another, and directs and monitors data through a system. The VIA can also format data in various ways as it directs it through a system.

The main board can include a Matrox-designed ASIC (the Neighborhood Operations Accelerator or NOA), which can accelerate neighborhood operations such as convolutions and morphology. In addition, one or more processor boards can be added to increase performance. A typical processor board (which requires one extra PCI slot) contains two 'C80s, each with additional memory, VIA, and optional NOA.

The Genesis-LC is a low-cost version of the main board. Basically, it is the main board without a processing section.

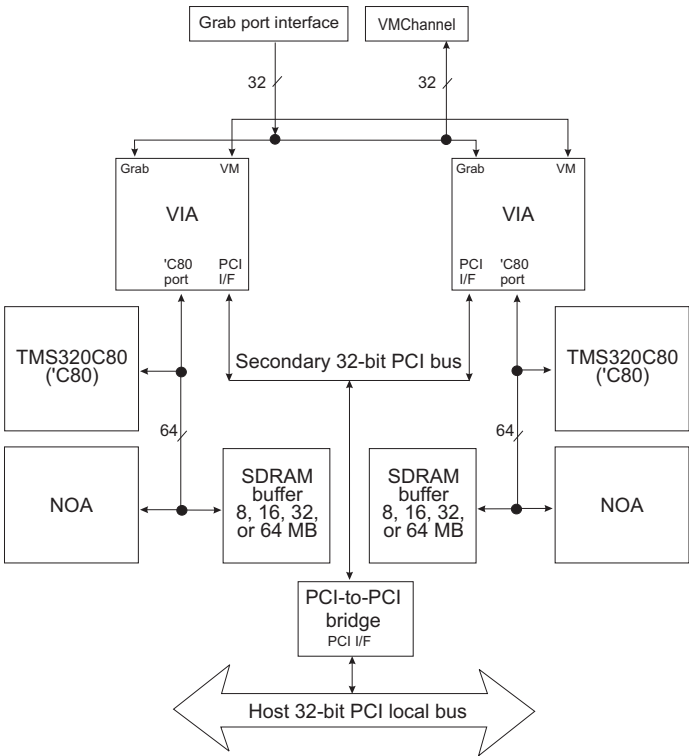
The next page contains block diagrams of the main board, processor board, and Genesis-LC. You will find more hardware-related information in the *Genesis Installation and Hardware Reference*.

Genesis Main Board

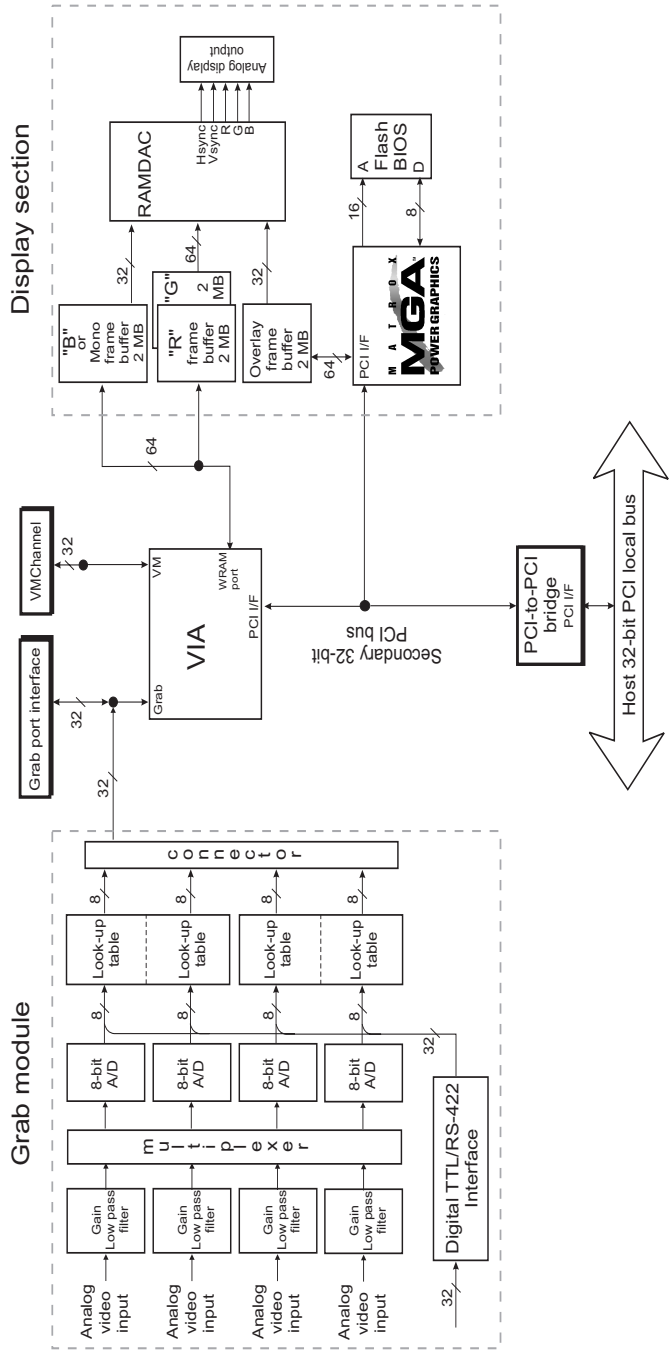


* You will find more detail on some of the above components in the *Genesis Installation and Hardware Reference*.

Genesis Processor Board



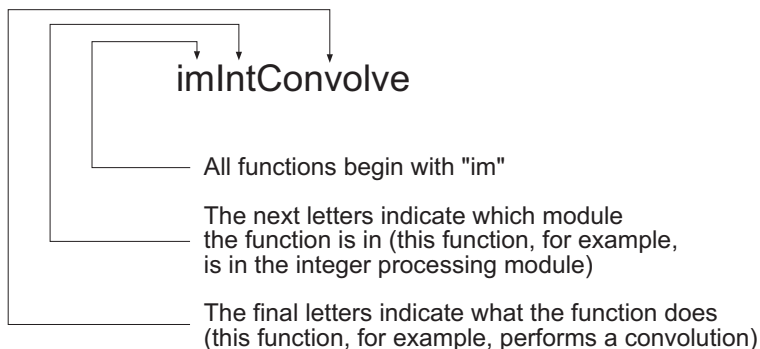
Genesis-LC



* You will find more detail on some of the above components in the *Genesis Installation and Hardware Reference*.

A quick command reference

This section lists and provides a short description of the functions of each Genesis Native Library module. Note that Library function names are meant to be easy-to-follow:



The application control module

Command	Parameters	Description
<code>imAppCatchError()</code>	Mode, HandlerFunc, HandlerParam	Catch application-wide errors.
<code>imAppControl()</code>	Item, Value	Set an application-wide attribute.
<code>imAppGetError()</code>	Item, ValuePtr	Check the application for errors.
<code>imAppInquire()</code>	Item, ValuePtr	Inquire about an application-wide attribute.

The binary processing module

Command	Parameters	Description
<code>imBinConvert()</code>	Thread, Src, Dst, Cond, Val1, Val2, OSB	Convert between integer and binary buffers.
<code>imBinCountDifference()</code>	Thread, Src1, Src2, Result, OSB	Count the difference between two binary buffers, or perform a binary event count on a single binary buffer.
<code>imBinMorphic()</code>	Thread, Src, Dst, Kernel, Op, Niter, Control, OSB	Perform a morphological operation.
<code>imBinThin()</code>	Thread, Src, Dst, Niter, Control, OSB	Perform a fast binary thinning operation.
<code>imBinTriadic()</code>	Thread, SrcA, SrcB, SrcC, Dst, Op, OSB	Perform a logical operation.

The blob analysis module

Command	Parameters	Description
imBlobAllocFeatureList()	Thread, FeatListPtr	Allocate a blob analysis feature list.
imBlobAllocResult()	Thread, ResultPtr	Allocate a blob analysis result buffer.
imBlobCalculate()	Thread, IdentBuf, GreyBuf, FeatList, Result, Mode, OSB	Perform a blob analysis calculation.
imBlobControl()	Thread, Result, Item, Value	Change a blob analysis control setting.
imBlobCopyResult()	Thread, Result, Feature, Mode, DstBuf, OSB	Copy blob analysis results.
imBlobCopyRuns()	Thread, Result, Label, XBuf, YBuf, LenBuf, OSB	Copy run information about a blob.
imBlobFill()	Thread, Result, DstBuf, Mode, Value, OSB	Draw filled blobs into a buffer.
imBlobFree()	Thread, ResultOrFeatList	Free a blob analysis result buffer or feature list.
imBlobGetLabel()	Thread, Result, X, Y, LabelPtr	Get the label value of a blob at a specified position.
imBlobGetNumber()	Thread, Result, NumPtr	Get the number of currently included blobs.
imBlobGetResult()	Thread, Result, Feature, Mode, ArrayPtr	Transfer blob analysis results to Host memory.
imBlobGetResultSingle()	Thread, Result, Label, Feature, Mode, ValuePtr	Read the feature value of a single blob.
imBlobGetRuns()	Thread, Result, Label, Type, XPtr, YPtr, LenPtr	Transfer run information about a blob into Host memory.
imBlobInquire()	Thread, Result, Item, ValuePtr	Inquire about a blob analysis control setting.
imBlobLabel()	Thread, Result, DstBuf, Mode, OSB	Draw labeled blobs into a buffer.
imBlobSelect()	Thread, Result, Operation, Feature, Mode, Cond, Low, High	Exclude or delete blobs.
imBlobSelectFeature()	Thread, FeatList, Feature, Mode	Select a feature for calculation.
imBlobSelectFeret()	Thread, FeatList, Angle	Select a specific Feret diameter for calculation.
imBlobSelectMoment()	Thread, FeatList, Mode, Type, XOrder, YOrder	Select a specific moment for calculation.

The buffer management module

Command	Parameters	Description
imBufAlloc()	Thread, Xsize, Ysize, Nbands, Type, Location, BufPtr	Allocate a buffer with multiple bands.
imBufAlloc1d()	Thread, Xsize, Type, Location, BufPtr	Allocate a 1-d buffer.
imBufAlloc2d()	Thread, Xsize, Ysize, Type, Location, BufPtr	Allocate a 2-d buffer.
imBufAllocControl()	Thread, BufPtr	Allocate a control buffer.
imBufChild()	Thread, Buf, Xstart, Ystart, Xsize, Ysize, ChildPtr	Allocate a child buffer within an existing buffer or on the display.
imBufChildBand()	Thread, Buf, BandNum, BandPtr	Allocate a child buffer from a band of a multi-band buffer.
imBufChildMove()	Thread, Child, Xoff, Yoff, Xsize, Ysize	Move and/or resize a child buffer.
imBufClear()	Thread, Buf, Value, OSB	Clear a buffer to a constant value.
imBufClone()	Thread, Buf, NewLocation, NewBufPtr	Duplicate a buffer.
imBufControl()	Thread, Buf, Item, Value	Modify a buffer attribute.
imBufCopy()	Thread, SrcBuf, DstBuf, Control, OSB	Copy a buffer.
imBufCopyField()	Thread, SrcBuf, SrcTag, DstBuf, DstTag	Copy buffer field(s).
imBufCopyROI()	Thread, SrcBuf, DstBuf, Mode, Control, OSB	Copy an ROI (or ROIs) from a source buffer into a destination buffer.
imBufCopyPCI()	Thread, SrcBuf, DstBuf, Control, OSB	Copy a buffer over the PCI bus.
imBufCopyVM()	Thread, SrcBuf, DstBuf, SrcControl, DstControl, OSB	Copy a buffer over the VMChannel.
imBufCreate()	Thread, Xsize, Ysize, Nbands, Type, Location, AddrPtr, Pitch, BufPtr	Create a buffer out of memory that has already been allocated.
imBufFree()	Thread, Buf	Free a buffer.
imBufGet()	Thread, Buf, Ptr	Get data from a buffer to Host memory.
imBufGet1d()	Thread, Buf, Xstart, Xsize, Ptr	Get a block of data from within a 1-d buffer to Host memory.
imBufGet2d()	Thread, Buf, Xstart, Ystart, Xsize, Ysize, Ptr	Get a block of data within a 2-d buffer to Host memory.
imBufGetField()	Thread, Buf, Tag, ValuePtr	Get the value of a field (and return it as type long).

Command	Parameters	Description
imBufGetFieldDouble()	Thread, Buf, Tag, ValuePtr	Get the value of a field (and return it as type double).
imBufGetNextField()	Thread, Buf, ContextPtr, TagPtr, ValuePtr	Get the tag and value of the next buffer field.
imBufInquire()	Thread, Buf, Item, ValuePtr	Inquire about a buffer attribute.
imBufLoad()	Thread, FileName, Format, Buf	Load data from a file into a buffer.
imBufMap()	Thread, Buf, Band, Ystart, AddrPtr, PitchPtr, NlinesPtr	Map a buffer into Host memory.
imBufModify()	Thread, Buf, Xsize, Ysize, Type	Modify a buffer's dimensions and/or type.
imBufPack()	Thread, SrcBuf, TagBuf, DstBuf, Mode, OSB	Pack or unpack a buffer.
imBufPut()	Thread, Buf, Ptr	Transfer data from Host memory to a buffer.
imBufPut1d()	Thread, Buf, Xstart, Xsize, Ptr	Transfer a block of data from Host memory into part of a 1-d buffer.
imBufPut2d()	Thread, Buf, Xstart, Ystart, Xsize, Ysize, Ptr	Transfer a block of data from Host memory into part of a 2-d buffer.
imBufPutField()	Thread, Buf, Tag, Value	Add or modify a buffer field.
imBufRemoveField()	Thread, Buf, Tag	Remove a field from a buffer.
imBufRestore()	Thread, FileName, Format, Location, BufPtr	Load data from a file into an automatically allocated buffer.
imBufSave()	Thread, FileName, Format, Buf	Save a buffer to a file.

The camera control module

Command	Parameters	Description
imCamAlloc()	Thread, CamFile, Mode, CameraPtr	Allocate a camera definition.
imCamClone()	Thread, Camera, Mode, NewCamPtr	Duplicate a camera definition.
imCamControl()	Thread, Camera, Item, Value	Change a setting of a camera definition.
imCamFree()	Thread, Camera	Free a camera definition.
imCamInquire()	Thread, Camera, Item, ValuePtr	Inquire about a camera definition setting.
imCamSave()	Thread, CamFile, Camera	Save a camera definition to a file.

The cursor control module

Command	Parameters	Description
imCurAlloc()	Dev, Mode, CursorPtr	Allocate a new cursor.
imCurDefine()	Dev, Cursor, SizeX, SizeY, HotX, HotY, DataPtr	Define the cursor's shape.
imCurEnable()	Dev, Flag	Enable or disable the current hardware cursor.
imCurFree()	Dev, Cursor	Free the specified cursor.
imCurGetPosition()	Dev, PosXPtr, PosYPtr	Get the current cursor's position.
imCurSelect()	Dev, Cursor	Select the specified cursor as the current cursor.
imCurSetColor()	Dev, Cursor, Red, Green, Blue	Set the specified cursor's colors.
imCurSetPosition()	Dev, PosX, PosY	Set the current cursor's position.

The device control module

Command	Parameters	Description
imDevAlloc()	System, Node, ShellFile, Mode, DevPtr	Allocate a device.
imDevFree()	Dev	Free a device.
imDevInquire()	Dev, Item, ValuePtr	Inquire about a device attribute.

The digitizer control module

Command	Parameters	Description
imDigAlloc()	Thread, System, Digitizer, Mode, DigPtr	Allocate a digitizer.
imDigCapture()	Thread, Dig, Cam, Mode	Enable a synchronized or software-triggered grab.
imDigControl()	Thread, Dig, Item, Value	Set a digitizer attribute.
imDigFree()	Thread, Dig	Free a digitizer.
imDigGrab()	Thread, Dig, Cam, Buf, Count, Control, OSB	Grab into a buffer.
imDigInquire()	Thread, Dig, Item, ValuePtr	Inquire about a digitizer attribute.

The display module

Command	Parameters	Description
imDispAlloc()	Thread, System, Display, DispFile, Mode, DispPtr	Allocate a display.
imDispControl()	Thread, Disp, Control, Mode	Set display attributes.
imDispFree()	Thread, Disp	Free a display.
imDispInquire()	Thread, Disp, Item, ValuePtr	Inquire about a display attribute.

The floating-point processing module

Command	Parameters	Description
imFloatConvert()	Thread, Src, Dst, Mode, OSB	Convert between integer and floating-point buffers.
imFloatDyadic()	Thread, Src1, Src2, Dst, Op, OSB	Perform an arithmetic operation between two buffers.
imFloatMac1()	Thread, Src, Dst, Fac, Const, OSB	Multiply and accumulate with one buffer.
imFloatMac2()	Thread, Src1, Src2, Dst, Fac1, Fac2, OSB	Multiply and accumulate with two buffers.
imFloatMonadic()	Thread, Src, Const, Dst, Op, OSB	Perform an arithmetic operation between a buffer and a constant.
imFloatUnary()	Thread, Src, Dst, Op, OSB	Perform a unary operation on a buffer.

The data generation module

Command	Parameters	Description
imGen1d()	Thread, Buf, Func, Start, End, NumCoefs, Coefs, OSB	Generate data into a 1-d buffer.
imGenWarp1stOrder()	Thread, Coef, Transform, Val1, Val2, Mode, OSB	Generate first-order warp coefficients.
imGenWarp4Corner()	Thread, Coef, X1, Y1, X2, Y2, X3, Y3, X4, Y4, Xstart, Ystart, Xend, Yend, Mode, OSB	Generate warp coefficients to map an arbitrary quadrilateral onto a rectangle.
imGenWarpLutMatrix()	Thread, Xlut, Ylut, Coef, Control, OSB	Generate address LUTs for a matrix-defined warping using imIntWarpLut() .

The graphics module

Command	Parameters	Description
imGraArc()	Thread, Context, Buf, Xcen, Ycen, Xrad, Yrad, StartAng, EndAng	Draw an elliptical arc.
imGraArcFill()	Thread, Context, Buf, Xcen, Ycen, Xrad, Yrad, StartAng, EndAng	Draw a filled elliptical arc.
imGraFill()	Thread, Context, Buf, Xstart, Ystart	Fill a connected region.
imGraLine()	Thread, Context, Buf, Xstart, Ystart, Xend, Yend	Draw a line.
imGraPlot()	Thread, Context, Buf, Xbuf, Ybuf, NumPoints	Plot a series of (x, y) points.
imGraRect()	Thread, Context, Buf, Xstart, Ystart, Xend, Yend	Draw a rectangle.
imGraRectFill()	Thread, Context, Buf, Xstart, Ystart, Xend, Yend	Draw a filled rectangle.
imGraText()	Thread, Context, Buf, Xstart, Ystart, String	Write text.

The integer processing module

Command	Parameters	Description
imIntBinarize()	Thread, Src, Dst, Cond, Low, High, Val1, Val2, OSB	Binarize an image.
imIntClip()	Thread, Src, Dst, Cond, Low, High, Val1, Val2, Mode, OSB	Clip or binarize an image.
imIntConnectMap()	Thread, Src, Dst, Lut, Control, OSB	Perform a 3x3 connectivity mapping.
imIntConvert()	Thread, Src, Dst, Mode, OSB	Convert a buffer from one integer type to another.
imIntConvertColor()	Thread, Src, Dst, Type, Coef, OSB	Perform a color conversion.
imIntConvolve()	Thread, Src, Dst, Kernel, Control, OSB	Perform a convolution.
imIntCorrelate()	Thread, Src, Dst, Model, Control, OSB	Perform normalized grayscale correlation.
imIntCountDifference()	Thread, Src1, Src2, Result, OSB	Count the differences between two images.
imIntDistance()	Thread, Src, Dst, Transform, Control, OSB	Perform a distance transform.
imIntDyadic()	Thread, Src1, Src2, Dst, Op, OSB	Perform an arithmetic or logical operation between two images.
imIntErodeDilate()	Thread, Src, Dst, Kernel, Op, Niter, Control, OSB	Perform grayscale erosion or dilation.

Command	Parameters	Description
imIntFindExtreme()	Thread, Src, Result, Mode, OSB	Find the minimum and/or maximum pixel value in an image.
imIntFFT()	Thread, SrcR, SrcI, DstR, DstI, Control, OSB	Perform a fast Fourier transform.
imIntFlip()	Thread, Src, Dst, Func, Mode, OSB	Flip or rotate an image.
imIntGainOffset()	Thread, Src, Dst, Offset, Gain, Shift, ClipVal, Mode, OSB	Apply per-pixel gain and offset correction.
imIntHistogram()	Thread, Src, Result, Mode, OSB	Perform a histogram.
imIntHistogramEqualize()	Thread, Src, Dst, HistSize, Func, Alpha, Min, Max, Mode, OSB	Perform a histogram equalization.
imIntLabel()	Thread, Src, Dst, Mode, OSB	Label connected regions.
imIntLocateEvent()	Thread, Src, X, Y, Pix, Num, Cond, Low, High, OSB	Locate pixels that satisfy a condition.
imIntLutMap()	Thread, Src, Dst, Lut, OSB	Perform a look-up table mapping.
imIntMac1()	Thread, Src, Dst, Fac, Const, Shift, OSB	Multiply and accumulate with one image.
imIntMac2()	Thread, Src1, Src2, Dst, Fac1, Fac2, Shift, OSB	Multiply and accumulate with two images.
imIntMonadic()	Thread, Src, Const, Dst, Op, OSB	Perform an arithmetic or logical operation between an image and a constant.
imIntProject()	Thread, Src, Result, Angle, Mode, OSB	Project a 2D image into 1D.
imIntRank()	Thread, Src, Dst, Kernel, Rank, Control, OSB	Perform a rank filter operation.
imIntRecFilter()	Thread, Src, Src2, Dst, Dst2, Lut, SrcBits, DstBits, Control, OSB	Perform adaptive recursive (temporal) filtering.
imIntScale()	Thread, SrcBuf, DstBuf, XFac, YFac, Control, OSB	Scale an image by integer or non-integer factors.
imIntSubsample()	Thread, Src, Dst, Xfac, Yfac, Control, OSB	Subsample an image.
imIntThickThin()	Thread, Src, Dst, Kernel, Op, Niter, Control, OSB	Perform grayscale thinning or thickening.
imIntTriadic()	Thread, SrcA, SrcB, SrcC, Dst, Rotate, Op, Mode, OSB	Perform an arithmetic or logical operation with three operands.
imIntWarpLut()	Thread, Src, Dst, Xlut, Ylut, Control, OSB	Perform a warping using a look-up table inverse address calculation.

Command	Parameters	Description
imIntWarpPolynomial()	Thread, Src, Dst, Coef, Control, OSB	Perform a warping using polynomial inverse address calculation.
imIntZoom()	Thread, Src, Dst, Xfac, Yfac, Control, OSB	Zoom an image.

The JPEG module

Command	Parameters	Description
imJpegAlloc()	Thread, Control, JpegPtr	Allocate a JPEG buffer.
imJpegControl()	Thread, Jpeg, Item, Value	Change a control setting of a JPEG buffer.
imJpegControlBand()	Thread, Jpeg, Band, Item, Value	Change a control setting of a JPEG buffer, for a specific band.
imJpegDecode()	Thread, Buf, Jpeg, OSB	Decompress a compressed image.
imJpegEncode()	Thread, Buf, Jpeg, OSB	Compress an image.
imJpegFree()	Thread, Jpeg	Free a JPEG buffer.
imJpegGetTable()	Thread, Jpeg, TableType, TableNum, TableSizePtr, TablePtr	Transfer a JPEG table to Host memory.
imJpegInquire()	Thread, Jpeg, Item, ValuePtr	Inquire about a JPEG buffer.
imJpegPutTable()	Thread, Jpeg, TableType, TableNum, TableSize, TablePtr	Transfer a table from Host memory to a JPEG buffer.
imJpegRead()	Thread, FileHandle, Jpeg	Read a compressed image from an open file.
imJpegReadBuf()	Thread, Buf, Jpeg, Start, OSB	Read a compressed image from a buffer.
imJpegRestore()	Thread, FileName, JpegPtr	Load a compressed image from a file into an automatically allocated JPEG buffer.
imJpegSave()	Thread, FileName, Jpeg	Save a compressed image from a JPEG buffer to a file.
imJpegWrite()	Thread, FileHandle, Jpeg	Write a compressed image to an open file.
imJpegWriteBuf()	Thread, Buf, Jpeg, Start, OSB	Write a compressed image to a buffer.

The pattern matching module

Command	Parameters	Description
imPatAllocAutoModel()	Thread, SrcBuf, Xsize, Ysize, XUncert, YUncert, Type, Mode, ModelPtr	Automatically select and allocate a pattern matching model.
imPatAllocModel()	Thread, SrcBuf, XOff, YOff, XSize, YSize, Type, ModelPtr	Allocate a pattern matching model.
imPatAllocResult()	Thread, NumEntries, ResultPtr	Allocate a pattern matching result buffer.
imPatCopy()	Thread, Model, DstBuf, Mode, OSB	Copy a pattern matching model.
imPatFindModel()	Thread, SrcBuf, Model, Result, OSB	Find a pattern matching model in an image.
imPatFree()	Thread, ModelOrResult	Free a pattern matching model or result buffer.
imPatGetNumber()	Thread, Result, NumPtr	Determine the number of matches above the acceptance level.
imPatGetResult()	Thread, Result, Type, Ptr	Transfer results of a search to Host memory.
imPatInquire()	Thread, ModelOrResult, Item, ValuePtr	Inquire about a pattern matching model or result buffer.
imPatPreprocModel()	Thread, Buf, Model, Mode, OSB	Preprocess a pattern matching model.
imPatRead()	Thread, FileHandle, ModelPtr	Read a pattern matching model from an open file.
imPatRestore()	Thread, FileName, ModelPtr	Restore a pattern matching model from a file.
imPatSave()	Thread, FileName, Model	Save a model to a file.
imPatSetAcceptance()	Thread, Model, Acceptance	Set the acceptance level of a search.
imPatSetAccuracy()	Thread, Model, Accuracy	Set the positional accuracy of a search.
imPatSetCenter()	Thread, Model, XCen, YCen	Set a model's center position (hot spot).
imPatSetCertainty()	Thread, Model, Certainty	Set the certainty level of a search.
imPatSetDontCare()	Thread, Model, SrcBuf, XOff, YOff, Value	Set model pixels to the "don't care" state.
imPatSetNumber()	Thread, Model, Number	Set the number of matches to find.

Command	Parameters	Description
imPatSetPosition()	Thread, Model, XOff, YOff, XSize, YSize	Set the search region of a search.
imPatSetSearchParameter()	Thread, Model, Param, Value	Set an internal search parameter.
imPatSetSpeed()	Thread, Model, Speed	Set the speed of a search.
imPatWrite()	Thread, FileHandle, Model	Write a model to an open file.

The run-length encoding module

Command	Parameters	Description
imRleDecode()	Thread, Buf, CompBuf, Control, OSB	Decode (decompress) a run-length encoded image.
imRleEncode()	Thread, Buf, CompBuf, Control, OSB	Run-length encode (compress) an image.

The synchronization module

Command	Parameters	Description
imSyncAlloc()	Thread, OSBPtr	Allocate an operation status block.
imSyncControl()	Thread, OSB, Item, Value	Change the state or mode of operation of an operation status block.
imSyncFree()	Thread, OSB	Free an operation status block.
imSyncGetError()	Thread, OSB, Item, ValuePtr	Check a function for errors.
imSyncHost()	Thread, OSB, State	Synchronize the Host with a function.
imSyncThread()	Thread, OSB, State	Synchronize a thread with an operation in another thread.

The system control module

Command	Parameters	Description
imSysClock()	Thread, Offset	Read the system clock.
imSysInquire()	System, Item, ValuePtr	Inquire about a system attribute.
imSysTimeStamp()	Thread, Buf, Tag, Offset	Write a time stamp to the specified buffer field.

The thread control module

Command	Parameters	Description
imThrAlloc()	Dev, Control, ThreadPtr	Allocate a thread.
imThrCancel()	Thread	Cancel all commands queued to a specified thread.
imThrControl()	Thread, Item, Value	Set a thread attribute.
imThrFree()	Thread	Free a thread.
imThrGetError()	Thread, Item, ValuePtr	Check a thread for errors.
imThrHalt()	Thread, Mode	Halt the current function.
imThrInquire()	Thread, Item, ValuePtr	Inquire about a thread.
imThrNop()	Thread, OSB	No operation.

Chapter 2: The command descriptions

Command description notes

A few notes about the command descriptions:

- They are presented in alphabetical order. As such, the functions of each module are grouped together.
- When a parameter requires a buffer, the allowed data types of the buffer are indicated. If no mention is made of the buffer's sign, it means the buffer can be either signed or unsigned.
- Each function header contains symbols that indicate whether the function:

- Is synchronous or asynchronous: Sync or Async
- Uses the parallel processors (PPs) of the 'C80: PP
- Uses the NOA (if available): NOA
- Supports in-place operation
(i.e. source and destination buffers can be the same): In-Place
- Supports direct processing of multiple-band images: Multi-band
- Can run on the Genesis-LC: Gen-LC

Note that in-place operation applies only to processing functions. In addition, the absence of a symbol means, for example, that the function does not use the PPs, does not use the NOA, does not support in-place operation, etc.

For general product support see our website, www.matrox.com, or contact the Matrox Customer Support Group.

imAppCatchError

Sync

Gen-LC

Synopsis Catch application-wide errors.

Format **void imAppCatchError(Mode, HandlerFunc, HandlerParam)**

long Mode;	Operation mode
void (*HandlerFunc) (void*);	Pointer to function (or NULL)
void* HandlerParam;	Parameter to pass to function (or NULL)

Description This function establishes a user-defined error handler (that is, establishes a user-defined function that is called automatically once an error in the application is detected). You can have this function called on subsequent errors by clearing error information within the user-defined function (call **imAppGetError()** with the IM_ERR_RESET flag). In addition, you can have a specified parameter value passed to the function.

To get information about the detected error, you must use **imAppGetError()** within the user-defined function. If you want to clear error information so that the user-defined function is called on subsequent errors, only use the IM_ERR_RESET flag when retrieving the last required error item.

Note that, if an error is caused by a synchronous function, the handler is called immediately, before the function returns. If an error is caused by an asynchronous function, you should never rely on the handler being called until you have explicitly waited for the completion of the function.

The **Mode** parameter specifies the mode of operation. This parameter must be set to IM_DEFAULT.

The **HandlerFunc** parameter specifies the address of the user-defined function to call upon detecting an error. If you want to disable error handling, call **imAppCatchError()** again and set the **HandlerFunc** parameter to NULL.

The **HandlerParam** parameter specifies the address of the parameter to pass to the user-defined function. This parameter can be set to NULL if you don't need a parameter.

Example The following code uses **imAppCatchError()** to print error messages whenever an error is detected:

```
...
void myhandler(void *param);

void main()
{
    /* Establish error handling */
    imAppCatchError(IM_DEFAULT, myhandler, NULL);

    /* From now on, myhandler() will be called on error */
    ...
}

void myhandler(void *param)
{
    char errmsg[IM_ERR_MSG_SIZE], errfunc[IM_ERR_FUNC_SIZE];

    /* Get the error message */
    imAppGetError(IM_ERR_MSG, errmsg);

    /* Get the name of the offending function and clear error items */
    imAppGetError(IM_ERR_FUNC + IM_ERR_RESET, errfunc);

    /* Print them out */
    printf("Error in %s(): <%s>\n", errfunc, errmsg);
}
```

imAppControl

Sync

Gen-LC

Synopsis Set an application-wide attribute.

Format void imAppControl(Item, Value)

long Item; Attribute to set
double Value; Attribute value

Description This function sets an application-wide attribute.

The **Item** parameter specifies the attribute, while the **Value** parameter specifies the value for this attribute. The table below lists those attributes that can be set, and their allowable values.

Item	Values	Meaning
IM_APP_TIMEOUT	any floating-point value > 0, or IM_INFINITE	The maximum time (in seconds) that the Host will wait for a synchronous function to return before generating an IM_ERR_TIMEOUT error and resuming execution. By default, IM_APP_TIMEOUT is set to 15 seconds.

imAppGetError

Sync

Gen-LC

Synopsis Check the application for errors.

Format **long imAppGetError(Item, ValuePtr)**

long Item; Error item to retrieve

void* ValuePtr; Address in which to return error item (or NULL)

Description This function returns error information about the application. The returned information pertains to the first error to occur in the application since error information about the application was last cleared. You clear error information by adding IM_ERR_RESET to the **Item** parameter.

Note that errors can only be detected for functions that have finished executing. Therefore, **imAppGetError()** might not detect errors caused by asynchronous functions, unless some synchronization is performed to ensure that these functions have finished executing.

The **Item** parameter specifies the error item to retrieve. It can be set to:

IM_ERR_CODE The error code.

IM_ERR_MSG The error message (maximum size of string: IM_ERR_MSG_SIZE bytes).

IM_ERR_FUNC The name of the offending function (maximum size of string: IM_ERR_FUNC_SIZE bytes).

IM_ERR_MSG_FUNC The error message and the name of the offending function (maximum size of string: IM_ERR_SIZE bytes).

To clear error information, add IM_ERR_RESET to the **Item** parameter (for example, IM_ERR_CODE + IM_ERR_RESET). Adding IM_ERR_RESET will simultaneously clear all error items (IM_ERR_CODE to IM_SUCCESS, and IM_ERR_MSG, IM_ERR_FUNC, and IM_ERR_MSG_FUNC to NULL). Note that you should only clear error information when retrieving the last required error item. This ensures that, at any time, all error items pertain to the same detected error.

The **ValuePtr** parameter specifies the address in which to return the error item. If you are retrieving IM_ERR_MSG, IM_ERR_FUNC, or IM_ERR_MSG_FUNC, **ValuePtr** should be the address of a character string;

if you are retrieving IM_ERR_CODE, **ValuePtr** should be the address of a long. Since **imAppGetError()** also returns the error code, **ValuePtr** can be set to NULL when you are retrieving the error code.

Return value The returned value is the error code.

Error code	Meaning
IM_SUCCESS	No error.
IM_ERR_BUFFER	Invalid buffer ID.
IM_ERR_DEVICE	Invalid device ID, or no such device.
IM_ERR_FILE	File access error.
IM_ERR_HALTED	Function halted by imThrHalt() .
IM_ERR_MEMORY	Insufficient memory to carry out the operation.
IM_ERR_NOT_PRESENT	Referenced item not present.
IM_ERR_OPCODE	Invalid opcode received.
IM_ERR_OSB	Invalid OSB ID.
IM_ERR_PARAMETER	Parameter (other than an ID-type parameter) invalid or unacceptable.
IM_ERR_RESTRICTION	Operation unable to execute due to a restriction.
IM_ERR_SYSTEM	No such system.
IM_ERR_TIMEOUT	Device unable to respond during timeout period.
IM_ERR_THREAD	Invalid thread ID.
IM_ERR_BUF_ATTRIBUTE	Unacceptable buffer attribute (size, data type, etc.).
IM_ERR_MISC	Miscellaneous error (the error message string will provide details on the cause of the error).

Example The following code checks for errors. Note that it reads the first items without clearing, then clears while reading the last item. This ensures that all error items pertain to the same detected error.

```
imAppGetError(IM_ERR_CODE, &errcode);
imAppGetError(IM_ERR_MSG, errmsg);
imAppGetError(IM_ERR_FUNC+IM_ERR_RESET, errfunc);
```

The following code checks for errors, then prints the error message. There is no need to clear if the call comes at the end of the application.

```
char Error[IM_ERR_SIZE];  
...  
if (imAppGetError(IM_ERR_MSG_FUNC, Error));  
    printf("%s\n", Error);
```

See also **imThrGetError()**. If your application uses only a single thread, **imThrGetError()** is generally a better alternative to **imAppGetError()** because it requires no explicit synchronization.

imAppInquire

Sync

Gen-LC

Synopsis Inquire about an application-wide attribute.

Format **long imAppInquire(Item, ValuePtr)**

long Item; Attribute about which to inquire
void* ValuePtr; Address of return value (or NULL)

Description This function inquires about an application-wide attribute.

The **Item** parameter specifies the attribute about which to inquire. It can be set to:

IM_APP_TIMEOUT Inquire about the timeout period.

The **ValuePtr** parameter specifies the address in which to return the value of the inquired attribute. The data type depends on the attribute being inquired. For IM_APP_TIMEOUT, **ValuePtr** must be the address of a double. Note that, since **imAppInquire()** also returns the value of the inquired attribute, **ValuePtr** can be set to NULL.

Return value The returned value is the value of the inquired attribute, cast to long if necessary.

imBinConvert

Async

PP

Multi-band

Synopsis Convert between integer and binary buffers.

Format **void imBinConvert(Thread, Src, Dst, Cond, Val1, Val2, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Cond; Conditional operator
 long Val1; Constant
 long Val2; Constant
 long OSB; OSB ID (or 0)

Description This function converts a buffer's data between integer and binary. For an integer to binary conversion, pixel values are converted to 1 if the specified condition is true, and to 0 otherwise. For a binary to integer conversion, 0's are converted to **Val1** and 1's to **Val2**.

The **Thread** parameter specifies the thread to which to send **imBinConvert()** for execution.

The **Src** parameter specifies the buffer to convert, while the **Dst** parameter specifies the buffer in which to place the results of the conversion. The buffer types are used to determine the type of conversion; therefore, one of these buffers must be a binary buffer, the other must be an integer buffer.

The **Cond** parameter specifies the condition with which to perform an integer to binary conversion. It can be set to:

IM_EQUAL	if equal to Val1 .
IM_NOT_EQUAL	if not equal to Val1 .
IM_LESS	if less than Val1 .
IM_LESS_OR_EQUAL	if less than or equal to Val1 .
IM_GREATER	if greater than Val1 .
IM_GREATER_OR_EQUAL	if greater than or equal to Val1 .
IM_IN_RANGE	if within Val1 to Val2 , inclusive.
IM_OUT_RANGE	if less than Val1 or greater than Val2 .

For a binary to integer conversion, the **Cond** parameter must be set to IM_DEFAULT.

The **Val1** and **Val2** parameters specify integer constants. For cases where **Val2** is not used, any value can be given for it.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imIntBinarize()**. Use **imIntBinarize()** instead of **imBinConvert()** if you are converting a buffer's data from integer to binary but want to store the binary data in an integer buffer (for example, as 0 and 0xff).

imBinCountDifference

Async

PP

Synopsis Count the differences between two binary buffers, or perform a binary event count on a single binary buffer.

Format **void imBinCountDifference(Thread, Src1, Src2, Result, OSB)**

long Thread; Thread ID
long Src1; First source buffer ID
long Src2; Second source buffer ID (or 0)
long Result; Result buffer ID
long OSB; OSB ID (or 0)

Description This function counts the differences between **Src1** and **Src2**, or the number of 1's in **Src1**, and writes the result to the IM_RES_NUM_DIFFERENCES field in the result buffer.

The **Src1** parameter specifies the first buffer with which to perform the operation. This must be a binary buffer.

The **Src2** parameter specifies the optional second buffer with which to perform the operation. This buffer must be a binary buffer as well. **Src2** can be set to 0, in which case the buffer is assumed to contain all 0's. As a result, the function counts the number of 1's in the first source buffer (**Src1**).

The **Result** parameter specifies the buffer in which to write the number of differences. Note that this buffer's size and data type are irrelevant, since the result is written to a field.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imBinMorphic

Async

PP

NOA

Multi-band

Synopsis Perform a morphological operation.

Format `void imBinMorphic(Thread, Src, Dst, Kernel, Op, Niter, Control, OSB)`

long Thread;	Thread ID
long Src;	Source buffer ID
long Dst;	Destination buffer ID
long Kernel;	Kernel buffer ID
long Op;	Type of operation to perform
long Niter;	Number of iterations
long Control;	Control buffer ID (or 0)
long OSB;	OSB ID (or 0)

Description This function performs a morphological operation on a binary image, using a specified structuring element (kernel). You can use your own kernel or a predefined kernel. In general, the predefined kernel will execute faster.

When you use your own kernel, you can control the center pixel of the kernel. In addition, any kernel value other than 0 or 1 is considered a "don't care" value; a "don't care" value is ignored during the operation.

If you want to thin or thicken with a series of different kernels (one applied after the other), you can provide a multi-band kernel. Each band of the kernel will be applied to the result of the previous one.

For binary thinning and thickening operations, **imBinMorphic()** checks if any pixel values have changed during the last iteration of the operation. If any have changed, the IM_RES_IDEMPOTENCE field of the destination buffer is set to a non-zero value (TRUE); otherwise, this field is set to 0 (FALSE). This field can later be read using **imBufGetField()**.

The **Thread** parameter specifies the thread to which to send **imBinMorphic()** for execution.

The **Src** parameter specifies the buffer on which to perform the operation. This must be a binary buffer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This buffer must be a binary buffer, except when you are performing binary template matching (in which case it can be binary, 8-bit integer, or 16-bit integer). Note that in-place operation is not supported for this function.

The **Kernel** parameter specifies the kernel with which to perform the operation. It can be set to the identifier of a kernel buffer or to `IM_3X3_RECT_1`, which is a predefined 3x3 kernel of all 1's.

If you are using your own kernel, the kernel buffer can be binary or of any integer type. The largest kernel supported by the NOA for binary operations is 32x32. If your operation does not use the NOA, then the largest kernel supported is 15x15. These kernel restrictions do not apply when performing binary pattern matching (**Op** parameter set to `IM_MATCH`).

Note that certain fields can be added to your kernel, to indicate its center position. These are listed below, with their default values. If these fields are not added to the kernel buffer, the default values are used.

Field	Values	Meaning
<code>IM_KER_CENTER_X</code>	0 – (Xsize-1) default: <code>int(Xsize-1)/2</code>	X coordinate of kernel center.
<code>IM_KER_CENTER_Y</code>	0 – (Ysize-1) default: <code>int(Ysize-1)/2</code>	Y coordinate of kernel center.

The **Op** parameter specifies the type of morphological operation to perform. It can be set to:

<code>IM_ERODE</code>	Erosion.
<code>IM_DILATE</code>	Dilation.
<code>IM_THIN</code>	Thinning.
<code>IM_THICK</code>	Thickening.
<code>IM_HIT_OR_MISS</code>	Hit-or-miss transformation.
<code>IM_MATCH</code>	Binary template matching (pattern matching).

For information on the algorithms used by the above operations, see the *Genesis Native Library User Guide*.

The **Niter** parameter specifies the number of times to apply the operation.

If you are using a multi-band kernel, note that the number of iterations is the number of passes through the entire kernel.

If you are performing a thickening or thinning operation, **Niter** can be set to IM_IDEMPOTENCE, which will cause the function to iterate until no more changes are produced (for a thinning, this typically occurs when the image has been reduced to its skeleton).

If you are performing a hit-or-miss transformation or binary template matching, **Niter** should be set to 1.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imBinMorphic()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_OVERSCAN	IM_TRANSPARENT	Use the pixels of the source buffer's parent buffer as the overscan pixels. If the source buffer is not a child buffer or if its parent buffer cannot provide values for the overscan pixels, the overscan pixels will be undefined, leading to unpredictable results.
	IM_REPLACE	Set the overscan pixels to a constant value. Specify the value with the IM_CTL_OVERSCAN_VAL field.
IM_CTL_OVERSCAN_VAL	0 or 1	Overscan replace value (used when IM_CTL_OVERSCAN is set to IM_REPLACE).
IM_CTL_THRESHOLD	any integer	When performing template matching with a binary destination buffer, 1 is written to the destination buffer if the result is greater than the indicated threshold value; otherwise, 0 is written.

To reduce the NOA set-up time, which can be significant (especially on small images), you can save some or all of the hardware register values in a control cache buffer using the control fields that follow. Doing so can speed up

processing time for a subsequent call to this function. The first call to this function will take slightly longer because the registers must be fully calculated and saved, but subsequent calls will be faster.

The increase in speed depends on the number of parameters that have changed since the setup information was saved. The increase is biggest when everything is the same (same buffers, kernel, control fields). The increase is slightly less if only the source and/or destination buffer addresses have changed (same size and type of buffer, same kernel, same control fields). This is useful if performing a double buffering operation. There is also some set-up time saved when only the kernel is the same as before (although buffers and control fields might have changed).

IM_CTL_CACHE_BUF	BufId	Use the specified buffer as the cache buffer in which to save the list of register values (or where to find them if they were saved previously). In this case, you will save a little time on the first call to this function. Note that you should allocate a 1-dimensional, 8-bit buffer of size IM_CACHE_BUF_SIZE.
	0	Automatically allocate the cache buffer in which to save the list of register values. The buffer ID will be returned to the IM_CTL_CACHE_BUF field.
IM_CTL_SETUP	IM_SAVE	If the cache buffer was given, save registers and all other information that might be useful later. Also perform the operation.
	IM_FASTEST	Assume everything is the same as when the setup was saved.
	IM_ADDRESS_ONLY	Assume everything except the source and destination addresses are the same as when the setup was saved.
	IM_SAME_KERNEL	Assume only the kernel is the same as when the setup was saved.

Note that, whether the cache buffer is allocated automatically or you allocate it yourself, you are responsible for freeing the buffer when you no longer need it.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note The NOA can only directly process binary image buffers that are byte-aligned (an image which starts on a byte boundary), and have a width that is a multiple of 16 pixels. If you pass a non-aligned source or destination buffer to **imBinMorphic()**, there are two possible outcomes, both of which will cause processing to be slower:

- The operation will be carried out by the 'C80 instead of the NOA, and the 15x15 maximum kernel size limitation (for operations that do not use the NOA) will apply.
- An aligned copy of the buffer will be made first, then processing will be done on the copy.

Note that this byte-aligned restriction can only apply to child buffers, since buffers are always initially allocated with the proper alignment, and with enough padding at the end of each line to satisfy the 16 pixel requirement.

Example For an example of thinning objects in an image to their skeleton, see *process.c* in Appendix B.

See also **imBinThin()**. Use **imBinThin()** instead of **imBinMorphic()** if you want to perform a fast binary thinning operation.

imBinThin

Async	PP	NOA	Multi-band
-------	----	-----	------------

Synopsis Perform a fast binary thinning operation.

Format **void imBinThin(Thread, Src, Dst, Niter, Control, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Niter; Number of iterations
 long Control; Control buffer ID (or 0)
 long OSB; OSB ID (or 0)

Description This function performs a fast binary thinning operation using a custom or a default connectivity map LUT.

Using this function is the fastest way to thin a binary image. The **Src** buffer is thinned for the specified number of iterations, and then the results are put in the **Dst** buffer.

The **Thread** parameter specifies the thread to which to send **imBinThin()** for execution.

The **Src** and **Dst** parameters specify the buffer on which to perform the operation and the buffer in which to place the results of the operation, respectively. Each must be a binary buffer.

Note that this function performs at optimal speed when both the source and destination buffers are aligned on 16-bit boundaries, and have a width which is a multiple of 16 pixels.

The **Niter** parameter specifies the number of times (iterations) this function is to be performed. This parameter can also be set to **IM_IDEMPOTENCE**, in which case the thinning operation will continue until the background and foreground pixels reach a steady state. As a result, the image will be reduced to its skeleton.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imBinThin()** are listed below, with default values in bold-face. Note that if no fields are specified, the default thinning method will be used.

Field	Value	Meaning
IM_CTL_OVERSCAN_VAL	0 or 1	Overscan replace value. Note that replace overscan is always used.
IM_CTL_LUT_BUF	BufID	Identifier of the connectivity map LUT buffer. The buffer must be 512 x 8-bit x N -band, where N is the number of sub-iterations (usually $N=2$).
	0	Use a default 3x3 connectivity map LUT.

Note that if you supply your own LUT, the format should be the same as described for **imIntConnectMap()**.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBinMorphic()**. It is also possible to use **imBinMorphic()** if you want to perform a binary thinning operation.

imIntConnectMap(). If you supply your own LUT buffer, follow the format conventions described in **imIntConnectMap()** for the **Lut** parameter.

imBinTriadic

Sync

PP

In-Place

Multi-band

Synopsis Perform a logical operation.**Format** **void imBinTriadic(Thread, SrcA, SrcB, SrcC, Dst, Op, OSB)**

long Thread; Thread ID
 long SrcA; First source buffer ID (or 0)
 long SrcB; Second source buffer ID (or 0)
 long SrcC; Third source buffer ID (or 0)
 long Dst; Destination buffer ID
 long Op; Type of operation to perform
 long OSB; OSB ID (or 0)

Description This function performs a logical operation on up to three binary images.

The PP ALU opcode specifies the type of operation to perform. Predefined opcodes exist for common operations but if one does not exist for the operation you want to perform, you can derive its opcode; for the details, see the *Genesis Native Library User Guide*.

The **Thread** parameter specifies the thread to which to send **imBinTriadic()** for execution.

The **SrcA**, **SrcB**, and **SrcC** parameters specify the buffers with which to perform the operation. These must be binary buffers. If you don't need all these parameters, set the unused ones to 0. Note that you must use **SrcA** before using **SrcB**, and **SrcB** before using **SrcC** (for example, if only two source buffers are needed, you must use **SrcA** and **SrcB**).

The **Dst** parameter specifies the buffer in which to place the results of the operation. This must be a binary buffer.

The **Op** parameter specifies the type of operation to perform. It can be set to the required PP ALU opcode or to one of the following predefined opcodes:

IM_PP_ZERO	Fill the destination buffer with 0.
IM_PP_ONE	Fill the destination buffer with 1.
IM_PP_PASS	pass SrcA
IM_PP_NOT	~SrcA
IM_PP_AND	SrcA & SrcB
IM_PP_OR	SrcA SrcB
IM_PP_XOR	SrcA ^ SrcB

IM_PP_XOR_XOR $\mathbf{SrcA} \wedge \mathbf{SrcB} \wedge \mathbf{SrcC}$
 IM_PP_NAND $\sim(\mathbf{SrcA} \& \mathbf{SrcB})$
 IM_PP_NOR $\sim(\mathbf{SrcA} \mid \mathbf{SrcB})$
 IM_PP_XNOR $\sim(\mathbf{SrcA} \wedge \mathbf{SrcB})$
 IM_PP_MERGE $(\mathbf{SrcA} \& \mathbf{SrcC}) \mid (\mathbf{SrcB} \& \sim\mathbf{SrcC})$

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imBlobAllocFeatureList

Sync

Synopsis Allocate a blob analysis feature list.

Format **void imBlobAllocFeatureList(Thread, FeatListPtr)**

long Thread; Thread ID
long* FeatListPtr; Address of feature list ID

Description This function allocates a feature list. A feature list holds the features to be calculated by **imBlobCalculate()**.

Note that, upon allocation, no features are in the feature list. You add features using **imBlobSelectFeature()**, **imBlobSelectFeret()**, and/or **imBlobSelectMoment()**.

The **Thread** parameter specifies the thread to which to send **imBlobAllocFeatureList()** for execution.

The **FeatListPtr** parameter specifies the address in which to return the feature list identifier. If the feature list could not be allocated, 0 is returned.

imBlobAllocResult

Sync

Synopsis Allocate a blob analysis result buffer.

Format **void imBlobAllocResult(Thread, ResultPtr)**

long Thread; Thread ID

long* ResultPtr; Address of result buffer ID

Description This function allocates a blob analysis result buffer. A result buffer is used to hold the results that are calculated by **imBlobCalculate()**.

Note that the control settings which affect blob calculations are also stored in result buffers. Upon allocation of a result buffer, these controls are set to default values; see **imBlobControl()** for a list of controls and their default values.

The **Thread** parameter specifies the thread to which to send **imBlobAllocResult()** for execution.

The **ResultPtr** parameter specifies the address in which to return the identifier of the blob analysis result buffer. If the result buffer could not be allocated, 0 is returned.

imBlobCalculate

Async

PP

Synopsis Perform a blob analysis calculation.

Format **void imBlobCalculate(Thread, IdentBuf, GreyBuf, FeatList, Result, Mode, OSB)**

long Thread;	Thread ID
long IdentBuf;	Blob identifier image ID
long GreyBuf;	Grayscale image ID (or 0)
long FeatList;	Feature list ID
long Result;	Result buffer ID
long Mode;	Mode of operation
long OSB;	OSB ID (or 0)

Description This function calculates features of an image's blobs. The feature list specifies the features to calculate. Binary features are calculated from the blob identifier image and grayscale features are calculated from the grayscale image. Results are written to the result buffer. The controls with which to perform a calculation are stored in the result buffer; if necessary, use **imBlobControl()** to change these controls.

Note that you can clear any existing results in the result buffer before writing results or you can accumulate results. You should clear results the first time you perform a calculation, as well as when you change the input images (either the identifiers of the images or the actual content of the images). Accumulating results is useful when you are using the same images but adding features with each calculation.

The **Thread** parameter specifies the thread to which to send **imBlobCalculate()** for execution.

The **IdentBuf** parameter specifies the blob identifier image. This can be a binary buffer, an 8-bit integer buffer, or a 16-bit integer buffer.

The **GreyBuf** parameter specifies the grayscale image. This can be an unsigned 8-bit integer buffer or an unsigned 16-bit integer buffer. If you are not calculating any grayscale features, set this parameter to 0.

The **FeatList** parameter specifies the feature list.

The **Result** parameter specifies the result buffer.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_CLEAR Clear results.

IM_NO_CLEAR Accumulate results.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imBlobControl

Async

Synopsis Change a blob analysis control setting.

Format **void imBlobControl(Thread, Result, Item, Value)**

long Thread; Thread ID
long Result; Result buffer ID
long Item; Item to set
double Value; Value for Item

Description This function changes a blob analysis control setting. These settings are stored in the blob analysis result buffer.

Note that changing a blob analysis control setting automatically re-includes any excluded blobs in the result buffer. Blobs are excluded using **imBlobSelect()**.

The **Thread** parameter specifies the thread to which to send **imBlobControl()** for execution.

The **Result** parameter specifies the blob analysis result buffer.

The **Item** parameter specifies the setting to change, while the **Value** parameter specifies the value for this setting. The table below lists blob analysis control settings, and their allowable values. The default values of these settings are in bold-face.

Item	Values	Meaning
IM_BLOB_IDENTIFICATION	IM_INDIVIDUAL	Produce separate results for each blob.
	IM_WHOLE_IMAGE	Group results for all blobs.
	IM_LABELLED	Group results for those blobs in the blob identifier image that have the same pixel value.
IM_BLOB_LATTICE	IM_8_CONNECTED	Consider diagonally adjacent pixels as touching.
	IM_4_CONNECTED	Don't consider diagonally adjacent pixels as touching.

IM_BLOB_PIXEL_ASPECT_RATIO	any floating-point value > 0 (default: 1.0)	Pixel aspect ratio to apply to blob calculations.
IM_BLOB_NUMBER_OF_FERETS	any integer between IM_MIN_FERETS and IM_MAX_FERETS, inclusive (default: 8)	Number of Feret diameters to check, for features that require multiple Feret diameters in order to be calculated. The Feret diameter is checked every $180^\circ/\text{IM_BLOB_NUMBER_OF_FERETS}$ starting at 0° .
IM_BLOB_FOREGROUND_VALUE	IM_NON_ZERO	Consider pixels in the blob identifier image with non-zero values as blob pixels.
	IM_ZERO	Consider pixels in the blob identifier image with zero values as blob pixels.
IM_BLOB_MAX_TIME	any floating-point value (default: 0.0)	Maximum time (in seconds) allowed for processing. If a non-zero value is specified, imBlobCalculate() will time out when the specified period expires. If imBlobCalculate() does time out, results will be invalid.
IM_BLOB_SAVE_RUNS	IM_ENABLE	Save run information.
	IM_DISABLE	Don't save run information. This reduces the amount of memory required for the result buffer and might increase the speed of imBlobCalculate() . However, you will not be able to use functions that require run information i.e. imBlobCopyRuns() , imBlobFill() , imBlobGetLabel() , imBlobGetRuns() , and imBlobLabel() .
IM_BLOB_TIME_SLICE	any floating-point value (default: 0.0)	The amount of processing time (in seconds) that imBlobCalculate() uses before it yields to other threads of equal priority. By default, imBlobCalculate() uses all of the master processor's time until it finishes.

imBlobCopyResult

Async

Synopsis Copy blob analysis results.

Format **void imBlobCopyResult(Thread, Result, Feature, Mode, DstBuf, OSB)**

long Thread;	Thread ID
long Result;	Result buffer ID
long Feature;	Feature to copy
long Mode;	Mode of operation
long DstBuf;	Destination buffer ID
long OSB;	OSB ID (or 0)

Description This function copies the results of a specified feature or of a predefined group of features, for all currently included blobs, into a buffer. Results that express units of measure are expressed in pixels or degrees. If the pixel aspect ratio is not equal to 1, results that represent a position or length are expressed in units of "pixel height".

Note that it is faster to retrieve a group of results at one time rather than retrieving the results individually.

Similar features appear in the same group. Some features (for example, the label value and the number of blobs) appear in all groups because you might need them no matter what other features you calculate. To save memory and reduce transfer time, features that can easily be derived from others are not included in any group. For the equations needed to derive certain features, see **imBlobSelectFeature()**.

When retrieving results of an individual feature, the required feature must first have been calculated, using **imBlobCalculate()**. When retrieving results for a group of features, only results for features you calculated will be valid. Values for features not calculated will be undefined, and no error messages will be generated.

The **Thread** parameter specifies the thread to which to send **imBlobCopyResult()** for execution.

The **Result** parameter specifies the result buffer containing the results to copy.

The **Feature** parameter specifies the #define of the required feature or the #define of the required group. For #defines of specific features, see **imBlobSelectFeature()**, **imBlobSelectFerret()**, and/or **imBlobSelectMoment()**.

Groups have the #defines IM_BLOB_GROUP1 to IM_BLOB_GROUP6 and are defined below, along with their data structure (when you retrieve results for a predefined group, the results are stored in a specific data structure). Note that the structure member names are exactly the same as the corresponding feature name #defines, with the IM_BLOB_ prefix removed. In order to save memory and reduce transfer time to the Host, each feature is stored in the smallest data type that can hold it. For example, integer results are returned as 16-bit values if possible, and floating-point values are returned as 32-bit single precision values. You must make sure that your compiler does not add any padding to the structure to change the alignment of structure members. However, the structure has been defined so that most compilers will not attempt to change the alignment.

```
/* IM_BLOB_GROUP1 Basic binary features */
typedef struct
{
    unsigned short number_of_blobs; /* Same as imBlobGetNumber() */
    unsigned short label_value;
    unsigned long area;
    unsigned short box_x_min;
    unsigned short box_y_min;
    unsigned short box_x_max;
    unsigned short box_y_max;
    unsigned short number_of_holes;
    unsigned short number_of_runs;
    float perimeter;
    float length;
    float breadth;
    float center_of_gravity_x;
    float center_of_gravity_y;
} IM_BLOB_GROUP1_ST;

/* IM_BLOB_GROUP2 Less common features */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    unsigned short x_min_at_y_min;
    unsigned short x_max_at_y_max;
    unsigned short y_min_at_x_max;
    unsigned short y_max_at_x_min;
    unsigned short intercept_0;
    unsigned short intercept_45;
    unsigned short intercept_90;
    unsigned short intercept_135;
} IM_BLOB_GROUP2_ST;
```

```

/* IM_BLOB_GROUP3  Binary second moments */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    float moment_x1_y1;
    float moment_x2_y0;
    float moment_x0_y2;
    float moment_central_x1_y1;
    float moment_central_x2_y0;
    float moment_central_x0_y2;
} IM_BLOB_GROUP3_ST;

/* IM_BLOB_GROUP4  Features derived from multiple feret diameters */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    float feret_min_diameter;
    float feret_min_angle;
    float feret_max_diameter;
    float feret_max_angle;
    float feret_mean_diameter;
    float convex_perimeter;
} IM_BLOB_GROUP4_ST;

/* IM_BLOB_GROUP5  Basic grayscale features */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    unsigned long sum_pixel;
    unsigned short min_pixel;
    unsigned short max_pixel;
    float sum_pixel_squared;
    float center_of_gravity_x;
    float center_of_gravity_y;
} IM_BLOB_GROUP5_ST;

/* IM_BLOB_GROUP6  Grayscale second moments */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    float moment_x1_y1;
    float moment_x2_y0;
    float moment_x0_y2;
    float moment_central_x1_y1;
    float moment_central_x2_y0;
    float moment_central_x0_y2;
} IM_BLOB_GROUP6_ST;

```

The **Mode** parameter specifies whether to copy the binary or grayscale result of a feature, for features that have both a binary and grayscale result. This parameter applies only when copying an individual feature, not when copying a group of features. The **Mode** parameter can be set to:

IM_BINARY Copy the binary result.

IM_GRAYSCALE Copy the grayscale result.

For features that do not have both a binary and grayscale result (or for groups of features), set the **Mode** parameter to IM_DEFAULT.

The **DstBuf** parameter specifies the buffer in which to copy results. This must be an on-board, one-dimensional buffer. It can be of any data type, but should be at least as big as the number of included blobs in the result buffer. Note that you can determine the number of included blobs using **imBlobGetNumber()**.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBlobGetResult()**. The **imBlobGetResult()** function can transfer blob analysis results to Host memory.

imBlobCopyRuns

Async

Synopsis Copy run information about a blob.

Format **void imBlobCopyRuns(Thread, Result, Label, XBuf, YBuf, LenBuf, OSB)**

long Thread;	Thread ID
long Result;	Result buffer ID
long Label;	Label value of the blob
long XBuf;	ID of buffer to place x coordinates (or 0)
long YBuf;	ID of buffer to place y coordinates (or 0)
long LenBuf;	ID of buffer to place length of each run (or 0)
long OSB;	OSB ID (or 0)

Description This function copies run information about a specified blob into a buffer. A run is a horizontal sequence of consecutive blob pixels. You can copy the x and y coordinate of the start of each run, as well as the length of each run.

Note that, before you can copy run information about a blob, you must calculate the number of runs feature (IM_BLOB_NUMBER_OF_RUNS), using **imBlobCalculate()**.

The **Thread** parameter specifies the thread to which to send **imBlobCopyRuns()** for execution.

The **Result** parameter specifies the result buffer from which to copy run information.

The **Label** parameter specifies the label value of the blob whose run information you wish to copy. Note that the label value of a blob can be obtained by using **imBlobGetLabel()**, or by transferring the IM_BLOB_LABEL_VALUE feature using **imBlobGetResult()**.

The **XBuf** parameter specifies the buffer in which to place the x coordinate of the start of each run. This must be an on-board, one-dimensional buffer. It can be of any integer data type but must be at least as big as the number of runs in the required blob. You can inquire about the number of runs in a blob using **imBlobGetResult()** or **imBlobGetResultSingle()**. Note that this parameter can be set to 0, in which case the x coordinates are not copied.

The **YBuf** parameter specifies the buffer in which to place the y coordinate of the start of each run. This must be an on-board, one-dimensional buffer. It can be of any integer data type but must be at least as big as the number of runs in the required blob. You can inquire about the number of runs in a blob using **imBlobGetResult()** or **imBlobGetResultSingle()**. Note that this parameter can be set to 0, in which case the y coordinates are not copied.

The **LenBuf** parameter specifies the buffer in which to place the length of each run. This must be an on-board, one-dimensional buffer. It can be of any integer data type but must be at least as big as the number of runs in the required blob. You can inquire about the number of runs in a blob using **imBlobGetResult()** or **imBlobGetResultSingle()**. Note that this parameter can be set to 0, in which case the lengths are not copied.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note Run results are in raw pixels and are not affected by the pixel aspect ratio.

See also **imBlobGetRuns()**. The **imBlobGetRuns()** function transfers run information about a blob into Host memory.

imBlobFill

Async

Synopsis Draw filled blobs into a buffer.

Format **void imBlobFill(Thread, Result, DstBuf, Mode, Value, OSB)**

long Thread;	Thread ID
long Result;	Result buffer ID
long DstBuf;	Destination buffer ID
long Mode;	Mode of operation
long Value;	Fill value
long OSB;	OSB ID (or 0)

Description This function draws selected blobs of a result buffer into a destination buffer, filling them with a specified value. The blobs are drawn into the same positions as they occupy in the blob identifier image associated with the result buffer. They are selected for drawing based on their status (included or excluded) in the result buffer. You exclude (or re-include) blobs using **imBlobSelect()**.

Note that this function can be used to remove unwanted blobs in a blob identifier image, by filling them with the background value and using the identifier image (or a copy of it) as the destination buffer.

The **Thread** parameter specifies the thread to which to send **imBlobFill()** for execution.

The **Result** parameter specifies the result buffer.

The **DstBuf** parameter specifies the buffer in which to draw the blobs. This must be an 8-bit or 16-bit integer buffer.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_INCLUDED_BLOBS	Draw included blobs.
IM_EXCLUDED_BLOBS	Draw excluded blobs.
IM_ALL_BLOBS	Draw all blobs, regardless of their status.

The **Value** parameter specifies the value with which to fill the blobs drawn into the destination buffer.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imBlobFree

Async

Synopsis Free a blob analysis result buffer or feature list.

Format **void imBlobFree(Thread, ResultOrFeatList)**

long Thread; Thread ID

long ResultOrFeatList; Result buffer or feature list ID

Description This function deallocates a blob analysis result buffer or feature list.

The **Thread** parameter specifies the thread to which to send **imBlobFree()** for execution.

The **ResultOrFeatList** parameter specifies the result buffer or feature list to deallocate.

imBlobGetLabel

Sync

Synopsis Get the label value of a blob at a specified position.

Format **long imBlobGetLabel(Thread, Result, X, Y, LabelPtr)**

long Thread;	Thread ID
long Result;	Result buffer ID
long X;	X coordinate of the blob
long Y;	Y coordinate of the blob
long* LabelPtr;	Address of label value (or NULL)

Description This function determines the label value of a blob at a specified position in an identifier image. The identifier image is the one that was given to **imBlobCalculate()** for the specified result buffer.

The **Thread** parameter specifies the thread to which to send **imBlobGetLabel()** for execution.

The **Result** parameter specifies the result buffer associated with the required identifier image.

The **X** and **Y** parameters specify the x and y coordinates of a pixel within the identifier image.

The **LabelPtr** parameter specifies the address of the user-supplied variable in which to place the label value of the specified blob. Since **imBlobGetLabel()** also returns this value, **LabelPtr** can be set to NULL.

Return value The returned value is the label value of the specified blob if there is a blob at the specified position; 0 if there is no blob at the specified position.

imBlobGetNumber

Sync

Synopsis Get the number of currently included blobs.

Format **long imBlobGetNumber(Thread, Result, NumPtr)**

long Thread; Thread ID

long Result; Result buffer ID

long* NumPtr; Address of count (or NULL)

Description This function determines the number of currently included blobs in a blob analysis result buffer. Included blobs are those that are included in blob calculations. You exclude blobs from blob calculations (or re-include them) using **imBlobSelect()**.

Note that, before retrieving results using **imBlobCopyResult()** or **imBlobGetResult()**, **imBlobGetNumber()** can be used to determine how much to memory to allocate for the results. However, if you are sure you have allocated enough memory and are retrieving results for a group of features, you do not need to call **imBlobGetNumber()**, since the number of included blobs is part of all feature groups.

The **Thread** parameter specifies the thread to which to send **imBlobGetNumber()** for execution.

The **Result** parameter specifies the result buffer. Note that this buffer must already have been used in a call to **imBlobCalculate()**.

The **NumPtr** parameter specifies the address of the user-supplied variable in which to place the number of blobs. Since **imBlobGetNumber()** also returns this value, **NumPtr** can be set to NULL.

Return value The returned value is the number of included blobs.

imBlobGetResult

Sync

Synopsis Transfer blob analysis results to Host memory.

Format **void imBlobGetResult(Thread, Result, Feature, Mode, ArrayPtr)**

long Thread;	Thread ID
long Result;	Result buffer ID
long Feature;	Type of feature required
long Mode;	Mode of operation
void* ArrayPtr;	Address of array

Description This function transfers the results of a specified feature or of a predefined group of features, for all currently included blobs, to an array in Host memory. Results that express units of measure are expressed in pixels or degrees. If the pixel aspect ratio is not equal to 1, results that represent a position or length are expressed in units of "pixel height".

Note that it is faster to retrieve a group of results at one time rather than retrieving the results individually.

Similar features appear in the same group. Some features (for example, the label value and the number of blobs) appear in all groups because you might need them no matter what other features you calculate. To save memory and reduce transfer time, features that can easily be derived from others are not included in any group. For the equations needed to derive certain features, see **imBlobSelectFeature()**.

When retrieving results of an individual feature, the required feature must first have been calculated, using **imBlobCalculate()**. When retrieving results for a group of features, only results for features you calculated will be valid. Values for features not calculated will be undefined, and no error messages will be generated.

The **Thread** parameter specifies the thread to which to send **imBlobGetResult()** for execution.

The **Result** parameter specifies the result buffer containing the results to transfer.

The **Feature** parameter specifies the #define of the required feature or the #define of the required group. For #defines of specific features, see **imBlobSelectFeature()**, **imBlobSelectFerret()**, and/or **imBlobSelectMoment()**.

Groups have the #defines IM_BLOB_GROUP1 to IM_BLOB_GROUP6 and are defined below, along with their data structure (when you retrieve results for a predefined group, the results are stored in a specific data structure). Note that the structure member names are exactly the same as the corresponding feature name #defines, with the IM_BLOB_ prefix removed. In order to save memory and reduce transfer time to the Host, each feature is stored in the smallest data type that can hold it. For example, integer results are returned as 16-bit values if possible, and floating-point values are returned as 32-bit single precision values. You must make sure that your compiler does not add any padding to the structure to change the alignment of structure members. However, the structure has been defined so that most compilers will not attempt to change the alignment.

```
/* IM_BLOB_GROUP1 Basic binary features */
typedef struct
{
    unsigned short number_of_blobs; /* Same as imBlobGetNumber() */
    unsigned short label_value;
    unsigned long area;
    unsigned short box_x_min;
    unsigned short box_y_min;
    unsigned short box_x_max;
    unsigned short box_y_max;
    unsigned short number_of_holes;
    unsigned short number_of_runs;
    float perimeter;
    float length;
    float breadth;
    float center_of_gravity_x;
    float center_of_gravity_y;
} IM_BLOB_GROUP1_ST;

/* IM_BLOB_GROUP2 Less common features */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    unsigned short x_min_at_y_min;
    unsigned short x_max_at_y_max;
    unsigned short y_min_at_x_max;
    unsigned short y_max_at_x_min;
    unsigned short intercept_0;
    unsigned short intercept_45;
    unsigned short intercept_90;
    unsigned short intercept_135;
} IM_BLOB_GROUP2_ST;
```

```

/* IM_BLOB_GROUP3 Binary second moments */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    float moment_x1_y1;
    float moment_x2_y0;
    float moment_x0_y2;
    float moment_central_x1_y1;
    float moment_central_x2_y0;
    float moment_central_x0_y2;
} IM_BLOB_GROUP3_ST;

/* IM_BLOB_GROUP4 Features derived from multiple feret diameters */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    float feret_min_diameter;
    float feret_min_angle;
    float feret_max_diameter;
    float feret_max_angle;
    float feret_mean_diameter;
    float convex_perimeter;
} IM_BLOB_GROUP4_ST;

/* IM_BLOB_GROUP5 Basic grayscale features */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    unsigned long sum_pixel;
    unsigned short min_pixel;
    unsigned short max_pixel;
    float sum_pixel_squared;
    float center_of_gravity_x;
    float center_of_gravity_y;
} IM_BLOB_GROUP5_ST;

/* IM_BLOB_GROUP6 Grayscale second moments */
typedef struct
{
    unsigned short number_of_blobs;
    unsigned short label_value;
    float moment_x1_y1;
    float moment_x2_y0;
    float moment_x0_y2;
    float moment_central_x1_y1;
    float moment_central_x2_y0;
    float moment_central_x0_y2;
} IM_BLOB_GROUP6_ST;

```

The **Mode** parameter specifies whether to transfer the binary or grayscale result of a feature, for features that have both a binary and grayscale result. This parameter applies only when transferring an individual feature, not when transferring a group of features. The **Mode** parameter can be set to:

IM_BINARY Transfer the binary result.

IM_GRAYSCALE Transfer the grayscale result.

For features that do not have both a binary and grayscale result (or for groups of features), set the **Mode** parameter to IM_DEFAULT.

The **ArrayPtr** parameter specifies the address of the user-supplied array in which to place the results. By default, for individual features, results are returned as type double. If you want results returned as a type other than double, combine the #define of the required feature with the required type: IM_TYPE_CHAR, IM_TYPE_SHORT, IM_TYPE_LONG, or IM_TYPE_FLOAT (for example, IM_BLOB_AREA+IM_TYPE_LONG). For groups of features, the **ArrayPtr** parameter should be the address of an array of the appropriate type of data structure.

Note that the array should be at least as big as the number of included blobs in the result buffer. You can determine the number of included blobs using **imBlobGetNumber()**.

See also **imBlobCopyResult()**. If for some reason you need to use several calls to **imBlobGetResult()** to transfer results, it might be more efficient to copy them to an on-board buffer (using multiple calls to **imBlobCopyResult()**), and then transfer them to the Host all at the same time (using **imBufGet()**). This is because **imBlobCopyResult()**, being an asynchronous function, is more efficient than **imBlobGetResult()**.

imBlobGetResultSingle

Sync

Synopsis Read the feature value of a single blob.

Format **void imBlobGetResultSingle(Thread, Result, Label, Feature, Mode, ValuePtr)**

long Thread;	Thread ID
long Result;	Result buffer ID
long Label;	Label value
long Feature;	Type of feature required
long Mode;	Mode of operation
void* ValuePtr;	Address of feature value

Description This function obtains the result of a specified feature of a specified blob.

Note that the required feature must first have been calculated, using **imBlobCalculate()**. Results that express units of measure are expressed in pixels or degrees. If the pixel aspect ratio is not equal to 1, results that represent a position or length are expressed in units of "pixel height".

The **Thread** parameter specifies the thread to which to send **imBlobGetResultSingle()** for execution.

The **Result** parameter specifies the result buffer containing the results to transfer.

The **Label** parameter specifies the label value of the specified blob. Note that the label value of a blob can be obtained using **imBlobGetLabel()**.

The **Feature** parameter specifies the #define of the required feature. For a list of features and their #defines, see **imBlobSelectFeature()**, **imBlobSelectFerret()**, and/or **imBlobSelectMoment()**.

The **Mode** parameter specifies whether to transfer the binary or grayscale result of a feature, for features that have both a binary and grayscale result. The **Mode** parameter can be set to:

IM_BINARY	Transfer the binary result.
IM_GRAYSCALE	Transfer the grayscale result.

For features that do not have both a binary and grayscale result, set the **Mode** parameter to IM_DEFAULT.

The **ValuePtr** parameter specifies the address of the user-supplied variable in which to place the result. By default, the result is returned as type double. If you want the result returned as a type other than double, combine the #define of the required feature with the required type: IM_TYPE_CHAR, IM_TYPE_SHORT, IM_TYPE_LONG, or IM_TYPE_FLOAT (for example, IM_BLOB_AREA+IM_TYPE_LONG).

imBlobGetRuns

Sync

Synopsis Transfer run information about a blob into Host memory.

Format **void imBlobGetRuns(Thread, Result, Label, Type, XPtr, YPtr, LenPtr)**

long Thread;	Thread ID
long Result;	Result buffer ID
long Label;	Label value of the blob
long Type;	Type of arrays in which run information will be placed
void* XPtr;	Address of array to place x-coordinates (or NULL)
void* YPtr;	Address of array to place y-coordinates (or NULL)
void* LenPtr;	Address of array to place length of each run (or NULL)

Description This function transfers run information about a specified blob to an array in Host memory. A run is a horizontal sequence of consecutive blob pixels. You can transfer the x and y coordinate of the start of each run, as well as the length of each run.

Note that, before you can copy run information about a blob, you must calculate the number of runs feature (IM_BLOB_NUMBER_OF_RUNS), using **imBlobCalculate()**.

The **Thread** parameter specifies the thread to which to send **imBlobGetRuns()** for execution.

The **Result** parameter specifies the result buffer from which to transfer run information.

The **Label** parameter specifies the label value of the blob whose run information you wish to transfer. Note that the label value of a blob can be obtained by using **imBlobGetLabel()**, or by transferring the IM_BLOB_LABEL_VALUE feature using **imBlobGetResult()**.

The **Type** parameter specifies the type of the arrays in which run information will be placed. It can be set to:

IM_TYPE_CHAR	Type char.
IM_TYPE_SHORT	Type short.
IM_TYPE_LONG	Type long.

The **XPtr** parameter specifies the user-supplied array in which to place the x coordinate of the start of each run. This array should be at least as big as the number of runs in the required blob. You can inquire about the number of runs in a blob using **imBlobGetResult()** or **imBlobGetResultSingle()**. Note that this parameter can be set to NULL, in which case the x coordinates are not transferred.

The **YPtr** parameter specifies the user-supplied array in which to place the y coordinate of the start of each run. This array should be at least as big as the number of runs in the required blob. You can inquire about the number of runs in a blob using **imBlobGetResult()** or **imBlobGetResultSingle()**. Note that this parameter can be set to NULL, in which case the y coordinates are not transferred.

The **LenPtr** parameter specifies the user-supplied array in which to place the length of each run. This array should be at least as big as the number of runs in the required blob. You can inquire about the number of runs in a blob using **imBlobGetResult()** or **imBlobGetResultSingle()**. Note that this parameter can be set to NULL, in which case the lengths are not transferred.

Note Run results are in raw pixels and are not affected by the pixel aspect ratio.

See also **imBlobCopyRuns()**. The **imBlobCopyRuns()** function copies run information about a blob into a buffer.

imBlobInquire

Sync

Synopsis Inquire about a blob analysis control setting.

Format **long imBlobInquire(Thread, Result, Item, ValuePtr)**

long Thread; Thread ID
 long Result; Result buffer ID
 long Item; Item to inquire
 void* ValuePtr; Address of return value (or NULL)

Description This function inquires about a blob analysis control setting. Note that these settings are stored in blob analysis result buffers.

The **Thread** parameter specifies the thread to which to send **imBlobInquire()** for execution.

The **Result** parameter specifies the blob analysis result buffer.

The **Item** parameter specifies the control setting about which to inquire. It can be set to:

IM_BLOB_IDENTIFICATION	Whether to produce separate results for each blob (IM_INDIVIDUAL), group results for all blobs (IM_WHOLE_IMAGE), or group results for those blobs in the blob identifier image that have the same pixel value (IM_LABELLED).
IM_BLOB_LATTICE	Whether diagonally adjacent pixels are considered touching (IM_8_CONNECTED) or not (IM_4_CONNECTED).
IM_BLOB_PIXEL_ASPECT_RATIO	Pixel aspect ratio to apply to blob calculations.
IM_BLOB_NUMBER_OF_FERETS	Number of Feret diameters to check, for features that require multiple Feret diameters in order to be calculated.
IM_BLOB_MAX_LABEL	The maximum label value of a blob in the result buffer.
IM_BLOB_FOREGROUND_VALUE	Whether pixels in the blob identifier image with non-zero values (IM_NON_ZERO) or with zero values (IM_ZERO) are considered blob pixels.

IM_BLOB_TIMEOUT	Whether imBlobCalculate() timed out (TRUE) or completed normally (FALSE). imBlobCalculate() will time out if the specified maximum time allowed for processing is exceeded.
IM_BLOB_MAX_TIME	Maximum time (in seconds) allowed for processing. If this value is zero, imBlobCalculate() will always run to completion.
IM_BLOB_SAVE_RUNS	Whether to save run information (IM_ENABLE) or not (IM_DISABLE). If run information is not saved, you cannot use functions that require run information i.e. imBlobCopyRuns() , imBlobFill() , imBlobGetLabel() , imBlobGetRuns() , and imBlobLabel() .
IM_BLOB_TIME_SLICE	The amount of processing time (in seconds) that imBlobCalculate() uses before it yields to other threads of equal priority. If this value is zero, imBlobCalculate() uses all of the master processor's time until it finishes.

The **ValuePtr** parameter specifies the address in which to return the value of the inquired control setting. By default, the value is returned as type double. To return the value as type long, combine the specified control setting with IM_TYPE_LONG (for example, IM_BLOB_IDENTIFICATION+IM_TYPE_LONG).

Note that, since **imBlobInquire()** also returns the value of the inquired setting, **ValuePtr** can be set to NULL.

Return value The returned value is the value of the inquired control setting, cast to long if necessary.

imBlobLabel

Async

Synopsis Draw labelled blobs into a buffer.

Format **void imBlobLabel(Thread, Result, DstBuf, Mode, OSB)**

long Thread;	Thread ID
long Result;	Result buffer ID
long DstBuf;	Destination buffer ID
long Mode;	Mode of operation
long OSB;	OSB ID (or 0)

Description This function draws blobs of a result buffer into a destination buffer, labelling them with their label value. The blobs are drawn into the same positions as they occupy in the blob identifier image associated with the result buffer. Blobs that are deleted from the result buffer are not drawn.

Note that label values must first have been generated for the blobs, using **imBlobCalculate()**.

The **Thread** parameter specifies the thread to which to send **imBlobLabel()** for execution.

The **Result** parameter specifies the result buffer from which to draw the labelled image.

The **DstBuf** parameter specifies the buffer in which to draw the labelled image. This buffer must be an 8-bit or 16-bit integer buffer, and must be at least the same size as the original image. In addition, it must be 16 bits deep if the maximum label value exceeds 255. To determine the maximum label value, use **imBlobInquire()**. Note that the number of blobs cannot tell you the maximum label value, since label values are not necessarily contiguous.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_CLEAR	Clear the destination buffer before drawing into it (background pixels will be set to the value 0).
IM_NO_CLEAR	Don't clear the destination buffer before drawing into it (background pixels will be unchanged).

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imBlobSelect

Async

Synopsis Exclude or delete blobs.

Format **void imBlobSelect(Thread, Result, Operation, Feature, Mode, Cond, Low, High)**

long Thread;	Thread ID
long Result;	Result buffer ID
long Operation;	Mode of operation
long Feature;	Feature to be used for selection
long Mode;	Mode of operation
long Cond;	Conditional operator
double Low;	Constant
double High;	Constant

Description This function filters (excludes or deletes) unwanted blobs from a blob analysis result buffer. It can also be used to re-include excluded blobs. Note that blobs that are excluded are ignored during subsequent calculations and result retrieval but can be re-included (either by calling this function again or by calling **imBlobControl()**). Blobs that are deleted are removed completely from the result buffer and cannot be re-included. However, deleted blobs are not removed from the blob identifier image (to remove blobs from a blob identifier image, use **imBlobFill()**).

Note that blobs are excluded, deleted, or re-included based on the result of a specified feature. Therefore, the required feature must first have been calculated, using **imBlobCalculate()**.

The **Thread** parameter specifies the thread to which to send **imBlobSelect()** for execution.

The **Result** parameter specifies the result buffer from which to exclude, delete, or re-include blobs.

The **Operation** parameter specifies whether to re-include, exclude, or delete blobs. It can be set to:

IM_INCLUDE	Re-include blobs that meet the specified condition.
IM_EXCLUDE	Exclude blobs that meet the specified condition.

IM_INCLUDE_ONLY	Include blobs that meet the specified condition, and exclude all blobs that do not meet the specified condition.
IM_EXCLUDE_ONLY	Exclude blobs that meet the specified condition, and re-include blobs that do not meet the specified condition.
IM_DELETE	Delete blobs that meet the specified condition.

The **Feature** parameter specifies the #define of the feature with which to exclude, delete, or re-include blobs. For a list of features and their #defines, see **imBlobSelectFeature()**, **imBlobSelectFeret()**, and/or **imBlobSelectMoment()**.

The **Mode** parameter specifies whether to use the binary or grayscale result to exclude, delete, or re-include blobs, for features that have both a binary and grayscale result. The **Mode** parameter can be set to:

IM_BINARY	Use the binary result.
IM_GRAYSCALE	Use the grayscale result.

For features that do not have both a binary and grayscale result, set the **Mode** parameter to IM_DEFAULT.

The **Cond** parameter specifies the condition with which to exclude, delete, or re-include blobs. It can be set to:

IM_OUT_RANGE	if the result of the specified feature is less than Low or greater than High .
IM_IN_RANGE	if the result of the specified feature is within Low to High , inclusive.
IM_EQUAL	if the result of the specified feature is equal to Low .
IM_NOT_EQUAL	if the result of the specified feature is not equal to Low .
IM_GREATER	if the result of the specified feature is greater than Low .
IM_LESS	if the result of the specified feature is less than Low .
IM_GREATER_OR_EQUAL	if the result of the specified feature is greater than or equal to Low .
IM_LESS_OR_EQUAL	if the result of the specified feature is less than or equal to Low .

The **Low** and **High** parameters specify constants. For cases where **High** is not used, any value can be given for it.

imBlobSelectFeature

Async

Synopsis Select a feature for calculation.

Format **void imBlobSelectFeature(Thread, FeatList, Feature, Mode)**

long Thread; Thread ID
 long FeatList; Feature list ID
 long Feature; Feature to be selected
 long Mode; Mode of operation

Description This function selects a feature for calculation by **imBlobCalculate()**. Note that features must be added to the feature list which is passed to **imBlobCalculate()**. A feature list is allocated using **imBlobAllocFeatureList()**.

The **Thread** parameter specifies the thread to which to send **imBlobSelectFeature()** for execution.

The **FeatList** parameter specifies the feature list.

The **Feature** parameter specifies the feature. It can be set to:

IM_BLOB_LABEL_VALUE	The label value of a blob. This is a positive integer (≥ 1) that is unique for each blob. This feature is always calculated; you do not need to select it.
IM_BLOB_AREA	The number of pixels in a blob (holes are not counted).
IM_BLOB_PERIMETER	The total length of edges in a blob (including the edges of any holes), with an allowance made for the staircase effect that is produced when diagonal edges are digitized (inside corners are counted as 1.414, rather than 2.0). A single pixel blob (area = 1) has a perimeter of 4.0.
IM_BLOB_BOX_X_MIN, IM_BLOB_BOX_Y_MIN, IM_BLOB_BOX_X_MAX, IM_BLOB_BOX_Y_MAX	The coordinates of the extreme left, top, right, and bottom pixels, respectively, of a blob.

IM_BLOB_FIRST_POINT_X,
IM_BLOB_FIRST_POINT_Y

The x coordinate is that of the left-most pixel on the top-most line of the blob; the y coordinate is that of the top-most line of the blob. Together, these define a unique point for each blob, that is always on the perimeter of the blob.

IM_BLOB_FERET_X,
IM_BLOB_FERET_Y

These are the dimensions of the minimum bounding box of a blob in the horizontal and vertical directions (respectively); that is, $\text{IM_BLOB_BOX_X_MAX} - \text{IM_BLOB_BOX_X_MIN} + 1$, and similarly for the y direction.

IM_BLOB_FERET_MIN_DIAMETER

The smallest Feret diameter found after checking a certain number of angles. More angles will give a more accurate result, but will take longer to calculate. You specify the number of angles to check using **imBlobControl()** (the default is 8). Note that this feature is not very accurate for long thin blobs. If you want an estimate of the width of a long thin blob, it is better to use **IM_BLOB_BREADTH**.

IM_BLOB_FERET_MIN_ANGLE

The angle at which the minimum Feret diameter is found. This value is in degrees. Positive values indicate a counter-clockwise displacement from the positive x-axis; negative values indicate a clockwise displacement from the positive x-axis.

IM_BLOB_FERET_MAX_DIAMETER

The largest Feret diameter found after checking a certain number of angles. More angles will give a more accurate result, but will take longer to calculate. You specify the number of angles to check using **imBlobControl** (the default is 8). Note that the maximum Feret diameter is not very sensitive to the number of angles; 8 usually gives an accurate result.

IM_BLOB_FERET_MAX_ANGLE

The angle at which the maximum Feret diameter is found. This value is in degrees. Positive values indicate a counter-clockwise displacement from the positive x-axis; negative values indicate a clockwise displacement from the positive x-axis.

IM_BLOB_FERET_MEAN_DIAMETER	<p>The average Feret diameter after checking a certain number of angles. You specify the number of angles to check using imBlobControl (the default is 8). More angles will give a more accurate result, but will take longer to calculate.</p>
IM_BLOB_FERET_ELONGATION	<p>Equal to:</p> $\frac{\text{IM_BLOB_FERET_MAX_DIAMETER}}{\text{IM_BLOB_FERET_MIN_DIAMETER}}$ <p>It is accurate for reasonably compact blobs, but becomes less accurate for long thin blobs (because IM_BLOB_FERET_MIN_DIAMETER becomes less accurate). For long thin blobs, it is better to use IM_BLOB_ELONGATION.</p>
IM_BLOB_CONVEX_PERIMETER	<p>An approximation of the perimeter of a blob's convex hull. It is derived from several Feret diameters. You specify the number of angles at which to calculate Ferets using imBlobControl (the default is 8). More angles will give a more accurate result, but will take longer to calculate.</p>
IM_BLOB_X_MIN_AT_Y_MIN, IM_BLOB_X_MAX_AT_Y_MAX, IM_BLOB_Y_MIN_AT_X_MAX, IM_BLOB_Y_MAX_AT_X_MIN IM_BLOB_COMPACTNESS	<p>These values, together with the four box coordinates, give four contact points on the convex perimeter of the blob.</p> <p>A measure of how close pixels in the blob are to one another. It is equal to:</p> $\frac{p^2}{(4\pi A)}$ <p>where A = the area and p = the perimeter of the blob. Note that a circle has the minimum compactness value (1.0); more convoluted shapes have higher compactness values.</p>
IM_BLOB_NUMBER_OF_HOLES	<p>The number of holes in a blob. Holes that intersect the edge of the image are not counted (they might not be holes). This value is equal to 1 - IM_BLOB_EULER_NUMBER and is therefore a true hole count only when results are not grouped for blobs i.e. when the IM_BLOB_IDENTIFICATION control setting is set to IM_INDIVIDUAL.</p>
IM_BLOB_NUMBER_OF_RUNS	<p>The total number of runs in a blob. A run is a horizontal string of consecutive blob pixels.</p>

IM_BLOB_ROUGHNESS

A measure of the unevenness or irregularity of a blob's surface. It is equal to:

$$\frac{\text{IM_BLOB_PERIMETER}}{\text{IM_BLOB_CONVEX_PERIMETER}}$$

The minimum roughness value is 1.0. Jagged blobs have much higher roughness values because their perimeters are much larger than their convex perimeters.

IM_BLOB_EULER_NUMBER

The number of blobs - number of holes. This value is more useful when results are grouped i.e. when the IM_BLOB_IDENTIFICATION control setting is set to IM_WHOLE_IMAGE than IM_INDIVIDUAL.

IM_BLOB_LENGTH

Derived from the perimeter (P) and area (A) of the blob, assuming that

$$P = 2(\text{length} + \text{breadth}) \text{ and}$$

$$A = \text{length} \times \text{breadth}.$$

IM_BLOB_LENGTH is better than IM_BLOB_FERET_MAX_DIAMETER at estimating the length of long thin blobs. For other blob types,

IM_BLOB_FERET_MAX_DIAMETER is often better.

IM_BLOB_BREADTH

See IM_BLOB_LENGTH for the equations used to derive IM_BLOB_BREADTH.

IM_BLOB_BREADTH is better than IM_BLOB_FERET_MIN_DIAMETER at estimating the width of long thin blobs. For other blob types,

IM_BLOB_FERET_MIN_DIAMETER is often better.

IM_BLOB_ELONGATION

Equal to

$$\frac{\text{IM_BLOB_LENGTH}}{\text{IM_BLOB_BREADTH}}$$

It should be used instead of

IM_BLOB_FERET_ELONGATION for long thin blobs.

IM_BLOB_INTERCEPT_0

The number of times a transition from background to foreground (not vice versa) occurs in the horizontal direction for the entire blob. In other words, it is equal to the number of times the neighborhood configuration $[B, F]$ occurs in a blob, where B is a background pixel and F is a blob pixel.

IM_BLOB_INTERCEPT_45	The number of times that the neighborhood configuration $\begin{bmatrix} \cdot & F \\ B & \cdot \end{bmatrix}$ occurs in a blob, where F is a blob pixel, B is a background pixel, and a dot can be any pixel value.
IM_BLOB_INTERCEPT_90	The number of times that the neighborhood configuration $\begin{bmatrix} F \\ B \end{bmatrix}$ occurs in a blob, where F is a blob pixel and B is a background pixel.
IM_BLOB_INTERCEPT_135	The number of times that the neighborhood configuration $\begin{bmatrix} F & \cdot \\ \cdot & B \end{bmatrix}$ occurs in a blob, where F is a blob pixel, B is a background pixel, and a dot can be any pixel value.

The following features require grayscale pixel values, and can only be calculated if you provide a grayscale image to **imBlobCalculate()**:

IM_BLOB_SUM_PIXEL	The sum of all pixel values in a blob.
IM_BLOB_MIN_PIXEL	The minimum pixel value found in a blob.
IM_BLOB_MAX_PIXEL	The maximum pixel value found in a blob.
IM_BLOB_MEAN_PIXEL	The mean pixel value in a blob. It is equal to $\frac{\text{IM_BLOB_SUM_PIXEL}}{\text{IM_BLOB_AREA}}$
IM_BLOB_SIGMA_PIXEL	The standard deviation of pixel values in a blob. It is equal to $\sqrt{\frac{\sum p_i^2 - (\sum p_i)^2 / N}{N}}$ where N = number of pixels and p = pixel value.
IM_BLOB_SUM_PIXEL_SQUARED	The sum of the squares of each pixel value in a blob.

The following features have two definitions: a binary definition, where all pixels are considered equal, and a grayscale definition, where pixels are weighted by their value in a grayscale image (the grayscale version is slower to calculate).

If you don't provide a grayscale image to **imBlobCalculate()**, only the binary version can be calculated. If you do provide a grayscale image, both versions can be calculated, according to the **Mode** parameter.

IM_BLOB_CENTER_OF_GRAVITY_X	<p>The x position of the center of gravity of a blob. The grayscale version is equal to</p> $\frac{\text{IM_BLOB_MOMENT_X1_Y0}}{\text{IM_BLOB_SUM_PIXEL}}$ <p>while the binary version uses IM_BLOB_AREA instead of IM_BLOB_SUM_PIXEL.</p>
IM_BLOB_CENTER_OF_GRAVITY_Y	<p>The y position of the center of gravity of a blob. The grayscale version is equal to</p> $\frac{\text{IM_BLOB_MOMENT_X0_Y1}}{\text{IM_BLOB_SUM_PIXEL}}$ <p>while the binary version uses IM_BLOB_AREA instead of IM_BLOB_SUM_PIXEL.</p>
IM_BLOB_MOMENT_X0_Y1, IM_BLOB_MOMENT_X1_Y0, IM_BLOB_MOMENT_X1_Y1, IM_BLOB_MOMENT_X0_Y2, IM_BLOB_MOMENT_X2_Y0, IM_BLOB_MOMENT_CENTRAL_X0_Y2, IM_BLOB_MOMENT_CENTRAL_X2_Y0, IM_BLOB_MOMENT_CENTRAL_X1_Y1	<p>Moments have the syntax IM_BLOB_MOMENT_Xn_Ym and are defined as</p> $\sum_i^{\infty} x_i^n y_i^m p_i,$ <p>where p_i = value of a pixel (always 1 for the binary version), x_i = its x coordinate, and y_i = its y coordinate. For central moments, coordinates are relative to a blob's center of gravity. For ordinary moments, coordinates are relative to the top-left corner of the blob identifier image. Note that you can calculate higher moments using imBlobSelectMoment().</p>

IM_BLOB_AXIS_PRINCIPAL_ANGLE This is the angle at which a blob has the least moment of inertia (the axis of symmetry). For elongated blobs, it is aligned with the longest axis. The result is always between -90° and $+90^\circ$, measured in a counter-clockwise direction from the positive x-axis. It is equal to:

$$-0.5 * \text{atan} \frac{(2 * \text{IM_BLOB_MOMENT_CENTRAL_X1_Y1})}{\text{IM_BLOB_MOMENT_CENTRAL_X2_Y0} - \text{IM_BLOB_MOMENT_CENTRAL_X0_Y2}}$$

IM_BLOB_AXIS_SECONDARY_ANGLE The angle perpendicular to **IM_BLOB_AXIS_PRINCIPAL_ANGLE**. It is always between -90° and $+90^\circ$.

The following defines allow you to select groups of features in a single call:

IM_BLOB_BOX	Adds all 4 box features plus x and y Ferets.
IM_BLOB_CONTACT_POINTS	Adds first point and other contact features (IM_BLOB_X_MIN_AT_Y_MIN, IM_BLOB_X_MAX_AT_Y_MAX, IM_BLOB_Y_MIN_AT_X_MAX, and IM_BLOB_Y_MAX_AT_X_MIN).
IM_BLOB_CENTER_OF_GRAVITY	Adds both x and y coordinates of the center of gravity.
IM_BLOB_ALL_FEATURES	Adds all features (except general Feret and general moment).
IM_BLOB_NO_FEATURES	Removes all features (except label value).

The **Mode** parameter specifies whether to calculate the binary and/or grayscale version of a feature, for features that have both a binary and grayscale definition. The **Mode** parameter can be set to:

IM_BINARY	Calculate only the binary version.
IM_GRAYSCALE	Calculate only the grayscale version. You must provide a grayscale image.
IM_DEFAULT	Calculate both the binary and grayscale version. You must provide a grayscale image.

For features that do not have both a binary and grayscale definition, use IM_DEFAULT.

See also **imBlobSelectFeret()**, **imBlobSelectMoment()**. These functions calculate a specific Feret diameter and specific moment, respectively.

imBlobSelectFeret

Async

Synopsis Select a specific Feret diameter for calculation.

Format **void imBlobSelectFeret(Thread, FeatList, Angle)**

long Thread; Thread ID

long FeatList; Feature list ID

double Angle; Angle at which to calculate the Feret diameter

Description This function selects a specific Feret diameter for calculation by **imBlobCalculate()**.

Results for this calculation can be obtained with **imBlobGetResult()** or **imBlobGetResultSingle()**, specifying IM_BLOB_GENERAL_FERET as the feature.

A call to this function overrides any previous angle specified for the general Feret (IM_BLOB_GENERAL_FERET) in the feature list.

The **Thread** parameter specifies the thread to which to send **imBlobSelectFeret()** for execution.

The **FeatList** parameter specifies the feature list.

The **Angle** parameter specifies the angle, in degrees, at which to calculate the Feret diameter.

See also **imBlobSelectFeature()**. If you want the Feret diameter at 0 or 90°, it is more efficient to use **imBlobSelectFeature()** to add IM_BLOB_FERET_X or IM_BLOB_FERET_Y to the feature list.

imBlobSelectMoment

Async

Synopsis Select a specific moment for calculation.

Format **void imBlobSelectMoment(Thread, FeatList, Mode, Type, XOrder, YOrder)**

long Thread;	Thread ID
long FeatList;	Feature list ID
long Mode;	Mode of operation
long Type;	Moment type
long XOrder;	X order of the moment
long YOrder;	Y order of the moment

Description This function selects a specific moment for calculation by **imBlobCalculate()**.

Results for this calculation can be obtained with **imBlobGetResult()** or **imBlobGetResultSingle()**, specifying IM_BLOB_GENERAL_MOMENT as the feature.

A call to this function overrides any previous values specified for IM_BLOB_GENERAL_MOMENT in the feature list.

The **Thread** parameter specifies the thread to which to send **imBlobSelectMoment()** for execution.

The **FeatList** parameter specifies the feature list.

The **Mode** parameter specifies whether to calculate the binary and/or grayscale version of the moment, if you will be providing a grayscale image to **imBlobCalculate()**. The **Mode** parameter can be set to:

IM_BINARY	Calculate only the binary version.
IM_GRAYSCALE	Calculate only the grayscale version.
IM_DEFAULT	Calculate both the binary and grayscale version.

If you will not be providing a grayscale image to **imBlobCalculate()**, set the **Mode** parameter to IM_DEFAULT.

The **Type** parameter specifies the type of moment to calculate. It can be set to either IM_CENTRAL or IM_ORDINARY.

The **XOrder** parameter specifies the x order of the moment.

The **YOrder** parameter specifies the y order of the moment.

See also **imBlobSelectFeature()**. If you require only a basic moment, it is more efficient to use **imBlobSelectFeature()** to add it to the feature list.

imBufAlloc

Sync

Gen-LC

Synopsis Allocate a buffer with multiple bands.

Format **void imBufAlloc(Thread, Xsize, Ysize, Nbands, Type, Location, BufPtr)**

long Thread;	Thread ID
long Xsize;	Buffer width
long Ysize;	Buffer height
long Nbands;	Number of bands
long Type;	Buffer data type
long Location;	Location of the memory
long* BufPtr;	Address of buffer ID

Description This function allocates a buffer with the specified number of bands, in the specified memory bank. When allocating processing or display memory, the memory is allocated on the node associated with the specified thread.

The **Thread** parameter specifies the thread to which to send **imBufAlloc()** for execution.

The **Xsize** and **Ysize** parameters specify the width and height of each band of the buffer (in pixels), respectively.

The **Nbands** parameter specifies the number of bands.

The **Type** parameter specifies the buffer data type. It can be set to:

IM_BINARY	1-bit packed binary.
IM_UBYTE	8-bit unsigned integer.
IM_BYTE	8-bit signed integer.
IM_USHORT	16-bit unsigned integer.
IM_SHORT	16-bit signed integer.
IM_ULONG	32-bit unsigned integer.
IM_LONG	32-bit signed integer.
IM_FLOAT	32-bit floating-point (IEEE-754 format).
IM_DOUBLE	64-bit floating-point (IEEE-754 format).
IM_RGB	24-bit packed RGB.

Note that, when allocated, the first pixel of a buffer will always be aligned on (at least) an 8-byte boundary, and the pitch of the buffer will always be a multiple of 4 bytes (padding is added if necessary).

The **Location** parameter specifies where to allocate the buffer. It can be set to:

IM_PROC	Processing memory.
IM_DISP	Display memory (any available bank: red, green, or blue).
IM_DISP_RED	Red display memory.
IM_DISP_GREEN	Green display memory.
IM_DISP_BLUE	Blue display memory.
IM_DISP_OVERLAY	Overlay display memory.
IM_HOST	Host system memory (specifically, DMA memory from non-paged pool).

Note that display memory refers to off-screen display memory.

The **BufPtr** parameter specifies the address in which to return the buffer identifier. If the buffer could not be allocated, 0 is returned.

See also **imBufChild()**, **imBufAllocControl()**. Use **imBufChild()** if you are allocating a child buffer in on-screen display memory (**imBufAlloc()** can allocate off-screen display memory but not on-screen). Use **imBufAllocControl()** if you are allocating a buffer that is to be used solely as a control buffer (**imBufAllocControl()** allocates a buffer that can only hold fields, not data, so it can save you memory).

imBufAlloc1d

Sync

Gen-LC

Synopsis Allocate a 1D buffer.**Format** **void imBufAlloc1d(Thread, XSize, Type, Location, BufPtr)**

long Thread; Thread ID
 long XSize; Buffer size
 long Type; Buffer data type
 long Location; Location of the memory
 long* BufPtr; Address of buffer ID

Description This function allocates a one-dimensional buffer, in the specified memory bank. When allocating processing or display memory, the memory is allocated on the node associated with the specified thread.

The **Thread** parameter specifies the thread to which to send **imBufAlloc1d()** for execution.

The **XSize** parameter specifies the size of the buffer (in pixels).

The **Type** parameter specifies the buffer data type. It can be set to:

IM_BINARY	1-bit packed binary.
IM_UBYTE	8-bit unsigned integer.
IM_BYTE	8-bit signed integer.
IM_USHORT	16-bit unsigned integer.
IM_SHORT	16-bit signed integer.
IM_ULONG	32-bit unsigned integer.
IM_LONG	32-bit signed integer.
IM_FLOAT	32-bit floating-point (IEEE-754 format).
IM_DOUBLE	64-bit floating-point (IEEE-754 format).
IM_RGB	24-bit packed RGB.

Note that, when allocated, the first pixel of a buffer will always be aligned on (at least) an 8-byte boundary, and the pitch of the buffer will always be a multiple of 4 bytes (padding is added if necessary).

The **Location** parameter specifies where to allocate the buffer. It can be set to:

IM_PROC	Processing memory.
IM_DISP	Display memory (any available bank: red, green, or blue).
IM_DISP_RED	Red display memory.
IM_DISP_GREEN	Green display memory.
IM_DISP_BLUE	Blue display memory.
IM_DISP_OVERLAY	Overlay display memory.
IM_HOST	Host system memory (specifically, DMA memory from non-paged pool).

Note that display memory refers to off-screen display memory.

The **BufPtr** parameter specifies the address in which to return the buffer identifier. If the buffer could not be allocated, 0 is returned.

imBufAlloc2d

Sync

Gen-LC

Synopsis Allocate a 2D buffer.**Format** **void imBufAlloc2d(Thread, Xsize, Ysize, Type, Location, BufPtr)**

long Thread;	Thread ID
long Xsize;	Buffer width
long Ysize;	Buffer height
long Type;	Buffer data type
long Location;	Location of the memory
long* BufPtr;	Address of buffer ID

Description This function allocates a two-dimensional buffer, in the specified memory bank. When allocating processing or display memory, the memory is allocated on the node associated with the specified thread.

The **Thread** parameter specifies the thread to which to send **imBufAlloc2d()** for execution.

The **Xsize** and **Ysize** parameters specify the width and height of the buffer (in pixels), respectively.

The **Type** parameter specifies the buffer data type. It can be set to:

IM_BINARY	1-bit packed binary.
IM_UBYTE	8-bit unsigned integer.
IM_BYTE	8-bit signed integer.
IM_USHORT	16-bit unsigned integer.
IM_SHORT	16-bit signed integer.
IM_ULONG	32-bit unsigned integer.
IM_LONG	32-bit signed integer.
IM_FLOAT	32-bit floating-point (IEEE-754 format).
IM_DOUBLE	64-bit floating-point (IEEE-754 format).
IM_RGB	24-bit packed RGB.

Note that, when allocated, the first pixel of a buffer will always be aligned on (at least) an 8-byte boundary, and the pitch of the buffer will always be a multiple of 4 bytes (padding is added if necessary).

The **Location** parameter specifies where to allocate the buffer. It can be set to:

IM_PROC	Processing memory.
IM_DISP	Display memory (any available bank: red, green, or blue).
IM_DISP_RED	Red display memory.
IM_DISP_GREEN	Green display memory.
IM_DISP_BLUE	Blue display memory.
IM_DISP_OVERLAY	Overlay display memory.
IM_HOST	Host system memory (specifically, DMA memory from non-paged pool).

Note that display memory refers to off-screen display memory.

The **BufPtr** parameter specifies the address in which to return the buffer identifier. If the buffer could not be allocated, 0 is returned.

See also **imBufChild()**, **imBufAllocControl()**. Use **imBufChild()** if you are allocating a child buffer in on-screen display memory (**imBufAlloc()** can allocate off-screen display memory but not on-screen). Use **imBufAllocControl()** if you are allocating a buffer that is to be used solely as a control buffer (**imBufAllocControl()** allocates a buffer that can only hold fields, not data, so it can save you memory).

imBufAllocControl

Sync

Gen-LC

Synopsis Allocate a control buffer.

Format **void imBufAllocControl(Thread, BufPtr)**

long Thread; Thread ID

long* BufPtr; Address of buffer ID

Description This function allocates a single band buffer with no size. Buffers allocated using **imBufAllocControl()** can be used as control buffers.

The **Thread** parameter specifies the thread to which to send **imBufAllocControl()** for execution.

The **BufPtr** parameter specifies the address in which to return the buffer identifier. If the buffer could not be allocated, 0 is returned.

imBufChild

Sync

Gen-LC

Synopsis Allocate a child buffer within an existing buffer or on the display.

Format **void imBufChild(Thread, Buf, Xstart, Ystart, Xsize, Ysize, ChildPtr)**

long Thread;	Thread ID
long Buf;	Parent buffer ID
long Xstart;	X origin of child buffer
long Ystart;	Y origin of child buffer
long Xsize;	Child buffer width
long Ysize;	Child buffer height
long* ChildPtr;	Address of child buffer ID

Description This function allocates a rectangular region of interest, within an existing buffer or at a specific location on the display (that is, in on-screen display memory). This region, called a child buffer, remains part of its parent buffer; it is not allocated its own memory. However, a child buffer is considered a data buffer in its own right and can therefore be used in the same way as any other buffer. Any modifications made to a child buffer affects its parent buffer (and vice versa).

A child buffer has the same number of bands as its parent buffer.

The **Thread** parameter specifies the thread to which to send **imBufChild()** for execution.

The **Buf** parameter specifies the parent buffer. If you are allocating a child buffer within an existing buffer, this parameter must be set to the identifier of the existing buffer. If you are allocating a child buffer on the display, this parameter can be set to:

IM_DISP	Display buffer (1 band if the display is in monochrome mode; 3 bands if it is in color mode).
IM_DISP_COLOR	Color display buffer (3 bands). This can only be allocated if the display is in color mode.
IM_DISP_RED	Red display buffer (1 band). This can only be allocated on the color version of the display.
IM_DISP_GREEN	Green display buffer (1 band). This can only be allocated on the color version of the display.

IM_DISP_BLUE	Blue display buffer (1 band). This can only be allocated on the color version of the display.
IM_DISP_OVERLAY	Overlay display buffer (1 band).

Note that, when you are allocating a child buffer on the display, you should generally set the **Buf** parameter to IM_DISP. This will allow the application to run on either the monochrome or color version of the display, regardless of the current display mode.

The **Xstart** and **Ystart** parameters specify the x and y coordinates of the child buffer's origin, respectively. These coordinates are relative to the top-left corner of its parent buffer.

If the parent buffer is a packed binary buffer, it is best to specify a buffer origin that is a multiple of 8, since some operations (such as **imBufGet()**) can only be performed on a byte-aligned buffer.

The **Xsize** and **Ysize** parameters specify the width and height of the child buffer, respectively.

To create a child buffer whose width extends from the specified **Xstart** position to the right side of the parent buffer, set **Xsize** to IM_ALL. To create a child buffer whose height extends from the specified **Ystart** position to the bottom of the parent buffer, set **Ysize** to IM_ALL. Setting both **Xsize** and **Ysize** to IM_ALL creates the largest child buffer that fits within the parent buffer (from the specified child buffer origin).

When the requested size of the child buffer extends beyond that of the parent buffer (that is, the child buffer does not fit entirely within the parent buffer), the child buffer will be clipped to fit within the parent buffer. Thus, the actual width and height of the child buffer will be smaller than those specified using the **Xsize** and **Ysize** parameters.

The **ChildPtr** parameter specifies the address in which to return the child buffer identifier. If the child buffer could not be allocated, 0 is returned.

Note If the requested child buffer does not lie within the parent buffer in either the x or y dimension, an error is generated.

Example The following code creates a child buffer that fills the entire display.

```
imBufChild(Thread, IM_DISP, 0, 0, IM_ALL, IM_ALL, &ChildBuf);
```

See also **imBufChildBand()**. The **imBufChildBand()** function allocates a child buffer from a band of a multi-band buffer.

imBufChildBand

Sync

Gen-LC

Synopsis Allocate a child buffer from a band of a multi-band buffer.

Format **void imBufChildBand(Thread, Buf, BandNum, BandPtr)**

long Thread;	Thread ID
long Buf;	Parent buffer ID
long BandNum;	Band number
long* BandPtr;	Address of child buffer ID

Description This function allocates a one-band child buffer from the specified band of a multi-band buffer. A child buffer remains part of its parent buffer; it is not allocated its own memory. However, a child buffer is considered a data buffer in its own right and can therefore be used in the same way as any other buffer. Any modifications made to a child buffer affects its parent buffer (and vice versa).

The **Thread** parameter specifies the thread to which to send **imBufChildBand()** for execution.

The **Buf** parameter specifies the parent buffer.

The **BandNum** parameter specifies the band of the parent buffer from which to allocate the child buffer. This parameter must be set to the index of the required band; the valid range is 0 to (number of bands of the parent buffer - 1).

The **BandPtr** parameter specifies the address in which to return the child buffer identifier. If the child buffer could not be allocated, 0 is returned.

See also **imBufChild()**. The **imBufChild()** function allocates a child buffer within an existing buffer or on the display (that is, in on-screen display memory).

imBufChildMove

Async

Gen-LC

Synopsis Move and/or resize a child buffer.

Format **void imBufChildMove(Thread, Child, Xoff, Yoff, Xsize, Ysize)**

long Thread;	Thread ID
long Child;	Child buffer ID
long Xoff;	X offset
long Yoff;	Y offset
long Xsize;	New child width
long Ysize;	New child height

Description This function moves and/or resizes a child buffer that was allocated with **imBufChild()**. The function does not move any data; instead, it modifies the previously allocated child buffer so that it occupies a different portion of its parent buffer.

Note that it is more efficient to move an existing child buffer than to free it and allocate a new one.

The **Thread** parameter specifies the thread to which to send **imBufChildMove()** for execution.

The **Child** parameter specifies the child buffer to move/resize.

The **Xoff** and **Yoff** parameters specify the amount by which to move the specified buffer, in the X and Y directions, respectively. Note that by default, these offsets are relative to the current child buffer position, not to the origin of its parent buffer. To set the offsets relative to the parent buffer origin, you should add IM_PARENT to the X and/or Y offset. You must add it to both offsets if you want both to be offset from the parent buffer origin. For example:

```
imBufChildMove(Thread, Buf, IM_PARENT+Xoff, IM_PARENT+Yoff, Xsize, Ysize)
```

Note that in this case, the specified offset is relative to the origin of the real parent buffer. That is, if the child being moved is a child of a child (and so on), the coordinates are given relative to the origin of the original parent buffer (the one that was allocated with **imBufAlloc()**, not the one allocated with **imBufChild()**).

The **Xsize** and **Ysize** parameters specify the new width and height for the child buffer, respectively. To maintain the current width, set **Xsize** to `IM_NO_CHANGE`; to maintain the current height, set **Ysize** to `IM_NO_CHANGE`.

Note You cannot move or resize a child buffer such that it extends outside its parent buffer.

imBufClear

Async

PP

Multi-band

Gen-LC

Synopsis Clear a buffer to a constant value.

Format `void imBufClear(Thread, Buf, Value, OSB)`

long Thread; Thread ID

long Buf; ID of buffer to clear

double Value; Value with which to clear buffer

long OSB; OSB ID (or 0)

Description This function clears a buffer to a specified constant value.

The **Thread** parameter specifies the thread to which to send **imBufClear()** for execution.

The **Buf** parameter specifies the buffer to clear. This buffer can be of any data type. In addition, it can have multiple bands (all bands will be cleared to the same value).

The **Value** parameter specifies the constant with which to clear the buffer.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note The PPs are only used if the buffer to clear is a packed binary buffer.

imBufClone

Sync

Gen-LC

Synopsis Duplicate a buffer.**Format** **void imBufClone(Thread, Buf, NewLocation, NewBufPtr)**

long Thread;	Thread ID
long Buf;	ID of buffer to duplicate
long NewLocation;	Location of the memory
long* NewBufPtr;	Address of new buffer ID

Description This function allocates a new buffer with the same attributes (size, number of bands, data type) as an existing buffer, in the specified memory bank. You can therefore duplicate an existing buffer without first inquiring about its attributes. When allocating processing or display memory, the memory is allocated on the node associated with the specified thread. Note that no data or control fields are copied to the new buffer. If necessary, use **imBufCopy()** to copy data and **imBufCopyField()** to copy control fields.

The **Thread** parameter specifies the thread to which to send **imBufClone()** for execution.

The **Buf** parameter specifies the buffer whose attributes to duplicate.

The **NewLocation** parameter specifies where to allocate the new buffer. It can be set to:

IM_DEFAULT	The same type of memory as that allocated for the original buffer.
IM_PROC	Processing memory.
IM_DISP	Display memory (any available bank: red, green, or blue).
IM_DISP_RED	Red display memory.
IM_DISP_GREEN	Green display memory.
IM_DISP_BLUE	Blue display memory.
IM_DISP_OVERLAY	Overlay display memory.
IM_HOST	Host system memory (specifically, DMA memory from non-paged pool).

Note that display memory refers to off-screen display memory.

The **NewBufPtr** parameter specifies the address in which to return the new buffer identifier. If the buffer could not be allocated, 0 is returned.

imBufControl

Async

Gen-LC

Synopsis Modify a buffer attribute.

Format `void imBufControl(Thread, Buf, Item, Value)`

long Thread;

Thread ID

long Buf;

Buffer ID

long Item;

Item to set

double Value;

Value for Item

Description This function modifies an attribute of an existing buffer.

The **Thread** parameter specifies the thread to which to send **imBufControl()** for execution.

The **Buf** parameter specifies the buffer.

The **Item** parameter specifies the attribute to modify, while the **Value** parameter specifies the value for this attribute. The table below lists those attributes that can be modified, and their allowable values.

Item	Value	Meaning
IM_BUF_PITCH	any value	Change the buffer’s pitch to the specified number of bytes.
IM_BUF_LOCK	IM_DEFAULT	Lock a paged (virtual) buffer in physical memory so it can be used for DMA transfers.
IM_BUF_UNLOCK	IM_DEFAULT	Unlock a paged (virtual) buffer, allowing it to be swapped. The buffer can no longer be used for DMA transfers.

imBufCopy

Async

PP

Multi-band

Gen-LC

Synopsis Copy a buffer.

Format **void imBufCopy(Thread, SrcBuf, DstBuf, Control, OSB)**

long Thread; Thread ID
long SrcBuf; Source buffer ID
long DstBuf; Destination buffer ID
long Control; Control buffer ID (or 0)
long OSB; OSB ID (or 0)

Description This function copies the contents of a specified source buffer to a specified destination buffer. Control fields are not copied (use **imBufCopyField()** to copy control fields).

The **Thread** parameter specifies the thread to which to send **imBufCopy()** for execution.

The **SrcBuf** parameter specifies the source buffer. This buffer can be of any data type. In addition, it can be located anywhere in your system (in processing or display memory on any node).

The **DstBuf** parameter specifies the destination buffer. This buffer must have the same pixel size as the source buffer. It can be located anywhere in your system (in processing or display memory on any node).

The **Control** parameter specifies the control buffer with which to perform the function. This parameter must be set to 0.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBufCopyPCI()**, **imBufCopyVM()**. These functions can perform a variety of formatting operations on the data as it is copied. However, these functions depend on the physical path involved, so you need to understand your system's architecture to exploit their capabilities.

Note The source and destination buffer can overlap if both are in display memory. Therefore, you can use **imBufCopy()** to scroll a region of the display (in any direction). In this case, the copy is performed entirely by the display VIA (it has no impact on concurrent operations which use processing memory), so it is a more efficient way to implement scrolling than copying the entire image from processing memory to display memory. (In this case, you only have to copy from processing to display memory those few new lines or columns needed to replace the ones that scrolled off the screen). Note that a display to display copy is faster when the display is in monochrome mode, so if you only need to display monochrome images, you should select monochrome mode (use **imDispControl()**).

imBufCopyField

Async

Gen-LC

Synopsis Copy buffer field(s).

Format **void imBufCopyField(Thread, SrcBuf, SrcTag, DstBuf, DstTag)**

long Thread; Thread ID
 long SrcBuf; Source buffer ID
 long SrcTag; Tag of field to copy
 long DstBuf; Destination buffer ID
 long DstTag; Tag of field to write to

Description This function copies one or all fields from a specified source buffer to a specified destination buffer. When copying one field, you can give the field a different tag in the destination buffer, if necessary.

Note that this function is most commonly used after copying a buffer's data, since a buffer's fields are not copied when the buffer is copied.

The **Thread** parameter specifies the thread to which to send **imBufCopyField()** for execution.

The **SrcBuf** parameter specifies the source buffer.

The **SrcTag** parameter specifies the field to copy. To copy all of a buffer's fields, set this parameter to IM_ALL.

The **DstBuf** parameter specifies the destination buffer.

The **DstTag** parameter specifies the field to write to, when copying just one field from the source buffer. Note that, if you do not want to give the field being copied a different tag, use the same tags for the **SrcTag** and **DstTag** parameters.

When copying all of a buffer's fields, set the **DstTag** parameter to IM_ALL.

Example The following code copies a buffer's data using **imBufCopy()**, then copies the buffer's fields.

```
/* Copy the buffer's data */
imBufCopy(Thread, SrcBuf, DstBuf, 0, 0);

/* Now copy all buffer fields */
imBufCopyField(Thread, SrcBuf, IM_ALL, DstBuf, IM_ALL);
```

imBufCopyPCI

Async

Gen-LC

Synopsis Copy a buffer over the PCI bus.

Format **void imBufCopyPCI(Thread, SrcBuf, DstBuf, Control, OSB)**

long Thread; Thread ID
 long SrcBuf; Source buffer ID
 long DstBuf; Destination buffer ID
 long Control; Control buffer ID (or 0)
 long OSB; OSB ID (or 0)

Description This function copies the contents of a specified source buffer to a specified destination buffer, over the PCI bus (using only one VIA). Control fields are not copied (use **imBufCopyField()** to copy control fields).

Note that, if you use **imBufCopyPCI()**, you should be familiar enough with you system's architecture to know if the source and destination buffers are connected by the PCI bus.

imBufCopyPCI() can perform a variety of formatting options on the data as it is copied. These options are specified through the function's control buffer. Options depend on whether the VIA controlling the copy is local to the source or the destination buffer.

The **Thread** parameter specifies the thread to which to send **imBufCopyPCI()** for execution.

The **SrcBuf** parameter specifies the source buffer and the **DstBuf** parameter specifies the destination buffer. Various combinations of bits per pixel and number of bands are allowed for these buffers, as outlined in the following table. Certain combinations are allowed only if a specific control field or VIA is used. However, if a specific VIA must be used, it will be chosen automatically. If either VIA can be used, you can force a specific one with the IM_CTL_VIA field; otherwise, one will be chosen by default. In all cases, the source and destination buffers must be in different memory banks (that is, in different nodes or in processing memory and display memory in the same node). Note that byte-aligned packed binary buffers are supported (they are treated as 8-bit buffers).

SrcBuf*	DstBuf*	VIA	Control field	Description
(8-32)	(same as SrcBuf)	Src/Dst	-	Simple copy.
nx(8-32)	(same as SrcBuf)	Src/Dst	-	Copy all bands sequentially.
8	3x8**	Dst	-	Copy same data to all 3 display bands.
(16-32)	8	Src	IM_CTL_BYTE_EXT	Copy most significant byte.
16	2x8	Dst	-	Separate color planes.
24	3x8	Dst	-	Separate color planes.
32	4x8	Dst	-	Separate color planes.
2x8	16	Src	-	Pack color planes.
3x8	24	Src	-	Pack color planes.
4x8	32	Src	-	Pack color planes.
24	4x8	Dst	IM_CTL_PACK	Add a 0 byte and separate color planes.
32	3x8	Dst	IM_CTL_PACK	Discard most-significant byte and separate color planes.
4x8	24	Src	IM_CTL_PACK	Pack color planes and discard most-significant byte.
3x8	32	Src	IM_CTL_PACK	Pack color planes and add a 0 byte.
16	3x8	Dst	IM_CTL_FMTCVR	Separate RGB555 or RGB565 into 3 planes.

* = number of bands x bits per pixel; if not stated, buffers are 1-band

** = must be in display memory

❖ The actual combinations supported are dictated by the hardware capabilities. The most useful combinations are listed above.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imBufCopyPCI()** are listed below, with default values in bold-face (you will find more information about these fields in the *Genesis Native Library User Guide*). Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

The following fields apply whether the source or destination VIA is used:

Field	Values	Meaning
IM_CTL_SETUP	IM_DEFAULT	Perform a full setup before copying.
	IM_ADDRESS_ONLY	Reprogram only the source and destination buffer address before copying. Skip programming of other VIA registers.
IM_CTL_BYTE_SWAP	IM_DISABLE	No effect.
	IM_ENABLE	Swap the 1st and 3rd bytes of each pixel (to convert, for example, RGB or RGBa color images to BGR or BGRa color images).
IM_CTL_DIR_X	IM_FORWARD	Copy left to right.
	IM_REVERSE	Copy right to left.
IM_CTL_DIR_Y	IM_FORWARD	Copy top to bottom.
	IM_REVERSE	Copy bottom to top.
IM_CTL_PACK	IM_DEFAULT	No effect.
	IM_24_TO_32	Add a 0 byte to each pixel of a 24-bit or 3x8-bit buffer, to produce a 32-bit buffer.
	IM_32_TO_24	Discard the last byte of each pixel of a 32-bit buffer, to produce a 24-bit buffer.
IM_CTL_SUBSAMP_X	1 - 16	Only copy every <i>nth</i> column to the destination buffer, starting with the first column.

IM_CTL_SUBSAMP_Y	1 - 16	Only copy every <i>n</i> th row to the destination buffer, starting with the first row.
IM_CTL_VIA	IM_DEFAULT	Use any suitable VIA to perform the copy.
	IM_VIA_SOURCE	Use the VIA local to the source buffer to perform the copy.
	IM_VIA_DESTINATION	Use the VIA local to the destination buffer to perform the copy.

The following fields only apply if the destination VIA is used:

Field	Values	Meaning
IM_CTL_TAG_BUF	0	No effect.
	A buffer ID	Copy with tag (using the specified packed binary buffer as a tag buffer). The tag buffer must be in the same memory bank as the destination buffer.
IM_CTL_FMTCVR	IM_DEFAULT	No effect.
	IM_RGB555	Expand 16-bit color images (in RGB555 format) to 3x8-bit.
	IM_RGB565	Expand 16-bit color images (in RGB565 format) to 3x8-bit.
IM_CTL_WRTMSK	0 - 0xFFFFFFFF	Don't overwrite bit planes in the destination buffer if the corresponding mask bit is 0. This option can only be used if you are copying to the display. The required value must be specified using 24 bits: the least significant 8 bits applies to the red buffer, the next 8 bits to the green buffer, and the most significant 8 bits to the blue buffer.
IM_CTL_ZOOM_X	1, 2, or 4	Replicate each column <i>n</i> times before writing to the destination buffer.
IM_CTL_ZOOM_Y	1, 2, or 4	Replicate each row <i>n</i> times before writing to the destination buffer.

The following fields only apply if the source VIA is used:

Field	Values	Meaning
IM_CTL_BYTE_EXT	IM_DISABLE	No effect.
	8 - 32	Copy only the most significant 8 bits to the destination buffer, from images with the specified pixel depth.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBufCopy0**. If you use **imBufCopy0**, you don't have to worry about the physical path involved because **imBufCopy0** will use whatever path is available.

imBufCopyROI

Async

PP

Multi-band

Synopsis Copy an ROI (or ROIs) from a source buffer into a destination buffer.

Format **void imBufCopyROI(Thread, SrcBuf, DstBuf, Mode, Control, OSB)**

 long Thread; Thread ID

 long SrcBuf; Source buffer ID

 long DstBuf; Destination buffer ID

 long Mode; Mode of operation

 long Control; Control buffer ID (or 0)

 long OSB; OSB ID (or 0)

Description This function can copy an entire list of ROIs from the source buffer to the destination buffer at arbitrary locations. The sizes and locations of each ROI are stored as lists of values in extra 1-dimensional buffers, and the identifiers of these extra buffers are specified through control fields.

 The **Thread** parameter specifies the thread to which to send **imBufCopyROI()** for execution.

 The **Src** parameter specifies the source buffer. This buffer can have a depth of 8, 16, or 32 bits.

 The **Dst** parameter specifies the destination buffer. This buffer should have the same pixel depth as the source buffer.

 The **Mode** parameter specifies the mode in which this function operates. The supported modes are:

Field	Description
IM_LINE	Each ROI is one horizontal line. Only the Y start position of each ROI needs to be specified using the appropriate control field. In this operating mode, you should supply valid IDs for the IM_ROI_SRC_START_Y and/or IM_ROI_DST_START_Y fields. When one of these two latter control fields is omitted, a ramp is assumed (that is, the lines have sequential Y values, starting at 0). This means that only a single list of Y values (either for the source buffer or destination buffer) needs to be supplied to rearrange the lines in an image.

Field	Description
IM_PATCH	Each ROI is a 2-dimensional patch. In this case, both the complete ROI size and positional information need to be specified using the appropriate control fields. In the IM_PATCH operating mode, valid buffer IDs must be supplied, whereby each buffer lists a size or location of all ROIs.

The **Control** parameter specifies the control buffer with which to perform the function. The possible control buffer fields are:

Field	Value	Meaning
IM_ROI_SIZE_X	BufID	Width of each ROI.
IM_ROI_SIZE_Y	BufID	Height of each ROI.
IM_ROI_SRC_START_X	BufID	Horizontal starting position of each ROI in the source buffer.
IM_ROI_SRC_START_Y	BufID	Vertical starting position of each ROI in the source buffer.
IM_ROI_DST_START_X	BufID	Horizontal starting position of each ROI in the destination buffer.
IM_ROI_DST_START_Y	BufID	Vertical starting position of each ROI in the destination buffer.
IM_ROI_NUMBER	1-n	Number of ROIs to copy (default is size of buffers containing ROI positions)

Note that all buffers which define the ROIs should be one-dimensional and of type IM_USHORT.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imBufCopyVM

Async

Gen-LC

Synopsis Copy a buffer over the VMChannel.

Format **void imBufCopyVM(Thread, SrcBuf, DstBuf, SrcControl, DstControl, OSB)**

long Thread;	Thread ID
long SrcBuf;	Source buffer ID
long DstBuf;	Destination buffer ID
long SrcControl;	Source control buffer ID (or 0)
long DstControl;	Destination control buffer ID (or 0)
long OSB;	OSB ID (or 0)

Description This function copies the contents of a specified source buffer to a specified destination buffer, over the VMChannel. Control fields are not copied (use **imBufCopyField()** to copy control fields).

Note that, if you use **imBufCopyVM()**, you should be familiar enough with your system's architecture to know if the source and destination buffers are connected by VMChannel.

imBufCopyVM() can perform a variety of formatting operations on the data as it is copied. These operations are specified through the function's control buffers. Some operations only apply to the transmitting VIA, and some only apply to the receiving VIA.

Instead of copying between buffers, you can use **imBufCopyVM()** to copy from a VM stream to a buffer, or from a buffer to a VM stream. Any formatting options that are specified will still be performed.

Normally, when you call this function with **SrcBuf** set to **IM_VM_CHANNEL** and a 3-band destination buffer is specified, it is assumed you will be sending a 3-band image over the VMChannel. If, however, you send a 1-band image over the VMChannel to a display buffer (for example, to display a monochrome image when the display is in color mode), you need the display's VIA to replicate the image in all three bands of the destination buffer. To perform this replication, you should enable the appropriate **DstControl** buffer field.

The **Thread** parameter specifies the thread to which to send **imBufCopyVM()** for execution.

The **SrcBuf** parameter specifies the source buffer and the **DstBuf** parameter specifies the destination buffer. Various combinations of bits per pixel and number of bands are allowed for these buffers, as outlined in the table below. Certain combinations are only allowed if a specific control field is used. Note that byte-aligned packed binary buffers are supported (they are treated as 8-bit buffers).

SrcBuf*	DstBuf*	Control field	Description
(8-32)	(same as SrcBuf)	-	Simple copy.
(2-4)x8	(same as SrcBuf)	-	Copy all bands simultaneously.
8	3x8**	-	Copy same data to all 3 display bands.
(16-32)	8	IM_CTL_BYTE_EXT***	Copy most significant byte.
(16-32)	3x8**	IM_CTL_BYTE_EXT***	Copy most significant byte to all 3 display bands.
16	2x8	-	Separate color planes.
24	3x8	-	Separate color planes.
32	4x8	-	Separate color planes.
2x8	16	-	Pack color planes.
3x8	24	-	Pack color planes.
4x8	32	-	Pack color planes.
24	4x8	IM_CTL_PACK****	Add a 0 byte and separate color planes.
32	3x8	IM_CTL_PACK***	Discard last byte and separate color planes.
4x8	24	IM_CTL_PACK***	Pack color planes and discard last byte.
3x8	32	IM_CTL_PACK****	Pack color planes and add a 0 byte.
16	3x8	IM_CTL_FMTCVR****	Separate RGB555 or 565 into 3 planes.

* = number of bands x bits per pixel; if not stated, buffers are 1-band

** = must be in display memory

*** = this field must be added to **SrcControl** buffer

**** = this field must be added to **DstControl** buffer

❖ The actual combinations supported are dictated by the hardware capabilities. The most useful combinations are listed above.

Note that, if you are copying from a VM stream to the destination buffer, set the **SrcBuf** parameter to IM_VM_CHANNEL. If you are copying the source buffer to a VM stream, set the **DstBuf** parameter to IM_VM_CHANNEL.

The **SrcControl** and **DstControl** parameters specify the control buffers with which to perform the function. The **SrcControl** buffer controls the transmitting VIA; the **DstControl** buffer controls the receiving VIA.

Relevant fields are listed below, with default values in bold-face (you will find more information about these fields in the *Genesis Native Library User Guide*). Note that if the **SrcControl/DstControl** parameters are set to 0 or if certain fields are not added to these control buffers, the default values are used.

The following fields can be added to the **SrcControl** or **DstControl** buffers:

Field	Values	Meaning
IM_CTL_SETUP	IM_DEFAULT	Perform a full setup before copying.
	IM_ADDRESS_ONLY	Reprogram only the source and destination buffer address before copying. Skip programming of other VIA registers.
IM_CTL_ADDR_MODE	IM_DEFAULT	Access memory according to the type of scanning specified by the IM_CTL_SCAN_MODE field.
	IM_PROGRESSIVE	Access memory using progressive scanning, even if the IM_CTL_SCAN_MODE field specifies interlaced scanning.
IM_CTL_BYTE_SWAP	IM_DISABLE	No effect.
	IM_ENABLE	Swap the 1st and 3rd bytes of each pixel (to convert, for example, RGB or RGBa color images to BGR or BGRa color images).

IM_CTL_COUNT	1	Copy a single frame (or field) to the destination buffer.
	IM_CONTINUOUS	Continuously copy the source buffer to the destination buffer.
IM_CTL_COUNT_MODE	IM_FRAME	Copy the number of frames specified by the IM_CTL_COUNT field.
	IM_FIELD	Copy the number of fields specified by the IM_CTL_COUNT field. (This option can only be used when using interlaced scanning).
IM_CTL_DIR_X	IM_FORWARD	Scan left to right.
	IM_REVERSE	Scan right to left.
IM_CTL_DIR_Y	IM_FORWARD	Scan top to bottom.
	IM_REVERSE	Scan bottom to top.
IM_CTL_HEADER_EOF	IM_ENABLE	Send end of field headers.
	IM_DISABLE	Don't send end of field headers.
IM_CTL_HEADER_SOF	IM_ENABLE	Send start of field headers.
	IM_DISABLE	Don't send start of field headers.
IM_CTL_PACK	IM_DEFAULT	No effect.
	IM_24_TO_32	Add a 0 byte to each pixel of a 24-bit or 3x8-bit buffer, to produce a 32-bit buffer.
	IM_32_TO_24	Discard the last byte of each pixel of a 32-bit buffer, to produce a 24-bit buffer.
IM_CTL_SCAN_MODE	IM_PROGRESSIVE	Copy the data using progressive scanning.
	IM_INTERLACED	Copy the data using interlaced scanning.

IM_CTL_START_FIELD (this field can only be used when using interlaced scanning)	IM_ODD	Start copying on the next odd field.
	IM_EVEN	Start copying on the next even field.
	IM_NEXT	Start copying on the very next field.
IM_CTL_SUBSAMP_X	1 - 16	Only copy every <i>nth</i> column, starting with the first column.
IM_CTL_SUBSAMP_Y	1 - 16	Only copy every <i>nth</i> row, starting with the first row.
IM_CTL_STREAM_ID	0 - 14	ID of the VM stream. Note that, when copying data between buffers, this field is not needed and should not be added to either control buffer; when copying data to a VM stream, this field should be added to the SrcControl buffer; when copying data from a VM stream, this field should be added to the DstControl buffer.

The following fields can only be added to the **DstControl** buffer:

Field	Values	Meaning
IM_CTL_FMTCVR	IM_DEFAULT	No effect.
	IM_RGB555	Expand 16-bit color images (in RGB555 format) to 3x8-bit.
	IM_RGB565	Expand 16-bit color images (in RGB565 format) to 3x8-bit.
IM_CTL_REPLICATE	IM_DISABLE	Don't replicate data.
	IM_ENABLE	Replicate 8-bit, 1-band image data in all three display bands when SrcBuf is set to IM_VM_CHANNEL, and DstBuf is a 3-band display buffer.

IM_CTL_START_X	0 - <i>n</i>	Start copying from the specified column.
IM_CTL_START_Y	0 - <i>n</i>	Start copying from the specified line.
IM_CTL_STOP_X	IM_DEFAULT 0 - <i>n</i>	Copy all subsequent columns. Copy just to the specified column.
IM_CTL_STOP_Y	IM_DEFAULT 0 - <i>n</i>	Copy all subsequent lines. Copy just to the specified line.
IM_CTL_TAG_BUF	0 A buffer ID	No effect. Copy with tag (using the specified packed binary buffer as a tag buffer). The tag buffer must be in the same memory bank as the destination buffer.
IM_CTL_WRTMSK	0 - 0xFFFFFFFF	Don't overwrite bit planes in the destination buffer if the corresponding mask bit is 0. This option can only be used if you are copying to the display. The required value must be specified using 24 bits: the low 8 bits apply to the red buffer, the next 8 bits to the green buffer, and the high 8 bits to the blue buffer.
IM_CTL_ZOOM_X	1 , 2, or 4	Replicate each column <i>n</i> times before writing to the destination buffer.
IM_CTL_ZOOM_Y	1 , 2, or 4	Replicate each row <i>n</i> times before writing to the destination buffer.

The following field can only be added to the **SrcControl** buffer:

Field	Values	Meaning
IM_CTL_BYTE_EXT	IM_DISABLE 8 - 32	No effect. Send only the most significant 8 bits, from images with the specified pixel depth.

IM_CTL_DISPLAY_SYNC (this field only applies when copying to the display)	IM_DISABLE	Perform the copy as soon as possible.
	IM_ENABLE	Wait before copying, if necessary, in order to avoid visible artifacts.

The **OSB** parameter specifies the operation status block in which to store status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBufCopy()**. If you use this function, you don't have to worry about the physical path involved because **imBufCopy()** will use whatever path is available.

imBufCreate

Sync

Gen-LC

Synopsis Create a buffer out of memory that has already been allocated.

Format **void imBufCreate(Thread, Xsize, Ysize, Nbands, Type, Location, AddrPtr, Pitch, BufPtr)**

long Thread;	Thread ID
long Xsize;	Buffer width
long Ysize;	Buffer height
long Nbands;	Number of bands
long Type;	Buffer data type
long Location;	Location of the memory
void** AddrPtr;	Pointer to array of addresses
long Pitch;	Buffer pitch
long* BufPtr;	Address of buffer ID

Description This function creates a buffer out of memory that has already been allocated. This memory can be contiguous physical memory, virtual memory, or memory from one or more existing Genesis buffers.

When using memory from existing Genesis buffers, **AddrPtr** should point to a list of the Genesis buffer IDs. The buffers must all be in the same memory bank and must have the same **Xsize**, **Ysize**, **Type**, and **Pitch**. The **Xsize**, **Ysize**, **Type**, and **Pitch** of the Genesis buffers are used to create the new buffer; the passed values for these parameters are ignored. The new buffer is usable by any Genesis function.

When using contiguous physical memory, **AddrPtr** should point to a list of the physical (PCI) addresses. Valid values must be given for **Xsize**, **Ysize**, **Type**, and **Pitch**. The new buffer is usable by any Genesis function, although it should only be used as the destination of a processing function.

When using non-paged virtual memory, **AddrPtr** should point to a list of virtual addresses with corresponding physical memory that is contiguous and fixed (it will not be moved by the operating system). Valid values must be given for **Xsize**, **Ysize**, **Type**, and **Pitch**. Note that a non-paged virtual buffer behaves the same way as a contiguous physical (non-paged) buffer; the only difference is that you pass a virtual address instead of a physical address.

When using virtual paged memory, **AddrPtr** should point to a list of virtual addresses in the calling processes's address space. Valid values must be given for **Xsize**, **Ysize**, **Type**, and **Pitch**. The new buffer can only be used by **imBufCopy()** and **imBufCopyPCI()**, and under Windows NT, the appropriate pages must be locked in physical memory before attempting to copy the buffer. To lock/unlock memory, use **imBufControl()**.

Note that **imBufCreate()** never allocates memory. In addition, when a created buffer is freed using **imBufFree()**, no memory is freed.

The **Thread** parameter specifies the thread to which to send **imBufCreate()** for execution.

The **Xsize** and **Ysize** parameters specify the width and height of the buffer (in pixels), respectively. When using existing Genesis buffers to create the new buffer, the values for these parameters are ignored; otherwise, valid values must be given.

The **Nbands** parameter specifies the number of bands for the buffer.

The **Type** parameter specifies the buffer data type. It can be set to:

IM_BINARY	1-bit packed binary.
IM_UBYTE	8-bit unsigned integer.
IM_BYTE	8-bit signed integer.
IM_USHORT	16-bit unsigned integer.
IM_SHORT	16-bit signed integer.
IM_ULONG	32-bit unsigned integer.
IM_LONG	32-bit signed integer.
IM_FLOAT	32-bit floating-point (IEEE-754 format).
IM_DOUBLE	64-bit floating-point (IEEE-754 format).
IM_RGB	24-bit packed RGB.

When using existing Genesis buffers to create the new buffer, the value for **Type** is ignored; otherwise, the required type must be given.

The **Location** parameter specifies the type of memory from which to create the new buffer. It can be set to:

IM_DEFAULT	Memory from existing Genesis buffers.
IM_NON_PAGED	Contiguous physical memory.
IM_PAGED	Virtual paged memory.
IM_NON_PAGED_VIRTUAL	Contiguous physical memory but the address supplied is its virtual address.

The **AddrPtr** parameter specifies the pointer to the array containing the Genesis buffer IDs, physical addresses, or virtual addresses from which to create the buffer. There must be one buffer ID/physical address/virtual address for each band to be created.

The **Pitch** parameter specifies the buffer pitch, in bytes. When you create a buffer, its pitch should be a multiple of 4 bytes in order to ensure that it can be used by all Genesis functions. When using existing Genesis buffers to create the new buffer, the value for this parameter is ignored; otherwise, a valid value must be given.

The **BufPtr** parameter specifies the address in which to return the buffer identifier. If the buffer could not be allocated, 0 is returned.

Note When you create an IM_PAGED buffer, it must be contiguous. The pitch must be equal to the buffer width in bytes (and it must still be a multiple of 4 bytes). Furthermore, an IM_PAGED buffer can only be copied to (or from) a buffer which is physically contiguous in Genesis memory.

There is also a Windows NT restriction on the use of IM_PAGED buffers created by **imBufCreate()**, which does not apply to IM_NON_PAGED buffers or to IM_HOST buffers allocated with **imBufAlloc()**. Under Windows NT, a paged (or virtual) buffer must be locked in a process's working set before it can be used in a DMA transfer (a process's working set is that set of memory pages which currently resides in physical memory). To accomplish this, call **imBufControl()** with IM_BUF_LOCK. Doing so will also try to increase the size of the process's working set, if necessary, so that it can contain the entire buffer.

This buffer restriction arises because a Windows NT process does not have the right to increase its working set unless it has the SE_INC_BASE_PRIORITY_NAME privilege. Users in the Administrators and Power Users groups normally have this privilege. If you want to grant it to ordinary users, you can do it through the User Manager (you need to enable "Show Advanced User Rights", and select the right to "Increase scheduling priority").

Without the right to increase your working set size, only relatively small paged buffers should be used. With the right, you can use larger paged buffers.

However, there is no absolute guarantee that paged buffers will work because there are other limitations to consider. First, if there is not enough available physical memory, then Windows NT will not be able to lock the entire buffer. Secondly, even if a buffer is successfully locked, Windows NT

might swap out the buffer if it swaps out your entire process. Therefore, paged buffers will only work reliably on a system with plenty of available resources, and even then there is unfortunately no absolute guarantee. If Windows swaps a page of a buffer out of physical memory while Genesis is performing a DMA transfer to that page, system memory will be corrupted and Windows will probably crash.

For the reasons stated above, it is recommended that you do not use paged Host buffers at all; instead, use non-paged buffers.

imBufFree

Async

Gen-LC

Synopsis Free a buffer.

Format **void imBufFree(Thread, Buf)**

long Thread; Thread ID

long Buf; ID of buffer to free

Description This function deallocates a previously allocated buffer.

Note that, once a buffer has been deallocated, its identifier should no longer be used. You should ensure that all functions using this buffer have completed before deallocating it.

The **Thread** parameter specifies the thread to which to send **imBufFree()** for execution. This thread must be on the same node as the thread which allocated the buffer.

The **Buf** parameter specifies the buffer to deallocate.

imBufGet

Sync

Multi-band

Gen-LC

Synopsis Get data from a buffer and transfer it to Host memory.

Format **void imBufGet(Thread, Buf, Ptr)**

long Thread; Thread ID
 long Buf; ID of buffer to transfer
 void* Ptr; Address of array

Description This function transfers data from a specified buffer to an array in Host memory. If the buffer has more than one band, they are all transferred, one after another.

The **Thread** parameter specifies the thread to which to send **imBufGet()** for execution.

The **Buf** parameter specifies the buffer whose contents to transfer. This buffer can be of any data type. If it is a packed binary buffer, it must be byte-aligned.

The **Ptr** parameter specifies the address of the user-supplied array in which to place the data. The array should be large enough to hold all the data to be transferred.

Note The Host CPU performs the transfer. For a DMA transfer to Host memory, you should allocate the Host memory using **imBufAlloc...()** (set its **Location** parameter to IM_HOST), then use **imBufCopy()** or **imBufCopyPCI()** to perform the transfer.

See also **imBufGet1d()**, **imBufGet2d()**. These functions can also transfer data to Host memory.

imBufGet1d

Sync

Multi-band

Gen-LC

Synopsis Get a 1D block of data from a buffer and transfer it to Host memory.

Format **void imBufGet1d(Thread, Buf, Xstart, Xsize, Ptr)**

long Thread;	Thread ID
long Buf;	ID of buffer to transfer
long Xstart;	Block origin
long Xsize;	Block width
void* Ptr;	Address of array

Description This function transfers a 1D block of data from a buffer to an array in Host memory. If the buffer has more than one band, they are all transferred, one after another.

The **Thread** parameter specifies the thread to which to send **imBufGet1d()** for execution.

The **Buf** parameter specifies the buffer whose contents to transfer. This buffer can be of any data type. If it is a packed binary buffer, it must be byte-aligned.

The **Xstart** parameter specifies the origin of the block, relative to the top-left corner of the source buffer.

The **Xsize** parameter specifies the size of the block (in pixels).

The **Ptr** parameter specifies the address of the user-supplied array in which to place the data. The array should be large enough to hold all the data to be transferred.

Note The Host CPU performs the transfer. For a DMA transfer to Host memory, you should allocate the Host memory using **imBufAlloc...()** (set its **Location** parameter to IM_HOST), then use **imBufCopy()** or **imBufCopyPCI()** to perform the transfer.

See also **imBufGet()**, **imBufGet2d()**. These functions can also transfer data to Host memory.

imBufGet2d

Sync

Multi-band

Gen-LC

Synopsis Get a 2D block of data from a buffer and transfer it to Host memory.

Format **void imBufGet2d(Thread, Buf, Xstart, Ystart, Xsize, Ysize, Ptr)**

long Thread; Thread ID
 long Buf; ID of buffer to transfer
 long Xstart; X origin of block
 long Ystart; Y origin of block
 long Xsize; Block width
 long Ysize; Block height
 void* Ptr; Address of array

Description This function transfers a 2D block of data from a buffer to an array in Host memory. If the buffer has more than one band, they are all transferred, one after another.

The **Thread** parameter specifies the thread to which to send **imBufGet2d()** for execution.

The **Buf** parameter specifies the buffer whose contents to transfer. This buffer must be two-dimensional and can be of any data type. If it is a packed binary buffer, it must be byte-aligned.

The **Xstart** and **Ystart** parameters specify the x and y coordinates of the block's origin (in pixels), respectively. These coordinates are relative to the top-left corner of the source buffer.

The **Xsize** and **Ysize** parameters specify the width and height of the block, respectively.

The **Ptr** parameter specifies the address of the user-supplied array in which to place the data. The array should be large enough to hold all the data to be transferred.

Note The Host CPU performs the transfer. For a DMA transfer to Host memory, you should allocate the Host memory using **imBufAlloc...()** (set its **Location** parameter to IM_HOST), then use **imBufCopy()** or **imBufCopyPCI()** to perform the transfer.

See also **imBufGet()**, **imBufGet1d()**. These functions can also transfer data to Host memory.

imBufGetField

Sync

Gen-LC

Synopsis Get the value of a field.

Format **long imBufGetField(Thread, Buf, Tag, ValuePtr)**

long Thread;	Thread ID
long Buf;	Buffer ID
long Tag;	Tag of field to get
long* ValuePtr;	Address of return value

Description This function reads the value of a field.

This function performs the same task as **imBufGetFieldDouble()** but returns the value of the specified field as type long, instead of type double. Although the values of fields are internally stored as type double, many are actually integers and can therefore be retrieved using this function.

The **Thread** parameter specifies the thread to which to send **imBufGetField()** for execution.

The **Buf** parameter specifies the buffer containing the required field.

The **Tag** parameter specifies the field. For a list of fields, see individual function descriptions.

The **ValuePtr** parameter specifies the address in which to return the value of the field.

Return value The returned value is IM_SUCCESS if the specified field exists; IM_ERR_NOT_PRESENT if the specified field does not exist.

See also **imBufGetFieldDouble()**, **imBufGetNextField()**. The **imBufGetFieldDouble()** function returns the value of a field as type double. The **imBufGetNextField()** function can be used to read all of a buffer's fields.

imBufGetFieldDouble

Sync

Gen-LC

Synopsis Get the value of a field.

Format **long imBufGetFieldDouble(Thread, Buf, Tag, ValuePtr)**

long Thread;	Thread ID
long Buf;	Buffer ID
long Tag;	Tag of field to get
double* ValuePtr;	Address of return value

Description This function reads the value of a field.

This function performs the same task as **imBufGetField()** but returns the value of the specified field as type double, instead of type long. Although the values of fields are internally stored as type double, many are actually integers and can therefore be retrieved using **imBufGetField()**.

The **Thread** parameter specifies the thread to which to send **imBufGetFieldDouble()** for execution.

The **Buf** parameter specifies the buffer containing the required field.

The **Tag** parameter specifies the field. For a list of fields, see individual function descriptions.

The **ValuePtr** parameter specifies the address in which to return the value of the field.

Return value The returned value is IM_SUCCESS if the specified field exists; IM_ERR_NOT_PRESENT if the specified field does not exist.

See also **imBufGetField()**, **imBufGetNextField()**. The **imBufGetField()** function returns the value of a field as type long. The **imBufGetNextField()** function can be used to read all of a buffer's fields.

imBufGetNextField

Sync

Gen-LC

Synopsis Get the tag and value of the next buffer field.

Format **long imBufGetNextField(Thread, Buf, ContextPtr, TagPtr, ValuePtr)**

long Thread;	Thread ID
long Buf;	ID of buffer from which to read
long* ContextPtr;	Address of function's context
long* TagPtr;	Address of field's tag
double* ValuePtr;	Address of field's value

Description This function reads the tag and value of the next field in a buffer. This function allows you to read a buffer's fields without knowing which fields are in the buffer.

To start reading from the first field, the **ContextPtr** parameter should be the address of a variable that is initialized to IM_FIRST_FIELD. After each call to **imBufGetNextField()**, **ContextPtr** is automatically set to the address of the next field; therefore, the next call to the function will read the next field in the buffer.

The **Thread** parameter specifies the thread to which to send **imBufGetNextField()** for execution.

The **Buf** parameter specifies the buffer from which to read fields.

The **ContextPtr** parameter specifies the address in which to save context information. To start reading from the first field, this parameter should be the address of a variable that is initialized to IM_FIRST_FIELD. Note that the variable will be automatically updated with the next call to the function; you must not change its value between calls.

The **TagPtr** parameter specifies the address in which to return the tag of the field.

The **ValuePtr** parameter specifies the address in which to return the value of the field.

Return value The returned value is IM_SUCCESS if a field was found; IM_ERR_NOT_PRESENT if a field was not found.

Example The following code reads all fields that are attached to a buffer.

```
long Context, Tag;
double Value;

/* Find all the fields (remember to initialize the context) */
Context = IM_FIRST_FIELD;
while (imBufGetNextField(Thr, Buf, &Context, &Tag, &Value) == IM_SUCCESS)
    printf("Tag = %li, Value = %lf\n", Tag, Value);
```

See also **imBufGetField()**, **imBufGetFieldDouble()**. You should use one of these functions if you know which fields are in a buffer and you want the value of a specific field.

imBufInquire

Sync

Gen-LC

Synopsis Inquire about a buffer attribute.

Format `long imBufInquire(Thread, Buf, Item, ValuePtr)`

long Thread; Thread ID
 long Buf; Buffer ID
 long Item; Attribute about which to inquire
 void* ValuePtr; Address of return value (or NULL)

Description This function inquires about an attribute of a specified buffer.

The **Thread** parameter specifies the thread to which to send **imBufInquire()** for execution.

The **Buf** parameter specifies the buffer.

The **Item** parameter specifies the attribute about which to inquire. It can be set to:

IM_BUF_FORMAT	Pixel format of the buffer: IM_UNSIGNED (unsigned integer buffer), IM_SIGNED (signed integer buffer), IM_IEEE754 (IEEE-754 floating-point buffer), or IM_PACKED (packed binary or packed color buffer).
IM_BUF_LOCATION	Memory location of buffer. See imBufAlloc() for a list of possible locations.
IM_BUF_NUM_BANDS	Number of bands in buffer.
IM_BUF_NUM_FIELDS	Number of fields in buffer.
IM_BUF_PARENT_ID	Parent ID of buffer (if buffer is not a child buffer, 0 is returned).
IM_BUF_PHYSICAL_ADDRESS	Physical address of the first pixel in the buffer. This address can be used for DMA transfers by any PCI bus master in the system.
IM_BUF_PITCH	Buffer pitch (in bytes). The buffer pitch should always be used when accessing a buffer directly since a buffer's pitch is not necessarily the same as its width.
IM_BUF_OFFSET_BAND	Band offset within parent.
IM_BUF_OFFSET_X	X offset within parent.

IM_BUF_OFFSET_Y	Y offset within parent.
IM_BUF_OWNER_ID	Device ID on which the buffer was allocated.
IM_BUF_SIZE_BIT	Number of bits per pixel in the buffer: 1, 8, 16, 24, 32, or 64.
IM_BUF_SIZE_X	Buffer width.
IM_BUF_SIZE_Y	Buffer height.
IM_BUF_TYPE	Buffer data type (bits per pixel and format). See imBufAlloc() for a list of possible types.

The **ValuePtr** parameter specifies the address in which to return the value of the inquired attribute. Unless otherwise stated, **ValuePtr** should be the address of a long. Note that, since **imBufInquire()** also returns this value, **ValuePtr** can be set to NULL.

Return value The returned value is the value of the inquired attribute, cast to long if necessary.

imBufLoad

Sync

Multi-band

Gen-LC

Synopsis Load data from a file into a buffer.

Format **void imBufLoad(Thread, FileName, Format, Buf)**

long Thread;	Thread ID
char* FileName;	File name
long Format;	File format
long Buf;	Destination buffer ID

Description This function loads data from a file into a specified buffer.

The **Thread** parameter specifies the thread to which to send **imBufLoad()** for execution.

The **FileName** parameter specifies the name of the file from which to load the buffer.

The **Format** parameter specifies the format of the specified file. It can be set to:

IM_TIFF	TIFF format.
IM_NATIVE	Genesis native format (same as TIFF, but can contain control fields in addition to data).
IM_RAW	Raw file format (there must be no header included with the file, only data).

When loading an IM_NATIVE file, any control fields in the file are also loaded.

The **Buf** parameter specifies the buffer in which to place the contents of the specified file. Note that this buffer must be of the appropriate size to hold the data in the file. In addition, it should have the same number of bits per pixel as the data and have the same number of bands as the data.

See also **imBufRestore()**. The **imBufRestore()** function loads data from a file into an automatically allocated buffer.

imBufMap

Sync

Gen-LC

Synopsis Map a buffer into Host memory.

Format **long imBufMap(Thread, Buf, Band, Ystart, AddrPtr, PitchPtr, NlinesPtr)**

long Thread;	Thread ID
long Buf;	ID of buffer to map
long Band;	Band to map
long Ystart;	First line to map
void** AddrPtr;	Address of pointer to first pixel mapped
long* PitchPtr;	Address of line pitch of buffer
long* NlinesPtr;	Address of number of lines mapped

Description This function maps a specified buffer into Host memory and returns a pointer to the data. Note that this function maps only one band of a buffer. To map all bands of a multi-band buffer, you must call this function for each band. There is no limit to the number of buffers that can be mapped concurrently.

The **Thread** parameter specifies the thread to which to send **imBufMap()** for execution.

The **Buf** parameter specifies the buffer to map.

The **Band** parameter specifies the band of the buffer to map. This parameter must be set to the index of the required band; the valid range is 0 to (number of bands - 1).

The **Ystart** parameter specifies the first line of the buffer to map.

The **AddrPtr** parameter specifies the address in which to return the pointer to the first pixel mapped. This pointer remains valid until the buffer is freed.

The **PitchPtr** parameter specifies the address in which to return the line pitch of the buffer. The line pitch is the number of bytes from a pixel to its neighboring pixel on the line below. Note that the line pitch is not given in pixels, but in bytes.

The **NlinesPtr** parameter specifies the address in which to return the number of consecutive lines mapped. Since all buffers are currently allocated contiguous memory, the number of consecutive lines mapped is (the number of lines in the buffer) - **Ystart**.

Note Buffers might be allocated with some padding at the end of each line. Therefore, the line pitch of a buffer is not necessarily the same as its width.

Return value The returned value is the number of consecutive lines mapped.

Example See *process.c* in Appendix B.

imBufModify

Async

Gen-LC

Synopsis Modify a buffer's dimensions and/or type.

Format **void imBufModify(Thread, Buf, Xsize, Ysize, Type)**

long Thread;	Thread ID
long Buf;	ID of buffer to modify
long Xsize;	New buffer width
long Ysize;	New buffer height
long Type;	New buffer type

Description This function modifies an existing buffer's dimensions and/or type.

Although you can change the size of a buffer, you cannot change the actual amount of memory allocated to it. Therefore, you could, for example, change a 512x512 8-bit buffer into a 512x256 16-bit buffer if the original buffer occupies contiguous memory locations. However, if the buffer does not occupy contiguous memory locations (if, for example, the buffer is a child buffer), the attempt to modify it would fail.

This function allows an application to use a given region of memory for different purposes, rather than allocating new memory. If, for example, you require signed and unsigned data at different points in an application, you can allocate just one buffer and modify it when necessary.

The **Thread** parameter specifies the thread to which to send **imBufModify()** for execution.

The **Buf** parameter specifies the buffer to modify.

The **Xsize** and **Ysize** parameters specify the new width and height for the buffer, respectively. To maintain the current width, set **Xsize** to **IM_NO_CHANGE**; to maintain the current height, set **Ysize** to **IM_NO_CHANGE**.

The **Type** parameter specifies the buffer data type. It can be set to `IM_NO_CHANGE`, which maintains the current type, or to one of the following:

<code>IM_BINARY</code>	1-bit packed binary.
<code>IM_UBYTE</code>	8-bit unsigned integer.
<code>IM_BYTE</code>	8-bit signed integer.
<code>IM_USHORT</code>	16-bit unsigned integer.
<code>IM_SHORT</code>	16-bit signed integer.
<code>IM_ULONG</code>	32-bit unsigned integer.
<code>IM_LONG</code>	32-bit signed integer.
<code>IM_FLOAT</code>	32-bit floating-point (IEEE-754 format).
<code>IM_DOUBLE</code>	64-bit floating-point (IEEE-754 format).
<code>IM_RGB</code>	24-bit packed RGB.

If the buffer originally occupied contiguous memory locations, the pitch of the buffer will be updated automatically by **imBufModify()**. The new pitch will be equal to the new **Xsize** multiplied by the new pixel size (in bytes). Note, however, that the pitch is changed only if the new value is a multiple of 4 bytes (a requirement for all Genesis buffers).

Note **imBufModify()** has many uses but can be dangerous because it performs little error checking. Use this function with care.

See also **imBufControl()**. This function can be used to change a buffer's pitch.

imBufPack

Async

PP

Multi-band

Synopsis Pack or unpack a buffer.

Format **void imBufPack(Thread, SrcBuf, TagBuf, DstBuf, Mode, OSB)**

long Thread; Thread ID
 long SrcBuf; Source buffer ID
 long TagBuf; Binary tag buffer ID
 long DstBuf; Destination buffer ID
 long Mode; Operation mode
 long OSB; OSB ID (or 0)

Description This function packs or unpacks the specified buffer, according to a tag buffer. When packing, the function copies selected pixels of the source buffer to the destination buffer, starting from the top-left corner of the destination buffer. When unpacking, the function copies pixels of the source buffer to selected positions in the destination buffer, again starting from the top-left corner of the destination buffer.

The tag buffer controls which pixels of the source buffer to copy (when packing), or which pixels of the destination buffer to overwrite (when unpacking). Specifically, if a pixel of the tag buffer has a particular value (0 or 1, depending on the specified mode), the corresponding pixel of the source/destination buffer is ignored (not copied or not overwritten, respectively). The actual number of pixels copied from the source buffer to the destination buffer is written to the IM_RES_NUM_PIXELS field of the destination buffer. This number will be correct even if the destination buffer is too small to hold all tagged pixels (in which case the contents of the destination buffer are undefined).

If the same tag buffer is used to pack a buffer and then to unpack the resulting buffer, the packed pixels will be written to their original positions.

Note that this function can be used to process non-rectangular regions. For more details, see the *Genesis Native Library User Guide*.

The **Thread** parameter specifies the thread to which to send **imBufPack()** for execution.

The **SrcBuf** parameter specifies the source buffer. This buffer can be of any integer type, or can be a 32-bit floating-point buffer. When unpacking, this buffer must be one-dimensional.

The **TagBuf** parameter specifies the tag buffer with which to perform the operation. This must be a packed binary buffer.

The **DstBuf** parameter specifies the destination buffer. This buffer must have the same pixel size as the source buffer, and be large enough to hold the result. When packing, this buffer must be one-dimensional.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_PACK_0	Pack source pixels tagged with the value 0.
IM_PACK_1	Pack source pixels tagged with the value 1.
IM_UNPACK_0	Unpack source pixels tagged with the value 0.
IM_UNPACK_1	Unpack source pixels tagged with the value 1.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Example See *process.c* in Appendix B.

imBufPut

Sync

Multi-band

Gen-LC

Synopsis Transfer data from Host memory to a buffer.

Format **void imBufPut(Thread, Buf, Ptr)**

long Thread; Thread ID
long Buf; Destination buffer ID
void* Ptr; Address of array

Description This function transfers data from an array in Host memory to a specified buffer. If the buffer has more than one band, they are all overwritten, one after another.

The **Thread** parameter specifies the thread to which to send **imBufPut()** for execution.

The **Buf** parameter specifies the buffer in which to copy the data. This buffer can be of any data type. If it is a packed binary buffer, it must be byte-aligned.

The **Ptr** parameter specifies the address of the user-supplied array containing the data to copy. The array must be large enough to fill the buffer.

Note The Host CPU performs the transfer. If Host memory was allocated using **imBufAlloc...()**, it is faster to use **imBufCopy()** or **imBufCopyPCI()**; for these functions, the VIA drives the transfer without involving the Host CPU. Note, however, that **imBufCopy()** and **imBufCopyPCI()** can only work with Host buffers allocated with **imBufAlloc...()**.

See also **imBufPut1d()**, **imBufPut2d()**. These functions can also transfer data from Host memory to a buffer.

imBufPut1d

Sync

Multi-band

Gen-LC

Synopsis Transfer a 1D block of data from Host memory into part of a buffer.

Format **void imBufPut1d(Thread, Buf, Xstart, Xsize, Ptr)**

long Thread;	Thread ID
long Buf;	Destination buffer ID
long Xstart;	Block origin
long Xsize;	Block width
void* Ptr;	Address of array

Description This function transfers a 1D block of data from Host memory into part of a buffer. If the buffer has more than one band, they are all overwritten, one after another.

The **Thread** parameter specifies the thread to which to send **imBufPut1d()** for execution.

The **Buf** parameter specifies the buffer into which to copy the data. This buffer can be of any data type. If it is a packed binary buffer, it must be byte-aligned.

The **Xstart** parameter specifies the origin of the block, relative to the top-left corner of the destination buffer.

The **Xsize** parameter specifies the size of the block (in pixels).

The **Ptr** parameter specifies the address of the user-supplied array containing the data to copy. The array must be large enough to fill the block of the buffer.

Note The Host CPU performs the transfer. If Host memory was allocated using **imBufAlloc...()**, it is faster to use **imBufCopy()** or **imBufCopyPCI()**; for these functions, the VIA drives the transfer without involving the Host CPU. Note, however, that **imBufCopy()** and **imBufCopyPCI()** can only work with Host buffers allocated with **imBufAlloc...()**.

See also **imBufPut()**, **imBufPut2d()**. These functions can also transfer data from Host memory to a buffer.

imBufPut2d

Sync

Multi-band

Gen-LC

Synopsis Transfer a 2D block of data from Host memory into part of a buffer.

Format **void imBufPut2d(Thread, Buf, Xstart, Ystart, Xsize, Ysize, Ptr)**

long Thread;	Thread ID
long Buf;	Destination buffer ID
long Xstart;	X origin of block
long Ystart;	Y origin of block
long Xsize;	Block width
long Ysize;	Block height
void* Ptr;	Address of array

Description This function transfers a 2D block of data from Host memory into part of a buffer. If the buffer has more than one band, they are all overwritten, one after another.

The **Thread** parameter specifies the thread to which to send **imBufPut2d()** for execution.

The **Buf** parameter specifies the buffer into which to copy the data. This buffer can be of any data type. If it is a packed binary buffer, it must be byte-aligned.

The **Xstart** and **Ystart** parameters specify the x and y coordinates of the block origin, respectively. These coordinates are relative to the top-left corner of the destination buffer.

The **Xsize** and **Ysize** parameters specify the width and height of the block (in pixels), respectively.

The **Ptr** parameter specifies the address of the user-supplied array containing the data to copy. The array must be large enough to fill the block of the buffer.

Note The Host CPU performs the transfer. If Host memory was allocated using **imBufAlloc...()**, it is faster to use **imBufCopy()** or **imBufCopyPCI()**; for these functions, the VIA drives the transfer without involving the Host CPU. Note, however, that **imBufCopy()** and **imBufCopyPCI()** can only work with Host buffers allocated with **imBufAlloc...()**.

See also **imBufPut()**, **imBufPut1d()**. These functions can also transfer data from Host memory to a buffer.

imBufPutField

Async

Gen-LC

Synopsis Add or modify a buffer field.

Format **void imBufPutField(Thread, Buf, Tag, Value)**

long Thread;	Thread ID
long Buf;	Buffer ID
long Tag;	Tag of field
double Value;	Value of field

Description This function adds a field to a specified buffer, or modifies an existing field of a specified buffer.

The **Thread** parameter specifies the thread to which to send **imBufPutField()** for execution.

The **Buf** parameter specifies the buffer into which to add the field, or the buffer containing the field to modify. This buffer can be of any data type, and can have any size.

The **Tag** parameter specifies the field to add/modify. For a list of fields, see individual function descriptions. Tags can have any non-zero value, and a new field will automatically be created if it doesn't already exist.

Note that you can add your own field as long as you use tags that do not conflict with those used by the Genesis Native Library. Tags from 1 to 9999 (inclusive) are reserved for user fields.

The **Value** parameter specifies the value for the specified field.

Note A buffer's fields are not copied when a buffer is copied. Use **imBufCopyField()** to copy a buffer's fields.

imBufRemoveField

*Async**Gen-LC*

Synopsis Remove a field from a buffer.

Format **void imBufRemoveField(Thread, Buf, Tag)**

long Thread; Thread ID

long Buf; Buffer ID

long Tag; Tag of field

Description This function removes a field from a specified buffer.

The **Thread** parameter specifies the thread to which to send **imBufRemoveField()** for execution.

The **Buf** parameter specifies the buffer.

The **Tag** parameter specifies the field. For a list of fields, see individual function descriptions.

imBufRestore

Sync

Multi-band

Gen-LC

Synopsis Load data from a file into an automatically allocated buffer.

Format **void imBufRestore(Thread, FileName, Format, Location, BufPtr)**

long Thread;	Thread ID
char* FileName;	File name
long Format;	File format
long Location;	Location of the memory
long* BufPtr;	Address of buffer ID

Description This function allocates a new buffer in the specified memory bank and loads it with data from the specified file. When allocating processing or display memory, the memory is allocated on the node associated with the specified thread.

The **Thread** parameter specifies the thread to which to send **imBufRestore()** for execution.

The **FileName** parameter specifies the name of the file from which to load the buffer.

The **Format** parameter specifies the format of the specified file. It can be set to:

IM_TIFF	TIFF format.
IM_NATIVE	Genesis native format (same as TIFF, but can contain control fields in addition to data).

Note that, when loading an IM_NATIVE file, any attribute fields in the file are also loaded.

The **Location** parameter specifies where to allocate the buffer. It can be set to:

IM_PROC	Processing memory.
IM_DISP	Display memory (any available bank: red, green, or blue).
IM_DISP_RED	Red display memory.
IM_DISP_GREEN	Green display memory.
IM_DISP_BLUE	Blue display memory.
IM_DISP_OVERLAY	Overlay display memory.
IM_HOST	Host system memory (specifically, DMA memory from non-paged pool).

Note that display memory refers to off-screen display memory.

The **BufPtr** parameter specifies the address in which to return the buffer identifier. If the buffer could not be allocated, 0 is returned.

imBufSave

Sync

Multi-band

Gen-LC

Synopsis Save a buffer to a file.

Format `void imBufSave(Thread, FileName, Format, Buf)`

long Thread;	Thread ID
char* FileName;	File name
long Format;	File format
long Buf;	Buffer ID

Description This function saves a specified buffer to a specified file.

The **Thread** parameter specifies the thread to which to send **imBufSave()** for execution.

The **FileName** parameter specifies the name of the file in which to save the buffer.

The **Format** parameter specifies the format with which to save the buffer to the file. It can be set to:

IM_TIFF	TIFF format.
IM_NATIVE	Genesis native format (same as TIFF, but can contain control fields in addition to data).
IM_RAW	Raw file format (no header is saved, just the data).

Note that, when saving an IM_NATIVE file, control fields are also saved.

The **Buf** parameter specifies the buffer to save to file. This buffer can be of any data type, and can have any number of bands. If the buffer is packed binary, it must be byte-aligned.

imCamAlloc

Sync

Gen-LC

Synopsis Allocate a camera definition.

Format **void imCamAlloc(Thread, CamFile, Mode, CameraPtr)**

long Thread;	Thread ID
char* CamFile;	File containing camera definition (or NULL)
long Mode;	Operation mode
long* CameraPtr;	Address of camera definition ID

Description This function allocates a camera definition, loading its definition from the specified file. Note that all the settings (input channel, gain and reference levels, etc.) in the file are sent to the board. However, the digitizer is only programmed to these settings when a grab is issued with this camera definition. Therefore, using **imCamAlloc()** does not affect the digitizer.

Once a camera definition is allocated, you can change its settings using **imCamControl()**. Using **imCamControl()** will not change the original camera definition file.

The **Thread** parameter specifies the thread to which to send **imCamAlloc()** for execution.

The **CamFile** parameter specifies the name of the file containing the camera definition. This parameter can be set to NULL, in which case the camera definition file specified during installation will be used.

The **Mode** parameter specifies the mode of operation. This parameter must be set to IM_DEFAULT.

The **CameraPtr** parameter specifies the address in which to return the identifier of the camera definition. If the camera definition could not be allocated, 0 is returned.

Note If you are running several applications simultaneously on a Genesis system and if these applications are not dependent on a particular camera type, you should use the camera definition file that was specified during installation. This will allow the applications to most efficiently share the digitizer, since the digitizer will not be re-programmed between grabs.

imCamClone

Sync

Gen-LC

Synopsis Duplicate a camera definition.

Format **void imCamClone(Thread, Camera, Mode, NewCamPtr)**

long Thread;	Thread ID
long Camera;	Camera definition ID
long Mode;	Operation mode
long* NewCamPtr;	Address of new camera definition ID

Description This function duplicates a camera definition.

Note that this function allows you to create several IDs for the same camera without accessing the camera definition file. It is useful when you need several versions of a camera definition simultaneously, each with different settings (input channel, reference levels, etc.). You can change settings using the **imCamControl()** function.

The **Thread** parameter specifies the thread to which to send **imCamClone()** for execution.

The **Camera** parameter specifies the camera definition to duplicate.

The **Mode** parameter specifies the mode of operation. This parameter must be set to IM_DEFAULT.

The **NewCamPtr** parameter specifies the address in which to return the identifier of the new camera definition. If the new camera definition could not be allocated, 0 is returned.

imCamControl

Async

Gen-LC

Synopsis Change a setting of a camera definition.

Format **void imCamControl(Thread, Camera, Item, Value)**

long Thread; Thread ID
 long Camera; Camera definition ID
 long Item; Item to set
 double Value; Value for Item

Description This function changes a setting of a specified camera definition.

Where supported, you can specify different settings (such as gain and reference levels) for each channel. To do so, combine the #define of the required setting with one or more of the channel #defines (for example, IM_DIG_GAIN + IM_CHANNEL_1). If you change a setting without specifying a channel, the setting is changed on all channels currently selected for the camera.

Note that the digitizer is only programmed to a specific camera definition when a grab is issued with the identifier of that camera definition. Therefore, using **imCamControl()** will not affect the digitizer hardware; it will simply change a camera definition already in memory.

The **Thread** parameter specifies the thread to which to send **imCamControl()** for execution.

The **Camera** parameter specifies the camera definition.

The **Item** parameter specifies the setting to change, while the **Value** parameter specifies its value. The table below lists those settings that can be changed, and their allowable values. Note that a #define specified as IM_XXX_0/1 means you can use IM_XXX_0 or IM_XXX_1 (it does not mean use IM_XXX_0/1). In addition, if it is more convenient to use a simple integer channel number, you can convert it to the corresponding #define with the macro IM_CHANNEL(n). For example, IM_CTL_CHANNEL(3) is equivalent to IM_CTL_CHANNEL_3.

Item	Values	Meaning
IM_DIG_CHANNEL	IM_CHANNEL_0/1/2/3	Input channel. To grab from multiple channels, combine the required channels (for example, IM_CHANNEL_0+IM_CHANNEL_1+IM_CHANNEL_2).
IM_DIG_GAIN (note that you can specify different gains for each channel, if necessary)	0 - 100	Specify the gain used by the analog-to-digital converters, as a percentage.
	IM_DEFAULT	Use the gain specified in the original camera definition file.
IM_DIG_LUT_BUF	0	Use a transparent LUT on the grabbed data.
	A buffer ID	Use the specified buffer to map the grabbed data.
IM_DIG_REF_BLACK (note that you can specify different reference levels for each channel, if necessary)	0 - 100	Specify the black reference level, as a percentage.
	IM_DEFAULT	Use the reference level specified in the original camera definition file.
IM_DIG_REF_WHITE (note that you can specify different reference levels for each channel, if necessary)	0 - 100	Specify the white reference level, as a percentage.
	IM_DEFAULT	Use the reference level specified in the original camera definition file.

Note that IM_DIG_REF_WHITE and IM_DIG_REF_BLACK can be set below 0% or above 100%. This is because the 0% and 100% levels are the black and white reference levels, respectively, specified in the DCF file, rather than the minimum and maximum values supported by the hardware.

Item	Values	Meaning
IM_DIG_SIZE_X	8 - n	Adjust the width of the grabbed frame to the specified number of pixels per line.
	IM_DEFAULT	Use the number of pixels per line specified in the original camera definition file.
IM_DIG_SIZE_Y	1 - n	Adjust the height of the grabbed frame to the specified number of lines.
	IM_DEFAULT	Use the number of lines specified in the original camera definition file.
IM_DIG_SYNC_CHANNEL	IM_CHANNEL_0/1/2/3	Use the specified channel as the synchronization channel.
	IM_CHANNEL_GRAB	Use the current grab channel as the synchronization channel as well.
	IM_DEFAULT	Use the channel specified in the original camera definition file as the synchronization channel.
IM_DIG_TRIG_SOURCE	IM_SOFTWARE	Use software to trigger the grab.
	IM_HARDWARE	Use hardware to trigger the grab. To select a specific hardware trigger, combine with IM_TRIGGER1/2 (for example, IM_HARDWARE+IM_TRIGGER1).
	IM_EXPOSURE	Use an exposure timer to trigger the grab. To select a specific timer, combine with IM_TIMER1/2 (for example, IM_EXPOSURE+IM_TIMER1).
	IM_NONE	Don't grab with a trigger.
	IM_DEFAULT	Use the source specified in the original camera definition file to trigger the grab.

Item	Values	Meaning
IM_DIG_TRIG_MODE	IM_RISING_EDGE	The hardware trigger is edge sensitive and generates a positive pulse.
	IM_FALLING_EDGE	The hardware trigger is edge sensitive and generates a negative pulse.
IM_DIG_TRIG_MODE (cont.)	IM_DEFAULT	The hardware trigger is of the type specified in the original camera definition file.
	IM_ACTIVE_HIGH	The trigger (software or hardware) is level sensitive and generates a high signal. Note that a level-sensitive trigger cannot be used to start an exposure timer.
	IM_ACTIVE_LOW	The trigger (software or hardware) is level sensitive and generates a low signal. Note that a level-sensitive trigger cannot be used to start an exposure timer.
IM_DIG_USER_IN_FORMAT	IM_TTL	Enable the TTL receivers for trigger and user inputs.
	IM_RS422	Enable the RS-422 receivers for trigger and user inputs.
	IM_DEFAULT	Enable the receivers specified in the original camera definition file for trigger and user inputs.
	IM_DISABLE	Disable trigger and user inputs.

Item	Values	Meaning
IM_DIG_USER_OUT	0 or 1	Set an output line on the grab section low (0) or high (1). To select a specific user bit, combine with IM_BIT0/1/2/3/4 (for example, IM_DIG_USER_OUT+IM_BIT0). Note that IM_BIT2/3/4 correspond to camera control bits 0/1/2, respectively, on the connector. For information on pinout and signal descriptions, see the <i>Genesis Installation and Hardware Reference</i> .
IM_DIG_USER_OUT_FORMAT	IM_TTL	Enable the TTL drivers for exposure and user outputs.
	IM_RS422	Enable the RS-422 drivers for exposure and user outputs.
	IM_DEFAULT	Enable the drivers specified in the original camera definition file for exposure and user outputs.
	IM_DISABLE	Disable exposure and user outputs.

The following settings are only relevant if you are using the exposure timers. To associate a specific timer with these settings, combine the required item with IM_TIMER1/2 (for example, IM_DIG_EXP_TIME+IM_TIMER2). If you don't select a specific timer, timer 1 is used.

Item	Values	Meaning
IM_DIG_EXP_TIME	any floating-point value > 0	The exposure time of the output signal, in seconds.
IM_DIG_EXP_DELAY	any floating-point value > 0	The delay before starting the output signal, in seconds.

IM_DIG_EXP_SOURCE	IM_SOFTWARE	Use software to start the timer.
	IM_HARDWARE	Use a hardware trigger input to start the timer. To select a specific trigger, combine with IM_TRIGGER1/2 (for example, IM_HARDWARE+IM_TRIGGER1).
	IM_VSYNC	Use the vertical sync of the camera signal to start the timer.
	IM_HSYNC	Use the horizontal sync of the camera signal to start the timer.
IM_DIG_EXP_SOURCE (cont.)	IM_CONTINUOUS	Restart the timer automatically as soon as it elapses.
	IM_TIMER2	Use timer 2 to start timer 1 (this only applies for timer 1).
	IM_TIMER1	Use timer 1 to start timer 2 (this only applies for timer 2).
	IM_NONE	Don't use the exposure timer.
	IM_DEFAULT	Use the source specified in the original camera definition file to start the timer.
IM_DIG_EXP_MODE (this setting only applies if you are generating an exposure output)	IM_ACTIVE_HIGH	Generate a positive exposure pulse.
	IM_ACTIVE_LOW	Generate a negative exposure pulse.
	IM_DEFAULT	Generate a pulse of the type specified in the original camera definition file.

imCamFree*Async**Gen-LC*

Synopsis Free a camera definition.

Format **void imCamFree(Thread, Camera)**

long Thread; Thread ID

long Camera; Camera definition ID

Description This function deallocates a previously allocated camera definition.

Before you deallocate a camera definition, you should ensure that all operations involving the camera have completed.

The **Thread** parameter specifies the thread to which to send **imCamFree()** for execution.

The **Camera** parameter specifies the camera definition to deallocate.

imCamInquire

Sync

Gen-LC

Synopsis Inquire about a camera definition setting.

Format **long imCamInquire(Thread, Camera, Item, ValuePtr)**

long Thread; Thread ID
 long Camera; Camera definition ID
 long Item; Setting about which to inquire
 void* ValuePtr; Address of return value (or NULL)

Description This function inquires about a setting of a specified camera definition.

The **Thread** parameter specifies the thread to which to send **imCamInquire()** for execution.

The **Camera** parameter specifies the camera definition about which to inquire.

The **Item** parameter specifies the setting about which to inquire. Note that certain settings can be combined with a specific channel, timer, etc. For example, IM_DIG_GAIN+IM_CHANNEL_0 will return the gain level for channel 1. See **imCamControl()** for a list of settings that can be combined.

Item can be set to:

IM_DIG_CHANNEL	Input channel.
IM_DIG_EXP_DELAY	Delay before starting the output signal, in seconds; returned as type double.
IM_DIG_EXP_MODE	Type of exposure pulse generated.
IM_DIG_EXP_SOURCE	Timer source.
IM_DIG_EXP_TIME	Exposure time of the output signal, in seconds; returned as type double.
IM_DIG_GAIN	Gain level; returned as type double.
IM_DIG_INPUT_MODE	Video input connector that is attached to the camera: IM_ANALOG or IM_DIGITAL.
IM_DIG_MAX_CHANNELS	Maximum number of channels for the current camera.
IM_DIG_NUM_BANDS	Number of bands grabbed.
IM_DIG_REF_BLACK	Black reference level; returned as type double.
IM_DIG_REF_WHITE	White reference level; returned as type double.

IM_DIG_SCAN_MODE	Type of scan mode used: IM_PROGRESSIVE, IM_INTERLACED, or IM_LINE.
IM_DIG_SIZE_BIT	Number of bits per pixel grabbed.
IM_DIG_SIZE_X	Width of grabbed frame (pixels per line).
IM_DIG_SIZE_Y	Height of grabbed frame (lines per frame).
IM_DIG_SYNC_CHANNEL	Synchronization channel.
IM_DIG_TRIG_MODE	When using an edge-sensitive trigger, whether to wait for a positive or negative pulse, or for the pulse specified in the original camera definition file: IM_RISING_EDGE, IM_FALLING_EDGE, or IM_DEFAULT. When using a level-sensitive trigger, whether to wait for a low or high signal: IM_ACTIVE_LOW or IM_ACTIVE_HIGH.
IM_DIG_TRIG_SOURCE	Trigger source.
IM_DIG_TYPE	Type of buffer needed for grabbed data. Note that this can be passed to imBufAlloc...() .
IM_DIG_USER_IN_FORMAT	Type of receivers enabled for digital trigger and user inputs: IM_TTL, IM_RS422, IM_DISABLE.
IM_DIG_USER_OUT	Whether output line on the grab section is low (0) or high (1).
IM_DIG_USER_OUT_FORMAT	Type of drivers enabled for digital exposure and user outputs: IM_TTL, IM_RS422, or IM_DISABLE.

The **ValuePtr** parameter specifies the address in which to return the value of the inquired setting. Unless otherwise stated, **ValuePtr** should be the address of a long. Note that, since **imCamInquire()** also returns this value, **ValuePtr** can be set to NULL.

Return value The returned value is the value of the inquired setting, cast to long if necessary.

imCamSave

Sync

Gen-LC

Synopsis Save a camera definition to a file.

Format **void imCamSave(Thread, CamFile, Camera)**

long Thread; Thread ID

char* CamFile; File name

long Camera; Camera definition ID

Description This function saves a camera definition to a specified file.

The **Thread** parameter specifies the thread to which to send **imCamSave()** for execution.

The **CamFile** parameter specifies the name of the file in which to save the camera definition.

The **Camera** parameter specifies the camera definition.

imCurAlloc

Sync

Gen-LC

Synopsis Allocate a new cursor.

Format **void imCurAlloc(Dev, Mode, CursorPtr)**

long Dev;	Device ID (or 0)
long Mode;	Cursor mode
long *CursorPtr;	Address of cursor ID

Description This function allocates a new cursor. You can allocate an unlimited number of cursors, but only one can be loaded into the hardware at a time.

After the initial allocation, the cursor's attributes are undefined. Use the **imCurDefine()** and **imCurSetColor()** functions to set these attributes. Then, call **imCurSelect()** to load the function definitions into the physical hardware and select the software copy of the cursor to the display and **imCurEnable()** to make the hardware cursor visible. You are responsible for tracking and moving the cursor using the **imCurGetPosition()** and **imCurSetPosition()** functions.

The **Dev** parameter specifies the device on which to allocate the cursor. You can pass 0 if your system has only one display.

The **Mode** parameter specified the cursor mode. Currently, only the default mode is supported; therefore, pass IM_DEFAULT.

The **CursorPtr** parameter specifies the address in which to return the cursor identifier. If cursor allocation fails, the **imCurAlloc()** function will return a cursor ID of 0; however, no error will be logged.

imCurDefine

[Sync](#)
[Gen-LC](#)

Synopsis Defines the cursor's shape.

Format **void imCurDefine(Dev, Cursor, SizeX, SizeY, HotX, HotY, DataPtr)**

long Dev;	Device ID (or 0)
long Cursor;	Cursor ID
long SizeX;	Width of cursor
long SizeY;	Height of cursor
long HotX;	X position of hot spot
long HotY;	Y position of hot spot
unsigned char *DataPtr;	Address of array of cursor values

Description This function defines the specified cursor's shape (only the software copy that appears on-screen, not the actual hardware).

Each cursor pixel has a color value. A cursor pixel with a value of 0 is always transparent. Cursor pixel values of 1, 2, and 3 represent the three user-definable colors. The three cursor colors are set by specifying the red, green, and blue color component using the **imCurSetColor()** function.

The maximum size for a cursor is 64x64, and the hotspot is defined relative to the top-left corner (0,0). Valid positions for the hotspot are between (0,0) and (63,63), inclusive.

You can allocate and define any number of cursors. Select the cursor to the display by calling **imCurSelect()**.

The **Dev** parameter specifies the device on which to define the cursor's attributes. This should be the same device identifier as the one on which the cursor is allocated. You can pass 0 if your system has only one display device.

The **Cursor** parameter specifies the cursor identifier.

The **SizeX** and **SizeY** parameters specify the width and height of the specified cursor, respectively. The possible range of values is from 1 to 64.

The **HotX** and **HotY** parameters specify the X and Y coordinate of the specified cursor's hotspot, respectively. The possible range of values is from 0 to 63. However, it is prudent to set hotspot coordinates that are within the region defined by **SizeX** and **SizeY**.

The **DataPtr** parameter specifies the address of the array defining the cursor's shape. **DataPtr** must point to **SizeX*SizeY** bytes of data, where each byte represents one cursor pixel.

Example See the *curdemo.c* that is included in the Genesis Native Library *\EXAMPLES\HOST\MISC* directory.

imCurEnable

Sync

Gen-LC

Synopsis

The **imCurEnable** function enables or disables the current hardware cursor on the specified display device.

Format

void imCurEnable(Dev, Flag)

long Dev;

long Flag;

Device ID (or 0)

Enable flag

Description

This function enables (makes visible) or disables (hides) the specified hardware cursor.

The **Dev** parameter specifies the device on which to enable/disable the cursor. You can pass 0 if your system has only one display device.

The **Flag** parameter specifies the new state of the cursor. This flag can be set to:

Value	Description
IM_DISABLE	Disable (hide) the current cursor.
IM_ENABLE	Enable (make visible) the current cursor.

Note

Since the hardware cursor is initially undefined, you should only call this function after you have selected a cursor to the display (**imCurSelect()**).

imCurFree

Sync

Gen-LC

Synopsis Free the specified cursor.

Format **void imCurFree(Dev, Cursor)**

long Dev; Device ID (or 0)

long Cursor; Cursor ID

Description This function frees the specified cursor.

The **Dev** parameter specifies the device on which to free the cursor. You can pass 0 if your system has only one display device.

The **Cursor** parameter specifies the identifier of the cursor.

Note This function does not affect the hardware copy of the cursor. For example, if the cursor is currently selected to the screen and enabled, freeing the cursor (**imCurFree()**) will not disable the cursor in hardware. That is, the hardware cursor will still be visible.

imCurGetPosition

Sync

Gen-LC

Synopsis Get the current cursor's position.

Format **void imCurSetPosition(Dev, PosXPtr, PosYPtr)**

long Dev;	Device ID (or 0)
long *PosXPtr;	Address of cursor's X position
long *PosYPtr;	Address of cursor's Y position

Description This function returns the specified cursor's current position.

The **Dev** parameter specifies the device on which to get the cursor's position. You can pass 0 if your system has only one display device.

The **PosXPtr** parameter specifies the address in which to return the cursor's X-coordinate.

The **PosYPtr** parameter specifies the address in which to return the cursor's Y-coordinate.

imCurSelect

Sync

Gen-LC

Synopsis Select the specified cursor as the current cursor.

Format void imCurSelect(Dev, Cursor)

long Dev;	Device ID (or 0)
-----------	------------------

long Cursor;	Cursor ID (or 0)
--------------	------------------

Description This function selects the specified cursor as the current hardware cursor, overriding any existing cursor. After you select a cursor, the cursor position is initially undefined. You should call **imCurSetPosition()** to set the position at which to display the cursor.

The **Dev** parameter specifies the device on which to select the cursor. You can pass 0 if your system has only one display device.

The **Cursor** parameter specifies the identifier of the cursor. If you pass 0 as the cursor ID, the default (arrow) cursor is selected.

Note Since the hardware cursor is initially undefined, you should call this function before you enable the cursor (**imCurEnable()**) for the first time.

imCurSetColor

Sync

Gen-LC

Synopsis Set the specified cursor's colors.

Format **void imCurSetColor(Dev, Cursor, Color, Red, Green, Blue)**

long Dev;	Device ID (or 0)
long Cursor;	Cursor ID (or 0)
long Color;	Color to change
long Red;	Red component of new color
long Green;	Green component of new color
long Blue;	Blue component of new color

Description This function selects the specified cursor's colors.

The **Dev** parameter specifies the device on which to set the cursor color, and is only needed when the hardware cursor is to be updated (cursor ID is 0). You can pass 0 if your system has only one display device.

The **Cursor** parameter specifies the cursor identifier. If you provide a valid cursor ID, only the software cursor structure will be changed, and you will not see the effect until you select the cursor to the display (**imCurSelect()**). You can provide 0 for the cursor ID, in which case the hardware will be updated immediately.

The **Color** parameter specifies the user-defined color value to change. The three user-defined color values are 1, 2, or 3.

The **Red** parameter specifies the red component of the cursor's color. The supported range of values is from 0 to 255.

The **Green** parameter specifies the green component of the cursor's color. The supported range of values is from 0 to 255.

The **Blue** parameter specifies the blue component of the cursor's color. The supported range of values is from 0 to 255.

Note You cannot change color 0; it is always transparent.

imCurSetPosition

Sync

Gen-LC

Synopsis Set the current cursor's position.

Format **void imCurSetPosition(Dev, PosX, PosY)**

long Dev;	Device ID (or 0)
long PosX;	Cursor's X coordinate
long PosY;	Cursor's Y coordinate

Description This function sets the current hardware cursor's horizontal and vertical position on screen.

You can set the position of the cursor whether or not it is currently visible.

Also, keep in mind that you are responsible for keeping track of the cursor's position. When the display is panned or zoomed, you should call **imCurSetPosition()** to reset the cursor's position. Similarly, if you specify a position that is not valid for the current screen resolution, the cursor will not be visible.

The **Dev** parameter specifies the device on which to set the cursor position. You can pass 0 if your system has only one display device.

The **PosX** parameter specifies the cursor's X-coordinate. The valid range is from 0 to the maximum screen resolution width.

The **PosY** parameter specifies the cursor's Y-coordinate. The valid range is from 0 to the maximum screen resolution height.

imDevAlloc

Sync

Gen-LC

Synopsis Allocate a device.

Format **void imDevAlloc(System, Node, ShellFile, Mode, DevPtr)**

long System;	System number
long Node;	Node number
char* ShellFile;	File name of 'C80 code (or NULL)
long Mode;	Operation mode
long* DevPtr;	Address of device ID

Description This function allocates a device. A device refers to a specific node on a specific system. If the board on which the specified node is located has a display section, then the device also includes the display section.

The **System** parameter specifies the number of the system (0, 1, 2, etc.) on which the node is located.

The **Node** parameter specifies the number of the node (0, 1, 2, etc.).

The **ShellFile** parameter specifies the name of the file containing the 'C80 code to download to the device when it is allocated. This parameter can be set to NULL, in which case the file specified during installation is downloaded.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_DEFAULT	Download code if necessary; otherwise, just establish communications with the device.
IM_DOWNLOAD	Download code unconditionally (this also resets all hardware and software).

The **DevPtr** parameter specifies the address in which to return the device identifier. If the device could not be allocated, 0 is returned.

imDevFree

*Async**Gen-LC*

Synopsis Free a device.

Format **void imDevFree(Dev)**

long Dev; Device ID

Description This function deallocates a previously allocated device.

Note that any functions still running on the device will not be affected.

The **Dev** parameter specifies the device to deallocate.

imDevInquire

Sync

Gen-LC

Synopsis Inquire about a device attribute.

Format **long imDevInquire(Dev, Item, ValuePtr)**

long Dev; Device ID
 long Item; Attribute about which to inquire
 void* ValuePtr; Address of return value (or NULL)

Description This function inquires about an attribute of a specified device.

The **Dev** parameter specifies the device.

The **Item** parameter specifies the attribute about which to inquire. It can be set to:

IM_DEV_MVP	Whether the MVP ('C80) is present (1) or not present (0).
IM_DEV_MVP_SPEED	MVP ('C80) clock speed in MHz: 50 or 60.
IM_DEV_VIA_PRIMARY	Whether the primary VIA is present (1) or not present (0).
IM_DEV_VIA_DISPLAY	Whether the display VIA is present (1) or not present (0).
IM_DEV_ACCELERATOR	Whether the NOA is present (1) or not present (0).
IM_DEV_MEM_PROC	Amount of processing memory, in MBytes: 0, 8, 16, 32, or 64.
IM_DEV_MEM_DISP	Amount of display memory, in MBytes: 0, 2, or 6.
IM_DEV_MEM_OVERLAY	Amount of overlay display memory, in MBytes: 0 or 2.
IM_DEV_OWNER_SYSTEM	System to which the device belongs: 0 - n.
IM_DEV_OWNER_NODE	Node on which the device was allocated: 0 - n.
IM_DEV_MEM_HOST	Total amount of Host DMA memory, in bytes.
IM_DEV_FREE_MEM_HOST	Total amount of free Host DMA memory, in bytes (not necessarily contiguous).

IM_DEV_MSG_TOTAL	Total number of message buffers. This defines the total number of commands that can be queued to the node (see the following note).
IM_DEV_MSG_FREE	Number of free message buffers. This defines the number of additional commands that can still be queued to the node (see the following note).
IM_DEV_FREE_MEM_PROC	Total amount of free processing memory (not necessarily contiguous).
IM_DEV_FREE_MEM_DISP	Total amount of free off-screen display memory (not necessarily contiguous).
IM_DEV_PRODUCT_ID	Type of device: IM_DEV_GENESIS (Genesis main board), IM_DEV_GENESIS_PRO (Genesis processor board), IM_DEV_GENESIS_LC (Genesis-LC).
IM_DEV_CUSTOMER_PRODUCT_ID	Product ID of device. This is reserved for customer use. It is not available for a Genesis processor board.

The **ValuePtr** parameter specifies the address in which to return the value of the inquired attribute. Unless otherwise stated, **ValuePtr** should be the address of a long. Note that, since **imDevInquire()** also returns this value, **ValuePtr** can be set to NULL.

Note Most Genesis functions send a single message from the Host to the 'C80. Each message requires one message buffer, which remains allocated to that command until it completes. The IM_DEV_MSG_TOTAL and IM_DEV_MSG_FREE items can be used to inquire about the total number of message buffers and the number of free message buffers, respectively. The total number of message buffers can be changed at Shell download time through the *CoffLoadOption* entry in the *genesis.ini* file. The message buffers on each node are shared by all applications using that node. It is not a good idea to let the number of free messages get too small since some applications might experience delays while they wait for a free message buffer.

Return Value The returned value is the value of the inquired attribute, cast to long if necessary.

imDigAlloc

Sync

Gen-LC

Synopsis Allocate a digitizer.

Format **void imDigAlloc(Thread, System, Digitizer, Mode, DigPtr)**

long Thread;	Thread ID
long System;	System number
long Digitizer;	Digitizer number
long Mode;	Operation mode
long DigPtr;	Address of digitizer ID

Description This function assigns an identifier to a digitizer.

Note that this function is normally only needed if you have more than one Genesis digitizer, since you sometimes have to specify which digitizer a function should use. If your Genesis system has just one digitizer, you do not need to use this function.

The **Thread** parameter specifies the thread to which to send **imDigAlloc()** for execution.

The **System** parameter specifies the number of the system (0, 1, 2, etc.) on which the digitizer is located.

The **Digitizer** parameter specifies the number of the digitizer (0, 1, 2, etc.).

The **Mode** parameter specifies the mode of operation. This parameter must be set to IM_DEFAULT.

The **DigPtr** parameter specifies the address in which to return the digitizer identifier. If the digitizer could not be allocated, 0 is returned.

imDigCapture

Async

Gen-LC

Synopsis Enable a synchronized or software-triggered grab.

Format **void imDigCapture(Thread, Dig, Cam, Mode)**

long Thread; Thread ID
long Dig; Digitizer ID (or 0)
long Cam; Camera ID
long Mode; Mode of operation

Description This function enables a synchronized or software-triggered grab. You must ensure that the digitizer is ready to grab before calling **imDigCapture()**, and you should not have changed the camera definition since it was used in **imDigGrab()**.

For synchronized grabs, **imDigCapture()** automatically enables the capture regardless of the trigger source. It simultaneously enables any exposure timers that are also being used, so that any exposure outputs will be generated only for the frame that is actually captured (avoiding the possibility of, for example, capturing a frame but failing to fire a strobe, or vice versa).

The **Thread** parameter specifies the thread to which to send **imDigCapture()** for execution.

The **Dig** parameter specifies the digitizer. If your system has just one digitizer, you can set this parameter to 0.

The **Cam** parameter specifies the camera definition used in **imDigGrab()**.

The **Mode** parameter specifies the mode of operation. This parameter must be set to IM_ENABLE.

Example The following code grabs to two nodes at the same time. A synchronized grab is required to ensure that the same frame is grabbed to both nodes.

```
/* Select synchronized capture mode */
imBufPutField(Thread1, ControlBuf, IM_CTL_CAPTURE_MODE, IM_SYNCHRONIZED);
imSyncHost(Thread1, 0, IM_COMPLETED);

/* Queue both grabs in synchronized mode */
imDigGrab(Thread1, 0, Camera, Buf1, 1, ControlBuf, OSB1);
imDigGrab(Thread2, 0, Camera, Buf2, 1, ControlBuf, OSB2);

/* Wait until both nodes are ready to grab */
imSyncThread(Thread3, OSB1, IM_READY);
imSyncThread(Thread3, OSB2, IM_READY);

/* Now enable the capture */
imDigCapture(Thread3, 0, Camera, IM_ENABLE);
```

The following code is an example of a software-triggered grab.

```
imCamAlloc(Thread1, NULL, IM_DEFAULT, &Camera);
imCamControl(Thread1, Camera, IM_DIG_TRIG_SOURCE, IM_SOFTWARE);
imDigGrab(Thread1, 0, Camera, DstBuf, 1, 0, 0); /* waits for software trigger */
.
.
imDigCapture(Thread2, 0, Camera, IM_ENABLE); /* give software trigger */
```

imDigControl

Async

Gen-LC

Synopsis Set a digitizer attribute.

Format **void imDigControl(Thread, Dig, Item, Value)**

 long Thread; Thread ID

 long Dig; Digitizer ID (or 0)

 long Item; Attribute to set

 double Value; Attribute value

Description This function sets an attribute of a specified digitizer.

 This function programs the digitizer directly. Therefore, using **imDigControl()** will interfere with other applications also using the digitizer. Since many digitizer attributes can be set using **imCamControl()** and since **imCamControl()** does not program the digitizer directly, it is always better to use **imCamControl()**, if possible.

 The **Thread** parameter specifies the thread to which to send **imDigControl()** for execution.

 The **Dig** parameter specifies the digitizer. If your system has just one digitizer, you can set this parameter to 0.

 The **Item** parameter specifies the attribute, while the **Value** parameter specifies the value for this attribute. The table below lists those attributes that can be set, and their allowable values. Note that a #define specified as IM_XXX_0/1 means that you can use IM_XXX_0 or IM_XXX_1 (it does not mean use IM_XXX_0/1).

Item	Values	Meaning
IM_DIG_TRIGGER	IM_ENABLE	Enable a software-triggered grab. For an edge-sensitive trigger, the required pulse is generated. For a level-sensitive trigger, the required level is set.
	IM_DISABLE	Disable a level-sensitive software triggered grab.

IM_DIG_USER_OUT	0 or 1	<p>Set an output line on the grab section to low (0) or high (1). To select a specific user bit, combine with IM_BIT0/1/2/3/4 (for example, IM_DIG_USER_OUT+IM_BIT0).</p> <p>Note that IM_BIT2/3/4 correspond to camera control bits 0/1/2, respectively, on the connector. For information on pinout and signal descriptions, see the <i>Genesis Installation and Hardware Reference</i>.</p>
IM_DIG_USER_IN_FORMAT	IM_TTL	Enable the TTL receivers for trigger and user inputs.
	IM_RS422	Enable the RS-422 receivers for trigger and user inputs.
	IM_DEFAULT	Enable the receivers specified in the original camera definition file for trigger and user inputs.
	IM_DISABLE	Disable trigger and user inputs.
IM_DIG_USER_OUT_FORMAT	IM_TTL	Enable the TTL drivers for exposure and user outputs.
	IM_RS422	Enable the RS-422 drivers for exposure and user outputs.
	IM_DEFAULT	Enable the drivers specified in the original camera definition file for exposure and user outputs.
	IM_DISABLE	Disable exposure and user outputs.

The following setting is only relevant if you are using the exposure timers. To associate a specific timer with this setting, combine with IM_TIMER1/2 (for example, IM_DIG_EXPOSURE+IM_TIMER2). If you don't select a specific timer, timer 1 is used.

Item	Values	Meaning
IM_DIG_EXPOSURE	IM_ENABLE	Start the exposure timer if source is IM_SOFTWARE (or IM_CONTINUOUS).
	IM_DISABLE	Stop the exposure timer if source is IM_CONTINUOUS.

See also **imCamControl()**. Most digitizer attributes should be set using **imCamControl()**, since this function does not program the digitizer directly.

imDigFree

Async

Gen-LC

Synopsis Free a digitizer.

Format **void imDigFree(Thread, Dig)**

long Thread; Thread ID

long Dig; Digitizer ID

Description This function deallocates a previously allocated digitizer.

The **Thread** parameter specifies the thread to which to send **imDigFree()** for execution.

The **Dig** parameter specifies the digitizer to deallocate.

imDigGrab

Async

Multi-band

Gen-LC

Synopsis Grab into a buffer.

Format **void imDigGrab(Thread, Dig, Cam, Buf, Count, Control, OSB)**

long Thread;	Thread ID
long Dig;	Digitizer ID (or 0)
long Cam;	Camera ID
long Buf;	Buffer ID
long Count;	Number of frames, fields, or lines to grab
long Control;	Control buffer ID (or 0)
long OSB;	OSB ID (or 0)

Description This function grabs frames or fields of data from a specified camera to a specified destination buffer. You can grab a specific number of frames or fields, or continuously grab frames until you call **imThrHalt()**.

Note that you can also use **imDigGrab()** to grab from the VMChannel, instead of grabbing from a camera.

The **Thread** parameter specifies the thread to which to send **imDigGrab()** for execution.

The **Dig** parameter specifies the digitizer from which to grab. If your system has just one digitizer, you can set this parameter to 0.

The **Cam** parameter specifies the camera definition with which to grab. If you are grabbing from the VMChannel, set this parameter to IM_VM_CHANNEL.

The **Buf** parameter specifies the destination buffer. This buffer should be of an appropriate pixel depth to hold the grabbed data. In addition, it should be a multi-band buffer if you are grabbing from several channels simultaneously (for example, if you are grabbing from a color camera).

The **Count** parameter specifies the number of frames, fields, or lines to grab. (You specify whether to grab frames, fields, or lines using the IM_CTL_COUNT_MODE field of the control buffer). The **Count** parameter can be set to the specified number, or to IM_CONTINUOUS, which will continuously grab frames until you call **imThrHalt()**.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imDigGrab()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_SETUP	IM_DEFAULT	Perform a full setup before grabbing.
	IM_VIA_ONLY	Reprogram only the VIA registers before grabbing. Skip programming of the grab module (the camera must be compatible with the previous grab).
	IM_ADDRESS_ONLY	Reprogram only those VIA registers which control the destination buffer address before grabbing. Skip programming of most VIA registers and the grab module.
IM_CTL_ADDR_MODE	IM_DEFAULT	Write data according to the type of scanning that your camera uses (progressive or interlaced).
	IM_PROGRESSIVE	Write data using progressive scanning (even if your camera uses interlaced scanning).
IM_CTL_BYTE_EXT	IM_DEFAULT	Extract the most-significant 8 bits, when grabbing at a pixel depth greater than 8 bits into an 8-bit destination buffer.
	10, 12, or 14	Extract the most-significant 8 bits, when grabbing at the specified pixel depth into an 8-bit destination buffer.
IM_CTL_BYTE_SWAP	IM_DISABLE	No effect.
	IM_ENABLE	Swap the 1st and 3rd bytes of each pixel (to convert, for example, RGB or RGBa color images to BGR or BGRa color images).

IM_CTL_CAPTURE_MODE	IM_DEFAULT	Grab is started in normal way.
	IM_SYNCHRONIZED	Grab must be enabled by calling imDigCapture() .
IM_CTL_CHANNEL	IM_DEFAULT	Grab from the channel(s) specified in the camera definition.
	IM_CHANNEL_0, IM_CHANNEL_1, IM_CHANNEL_2, or IM_CHANNEL_3	Grab from the specified channel(s). To specify more than one channel, combine the #defines i.e. IM_CHANNEL_0+ IM_CHANNEL_1+ IM_CHANNEL_2.
IM_CTL_COUNT_MODE	IM_DEFAULT	Grab the number of frames specified by the Count parameter (or, the number of lines, if you are using a line-scan camera).
	IM_FIELD	Grab the number of fields specified by the Count parameter. (This option only applies if your camera uses interlaced scanning).
	IM_LINE	Grab the number of lines specified by the Count parameter (for line-scan cameras).
IM_CTL_DIR_X	IM_FORWARD	Grab left to right.
	IM_REVERSE	Grab right to left.
IM_CTL_DIR_Y	IM_FORWARD	Grab top to bottom.
	IM_REVERSE	Grab bottom to top.
IM_CTL_STOP_X	IM_DEFAULT	Grab all subsequent columns.
	0 - n	Grab until the specified column.
IM_CTL_STOP_Y	IM_DEFAULT	Grab all subsequent lines.
	0 - n	Grab until the specified line.
IM_CTL_STREAM_ID	0 - 14	Grab from the specified VM stream, instead of from a camera. The Cam parameter must be set to IM_VM_CHANNEL.

IM_CTL_SUBSAMP_X	1 - 16	Grab only every <i>n</i> th column.
IM_CTL_SUBSAMP_Y	1 - 16	Grab only every <i>n</i> th row.
IM_CTL_TAG_BUF	0	No effect.
	A buffer ID	Grab with tag (using the specified buffer as a tag buffer).
IM_CTL_WRTMSK	0 - 0xFFFFFFFF	Don't overwrite bit planes in the destination buffer if the corresponding mask bit is 0. This option can only be used if you are grabbing to the display. The required value must be specified using 24 bits: the high 8 bits apply to the red buffer, the next 8 bits to the green buffer, and the low 8 bits to the blue buffer.
IM_CTL_ZOOM_X	1, 2, or 4	Replicate each column <i>n</i> times before writing to the destination buffer.
IM_CTL_ZOOM_Y	1, 2, or 4	Replicate each row <i>n</i> times before writing to the destination buffer.
IM_CTL_GRAB_MODE	IM_SYNCHRONOUS	Block the thread to which imDigGrab() is sent, until the grab finishes.
	IM_ASYNCHRONOUS	Don't block the thread.
IM_CTL_LINE_INT	IM_DISABLE	Disable line interrupts.
	0 - n	Interrupt once after the specified line.
	IM_CONTINUOUS	Interrupt after every line, or after every few lines if the line rate is too high. By default, the interval between interrupts is chosen automatically. You can force a specific interval using the IM_CTL_LINE_INT_STEP field.

IM_CTL_LINE_INT_FIELD (this field only applies if you are interrupting once and your camera uses interlaced scanning)	IM_DEFAULT	Interrupt on the second field if grabbing two fields; otherwise, interrupt on the first field.
	IM_FIRST	Interrupt on the first field.
	IM_BOTH	Interrupt on both fields.
IM_CTL_LINE_INT_STEP (this field only applies if IM_CTL_LINE_INT is set to IM_CONTINUOUS)	IM_DEFAULT	Automatically choose the interval between interrupts.
	1 - n	Force interrupts every specified number of lines.
IM_CTL_START_FIELD (this field only applies if your camera uses interlaced scanning)	IM_DEFAULT	Start grabbing on the field specified in the camera definition.
	IM_ODD	Start grabbing on the next odd field.
	IM_EVEN	Start grabbing on the next even field.
	IM_NEXT	Start grabbing on the very next field.
IM_CTL_START_X	0 - n	Start grabbing from the specified column.
IM_CTL_START_Y	0 - n	Start grabbing from the specified line.

When grabbing directly to a buffer across the primary PCI bus (for example, when grabbing to a Host buffer), a small, local work buffer (local to the VIA performing the grab) is used to store the data temporarily before it is sent to its final destination. The larger this buffer, the less chance there is of a data overrun when the PCI bus is heavily loaded. It can also be useful (especially on Genesis-LC) to copy this buffer to the display in order to view the live image as it is being grabbed.

To control the size and location of the local work buffer, add the following field on the control buffer:

IM_CTL_WORK_BUFFER	0	Use the default work buffer.
	BufId	Use the specified buffer as a work buffer.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note that you must specify an operation status block when using line interrupts. The current grab line will be updated in the operation status block each time an interrupt occurs.

imDigInquire

Sync

Gen-LC

Synopsis Inquire about a digitizer attribute.

Format **long imDigInquire(Thread, Dig, Item, ValuePtr)**

long Thread; Thread ID
 long Dig; Digitizer ID (or 0)
 long Item; Attribute about which to inquire
 void *ValuePtr; Address of return value (or NULL)

Description This function inquires about an attribute of a specified digitizer.

The **Thread** parameter specifies the thread to which to send **imDigInquire()** for execution.

The **Dig** parameter specifies the digitizer. If your system has just one digitizer, you can set this parameter to 0.

The **Item** parameter specifies the attribute about which to inquire. It can be set to:

IM_DIG_USER_IN Whether input line on the grab section is low (0) or high (1). To select a specific user bit, combine with IM_BIT0/1 (for example, IM_DIG_USER_IN+IM_BIT1 will return the current value of input line 1).

The **ValuePtr** parameter specifies the address in which to return the value of the inquired attribute. Unless otherwise stated, **ValuePtr** should be the address of a long. Note that, since **imDigInquire()** also returns the value of the inquired attribute, **ValuePtr** can be set to NULL.

Return value The returned value is the value of the inquired attribute, cast to long if necessary.

See also **imCamInquire()**. This function should be used to inquire about most digitizer attributes.

imDispAlloc

Sync

Gen-LC

Synopsis Allocate a display.

Format **void imDispAlloc(Thread, System, Display, DispFile, Mode, DispPtr)**

long Thread;	Thread ID
long System;	System number
long Display;	Display number
char* DispFile;	Display configuration file (or NULL)
long Mode;	Operation mode
long* DispPtr;	Address of display ID

Description This function assigns an identifier to a display.

Note that this function is normally only needed if you have more than one Genesis display section, since you sometimes have to specify which display a function should use. If your Genesis system has just one display, you do not need to use this function.

The **Thread** parameter specifies the thread to which to send **imDispAlloc()** for execution.

The **System** parameter specifies the number of the system (0, 1, 2, etc.) on which the display is located.

The **Display** parameter specifies the number of the display (0, 1, 2, etc.).

The **DispFile** parameter specifies the name of a display configuration format (.vcf) file. This parameter can be set to NULL, in which case the .vcf file specified during installation will be used. Note that the *readme.txt* file in the \GENESIS\VCF directory specifies the resolution set with each corresponding .vcf file.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_DEFAULT	Initialize the display only if not already done.
IM_DOWNLOAD	Initialize the display by downloading the specified <i>.vcf</i> file.

The **DispPtr** parameter specifies the address in which to return the display identifier. If the display could not be allocated, 0 is returned.

Note You should generally set the **DispFile** parameter to NULL and the **Mode** parameter to IM_DEFAULT. This will allow several applications to share the display simultaneously.

imDispControl

AsyncGen-LC

Synopsis

Set display attributes.

Format

void imDispControl(Thread, Disp, Control, Mode)

long Thread; Thread ID

long Disp; Display ID (or 0)

long Control; Control buffer ID

long Mode; Operation mode

Description

This function sets attributes of the specified display.

The **Thread** parameter specifies the thread to which to send **imDispControl()** for execution.

The **Disp** parameter specifies the display. If your system has just one display, you can set this parameter to 0.

The **Control** parameter specifies the control buffer with which to perform the function. For **imDispControl()**, the control buffer specifies which attributes of the display to set, and their values. Relevant fields are listed below. Any fields not added will not affect the display.

Field	Values	Meaning
IM_DISP_BUF	0	Display from the memory base address.
	A display buffer ID	Display from the origin of the specified buffer.
IM_DISP_KEY_MODE	IM_KEY_OFF	Show only the overlay frame buffer.
	IM_KEY_ALWAYS	Show only the main frame buffer.
	IM_KEY_IN_RANGE	Show the main frame buffer only where the pixels of the overlay frame buffer are in a specified range. Specify the range with the IM_DISP_KEY_LOW and IM_DISP_KEY_HIGH fields.

IM_DISP_KEY_MODE (cont.)	IM_KEY_OUT_RANGE	Show the main frame buffer only where the pixels of the overlay frame buffer are out of a specified range. Specify the range with the IM_DISP_KEY_LOW and IM_DISP_KEY_HIGH fields.
IM_DISP_KEY_LOW	0 - 255	Lowest value of the keying range.
IM_DISP_KEY_HIGH	0 - 255	Highest value of the keying range.
IM_DISP_LUT_BUF	0	Associate a transparent LUT with the display.
	A buffer ID	Associate the specified LUT with the display. For a color display, the LUT buffer should be 3 bands of 8 bits per pixel or 1 band of 24 bits per pixel. For a monochrome display, the LUT buffer should be 1 band of 8 bits per pixel.
IM_DISP_MODE	IM_DISP_MONO	Display the main frame buffer in monochrome.
	IM_DISP_COLOR	Display the main frame buffer in true color. You must have the color version of the display section.
	IM_DISP_RED	Display in monochrome the red band of the main frame buffer. You must have the color version of the display section.
	IM_DISP_GREEN	Display in monochrome the green band of the main frame buffer. You must have the color version of the display section.
	IM_DISP_BLUE	Display in monochrome the blue band of the main frame buffer. You must have the color version of the display section.

IM_DISP_PAN_X	a multiple of 4	Displace the image horizontally so that the specified column is at the left of the screen.
IM_DISP_PAN_Y	any integer	Displace the image vertically so that the specified line is at the top of the screen.
IM_DISP_WRTMSK	0 - 0xFFFFFFFF	Protect bit planes in the main frame buffer from being overwritten by direct accesses (by the Host or 'C80), if the corresponding mask bit is 0. The required value must be specified using 24 bits: the low 8 bits of the value applies to the red buffer, the next 8 bits to the green buffer, and the high 8 bits to the blue buffer.
IM_DISP_ZOOM	1, 2, or 4	Zoom the image by the specified factor.
IM_DISP_OVERLAY_MODE	IM_ENABLE	Enable the overlay, and use the display LUT for the overlay. Note that enabling or disabling the overlay does not change the current display settings (mode, keying). You can still select monochrome or color modes for the main (underlay) frame buffer with IM_DISP_MODE. And if you enable the overlay at a later time, the previous keying mode will still be in effect.
	IM_DISABLE	Disable the overlay, and use the display LUT for the main frame buffer. This is intended to be used in a dual-screen display configuration only. Note that while the overlay is disabled, the keying mode has no effect, and you always see the LUT-mapped main (underlay) frame buffer.

The **Mode** parameter specifies the mode of operation. It can be set to:

- | | |
|----------|---|
| IM_NOW | Update the display immediately. (This might cause display artifacts). |
| IM_FRAME | Update the display at the end of the current frame. |

Note If you want several applications to share the display, you should avoid calling **imDispControl()** (except to set the keying when using the overlay).

imDispFree

Async

Gen-LC

Synopsis Free a display.

Format **void imDispFree(Thread, Disp)**

long Thread; Thread ID

long Disp; Display ID

Description This function deallocates a previously allocated display.

The **Thread** parameter specifies the thread to which to send **imDispFree()** for execution.

The **Disp** parameter specifies the display to deallocate.

imDispInquire

Sync

Gen-LC

Synopsis Inquire about a display attribute.

Format **long imDispInquire(Thread, Disp, Item, ValuePtr)**

long Thread;	Thread ID
long Disp;	Display ID (or 0)
long Item;	Attribute about which to inquire
void* ValuePtr;	Address of return value (or NULL)

Description This function inquires about an attribute of a specified display. You can change the value of an attribute using **imDispControl()**.

The **Thread** parameter specifies the thread to which to send **imDispInquire()** for execution.

The **Disp** parameter specifies the display. If your system has just one display, you can set this parameter to 0.

The **Item** parameter specifies the attribute about which to inquire. It can be set to:

IM_DISP_RESOLUTION_X	Display width.
IM_DISP_RESOLUTION_Y	Display height.
IM_DISP_KEY_MODE	Keying mode: IM_KEY_OFF, IM_KEY_ALWAYS, IM_KEY_IN_RANGE, or IM_KEY_OUT_RANGE (see imDispControl() for a description of these return values).
IM_DISP_KEY_LOW	Lowest value of the keying range.
IM_DISP_KEY_HIGH	Highest value of the keying range.
IM_DISP_MODE	Display mode: IM_DISP_MONO, IM_DISP_COLOR, IM_DISP_RED, IM_DISP_GREEN, IM_DISP_BLUE (see imDispControl() for a description of these return values), or IM_NONE (system does not have a display).

IM_DISP_VGA_MODE	Display configuration: IM_DISABLE (VGA disabled), IM_SINGLE_SCREEN (single-screen configuration), IM_DUAL_SCREEN (dual-screen configuration), IM_DUAL_HEAD (dual-head configuration; available only under Windows), or IM_NONE (no display; for example, a Genesis processor board with no main board).
IM_DISP_PAN_X	Horizontal displacement of the image (in columns).
IM_DISP_PAN_Y	Vertical displacement of the image (in lines).
IM_DISP_WRTMSK	Value of the write mask.
IM_DISP_ZOOM	Zoom factor.

The **ValuePtr** parameter specifies the address in which to return the value of the inquired attribute. Unless otherwise stated, **ValuePtr** should be the address of a long. Note that, since **imDisplnquire()** also returns this value, **ValuePtr** can be set to NULL.

Return value The returned value is the value of the inquired attribute, cast to long if necessary.

imFloatConvert

Async

PP

In-Place

Synopsis Convert between integer and floating-point buffers.

Format **void imFloatConvert(Thread, Src, Dst, Mode, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Mode; Operation mode
 long OSB; OSB ID (or 0)

Description This function converts a buffer's data between integer and floating-point.

For a floating-point to integer conversion, the data can be rounded towards zero or rounded towards the nearest integer. Overflows or underflows are set to the maximum or minimum value, respectively, of the destination buffer's data type.

The **Thread** parameter specifies the thread to which to send **imFloatConvert()** for execution.

The **Src** parameter specifies the buffer to convert, while the **Dst** parameter specifies the buffer in which to place the results of the conversion. For an integer to floating-point conversion, the source buffer can be of any integer type, while the destination buffer must be a 32-bit floating-point buffer. For a floating-point to integer conversion, the source buffer must be a 32-bit floating-point buffer and the destination buffer must be a 32-bit integer buffer.

The **Mode** parameter specifies the mode of operation. For a floating-point to integer conversion, **Mode** can be set to:

IM_TRUNCATE Round towards zero.
 IM_ROUND Round towards the nearest integer.

For an integer to floating-point conversion, **Mode** must be set to IM_DEFAULT.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imFloatDyadic

Async

PP

In-Place

Synopsis Perform an arithmetic operation between two floating-point buffers.

Format **void imFloatDyadic(Thread, Src1, Src2, Dst, Op, OSB)**

long Thread; Thread ID
 long Src1; First source buffer ID
 long Src2; Second source buffer ID
 long Dst; Destination buffer ID
 long Op; Operation to perform
 long OSB; OSB ID (or 0)

Description This function performs an arithmetic operation between two floating-point buffers.

The **Thread** parameter specifies the thread to which to send **imFloatDyadic()** for execution.

The **Src1** and **Src2** parameters specify the buffers with which to perform the operation. These must be 32-bit floating-point buffers.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This must be a 32-bit floating-point buffer.

The **Op** parameter specifies the type of operation to perform. It can be set to:

IM_ADD	Add.
IM_SUB	Subtract: Src1 - Src2 .
IM_SUB_ABS	Subtract and take the absolute value: Src1 - Src2 .
IM_MIN	Compare Src1 and Src2 on a pixel-by-pixel basis and take the minimum of the two.
IM_MAX	Compare Src1 and Src2 on a pixel-by-pixel basis and take the maximum of the two.
IM_MULT	Multiply.
IM_DIV	Divide: Src1 / Src2 .
IM_SQUARE_ADD	Square and add: Src1 ² + Src2 ² .
IM_ATAN2	Arctangent (Src1 / Src2).

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imFloatMonadic()**, **imFloatUnary()**. These functions perform arithmetic operations with only one floating-point buffer.

imFloatMac1

Async

PP

In-Place

Synopsis Multiply and accumulate with one floating-point buffer.

Format **void imFloatMac1(Thread, Src, Dst, Fac, Const, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 double Fac; Constant
 double Const; Constant
 long OSB; OSB ID (or 0)

Description This function scales a floating-point buffer by a specified factor, adds a specified constant to the result, and then stores final results in the destination buffer (**Dst = Fac*Src + Const**). If the source buffer is of single-precision, the constants are converted to single-precision values before the operation.

The **Thread** parameter specifies the thread to which to send **imFloatMac1()** for execution.

The **Src** parameter specifies the buffer with which to perform the operation. This must be a 32-bit floating-point buffer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This must be a 32-bit floating-point buffer.

The **Fac** parameter specifies the factor with which to multiply the source buffer.

The **Const** parameter specifies the constant to add to the scaled source buffer.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imFloatMac2()**. This function performs a multiply-and-accumulate operation with two floating-point buffers.

imFloatMac2

Async	PP	In-Place
-------	----	----------

Synopsis Multiply and accumulate with two floating-point buffers.

Format `void imFloatMac2(Thread, Src1, Src2, Dst, Fac1, Fac2, OSB)`

<code>long Thread;</code>	Thread buffer ID
<code>long Src1;</code>	First source buffer ID
<code>long Src2;</code>	Second source buffer ID
<code>long Dst;</code>	Destination buffer ID
<code>double Fac1;</code>	Constant
<code>double Fac2;</code>	Constant
<code>long OSB;</code>	OSB ID (or 0)

Description This function multiplies two floating-point buffers by specified factors, then adds the results. Final results are stored in the destination buffer (**Dst** = **Fac1*****Src1** + **Fac2*****Src2**). If a source buffer is of single-precision, its constant (**Fac1** or **Fac2**) is converted to a single-precision value before the operation.

The **Thread** parameter specifies the thread to which to send **imFloatMac2()** for execution.

The **Src1** and **Src2** parameters specify the buffers with which to perform the operation. These must be 32-bit floating-point buffers.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This must be a 32-bit floating-point buffer.

The **Fac1** and **Fac2** parameters specify the factors by which to multiply the first and second source buffers, respectively.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imFloatMac1()**. This function performs a multiply-and-accumulate operation with one floating-point buffer.

imFloatMonadic

Async

PP

In-Place

Synopsis	Perform an arithmetic operation between a floating-point buffer and a constant.
Format	<div><div>void imFloatMonadic(Thread, Src, Const, Dst, Op, OSB)</div><div><div>long Thread;</div><div>long Src;</div><div>double Const;</div><div>long Dst;</div><div>long Op;</div><div>long OSB;</div></div><div><div>Thread ID</div><div>Source buffer ID</div><div>Constant</div><div>Destination buffer ID</div><div>Operation to perform</div><div>OSB ID (or 0)</div></div></div>
Description	<p>This function performs an arithmetic operation between a floating-point buffer and a specified constant. If the buffer is of single precision, the constant is converted to a single precision value before the operation.</p> <p>The Thread parameter specifies the thread to which to send imFloatMonadic() for execution.</p> <p>The Src parameter specifies the buffer with which to perform the operation. This must be a 32-bit floating-point buffer.</p> <p>The Const parameter specifies the constant to use in the operation.</p> <p>The Dst parameter specifies the buffer in which to place the results of the operation. This must be a 32-bit floating-point buffer.</p> <p>The Op parameter specifies the type of operation to perform. It can be set to:</p> <div><div><div>IM_ADD</div><div>IM_SUB</div><div>IM_SUB_ABS</div><div>IM_SUB_NEG</div><div>IM_MIN</div><div>IM_MAX</div><div>IM_MULT</div><div>IM_DIV</div><div>IM_DIV_INT0</div></div><div><div>Add.</div><div>Subtract: Src - Const.</div><div>Subtract and take the absolute value: Src - Const .</div><div>Subtract and negate: Const - Src.</div><div>Compare Src and Const on a pixel-by-pixel basis and take the minimum of the two.</div><div>Compare Src and Const on a pixel-by-pixel basis and take the maximum of the two.</div><div>Multiply.</div><div>Divide by Const: Src/Const.</div><div>Divide into Const: Const/Src.</div></div></div>

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imFloatDyadic()**, **imFloatUnary()**. These functions also perform an arithmetic operation between floating-point buffers.

imFloatUnary

Async

PP

In-Place

Synopsis Perform a unary operation on a floating-point buffer.

Format **void imFloatUnary(Thread, Src, Dst, Op, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Op; Operation to perform
 long OSB; OSB ID (or 0)

Description This function performs a unary operation on a floating-point buffer.

The **Thread** parameter specifies the thread to which to send **imFloatUnary()** for execution.

The **Src** parameter specifies the buffer on which to perform the operation. This must be a 32-bit floating-point buffer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This must be a 32-bit floating-point buffer.

The **Op** parameter specifies the type of operation to perform. It can be set to:

IM_NEG Negate.
 IM_ABS Absolute value.
 IM_LOG Natural logarithm.
 IM_EXP Exponential.
 IM_SQUARE Square.
 IM_SQRT Square root.
 IM_SIN Sine.
 IM_COS Cosine.
 IM_TAN Tangent.
 IM_ATAN Arctangent.
 IM_CUBE Cube.
 IM_CBRT Cube root.

Note that, for trigonometric operations, pixel values are considered to be in radians.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imFloatDyadic()**, **imFloatMonadic()**. These functions also perform an arithmetic operation between floating-point buffers.

imGen1d

Async

Gen-LC

Synopsis Generate data into a one-dimensional buffer.

Format **void imGen1d(Thread, Buf, Func, Start, End, NumCoefs, Coefs, OSB)**

long Thread;	Thread ID
long Buf;	Destination buffer ID
long Func;	Function to generate
long Start;	Start value
long End;	End value
long NumCoefs;	Number of coefficients in the function
double* Coefs;	Address of array containing the coefficients
long OSB;	OSB ID (or 0)

Description This function generates data into a one-dimensional buffer, according to a specified function.

The **Thread** parameter specifies the thread to which to send **imGen1d()** for execution.

The **Buf** parameter specifies the one-dimensional buffer in which to generate data. This buffer can be of any integer type, or can be a 32-bit floating-point buffer.

The **Func** parameter specifies the function with which to generate data. It can be set to:

$$\text{IM_POLYNOMIAL} \quad C_0 + C_1x + C_2x^2 + \dots + C_{N-1}x^{N-1}$$

where N is the number of coefficients.

The **Start** and **End** parameters specify the first and last values at which to evaluate the function. In other words, the function is evaluated at $x = \text{Start}$, $x = \text{Start} + 1$, ..., $x = \text{End}$. The results are written into the first $(\text{End} - \text{Start} + 1)$ positions of the destination buffer. Note that all calculations are done in double precision and the final result is converted to the destination buffer's type.

The **NumCoefs** parameter specifies the number of coefficients to use in the function. This function supports a maximum of 10 coefficients.

The **Coefs** parameter specifies the address of the array containing the double precision floating-point coefficients.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Example The following code generates the numbers 255, 254, ..., 1, 0 into a buffer. The buffer could therefore be used to map an 8-bit image (using **imIntLutMap()**) such that pixel values in the image get inverted (that is, such that pixel value 0 is converted to 255, 1 to 254, etc.).

```
double Coef[2] = {255.0, -1.0};  
  
/* Generate an inverse ramp */  
imGen1d(Thread, LutBuf, IM_POLYNOMIAL, 0, 255, 2, Coef, 0);
```

imGenWarp1stOrder

Async

Synopsis Generate first-order warp coefficients.

Format **void imGenWarp1stOrder(Thread, Coef, Transform, Val1, Val2, Mode, OSB)**

long Thread;	Thread ID
long Coef;	Warp coefficient buffer ID
long Transform;	Type of transformation to perform
double Val1;	Constant
double Val2;	Constant
long Mode;	Operation mode
long OSB;	OSB ID (or 0)

Description This function generates first-order warp coefficients for use with **imIntWarpPolynomial()** or **imGenWarpLutMatrix()**.

Coefficients can be generated for a rotation, an image scaling, a shearing, or a translation. Note that, with **imGenWarp1stOrder()**, you specify how to perform the required forward transformation (source to destination) but the function generates coefficients for the inverse transformation (destination to source), as required by **imIntWarpPolynomial()** and **imGenWarpLutMatrix()**.

To combine transformations, you need to use separate calls to this function. For example, to generate coefficients for a rotation and translation, you would call this function twice, and write to the same buffer on both calls.

The **Thread** parameter specifies the thread to which to send **imGenWarp1stOrder()** for execution.

The **Coef** parameter specifies the buffer in which to write the coefficients. This must be a 32-bit floating-point buffer. If you are generating coefficients for **imIntWarpPolynomial()**, the **Coef** buffer must have a size of 3x2; if you are generating coefficients for **imGenWarpLutMatrix()**, the **Coef** buffer must have a size of 3x3.

The **Transform** parameter specifies the coefficients to generate. It can be set to:

IM_ROTATE	Generate coefficients for a counter-clockwise rotation around (0,0) by Val1 °.
IM_SCALE	Generate coefficients for an image scaling, by a factor of Val1 in the x direction and by a factor of Val2 in the y direction.
IM_SHEAR_X	Generate coefficients for a shearing in the x direction, by a factor of Val1 .
IM_SHEAR_Y	Generate coefficients for a shearing in the y direction, by a factor of Val1 .
IM_TRANSLATE	Generate coefficients for a translation by Val1 pixels in the x direction and by Val2 pixels in the y direction.

The **Val1** and **Val2** parameters specify transformation parameters. For cases where **Val2** is not used, any value can be given for it.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_CLEAR	Clear the coefficient buffer before generating coefficients.
IM_NO_CLEAR	Don't clear the coefficient buffer before generating coefficients.

Note that, if you are combining transformations, you should clear the coefficient buffer on the first call, and not clear it on subsequent calls.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Example The following code generates coefficients for a 30° rotation about the point (200, 100). Note that rotations are performed about (0, 0). Therefore, both rotation and translation coefficients are required. One way to generate the required coefficients is to first specify the rotation and then specify the translation, but the translation amount would be difficult to calculate. Instead, it is easier to translate the center of rotation to the origin, rotate, then translate the center of rotation back to its original position. The coefficient buffer is 3x2 since **imIntWarpPolynomial()** is used to perform the transformation.

```
imBufAlloc2d(Thr, 3, 2, IM_FLOAT, IM_PROC, &Coef);  
  
imGenWarp1stOrder(Thr, Coef, IM_TRANSLATE, -200.0, -100.0, IM_CLEAR, 0);  
imGenWarp1stOrder(Thr, Coef, IM_ROTATE, 30.0, 0.0, IM_NO_CLEAR, 0);  
imGenWarp1stOrder(Thr, Coef, IM_TRANSLATE, 200.0, 100.0, IM_NO_CLEAR, 0);  
  
imIntWarpPolynomial(Thr, Src, Dst, Coef, 0, 0);
```

For another example of using **imGenWarp1stOrder()**, see *process.c* in Appendix B.

imGenWarp4Corner

Async

Synopsis Generate warp coefficients to map an arbitrary quadrilateral onto a rectangle.

Format **void imGenWarp4Corner(**Thread, Coef, X1, Y1, X2, Y2, X3, Y3, X4, Y4, Xstart, Ystart, Xend, Yend, Mode, OSB**)**

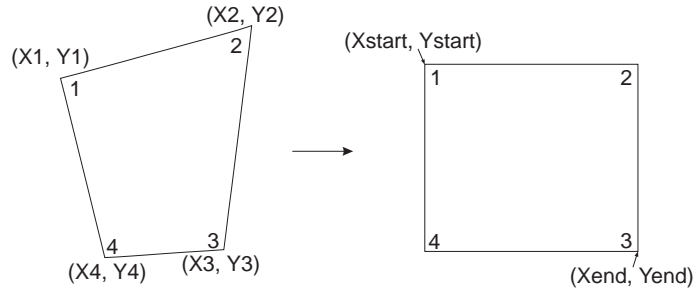
long Thread;	Thread ID
long Coef;	Warp coefficient buffer ID
double X1;	X coordinate of quadrilateral's 1st corner
double Y1;	Y coordinate of quadrilateral's 1st corner
double X2;	X coordinate of quadrilateral's 2nd corner
double Y2;	Y coordinate of quadrilateral's 2nd corner
double X3;	X coordinate of quadrilateral's 3rd corner
double Y3;	Y coordinate of quadrilateral's 3rd corner
double X4;	X coordinate of quadrilateral's 4th corner
double Y4;	Y coordinate of quadrilateral's 4th corner
long Xstart;	X coordinate of rectangle's top-left corner
long Ystart;	Y coordinate of rectangle's top-left corner
long Xend;	X coordinate of rectangle's bottom-right corner
long Yend;	Y coordinate of rectangle's bottom-right corner
long Mode;	Operation mode
long OSB;	OSB ID (or 0)

Description This function generates warp coefficients for perspective transformations that map an arbitrary quadrilateral onto a rectangle. To perform the warping, pass the generated coefficients to **imGenWarpLutMatrix()**, which will generate the address look-up tables required by **imIntWarpLut()**.

The **Thread** parameter specifies the thread to which to send **imGenWarp4Corner()** for execution.

The **Coef** parameter specifies the buffer in which to write the coefficients. This must be a 32-bit floating-point buffer of size 3x3.

The **(X1, Y1)** through **(X4, Y4)** parameters specify the (generally non-integer) corner points of the source quadrilateral; the **(Xstart, Ystart)** and **(Xend, Yend)** parameters specify the integer corner points of the rectangle, as follows:



The **Mode** parameter specifies the mode of operation. This parameter must be set to `IM_DEFAULT`.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Example See *process.c* in Appendix B.

imGenWarpLutMatrix

Async

PP

Synopsis Generate address LUTs for a matrix-defined warping using **imIntWarpLut()**.

Format **void imGenWarpLutMatrix(Thread, Xlut, Ylut, Coef, Control, OSB)**

long Thread;	Thread ID
long Xlut;	X-LUT buffer ID
long Ylut;	Y-LUT buffer ID
long Coef;	Warp coefficient buffer ID
long Control;	Control buffer ID (or 0)
long OSB;	OSB ID (or 0)

Description This function generates the address look-up tables (LUTs) needed to perform a 3x3 matrix-defined warping using **imIntWarpLut()**. Note that a warping is performed by first associating each pixel position of the destination buffer, (x_d, y_d) , with a specific point (address) in the source buffer, (x_s, y_s) . The pixel value of (x_d, y_d) is then determined from its associated point and from a specified interpolation mode. A 3x3 matrix-defined warping associates (x_d, y_d) with (x_s, y_s) through the following:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} c_{00} & c_{10} & c_{20} \\ c_{01} & c_{11} & c_{21} \\ c_{02} & c_{12} & c_{22} \end{bmatrix} \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix}$$

where

$$x_s = \frac{x}{w} = \frac{c_{00}x_d + c_{10}y_d + c_{20}}{c_{02}x_d + c_{12}y_d + c_{22}}$$

$$y_s = \frac{y}{w} = \frac{c_{01}x_d + c_{11}y_d + c_{21}}{c_{02}x_d + c_{12}y_d + c_{22}}$$

To perform 3x3 matrix-defined warpings, you must supply the 3x3 coefficients $(c_{00} \dots c_{22})$ to **imGenWarpLutMatrix()**. The **imGenWarpLutMatrix()** function generates the LUTs required by **imIntWarpLut()** to perform the warping. Note that **imIntWarpLut()** determines x_s from one LUT and y_s from another LUT.

The 3x3 coefficients can be user-supplied or automatically generated using either **imGenWarp4Corner()** (for perspective transformations that map an arbitrary quadrilateral onto a rectangle) or **imGenWarp1stOrder()** (for first-order polynomial transformations).

The **Thread** parameter specifies the thread to which to send **imGenWarpLutMatrix()** for execution.

The **Xlut** parameter specifies the buffer in which to place the x_s points. This must be a signed 16-bit buffer, and have the same size as the destination buffer that you will eventually pass to **imIntWarpLut()**.

The **Ylut** parameter specifies the buffer in which to place the y_s points. This must be a signed 16-bit buffer, and have the same size as the destination buffer that you will eventually pass to **imIntWarpLut()**.

The **Coef** parameter specifies the buffer containing the coefficients required to produce the desired warping. This must be a 32-bit floating-point buffer of size 3x3.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imGenWarpLutMatrix()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_PRECISION	0 - n	Number of fractional bits for the address. If you are not going to perform interpolation with imIntWarpLut() (i.e. you are going to use nearest neighbor interpolation), give 0. If you require interpolation, give a value > 0.

IM_CTL_OVERSCAN	IM_TRANSPARENT	If addresses fall outside of the source buffer of imIntWarpLut() , use them anyway. Note that, if the source buffer is not a child buffer, these addresses will point to undefined pixel values, leading to unpredictable results.
	IM_REPLACE	Replace addresses that fall outside of the source buffer of imIntWarpLut() with a constant address. Specify the values of the address with the IM_CTL_OVERSCAN_X and IM_CTL_OVERSCAN_Y fields. If you use replace overscan, the size of the source buffer is required; specify this with the IM_CTL_SRC_SIZE_X and IM_CTL_SRC_SIZE_Y fields.
IM_CTL_OVERSCAN_X	any integer (default: 0)	Replace value for X address.
IM_CTL_OVERSCAN_Y	any integer (default: 0)	Replace value for Y address.
IM_CTL_SRC_SIZE_X	any integer (default: value of Xlut)	X size of source buffer to be given to imIntWarpLut() .
IM_CTL_SRC_SIZE_Y	any integer (default: value of Ylut)	Y size of source buffer to be given to imIntWarpLut() .
IM_CTL_ZOOM	1, 2, or 4	Multiply destination addresses by the specified factor before using them to calculate source addresses. This field allows you to generate LUTs that are smaller than the image you want to warp, which will considerably speed up LUT generation. You should zoom up the address LUTs (with interpolation), before using them in imIntWarpLut() .

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Example See *process.c* in Appendix B.

imGraArc

Async

Multi-band

Synopsis Draw an elliptical arc.

Format **void imGraArc(Thread, Context, Buf, Xcen, Ycen, Xrad, Yrad, StartAng, EndAng)**

long Thread;

long Context;

long Buf;

long Xcen;

long Ycen;

long Xrad;

long Yrad;

double StartAng;

double EndAng;

Thread ID

Control buffer ID (or 0)

Destination buffer ID

X coordinate of arc centre

Y coordinate of arc centre

X radius of arc

Y radius of arc

Starting angle

Ending angle

Description This function draws an elliptical arc.

The **Thread** parameter specifies the thread to which to send **imGraArc()** for execution.

The **Context** parameter specifies the control buffer with which to perform the function. Relevant fields for **imGraArc()** are listed below, with default values in bold-face. Note that if the **Context** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_GRA_COLOR	0 - 0xFFFFFFFF	Color with which to draw the arc.
IM_GRA_COLOR_MODE	IM_DEFAULT	Use the specified color in all bands of a multi-band buffer.
	IM_PACKED	Use the least-significant byte of the specified color in the first band of a multi-band buffer, the next byte in the second band of the buffer, etc. The buffer must be 8-bit, with no more than 4 bands.

IM_GRA_DRAW_MODE	IM_PASS	Draw the arc in the specified color.
	IM_XOR	Draw the arc in the colors that result from performing an XOR between the specified color and the pixels of the destination buffer.

The **Buf** parameter specifies the buffer in which to draw. This must be an 8-bit, 16-bit, or 32-bit integer buffer.

The **Xcen** and **Ycen** parameters specify the x and y coordinates of the arc center, relative to the top-left corner of the destination buffer.

The **Xrad** and **Yrad** parameters specify the elliptic arc radii.

The **StartAng** and **EndAng** parameters specify the angles at which to start and to end drawing of the arc, moving in a counter-clockwise direction. Express these angles as degrees, relative to the positive x-axis. To draw a complete ellipse, set **StartAng** to 0 and **EndAng** to 360.

See also **imGraArcFill()**. This function draws a filled elliptical arc.

imGraArcFill

Async

Multi-band

Synopsis Draw a filled elliptical arc.

Format **void imGraArcFill(Thread, Context, Buf, Xcen, Ycen, Xrad, Yrad, StartAng, EndAng)**

long Thread;	Thread ID
long Context;	Control buffer ID (or 0)
long Buf;	Destination buffer ID
long Xcen;	X coordinate of arc centre
long Ycen;	Y coordinate of arc centre
long Xrad;	X radius of arc
long Yrad;	Y radius of arc
double StartAng;	Starting angle
double EndAng;	Ending angle

Description This function draws a filled elliptical arc.

The **Thread** parameter specifies the thread to which to send **imGraArcFill()** for execution.

The **Context** parameter specifies the control buffer with which to perform the function. Relevant fields for **imGraArcFill()** are listed below, with default values in bold-face. Note that if the **Context** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_GRA_COLOR	0 - 0xFFFFFFFF	Color with which to draw the arc.
IM_GRA_COLOR_MODE	IM_DEFAULT	Use the specified color in all bands of a multi-band buffer.
	IM_PACKED	Use the least-significant byte of the specified color in the first band of a multi-band buffer, the next byte in the second band of the buffer, etc. The buffer must be 8-bit, with no more than 4 bands.

The **Buf** parameter specifies the buffer in which to draw. This must be an 8-bit, 16-bit, or 32-bit integer buffer.

The **Xcen** and **Ycen** parameters specify the x and y coordinates of the arc center, relative to the top-left corner of the destination buffer.

The **Xrad** and **Yrad** parameters specify the elliptic arc radii.

The **StartAng** and **EndAng** parameters specify the angles at which to start and to end drawing of the arc, moving in a counter-clockwise direction.

Express these angles as degrees, relative to the positive x-axis. To draw a complete ellipse, set **StartAng** to 0 and **EndAng** to 360.

See also **imGraArc()**. This function draws an unfilled elliptical arc.

imGraFill

Async

Multi-band

Synopsis Fill a connected region.

Format **void imGraFill(Thread, Context, Buf, Xstart, Ystart)**

long Thread; Thread ID
long Context; Control buffer ID (or 0)
long Buf; Destination buffer ID
long Xstart; X coordinate of seed pixel
long Ystart; Y coordinate of seed pixel

Description This function fills the connected region around a specified pixel (called the seed pixel). Note that a connected region is an area of touching pixels that have the same value as the seed pixel (horizontally and vertically adjacent pixels are considered touching; diagonally adjacent pixels are not).

This function also records the bounding box of the filled region in the specified control buffer. After calling **imGraFill()**, you can read the maximum and minimum coordinate of filled pixels from the appropriate fields in the control buffer using **imBufGetField()**.

The **Thread** parameter specifies the thread to which to send **imGraFill()** for execution.

The **Context** parameter specifies the control buffer with which to perform the function. Relevant fields for **imGraFill()** are listed below, with default values in bold-face. Note that if the **Context** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_GRA_COLOR	0 - 0xFFFFFFFF	Color with which to fill.
IM_GRA_COLOR_MODE	IM_DEFAULT	Use the specified color in all bands of a multi-band buffer.
	IM_PACKED	Use the least-significant byte of the specified color in the first band of a multi-band buffer, the next byte in the second band of the buffer, etc. The buffer must be 8-bit, with no more than 4 bands.

imGraFill() records the bounding box of the region it fills in the **Context** buffer. After calling this function, you can read the following fields from the Context buffer using **imBufGetField()**:

Field	Meaning
IM_GRA_BOX_X_MIN	X-coordinate of left-most pixel filled.
IM_GRA_BOX_X_MAX	X-coordinate of right-most pixel filled.
IM_GRA_BOX_Y_MIN	Y-coordinate of top-most pixel filled.
IM_GRA_BOX_Y_MAX	Y-coordinate of bottom-most pixel filled.

The **Buf** parameter specifies the buffer in which to perform the fill. This must be an 8-bit, 16-bit, or 32-bit integer buffer.

The **Xstart** and **Ystart** parameters specify the X and Y coordinates of the seed pixel, relative to the top-left corner of the destination buffer.

See also **imGraArcFill()**, **imGraRectFill()**. These functions draw filled elliptical arcs and filled rectangles, respectively.

The **Buf** parameter specifies the buffer in which to draw. This must be an 8-bit, 16-bit, or 32-bit integer buffer.

The **Xstart** and **Ystart** parameters specify the x and y coordinates of the line's start point, relative to the top-left corner of the destination buffer.

The **Xend** and **Yend** parameters specify the x and y coordinates of the line's end point, relative to the top-left corner of the destination buffer.

See also **imGraPlot()**. This function can plot a series of lines faster than separate calls to **imGraLine()**.

imGraPlot

Async

Multi-band

Synopsis Plot a series of (x,y) points.

Format **void imGraPlot(Thread, Context, Buf, Xbuf, Ybuf, NumPoints)**

long Thread;	Thread ID
long Context;	Control buffer ID (or 0)
long Buf;	Destination buffer ID
long Xbuf;	Buffer ID of X coordinates
long Ybuf;	Buffer ID of Y coordinates
long NumPoints;	Number of points to plot

Description This function plots a series of (x,y) points. Prior to plotting, the x and y coordinates are multiplied by a specified scale factor; then an offset is added. This allows you to control the slope and location of your plot.

The **Thread** parameter specifies the thread to which to send **imGraPlot()** for execution.

The **Context** parameter specifies the control buffer with which to perform the function. Relevant fields for **imGraPlot()** are listed below, with default values in bold-face. Note that if the **Context** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_GRA_COLOR	0 - 0xFFFFFFFF	Color with which to plot.
IM_GRA_COLOR_MODE	IM_DEFAULT	Use the specified color in all bands of a multi-band buffer.
	IM_PACKED	Use the least-significant byte of the specified color in the first band of a multi-band buffer, the next byte in the second band of the buffer, etc. The buffer must be 8-bit, with no more than 4 bands.
IM_GRA_PLOT_MODE	IM_PLOT_POLY	Connect the points with a single line.
	IM_PLOT_LINES	Connect each pair of points separately.
	IM_PLOT_DOTS	Draw a dot at each point.

IM_GRA_SCALE_X	any floating-point value (default: 1.0)	Scale all X coordinates by the specified factor.
IM_GRA_SCALE_Y	any floating-point value (default: 1.0)	Scale all Y coordinates by the specified factor.
IM_GRA_OFFSET_X	any integer (default: 0)	Add the specified offset to all scaled X coordinates.
IM_GRA_OFFSET_Y	any integer (default: 0)	Add the specified offset to all scaled Y coordinates.
IM_GRA_DRAW_MODE	IM_PASS IM_XOR	Draw the plot in the specified color. Draw the plot in the colors that result from performing an XOR between the specified color and the pixels of the destination buffer.

The **Buf** parameter specifies the buffer in which to plot. This must be an 8-bit, 16-bit, or 32-bit integer buffer.

The **Xbuf** parameter specifies the buffer containing the X coordinates.

The **Ybuf** parameter specifies the buffer containing the Y coordinates.

The **NumPoints** parameter specifies the number of points to plot. This parameter can be set to **IM_ALL**, in which case all points are plotted.

Example See *process.c* in Appendix B.

imGraRect

Async

Multi-band

Synopsis Draw a rectangle.

Format **void imGraRect(Thread, Context, Buf, Xstart, Ystart, Xend, Yend)**

- long Thread; Thread ID
- long Context; Control buffer ID (or 0)
- long Buf; Destination buffer ID
- long Xstart; X coordinate of top-left corner
- long Ystart; Y coordinate of top-left corner
- long Xend; X coordinate of bottom-right corner
- long Yend; Y coordinate of bottom-right corner

Description This function draws a rectangle.

The **Thread** parameter specifies the thread to which to send **imGraRect()** for execution.

The **Context** parameter specifies the control buffer with which to perform the function. Relevant fields for **imGraRect()** are listed below, with default values in bold-face. Note that if the **Context** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_GRA_COLOR	0 - 0xFFFFFFFF	Color with which to draw.
IM_GRA_COLOR_MODE	IM_DEFAULT	Use the specified color in all bands of a multi-band buffer.
	IM_PACKED	Use the least-significant byte of the specified color in the first band of a multi-band buffer, the next byte in the second band of the buffer, etc. The buffer must be 8-bit, with no more than 4 bands.
IM_GRA_DRAW_MODE	IM_PASS	Draw the rectangle in the specified color.
	IM_XOR	Draw the rectangle in the colors that result from performing an XOR between the specified color and the pixels of the destination buffer.

The **Buf** parameter specifies the buffer in which to draw. This must be an 8-bit, 16-bit, or 32-bit integer buffer.

The **Xstart** and **Ystart** parameters specify the x and y coordinates of the rectangle's top-left corner. These coordinates are relative to the top-left corner of the destination buffer.

The **Xend** and **Yend** parameters specify the x and y coordinates of the rectangle's bottom-right corner. These coordinates are relative to the top-left corner of the destination buffer.

See also **imGraRectFill()**. This function draws a filled rectangle.

imGraRectFill

Async

Multi-band

Synopsis Draw a filled rectangle.

Format **void imGraRectFill(Thread, Context, Buf, Xstart, Ystart, Xend, Yend)**

long Thread;	Thread ID
long Context;	Control buffer ID (or 0)
long Buf;	Destination buffer ID
long Xstart;	X coordinate of top-left corner
long Ystart;	Y coordinate of top-left corner
long Xend;	X coordinate of bottom-right corner
long Yend;	Y coordinate of bottom-right corner

Description This function draws a filled rectangle.

The **Thread** parameter specifies the thread to which to send **imGraRectFill()** for execution.

The **Context** parameter specifies the control buffer with which to perform the function. Relevant fields for **imGraRectFill()** are listed below, with default values in bold-face. Note that if the **Context** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_GRA_COLOR	0 - 0xFFFFFFFF	Color with which to draw.
IM_GRA_COLOR_MODE	IM_DEFAULT	Use the specified color in all bands of a multi-band buffer.
	IM_PACKED	Use the least-significant byte of the specified color in the first band of a multi-band buffer, the next byte in the second band of the buffer, etc. The buffer must be 8-bit, with no more than 4 bands.

The **Buf** parameter specifies the buffer in which to draw. This must be an 8-bit, 16-bit, or 32-bit integer buffer.

The **Xstart** and **Ystart** parameters specify the x and y coordinates of the rectangle's top-left corner. These coordinates are relative to the top-left corner of the destination buffer.

The **Xend** and **Yend** parameters specify the x and y coordinates of the rectangle's bottom-right corner. These coordinates are relative to the top-left corner of the destination buffer.

See also **imGraRect()**. This function draws an unfilled rectangle.

imGraText

Async

Multi-band

Synopsis Write text.

Format **void imGraText(Thread, Context, Buf, Xstart, Ystart, String)**

long Thread; Thread ID
 long Context; Control buffer ID (or 0)
 long Buf; Destination buffer ID
 long Xstart; X coordinate of start of string
 long Ystart; Y coordinate of start of string
 char* String; Null terminated ASCII string

Description This function writes an ASCII string into the specified buffer. The size of the text is determined from the specified font type (default, small, medium, large) and by the specified x and y scale factors.

The **Thread** parameter specifies the thread to which to send **imGraText()** for execution.

The **Context** parameter specifies the control buffer with which to perform the function. Relevant fields for **imGraText()** are listed below, with default values in bold-face. Note that if the **Context** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_GRA_COLOR	0 - 0xFFFFFFFF	Foreground color.
IM_GRA_BACK_COLOR	0 - 0xFFFFFFFF	Background color.
IM_GRA_BACK_MODE	IM_OPAQUE	Draw the background in the color specified by the IM_GRA_BACK_COLOR field.
	IM_TRANSPARENT	Don't draw the background.

IM_GRA_COLOR_MODE	IM_DEFAULT	Use the specified foreground and background colors in all bands of a multi-band buffer.
	IM_PACKED	Use the least-significant byte of the specified foreground and background colors in the first band of a multi-band buffer, the next byte of the specified colors in the second band of the buffer, etc. The buffer must be 8-bit, with no more than 4 bands.
IM_GRA_FONT	IM_FONT_DEFAULT	Use the default font.
	IM_FONT_SMALL	Use a small version of the default font.
	IM_FONT_MEDIUM	Use a medium version of the default font.
	IM_FONT_LARGE	Use a large version of the default font.
IM_GRA_FONT_SCALE_X	any integer (default: 1)	Scale the size of characters in the x direction by the specified factor.
IM_GRA_FONT_SCALE_Y	any integer (default: 1)	Scale the size of characters in the y direction by the specified factor.

The **Buf** parameter specifies the buffer in which to write. This must be an 8-bit, 16-bit, or 32-bit integer buffer.

The **Xstart** and **Ystart** parameters specify the x and y coordinates at which to start writing the top-left corner of the first character. These coordinates are relative to the top-left corner of the destination buffer.

The **String** parameter specifies the address of the string.

Example See *process.c* in Appendix B.

imIntBinarize

Async

PP

In-Place

Multi-band

Synopsis Binarize an image.

Format `void imIntBinarize(Thread, Src, Dst, Cond, Low, High, Val1, Val2, OSB)`

long Thread;	Thread ID
long Src;	Source buffer ID
long Dst;	Destination buffer ID
long Cond;	Conditional operator
long Low;	Low threshold
long High;	High threshold
long Val1;	Constant
long Val2;	Constant
long OSB;	OSB ID (or 0)

Description This function binarizes an integer image. It converts pixel values to **Val1** if a specified condition is true; to **Val2** otherwise.

The **Thread** parameter specifies the thread to which to send **imIntBinarize()** for execution.

The **Src** parameter specifies the buffer to binarize. This buffer can be of any integer type.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This must be an 8-bit integer buffer.

The **Cond** parameter specifies the condition with which to binarize the image. It can be set to:

IM_EQUAL	if equal to Low .
IM_NOT_EQUAL	if not equal to Low .
IM_LESS	if less than Low .
IM_LESS_OR_EQUAL	if less than or equal to Low .
IM_GREATER	if greater than Low .
IM_GREATER_OR_EQUAL	if greater than or equal to Low .
IM_IN_RANGE	if within Low to High , inclusive.
IM_OUT_RANGE	if less than Low or greater than High .

The **Low** and **High** parameters specify integer constants. Note that, when the source buffer is a 32-bit integer buffer, **Low** and **High** are interpreted as signed if the source buffer is signed; they are interpreted as unsigned if the source buffer is unsigned.

For cases where **High** is not used, any value can be given for it.

The **Val1** and **Val2** parameters specify integer constants.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBinConvert()**. Use **imBinConvert()** instead of **imIntBinarize()** if you want to store the binary data in a binary buffer (**imIntBinarize()** stores the binary data in an integer buffer).

imIntClip

Async

PP

In-Place

Multi-band

Synopsis Clip or binarize an image.**Format** **void imIntClip(Thread, Src, Dst, Cond, Low, High, Val1, Val2, Mode, OSB)**

long Thread; Target thread
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Cond; Clipping condition
 long Low; First threshold
 long High; Second threshold (optional)
 long Val1; Constant
 long Val2; Constant
 long Mode; Mode of operation
 long OSB; OSB ID (or 0)

Description This function clips or binarizes an image, using constant or non-constant thresholds. This function is similar to **imIntBinarize()** and **imBinConvert()**, except that with **imIntClip()** the threshold levels can be images instead of constants. Using images allows the threshold level to be different for each pixel. This feature is useful, for example, when the lighting in the original image is not uniform.

The **Thread** parameter specifies the thread to which to send **imIntClip()** for execution.

The **Src** parameter specifies the buffer to clip or binarize. This buffer can be of any integer type (8, 16, or 32 bits, signed or unsigned).

The **Dst** parameter specifies the buffer in which to place the results of the operation. If the **Mode** parameter is set to IM_PASS_PIXEL, the destination buffer must be of the same type as the source buffer (**Src**). If the **Mode** parameter is set to IM_PASS_CONSTANT, the destination buffer depth must be 1 bit or 8 bits.

The **Cond** parameter specifies the condition with which to clip or binarize the image. It can be set to:

IM_EQUAL	If equal to Low .
IM_NOT_EQUAL	If not equal to Low .
IM_LESS	If less than Low .
IM_LESS_OR_EQUAL	If less than or equal to Low .
IM_GREATER	If greater than Low .
IM_GREATER_OR_EQUAL	If greater than or equal to Low .
IM_IN_RANGE	If within Low to High , inclusive.
IM_OUT_RANGE	If less than Low or greater than High .

For all **Cond** values, with the exception of IM_IN_RANGE and IM_OUT_RANGE, the result is as follows:

```
if (Src cond Low)      // Low can be an image or a constant
    Dst = Val1
else
    Dst = (Mode & IM_PASS_CONSTANT) ? Val2 : Src
```

When **Cond** is IM_IN_RANGE, the result is as follows:

```
if (Src >= Low && Src <= High) // Low and High can be images or constants
    Dst = Val1
else
    Dst = (Mode & IM_PASS_CONSTANT) ? Val2 : Src
```

When **Cond** is IM_OUT_RANGE, the result is as follows:

```
if (Src < Low)          // Low can be an image or a constant
    Dst = Val1
else if (Src > High)    // High can be an image or a constant
    Dst = Val2
else                    // i.e., it is assumed you specified IM_PASS_PIXEL
    Dst = Src
```

The **Low** and **High** parameters specify integer constants or image buffer identifiers. In the latter case, the buffers must be of the same type as the source buffer (**Src**).

For a 32-bit signed source buffer, if the **Low** and **High** thresholds are constants, they are interpreted as signed constants. Similarly, for a 32-bit unsigned source buffer, if the **Low** and **High** thresholds are constants, they are interpreted as unsigned constants.

The **Val1** and **Val2** parameters specify integer values to use as replacement values.

The **Mode** parameter specifies the mode of operation to be used for this clipping operation. The **Mode** can be one of the following:

IM_THRESH_CONSTANT	The Low and High parameters are both constants.
IM_THRESH_PIXEL	The Low and High parameters are both images.

The modes listed above can be combined with one of the following:

IM_PASS_CONSTANT	Pass Val2 when condition is FALSE.
IM_PASS_PIXEL	Pass Src pixel when condition is FALSE.

These mode values can be combined in all four possible combinations.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imIntBinarize()** and **imBinConvert()**. Use **imIntBinarize()** or **imBinConvert()** to binarize or clip an image using constant or non-constant thresholds when you do not want to use images as the threshold levels.

imIntConnectMap

Async

PP

Multi-band

Synopsis Perform a 3x3 connectivity mapping.

Format **void imIntConnectMap(Thread, Src, Dst, Lut, Control, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Lut; LUT buffer ID
 long Control; Control buffer ID (or 0)
 long OSB; OSB ID (or 0)

Description This function performs a 3x3 connectivity mapping on an integer image. It calculates a connectivity code for each pixel in the image and then maps these codes through the specified LUT buffer.

Connectivity codes are determined in the following order:

$$\begin{bmatrix} n_3 & n_2 & n_1 \\ n_4 & n_8 & n_0 \\ n_5 & n_6 & n_7 \end{bmatrix}$$

where n_i is either 0 or 1 (non-zero pixels are treated as 1).

$$\text{Connectivity code} = \sum_{i=0}^8 2^i n_i$$

Result = LUTMAP (Connectivity code).

The **Thread** parameter specifies the thread to which to send **imIntConnectMap()** for execution.

The **Src** parameter specifies the buffer on which to perform the connectivity mapping. This must be an 8-bit integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the mapping. This can be an 8-bit or 16-bit integer buffer. Note that in-place operation is not supported for this function.

The **Lut** parameter specifies the LUT buffer. Since each connectivity code has 9 bits, the LUT buffer should have at least 2^9 (or 512) entries. The LUT buffer must be of the same type as the destination buffer.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntConnectMap()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_OVERSCAN	IM_TRANSPARENT	Process the bordering pixels of the source buffer using the pixels of its parent buffer as the overscan pixels. If the source buffer is not a child buffer or if its parent buffer can't provide values for the overscan pixels, the overscan pixels will be undefined, leading to unpredictable results.
	IM_REPLACE	Process the bordering pixels of the source buffer by assigning a specific value to the overscan pixels. Specify the value with the IM_CTL_OVERSCAN_VAL field.
IM_CTL_OVERSCAN_VAL	0 or 1	Overscan replace value (used when IM_CTL_OVERSCAN is set to IM_REPLACE).

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imIntConvert

Async	PP	In-Place	Multi-band
-------	----	----------	------------

Synopsis Convert a buffer from one integer type to another.

Format `void imIntConvert(Thread, Src, Dst, Mode, OSB)`

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Mode; Operation mode
 long OSB; OSB ID (or 0)

Description This function converts a buffer's data from one integer type to another. When converting to a lower type, the most significant bits are discarded. When converting to a higher type, the most significant bit is extended if the source buffer is signed; otherwise, 0 is extended.

Note that you can clip the source buffer's pixel values to the dynamic range of the destination buffer (after sign-extending if necessary). Alternatively, you can take the absolute value and clip to the dynamic range of the destination buffer (after sign-extending if necessary).

The **Thread** parameter specifies the thread to which to send **imIntConvert()** for execution.

The **Src** parameter specifies the buffer to convert. This buffer can be of any integer type.

The **Dst** parameter specifies the buffer in which to place the results of the conversion. This buffer must be of the type to which you want to convert (note that you can convert to any integer type).

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_DEFAULT	Sign-extend when converting to a higher type; discard high bits when converting to a lower type.
IM_CLIP	Clip to the dynamic range of the destination buffer (after sign-extending if necessary).
IM_ABS_CLIP	Take the absolute value and clip to the dynamic range of the destination buffer (after sign-extending if necessary).

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imIntConvertColor

Async

PP

In-Place

Synopsis Perform a color conversion.

Format **void imIntConvertColor(Thread, Src, Dst, Type, Coef, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Type; Conversion type
 long Coef; Coefficient buffer ID (or 0)
 long OSB; OSB ID (or 0)

Description This function converts a color image to a different color space, or converts an image between color and grayscale. You can perform a predefined conversion or a custom matrix-defined conversion.

The **Thread** parameter specifies the thread to which to send **imIntConvertColor()** for execution.

The **Src** parameter specifies the buffer to convert. This must be a 1-band or 3-band, 8-bit integer buffer.

❖ RGB and HSL values are always assumed to be unsigned, regardless of the data type of the buffer in which they are stored.

The **Dst** parameter specifies the buffer in which to place the results of the conversion. This must be a 1-band or 3-band, 8-bit integer buffer. Note that results might be scaled to cover the range of the destination buffer. For example, 0° to 360° H (Hue) values of the HSL color space will generate values from 0 to 255 in the 8-bit destination buffer.

The **Type** parameter specifies the type of conversion to perform. It can be set to:

Field	Meaning
IM_RGB_TO_HSL	Convert from the RGB color space to the HSL color space. The source and destination buffers must have 3 bands.
IM_HSL_TO_RGB	Convert from the HSL color space to the RGB color space. The source and destination buffers must have 3 bands.

Field	Meaning
IM_RGB_TO_L	Convert from the RGB color space to grayscale, where the grayscale values represent the luminance of each pixel. The source buffer must be 3-band. The destination buffer can be 1-band or 3-band. If it is 3-band, it is assumed to be an HSL image and only the last band (that is, band 2) is written.
IM_RGB_TO_I	Convert from the RGB color space to grayscale, where the grayscale values represent the intensity of each pixel, that is, $I = (R + G + B)/3$. The source buffer must be 3-band. The destination buffer can be 1-band or 3-band. If it is 3-band, it is assumed to be an HSI image and only the last band (that is, band 2) is written.
IM_L_TO_RGB	Convert from grayscale to the RGB color space (the grayscale values are repeated in each color band, producing a monochromatic RGB image). The source buffer can be 1-band or 3-band. If it is 3-band, it is assumed to be an HSL image and only the last band (that is, band 2) is used. The destination buffer must be 3-band.
IM_RGB_TO_H	Calculates only the H component of HSL (faster than calculating all three components). The source buffer must be 3-band; the destination buffer can be 1-band or 3-band (in the latter case, only band 0 is written).
IM_MATRIX	Perform an arbitrary matrix-defined color conversion. You must also pass a 3x3 or 3x1 floating point coefficient buffer.

The **Coef** parameter specifies the coefficient buffer used to perform certain conversions.

For a matrix-defined conversion, the coefficient buffer (**Coef**) should be 3x3 if both source and destination buffers have 3 bands. If the source buffer has 3 bands and the destination buffer has 1 band, then the coefficient buffer should be 3x1.

For 3x3 **Coef** buffers:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

each band of the destination is calculated from the source bands as follows:

$$\text{Dst}[0] = a.\text{Src}[0] + b.\text{Src}[1] + c.\text{Src}[2]$$

$$\text{Dst}[1] = d.\text{Src}[0] + e.\text{Src}[1] + f.\text{Src}[2]$$

$$\text{Dst}[2] = g.\text{Src}[0] + h.\text{Src}[1] + i.\text{Src}[2]$$

By default, underflows and overflows are not clipped; the input format is the same as that of the source buffer, and the output format is the same as that of the destination buffer. The following control fields can be added to the **Coef** buffer in matrix-defined mode, with default values in bold-face.

Field	Value	Meaning
IM_CTL_CLIP	IM_DISABLE	Do not clip underflows and overflows.
	IM_ENABLE	Clip underflows and overflows. When clipping is enabled, underflows and overflows are clipped to the output range $[0, 255]$ for unsigned outputs, and $[-128, 127]$ for signed outputs).
IM_CTL_INPUT_FORMAT (The default is the source buffer format)	IM_UNSIGNED	All input bands are unsigned (for example, RGB).
	IM_SIGNED	Last two input bands are signed (for example, YUV). Note that the first input band is always unsigned.
IM_CTL_OUTPUT_FORMAT (The default is the destination buffer format)	IM_UNSIGNED	All output bands are unsigned.
	IM_SIGNED	Last two output bands are signed. Note that the first output band is always unsigned.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imIntConvolve

Async

PP

NOA

Multi-band

Synopsis Perform a convolution.

Format **void imIntConvolve(Thread, Src, Dst, Kernel, Control, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Kernel; Kernel buffer ID
 long Control; Control buffer ID (or 0)
 long OSB; OSB ID (or 0)

Description This function performs a convolution operation on an integer image, using a specified kernel. You can use your own kernel or a predefined kernel. In general, the predefined kernels will execute faster.

If you use your own kernel rather than a predefined one, you can shift, take the absolute value of, and/or clip the results of the convolution. You can also control the center pixel of the kernel.

The **Thread** parameter specifies the thread to which to send **imIntConvolve()** for execution.

The **Src** parameter specifies the buffer on which to perform the convolution. This can be an 8-bit or 16-bit integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the convolution. For user-defined kernels, this buffer can be of any integer type. For predefined kernels, this buffer must be of the same type as the source buffer. Note that in-place operation is not supported for this function.

The **Kernel** parameter specifies the kernel with which to perform the convolution. It can be set to the identifier of a kernel buffer or to a predefined kernel. The kernel buffer must be a single-band, 8- or 16-bit integer buffer. If the source buffer has more than one band, the kernel is used on each band.

The maximum kernel width or height supported for a convolution with the NOA is normally 33, but is dependent on the total number of kernel elements. The total number of kernel elements cannot exceed 512 if using a 16-bit integer buffer, and cannot exceed 1024 if using an 8-bit integer

buffer. Note that a 16-bit kernel buffer will be considered as an 8-bit buffer whenever all the kernel values can fit in 8 bits.

However, there are some of exceptions to the rule above:

- If both the image and kernel are 8-bit, then the kernel width can go up to 64. Nonetheless, the kernel height is still limited to 33 and the total number of kernel values has the same limit.
- If the kernel is 1xN (width is 1), the kernel height can go up to 64.

If your operation does not use the NOA, then the largest kernel supported is usually 15x15; however, the 'C80 does support kernels up to 31x31 if all of the following conditions are met:

- Source buffer is unsigned 8-bit.
- Destination buffer is 8- or 16-bit.
- Kernel is signed 8-bit, i.e. [-128, +127].

The predefined kernels are listed below, along with the normalization operations associated with the kernel. Note that clipping, when performed, is to the range of the destination buffer.

Identifier	Kernel
IM_HORIZ_EDGE	$clip \left[\begin{bmatrix} -2 & -2 & -2 \\ 0 & 0 & 0 \\ 2 & 2 & 2 \end{bmatrix} \right]$
IM_LAPLACIAN_EDGE	$clip \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
IM_LAPLACIAN_EDGE2	$clip \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
IM_PREWITT_EDGE	$clip \left(\left(\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \right) \gg 1 \right)$

Identifier	Kernel
IM_ROBERTS_EDGE	$clip\left(\left[\begin{bmatrix}-1 & 0 \\ 0 & 1\end{bmatrix}\right] + \left[\begin{bmatrix}0 & -1 \\ 1 & 0\end{bmatrix}\right]\right)$
IM_SHARPEN	$clip\begin{bmatrix}0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0\end{bmatrix}$
IM_SHARPEN2	$clip\begin{bmatrix}-1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1\end{bmatrix}$
IM_SMOOTH	$\begin{bmatrix}1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1\end{bmatrix} \gg 4$
IM_SOBEL_EDGE	$clip\left(\left(\left[\begin{bmatrix}1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1\end{bmatrix}\right] + \left[\begin{bmatrix}-1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1\end{bmatrix}\right]\right) \gg 1\right)$
IM_VERT_EDGE	$clip\begin{bmatrix}-2 & 0 & 2 \\ -2 & 0 & 2 \\ -2 & 0 & 2\end{bmatrix}$

If you are using your own kernel, certain fields can be added to the kernel buffer, to control the behavior of the kernel. These are listed below, with their default values in bold-face. Note that, if these fields are not added to the kernel buffer, the default values are used.

Field	Values	Meaning
IM_KER_SHIFT	0 - 31	Shift results by the specified number of bits.
IM_KER_CLIP	IM_DISABLE	Don't clip results.
	IM_ENABLE	Clip results to the range of the destination buffer.
IM_KER_ABSOLUTE	IM_DISABLE	Don't take the absolute value of results.
	IM_ENABLE	Take the absolute value of the results.
IM_KER_CENTER_X	0 – (Xsize-1) default: int(Xsize-1)/2	X coordinate of kernel center.
IM_KER_CENTER_Y	0 – (Ysize-1) default: int(Ysize-1)/2	Y coordinate of kernel center.

Note that the order of operation is: perform a convolution, shift by the specified number of bits, take the absolute value, and then clip to the range of the destination buffer.

If you shift, take the absolute value, or clip results, the function might run slower. Therefore, only use these options if necessary.

The following field can be used to add a constant offset to the kernel buffer. However, this is only supported when using the NOA.

Field	Values	Meaning
IM_KER_OFFSET	<integer>	Constant offset added to result before normalization (default is 0).

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntConvolve()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_COMPUTATION	IM_EXACT	Don't make any approximations.
	IM_FAST	Make approximations, if appropriate, so as to increase operation speed. Note that some rounding errors (usually small) can occur if you set this field.
IM_CTL_INPUT_BITS	1 - 16 (default: source buffer pixel size)	Actual number of bits needed to represent the input data. This field can increase operation speed. For example, if a 16-bit source buffer contains only 10-bit data, operation speed might be increased if you set IM_CTL_INPUT_BITS to 10.
IM_CTL_OUTPUT_BITS	1 - 32 (default: destination buffer pixel size)	Number of bits required to hold the output data. Clipping (when enabled) is to this range.

IM_CTL_OVERSCAN	IM_TRANSPARENT	Process the bordering pixels of the source buffer using the pixels of its parent buffer as the overscan pixels. If the source buffer is not a child buffer or if its parent buffer cannot provide values for the overscan pixels, the overscan pixels will be undefined, leading to unpredictable results.
	IM_REPLACE	Process the bordering pixels of the source buffer by assigning a specific value to the overscan pixels. Specify the value with the IM_CTL_OVERSCAN_VAL field.
IM_CTL_OVERSCAN_VAL	any integer (default: 0)	Overscan replace value (used when IM_CTL_OVERSCAN is set to IM_REPLACE).

To reduce the NOA set-up time, which can be significant (especially on small images), you can save some or all of the hardware register values in a control cache buffer using the control fields that follow. Doing so can speed up processing time for a subsequent call to this function. The first call to this function will take slightly longer because the registers must be fully calculated and saved, but subsequent calls will be faster.

The increase in speed depends on the number of parameters that have changed since the setup information was saved. The increase is biggest when everything is the same (same buffers, kernel, control fields). The increase is slightly less if only the source and/or destination buffer addresses have changed (same size and type of buffer, same kernel, same control

fields). This is useful if performing a double buffering operation. There is also some set-up time saved when only the kernel is the same as before (although buffers and control fields might have changed).

IM_CTL_CACHE_BUF	BufId	Use the specified buffer as the cache buffer in which to save the list of register values (or where to find them if they were saved previously). In this case, you will save a little time on the first call to this function. Note that you should allocate a 1-dimensional, 8-bit buffer of size IM_CACHE_BUF_SIZE.
	0	Automatically allocate the cache buffer in which to save the list of register values. The buffer ID will be returned to the IM_CTL_CACHE_BUF field.
IM_CTL_SETUP	IM_SAVE	If the cache buffer was given, save registers and all other information that might be useful later. Also perform the operation.
	IM_FASTEST	Assume everything is the same as when the setup was saved.
	IM_ADDRESS_ONLY	Assume everything except the source and destination addresses are the same as when the setup was saved.
	IM_SAME_KERNEL	Assume only the kernel is the same as when the setup was saved.

Note that, whether the cache buffer is allocated automatically or you allocate it yourself, you are responsible for freeing the buffer when you no longer need it.

In addition, the following fields can be added to the control buffer. However, they are only supported when using the NOA.

Field	Values	Meaning
IM_CTL_ZOOM_X	1, 2, or 4	Horizontal zoom factor.
IM_CTL_ZOOM_Y	1, 2, or 4	Vertical zoom factor.
IM_CTL_SUBSAMP_X	1 or 2	Horizontal subsample factor.
IM_CTL_SUBSAMP_Y	1 or 2	Vertical subsample factor.
IM_CTL_COEF_TYPE	IM_DEFAULT IM_PIXEL	Normal convolution. Sum of pixels squared (SrcBuf must be unsigned 8-bit). Kernel values should be 0 or 1, where 0 implies a "don't care".

❖ Zooming is only supported for transparent overscan.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Example See *process.c* in Appendix B.

Note The NOA kernel and control features should not be used if you want your code to run on a Genesis board without a NOA.

imIntCorrelate

Async

PP

NOA

Multi-band

Synopsis Perform normalized grayscale correlation.

Format **void imIntCorrelate(Thread, Src, Dst, Model, Control, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Model; Model buffer ID
 long Control; Control buffer ID (or 0)
 long OSB; OSB ID (or 0)

Description This function performs normalized grayscale correlation on an integer image. Normalized grayscale correlation is a neighborhood operation that determines the new value (r) for a pixel based on a specified kernel (model):

$$r = \frac{N \sum IM - \left(\sum I \right) \sum M}{\sqrt{\left[N \sum I^2 - \left(\sum I \right)^2 \right] \left[N \sum M^2 - \left(\sum M \right)^2 \right]}}$$

where M = the value of a model pixel and I = the value of the underlying image pixel. All summations are over the N model pixels that are not set to a "don't care" state.

The above equation reaches its maximum value of 1 where the image and model match exactly, gives 0 where the image and model are uncorrelated, and is negative where the similarity is less than might be expected by chance (reaching -1 when the image is a negative version of the model). Note that normalized grayscale correlation is similar to convolution, but is unaffected by linear (constant gain and offset) changes in the image or model pixel values.

You can specify that r^2 be calculated instead of r to avoid the calculation of the square root. You can also specify that negative values be clipped to 0, if such values are not useful to your application.

Note that model pixels that are set to the "don't care" state are ignored in the operation. In order to set model pixels to the "don't care" state, you must supply a mask buffer (through the control buffer) that is at least as big as the model. If a pixel in the mask buffer has the value 0, its corresponding pixel in the model is set to the "don't care" state; if a pixel in the mask buffer has the value 1, its corresponding pixel in the model is used.

The **Thread** parameter specifies the thread to which to send **imIntCorrelate()** for execution.

The **Src** parameter specifies the buffer on which to perform the correlation. This must be an 8-bit unsigned integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This must be a 16-bit signed integer buffer or a 32-bit floating-point buffer. If it is floating-point, the floating-point correlation value is output (maximum value is 1.0). If it is integer, results are scaled so that the maximum correlation value is equal to the value specified by the IM_CTL_MAX_SCORE field of the control buffer.

The **Model** parameter specifies the model. This must be an unsigned 8-bit integer buffer.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntCorrelate()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_MASK_BUF	0 A buffer ID	Use all model pixels. Use the specified mask buffer to set model pixels to the "don't care" state. The buffer must be unsigned 8-bit integer. Pixels in the buffer must be 0 or 1; other values will produce undefined results.

IM_CTL_OVERSCAN	IM_TRANSPARENT	Process the bordering pixels of the source buffer using the pixels of its parent buffer as the overscan pixels. If the source buffer is not a child buffer or if its parent buffer can't provide values for the overscan pixels, the overscan pixels will be undefined, leading to unpredictable results.
	IM_REPLACE	Process the bordering pixels of the source buffer by assigning a specific value to the overscan pixels. Specify the value with the IM_CTL_OVERSCAN field.
IM_CTL_OVERSCAN_VAL	any integer (default: 0)	Overscan replace value (used when IM_CTL_OVERSCAN is set to IM_REPLACE).
IM_CTL_MAX_SCORE	any integer (default: 10000)	If the destination buffer is integer, scale results so that the maximum correlation value is equal to the specified value.
IM_CTL_SCORE_TYPE	IM_DEFAULT IM_CLIP IM_SQUARE IM_CLIP_SQUARE	Output r . Output $\max(r, 0)$. Output r^2 . Output $\max(r, 0)^2$.
IM_CTL_STEP	1 2	Use all model pixels during the correlation. To increase speed, skip every second model pixel (in both x and y) during the correlation.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also The pattern matching module.

imIntCountDifference

Async

PP

Synopsis Count the differences between two images.

Format **void imIntCountDifference(Thread, Src1, Src2, Result, OSB)**

long Thread; Thread ID
long Src1; First source buffer ID
long Src2; Second source buffer ID
long Result; Result buffer ID
long OSB; OSB ID (or 0)

Description This function counts the differences between two integer images.

The number of differences is written into the IM_RES_NUM_DIFFERENCES field of the result buffer. This number can later be read using **imBufGetField()**.

The **Thread** parameter specifies the thread to which to send **imIntCountDifference()** for execution.

The **Src1** and **Src2** parameters specify the buffers to compare. These buffers can be of any integer type but must have the same type.

The **Result** parameter specifies the buffer in which to write the number of differences. Note that this buffer's size and data type are irrelevant, since the result is written to a field.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imIntDistance

Async	PP	In-Place	Multi-band
-------	----	----------	------------

Synopsis Perform a distance transform.

Format **void imIntDistance(Thread, Src, Dst, Transform, Control, OSB)**

long Thread;	Thread ID
long Src;	Source buffer ID
long Dst;	Destination buffer ID
long Transform;	Distance transform
long Control;	Control buffer ID (or 0)
long OSB;	OSB ID (or 0)

Description This function determines the shortest distance from each foreground (non-zero) pixel to a background (zero) pixel, and assigns this distance to the foreground pixel. Background pixels remain as zero.

The **Thread** parameter specifies the thread to which to send **imIntDistance()** for execution.

The **Src** parameter specifies the buffer on which to perform the transform. This can be an 8-bit or 16-bit integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the transform. This can be an 8-bit or 16-bit integer buffer, but must be at least as deep as the source buffer (i.e., you cannot use a 16-bit source buffer and an 8-bit destination buffer).

The **Transform** parameter specifies how to calculate the distance. Specifically, it determines in which direction you can step from one pixel to the next, and how much to count for each step. This parameter can be set to:

IM_CITY_BLOCK	Count horizontal and vertical steps as 1; do not allow diagonal steps.
IM_CHESSBOARD	Count horizontal, vertical, and diagonal steps as 1.
IM_CHAMFER_3_4	Count horizontal and vertical steps as 3; count diagonal steps as 4.

Note that `IM_CHAMFER_3_4` produces the most accurate results, because it is a better approximation of the true (Euclidean) distance between a foreground and background pixel. However, it requires that the destination buffer be large enough to hold a number at least three times the maximum distance between a foreground and background pixel (even if you choose to normalize results). In general, you should be able to avoid overflows by using a 16-bit destination buffer. Note that you normalize results through the control buffer.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntDistance()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_NORMALIZE (this field only has an effect if you are using IM_CHAMFER_3_4)	IM_DISABLE	Don't normalize results.
	IM_ENABLE	Normalize results (that is, divide results by 3 so that each horizontal or vertical step is counted as 1).
IM_CTL_OVERSCAN_VAL	IM_REPLACE_MIN	Replace pixels bordering the source buffer with the minimum value that can be held in the source buffer. This has the effect that pixels outside the source buffer are considered background pixels.
	IM_REPLACE_MAX	Replace pixels bordering the source buffer with the maximum value that can be held in the source buffer. This has the effect that pixels outside the source buffer are considered foreground pixels.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imIntDyadic

Async	PP	In-Place	Multi-band
-------	----	----------	------------

Synopsis Perform an arithmetic or logical operation between two integer images.

Format **void imIntDyadic(Thread, Src1, Src2, Dst, Op, OSB)**

long Thread;	Thread ID
long Src1;	First source buffer ID
long Src2;	Second source buffer ID
long Dst;	Destination buffer ID
long Op;	Type of operation to perform
long OSB;	OSB ID (or 0)

Description This function performs an arithmetic or logical operation between two integer images. The source buffers and destination buffer of the operation can be of any integer type, although the operation might execute faster if they are all of the same type. For information on how mixed data types are handled, see the *Genesis Native Library User Guide*.

Note that, for operations that can produce overflows or underflows, results that will not fit in the destination buffer generally have their high bits discarded (in other words, these results wrap around). However, when saturation is used in an operation, results that overflow or underflow are set to the maximum or minimum value, respectively, in the destination buffer.

The **Thread** parameter specifies the thread to which to send **imIntDyadic()** for execution.

The **Src1** and **Src2** parameters specify the buffers with which to perform the operation. These buffers can be of any integer type.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This buffer can be of any integer type.

The **Op** parameter specifies the operation to perform. This parameter can be set to:

IM_ADD	Add.
IM_ADD_SAT	Add and saturate.
IM_SUB	Subtract: Src1 - Src2 .
IM_SUB_SAT	Subtract, Src1 - Src2 , and saturate.
IM_SUB_CLIP	Subtract and, if necessary, clip results so that they fit in the destination buffer. This operation should be used instead of IM_SUB_SAT when the source and destination buffers have the same pixel depth but the source buffers are unsigned while the destination buffer is signed (under this condition, IM_SUB_SAT does not produce the desired effect).
IM_SUB_ABS	Subtract and take the absolute value: $ \mathbf{Src1} - \mathbf{Src2} $.
IM_MIN	Compare Src1 and Src2 on a pixel-by-pixel basis and take the minimum of the two.
IM_MAX	Compare Src1 and Src2 on a pixel-by-pixel basis and take the maximum of the two.
IM_MULT	Multiply.
IM_MULT_MSB	Multiply and, if the product requires more bits than the destination buffer can hold, keep just the most significant bits. For example, when multiplying an 8-bit buffer by a 16-bit buffer, keep only the most significant bits of the 24-bit result.
IM_DIV	Divide: Src1 / Src2 .
IM_DIV_FRAC	Divide and, if the destination buffer has more bits per pixel than the first source buffer, use the extra bits to store the fractional part of the result. For example, if Src1 is 8-bit and Dst is 16-bit, the result will be in a $8 \cdot 8$ fixed-point format (that is, there will be 8 bits for the integer part of the result and 8 bits for the fractional part).
IM_AND	Logical AND.
IM_OR	Logical OR.
IM_XOR	Logical XOR.
IM_NAND	Logical NAND.
IM_NOR	Logical NOR.
IM_XNOR	Logical XNOR.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBinTriadic()**, **imIntMonadic()**, **imIntTriadic()**. **imBinTriadic()** performs an arithmetic or logical operation on up to three binary images. **imIntMonadic()** performs an arithmetic or logical operation between an integer image and a constant. **imIntTriadic()** performs an arithmetic or logical operation on up to three integer images.

imIntErodeDilate

Async

PP

NOA

Multi-band

Synopsis Perform grayscale erosion or dilation.

Format `void imIntErodeDilate(Thread, Src, Dst, Kernel, Op, Niter, Control, OSB)`

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Kernel; Kernel buffer ID
 long Op; Type of operation to perform
 long Niter; Number of iterations
 long Control; Control buffer ID (or 0)
 long OSB; OSB ID (or 0)

Description This function performs a grayscale erosion or dilation on an integer image, using a specified structuring element (kernel). You can use your own kernel or a predefined 3x3 kernels of all 0's.

When you use your own kernel, you can control the center pixel of the kernel. In addition, any kernel value set to IM_DONT_CARE is ignored during the operation.

The **Thread** parameter specifies the thread to which to send **imIntErodeDilate()** for execution.

The **Src** parameter specifies the buffer on which to perform the operation. This can be an 8-bit or 16-bit integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This must be of the same type as the source buffer. Note that in-place operation is not supported for this function.

The **Kernel** parameter specifies the kernel with which to perform the operation. It can be set to the identifier of a kernel buffer or to IM_3X3_RECT_0, which is a predefined 3x3 kernel of all 0's. Note that the predefined kernel simply takes the minimum pixel value in the neighborhood (for an erosion) or the maximum pixel value (for a dilation).

If you are using your own kernel, it can be of any data type. In addition, it must be 16-bit signed integer if you are using IM_DONT_CARE values.

The maximum kernel width or height supported for a convolution with the NOA is normally 33, but is dependent on the total number of kernel elements. The total number of kernel elements cannot exceed 512. For example, a 23x23 kernel is too big because the number of kernel elements is 529. However, if your operation does not use the NOA, then the largest kernel must be no larger than 15x15.

Note that certain fields can be added to your kernel, to indicate its center position. These are listed below, with their default values. If these fields are not added to the kernel buffer, the default values are used.

Field	Values	Meaning
IM_KER_CENTER_X	0 – (Xsize-1) default: int(Xsize-1)/2	X coordinate of kernel center.
IM_KER_CENTER_Y	0 – (Ysize-1) default: int(Ysize-1)/2	Y coordinate of kernel center.

The **Op** parameter specifies the type of operation to perform. It can be set to:

IM_ERODE	Erosion.
IM_DILATE	Dilation.

The **Niter** parameter specifies the number of times to apply the operation.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntErodeDilate()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_OVERSCAN	IM_TRANSPARENT	Process the bordering pixels of the source buffer using the pixels of its parent buffer as the overscan pixels. If the source buffer is not a child buffer or if its parent buffer cannot provide values for the overscan pixels, the overscan pixels will be undefined, leading to unpredictable results.

IM_CTL_OVERSCAN	IM_REPLACE	Process the bordering pixels of the source buffer by assigning a specific value to the overscan pixels. Specify the value with the IM_CTL_OVERSCAN_VAL field.
IM_CTL_OVERSCAN_VAL	any integer (default: 0)	Overscan replace value (used when IM_CTL_OVERSCAN is set to IM_REPLACE).

To reduce the NOA set-up time, which can be significant (especially on small images), you can save some or all of the hardware register values in a control cache buffer using the control fields that follow. Doing so can speed up processing time for a subsequent call to this function. The first call to this function will take slightly longer because the registers must be fully calculated and saved, but subsequent calls will be faster.

The increase in speed depends on the number of parameters that have changed since the setup information was saved. The increase is biggest when everything is the same (same buffers, kernel, control fields). The increase is slightly less if only the source and/or destination buffer addresses have changed (same size and type of buffer, same kernel, same control fields). This is useful if performing a double buffering operation. There is also some set-up time saved when only the kernel is the same as before (although buffers and control fields might have changed).

IM_CTL_CACHE_BUF	BufId	Use the specified buffer as the cache buffer in which to save the list of register values (or where to find them if they were saved previously). In this case, you will save a little time on the first call to this function. Note that you should allocate a 1-dimensional, 8-bit buffer of size IM_CACHE_BUF_SIZE. Automatically allocate the cache buffer in which to save the list of register values. The buffer ID will be returned to the IM_CTL_CACHE_BUF field.
	0	

IM_CTL_SETUP	IM_SAVE	If the cache buffer was given, save registers and all other information that might be useful later. Also perform the operation.
	IM_FASTEST	Assume everything is the same as when the setup was saved.
	IM_ADDRESS_ONLY	Assume everything except the source and destination addresses are the same as when the setup was saved.
	IM_SAME_KERNEL	Assume only the kernel is the same as when the setup was saved.

Note that, whether the cache buffer is allocated automatically or you allocate it yourself, you are responsible for freeing the buffer when you no longer need it.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBinMorphic()**. This function performs morphological operations (including erosions and dilations) on binary images.

Note This function treats overflows differently depending on whether the NOA or 'C80 is used. When using the NOA, overflows (or underflows) are properly clipped to the maximum (or minimum) value that can be represented in the destination buffer. When the 'C80 is used, overflows are not clipped.

imIntFindExtreme

Async

PP

Synopsis Find the minimum and/or maximum pixel value in an image.

Format **void imIntFindExtreme(Thread, Src, Result, Mode, OSB)**

long Thread;	Thread ID
long Src;	Source buffer ID
long Result;	Result buffer ID
long Mode;	Operation mode
long OSB;	OSB ID (or 0)

Description This function finds the minimum and/or maximum pixel value in an image.

The minimum pixel value is written into the IM_RES_MIN_PIXEL field of the result buffer; the maximum pixel value is written into the IM_RES_MAX_PIXEL field of the result buffer. These fields can later be read using **imBufGetField()**.

The **Thread** parameter specifies the thread to which to send **imIntFindExtreme()** for execution.

The **Src** parameter specifies the buffer on which to perform the operation. This buffer can be of any integer type.

The **Result** parameter specifies the buffer in which to write the results of the operation. Note that this buffer's size and data type are irrelevant, since the result is written to a field.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_MIN_PIXEL	Find the minimum value.
IM_MAX_PIXEL	Find the maximum value.
IM_MIN_PIXEL+IM_MAX_PIXEL	Find the minimum and maximum value.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imIntLocateEvent()**. This function can be used to determine the position of the minimum or maximum pixel.

imIntFFT

Async

PP

In-Place

Synopsis Perform a fast Fourier transform.

Format **void imIntFFT(Thread, SrcR, SrcI, DstR, DstI, Control, OSB)**

long Thread;	Thread ID
long SrcR;	Source buffer ID (real part)
long SrcI;	Source buffer ID (imaginary part)
long DstR;	Destination buffer ID (real part)
long DstI;	Destination buffer ID (imaginary part)
long Control;	Control buffer ID (or 0)
long OSB;	OSB ID (or 0)

Description This function performs a fast Fourier transform (FFT) on an integer image. It can also be used to perform an inverse FFT on a transformed image.

Note that **imIntFFT()** uses a fixed-point integer representation of the image. This is faster than using a floating-point representation. It can also be just as accurate if you left-shift the input image by enough bits before performing the transform. To avoid overflows, you should then enable normalization (through the control buffer). Normalization will right-shift results at each stage of the transform so that the dynamic range does not get larger.

The **Thread** parameter specifies the thread to which to send **imIntFFT()** for execution.

The **SrcR** parameter specifies the real part of the source image. This must be a signed 32-bit integer buffer.

The **SrcI** parameter specifies the imaginary part of the source image. This must be a signed 32-bit integer buffer. If the source image is real, you must clear this buffer to zero before calling **imIntFFT()**.

The **DstR** parameter specifies the buffer in which to write the real part of the result. This must be a signed 32-bit integer buffer.

The **DstI** parameter specifies the buffer in which to write the imaginary part of the result. This must be a signed 32-bit integer buffer.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntFFT()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_DIRECTION	IM_FORWARD	Perform an FFT.
	IM_REVERSE	Perform an inverse FFT.
IM_CTL_BLOCK_X	IM_ALL	X-block size is full width of image (use this for a normal 2D FFT).
	1	X-block size is 1 column i.e. perform a 1D FFT on all the columns of the image (IM_CTL_BLOCK_Y should be set to IM_ALL).
IM_CTL_BLOCK_Y	IM_ALL	Y-block size is full height of image (use this for a normal 2D FFT).
	1	Y-block size is 1 row i.e. perform a 1D FFT on all the rows of the image (IM_CTL_BLOCK_X should be set to IM_ALL).
IM_CTL_NORMIMIZE	IM_ENABLE	Normalize results (divide by 2 after each stage of the FFT).
	IM_DISABLE	Don't normalize results.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note The width and height of the source and destination buffers must be a power of 2.

Example See *process.c* in Appendix B.

imIntFlip

Async

PP

Multi-band

Synopsis Flip or rotate an image.

Format **void imIntFlip(Thread, Src, Dst, Func, Mode, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Func; Type of operation to perform
 long Mode; Operation mode
 long OSB; OSB ID (or 0)

Description This function flips an image (horizontally or vertically) or rotates it (by 90, 180, or 270°). Note that this function is faster than the more general warping functions (**imIntWarp...()**) at performing these operations.

The **Thread** parameter specifies the thread to which to send **imIntFlip()** for execution.

The **Src** parameter specifies the buffer on which to perform the transform. This buffer can be of any integer type, or can be a 32-bit floating-point buffer.

The **Dst** parameter specifies the buffer in which to place the results of the transform. This buffer must have the same pixel size as the source buffer. Note that in-place operation is not supported for this function.

The **Func** parameter specifies the type of transform to perform. It can be set to:

IM_FLIP_H	Flip horizontally (left to right).
IM_FLIP_V	Flip vertically (top to bottom).
IM_ROTATE_90	Rotate 90° counter-clockwise.
IM_ROTATE_180	Rotate 180° counter-clockwise.
IM_ROTATE_270	Rotate 270° counter-clockwise.

The **Mode** parameter specifies the mode of operation. This parameter must be set to IM_DEFAULT.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note If rotating by 90 or 270°, the x and y dimensions of the source buffer are swapped. Therefore, the destination buffer should be of an appropriate size to hold all of the result. If, for example, the source buffer is 512x480, the destination buffer should be 480x512 to hold all of the result.

See also **imIntWarpPolynomial()**. The **imIntWarpPolynomial()** function can rotate an image by any angle, but takes longer to perform.

imIntGainOffset

Async

PP

In-Place

Multi-band

Synopsis Apply per-pixel gain and offset correction.

Format `void imIntGainOffset(Thread, Src, Dst, Offset, Gain, Shift, ClipVal, Mode, OSB)`

long Thread;	Thread ID
long Src;	Source buffer ID
long Dst;	Destination buffer ID
long Offset;	Offset buffer ID (or 0)
long Gain;	Gain buffer ID
long Shift;	Number of bits to shift
long ClipVal;	Constant used to replace overflows
long Mode;	Operation mode
long OSB;	OSB ID (or 0)

Description This function applies per-pixel gain and offset correction to an integer image: **Dst** = ((**Src** - **Offset**) * **Gain**) >> **Shift**.

You can clip the results of the operation, in which case underflows are set to zero after the offset is subtracted and overflows in the final result are set to a specified integer value. Note that you can set overflows to any integer value. If, for example, you are applying gain and offset correction to 10-bit data (range 0 - 1023), you might want to use a 16-bit destination buffer and clip overflows to 1023 (so that any result exceeding 1023 is set to this value). Alternatively, you might want to use an 8-bit destination buffer, a larger right-shift, and clip overflows to 255, to display the result of the correction.

The **Thread** parameter specifies the thread to which to send **imIntGainOffset()** for execution.

The **Src** parameter specifies the buffer on which to apply the correction. This must be an 8-bit or 16-bit unsigned integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the correction. This must be of the same type as the source buffer, or have a smaller pixel size than the source buffer.

The **Offset** parameter specifies the offset buffer. This must be of the same type as the source buffer. If this buffer is one-dimensional, it is used on each line of the source buffer. Note that this parameter can be set to 0, in which case no offset is subtracted.

The **Gain** parameter specifies the gain buffer. This must be an 8-bit or 16-bit, unsigned integer buffer. If this buffer is one-dimensional, it is used on each line of the source buffer.

The **Shift** parameter specifies the number of bits by which to right-shift. Note that, if the gain buffer is in fixed-point format, an appropriate shift must be applied.

The **ClipVal** parameter specifies the integer value to which to set overflows.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_DEFAULT	Don't clip results.
IM_CLIP	Clip results (that is, set underflows to zero after the offset is subtracted and set overflows in the final result to ClipVal).

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imIntMac1()**, **imIntMac2()**. These functions perform a similar arithmetic operation.

Example See *process.c* in Appendix B.

imIntHistogram

Async PP Multi-band

Synopsis Perform a histogram.

Format void imIntHistogram(Thread, Src, Result, Mode, OSB)

long Thread; Thread ID
long Src; Source buffer ID
long Result; Result buffer ID
long Mode; Operation mode
long OSB; OSB ID (or 0)

Description This function performs a histogram on an integer image.

The **Thread** parameter specifies the thread to which to send **imIntHistogram()** for execution.

The **Src** parameter specifies the buffer on which to perform the histogram. This must be an 8-bit or 16-bit integer buffer. Note that the buffer's values are always considered to be unsigned, even if the buffer is of a signed type.

The **Result** parameter specifies the buffer in which to write the results of the histogram. This must be a one-dimensional, 16-bit or 32-bit integer buffer. To guarantee maximum speed, the buffer should be no larger than necessary. However, it should be larger than the maximum pixel value in the source image, or results will be unpredictable. For 10-bit data, an appropriate size is $2^{10}=1024$.

To speed up the operation, certain fields can be added to the result buffer. These are listed below, with their default values in bold-face. Note that, if these fields are not added to the result buffer, the default values are used.

Field	Values	Meaning
IM_CTL_INPUT_BITS (this field only applies to 16-bit source buffers)	9 - 16 (default value depends on result buffer size)	Input data size. If the input data size exceeds the result buffer size, input pixels are right shifted accordingly.
IM_CTL_SUBSAMP_X	1 - 16	Only use the columns of the source buffer that are multiples of the specified value when generating the histogram, starting with the first column.

IM_CTL_SUBSAMP_Y 1 - 16

Only use the row of the source buffer that are multiples of the specified value when generating the histogram, starting with the first row.

The **Mode** parameter specifies the mode of operation. This parameter must be set to IM_DEFAULT.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Example See *process.c* in Appendix B.

See also **imIntHistogramEqualize()**. You can use this function to transform an image's histogram into a look-up table with which to map the image.

imIntHistogramEqualize

Async

PP

In-Place

Multi-band

Synopsis Perform a histogram equalization.

Format `void imIntHistogramEqualize(Thread, Src, Dst, HistSize, Func, Alpha, Min, Max, Mode, OSB)`

long Thread;	Thread ID
long Src;	Source buffer ID
long Dst;	Destination buffer ID
long HistSize;	Intermediate histogram size
long Func;	Type of equalization to perform
double Alpha;	Adjustment parameter
long Min;	Lowest pixel value to equalize
long Max;	Highest pixel value to equalize
long Mode;	Operation mode
long OSB;	OSB ID (or 0)

Description This function performs a histogram equalization on an integer image, or generates a look-up table (LUT) which can later be used to equalize an image (using, for example, **imIntLutMap()**). In the former case, a histogram is performed on the source image, the histogram is transformed into a LUT using the specified equalization operation, and then this LUT is used to transform the source image. In the latter case, a histogram is transformed into a LUT using the specified equalization operation; the histogram can be of the source image or user-supplied.

Note that you perform histograms using **imIntHistogram()**.

The **Thread** parameter specifies the thread to which to send **imIntHistogramEqualize()** for execution.

The **Src** parameter specifies the image buffer from which to generate the histogram, or the histogram buffer from which to generate the LUT. The image buffer must be 8-bit or 16-bit integer; the histogram buffer must be 16-bit or 32-bit integer. Note that the image buffer's values are always considered to be unsigned, even if this buffer is of a signed type.

The **Dst** parameter specifies the image buffer in which to write the results of the equalization, or the LUT buffer in which to write the generated LUT. The image or LUT buffer must be 8-bit or 16-bit integer.

The **HistSize** parameter specifies the size required for the histogram of the source image. To guarantee maximum speed, this should be no larger than necessary. However, it should be larger than the maximum pixel value in the source image, or results will be unpredictable. For 10-bit data, an appropriate size is $2^{10}=1024$.

The **Func** parameter specifies the type of equalization operation with which to generate the LUT. The table below lists available equalization operations and their density functions.

	Output probability density model	Transfer functions
IM_UNIFORM	$P_g(g) = \frac{1}{g_{max} - g_{min}}$ $g_{min} \leq g \leq g_{max}$	$g = [g_{max} - g_{min}]P_f(f) + g_{min}$
IM_EXPONENTIAL	$P_g(g) = \alpha \left(e^{[(-\alpha)(g - g_{min})]} \right)$ $g \geq g_{min}$	$g = g_{min} - \frac{1}{\alpha} \ln[1 - P_f(f)]$
IM_RAYLEIGH	$P_g(g) = \frac{g - g_{min}}{\alpha^2} e^{\left[-\frac{(g - g_{min})^2}{2\alpha^2} \right]}$ $g \geq g_{min}$	$g = g_{min} + \left[2\alpha^2 \ln \left\{ \frac{1}{1 - P_f(f)} \right\} \right]^{1/2}$
IM_HYPER_CUBE_ROOT	$P_g(g) = \frac{1}{3} \left(\frac{g^{-2/3}}{\sqrt[3]{g_{max}} - \sqrt[3]{g_{min}}} \right)$	$g = \left(\left[\sqrt[3]{g_{max}} - \sqrt[3]{g_{min}} \right] [P_f(f)] + \sqrt[3]{g_{min}} \right)^3$
IM_HYPER_LOG	$P_g(g) = \frac{1}{g[\ln(g_{max}) - \ln(g_{min})]}$	$g = g_{min} \left[\frac{g_{max}}{g_{min}} \right]^{P_f(f)}$
<p>- The cumulative probability distribution, $P_f(f)$, of the input image is approximated by its cumulative histogram:</p> $P_f(f) \approx \sum_{m=0}^i H_F(m)$ <p>- Refer to Digital Image Processing, William K. Pratt, United States, John Wiley & Sons, 1978, p.318.</p>		

The **Alpha** parameter specifies the value of the " α " constant used in the equations for IM_EXPONENTIAL and IM_RAYLEIGH. For IM_EXPONENTIAL, a greater **Alpha** results in a LUT with a lower occurrence of the most frequent histogram values. For IM_RAYLEIGH, a greater **Alpha** results in a LUT with a higher occurrence of the most frequent histogram values.

The **Min** and **Max** parameters specify the range of pixel values in the source image on which to apply the equalization. Pixel values outside this range are ignored.

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_DEFAULT	Perform an equalization on the source image.
IM_IMAGE_TO_LUT	Generate a LUT from a histogram of the source image.
IM_HIST_TO_LUT	Generate a LUT from the supplied histogram.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Example The following code performs uniform histogram equalization on an 8-bit image:

```
imInthHistogramEqualize(Thread, Buf, Buf, 256, IM_UNIFORM, 0.0, 0, 255, IM_DEFAULT, 0);
```

imIntLabel

Async

PP

Multi-band

Synopsis Label connected regions.**Format** **void imIntLabel(Thread, Src, Dst, Mode, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Mode; Labelling mode
 long OSB; OSB ID

Description This function labels each connected region in an integer image. Starting from the top left corner, connected regions are given unique consecutive values starting with 1.

Note that a connected region is an area of touching pixels with non-zero values. Pixels are considered touching if they are horizontally or vertically adjacent. If they are diagonally adjacent, you can specify whether or not they are considered touching.

The number of labelled regions is written into the IM_RES_MAX_PIXEL field of the result buffer. This number can later be read using **imBufGetField()**.

The **Thread** parameter specifies the thread to which to send **imIntLabel()** for execution.

The **Src** parameter specifies the buffer to label. This must be an 8-bit or 16-bit integer buffer.

The **Dst** parameter specifies the buffer in which to write the number of labelled regions. This must be an 8-bit or 16-bit integer buffer.

The **Mode** parameter specifies whether or not diagonally adjacent pixels are considered touching. It can be set to:

IM_4_CONNECTED Don't consider diagonally adjacent pixels as touching.
 IM_8_CONNECTED Consider diagonally adjacent pixels as touching.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note A complex image with many regions can take long to label. If your image is noisy, it might be more efficient to first filter it to remove unwanted regions caused by noise.

imIntLocateEvent

Async

PP

Synopsis Locate pixels that satisfy a condition.

Format **void imIntLocateEvent(Thread, Src, X, Y, Pix, Num, Cond, Low, High, OSB)**

long Thread;	Thread ID
long Src;	Source buffer ID
long X;	Buffer ID for X coordinates (or 0)
long Y;	Buffer ID for Y coordinates (or 0)
long Pix;	Buffer ID for pixel values (or 0)
long Num;	Buffer ID for number of events
long Cond;	Conditional operator
long Low;	Constant
long High;	Constant
long OSB;	OSB ID (or 0)

Description This function locates pixels in an integer image that satisfy a specified condition.

The number of located pixels is written into the `IM_RES_NUM_EVENTS` field of the result buffer. This number can later be read using **imBufGetField()**.

The **Thread** parameter specifies the thread to which to send **imIntLocateEvent()** for execution.

The **Src** parameter specifies the buffer whose pixels to locate. This buffer can be of any integer type.

The **X** parameter specifies the one-dimensional buffer in which to write the x coordinates of the located pixels. This must be a 16-bit or 32-bit integer buffer. If this buffer is too small to hold all results, some will be lost. Note that this parameter can be set to 0, in which case the x coordinates will not be written.

The **Y** parameter specifies the one-dimensional buffer in which to write the y coordinates of the located pixels. This must be a 16-bit or 32-bit integer buffer, and have the same type as the **X** buffer. If this buffer is too small to hold all results, some will be lost. Note that this parameter can be set to 0, in which case the y coordinates will not be written.

The **Pix** parameter specifies the one-dimensional buffer in which to write the pixel values of the located pixels. This buffer must be of the same type as the source buffer. If this buffer is too small to hold all results, some will be lost. Note that this parameter can be set to 0, in which case the pixel values will not be written.

The **Num** parameter specifies the buffer in which to write the number of located pixels. Note that this buffer's size and data type are irrelevant, since the result is written to a field. In addition, even if the buffers used to hold results are not large enough, the number written to the field will always be correct.

The **Cond** parameter specifies the condition with which to locate pixels. It can be set to:

IM_EQUAL	if equal to Low .
IM_NOT_EQUAL	if not equal to Low .
IM_LESS	if less than Low .
IM_LESS_OR_EQUAL	if less than or equal to Low .
IM_GREATER	if greater than Low .
IM_GREATER_OR_EQUAL	if greater than or equal to Low .
IM_IN_RANGE	if within Low to High , inclusive.
IM_OUT_RANGE	if less than Low or greater than High .

The **Low** and **High** parameters specify integer constants. For cases where **High** is not used, any value can be given for it.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imIntLutMap

Async

PP

In-Place

Multi-band

Synopsis Perform a look-up table mapping.**Format** **void imIntLutMap(Thread, Src, Dst, Lut, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Lut; LUT buffer ID
 long OSB; OSB ID (or 0)

Description This function maps an integer image using the specified look-up table (LUT). Note that you can use the **imGen1d()** function to generate the LUT.

The speed of the LUT mapping decreases as the LUT size increases. To help speed up the operation, this function can perform both non-interpolated LUT mappings and interpolated LUT mappings. A non-interpolated LUT mapping does not subsample the LUT buffer, so the actual values in the LUT buffer are used. An interpolated LUT mapping subsamples the LUT buffer to reduce its size and, therefore, increase speed.

The **Thread** parameter specifies the thread to which to send **imIntLutMap()** for execution.

The **Src** parameter specifies the buffer to map. This can be an 8, 16, or 32-bit integer buffer, with the exception of interpolated modes which do not accept 32-bit source buffers. Note that the buffer's values are always considered to be unsigned, even if the buffer is of a signed type.

The **Dst** parameter specifies the buffer in which to place the results of the mapping. This can be of any integer type, but must be of the same type as the LUT buffer.

The **Lut** parameter specifies the look-up table buffer. The LUT buffer should be no larger than necessary, to guarantee maximum speed. However, it should be no smaller than the maximum pixel value that will be mapped through it; otherwise, the results will be unpredictable.

Certain fields can be added to the LUT buffer to control the behavior (mode) of the mapping. These fields and their corresponding LUT mapping behaviors are described below, with their default values in bold-face. Each LUT mapping mode is also fully described in *Chapter 3:Processing Functions* of the *Genesis Native Library User Guide*.

Basic mode. The most basic (non-interpolated) mode is used when no control fields are added to the LUT buffer to specify another mode. The operation used is,

$$\mathbf{Dst} = \mathbf{Lut}[\mathbf{Src}]$$

In this mode, make sure that no pixel value in the source buffer exceeds the size of the LUT buffer. For example, with 10-bit data, an appropriate LUT size is $2^{10}=1024$.

For LUTs bigger than 16 KBytes, the function can be quite slow in this mode because the LUT can not fit entirely in on-chip RAM. In this case, the function can work in one of two ways internally, depending on whether it uses the default or PP option.

By default, large LUTs are processed using the transfer controller of the 'C80, making the operation strongly data-dependent. In general, this method should only be used for 16:32 mappings.

Using the PP option, you can increase the speed of the operation for large LUTs in a way that makes multiple passes with the PPs, but still uses the basic non-interpolated LUT mode. This option requires a large temporary work buffer for temporary data storage. You can specify whether to have the temporary work buffer automatically allocated, or you can supply it yourself, by adding the IM_CTL_WORK_BUF field to the LUT buffer as follows:

Field	Values	Meaning
IM_CTL_WORK_BUF	0	Automatically allocate a temporary work buffer for the PP option.
	A buffer ID	Use the specified buffer as a temporary work buffer.

Note that performance drops if this buffer is too small, so be sure you have enough memory for a large work buffer.

Shift & mask mode. Another (non-interpolated) mode of this function allows source values to be right shifted and/or masked before indexing the LUT. The operation performed is,

$$\mathbf{Dst} = \mathbf{Lut}[(\mathbf{Src} \gg \mathit{shift}) \& \mathit{mask}]$$

This mode is particularly useful when the source buffer contains negative values (it saves a separate masking operation), or when you want to increase the speed of the operation by using a smaller LUT (it saves a separate right shift operation). In this mode, the source buffer must be 16- or 32-bit. The amount of shift and the mask value are both determined from the difference between the dynamic range of the input data and the size of the LUT buffer used. To select this mode you need to specify the dynamic range of the source pixels (size of the input data) with the IM_CTL_INPUT_BITS field as follows:

Field	Values	Meaning
IM_CTL_INPUT_BITS	9-16	The input data size when performing a non-interpolated LUT mapping on a 16-bit source buffer.
	17-32	The input data size when performing a non-interpolated LUT mapping on a 32-bit source buffer.

Clip mode. A third (non-interpolated) mode of this function allows you to have large values clipped to the maximum value that the LUT can handle. The operation performed is,

$$\mathbf{Dst} = \mathbf{Lut}[\mathit{clip}(\mathbf{Src})]$$

Keep in mind that unsigned clipping is performed. That is, only values above the upper limit are clipped, so you should not have any negative values in your source buffer. There is no right shift in this mode; therefore, the dynamic range of the source data is always deduced from the LUT's size, and the clipping value is equal to (Size of LUT - 1). For example, 1023 for a 1024-entry (10-bit) LUT. To select this mode, you need to enable clipping as follows:

Field	Values	Meaning
IM_CTL_CLIP	IM_ENABLE	Clip the largest input values to avoid reading beyond the end of the LUT.
	IM_DISABLE	Assume that the data has already been clipped.

❖ Note that when you want this mode, you should not add the IM_CTL_INPUT_BITS field to the LUT buffer.

Interpolated mode. Alternatively, you can improve performance with large LUTs by using an interpolated LUT. When using an interpolated LUT, the output for each 16-bit input is determined by linearly interpolating between two values of the subsampled LUT. As a result, in this mapping mode, the LUT buffer is subsampled to reduce its size (that is, number of entries x number of bytes) to 16 KByte. You can control the subsampling of the LUT buffer by specifying one of the following modes:

Field	Values	Meaning
IM_CTL_RESAMPLE	IM_INTERPOLATE	Use an interpolated (subsampled) LUT.
	IM_NO_INTERPOLATE	Do not use an interpolated (subsampled) LUT.

For example, an interpolated LUT can be used to perform a 16:8 or 16:32 LUT mapping. (Note that since a 16:32 LUT mapping is normally used to display an image in pseudo-color, a 32-bit destination buffer is always assumed to be in RGBa format.)

In interpolated mode, the output for each 16-bit input is determined by linearly interpolating between two values of the subsampled LUT. You need to specify the number of input bits in the control fields that follow. When performing an interpolated LUT mapping, you might prefer to provide the LUT already scaled down, since it will be quicker to generate and load. In this case, the input data size will be bigger than the LUT size (for example, 16-bit data with a 12-bit LUT). Therefore, you must also specify the input data size, since the input data size is normally deduced from the size of the LUT.

Field	Values	Meaning
IM_CTL_INPUT_BITS	9-16 (default value depends on the LUT size)	Input data size.

Optionally, in the interpolated mode, you can specify that input values be clipped to avoid reading beyond the end of the subsampled LUT. To select this clipped mode, you need to enable clipping as follows:

Field	Values	Meaning
IM_CTL_CLIP	IM_ENABLE	Clip the largest input values, to avoid reading beyond the end of the LUT.
	IM_DISABLE	Assume that the data has already been clipped.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note The IM_CTL_INPUT_BITS and IM_CTL_CLIP fields produce different results depending on whether this function uses an interpolated or non-interpolated LUT mapping mode.

imIntMac1

Async

PP

In-Place

Multi-band

Synopsis Multiply and accumulate with one integer image.

Format `void imIntMac1(Thread, Src, Dst, Fac, Const, Shift, OSB)`

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Fac; Constant
 long Const; Constant
 long Shift; Constant
 long OSB; OSB ID (or 0)

Description This function multiplies an integer image by a specified factor, adds a specified constant to the result, and then shifts results by the specified number of bits. Final results are written into the destination buffer:

$Dst = (Fac * Src + Const) \gg Shift$.

Note that a shift is useful if the images contain data in fixed-point format. The shift is a signed shift if the source buffer is signed.

The **Thread** parameter specifies the thread to which to send **imIntMac1()** for execution.

The **Src** parameter specifies the buffer with which to perform the operation. This can be an 8-bit or 16-bit integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This buffer can be of any integer type.

The **Fac** parameter specifies the factor with which to multiply the source buffer. This can be an integer value between -32768 and 32767 if the source buffer is signed; between 0 and 65535 if the source buffer is unsigned.

The **Const** parameter specifies the constant to add to the scaled source buffer.

The **Shift** parameter specifies the number of bits by which to right-shift. This can be an integer value between 0 and 31.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note After the multiply stage of the operation, internal precision is always 32 bits.

See also **imIntGainOffset()**, **imIntMac2()**. These functions perform a similar arithmetic operation.

imIntMac2

Async

PP

In-Place

Multi-band

Synopsis Multiply and accumulate with two integer images.

Format `void imIntMac2(Thread, Src1, Src2, Dst, Fac1, Fac2, Shift, OSB)`

<code>long Thread;</code>	Thread ID
<code>long Src1;</code>	First source buffer ID
<code>long Src2;</code>	Second source buffer ID
<code>long Dst;</code>	Destination buffer ID
<code>long Fac1;</code>	Constant
<code>long Fac2;</code>	Constant
<code>long Shift;</code>	Constant
<code>long OSB;</code>	OSB ID (or 0)

Description This function multiplies two integer images by specified factors, adds the results, then shifts the sum by the specified number of bits. Final results are written into the destination buffer:

Dst = (Fac1*Src1 + Fac2*Src2) >> Shift.

The **Thread** parameter specifies the thread to which to send **imIntMac2()** for execution.

The **Src1** and **Src2** parameters specify the buffers with which to perform the operation. These can be 8-bit or 16-bit integer buffers, but both must either be signed or unsigned.

The **Dst** parameter specifies the buffer in which to write the results of the operation. This buffer can be of any integer type.

The **Fac1** and **Fac2** parameters specify the factors with which to multiply the first and second source buffers, respectively. These parameters can have an integer value between -32768 and 32767 if the source buffers are signed; between 0 and 65535 if the source buffers are unsigned.

The **Shift** parameter specifies the number of bits by which to right-shift. This can be an integer value between 0 and 31. Note that the shift is a signed shift if the source buffers are signed.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note After the multiply stage of the operation, internal precision is always 32 bits.

See also **imIntGainOffset()**, **imIntMac1()**. These functions perform a similar arithmetic operation.

imIntMonadic

Async

PP

In-Place

Multi-band

Synopsis Perform an arithmetic or logical operation between an integer image and a constant.

Format `void imIntMonadic(Thread, Src, Const, Dst, Op, OSB)`

long Thread; Thread ID
 long Src; Source buffer ID
 double Const; Constant
 long Dst; Destination buffer ID
 long Op; Type of operation to perform
 long OSB; OSB ID (or 0)

Description This function performs an arithmetic or logical operation between an integer image and a constant. The image buffers and constant can be of any integer type, although the operation might execute faster if they are of the same type. For information on how to handle mixed data types, see the *Genesis Native Library User Guide*.

Note that, for operations that can produce overflows or underflows, results that will not fit in the destination buffer generally have their high bits discarded (in other words, these results wrap around). However, when saturation is used in an operation, results that overflow or underflow are set to the maximum or minimum value, respectively, in the destination buffer.

The **Thread** parameter specifies the thread to which to send **imIntMonadic()** for execution.

The **Src** parameter specifies the buffer with which to perform the operation. This buffer can be of any integer type.

The **Const** parameter specifies the constant to use in the operation. The effective type of the constant is determined from its actual value.

The **Dst** parameter specifies the buffer in which to write the results of the operation. This buffer can be of any integer type.

The **Op** parameter specifies the type of operation to perform. It can be set to:

IM_ADD	Add.
IM_ADD_SAT	Add and saturate.
IM_SUB	Subtract: Src - Const .
IM_SUB_SAT	Subtract, Src - Const , and saturate.
IM_SUB_ABS	Subtract and take the absolute value: Src - Const .
IM_SUB_NEG	Subtract: Const - Src .
IM_MIN	Compare Src and Const on a pixel-by-pixel basis and take the minimum of the two.
IM_MAX	Compare Src and Const on a pixel-by-pixel basis and take the maximum of the two.
IM_MULT	Multiply.
IM_MULT_MSB	Multiply and, if the product requires more bits than the destination buffer can hold, keep just the most significant bits.
IM_DIV	Divide by Const : Src/Const .
IM_DIV_FRAC	Divide by Const and, if the destination buffer has more bits per pixel than the source buffer, use the extra bits to store the fractional part of the result.
IM_DIV_INT0	Divide into Const : Const/Src .
IM_SHIFT	Bitwise shift by Const (left-shift if Const is positive; right-shift if Const is negative).
IM_AND	Logical AND.
IM_OR	Logical OR.
IM_XOR	Logical XOR.
IM_NAND	Logical NAND.
IM_NOR	Logical NOR.
IM_XNOR	Logical XNOR.

In addition to the above operations, you can perform certain unary operations, by appropriately setting the **Const** parameter. For example, you can negate the source image (set **Op** to IM_SUB_NEG and **Const** to 0), take its absolute value (set **Op** to IM_SUB_ABS and **Const** to 0), or perform a logical NOT (set **Op** to IM_XOR and **Const** to 0xffffffff for 32-bit images, 0xffff for 16-bit images, or 0xff for 8-bit images).

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBinTriadic()**, **imIntDyadic()**, **imIntTriadic()**. **imBinTriadic()** performs an arithmetic or logical operation on up to three binary images. **imIntDyadic()** performs an arithmetic or logical operation between two integer images. **imIntTriadic()** performs an arithmetic or logical operation on up to three integer images.

imIntProject

Async

PP

Multi-band

Synopsis Project a 2D integer image into 1D.

Format **void imIntProject(Thread, Src, Result, Angle, Mode, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Result; Result buffer ID
 double Angle; Angle to project
 long Mode; Operation mode
 long OSB; OSB ID (or 0)

Description This function adds pixel values along all lines in an image that are at a specified angle (called the projection angle). The sum of pixel values along each of these lines is written into the result buffer.

This function can be used to perform a *column profile* (the sum of pixel values along each column in an image) or a *row profile* (the sum of pixel values along each row in an image).

The **Thread** parameter specifies the thread to which to send **imIntProject()** for execution.

The **Src** parameter specifies the buffer with which to perform the operation. This can be an 8-bit unsigned, 16-bit signed or unsigned, or 32-bit signed, integer buffer.

The **Result** parameter specifies the one-dimensional integer buffer in which to write the results of the operation. If the source buffer is a 32-bit signed buffer, this result buffer must be a 32-bit buffer. Otherwise, this result buffer can be either a 16-bit or 32-bit buffer.

The **Angle** parameter specifies the projection angle. This parameter can be set to:

0.0 Project onto a line at 0° (that is, sum all columns).
 90.0 Project onto a line at 90° (that is, sum all rows).

The **Mode** parameter specifies the mode of operation. This parameter must be set to IM_DEFAULT.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imIntRank

Async

PP

Multi-band

Synopsis Perform a rank filter operation.

Format `void imIntRank(Thread, Src, Dst, Kernel, Rank, Control, OSB)`

<code>long Thread;</code>	Thread ID
<code>long Src;</code>	Source buffer ID
<code>long Dst;</code>	Destination buffer ID
<code>long Kernel;</code>	Kernel buffer ID
<code>long Rank;</code>	Rank value
<code>long Control;</code>	Control buffer ID (or 0)
<code>long OSB;</code>	OSB ID (or 0)

Description This function performs a rank filter operation on an integer image, using a specified kernel. It sorts a pixel's neighborhood values in increasing order, and then replaces the pixel with the **Rank***th* value in the list.

The **Thread** parameter specifies the thread to which to send **imIntRank()** for execution.

The **Src** parameter specifies the buffer on which to perform the rank filter operation. This can be an 8-bit or 16-bit integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This can be an 8-bit or 16-bit integer buffer, but must be of the same type as the source buffer. Note that in-place operation is not supported for this function.

The **Kernel** parameter specifies the kernel with which to perform the operation. It can be set to one of the following predefined kernels:

<code>IM_3X3_RECT_0</code>	A 3x3 kernel.
<code>IM_3X3_CROSS_0</code>	A 3x3 kernel whose four corners are set to <code>IM_DONT_CARE</code> .

The following kernels can only be used when **Rank** is set to IM_MEDIAN:

IM_3X3_X	3x3 "X" shaped kernel (5 active elements in the shape of an "X").
IM_3X1	Horizontal one-dimensional kernel with 3 elements.
IM_5X1	Horizontal one-dimensional kernel with 5 elements.
IM_7X1	Horizontal one-dimensional kernel with 7 elements.
IM_9X1	Horizontal one-dimensional kernel with 9 elements.
IM_1X3	Vertical one-dimensional kernel with 3 elements.
IM_1X5	Vertical one-dimensional kernel with 5 elements.
IM_1X7	Vertical one-dimensional kernel with 7 elements.
IM_1X9	Vertical one-dimensional kernel with 9 elements.

Note that the 1-dimensional horizontal and vertical kernels can be used in combination to achieve approximations to larger 2-dimensional kernels. For example, applying a 5x1 median followed by a 1x5 median will give a reasonable approximation to a 5x5 median.

The **Rank** parameter specifies the rank value. This parameter can be set to a value from 1 to the number of valid neighborhood values, or to IM_MEDIAN, which is a median rank value, ($\text{int}((\text{valid neighborhood values} + 1)/2)$). Note that the number of valid neighborhood values corresponds to the number of kernel values that are not set to IM_DONT_CARE. Therefore, for IM_3X3_RECT_0, the median rank value is 5; for IM_3X3_CROSS_0, the median rank value is 3.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntRank()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_OVERSCAN	IM_TRANSPARENT	Process the bordering pixels of the source buffer using the pixels of its parent buffer as the overscan pixels. If the source buffer is not a child buffer or if its parent buffer cannot provide values for the overscan pixels, the overscan pixels will be undefined, leading to unpredictable results.
	IM_REPLACE	Process the bordering pixels of the source buffer by assigning a specific value to the overscan pixels. Specify the value with the IM_CTL_OVERSCAN_VAL field.
IM_CTL_OVERSCAN_VAL	any integer (default: 0)	Overscan replace value (used when IM_CTL_OVERSCAN is set to IM_REPLACE).

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imIntErodeDilate()**. This function can use a custom-made kernel to replace a pixel by the minimum or maximum value in its neighborhood.

imIntRecFilter

Async

PP

In-Place

Multi-band

Synopsis Perform adaptive recursive (temporal) filtering.

Format **void imIntRecFilter(Thread, Src, Src2, Dst, Dst2, Lut, SrcBits, DstBits, Control, OSB)**

long Thread;	Thread ID
long Src;	First source buffer ID
long Src2;	Second source buffer ID
long Dst;	Destination buffer ID
long Dst2;	Second destination buffer ID (or 0)
long Lut;	LUT buffer ID
long SrcBits;	Number of bits in Src buffer
long DstBits;	Number of bits in Dst and Src2 buffers
long Control;	Control buffer ID (or 0)
long OSB;	OSB ID (or 0)

Description This function performs adaptive recursive (temporal) filtering. It determines the difference between two source buffers, assigns a weight to the difference according to the specified look-up table (LUT), then adds the weighted difference to the second source buffer, that is,

$$\mathbf{Dst} = \mathbf{LUT}[\mathbf{Src} - \mathbf{Src2}] + \mathbf{Src2}$$

In the interests of accuracy and efficiency, the actual implementation is closer to:

$$\begin{aligned} a &= \mathbf{LUT}[\mathbf{Src} - \mathbf{Src2}] \\ \mathbf{Dst} &= a(\mathbf{Src} - \mathbf{Src2}) + \mathbf{Src2} \end{aligned}$$

Since the **Dst** and **Src2** buffers usually have more bits than the **Src** buffer (the **DstBits** and **SrcBits** parameters indicate the actual number of bits in these buffers), the exact operations carried out are actually:

1. determine no. of fractional bits: $\text{frac} = \text{DstBits} - \text{SrcBits}$
2. calculate full precision difference: $\text{tmp} = (\text{Src} \ll \text{frac}) - \text{Src2}$
3. lookup “a” from reduced precision difference, where $a = \mathbf{LUT}[\text{tmp} \gg (\text{DstBits} - 8)]$

4. make sure result has same precision as Src2: $\text{Dst} = (\text{a} * \text{tmp} \gg 14) + \text{Src2}$
5. optionally produce an 8-bit version of the result: $\text{Dst2} = (\text{Dst} \gg \text{frac})$

Note that the above uses a limited number of bits to determine “a” but the actual multiplication is performed with full precision.

The first half of the LUT is used to assign a weight to positive differences, and the second half of the LUT is used to assign a weight to negative differences. By using an appropriate LUT, various effects can be achieved. For example, if the first half of the LUT is set to a constant value and the second half is set to 0, then increasing pixel values will have an effect on the output, but decreasing values will have no effect (the previous output will be maintained). This makes bright areas of the image “sticky”.

The **Thread** parameter specifies the thread to which to send **imIntRecFilter()** for execution.

The **Src** and **Src2** parameters specify the buffers on which to perform the operation, while the **Dst** and **Dst2** parameters specify the buffers in which to place the results of the operation. All of these buffers must be unsigned. The allowed pixel depth combinations for these buffers are outlined below.

Src	Src2	Dst	Dst2
8-bit	8-bit	8-bit	—
8-bit	16-bit	16-bit	8-bit
16-bit	16-bit	16-bit	8-bit

If an 8-bit result is not needed, **Dst2** should be set to 0. Note that **Dst2** should be set to 0 when **Dst** is 8-bit, since it is not needed in this case.

The **Lut** parameter specifies the look-up table buffer. The LUT buffer must have 512 entries of 16 bits each. The first half of the LUT is used to assign a weight to positive differences, and the second half of the LUT is used to assign a weight to negative differences. Each weight must be in the range 0.0 to 1.0 (inclusive) with 14 fractional bits (hence 1.0 is encoded as 0x4000, 0.5 is encoded as 0x2000, etc.). Note that, if **Dst** and **Src2** have more than 8 bits, there are not enough LUT entries for all possible differences. In this case, each entry is used for a range of differences. For example, if

DstBits = 12, there are $2^{12} * 2 = 8192$ possible differences, so each entry is used for $8192/512 = 16$ differences, that is, LUT[0] is used for differences of 0 to 15, LUT[1] for differences of 16 to 31, etc.

The **SrcBits** parameter specifies the actual number of bits in the **Src** buffer (note, for example, that **Src** can be a 16-bit buffer but contain only 10-bit data). The allowed number of bits is 8 to 15.

The **DstBits** parameter specifies the actual number of bits in the **Src2** and **Dst** buffers. The allowed number of bits is 8 to 15, and must not be less than **SrcBits**.

The **Control** parameter specifies the control buffer with which to perform the function. Currently, there are no fields defined for this function. You can set this parameter to 0.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imIntMac2()**. If all of the LUT is set to a constant value, the effect is simple (non-adaptive) recursive filtering, which might be performed faster using **imIntMac2()**.

imIntScale

Async

PP

Multi-band

Synopsis Scale an image by integer or non-integer factors.

Format **void imIntScale(Thread, SrcBuf, DstBuf, XFac, YFac, Control, OSB)**

long Thread;	Thread ID
long SrcBuf;	Source buffer ID
long DstBuf;	Destination buffer ID
double XFac;	Horizontal scale factor
double YFac;	Vertical scale factor
long Control;	Control buffer ID (or 0)
long OSB;	OSB ID (or 0)

Description This function scales an image by an integer or non-integer factor. This function supports an interpolated and a non-interpolated mode. In interpolated mode, an approximation to the specified scale factor might be used; however, in non-interpolated mode, the exact scale factor that is specified is used.

imIntScale() is the fastest way to perform an interpolated scaling by an integer factor (particularly for factors of 2, 4, and 8). Note that if you specify a non-integer factor in interpolated mode, the actual scale factor used might be slightly different, in the interest of speed. However, integer factors will be exactly as specified, as will those that can be expressed as the ratio of two small integers (such as $1.5 = 3/2$).

The **Thread** parameter specifies the thread to which to send **imIntScale()** for execution.

The **SrcBuf** parameter specifies the buffer to scale. This buffer can be an 8-bit or 16-bit unsigned integer buffer.

The **DstBuf** parameter specifies the buffer in which to place the results of the operation. This buffer must be of the same type as the source buffer, or have a smaller pixel size.

The **XFac** and **YFac** parameters specify the horizontal and vertical scale factors, respectively. These can be set to a value less than or greater than 1, or to IM_FILL. In the latter case, the scale factor is automatically chosen such that the re-scaled image will just fill the destination buffer, in the horizontal and/or vertical dimension.

In interpolated mode, the scaling factor is used exactly as specified when:

scaling factor (that is, **XFac** and/or **YFac**) = n/m ,

where n and m are integers between 1 and 16 (inclusive).

In other cases, the scale factor used might be slightly different from the one requested.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntScale()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_RESAMPLE	IM_NO_INTERPOLATE	Don't interpolate.
	IM_INTERPOLATE	Perform bilinear interpolation.
IM_CTL_SHIFT	1 - 8	Right shift by the specified number of bits. This field applies only when SrcBuf is 16-bit and DstBuf is 8-bit.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imIntSubsample()**, **imIntWarpPolynomial()**, **imIntZoom()**.

imIntSubsample() and **imIntZoom()** are the fastest way to scale an image by an integer factor without interpolation.

imIntWarpPolynomial() is the slowest way to scale an image, but has the fewest restrictions. **imIntScale()** can be used as a faster alternative to **imIntWarpPolynomial()** when you only require scaling (without interpolation).

imIntSubsample

Async

PP

NOA

Multi-band

Synopsis Subsample an image.

Format void imIntSubsample(Thread, Src, Dst, Xfac, Yfac, Control, OSB)

long Thread; Thread ID
long Src; Source buffer ID
long Dst; Destination buffer ID
long Xfac; Horizontal subsampling factor
long Yfac; Vertical subsampling factor
long Control; Control buffer ID (or 0)
long OSB; OSB ID (or 0)

Description This function subsamples an integer image. It divides the image into **Xfac** x **Yfac** blocks, and then copies a value from each block into the appropriate pixel of the destination buffer. This value can be the value of a pixel at a specified position in the block, or the average of values in that block.

The **Thread** parameter specifies the thread to which to send **imIntSubsample()** for execution.

The **Src** parameter specifies the buffer to subsample. If not interpolating (that is, if not taking the average of values in a block), this buffer can be of any integer type, or can be a 32-bit floating-point buffer. If interpolating, this buffer must be either 8-bit unsigned integer or 16-bit signed integer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This buffer must be of the same type as the source buffer. Note that in-place operation is not supported for this function.

The **Xfac** and **Yfac** parameters specify the horizontal and vertical subsampling factors, respectively. These can be set to an integer value between 1 and 16.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntSubsample()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_RESAMPLE	IM_NO_INTERPOLATE	Use the pixel at the specified position within the block (the position is specified through the IM_CTL_SAMPLE_X and IM_CTL_SAMPLE_Y fields).
	IM_INTERPOLATE	Use the average value of each block.
	IM_MAX	Use the maximum value of each block.
IM_CTL_SAMPLE_X	0 – (Xfac-1) default: int(Xfac-1)/2	X coordinate of pixel to use (if not interpolating).
IM_CTL_SAMPLE_Y	0 – (Yfac-1) default: int(Yfac-1)/2	Y coordinate of pixel to use (if not interpolating).

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imIntScale()**, **imIntWarpPolynomial()**, **imIntZoom()**. **imIntScale()** is the fastest way to subsample or zoom an image by an integer factor with interpolation. **imIntWarpPolynomial()** is the slowest way to subsample an image but has the fewest restrictions. **imIntZoom()** can zoom an image.

imIntThickThin

Async

PP

Multi-band

Synopsis Perform grayscale thinning or thickening.

Format **void imIntThickThin(Thread, Src, Dst, Kernel, Op, Niter, Control, OSB)**

long Thread;	Thread ID
long Src;	Source buffer ID
long Dst;	Destination buffer ID
long Kernel;	Kernel buffer ID
long Op;	Type of operation to perform
long Niter;	Number of iterations
long Control;	Control buffer ID (or 0)
long OSB;	OSB ID (or 0)

Description This function performs a grayscale thinning or thickening on an integer image, using a specified structuring element (kernel). Kernel values other than 0 or 1 are ignored during the operation.

If you want to thin or thicken with a series of different kernels (one applied after the other), you can use a multi-band kernel. Each band of the kernel will be applied to the result of the previous one.

For information on the thinning and thickening algorithms, see the *Genesis Native Library User Guide*.

Note that **imIntThickThin()** checks if any pixel values have changed during the last iteration of the operation. If any have changed, the **IM_RES_IDEMPOTENCE** field of the destination buffer is set to a non-zero value (TRUE); otherwise, this field is set to 0 (FALSE). This field can later be read using **imBufGetField()**.

The **Thread** parameter specifies the thread to which to send **imIntThickThin()** for execution.

The **Src** parameter specifies the buffer on which to perform the operation. This must be an 8-bit or 16-bit integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This must be of the same type as the source buffer. Note that in-place operation is not supported for this function.

The **Kernel** parameter specifies the kernel buffer with which to perform the operation. This must be a 3x3 buffer of any data type.

The **Op** parameter specifies the type of operation to perform. It can be set to:

- IM_THICK Thickening operation.
- IM_THIN Thinning operation.

The **Niter** parameter specifies the number of times to apply the operation. Note that, for a multi-band kernel, the number of iterations is the number of passes through the entire kernel. This parameter can be set to IM_IDEMPOTENCE, which will cause the function to iterate until no more changes are produced (for a thinning operation, this will typically occur when the image has been reduced to its skeleton).

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntThickThin()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_OVERSCAN	IM_TRANSPARENT	Process the bordering pixels of the source buffer using the pixels of its parent buffer as the overscan pixels. If the source buffer is not a child buffer or if its parent buffer can't provide values for the overscan pixels, the overscan pixels will be undefined, leading to unpredictable results.
	IM_REPLACE	Process the bordering pixels of the source buffer by assigning a specific value to the overscan pixels. Specify the value with the IM_CTL_OVERSCAN_VAL field.

IM_CTL_OVERSCAN_VAL	any integer (default is 0)	Overscan replace value (used when IM_CTL_OVERSCAN is set to IM_REPLACE).
---------------------	-------------------------------	---

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imBinMorphic()**, **imIntErodeDilate()**. The **imBinMorphic()** function can thin or thicken a binary image. The **imIntErodeDilate()** function can perform grayscale erosion or dilation on an integer image.

imIntTriadic

Async

PP

In-Place

Multi-band

Synopsis Perform an arithmetic or logical operation with three operands.

Format **void imIntTriadic(Thread, SrcA, SrcB, SrcC, Dst, Rotate, Op, Mode, OSB)**

long Thread; Thread ID
 long SrcA; First source operand (buffer ID or constant)
 long SrcB; Second source operand (buffer ID)
 long SrcC; Third source operand (buffer ID or constant)
 long Dst; Destination buffer ID
 long Rotate; Amount to rotate **SrcB** operand
 long Op; Type of operation to perform
 long Mode; Operation mode
 long OSB; OSB ID (or 0)

Description This function performs an arithmetic or logical operation between three operands.

The **SrcB** operand must be an integer image; the other two operands can be integer images or constants. In all cases, each non-constant input is sign-extended to 32 bits and the **SrcB** input is rotated by a specified number of bits before the operation is performed. If the destination buffer of the operation has fewer than 32 bits, the high bits of the result are discarded.

The PP ALU opcode specifies the type of operation to perform. Predefined opcodes exist for common operations but if one does not exist for the operation you want to perform, you can derive its opcode; see the *Genesis Native Library User Guide* for details.

The **Thread** parameter specifies the thread to which to send **imIntTriadic()** for execution.

The **SrcA** parameter specifies the first operand. This can be the identifier of an image buffer or a specified constant. This operand can be of any integer type.

The **SrcB** parameter specifies the second operand. This must be the identifier of an image buffer. This operand can be of any integer type.

The **SrcC** parameter specifies the third operand. This can be the identifier of an image buffer or a specified constant. This operand can be of any integer type.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This buffer can be of any integer type.

The **Rotate** parameter specifies the number of bits by which to rotate the **SrcB** image buffer. This parameter can be set to an integer value between -31 and 31. A positive value indicates a rotation to the left; a negative value indicates a rotation to the right. Note that a rotation to the right causes the least-significant bits to wrap around to the high bits of a pixel value; a rotation to the left causes the high bits to wrap around to the least-significant bits of a pixel value.

The **Op** parameter specifies the type of operation to perform. It can be set to the required PP ALU opcode (that is, to the 32-bit part of the opcode that resides in register d0 for the instruction class EALU || ROTATE) or to a predefined opcode. The predefined opcodes are listed below.

IM_PP_PASS_A	pass SrcA
IM_PP_PASS_B	pass SrcB
IM_PP_PASS_C	pass SrcC
IM_PP_PASS	pass SrcA (same as IM_PP_PASS_A)
IM_PP_NOT	\sim SrcA
IM_PP_AND	SrcA & SrcB
IM_PP_OR	SrcA SrcB
IM_PP_XOR	SrcA ^ SrcB
IM_PP_XOR_XOR	SrcA ^ SrcB ^ SrcC
IM_PP_NAND	\sim (SrcA & SrcB)
IM_PP_NOR	\sim (SrcA SrcB)
IM_PP_XNOR	\sim (SrcA ^ SrcB)
IM_PP_ADD_ADD	SrcA + SrcB + SrcC (Only SrcC can be constant)
IM_PP_ADD_SUB	SrcA + SrcB - SrcC (Only SrcC can be constant)
IM_PP_SUB_SUB	SrcA - SrcB - SrcC (Only SrcC can be constant)
IM_PP_SRA_ADD	SrcA + (SrcB & SrcC) i.e. shift right arithmetic and add (sign bit is propagated as SrcB is rotated right)
IM_PP_SRA_SUB	SrcA - (SrcB & SrcC) i.e. shift right arithmetic and subtract (sign bit is propagated as SrcB is rotated right)

IM_PP_ADD_ABS	SrcA + SrcB (SrcC must be constant 0)
IM_PP_SUB_ABS	SrcA - SrcB (SrcC must be constant 0)
IM_PP_ADD_AND	SrcA + (SrcB & SrcC)
IM_PP_SUB_AND	SrcA - (SrcB & SrcC)
IM_PP_ADD_OR	SrcA + (SrcB SrcC)
IM_PP_SUB_OR	SrcA - (SrcB SrcC)
IM_PP_EXT_FIELD	SrcB & SrcC
IM_PP_INS_FIELD	(SrcA & SrcC) (SrcB & ~ SrcC)
IM_PP_ADD_FIELD	(SrcA & SrcC) + (SrcB & SrcC)
IM_PP_MERGE	(SrcA & SrcC) (SrcB & ~ SrcC)

Note that some operations require that you pass particular constants for certain operands. For example, a bit-shift requires a rotate followed by a boolean operation to mask out the unwanted bits. You must supply the mask by passing the appropriate constant as the **SrcB** operand (see the examples).

The **Mode** parameter specifies the mode of operation. It can be set to:

IM_DEFAULT	All operands are images.
IM_CONSTANT_A	SrcA is a constant.
IM_CONSTANT_C	SrcC is a constant.
IM_CONSTANT_AC	SrcA and SrcC are constants.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Example The following code performs the operation $A + (B \gg 4)$. Note that the **SrcC** operand is used to clear the unwanted bits (four in this case) that were rotated from the least-significant bits into the most-significant bits.

```
imIntTriadic(Thr, A, B, 0x0fffffff, Dst, -4, IM_PP_ADD_AND, IM_CONSTANT_C, 0);
```

The following code merges A and B, using C as a pixel mask (i.e. passes A where C is 0xff and B where C is 0). For more information on merging images, see *process.c* in Appendix B.

```
imIntTriadic(Thr, A, B, C, Dst, 0, IM_PP_MERGE, IM_DEFAULT, 0);
```

The following code extracts bits 10-14 from B and right-justifies them (using C to mask out the unwanted bits).

```
imIntTriadic(Thr, 0, B, 0x1f, Dst, -10, IM_PP_EXT_FIELD, IM_CONSTANT_AC, 0);
```

See also **imBinTriadic()**, **imIntDyadic()**, **imIntMonadic()**. The **imBinTriadic()** function can perform a logical operation on up to three binary images. The **imIntDyadic()** and **imIntMonadic()** functions perform an arithmetic or logical operation on integer images.

imIntWarpLut

Async

PP

Multi-band

Synopsis Perform a warping using a look-up table inverse address calculation.

Format **void imIntWarpLut(Thread, Src, Dst, Xlut, Ylut, Control, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Xlut; X-LUT buffer ID
 long Ylut; Y-LUT buffer ID
 long Control; Control buffer ID (or 0)
 long OSB; OSB ID (or 0)

Description This function warps an integer image using a look-up table (LUT) address mapping. Each pixel position of the destination buffer, (x_d, y_d) , gets associated with a specific point (address) in the source buffer, (x_s, y_s) ; x_s is determined from (x_d, y_d) through one LUT, and y_s is determined from (x_d, y_d) through another LUT. The pixel value of (x_d, y_d) is then determined from its associated point and from a specified interpolation mode.

The LUTs with which to perform the warping can be user-supplied or, for 3x3 matrix-defined warpings, automatically generated using **imGenWarpLutMatrix()**. In general, x_s and y_s are non-integer values and the LUTs contain fixed-point values with the precision specified in the control buffer. The points x_s and y_s are interpreted as signed numbers so that transparent overscan can be supported (note, for example, that negative y_s values correspond to pixels above the top line of the source buffer).

The **Thread** parameter specifies the thread to which to send **imIntWarpLut()** for execution.

The **Src** parameter specifies the buffer on which to perform the warping. This can be of any integer type if not interpolating. If interpolating, this must be an 8-bit or 16-bit integer buffer.

The **Dst** parameter specifies the buffer in which to place the results of the warping. This buffer must be of the same type as the source buffer. Note that in-place operation is not supported for this function.

The **Xlut** parameter specifies the two-dimensional LUT buffer from which x_s is determined. This must be a signed, 16-bit integer buffer and of the same size as the destination buffer.

The **Ylut** parameter specifies the two-dimensional LUT buffer from which y_s is determined. This must be a signed, 16-bit integer buffer and of the same size as the destination buffer.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntWarpLut()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_PRECISION	0 - n	Number of fractional bits in the LUT buffers for x_s and y_s . This field only has an effect if you are interpolating (that is, if IM_CTL_RESAMPLE is set to IM_BILINEAR).
IM_CTL_RESAMPLE	IM_NEAREST_NEIGHBOR	Don't interpolate. The points x_s and y_s are interpreted as integers.
	IM_BILINEAR	Perform bilinear interpolation.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note Various overscan modes can be implemented simply by using appropriate address values. For transparent overscan, use values which address pixels outside the source buffer (if the source buffer is a child buffer). For replace overscan, use a constant address which points to a source pixel with a known value.

Example See *process.c* in Appendix B.

imIntWarpPolynomial

Async

PP

Multi-band

Synopsis Perform a warping using polynomial inverse address calculation.

Format **void imIntWarpPolynomial(Thread, Src, Dst, Coef, Control, OSB)**

long Thread; Thread ID
 long Src; Source buffer ID
 long Dst; Destination buffer ID
 long Coef; Warp coefficient buffer ID
 long Control; Control buffer ID (or 0)
 long OSB; OSB ID (or 0)

Description This function warps an integer image using a first-order polynomial mapping. Each pixel position of the destination buffer, (x_d, y_d) , gets associated with a specific point in the source buffer, (x_s, y_s) using the following equations:

$$x_s = a_0 + a_1 x_d + a_2 y_d$$

$$y_s = b_0 + b_1 x_d + b_2 y_d$$

The pixel value of (x_d, y_d) is then determined from its associated point and from a specified interpolation mode.

Note that a first-order warping is equivalent to linearly translating, scaling, rotating, and shearing the source image. First-order warp coefficients can be generated using **imGenWarp1stOrder()**.

The **Thread** parameter specifies the thread to which to send **imIntWarpPolynomial()** for execution.

The **Src** parameter specifies the buffer on which to perform the operation. This can be of any integer type if not interpolating. If interpolating, this must be an 8-bit or 16-bit integer buffer.

The **Dst** parameter specifies the buffer in which to write the results of the operation. This buffer must be of the same type as the source buffer. Note that in-place operation is not supported for this function.

The **Coef** parameter specifies the buffer containing the required coefficients. This buffer must be a single-band 32-bit floating-point buffer, of size 3x2 or 3x3. If the source buffer has more than one band, the same coefficients are used for each band.

If the **Coef** buffer is 3x2, the first row specifies the a_n coefficients and the second row specifies the b_n coefficients.

If the **Coef** buffer is 3x3, the order of coefficients is the same as that expected by **imGenWarpLutMatrix()**. Note that the third row of coefficients is ignored (they are assumed to be 0, 0, 1) since **imIntWarpPolynomial()** only performs first-order warpings.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntWarpPolynomial()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_RESAMPLE	IM_NEAREST_NEIGHBOR	Don't interpolate.
	IM_BILINEAR	Perform bilinear interpolation.
	IM_BICUBIC	Perform bicubic interpolation.
IM_CTL_OVERSCAN	IM_TRANSPARENT	When (x_s, y_s) falls outside the source buffer, use pixels of its parent buffer as the overscan pixels. If the source buffer is not a child buffer or if its parent buffer can't provide values for the overscan pixels, the overscan pixels will be undefined, leading to unpredictable results.
	IM_REPLACE	When (x_s, y_s) falls outside the source buffer, set the overscan pixels to 0.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note Every pixel of the destination buffer is always written, regardless of the size of the source buffer. If your transformation requires source pixels outside the source buffer, you should use replace overscan mode.

Example See *process.c* in Appendix B.

See also **imIntFlip()**, **imIntScale()**, **imIntSubsample()**, **imIntZoom()**. If the required warping can be performed using one of these functions, your application will run faster.

imIntZoom

Async

PP

NOA

Multi-band

Synopsis Zoom an image.

Format `void imIntZoom(Thread, Src, Dst, Xfac, Yfac, Control, OSB)`

<code>long Thread;</code>	Thread ID
<code>long Src;</code>	Source buffer ID
<code>long Dst;</code>	Destination buffer ID
<code>long Xfac;</code>	Horizontal zooming factor
<code>long Yfac;</code>	Vertical zooming factor
<code>long Control;</code>	Control buffer ID (or 0)
<code>long OSB;</code>	OSB ID (or 0)

Description This function zooms an integer image. It replicates each pixel into an **Xfac** x **Yfac** block of pixels, and then copies this block to the appropriate area of the destination buffer. It can then apply an averaging filter (of size **Xfac** x **Yfac**) to the destination buffer, thereby producing an interpolated zoom.

The **Thread** parameter specifies the thread to which to send **imIntZoom()** for execution.

The **Src** parameter specifies the buffer to zoom. If not interpolating, this buffer can be of any integer type or can be a 32-bit floating-point buffer. If interpolating, this buffer can be 8-bit unsigned integer or 16-bit signed integer.

The **Dst** parameter specifies the buffer in which to place the results of the operation. This buffer must be of the same type as the source buffer. If this buffer is too small, the result is clipped to fit. Note that in-place operation is not supported for this function.

The **Xfac** and **Yfac** parameters specify the horizontal and vertical zooming factors, respectively. These can be set to an integer value between 1 and 16.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imIntZoom()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Values	Meaning
IM_CTL_RESAMPLE	IM_NO_INTERPOLATE	Don't filter the destination buffer.
	IM_INTERPOLATE	Filter the destination buffer.
IM_CTL_CENTER_X	0 – (Xfac-1) default: int(Xfac-1)/2	X coordinate of filter center.
IM_CTL_CENTER_Y	0 – (Yfac-1) default: int(Yfac-1)/2	Y coordinate of filter center.
IM_CTL_OVERSCAN (this field only applies if the destination buffer is to be filtered)	IM_TRANSPARENT	Filter the bordering pixels of the source buffer using the pixels of its parent buffer as the overscan pixels. If the source buffer is not a child buffer or if its parent buffer can't provide values for the overscan pixels, the overscan pixels will be undefined, leading to unpredictable results.
	IM_REPLACE	Filter the bordering pixels of the source buffer by setting the overscan pixels to 0.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imIntScale()**, **imIntSubsample()**, **imIntWarpPolynomial()**.

imIntScale() is the fastest way to zoom or subsample an image by an integer factor with interpolation. **imIntWarpPolynomial()** is the slowest way to zoom an image but has the fewest restrictions. **imIntSubsample()** can subsample an image.

imJpegAlloc

[Sync](#)

Synopsis Allocate a JPEG buffer.

Format **void imJpegAlloc(Thread, Control, JpegPtr)**

long Thread; Thread ID
 long Control; Control buffer ID (or 0)
 long* JpegPtr; Address of JPEG buffer ID

Description This function allocates a JPEG buffer. JPEG buffers are used to hold JPEG compressed images.

Note that the control settings of a JPEG buffer affect how an image is compressed into that JPEG buffer (using **imJpegEncode()**), or how that JPEG buffer is decompressed (using **imJpegDecode()**). When a JPEG buffer is allocated, these control settings are set to default values. For a list of control settings and their default values, see **imJpegControl()**.

The **Thread** parameter specifies the thread to which to send **imJpegAlloc()** for execution.

The **Control** parameter specifies the control buffer with which to perform the function. This parameter must be set to 0.

The **JpegPtr** parameter specifies the address in which to return the identifier of the JPEG buffer. If the JPEG buffer could not be allocated, 0 is returned.

imJpegControl

Async

Synopsis Change a control setting of a JPEG buffer.

Format `void imJpegControl(Thread, Jpeg, Item, Value)`

long Thread; Thread ID
long Jpeg; JPEG buffer ID
long Item; Item to set
double Value; Value for Item

Description This function changes a control setting of a JPEG buffer. Among other things, these settings affect how an image is compressed into that JPEG buffer (using **imJpegEncode()**), or how that JPEG buffer is decompressed (using **imJpegDecode()**).

The **Thread** parameter specifies the thread to which to send **imJpegControl()** for execution.

The **Jpeg** parameter specifies the JPEG buffer.

The **Item** parameter specifies the setting to change, while the **Value** parameter specifies the value for this setting. The table below lists available settings, and their allowable values. The default values of settings are in bold-face.

Item	Values	Meaning
IM_JPEG_MODE	IM_BASELINE	Perform lossy compression (baseline sequential mode).
	IM_LOSSLESS	Perform lossless compression.
IM_JPEG_PREDICTOR	1	Use predictor #1 for lossless compression modes.
	0	Don't use a predictor for lossless compression modes.
IM_JPEG_Q_FACTOR	1 - 99 (default: 50)	Quality factor for lossy compression modes. The higher the factor, the more the compression, but the lower the image quality.

IM_JPEG_RESTART_ROWS	any integer (default: 32)	Place restart markers after every n rows of data (for lossless compressions) or after every n 8x8 blocks of data (for lossy compressions).
IM_JPEG_SAVE_TABLES	IM_ENABLE	Save the compression tables, as well as the compressed image, when writing to a file.
	IM_DISABLE	Don't save the compression tables.
IM_JPEG_SAVE_IMAGE	IM_ENABLE	Include the compressed image when writing to a file.
	IM_DISABLE	Don't include the compressed image. (This allows a table-only file to be created.)
IM_JPEG_SIZE_BIT	9 - 16 (default value depends on source buffer size)	Actual number of bits per pixel in the source buffer. This setting only applies to lossless compressions, since the source buffer can be 16-bit but only contain, for example, 10- or 12-bit data. (For lossy compressions, the source buffer must always be 8-bit).
IM_JPEG_NUM_BLOCKS	any integer (default: 1)	Divide an image (internally) into the specified number of blocks, then compress it block by block. There are some restrictions on the size of each block; for details, see the <i>Genesis Native Library User Guide</i> .
IM_JPEG_RESET (the Value parameter for this item is ignored; any value can be given for it)		Free all memory associated with the JPEG buffer. This also stops a block-by-block compression before all blocks are compressed.

The following settings affect all bands of a multi-band image that is to be compressed. If necessary, you can set different values for each band, using **imJpegControlBand()**.

Item	Values	Meaning
IM_JPEG_TABLE_QUANT	0 - 3	Quantization table to use. This table is only used for lossy compression modes.
IM_JPEG_TABLE_AC	0 - 3	AC Huffman table to use. This table is only used for lossy compression modes.
IM_JPEG_TABLE_DC	0 - 3	DC Huffman table to use. This table is used for lossy and lossless compression modes.

Example See *jpeg.c* in Appendix B.

imJpegControlBand

Async

Synopsis Change a control setting of a JPEG buffer, for a specific band.

Format **imJpegControlBand(Thread, Jpeg, Band, Item, Value)**

long Thread; Thread ID
 long Jpeg; JPEG buffer ID
 long Band; Band number
 long Item; Item to set
 double Value; Value for Item

Description This function changes a control setting of a JPEG buffer, for a specific band. This function is useful if you are compressing a multi-band image and want one or more control settings to be different for each band.

The **Thread** parameter specifies the thread to which to send **imJpegControlBand()** for execution.

The **Jpeg** parameter specifies the JPEG buffer.

The **Band** parameter specifies the band. This parameter must be set to the number of the required band; the valid range is 0 to 3.

The **Item** parameter specifies the setting to change, while the **Value** parameter specifies the value for this setting. The table below lists settings that can be changed for each band, and their allowable values.

Item	Values	Meaning
IM_JPEG_TABLE_QUANT	0 - 3	Quantization table to use. This table is only used for lossy compression modes.
IM_JPEG_TABLE_AC	0 - 3	AC Huffman table to use. This table is only used for lossy compression modes.
IM_JPEG_TABLE_DC	0 - 3	DC Huffman table to use. This table is used for lossy and lossless compression modes.

imJpegDecode

Async

PP

NOA

Synopsis Decompress a compressed image.

Format **void imJpegDecode(Thread, Buf, Jpeg, OSB)**

long Thread; Thread ID
long Buf; Destination buffer ID
long Jpeg; JPEG buffer ID
long OSB; OSB ID (or 0)

Description This function decompresses a JPEG compressed image.

Note that the control settings of the JPEG buffer affects how the image is decompressed. These control settings should not be changed before decompressing because, for the reconstructed image to match the original image, the same controls must be used to compress and decompress.

The **Thread** parameter specifies the thread to which to send **imJpegDecode()** for execution.

The **Buf** parameter specifies the buffer in which to place the decompressed image. This buffer must be of the appropriate size and type to hold the decompressed image.

The **Jpeg** parameter specifies the JPEG buffer containing the compressed image.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note The NOA is only used for lossless compression modes.

Example See *jpeg.c* in Appendix B.

imJpegEncode

Async

PP

NOA

Synopsis Compress an image.

Format **void imJpegEncode(Thread, Buf, Jpeg, OSB)**

long Thread; Thread ID
 long Buf; Source buffer ID
 long Jpeg; JPEG buffer ID
 long OSB; OSB ID (or 0)

Description This function compresses an image.

Note that the control settings of the JPEG buffer affect how the image is compressed. You can change these settings using **imJpegControl()**, **imJpegControlBand()**, and/or **imJpegPutTable()**.

The **Thread** parameter specifies the thread to which to send **imJpegEncode()** for execution.

The **Buf** parameter specifies the buffer to compress. For a lossless compression, this can be an 8-bit or 16-bit integer buffer, with 1 to 4 bands. For a lossy compression, this must be an 8-bit integer buffer, with 1 to 4 bands.

The **Jpeg** parameter specifies the JPEG buffer in which to place the compressed image.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note The NOA is only used for lossless compression modes.

Example See *jpeg.c* in Appendix B.

imJpegFree

Async

Synopsis Free a JPEG buffer.

Format **void imJpegFree(Thread, Jpeg)**

long Thread; Thread ID

long Jpeg; Jpeg buffer ID

Description This function deallocates a previously allocated JPEG buffer.

The **Thread** parameter specifies the thread to which to send **imJpegFree()** for execution.

The **Jpeg** parameter specifies the JPEG buffer to deallocate.

imJpegGetTable

[Sync](#)

Synopsis Transfer a JPEG table to Host memory.

Format **void imJpegGetTable(Thread, Jpeg, TableType, TableNum, TableSizePtr, TablePtr)**

long Thread;	Thread ID
long Jpeg;	JPEG buffer ID
long TableType;	Table type
long TableNum;	Table number
long* TableSizePtr;	Address in which to return table size
void* TablePtr;	Address of array

Description This function transfers a specified table of a JPEG buffer to an array in Host memory.

The **Thread** parameter specifies the thread to which to send **imJpegGetTable()** for execution.

The **Jpeg** parameter specifies the JPEG buffer containing the required table.

The **TableType** parameter specifies the type of table to transfer. It can be set to:

IM_JPEG_TABLE_QUANT	Quantization table.
IM_JPEG_TABLE_AC	AC Huffman table.
IM_JPEG_TABLE_DC	DC Huffman table.

The **TableNum** parameter specifies the number of the specified table type. It can be set to a number between 0 and 3.

The **TableSizePtr** parameter specifies the address in which to return the size of the transferred table (in bytes).

The **TablePtr** parameter specifies the address of the array in which to place the table.

imJpegInquire

Sync

Synopsis Inquire about a JPEG buffer.

Format **long imJpegInquire(Thread, Jpeg, Item, ValuePtr)**

long Thread;	Thread ID
long Jpeg;	JPEG buffer ID
long Item;	Attribute about which to inquire
void* ValuePtr;	Address of return value (or NULL)

Description This function inquires about an attribute of a specified JPEG buffer.

Note that certain attributes describe the original image that was compressed into the JPEG buffer. Therefore, if you are decompressing an image and do not know the buffer attributes of the original image, you can use **imJpegInquire()** to allocate an appropriate buffer (by passing the return values of these attributes to **imBufAlloc()**).

The **Thread** parameter specifies the thread to which to send **imJpegInquire()** for execution.

The **Jpeg** parameter specifies the JPEG buffer.

The **Item** parameter specifies the attribute about which to inquire. It can be set to:

IM_JPEG_MODE	Mode of operation: IM_BASELINE or IM_LOSSLESS.
IM_JPEG_PREDICTOR	Predictor for lossless compression modes.
IM_JPEG_Q_FACTOR	Quality factor for lossy compression modes.
IM_JPEG_RESTART_ROWS	Number of rows after which restart markers are placed.
IM_JPEG_SAVE_TABLES	Whether tables are saved to the JPEG buffer (non-zero i.e. TRUE) or not saved (zero i.e. FALSE).
IM_JPEG_SAVE_IMAGE	Whether the image is saved to the JPEG buffer (non-zero i.e. TRUE) or not saved (zero i.e. FALSE).
IM_JPEG_TABLE_QUANT	Quantization table used.
IM_JPEG_TABLE_AC	AC Huffman table used.
IM_JPEG_TABLE_DC	DC Huffman table used.

The following attributes can only be inquired if the JPEG buffer contains a compressed image (that is, if the JPEG buffer was used in a call to **imJpegEncode()**, **imJpegRead()**, or **imJpegReadBuf()**, or was created by **imJpegRestore()**).

IM_JPEG_SIZE	Size of compressed image (in bytes).
IM_JPEG_SIZE_X	Width of the original image (can be passed to imBufAlloc()).
IM_JPEG_SIZE_Y	Height of the original image (can be passed to imBufAlloc()).
IM_JPEG_SIZE_BIT	Number of bits per pixel in the original image.
IM_JPEG_NUM_BANDS	Number of bands in the original image (can be passed to imBufAlloc()).
IM_JPEG_TYPE	Data type of the original image (can be passed to imBufAlloc()).

The **ValuePtr** parameter specifies the address in which to return the value of the inquired attribute. Unless otherwise stated, **ValuePtr** should be the address of a long. Note that, since **imJpegInquire()** also returns this value, **ValuePtr** can be set to NULL.

Return value The returned value is the value of the inquired attribute, cast to long if necessary.

Example See *jpeg.c* in Appendix B.

imJpegPutTable

Sync

Synopsis Transfer a table from Host memory to a JPEG buffer.

Format **void imJpegPutTable(Thread, Jpeg, TableType, TableNum, TableSize, TablePtr)**

long Thread;	Thread ID
long Jpeg;	JPEG buffer ID
long TableType;	Table type
long TableNum;	Table number
long TableSize;	Table size
void* TablePtr;	Address of array

Description This function transfers a table from an array in Host memory to a JPEG buffer.

The **Thread** parameter specifies the thread to which to send **imJpegPutTable()** for execution.

The **Jpeg** parameter specifies the JPEG buffer into which to transfer the table.

The **TableType** parameter specifies the type of table being transferred. It can be set to:

IM_JPEG_TABLE_QUANT	Quantization table.
IM_JPEG_TABLE_AC	AC Huffman table.
IM_JPEG_TABLE_DC	DC Huffman table.

The **TableNum** parameter specifies the number of the specified table type to update. It can be set to a number between 0 and 3.

The **TableSize** parameter specifies the size (in bytes) of the table being transferred.

The **TablePtr** parameter specifies the address of the array containing the table to transfer. The array should be a byte array with 64 values for a quantization table, and a short array for a Huffman table.

imJpegRead

Sync

Synopsis Read a compressed image from an open file.

Format **void imJpegRead(Thread, FileHandle, Jpeg)**

long Thread;	Thread ID
FILE* FileHandle;	Handle of open file
long Jpeg;	JPEG buffer ID

Description This function reads a compressed image from an open file to an existing JPEG buffer.

Note that tables and other controls in the file overwrite the corresponding controls in the JPEG buffer. If corresponding controls are not found in the file, the current controls in the JPEG buffer are used.

The **Thread** parameter specifies the thread to which to send **imJpegRead()** for execution.

The **FileHandle** parameter specifies the handle of the open file (opened with **fopen()**). Before calling this function, the file must be positioned just before the start of the compressed image. After the function call, the file remains open and is positioned immediately after the compressed image.

The **Jpeg** parameter specifies the JPEG buffer in which to store the compressed image.

See also **imJpegReadBuf()**, **imJpegRestore()**. The **imJpegReadBuf()** function reads a compressed image from an ordinary (contiguous) buffer. The **imJpegRestore()** function loads a compressed image from a file into an automatically allocated JPEG buffer.

imJpegReadBuf

Async

Synopsis Transfer a JPEG compressed image from an ordinary (contiguous) buffer to a JPEG buffer.

Format **void imJpegReadBuf(Thread, Buf, Jpeg, Start, OSB)**

long Thread;	Thread ID
long Buf;	Source buffer ID
long Jpeg;	JPEG buffer ID
long Start;	Byte offset
long OSB;	OSB ID (or 0)

Description This function transfers a JPEG compressed image from an ordinary (contiguous) buffer to an existing JPEG buffer. This operation will permit you to use other JPEG functions, which require that image data be in a JPEG buffer.

Note that tables and other control fields in the ordinary buffer overwrite the corresponding control fields in the JPEG buffer. If corresponding control fields are not found in the ordinary buffer, the current control fields in the JPEG buffer are used.

The **Thread** parameter specifies the thread to which to send **imJpegReadBuf()** for execution.

The **Buf** parameter specifies the ordinary (contiguous) buffer containing the compressed image.

The **Jpeg** parameter specifies the JPEG buffer in which to store the compressed image.

The **Start** parameter specifies the byte offset, from the beginning of the source buffer, at which the compressed image is stored.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imJpegRead()**, **imJpegRestore()**. These functions can transfer compressed images from files into JPEG buffers directly.

Note It is recommended to copy ordinary Host buffers to ordinary processing buffers before calling **imJpegReadBuf()** because it is much slower to read from a Host buffer than from a buffer in processing memory. This is particularly useful for JPEG lossless compression when the compressed image is relatively large.

imJpegRestore

Sync

Synopsis Load a compressed image from a file into an automatically allocated JPEG buffer.

Format **void imJpegRestore(Thread, FileName, JpegPtr)**

long Thread;	Thread ID
char* FileName;	File name
long* JpegPtr;	Address of JPEG buffer ID

Description This function allocates a JPEG buffer, and loads it with a compressed image from a file. You can then inquire about the attributes of the compressed image using **imJpegInquire()** (in order, for example, to allocate a suitable buffer in which to decompress the image).

Note that all controls that were used to perform the compression are copied from the file to the JPEG buffer. These should not be changed before decompressing because, for the reconstructed image to match the original image, the same controls must be used to decompress.

The **Thread** parameter specifies the thread to which to send **imJpegRestore()** for execution.

The **FileName** parameter specifies the name of the file from which to load the JPEG buffer.

The **JpegPtr** parameter specifies the address in which to return the identifier of the JPEG buffer. If the JPEG buffer could not be allocated, 0 is returned.

See also **imJpegRead()**, **imJpegReadBuf()**. The **imJpegRead()** function allows you to load several images from the same file. The **imJpegReadBuf()** function reads a compressed image from an ordinary (contiguous) buffer.

imJpegSave

Sync

Synopsis Save a compressed image from a JPEG buffer to a file.

Format **void imJpegSave(Thread, FileName, Jpeg)**

long Thread; Thread ID
char* FileName; File name
long Jpeg; JPEG buffer ID

Description This function saves a compressed image from a JPEG buffer to a file. Note that tables and other controls in the JPEG buffer are also written to the file.

The **Thread** parameter specifies the thread to which to send **imJpegSave()** for execution.

The **FileName** parameter specifies the name of the file in which to save the compressed image.

The **Jpeg** parameter specifies the JPEG buffer containing the compressed image to save.

See also **imJpegWriteBuf()**. This function writes a compressed image from a JPEG buffer to an ordinary (contiguous) buffer. You can then save the data to disk using whatever method you like. Depending on your system, this might be faster than using **imJpegSave()**.

imJpegWrite

Sync

Synopsis Write a compressed image to an open file.

Format **void imJpegWrite(Thread, FileHandle, Jpeg)**

long Thread; Thread ID

FILE* FileHandle; Handle of open file

long Jpeg; JPEG buffer ID

Description This function writes a compressed image from a JPEG buffer to an open file. Note that tables and other controls in the JPEG buffer are also written to the file.

The **Thread** parameter specifies the thread to which to send **imJpegWrite()** for execution.

The **FileHandle** parameter specifies the handle of the open file (opened with **fopen()**). The compressed image is written starting at the current file position. After writing, the file remains open and is positioned immediately after the image just written.

The **Jpeg** parameter specifies the JPEG buffer containing the compressed image to write.

imJpegWriteBuf

Async

Synopsis Transfer a compressed image from a JPEG buffer to an ordinary (contiguous) buffer.

Format **void imJpegWriteBuf(Thread, Buf, Jpeg, Start, OSB)**

long Thread;	Thread ID
long Buf;	Destination buffer ID
long Jpeg;	JPEG buffer ID
long Start;	Byte offset
long OSB;	OSB ID (or 0)

Description This function transfers a compressed image from a JPEG buffer to an ordinary (contiguous) buffer. Note that the compressed image remains compressed in the ordinary buffer, allowing you to save the data to disk using whichever method is fastest on your system. Also, note that tables and other controls in the JPEG buffer are written to the ordinary buffer as well. However, it is not possible to operate on compressed image data in an ordinary buffer using the **imJpeg...()** functions; these functions require that image data be in a JPEG buffer.

The **Thread** parameter specifies the thread to which to send **imJpegWriteBuf()** for execution.

The **Buf** parameter specifies the buffer in which to write the compressed image. This buffer should be big enough to hold all of the data. If necessary, you can inquire about the required size of the buffer using the IM_JPEG_SIZE item of **imJpegInquire()**.

The **Jpeg** parameter specifies the JPEG buffer containing the compressed image to write.

The **Start** parameter specifies the byte offset, from the beginning of the destination buffer, at which to start writing the compressed image. This parameter allows you to store several compressed images in a single buffer.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

See also **imJpegReadBuf()**. Use the **imJpegReadBuf()** function to transfer the compressed image back to a JPEG buffer.

imPatAllocAutoModel

Sync

PP

NOA

Synopsis Automatically select and allocate a pattern matching model.

Format **void imPatAllocAutoModel(Thread, SrcBuf, XSize, YSize, XUncert, YUncert, Type, Mode, ModelPtr)**

long Thread;	Thread ID
long SrcBuf;	Source buffer ID
long XSize;	Model width
long YSize;	Model height
long XUncert;	Maximum uncertainty in X
long YUncert;	Maximum uncertainty in Y
long Type;	Model type
long Mode;	Mode of operation
long *ModelPtr;	Address of model ID(s)

Description This function automatically searches for and allocates one or more models of the specified size from a specified source buffer. Each model is unique within the area defined by the maximum positional uncertainty in the X and Y direction (**XUncert** and **YUncert**), respectively. It is not guaranteed to be unique within the entire target image.

Each model will be allocated with the usual default search parameters, except that the search position will be set to the actual position of the model, plus or minus the specified positional uncertainty.

Automatic allocation can take several seconds, and the speed can vary according to the **Mode** parameter setting.

The **Thread** parameter specifies the thread to which to send **imPatAllocAutoModel()** for execution.

The **SrcBuf** parameter specifies the buffer from which to allocate the model. This must be an unsigned 8-bit buffer.

The **XSize** and **YSize** parameters specify the width and height of the model, respectively.

The **XUncert** and **YUncert** parameters specify the maximum displacement (shift) expected between the reference position of the model(s) in the source and target images. These values represent the number of pixels by which the target location can be offset from the model location, in either direction on the vertical and horizontal axes. This function takes into account these uncertainty values and automatically selects models that are far enough from the image border to ensure their presence in the target images. This means that the positional uncertainty also corresponds to the minimum distance from the image border (in the corresponding X and Y directions), from which the function can select a model.

The **Type** parameter specifies the model type. This parameter must be set to IM_NORMALIZED.

The **Mode** parameter specifies the manner in which this function operates. The supported operating mode settings are:

IM_BEST	The slowest operating speed, which produces the best overall result.
IM_FAST	A fast operating speed.
IM_VERY_FAST	The fastest operating speed.
IM_DEFAULT	Same as IM_FAST
IM_MULTIPLE+ <i>n</i>	Allocate more than one model, where <i>n</i> is the number of models to allocate (up to a maximum of 10). You can combine this mode with one of the other operating speed modes. By default, only one model is chosen (<i>n</i> =1).

The **ModelPtr** parameter specifies the address in which to return the model identifier(s). The **ModelPtr** must point to enough memory to hold all the returned model IDs. During each attempt to search for and allocate a model, if no suitable model is not found (allocated), 0 is returned. Therefore, it is possible that some model IDs will be 0.

imPatAllocModel

Sync

Synopsis Allocate a pattern matching model.

Format **void imPatAllocModel(Thread, SrcBuf, XOff, YOff, XSize, YSize, Type, ModelPtr)**

long Thread;	Thread ID
long SrcBuf;	Source buffer ID
long XOff;	X coordinate of model origin
long YOff;	Y coordinate of model origin
long XSize;	Model width
long YSize;	Model height
long Type;	Model type
long* ModelPtr;	Address of model ID

Description This function allocates a model from a specified source buffer.

Note that the search parameters of a model affect how a search is performed with that model. When a model is allocated, these search parameters are set to default values (listed below). If a default value is not suitable, use the appropriate **imPatSet...()** function.

Acceptance level:	70%
Positional accuracy:	IM_MEDIUM (± 0.25 pixels)
Model center (hot spot):	int $((\mathbf{Xsize}-1)/2, (\mathbf{Ysize}-1)/2)$
Certainty level:	80%
Number of matches:	1
Search region:	IM_ALL (entire image)
Search speed:	IM_MEDIUM

The **Thread** parameter specifies the thread to which to send **imPatAllocModel()** for execution.

The **SrcBuf** parameter specifies the buffer from which to allocate the model. This must be an unsigned 8-bit integer buffer.

The **XOff** and **YOff** parameters specify the X and Y coordinates of the model origin, relative to the top-left corner of the source buffer.

The **XSize** and **YSize** parameters specify the width and height of the model, respectively.

The **Type** parameter specifies the model type. This parameter must be set to IM_NORMALIZED.

If model allocation is time critical in your application, you can speed it up by combining one of the following flags with IM_NORMALIZED:

- | | |
|--------------|--|
| IM_FAST | Faster model allocation. This leads to some very small differences in the model, but this should not affect matching in most applications. |
| IM_VERY_FAST | Fastest model allocation. This is a bit more likely to cause problems than IM_FAST. |

The **ModelPtr** parameter specifies the address in which to return the model identifier. If the model could not be allocated, 0 is returned.

imPatAllocResult

Sync

Synopsis Allocate a pattern matching result buffer.

Format **void imPatAllocResult(Thread, NumEntries, ResultPtr)**

long Thread; Thread ID
long NumEntries; Number of result buffer entries
long* ResultPtr; Result buffer ID

Description This function allocates a pattern matching result buffer.

Note that a pattern matching result buffer is used to store the results of a pattern matching operation.

The **Thread** parameter specifies the thread to which to send **imPatAllocResult()** for execution.

The **NumEntries** parameter specifies the number of entries for the result buffer. Note that this should be at least equal to the number of matches you want found.

The **ResultPtr** parameter specifies the address in which to return the result buffer identifier. If the result buffer could not be allocated, 0 is returned.

imPatCopy

Async

Synopsis Copy a pattern matching model.

Format `void imPatCopy(Thread, Model, DstBuf, Mode, OSB)`

<code>long Thread;</code>	Thread ID
<code>long Model;</code>	Model ID
<code>long DstBuf;</code>	Destination buffer ID
<code>long Mode;</code>	Mode of operation
<code>long OSB;</code>	OSB ID (or 0)

Description This function copies a model to a destination buffer. This function can also be used to copy only the model's "don't care" pixels.

The **Thread** parameter specifies the thread to which to send **imPatCopy()** for execution.

The **Model** parameter specifies the model to copy.

The **DstBuf** parameter specifies the destination buffer. This must be an unsigned 8-bit integer buffer, and be at least as large as the model.

The **Mode** parameter specifies the mode of operation. It can be set to:

`IM_DEFAULT` Copy the model to the buffer.

`IM_DONT_CARE` Copy only the model's "don't care" pixels to the buffer.

When copying the entire model, any "don't care" pixels in the model are given their original values. When copying only the model's "don't care" pixels, "don't care" pixels are given the value zero, while other pixels in the destination buffer are not overwritten. To give "don't care" pixels a value other than zero, combine (using + or |) `IM_DONT_CARE` with the required value (for example, `IM_DONT_CARE+255`). To clear pixels in the destination buffer that are not "don't care" to the value zero, combine `IM_DONT_CARE` with `IM_CLEAR_BACKGROUND` (for example, `IM_DONT_CARE+255+IM_CLEAR_BACKGROUND`).

Note that, by copying the entire model to a buffer and then copying only the model's "don't care" pixels to the same buffer, you can overlay "don't care" pixels onto the original model.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imPatFindModel

Async

PP

NOA

Synopsis Find a pattern matching model in an image.

Format **void imPatFindModel(Thread, SrcBuf, Model, Result, OSB)**

long Thread; Thread ID
 long SrcBuf; Source buffer ID
 long Model; Model ID
 long Result; Result buffer ID
 long OSB; OSB ID (or 0)

Description This function searches for a model in an image. The results of the search are written to the specified result buffer. Specifically, the match score and coordinates of each found match are written to the result buffer (in decreasing order of match score). Results can be read using **imPatGetResult()**.

Note that the search parameters of the model affect how the search is performed. To change the value of a search parameter, use the appropriate **imPatSet...()** function.

The **Thread** parameter specifies the thread to which to send **imPatFindModel()** for execution.

The **SrcBuf** parameter specifies the buffer in which to search for the model. This must be an unsigned 8-bit integer buffer.

The **Model** parameter specifies the model for which to search.

The **Result** parameter specifies the result buffer in which to store results.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imPatFree

Async

Synopsis Free a pattern matching model or result buffer.

Format **void imPatFree(Thread, ModelOrResult)**

long Thread; Thread ID

long ModelOrResult; Model or result buffer ID

Description This function deallocates a previously allocated model or result buffer.

The **Thread** parameter specifies the thread to which to send **imPatFree()** for execution.

The **ModelOrResult** parameter specifies the model or result buffer to deallocate.

imPatGetNumber

Sync

Synopsis Determine the number of matches above the acceptance level.

Format **long imPatGetNumber(Thread, Result, NumPtr)**

long Thread;	Thread ID
long Result;	Result buffer ID
long* NumPtr;	Address of return value (or NULL)

Description This function determines the number of matches of a search that are above the acceptance level.

Note that, before retrieving results using **imPatGetResult()**, **imPatGetNumber()** can be used to determine how much memory is required for the results. However, if you are sure you have allocated enough memory, there is no need to call **imPatGetNumber()**, since **imPatGetResult()** can also return the number of matches above the acceptance level.

The **Thread** parameter specifies the thread to which to send **imPatGetNumber()** for execution.

The **Result** parameter specifies the result buffer of the required search. Note that this buffer must have already been used in a call to **imPatFindModel()**.

The **NumPtr** parameter specifies the address (of a long) in which to place the number of matches. Since **imPatGetNumber()** also returns the number of matches, this parameter can be set to NULL.

Return value The returned value is the number of matches above the acceptance level.

imPatGetResult

Sync

Synopsis Transfer results of a search to Host memory.

Format **void imPatGetResult(Thread, Result, Type, Ptr)**

long Thread;	Thread ID
long Result;	Result buffer ID
long Type;	Result type
void* Ptr;	Address of array

Description This function transfers the results of a search to an array in Host memory. You can transfer the match scores, x coordinates, and y coordinates of the found matches, as well as the number of matches above the acceptance level. Results are returned in decreasing order of match score.

Note that the number of found matches might be less than the number of matches you requested (with **imPatSetNumber()**) because a match is considered found only if it is above the acceptance level. Before you retrieve results, you can call **imPatGetNumber()** to determine how much memory is required for the results. The **imPatGetNumber()** function returns the number of matches above the acceptance level. If you are sure you have allocated enough memory, however, there is no need to call **imPatGetNumber()** since **imPatGetResult()** can also return the number of matches above the acceptance level.

The **Thread** parameter specifies the thread to which to send **imPatGetResult()** for execution.

The **Result** parameter specifies the result buffer of the required search. Note that this buffer must have already been used in a call to **imPatFindModel()**.

The **Type** parameter specifies the type of result to transfer. It can be set to:

IM_ALL	The match scores, x coordinates, and y coordinates of the found matches, as well as the number of matches above the acceptance level.
IM_PAT_SCORE	The match scores of the found matches.
IM_PAT_POSITION_X	The x coordinates of the found matches.
IM_PAT_POSITION_Y	The y coordinates of the found matches.

When **Type** is set to IM_ALL, the results are stored in a specific data structure (defined below). Note that the structure member names are exactly the same as the corresponding #defines, with the IM_PAT_ prefix removed. In order to save memory and reduce transfer time to the Host, each result is stored in the smallest data type that can hold it. For example, integer results are returned as 16-bit values if possible, and floating-point values are returned as 32-bit single precision values. You must make sure that your compiler does not add any padding to the structure to change the alignment of structure members. However, the structure has been defined so that most compilers will not attempt to change the alignment.

```
typedef struct
{
    short number; /* Same as imPatGetNumber() */
    short reserved;
    float score;
    float position_x;
    float position_y;
} IM_PAT_RESULT_ST;
```

The **Ptr** parameter specifies the address of the user-supplied array in which to place results. If **Type** is set to IM_ALL, the array must be of type IM_PAT_RESULT_ST. If **Type** is set to anything else, the array should be of type double.

imPatInquire

Sync

Synopsis Inquire about a pattern matching model or result buffer.

Format **long imPatInquire(Thread, ModelOrResult, Item, ValuePtr)**

long Thread;	Thread ID
long ModelOrResult;	Model or result buffer ID
long Item;	Attribute about which to inquire
void* ValuePtr;	Address of return value (or NULL)

Description This function inquires about an attribute of a specified model or result buffer.

The **Thread** parameter specifies the thread to which to send **imPatInquire()** for execution.

The **ModelOrResult** parameter specifies the model or result buffer.

The **Item** parameter specifies the attribute about which to inquire. For a result buffer, **Item** can be set to:

IM_PAT_RESULT_SIZE	Number of entries in the result buffer. This is fixed during result buffer allocation.
--------------------	--

After calling **imPatFindModel()**, you can call **imPatInquire()** to get information about the largest match peak that was not returned as a match result. At the same time, because the peak might have been rejected at a low resolution level where the score is not very reliable, you can also inquire the level at which the score was obtained. To do so, use the following inquire items:

IM_PAT_BEST_REJECT_SCORE	The largest peak that was rejected during the search (either because it was below the acceptance level, or because you did not ask for enough matches).
IM_PAT_BEST_REJECT_LEVEL	The level at which the best reject score was obtained.

For a model, **Item** can be set to one of the attributes listed below, or to **IM_ALL**. In the latter case, the values of all attributes listed below are returned at the same time.

IM_PAT_ORIGINAL_X	X offset from model's center (hot spot) to the top-left corner of the image from which model was created. Model's hot spot can be set using imPatSetCenter() .
IM_PAT_ORIGINAL_Y	Y offset from model's center (hot spot) to the top-left corner of the image from which model was created. Model's hot spot can be set using imPatSetCenter() .
IM_PAT_OFFSET_X	X offset from model's top-left corner to the top-left corner of the image from which model was created. This is fixed during model allocation.
IM_PAT_OFFSET_Y	Y offset from model's top-left corner to the top-left corner of the image from which model was created. This is fixed during model allocation.
IM_PAT_SIZE_X	Model width. This is fixed during model allocation.
IM_PAT_SIZE_Y	Model height. This is fixed during model allocation.
IM_PAT_TYPE	Model type. This is fixed during model allocation.
IM_PAT_POSITION_START_X	X coordinate of search region's origin, relative to the top-left corner of the image being searched. This can be set using imPatSetPosition() .
IM_PAT_POSITION_START_Y	Y coordinate of search region's origin, relative to the top-left corner of the image being searched. This can be set using imPatSetPosition() .
IM_PAT_POSITION_SIZE_X	Width of search region. This can be set using imPatSetPosition() .
IM_PAT_POSITION_SIZE_Y	Height of search region. This can be set using imPatSetPosition() .
IM_PAT_POSITION_ACCURACY	Positional accuracy. This can be set using imPatSetAccuracy() .
IM_PAT_NUMBER	Number of matches to find. This can be set using imPatSetNumber() .
IM_PAT_SPEED	Search speed. This can be set using imPatSetSpeed() .

IM_PAT_ACCEPTANCE	Acceptance level. This can be set using imPatSetAcceptance() .
IM_PAT_CERTAINTY	Certainty level. This can be set using imPatSetCertainty() .
IM_PAT_CENTER_X	X offset from model's center (hot spot) to model's top-left corner. Model's hot spot can be set using imPatSetCenter() .
IM_PAT_CENTER_Y	Y offset from model's center (hot spot) to model's top-left corner. Model's hot spot can be set using imPatSetCenter() .
IM_PAT_PREPROCESSED	Whether the model was preprocessed (non-zero i.e. TRUE) or not preprocessed (zero i.e. FALSE). Model can be preprocessed using imPatPreprocModel() .
IM_PAT_FIRST_LEVEL	Resolution level for the initial stage of the search. The level can be set using imPatSetSearchParameter() .
IM_PAT_LAST_LEVEL	Resolution level for the final stage of the search. The level can be set using imPatSetSearchParameter() .
IM_PAT_MODEL_STEP	Model step for search: 1 or 2. The model step can be set using imPatSetSearchParameter() .
IM_PAT_FAST_FIND	Whether fast peak finding is enabled. This can be set using imPatSetSearchParameter() .
IM_PAT_SCORE_TYPE	Whether negative match scores are clipped to 0 (IM_DEFAULT) or set to their absolute values (IM_ABSOLUTE). Match scores can be set to their absolute values using imPatSetSearchParameter() .

When **Item** is set to IM_ALL, the results are stored in a specific data structure (defined below). Note that the structure member names are exactly the same as the corresponding predefined constants, with the IM_PAT_ prefix removed. To save memory and reduce transfer time to the Host, each result is stored in the smallest data type that can hold it. For example, integer results are returned as 16-bit values if possible, and floating-point values are returned as 32-bit single precision values. You must make sure that your compiler does not add any padding to the

structure to change the alignment of structure members. However, the structure has been defined so that most compilers will not attempt to change the alignment.

```
typedef struct
{
    short type;
    short offset_x;
    short offset_y;
    short size_x;
    short size_y;
    short preprocessed;
    float center_x;
    float center_y;
    float original_x;
    float original_y;
    short number;
    short speed;
    short position_start_x;
    short position_start_y;
    short position_size_x;
    short position_size_y;
    short position_accuracy;
    short reserved;
    float acceptance;
    float certainty;
    long first_level;
    long last_level;
    long model_step;
    long fast_find;
    long score_type;
} IM_PAT_INQUIRE_ST;
```

The **ValuePtr** parameter specifies the address in which to return the value of the inquired attribute. If **Item** is set to IM_ALL, the address must be to an array of type IM_PAT_INQUIRE_ST. If **Item** is set to a specific attribute, the address should be of type double, although you can have the attribute returned as type long by setting **ValuePtr** to NULL and using the function's return value instead.

Note The values for IM_PAT_ORIGINAL_X and IM_PAT_ORIGINAL_Y can be compared with the coordinates returned by **imPatGetResult()**, to indicate the shift between the image from which the model was created and the image being searched.

Return value The returned value is the value of the inquired attribute, cast to long if necessary. This only applies when inquiring about a specific attribute.

imPatPreprocModel

Async

Synopsis Preprocess a pattern matching model.

Format `void imPatPreprocModel(Thread, Buf, Model, Mode, OSB)`

<code>long Thread;</code>	Thread ID
<code>long Buf;</code>	Target image ID (or 0)
<code>long Model;</code>	Model ID
<code>long Mode;</code>	Mode of operation
<code>long OSB;</code>	OSB ID (or 0)

Description This function preprocesses a specified model. Preprocessing analyzes a model to determine which shortcuts can safely be used during a search with the model.

More shortcuts might be found if you provide a typical image that will be searched. However, you should only provide such an image if all images on which you will be searching for the model have the same type of background. If the images might have different backgrounds, do not provide one.

Note that, when you save a model to disk, the preprocessing changes are also saved; there is no need to preprocess again after restoring it. Therefore, you normally need to preprocess a model just once, right after creating it. However, if you use **imPatSetDontCare()**, the effect of preprocessing is undone; in this case, you will need to preprocess again.

The **Thread** parameter specifies the thread to which to send **imPatPreprocModel()** for execution.

The **Buf** parameter specifies the "typical image" with which to preprocess the model. This must be an 8-bit unsigned integer buffer. Set this parameter to 0 if you are not providing an image (note that you should not provide an image if the images on which you will be searching for the model might have different backgrounds).

The **Model** parameter specifies the model to preprocess.

The **Mode** parameter specifies the mode of operation. This parameter must be set to `IM_DEFAULT`.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imPatRead

Sync

Synopsis Read a pattern matching model from an open file.

Format **void imPatRead(Thread, FileHandle, ModelPtr)**

long Thread;	Thread ID
FILE* FileHandle;	Handle of open file
long* ModelPtr;	Address of model ID

Description The function reads a model from an open file, and assigns it an identifier.

Note that the model's search parameters (including any effects of preprocessing) are also read from the file.

The **Thread** parameter specifies the thread to which to send **imPatRead()** for execution.

The **FileHandle** parameter specifies the handle of the open file (opened with **fopen()**). Before calling this function, the file pointer must be positioned just before the start of a valid pattern matching model. After the function call, the file remains open and is positioned immediately after the model.

The **ModelPtr** parameter specifies the address in which to return the model identifier. If the model could not be allocated, 0 is returned.

See also **imPatRestore()**. The **imPatRestore()** function restores a previously saved model from file.

imPatRestore

Sync

Synopsis Restore a pattern matching model from a file.

Format **void imPatRestore(Thread, FileName, ModelPtr)**

long Thread;	Thread ID
char* FileName;	File name
long* ModelPtr;	Address of model ID

Description This function restores a previously saved model from a file, and assigns it an identifier.

Note that the model's search parameters (including its "don't care" pixels and any effects of preprocessing) are also restored from the file.

The **Thread** parameter specifies the thread to which to send **imPatRestore()** for execution.

The **FileName** parameter specifies the name of the file from which to restore the model.

The **ModelPtr** parameter specifies the address in which to return the model identifier. If the model could not be allocated, 0 is returned.

See also **imPatRead()**. The **imPatRead()** function allows you to read several models from the same file.

imPatSave

[Sync](#)

Synopsis Save a model to a file.

Format **void imPatSave(Thread, FileName, Model)**

long Thread;	Thread ID
char* FileName;	File name
long Model;	Model ID

Description This function saves a model to a file.

Note that the model's search parameters (including its "don't care" pixels and any effects of preprocessing) are also saved to the file.

The **Thread** parameter specifies the thread to which to send **imPatSave()** for execution.

The **FileName** parameter specifies the name of the file in which to save the model.

The **Model** parameter specifies the model to save.

See also **imPatWrite()**. The **imPatWrite()** function allows you to save several models in one file.

imPatSetAcceptance

Async

Synopsis Set the acceptance level for a search.

Format **void imPatSetAcceptance(Thread, Model, Acceptance)**

long Thread;	Thread ID
long Model;	Model ID
double Acceptance;	Acceptance level

Description This function sets the acceptance level for a search.

Note that the acceptance level is the match score above which a match is considered to be found. The default acceptance level is 70%.

The **Thread** parameter specifies the thread to which to send **imPatSetAcceptance()** for execution.

The **Model** parameter specifies the model that will be used in the search.

The **Acceptance** parameter specifies the acceptance level. It should be set to a value greater than 0 but less than or equal to 100.

imPatSetAccuracy

Async

Synopsis Set the positional accuracy for a search.

Format **void imPatSetAccuracy(Thread, Model, Accuracy)**

long Thread; Thread ID
long Model; Model ID
long Accuracy; Positional accuracy

Description This function sets the positional accuracy for a search.

Note that the positional accuracy specifies the degree to which the position of a found match is refined. The position can be refined to within ± 0.5 pixels, ± 0.25 pixels, or ± 0.1 pixels. The more accuracy you require, the longer the search process.

The **Thread** parameter specifies the thread to which to send **imPatSetAccuracy()** for execution.

The **Model** parameter specifies the model that will be used in the search.

The **Accuracy** parameter specifies the positional accuracy. It can be set to:

IM_LOW Low accuracy (± 0.5 pixels).
IM_MEDIUM Medium accuracy (± 0.25 pixels). This is the default positional accuracy.
IM_HIGH High accuracy (± 0.1 pixels).

imPatSetCenter

Async

Synopsis Set a model's center position (hot spot).

Format **void imPatSetCenter(Thread, Model, XCen, YCen)**

long Thread; Thread ID
long Model; Model ID
double XCen; X coordinate of center
double YCen; Y coordinate of center

Description This function sets a model's hot spot.

Note that a model's hot spot determines the returned coordinates of a match. Specifically, the returned coordinates of a match are the coordinates of the model's hot spot relative to the top-left corner of the image being searched. By default, a model's hot spot is its center pixel.

The **Thread** parameter specifies the thread to which to send **imPatSetCenter()** for execution.

The **Model** parameter specifies the model to set.

The **XCen** and **YCen** parameters specify the x and y coordinates of the model's hot spot, relative to the top-left corner of the model.

imPatSetCertainty

Async

Synopsis Set the certainty level for a search.

Format **void imPatSetCertainty(Thread, Model, Certainty)**

long Thread;	Thread ID
long Model;	Model ID
double Certainty;	Certainty level

Description This function sets the certainty level for a search.

Note that the certainty level is the match score above which the search algorithm can assume that it has found a match and can stop searching the rest of the image for a better score. The default certainty level is 80%.

The **Thread** parameter specifies the thread to which to send **imPatSetCertainty()** for execution.

The **Model** parameter specifies the model that will be used in the search.

The **Certainty** parameter specifies the certainty level. It should be set to a value greater than 0 but less than or equal to 100.

imPatSetDontCare

Async

Synopsis Set model pixels to the "don't care" state.

Format **void imPatSetDontCare(Thread, Model, SrcBuf, XOff, YOff, Value)**

long Thread;	Thread ID
long Model;	Model ID
long SrcBuf;	Source buffer ID
long XOff;	X offset
long YOff;	Y offset
long Value;	Don't care value

Description This function sets model pixels to the "don't care" state. The "don't care" pixels of a model are ignored during the search process (they do not affect the match score).

Note that model pixels are set to the "don't care" state by using a region of a specified source buffer that is equal in size to the model. Specifically, each pixel in this region is compared against a specified value; if it has this value, its corresponding pixel in the model is set to the "don't care" state.

The **Thread** parameter specifies the thread to which to send **imPatSetDontCare()** for execution.

The **Model** parameter specifies the model whose pixels are to be set to the "don't care" state.

The **SrcBuf** parameter specifies the source buffer. This must be an unsigned 8-bit integer buffer.

The **XOff** and **YOff** parameters specify the horizontal and vertical offset from the top-left corner of the source buffer to the origin of the required region. These parameters, and the size of the model, determine the source buffer pixels that are used to set model pixels to the "don't care" state.

The **Value** parameter specifies the value against which to compare source buffer pixels.

imPatSetNumber

Async

Synopsis Set the number of matches to find.

Format **void imPatSetNumber(Thread, Model, Number)**

long Thread; Thread ID
long Model; Model ID
long Number; Number of matches

Description This function sets the number of matches to find in a search.

By default, the search algorithm finds only one match (the one with the highest match score above the acceptance level). If you specify that n matches be found, then the n highest match scores above the acceptance level are returned, in decreasing order of match score. You can also specify that all matches be found, in which case all matches above the acceptance level are returned, in decreasing order of match score.

Note that, the more matches you require, the longer the search process.

The **Thread** parameter specifies the thread to which to send **imPatSetNumber()** for execution.

The **Model** parameter specifies the model that will be used in the search.

The **Number** parameter specifies the number of matches. To search for all matches above the acceptance level, set this parameter to IM_ALL.

imPatSetPosition

Async

Synopsis Set the search region for a search.

Format **void imPatSetPosition(Thread, Model, XOff, YOff, XSize, YSize)**

long Thread; Thread ID
 long Model; Model ID
 long XOff; X offset
 long YOff; Y offset
 long XSize; Search width
 long YSize; Search height

Description This function sets the search region for a search.

By default, the search region is the entire image being searched. To increase search speed, however, you should make the search region as small as possible. Since the search region is the area in which to find the model's center (hot spot), the search region can be even smaller than the model (as small as a single pixel).

Note that, in general, it is better to use this function rather than a child buffer when you want the search region to be smaller than the entire image being searched. A child buffer can cause misleading results because the search algorithm will not use the area outside the child buffer.

The **Thread** parameter specifies the thread to which to send **imPatSetPosition()** for execution.

The **Model** parameter specifies the model that will be used in the search.

The **XOff** and **YOff** parameters specify the horizontal and vertical offset from the top-left corner of the image being searched to the origin of the search region.

The **XSize** and **YSize** parameters specify the width and height of the search region.

imPatSetSearchParameter

Async

Synopsis Set an internal search parameter.

Format `void imPatSetSearchParameter(Thread, Model, Param, Value)`

long Thread; Thread ID
 long Model; Model ID
 long Param; Parameter to set
 double Value; Parameter value

Description This function sets an internal search parameter. These search parameters are considered "internal" since, by default, they are automatically derived by the pattern matching algorithm, from the model's more basic search parameters (such as its speed and accuracy). You rarely need to explicitly set an internal search parameter, since its automatically derived value is suitable for most applications. If you do set an internal search parameter, you should know how it can affect the pattern matching algorithm; see the *Genesis Native Library User Guide* for details.

The **Thread** parameter specifies the thread to which to send **imPatSetSearchParameter()** for execution.

The **Model** parameter specifies the model that will be used in the search.

The **Param** parameter specifies the search parameter to set, while the **Value** parameter specifies its value. The table below lists available parameters, and their allowable values.

Parameter	Values	Meaning
IM_PAT_FIRST_LEVEL	0 - 7	Resolution level for the initial stage of the search.
	IM_DEFAULT	Determine first level automatically.

Note that Level 0 is the original image being searched. Each higher level is half the size (and resolution) of the previous one. If the specified level does not exist, the highest available level will be used. A higher first level speeds up the initial search but makes it less reliable, because the model might not retain enough distinctive features at such a low resolution.

IM_PAT_LAST_LEVEL	0 - 7	Resolution level for the final stage of the search.
	IM_DEFAULT	Determine last level automatically.

If the specified level does not exist, the highest available level will be used. A higher last level reduces search time but also reduces positional accuracy. Match scores might also be less reliable for levels above 0, depending on the characteristics of the model. Specify a last level of 0 for a positional accuracy of roughly ± 0.1 pixels, a level of 1 for a positional accuracy of ± 0.25 pixels, and a level of 2 for a positional accuracy of ± 0.5 pixels.

IM_PAT_MODEL_STEP	1 or 2	Set the model step to 1 or 2 (that is, use all or every second model pixel in the high resolution stage of the search).
	IM_DEFAULT	Preprocessing determines the model step.

When the model step is set to 1, all model pixels are used in the correlation. When set to 2, only every second model pixel (in both the x and y directions) are used. This speeds up the last (high resolution) stage of the search, particularly for large models. The match score might be affected if the model has many fine features, but will tend not to be affected if the model has mainly coarse features.

IM_PAT_FAST_FIND	IM_ENABLE	Force fast peak finding.
	IM_DISABLE	Prevent fast peak finding.
	IM_DEFAULT	Preprocessing decides if fast peak finding is appropriate.

When fast peak finding is disabled, the initial search (at the resolution level determined by IM_PAT_FIRST_LEVEL) computes the correlation at every position in the search region. This guarantees that the biggest match peak will be found, and that it will be investigated first. If fast peak finding is enabled, the search algorithm attempts to find the peaks without checking every point. This is safe in most cases, but can cause matches to be missed for models that produce very narrow peaks.

IM_PAT_SCORE_TYPE	IM_DEFAULT	Clip negative match scores to 0.
	IM_ABSOLUTE	Take the absolute value of match scores. This allows you to find negative versions of the model.

IM_ALL	IM_DEFAULT	Determine all internal search parameters automatically.
IM_PAT_OFFSET_X	any integer ≥ 0	Set the model's allocation X offset to the specified value. This might be useful when the original value is lost after rotating a model. You can use imPatInquire() to inquire about the current X offset.
IM_PAT_OFFSET_Y	any integer ≥ 0	Set the model's allocation Y offset to the specified value. This might be useful when the original value is lost after rotating a model. You can use imPatInquire() to inquire about the current y offset.
IM_PAT_REJECTION	Threshold value (in percent)	<p>Set the rejection threshold to use for rejecting candidate model peaks at low resolution levels. This can speed up the search when some of the matches you request do not reach the certainty threshold, or when you request more matches than are really present in the image.</p> <p>The rejection threshold should usually be set much lower than the acceptance threshold. For example, a good level to set it to is about 20% to 30%. Note that if it is too low, you will not see any increase in speed. However, if it is too high, you risk rejecting real match peaks.</p> <p>Note that the rejection threshold is not saved and restored with the model. So, if you always want to use your own rejection threshold level, you should set it each time after restoring your model and before calling imPatFindModel(). Otherwise, the default rejection threshold value is used.</p>
	IM_DEFAULT	Determine the rejection threshold automatically

imPatSetSpeed

Async

Synopsis Set the speed for a search.

Format **void imPatSetSpeed(Thread, Model, Speed)**

long Thread; Thread ID

long Model; Model ID

long Speed; Search speed

Description This function sets the speed (high, medium, low, very low) at which to perform a search. Note that, as you increase the speed, the likelihood of finding the model decreases slightly. In addition, match scores and positional accuracies might be a little less accurate.

You can search at high speed if the image being searched is of good quality or if your model is relatively simple. However, you should not search at high speed if you need the highest possible accuracy (search at medium or low speed instead).

You should search at medium speed if the image being searched is of medium quality or if your model is relatively complex. You should only search at low or very low speeds if the image being searched is of poor quality or if you have encountered problems at higher speeds.

The **Thread** parameter specifies the thread to which to send **imPatSetSpeed()** for execution.

The **Model** parameter specifies the model that will be used in the search.

The **Speed** parameter specifies the search speed. It can be set to:

IM_HIGH Search at high speed.

IM_MEDIUM Search at medium speed. This is the default search speed.

IM_LOW Search at low speed.

IM_VERY_LOW Search at very low speed.

imPatWrite

Sync

Synopsis Write a model to an open file.

Format **void imPatWrite(Thread, FileHandle, Model)**

long Thread;	Thread ID
FILE* FileHandle;	Handle of open file
long Model;	Model ID

Description This function writes a model to an open file.

Note that the model's search parameters (including its "don't care" pixels and any effects of preprocessing) are also written to the file.

The **Thread** parameter specifies the thread to which to send **imPatWrite()** for execution.

The **FileHandle** parameter specifies the handle of the open file (opened with **fopen()**). The model is written starting at the current file position. After writing, the file remains open and is positioned immediately after the model just written.

The **Model** parameter specifies the model to write to the file.

imRleDecode

Async

PP

Synopsis Decode (decompress) a run-length encoded image and write into a buffer.

Format `void imRleDecode(Thread, Buf, CompBuf, Control, OSB)`

long Thread;	Thread ID
long Buf;	Decompressed buffer ID
long CompBuf;	Compressed buffer ID
long Control;	Control Buffer ID (or 0)
long OSB	OSB ID (or 0)

Description This function decompresses a run-length encoded image from the specified 1-dimensional buffer (**CompBuf**) and writes the result into another buffer (**Buf**).

The **Thread** parameter specifies the thread to which to send **imRleDecode()** for execution.

The **Buf** parameter specifies the buffer in which to place the decompressed image. This buffer can have a depth of 1 or 8 bits. The decompressed buffer must be exactly the right size for the image data.

The **CompBuf** specifies the identifier of the compressed buffer. This buffer must be 1-dimensional and have a depth of 8 bits. This buffer can be larger than necessary; decoding will stop once the decompressed buffer (**Buf**) is completely filled.

The **Control** parameter specifies the control buffer with which to control decoding (decompression). Relevant fields for **imRleDecode()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Value	Description
IM_RLE_BACK_COLOR	0 to 255	Color to use for background runs. The default is 0 .
IM_RLE_COLOR	0 to 255	Color to use for foreground runs. The default is 255 .

Field	Value	Description
IM_RLE_MODE	IM_BINARY	Compression mode. Currently only IM_BINARY is supported. Accordingly, pixels with a 0 value are treated as background pixels, and all non-zero pixels are treated as foreground pixels.
IM_RLE_START	<integer>	Starting position, in bytes, within CompBuf . The default is 0.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

Note The size of the decompressed buffer must be an exact fit for the image data, since no size information is recorded in the compressed buffer.

imRleEncode

Async

PP

Synopsis Run-length encode (compress) an image and write the contents into a 1-dimensional buffer.

Format **void imRleEncode(Thread, Buf, CompBuf, Control, OSB)**

long Thread;	Thread ID
long Buf;	Uncompressed buffer ID
long CompBuf;	Compressed buffer ID
long Control;	Control Buffer ID (or 0)
long OSB	OSB ID (or 0)

Description This function run-length encodes (compresses) the image and writes the contents into a 1-dimensional buffer (**CompBuf**). After calling this function, the actual size of the compressed image can be determined by reading the `IM_RLE_SIZE` field from the control buffer. You can use this value to allocate a child buffer and limit operations on the compressed buffer (such as saving) to the compressed data.

Note that the compressed buffer contains no header information. Therefore, it is necessary to keep track of the dimensions of the original image so that a suitable buffer can be allocated for any subsequent decoding (decompression) operation.

The **Thread** parameter specifies the thread to which to send **imRleEncode()** for execution.

The **Buf** parameter specifies the buffer to compress. This buffer can be a 1- or 8-bit buffer.

The **CompBuf** specifies the identifier of the compressed buffer. This buffer must be one-dimensional, have a depth of 8 bits, and be large enough to hold the entire compressed image.

The **Control** parameter specifies the control buffer with which to control encoding (compression). Relevant fields for **imRleEncode()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used.

Field	Value	Description
IM_RLE_MODE	IM_BINARY	Compression mode. Currently only IM_BINARY is supported. Accordingly, pixels with a 0 value are treated as background pixels, and all non-zero pixels are treated as foreground pixels.
IM_RLE_START	<integer>	Starting position, in bytes, within CompBuf . This allows several compressed images to be appended to the same buffer. The default is 0.

After compression, the following field can be read to determine the size of the compressed image:

IM_RLE_SIZE	<integer>	The size of the compressed image (in bytes). The value returned for IM_RLE_SIZE will be correct even if the compressed buffer is too large or too small to hold the entire image.
-------------	-----------	---

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function.

imSyncAlloc

Sync

Gen-LC

Synopsis Allocate an operation status block.

Format **void imSyncAlloc(Thread, OSBPtr)**

long Thread; Thread ID
long* OSBPtr; Address of OSB ID

Description This function allocates an operation status block (OSB).

Upon allocation, an OSB is in auto reset mode and is initialized to the IM_WAITING state. To change an OSB's mode of operation, use **imSyncControl()**.

The **Thread** parameter specifies the thread to which to send **imSyncAlloc()** for execution. The OSB is allocated on the device associated with this thread.

The **OSBPtr** parameter specifies the address in which to return the OSB identifier. If the OSB could not be allocated, 0 is returned.

imSyncControl

Async

Gen-LC

Synopsis Change the state or mode of operation of an OSB.

Format **void imSyncControl(Thread, OSB, Item, Value)**

long Thread;	Thread ID
long OSB;	OSB ID
long Item;	Item to set
double Value;	Value for Item

Description This function changes the state or mode of operation of an operation status block (OSB). You can use an OSB in either auto reset or manual reset mode. In auto reset mode, an OSB's state is automatically changed, according to the state of the function to which it is associated. In manual reset mode, you must explicitly change an OSB's state, using **imSyncControl()**.

Auto-reset OSBs are the most widely used. Once in the completed (signalled) state, they remain there only until the Host or another thread waits on them (with **imSyncHost()** or **imSyncThread()**). As soon as the wait condition is satisfied, the waiting task is released and the OSB is automatically reset to the waiting (non-signalled) state. You can then immediately re-use the OSB, if desired, in another command. Only one task at a time should wait on an auto-reset OSB. Note that only the OSB state is reset; any associated error code is retained until the next time the completed state is reached, then it is overwritten.

Manual-reset OSBs remain signalled until explicitly reset, so several tasks can wait on them.

You can also set a timeout on an OSB that is independent of the normal application timeout. Each OSB has its own associated timeout value (the default value is infinity). It is easier to recover from an OSB timeout than from an application timeout. For example, to detect the presence of a camera, try to grab using an OSB with a short timeout value. Then, wait on the OSB and see if the wait completed normally or timed-out. If the wait timed-out, then the camera is not being recognized on the system.

The **Thread** parameter specifies the thread to which to send **imSyncControl()** for execution.

The **OSB** parameter specifies the OSB whose state or mode to change.

The **Item** parameter specifies whether to change the OSB's state or mode, while the **Value** parameter specifies its new state or mode. Possible values for **Item** and **Value** are listed below (default values, if any, are in bold-face).

IM_OSB_STATE	IM_WAITING (also known as IM_NON_SIGNALLED)
	IM_EXECUTING
	IM_READY
	IM_STARTED
IM_OSB_MODE	IM_COMPLETED (also known as IM_SIGNALLED)
	IM_AUTO_RESET
	IM_MANUAL_RESET

You can set an OSB's timeout value using the following control item.

Item	Value	Meaning
IM_OSB_TIMEOUT	a floating point value > 0	Timeout value in seconds.
	IM_INFINITE	No timeout.

imSyncFree

Async

Gen-LC

Synopsis Free an operation status block.

Format **void imSyncFree(Thread, OSB)**

long Thread; Thread ID

long OSB; OSB ID

Description This function deallocates a previously allocated operation status block (OSB). Note that, if the OSB has been associated with a function, you should not free the OSB until that function has finished executing.

The **Thread** parameter specifies the thread to which to send **imSyncFree()** for execution.

The **OSB** parameter specifies the OSB to deallocate.

imSyncGetError

Sync

Gen-LC

Synopsis Check a function for errors.

Format **long imSyncGetError(Thread, OSB, Item, ValuePtr)**

long Thread; Thread ID
 long OSB; OSB ID
 long Item; Error item to retrieve
 void* ValuePtr; Address in which to return error item (or NULL)

Description This function checks a specific function for errors.

Note that errors can only be detected for functions that have finished executing. Therefore, you must ensure that the required function has completed before calling **imSyncGetError()**.

The **Thread** parameter specifies the thread to which to send **imSyncGetError()** for execution.

The **OSB** parameter specifies the operation status block associated with the function that you are checking.

The **Item** parameter specifies the error item to retrieve. It can be set to:

IM_ERR_CODE	The error code.
IM_ERR_MSG	The error message (maximum size of string: IM_ERR_MSG_SIZE bytes).
IM_ERR_FUNC	The name of the offending function (maximum size of string: IM_ERR_FUNC_SIZE bytes).
IM_ERR_MSG_FUNC	The error message and the name of the offending function (maximum size of string: IM_ERR_SIZE bytes).

The **ValuePtr** parameter specifies the address in which to return the error item. If you are retrieving IM_ERR_MSG, IM_ERR_FUNC, or IM_ERR_MSG_FUNC, **ValuePtr** should be the address of a char; if you are retrieving IM_ERR_CODE, **ValuePtr** should be the address of a long.

Since **imSyncGetError()** also returns the error code, **ValuePtr** can be set to NULL when you are retrieving the error code.

Return value The returned value is the error code.

Error code	Meaning
IM_SUCCESS	No error.
IM_ERR_BUFFER	Invalid buffer ID.
IM_ERR_DEVICE	Invalid device ID, or no such device.
IM_ERR_FILE	File access error.
IM_ERR_HALTED	Function halted by imThrHalt() .
IM_ERR_MEMORY	Insufficient memory to carry out the operation.
IM_ERR_NOT_PRESENT	Referenced item not present.
IM_ERR_OPCODE	Invalid opcode received.
IM_ERR_OSB	Invalid OSB ID.
IM_ERR_PARAMETER	Parameter (other than an ID-type parameter) invalid or unacceptable.
IM_ERR_RESTRICTION	Operation unable to execute due to a restriction.
IM_ERR_SYSTEM	No such system.
IM_ERR_TIMEOUT	Device unable to respond during timeout period.
IM_ERR_THREAD	Invalid thread ID.
IM_ERR_BUF_ATTRIBUTE	Unacceptable buffer attribute (size, data type, etc.).
IM_ERR_MISC	Miscellaneous error (the error message string will provide details on the cause of the error).

imSyncHost

Sync

Gen-LC

Synopsis Synchronize the Host with a function.

Format `long imSyncHost(Thread, OSB, State)`

<code>long Thread;</code>	Thread ID
<code>long OSB;</code>	OSB ID (or 0)
<code>long State;</code>	State to wait for

Description This function halts execution on the Host, until a specified function is in a specified state, or until all functions in a specified thread have completed. Alternatively, this function can be used when you simply want the current state of a function, without halting Host execution.

The **Thread** parameter specifies the thread to which to send **imSyncHost()** for execution.

The **OSB** parameter specifies the operation status block associated with the function for which you are waiting. To wait for the completion of all functions in the specified thread, set this parameter to 0.

The **State** parameter specifies the OSB state for which to wait. If you simply want the current state of the specified function, without halting Host execution, set **State** to **IM_INQUIRE**. If you are waiting for all functions in a thread to complete (i.e. have set **OSB** to 0), set **State** to **IM_COMPLETED**. If you are waiting on a specific OSB, **State** can be set to:

<code>IM_EXECUTING</code>	Wait until the function has started executing.
<code>IM_COMPLETED</code> (also known as <code>IM_SIGNALLED</code>)	Wait until the function has completed.
<code>IM_LINE_INT + n</code>	Wait until the grab line count is greater than or equal to <i>n</i> . This setting only applies if line interrupts were enabled. An OSB must have been passed to the imDigGrab() command.
<code>IM_READY</code>	Wait until the function is ready to send or receive data.
<code>IM_STARTED</code>	Wait until the function has started to send or receive data.

Note that **IM_READY** and **IM_STARTED** apply only to certain functions (see individual function descriptions).

Example The following code shows how to inquire about the current grab line when line interrupts are enabled.

```
Line = imSyncHost(Thread, OSB, IM_LINE_INT + IM_INQUIRE);
```

Return value The returned value indicates the current state of the specified OSB. The returned value can be one of the following:

IM_EXECUTING	Function is executing.
IM_COMPLETED	Function has completed.
IM_SIGNALLED	Synonym for IM_COMPLETED.
IM_WAITING	Function has not started to execute.
IM_NON_SIGNALLED	Synonym for IM_WAITING.
IM_READY	Function is ready to receive data.
IM_STARTED	Function has started to receive data.
IM_ERR_TIMEOUT	Wait ended because the OSB timed-out.

A wait can only time out if you explicitly set the OSB timeout value by calling **imSyncControl()** (there is no timeout by default).

Note that no error is logged when an OSB times out because you might want to return from a wait after a maximum time, but you cannot detect the timeout condition through the error handling functions. For example, you might want to do some useful work, then continue waiting on the OSB later. To do so, you should explicitly test the return value of **imSyncHost()** to detect if the OSB timed-out. If you want to continue waiting after a timeout, just call **imSyncHost()** again with the same parameters. You can always try the wait operation again because the OSB's state is not affected by a wait that times out.

The timeout period starts once the **imSyncHost()** command begins executing on-board, which might be some time after the function is called if other commands are queued ahead of it in the same thread.

imSyncThread

Async

Gen-LC

Synopsis Synchronize a thread with an operation in another thread.

Format `void imSyncThread(Thread, OSB, State)`

long Thread; Thread ID
long OSB; OSB ID
long State; State to wait for

Description This function halts execution of a thread, until an operation in another thread is in a specified state.

The **Thread** parameter specifies the thread to halt.

The **OSB** parameter specifies the operation status block with which to synchronize. To synchronize with a function when you do not have its OSB identifier, send the **imThrNop()** function to the same thread as the function, and synchronize with **imThrNop()** instead.

The **State** parameter specifies the OSB state for which to wait. This parameter can be set to:

IM_EXECUTING	Wait until the function is executing.
IM_COMPLETED (also known as IM_SIGNALLED)	Wait until the function has completed.
IM_LINE_INT + n	Wait until the grab line count is greater than or equal to <i>n</i> . This setting only applies if line interrupts were enabled. An OSB must have been passed to the imDigGrab() command.
IM_READY	Wait until the function is ready to send or receive data.
IM_STARTED	Wait until the function has started to send or receive data.

Note that IM_READY and IM_STARTED apply only to certain functions (see individual function descriptions).

Note This function also halts execution of a thread during an OSB timeout (**imSyncControl()**). One use for this is to implement a delay in the execution of commands queued to a thread. To do so, set the timeout period

of an OSB that is not used by any other function, and then call **imSyncThread()**. The specified thread will be blocked from executing other functions for the timeout period.

The timeout period starts once the function begins executing on-board, which might be some time after the function is called if other functions are queued ahead of it in the same thread.

imSysClock

Sync

Gen-LC

Synopsis Read the system clock.

Format **double imSysClock(Thread, Offset)**

long Thread; Thread ID

double Offset; Offset to subtract from current time

Description This function returns the current on-board clock value, minus a specified offset.

The on-board clock value is specified in seconds as a double precision number. As the clock value is of very high resolution, this function can accurately time short intervals, such as the execution time of one function.

The **Thread** parameter specifies the thread to which to send **imSysClock()** for execution.

The **Offset** parameter specifies the offset to subtract from the current on-board clock value. Express this parameter in seconds.

Return value The returned value is the current on-board clock value minus the specified offset.

Example The following code uses **imSysClock()** to measure the execution time of one function. All functions are sent to the same thread to ensure proper synchronization.

```
InitialTime = imSysClock(Thread, 0.0);
imIntConvolve(Thread, Src, Dst, IM_SMOOTH, 0, 0); /* Operation to measure */
Time = imSysClock(Thread, InitialTime);
```

Note that the time measured above includes the overhead of **imSysClock()**. If you are timing very short intervals where this overhead might be significant, you should first determine the overhead (with two successive calls to **imSysClock()**) and then subtract it from the measured interval.

imSysInquire

Sync

Gen-LC

Synopsis Inquire about a system attribute.

Format **long imSysInquire(System, Item, ValuePtr)**

long System; System number
long Item; Attribute about which to inquire
void* ValuePtr; Address of return value (or NULL)

Description This function inquires about an attribute of a specified system.

The **System** parameter specifies the number of the system (0, 1, 2, etc.).

The **Item** parameter specifies the attribute about which to inquire. It can be set to:

IM_SYS_NUM_NODES	Number of processing nodes.
IM_SYS_NUM_DIGITIZERS	Number of digitizers.
IM_SYS_NUM_DISPLAYS	Number of displays.
IM_SYS_NUM_SYSTEMS	Number of systems. Note that, in this case, the System parameter is ignored.

The **ValuePtr** parameter specifies the address in which to return the value of the inquired attribute. Unless otherwise stated, **ValuePtr** should be the address of a long. Note that, since **imSysInquire()** also returns this value, **ValuePtr** can be set to NULL.

Return value The returned value is the value of the inquired attribute.

imSysTimeStamp

Async

Gen-LC

Synopsis Write a time stamp (current time minus offset) to the specified buffer field.

Format **long imSysTimeStamp(Thread, Buf, Tag, Offset)**

long Thread;	Thread ID
long Buf;	Buffer ID
long Tag;	Tag of field
double Offset;	Offset to subtract from current time

Description This function writes a time stamp (current time minus offset) to the specified buffer field. Later, in a non time-critical part of your application, the time stamp can be read by calling **imBufGetFieldDouble()**.

The on-board clock value is measured in seconds as a double precision number. As the clock value is of very high resolution, this function can accurately time short intervals, such as the execution time of one function.

Similar functionality is provided by **imSysClock()**, but **imSysTimeStamp()** is more flexible because it is asynchronous. That is, it does not block Host execution. For example, you can queue a series of functions and calls to **imSysTimeStamp()** using a different field for each time stamp; once the functions are queued, Host activity will not affect timing.

The **Thread** parameter specifies the thread to which to send **imSysTimeStamp()** for execution.

The **Buf** parameter specifies the buffer into which to put the field. This buffer can be of any data type, and can have any size.

The **Tag** parameter specifies the buffer field in which to write the time stamp. Any non-zero tag value can be specified, but normally the value specified is one of the reserved user-defined fields (1 to 9999).

The **Offset** parameter specifies the offset to subtract from the current time (in seconds).

imThrAlloc

Sync

Gen-LC

Synopsis Allocate a thread.

Format **void imThrAlloc(Dev, Control, ThreadPtr)**

long Dev; Device ID
long Control; Control buffer ID (or 0)
long* ThreadPtr; Address in which to return thread ID

Description This function allocates a thread on the specified device.

The **Dev** parameter specifies the device on which to allocate the thread. Note that you allocate a device using **imDevAlloc()**.

The **Control** parameter specifies the control buffer with which to perform the function. Relevant fields for **imThrAlloc()** are listed below, with default values in bold-face. Note that if the **Control** parameter is set to 0 or if certain fields are not added to the control buffer, the default values are used. (You can later change the values of these fields, except for the stack size, using **imThrControl()**).

Field	Values	Meaning
IM_THR_ACCELERATOR	IM_ENABLE	Allow functions sent to this thread to use the NOA, when possible.
	IM_DISABLE	Don't allow functions to use the NOA.
IM_THR_MAX_PPS	IM_ALL	Allow functions sent to this thread to use all available parallel processors (if needed).
	1 - n	Limit functions to at most <i>n</i> parallel processors.
IM_THR_PRIORITY	1 - 15 (default: 8)	Thread priority (high priority threads might be serviced faster than low priority threads).
IM_THR_STACK_SIZE	4096 - 65536	Stack size of thread (in bytes).

IM_THR_VERIFY	IM_ENABLE	Verify the command parameters of functions sent to this thread.
	IM_DISABLE	Don't verify the command parameters.

The **ThreadPtr** parameter specifies the address in which to return the thread identifier. If the thread could not be allocated, 0 is returned.

imThrCancel

Async

Gen-LC

Synopsis Cancel all commands queued to a specified thread.

Format **void imThrCancel(Thread)**

long Thread; Thread ID

Description This function cancels all commands queued to a specified thread. The function does not affect the currently executing command; use **imThrHalt()** to stop it if necessary.

The **Thread** parameter specifies the thread.

imThrControl

Async

Gen-LC

Synopsis Set a thread attribute.

Format void imThrControl(Thread, Item, Value)

long Thread; Thread ID
long Item; Attribute to set
double Value; Attribute value

Description This function sets an attribute of a specified thread.

The **Thread** parameter specifies the thread.

The **Item** parameter specifies the attribute, while the **Value** parameter specifies the value for this attribute. The table below lists those attributes that can be set, and their allowable values. See **imThrAlloc()** for the default values of these attributes.

Item	Values	Meaning
IM_THR_ACCELERATOR	IM_ENABLE	Allow functions sent to this thread to use the NOA, when possible.
	IM_DISABLE	Don't allow functions to use the NOA.
IM_THR_MAX_PPS	IM_ALL	Allow functions sent to this thread to use all available parallel processors (if needed).
	1 - n	Limit functions to at most <i>n</i> parallel processors.
IM_THR_PRIORITY	1 - 15	Thread priority (high priority threads might be serviced faster than low priority threads).
IM_THR_VERIFY	IM_ENABLE	Verify the command parameters of functions sent to this thread.
	IM_DISABLE	Don't verify the command parameters.

imThrFree

Async

Gen-LC

Synopsis Free a thread.**Format** **void imThrFree(Thread)**

long Thread; Thread ID

Description This function deallocates a previously allocated thread.

The thread is only deallocated once **imThrFree()** reaches the head of its queue. Therefore, all functions sent to this thread before **imThrFree()** will execute.

The **Thread** parameter specifies the thread to deallocate.

imThrGetError

Sync

Gen-LC

Synopsis Check a thread for errors.

Format **long imThrGetError(Thread, Item, ValuePtr)**

long Thread;	Thread ID
long Item;	Error item to retrieve
void* ValuePtr;	Address in which to return error item

Description This function returns error information about the specified thread. The returned information pertains to the first error to occur in the thread since error information about the thread was last cleared. You clear error information by adding an IM_ERR_RESET flag to the **Item** parameter.

The **Thread** parameter specifies the thread on which to return error information.

The **Item** parameter specifies the type of error information to retrieve. It can be set to:

IM_ERR_CODE	The error code.
IM_ERR_MSG	The error message (maximum size of string: IM_ERR_MSG_SIZE bytes).
IM_ERR_FUNC	The name of the offending function (maximum size of string: IM_ERR_FUNC_SIZE bytes).
IM_ERR_MSG_FUNC	The error message and the name of the offending function (maximum size of string: IM_ERR_SIZE bytes).

To clear error information, add IM_ERR_RESET to the **Item** parameter (for example, IM_ERR_CODE + IM_ERR_RESET). Adding IM_ERR_RESET will simultaneously clear all error items (IM_ERR_CODE to IM_SUCCESS, and IM_ERR_MSG, IM_ERR_FUNC, and IM_ERR_MSG_FUNC to NULL). Note that you should only clear error information when retrieving the last required error item. This ensures that, at any time, all error items pertain to the same detected error.

The **ValuePtr** parameter specifies the address in which to return the error information. If you are retrieving IM_ERR_MSG, IM_ERR_FUNC, or IM_ERR_MSG_FUNC, **ValuePtr** should be the address of a character string; if you are retrieving IM_ERR_CODE, **ValuePtr** should be the address of a long.

Since **imThrGetError()** also returns the error code, **ValuePtr** can be set to NULL when you are retrieving the error code.

Return value The returned value is the error code.

Error code	Meaning
IM_SUCCESS	No error.
IM_ERR_BUFFER	Invalid buffer ID.
IM_ERR_DEVICE	Invalid device ID, or no such device.
IM_ERR_FILE	File access error.
IM_ERR_HALTED	Function halted by imThrHalt() .
IM_ERR_MEMORY	Insufficient memory to carry out the operation.
IM_ERR_NOT_PRESENT	Referenced item not present.
IM_ERR_OPCODE	Invalid opcode received.
IM_ERR_OSB	Invalid OSB ID.
IM_ERR_PARAMETER	Parameter (other than an ID-type parameter) invalid or unacceptable.
IM_ERR_RESTRICTION	Operation unable to execute due to a restriction.
IM_ERR_SYSTEM	No such system.
IM_ERR_TIMEOUT	Device unable to respond during timeout period.
IM_ERR_THREAD	Invalid thread ID.
IM_ERR_BUF_ATTRIBUTE	Unacceptable buffer attribute (size, data type, etc.).
IM_ERR_MISC	Miscellaneous error (the error message string will provide details on the cause of the error).

Example The following code checks a thread for errors, then prints the error message.

```
char Error[IM_ERR_SIZE];
if (imThrGetError(Thr, IM_ERR_MSG_FUNC + IM_ERR_RESET, Error))
    printf("%s\n", Error);
```

imThrHalt

Async

Gen-LC

Synopsis Halt the current function.

Format `void imThrHalt(Thread, Mode)`

long Thread; Thread ID

long Mode; Operation mode

Description This function halts the function currently executing on the specified thread. When halting a grab or a function that executes for several iterations, you can halt immediately or at the end of the current frame or iteration.

The **Thread** parameter specifies the thread whose current function to halt.

The **Mode** parameter specifies the mode of operation. **Mode** can be set to:

IM_NOW Halt immediately. Use this mode setting to halt any function.

IM_FRAME Halt at the end of the current frame or iteration. Use this mode setting to halt a grab or an iterating function.

When you are only performing grabs, you can set the **Mode** parameter to:

IM_URGENT Halt the current grab and also any queued grabs that have advanced too far to be caught by **imThrCancel()**. Note that if you use the IM_NOW mode when more than two grabs have been queued, there is a chance that one grab will be in a state that causes it to be unaffected by both **imThrCancel()** and **imThrHalt()**.

Note that if you set **Mode** to IM_NOW or IM_URGENT, the results of the halted function are undefined.

imThrInquire

Sync

Gen-LC

Synopsis Inquire about a thread.**Format** **long imThrInquire(Thread, Item, ValuePtr)**

long Thread; Thread ID
 long Item; Attribute about which to inquire
 void* ValuePtr; Address of return value

Description This function inquires about an attribute of a specified thread.The **Thread** parameter specifies the thread.The **Item** parameter specifies the attribute about which to inquire. It can be set to:

IM_THR_ACCELERATOR	Whether functions sent to this thread can use the NOA, if present (IM_ENABLE or IM_DISABLE).
IM_THR_MAX_PPS	The maximum number of parallel processors that functions sent to this thread are allowed to use.
IM_THR_PRIORITY	The thread's priority.
IM_THR_OWNER_ID	The ID of the device on which the thread was allocated.
IM_THR_STACK_SIZE	The stack size of the thread, in bytes.
IM_THR_VERIFY	Whether the command parameters of functions sent to this thread are verified.

The **ValuePtr** parameter specifies the address in which to return the value of the inquired attribute. Unless otherwise stated, **ValuePtr** should be the address of a long. Note that, since **imThrInquire()** also returns this value, **ValuePtr** can be set to NULL.

Return value The returned value is the value of the inquired attribute, cast to long if necessary.

imThrNop

Async

Gen-LC

Synopsis No operation.

Format **void imThrNop(Thread, OSB)**

long Thread; Thread ID
long OSB; OSB ID (or 0)

Description This function performs no operation; it is simply sent to a thread and, once it reaches the head of that thread's queue, is considered to have executed.

imThrNop0 exists for synchronization purposes. Specifically, if you want to synchronize with a function that does not accept an OSB parameter or with a function whose OSB ID you do not know, you should send **imThrNop0** immediately afterwards to the same thread this function was sent, then synchronize with the OSB passed to **imThrNop0**.

The **Thread** parameter specifies the thread to which to send **imThrNop0** for execution. For synchronization purposes, this should be the same thread as the function with which you want to synchronize.

The **OSB** parameter specifies the operation status block in which to write status information regarding this function. This parameter can be set to 0, in which case no status information is stored for this function. For synchronization purposes, however, a valid OSB identifier should be given.

Appendix A: Glossary

This appendix defines some of the specialized terms used in the Genesis documentation.

■ ALU

Arithmetic and Logic Unit. The hardware used to perform arithmetic and logical operations.

■ ASIC

Application-specific integrated circuit. A custom-made integrated circuit made to meet the requirements of a specific application by integrating several digital and/or analog functions into a single die. Integrating the functions into a single die results in a reduction in cost, board area, and power consumption, while improving performance when compared to an equivalent implementation using off-the-shelf components.

■ Asynchronous function

A function that queues its command to the hardware and then immediately returns control to the caller.

See also *synchronous function*.

■ Backplane

A circuit board that acts as a pathway between multiple Genesis boards. If a backplane is inserted between the grab ports of Genesis boards and one is inserted between the VMChannels of these boards, the boards are part of the same system and can share data through their VMChannel and grab port interface.

■ Band

One of the surfaces of a buffer. A grayscale image requires just one band. A color image requires three bands, one for each color component.

■ Bandwidth

A term describing the capacity to transfer data. Greater bandwidth is needed to sustain a higher transfer rate. Greater bandwidth can be achieved, for example, by using a wider bus.

■ Bicubic interpolation

An interpolation mode that takes a weighted average of the sixteen pixels nearest a point. The pixels closest to the point are given the most weight. Bicubic interpolation produces more accurate results than bilinear interpolation but is slower.

■ Bilinear interpolation

An interpolation mode that takes a weighted average of the four pixels nearest a point. The pixels closest to the point are given the most weight. Bilinear interpolation produces less accurate results than bicubic interpolation (it tends to blur the image slightly). However, it is faster than bicubic interpolation.

■ Binarize

To convert data to one of two values.

■ Bit

A digit of a binary number. An image is referred to as 1-bit, 8-bit, 16-bit, etc., meaning that many bits are available to store the value of each pixel in the image.

■ Broadcast

To send data to multiple memory banks at the same time. On Matrox Genesis, this can be done for data passing through the grab port and the VMChannel, but not for data passing through the PCI bus.

■ Blanking period

The portion of a video signal after the end of a line or frame, and before the beginning of a new line or frame. During this period, the video signal is "blank" so that a scan line can be brought back to the beginning of the new line or frame. The portion of a video signal after the end of a line and before the beginning of a new line is known as the *horizontal blanking period*. The portion of a video signal after the end of a frame and before the beginning of a new frame is known as the *vertical blanking period*.

■ Blob

An area of touching pixels that have the same value. Horizontally and vertically adjacent pixels are considered touching. Usually, you can specify whether diagonally adjacent pixels are considered touching. Pixels in the image that are not part of a blob make up the background.

Also known as a *connected region*.

■ Buffer pitch

The number of bytes from a pixel to its neighboring pixel on the line below. Note that a buffer's pitch is not necessarily the same as its width in bytes, since the buffer could be a child buffer or could have been allocated with some padding at the end of each line.

Also known as *line pitch* or *pitch*.

■ Byte-aligned

Describes a packed binary buffer which starts on an 8-bit boundary, that is, whose first pixel represents bit 0 of a data byte. Note that packed binary buffers are byte-aligned when allocated; the only way to have a misaligned packed binary buffer is to create a child buffer with an origin that is not a multiple of 8.

■ 'C80

A single-chip multiprocessor device that performs most of the processing on the Genesis board. It includes four parallel processors (these are advanced, 32-bit integer DSPs), a 32-bit RISC master processor with an IEEE-754 floating-point unit, and a transfer controller (this transfers data between external and internal memory). The 'C80 is much more flexible than custom ASICs or other specialized hardware because it is fully programmable.

Also known as the *TMS320C80*.

■ C-binding

The set of functions, callable from a Host C (or C++) application, available for controlling the Genesis system.

■ Child buffer

A buffer corresponding to a rectangular region within another buffer, or to a specific band of a multi-band buffer. Child buffers are therefore useful when you want to restrict processing to a rectangular region of a buffer, or to a band of a buffer.

■ Clip

To replace overflows (or underflows) in an operation with the highest (or lowest) possible value that can be held in the destination buffer of the operation.

■ Closing

A dilation followed by an erosion.

See also *opening*.

■ Color component

One of the components that make up a color space. Typically, each component of a color image is stored in a separate band of a multi-band buffer.

■ Color space

The way color information in a color image is represented. Common color spaces are RGB and HSL.

■ Composite sync

A synchronization signal made up of two components: one horizontal and one vertical.

■ Compression ratio

The ratio of the uncompressed data size of an image to its compressed data size.

■ Compute bound

Describes a function whose performance is limited strictly by the speed at which the 'C80 can process the data, and not by other factors such as how fast the data can be accessed in memory.

See also *I/O bound*.

- **Connected region**

See *blob*.

- **Contiguous memory**

A block of memory occupying a single, unbroken series of addresses.

- **Control buffer**

A buffer whose control fields specify certain options of a function. The Genesis Native Library uses control buffers because some functions have so many options that it is impractical to have these options as parameters of the function. Instead, you specify the options you want performed by adding the required control fields to a buffer and passing this buffer to the function.

- **Control field**

A field that is used to specify a certain option of a function. The option is performed by adding the field to the function's control buffer. A field holds a single value (integer or floating-point) and is identified by a unique "tag". The tag itself is just an integer value.

- **Convolution**

A neighborhood operation that determines the new value for a pixel based on the weighted sum of the pixel and the pixel's neighboring values.

- **Dilation**

A morphological operation that adds layers to objects in an image. In general, this is done by changing background pixels that touch object pixels into object pixels.

See also *erosion*.

- **Display artifacts**

Unwanted visual effects sometimes seen when the transfer of data to display memory is not synchronized with the reading of display memory by the RAMDAC.

■ Display buffer

See *main frame buffer*.

■ Double buffering

Alternating the destination of an operation between two buffers. Double buffering allows you to, for example, process one buffer while grabbing into the other buffer.

■ DSP

Digital Signal Processor: Microprocessor designed for high-speed processing of digital signals.

■ Dual-screen mode

A display configuration using two monitors; one to display images from the Genesis display memory, and another to display the Host operating system's user interface.

See also *multi-display mode* and *single-screen mode*.

■ Dynamic range

The range of values present in a buffer. An unsigned 8-bit buffer, for example, has an allowable range of 0 to 255; its dynamic range can be any range within these values.

■ Erosion

A morphological operation that peels layers from objects in an image. In general, this is done by changing object pixels that touch background pixels into background pixels.

See also *dilation*.

■ Exposure signal

The signal generated by one of the programmable timers of the grab module. The exposure signal can be used to control external hardware. For example, it can be fed to the camera to control its exposure time or used to fire a strobe light.

■ Exposure time

Refers to the period during which the image sensor of a camera is exposed to light. As the length of this period increases, so does the image brightness.

■ Field

One of the two halves that together make up the image grabbed from an interlaced camera. One half consists of the image's odd lines (known as the *odd field*); the other half consists of the image's even lines (known as the *even field*).

■ Fixed-point

A format for representing non-integer values that contains a fixed number of digits for the integer and fractional parts. A 16-bit fixed-point buffer, for example, might contain 8 integer bits and 8 fractional bits. Fixed-point buffers are a compromise between floating-point and integer buffers, since they offer the speed of integer processing with some of the precision of floating-point processing.

■ Floating-point

A format for representing numbers that contains two parts: a mantissa and an exponent. The mantissa specifies the digits in the number, while the exponent expresses the magnitude of the number. This format provides a constant number of significant digits of precision over a very large dynamic range. Floating-point buffers take longer to process than integer buffers.

■ Frame

A single image grabbed from a video camera.

■ Gain level

The factor by which an analog input signal is scaled. The gain affects the brightness and contrast of the resulting image.

■ Gain and offset correction

To offset and multiply each pixel in an image by specified values:

*new pixel value = (old pixel value - offset) * gain.*

The offset and gain values can be constant for the whole image, or they can be different for each pixel. The latter can be useful when performing shading corrections.

■ Geometric operation

A processing operation that repositions pixels in an image.

■ Grab

To acquire an image from a camera.

■ Histogram

A statistical operation that measures the frequency with which each pixel value occurs in an image.

■ Histogram equalization

A point-to-point operation that changes each pixel value in an image so as to reshape the image's histogram in a specified way. A histogram equalization operation can be used to improve the contrast or brightness of an image.

■ Horizontal blanking period

The portion of a video signal after the end of a line and before the beginning of a new line. During this period, the video signal is "blank".

See also *vertical blanking period*.

■ Horizontal sync

The part of a video signal that indicates the end of a line and the start of a new one.

See also *vertical sync*.

■ HSL

A color space that represents color using components of hue, saturation, and luminance. The hue component describes the actual color of a pixel. The saturation component describes the concentration of that color. The luminance component describes the combined brightness of the primary colors.

■ In-place operation

Describes a processing operation in which the results overwrite one of the source buffers.

■ Interlaced scanning

Describes a transfer of data in which the odd-numbered lines of the source are written to the destination buffer first, and then the even-numbered lines (or vice-versa).

See also *progressive scanning*.

■ Interpolation

A neighborhood operation that estimates the intensity at a point in an image between pixel positions. To estimate the intensity, the operation takes a weighted-sum of the point's neighboring pixel values. Two common interpolation modes are *bicubic interpolation* and *bilinear interpolation*.

■ I/O bound

Describes a function whose performance is limited by the speed at which it can access data in memory.

See also *compute bound*.

■ JPEG

Joint Photographic Experts Group. A standard for compressing images.

■ Kernel

The set of numbers that are used by a neighborhood operation to determine new pixel values. The type of neighborhood operation determines how the kernel is used.

Also known as a *structuring element* (particularly for morphological operations).

■ Keying

A display effect that switches between two display sources depending on the pixel values in one of the sources. On Genesis, keying is usually used to make portions of the overlay frame buffer transparent so that corresponding areas of the main frame buffer can show through it.

■ Latency

The time from when an operation is started to when the final result is produced.

■ Line pitch

See *buffer pitch*.

■ Live processing

See *real-time processing*.

■ LUT mapping

Look-up table mapping. A point-to-point operation that uses a table to define a replacement value for each possible pixel value in an image.

■ Main frame buffer

The buffer whose contents are displayed by the display section of Matrox Genesis. If keying is enabled, those areas of the overlay frame buffer that have a specified color allow the main frame buffer to show through.

Also known as the *display buffer*.

■ Message

The operation code and its various optional parameters that a C-binding function sends to the board so that the board can execute the function.

■ MGA

Matrox Graphics Architecture. As part of Matrox Genesis's display section, it allows you to draw into the overlay buffer using the graphics functions of the Host operating system.

■ Morphological operation

A neighborhood operation that determines the new value for a pixel based on the results of a comparison between the pixel's neighborhood and the operation's kernel, or based on the extreme values in the pixel's neighborhood.

■ Multi-display mode

A multi-board configuration that uses Genesis boards and/or MGA Millennium boards to create one large desktop on two, three, or four screens.

■ **Multi-processing**

Executing two or more operations in parallel.

Also known as *parallel processing*.

■ **Neighborhood operation**

A processing operation that replaces a pixel's value according to the values of its surrounding pixels (called its neighborhood). The size of the neighborhood is determined by the operation's kernel. The type of operation determines how the new pixel value is determined. Convolutions and morphological operations are two types of neighborhood operations.

■ **NOA**

Neighborhood Operations Accelerator. A Matrox-designed ASIC that can accelerate neighborhood operations such as convolutions and morphology.

■ **Node**

The basic building block of a Genesis system; it consists of the TMS320C80 (C80), the VIA, and processing memory. A node can also include a NOA.

■ **Normalized grayscale correlation**

A neighborhood operation that determines the new value for a pixel (r), based on a specified kernel (model):

$$r = \frac{N \sum IM - (\sum I) \sum M}{\sqrt{[N \sum I^2 - (\sum I)^2][N \sum M^2 - (\sum M)^2]}}$$

where M = the value of a model pixel and I = the value of the underlying image pixel. Note that the above equation reaches its maximum value of 1 where the image and model match exactly, gives 0 where the image and model are uncorrelated, and is negative where the similarity is less than might be expected by chance (reaching -1 when the image is a negative version of the model). Normalized grayscale correlation is widely used in industry for pattern matching applications.

■ Normalization

Adjusting the results of a processing operation so that they have the correct magnitude. After multiplying an image by a fixed-point integer, for example, normalization is needed to right-shift results to remove the fractional bits.

■ Off-screen display memory

Memory that is allocated in the main or overlay frame buffer (in Matrox Genesis's display section) that is not visible on the screen.

■ Opening

An erosion followed by a dilation.

See also *closing*.

■ Operand

One of the terms of an arithmetic or logical operation. In the arithmetic operation $A + B$, for example, the operands are A and B. In the Genesis Native Library, one of the operands of an arithmetic or logical operation must be a buffer; the other(s) can be buffers or constants. Note that the buffers can hold any type of data, for example, image data, LUT values, and kernel values.

■ Overflows

Results of a processing operation that are above the range of the destination buffer. For example, in an unsigned 8-bit destination buffer, overflows are those results above 255.

See also *underflows*.

■ Overlay frame buffer

The buffer used to annotate the main frame buffer. On Genesis, portions of the overlay frame buffer that have a specified color allow the corresponding areas of the main frame buffer to show through (if keying is enabled). Note that, in single-screen mode, the overlay frame buffer is also used to display the Host operating system's user interface.

■ Parallel processing

See *multi-processing*.

■ **Pitch**

See *buffer pitch*.

■ **Point-to-point operation**

A processing operation that does not use a pixel's neighbors when determining the pixel's new value. Examples of point-to-point operations are LUT mappings, arithmetic operations, and logical operations.

■ **Processing operation**

An operation that results in a new image. Examples of processing operations are geometric operations, point-to-point operations, and neighborhood operations.

See also *statistical operation*.

■ **Progressive scanning**

Describes a transfer of data in which the lines of the source are written sequentially into the destination buffer.

See also *interlaced scanning*.

■ **RAMDAC**

Random Access Memory Digital-to-Analog Converter: A chip that converts data from digital to analog so that it can be displayed on a monitor. The RAMDAC can also implement various display effects.

■ **Rank filter operation**

A neighborhood operation that sorts a pixel's neighborhood values in increasing order, and then replaces the pixel's value with the *n*th highest value in the list. A *median filter* is a type of rank filter that uses the middle value in the list.

■ **Real-time processing**

The processing of an image as quickly as the next image is grabbed.

Also known as *live processing*.

■ Reference levels

The zero and full-scale levels of an analog-to-digital converter. Voltages below a *black reference level* are converted to a zero pixel value; voltages above a *white reference level* are converted to the maximum pixel value. Together with the analog gain factor, the reference levels affect the brightness and contrast of the resulting image.

■ RGB

A color space that represents color using the primary colors (red, green, and blue) as components.

■ RISC

Reduced Instruction Set Computing. A microprocessor design that focuses on efficiently processing a small set of instructions.

■ ROI

Region of interest. The area of a buffer that is processed. The region of interest can be the entire buffer or a rectangular portion of the buffer.

■ Run

A horizontal sequence of consecutive pixels with the same value. Often used in blob analysis, since each blob can be efficiently described as a list of runs.

■ Saturate

To replace overflows (or underflows) in an operation with the highest (or lowest) possible value that can be held in the destination buffer of the operation.

■ Scalability

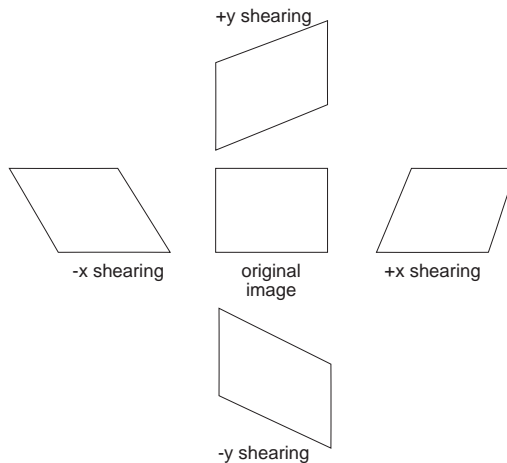
Describes a board whose configuration is designed to include additional modules, if desired. The Genesis main board, for example, can include a display section and/or grab module. In addition, one or more processor boards can be added to increase performance.

■ SDRAM

Synchronous Dynamic Random Access Memory. A type of memory used for processing. SDRAM allows the 'C80 to access data as fast as possible, which is important for I/O-bound functions.

■ Shearing

A geometric operation that translates pixels along only one axis, by an amount proportional to the distance from that axis (see below).



■ Signed

Describes a buffer that can have negative values. A signed 8-bit buffer, for example, has values between -128 and 127.

See also *unsigned*.

■ Sign-extension

To extend a value from one data type to a larger data type by copying the sign bit of the source type to all the higher bits of the destination (that is, by copying 1's if the value is negative; 0's if the value is positive).

See also *zero-extension*.

■ **Single-screen mode**

A display configuration using a single monitor to display both the Host operating system's user interface and images from the Genesis display memory.

See also *dual-screen mode* and *multi-display mode*.

■ **Spatial filtering operation**

See *convolution*.

■ **Statistical operation**

An operation that extracts information from an image. A histogram is an example of a statistical operation.

See also *processing operation*.

■ **Structuring element**

See *kernel*.

■ **Synchronous function**

A function that does not return control to the caller until it has finished executing.

See also *asynchronous function*.

■ **System**

A group of Genesis boards (main board(s) and/or processor board(s)) connected to each other by the grab port and the VM port.

■ **Temporal filtering**

An operation that takes a weighted sum of the currently grabbed frame and the previous output of the filter operation. Temporal filtering is often used to remove the effects of random noise because it acts as an averaging filter.

■ **Thickening**

A morphological operation that converts background pixels into object pixels when the neighborhood exactly matches a kernel. Thickening is similar to dilation except that it is more selective because, when iterated, it will not convert all pixels to object pixels. Instead, it will eventually reach a steady state (known as *idempotence*).

■ Thinning

A morphological operation that converts object pixels into background pixels when the neighborhood exactly matches a kernel. Thinning is similar to erosion except that it is more selective because, when iterated, it will not convert all pixels to background pixels. Instead, it will eventually reach a steady state (known as *idempotence*).

■ Thread

An execution queue. In the Genesis Native Library, all functions are sent to a specified thread, and execute on the node associated with this thread. Threads execute independently of one another, allowing operations to run in parallel.

■ Threshold

A point-to-point operation that converts pixels whose values are above, below, and/or within a specified range, to a specified value.

■ TMS320C80

See *'C80*.

■ Translation

A geometric operation that displaces an image vertically and/or horizontally.

■ Underflows

Results of a processing operation that are below the range of the destination buffer. For example, in an unsigned 8-bit destination buffer, underflows are those results below 0.

See also *overflows*.

■ Unsigned

Describes a buffer that can have only positive values. An unsigned 8-bit buffer, for example, has values between 0 and 255.

See also *signed*.

■ Vertical blanking period

The portion of a video signal after the end of a frame and before the beginning of a new frame. During this period, the video signal is "blank".

See also *horizontal blanking period*.

■ Vertical sync

The part of a video signal that indicates the end of a frame and the start of a new one.

See also *horizontal sync*.

■ VIA

Video Interface ASIC. A custom ASIC that connects all the data buses on Matrox Genesis (the grab, VMChannel, 'C80 and PCI bus) to one another, and directs and monitors data flow "traffic" throughout the system. It is a video interface that provides various ways of inputting and outputting data.

■ VMChannel

Vesa Media Channel. An industry standard 32-bit bus designed for carrying video data. On Genesis, it is used primarily to copy images between nodes or from processing to display memory.

■ WRAM

Window Random Access Memory. A type of dual-ported memory used for displays.

■ Zero-extension

To extend a value from one data type to a larger data type by copying 0's into all the higher bits of the destination.

See also *sign-extension*.

Appendix B: Examples

This appendix gives the complete source code of each example referenced in this manual. To compile these examples, refer to the `readme.txt` file in the `\GENESIS\DOC` directory. Note that there might be more up-to-date or other examples in the `\GENESIS\EXAMPLES` directory.

blob.c

```

/*****
 *
 * Demonstrate the use of the BLOB module.
 *
 * (Note that if you are running in single screen mode under Windows,
 * you will not see anything in the Genesis image buffer until you
 * enable keying with a separate program).
 *
 *****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "imapi.h"

/* Maximum blobs to draw */
#define MAX_BLOBS 20

/* List of supported functions */
#define COUNT      0
#define BOX        1
#define FILTER      2

void main(int argc, char **argv)
{
    long Device;           /* Genesis device */
    long Thread;           /* Thread to execute all
                           * functions */
    long IdentBuf;         /* Blob identifier image */
    long DispBuf;         /* Display buffer */
    long FeatList;         /* Blob feature list */
    long Result;           /* Blob result buffer */
    long SizeX = 512, SizeY = 512; /* Image Size */
    long Func = 0;         /* The function to use */
    char Error[IM_ERR_SIZE]; /* String to hold error message */
    long i;

    /* Check arguments */
    if (argc < 2 || *argv[1] == '?')
    {
        printf("Usage: BLOB func\n");
        printf("func = %2d Count number of blobs\n", COUNT);
        printf("      %2d Find bounding box of each blob\n", BOX);
        printf("      %2d Filter unwanted blobs\n", FILTER);
        exit(1);
    }
    if (argc > 1)
        sscanf(argv[1], "%li", &Func);

```

```

/* Allocate a device and a thread */
imDevAlloc(0, 0, NULL, IM_DEFAULT, &Device);
imThrAlloc(Device, 0, &Thread);

/* Allocate the blob identifier image */
imBufAlloc2d(Thread, SizeX, SizeY, IM_UBYTE, IM_PROC, &IdentBuf);

/* Allocate a full-screen display buffer and clear it */
imBufChild(Thread, IM_DISP, 0, 0, IM_ALL, IM_ALL, &DispBuf);
imBufClear(Thread, DispBuf, 0, 0);

/* Draw random blobs in the identifier image */
imBufClear(Thread, IdentBuf, 0, 0);
imBufPutField(Thread, IdentBuf, IM_GRA_COLOR, 150);
for (i = 0; i < MAX_BLOBS; i++)
{
    imGraArcFill(Thread, IdentBuf, IdentBuf, rand() % SizeX,
                  rand() % SizeY, rand() % 30 + 15,
                  rand() % 30 + 15, 0, 360);
}

/* Copy the image to the display */
imBufCopy(Thread, IdentBuf, DispBuf, 0, 0);

/* Perform the selected processing operation */
switch (Func)
{
    case COUNT:
    {
        printf("Count the number of blobs...\n");

        /* Allocate a blob feature list and result buffer */
        imBlobAllocFeatureList(Thread, &FeatList);
        imBlobAllocResult(Thread, &Result);

        /* Increase speed by not saving runs */
        imBlobControl(Thread, Result, IM_BLOB_SAVE_RUNS, IM_DISABLE);

        /* Count the blobs */
        imBlobCalculate(Thread, IdentBuf, 0, FeatList, Result,
                        IM_CLEAR, 0);

        /* Get the number */
        printf("There are %li blobs\n", imBlobGetNumber(Thread,
                                                         Result, NULL));

        /* Free the feature list and result buffer */
        imBlobFree(Thread, FeatList);
        imBlobFree(Thread, Result);

        break;
    }
}

```

```

case BOX:
{
    printf("Find the bounding box of each blob...\n");

    long Number;          /* Number of blobs */
    IM_BLOB_GROUP1_ST *Group1; /* Results */

    /* Allocate a blob feature list and result buffer */
    imBlobAllocFeatureList(Thread, &FeatList);
    imBlobAllocResult(Thread, &Result);

    /* Select box feature for calculation */
    imBlobSelectFeature(Thread, FeatList, IM_BLOB_BOX,
                        IM_DEFAULT);

    /* Increase speed by not saving runs */
    imBlobControl(Thread, Result, IM_BLOB_SAVE_RUNS, IM_DISABLE);

    /* Calculate selected features */
    imBlobCalculate(Thread, IdentBuf, 0, FeatList, Result,
                    IM_CLEAR, 0);

    /* Get the number of blobs */
    imBlobGetNumber(Thread, Result, &Number);

    /* Allocate enough memory for the results */
    Group1 = (IM_BLOB_GROUP1_ST *)
        malloc(Number * sizeof(IM_BLOB_GROUP1_ST));

    /* Get the results */
    imBlobGetResult(Thread, Result, IM_BLOB_GROUP1,
                    IM_DEFAULT, Group1);

    /* Mark the bounding boxes */
    for (i = 0; i < Number; i++)
        imGraRect(Thread, 0, IdentBuf, Group1[i].box_x_min,
                    Group1[i].box_y_min, Group1[i].box_x_max,
                    Group1[i].box_y_max);

    /* Display the result */
    imBufCopy(Thread, IdentBuf, DispBuf, 0, 0);

    /* Free the feature list and result buffer */
    free(Group1);
    imBlobFree(Thread, FeatList);
    imBlobFree(Thread, Result);

    break;
}

```



```

case FILTER:
{
    printf("Find convex blobs that don't touch the edge of
           the image...\n");

    /* Allocate a blob feature list and result buffer */
    imBlobAllocFeatureList(Thread, &FeatList);
    imBlobAllocResult(Thread, &Result);

    /* Select the required features for calculation */
    imBlobSelectFeature(Thread, FeatList, IM_BLOB_BOX,
                        IM_DEFAULT);
    imBlobSelectFeature(Thread, FeatList,
                        IM_BLOB_ROUGHNESS, IM_DEFAULT);

    /* Calculate selected features */
    imBlobCalculate(Thread, IdentBuf, 0, FeatList, Result,
                    IM_CLEAR, 0);

    /* Exclude blobs that touch any edge of the image */
    imBlobSelect(Thread, Result, IM_EXCLUDE,
                IM_BLOB_BOX_X_MIN, IM_DEFAULT, IM_EQUAL, 0, 0);
    imBlobSelect(Thread, Result, IM_EXCLUDE,
                IM_BLOB_BOX_X_MAX, IM_DEFAULT, IM_EQUAL,
                SizeX - 1, 0);
    imBlobSelect(Thread, Result, IM_EXCLUDE,
                IM_BLOB_BOX_Y_MIN, IM_DEFAULT, IM_EQUAL, 0, 0);
    imBlobSelect(Thread, Result, IM_EXCLUDE,
                IM_BLOB_BOX_Y_MAX, IM_DEFAULT, IM_EQUAL,
                SizeY - 1, 0);

    /* Exclude blobs that are too rough */
    imBlobSelect(Thread, Result, IM_EXCLUDE,
                IM_BLOB_ROUGHNESS, IM_DEFAULT, IM_GREATER,
                1.04, 0);

    /* Fill the two groups of blobs with different colours */
    imBlobFill(Thread, Result, IdentBuf, IM_EXCLUDED_BLOBS,
                150, 0);
    imBlobFill(Thread, Result, IdentBuf, IM_INCLUDED_BLOBS,
                255, 0);

    /* Display the result */
    imBufCopy(Thread, IdentBuf, DispBuf, 0, 0);

    /* Free the feature list and result buffer */
    imBlobFree(Thread, FeatList);
    imBlobFree(Thread, Result);

    break;
}

default:
    printf("Unsupported function\n");
    break;
}

```

```
/* Wait for everything to finish, then check for errors */
imSyncHost(Thread, 0, IM_COMPLETED);
if (imAppGetError(IM_ERR_MSG_FUNC, Error))
    printf("%s\n", Error);

/* Clean up */
imBufFree(Thread, IdentBuf);
imBufFree(Thread, DispBuf);
imThrFree(Thread);
imDevFree(Device);
}
```

first.c

```

/*****
 *
 * A very simple Genesis program.
 * If anything goes wrong, an error message will be printed.
 *
 * (Note that if you are running in single screen mode under
 * Windows, you will not see anything in the Genesis image buffer
 * until you enable keying with a separate program).
 *
 *****/

#include <stdio.h>

#include "imapi.h"

/* Prototype for error handler function */
void ErrHandler(void *Param);

void main(void)
{
    long Device;          /* Genesis device */
    long Thread;          /* Thread to execute all functions */
    long ProcBuf;         /* Buffer allocated in processing memory */
    long DispBuf;         /* Buffer allocated in display memory */
    long Success = 1;     /* Flag to record success or failure */

    printf("Allocating the Genesis system...\n");

    /* Establish an error handler */
    imAppCatchError(IM_DEFAULT, ErrHandler, (void *) &Success);

    /* Allocate the board and a thread */
    imDevAlloc(0, 0, NULL, IM_DEFAULT, &Device);
    imThrAlloc(Device, 0, &Thread);

    /* Allocate a full screen display buffer and clear it */
    imBufChild(Thread, IM_DISP, 0, 0, IM_ALL, IM_ALL, &DispBuf);
    imBufClear(Thread, DispBuf, 0, 0);

    /* Allocate a processing buffer */
    imBufAlloc2d(Thread, 512, 512, IM_UBYTE, IM_PROC, &ProcBuf);

```

```

/* Clear the buffer and then write text in it */
imBufClear(Thread, ProcBuf, 0, 0);
imGraRect(Thread, 0, ProcBuf, 180, 230, 335, 285);
imGraText(Thread, 0, ProcBuf, 200, 250, "Matrox Genesis");

/* Copy it to the display */
imBufCopy(Thread, ProcBuf, DispBuf, 0, 0);

/* Synchronize to give all errors a chance to be reported */
imSyncHost(Thread, 0, IM_COMPLETED);

/* If no errors occurred, report success */
if (Success)
    printf("Completed successfully\n");

/* Clean up */
imBufFree(Thread, DispBuf);
imBufFree(Thread, ProcBuf);
imThrFree(Thread);
imDevFree(Device);
}

void ErrHandler(void *Success)
{
    char Error[IM_ERR_SIZE];
    /*
     * Get the error message and print it. Don't reset the error
     * because we only want the first to be printed.
     */
    imAppGetError(IM_ERR_MSG_FUNC, Error);
    printf("%s\n", Error);

    /* Record that the error occurred */
    *(long *)Success = 0;
}

```

grab.c

```

/*****
 *
 * Grab an image, and optionally save it.
 *
 * (Note that if you are running in single screen mode under
 * Windows, you will not see anything in the Genesis image buffer
 * until you enable keying with a separate program).
 *
 *****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "imapi.h"

void main(int argc, char **argv)
{
    long Device;          /* Genesis device */
    long Thread;          /* Thread to execute all functions */
    long DispBuf;         /* Buffer allocated in display memory */
    long ScreenBuf;       /* Display buffer full size of screen */
    long Camera;          /* Camera */
    long SizeX, SizeY;     /* Image Size */
    char Error[IM_ERR_SIZE]; /* String to hold error message */
    int Save = 0, i;       /* Miscellaneous variables */

    /* Check arguments */
    if (argc > 1 && *argv[1] == '?')
    {
        printf("Usage: GRAB [file.tif] [-x] [-y]\n");
        printf("      -x size\t Image X size\n");
        printf("      -y size\t Image Y size\n");
        exit(1);
    }

    if (argc > 1 && *argv[1] != '.')
        Save = 1;

    /* Allocate the board, a thread and a camera */
    imDevAlloc(0, 0, NULL, IM_DEFAULT, &Device);
    imThrAlloc(Device, 0, &Thread);
    imCamAlloc(Thread, NULL, IM_DEFAULT, &Camera);

    /* Determine the image size */
    imCamInquire(Thread, Camera, IM_DIG_SIZE_X, &SizeX);
    imCamInquire(Thread, Camera, IM_DIG_SIZE_Y, &SizeY);

```

```

/* Check if the user requested a different size */
for (i = 1; i < argc; i++)
{
    if (!strcmp(argv[i], "-x"))
        sscanf(argv[i+1], "%li", &SizeX);
    else if (!strcmp(argv[i], "-y"))
        sscanf(argv[i+1], "%li", &SizeY);
}

printf("Image size is %lix%li\n", SizeX, SizeY);

/* Allocate a full screen display buffer and clear it */
imBufChild(Thread, IM_DISP, 0, 0, IM_ALL, IM_ALL, &ScreenBuf);
imBufClear(Thread, ScreenBuf, 0, 0);

/* Allocate a buffer at a specific location on the display */
imBufChild(Thread, IM_DISP, 0, 0, SizeX, SizeY, &DispBuf);

/* Start a continuous grab into the display buffer */
imDigGrab(Thread, 0, Camera, DispBuf, IM_CONTINUOUS, 0, 0);

/* Halt when the user hits Enter */
printf("Press <Enter> to stop");
getchar();
imThrHalt(Thread, IM_FRAME);

/* Optionally save the image */
if (Save)
{
    if (imCamInquire(Thread, Camera, IM_DIG_NUM_BANDS, NULL) == 3)
    {
        /* Save all bands if colour */
        imBufSave(Thread, argv[1], IM_TIFF, DispBuf);
    }
    else
    {
        /* Save just the first band of the display if not colour */
        long MonoBuf;

        imBufChildBand(Thread, DispBuf, 0, &MonoBuf);
        imBufSave(Thread, argv[1], IM_TIFF, MonoBuf);
        imBufFree(Thread, MonoBuf);
    }
}

/* Wait for everything to finish, then check for errors */
imSyncHost(Thread, 0, IM_COMPLETED);
if (imAppGetError(IM_ERR_MSG_FUNC, Error))
    printf("%s\n", Error);

/* Clean up */
imCamFree(Thread, Camera);
imBufFree(Thread, DispBuf);
imBufFree(Thread, ScreenBuf);
imThrFree(Thread);
imDevFree(Device);
}

```

jpeg.c

```

/*****
 *
 * Demonstrate the use of the JPEG module.
 *
 * (Note that if you are running in single screen mode under Windows,
 * you will not see anything in the Genesis image buffer until you
 * enable keying with a separate program).
 *
 *****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "imapi.h"

/* List of supported functions */
#define COMPRESS      0
#define DECOMPRESS   1

void main(int argc, char **argv)
{
    long Device;          /* Genesis device */
    long Thread;          /* Thread to execute all functions */
    long ImageBuf;        /* Uncompressed image */
    long JpegBuf;         /* Compressed image */
    long DispBuf;         /* Display buffer */
    long Func;            /* The function to use */
    char Error[IM_ERR_SIZE]; /* String to hold error message */
    char InFile[100];     /* Name of input image file */
    char OutFile[100];    /* Name of output image file */

    /* Check arguments */
    if (argc < 3 || *argv[1] == '?')
    {
        printf("Usage: JPEG infile outfile [func]\n");
        printf("func = %2d Load TIFF file, compress, and save JPEG\n", COMPRESS);
        printf("      %2d Load JPEG file, decompress, and save TIFF\n", DECOMPRESS);
        printf("      (default is determined from file types)\n");
        printf("\n");
        printf("Ex.    JPEG file.tif file.jpg will compress\n");
        printf("      JPEG file.jpg jpeg.tif will decompress\n");
        exit(1);
    }
    strcpy(InFile, argv[1]);
    strcpy(OutFile, argv[2]);

```

```

/* Determine whether to compress or decompress */
if (argc > 3)
    sscanf(argv[3], "%li", &Func);
else
{
    if ((strstr(InFile, ".tif") || strstr(InFile, ".TIF")) &&
        (strstr(OutFile, ".jpg") || strstr(OutFile, ".JPG")))
        Func = COMPRESS;
    else if ((strstr(InFile, ".jpg") || strstr(InFile, ".JPG")) &&
             (strstr(OutFile, ".tif") || strstr(OutFile, ".TIF")))
        Func = DECOMPRESS;
    else
    {
        printf("Cannot determine what to do from file names\n");
        exit(1);
    }
}

/* Allocate a device and a thread */
imDevAlloc(0, 0, NULL, IM_DEFAULT, &Device);
imThrAlloc(Device, 0, &Thread);

/* Perform the selected processing operation */
switch (Func)
{
    case COMPRESS:
    {
        printf("Load TIFF file, compress, and save JPEG file\n");

        /* Allocate a JPEG buffer */
        imJpegAlloc(Thread, 0, &JpegBuf);

        /* Select lossless mode */
        imJpegControl(Thread, JpegBuf, IM_JPEG_MODE, IM_LOSSLESS);

        /* Load the uncompressed image into a processing buffer */
        imBufRestore(Thread, InFile, IM_TIFF, IM_PROC, &ImageBuf);

        /* Compress the image */
        imJpegEncode(Thread, ImageBuf, JpegBuf, 0);

        /* Save the compressed image */
        imJpegSave(Thread, OutFile, JpegBuf);

        /* Free the JPEG buffer */
        imJpegFree(Thread, JpegBuf);

        break;
    }
}

```



```

case DECOMPRESS:
{
    printf("Load JPEG file, decompress, and save TIFF file\n");

    /* Load the compressed image into a JPEG buffer */
    imJpegRestore(Thread, InFile, &JpegBuf);

    /* Allocate a processing buffer of the same size */
    imBufAlloc(Thread, imJpegInquire(Thread, JpegBuf,
                                      IM_JPEG_SIZE_X, NULL),
               imJpegInquire(Thread, JpegBuf,
                              IM_JPEG_SIZE_Y, NULL),
               imJpegInquire(Thread, JpegBuf,
                              IM_JPEG_NUM_BANDS, NULL),
               imJpegInquire(Thread, JpegBuf,
                              IM_JPEG_TYPE, NULL),
               IM_PROC, &ImageBuf);

    /* Decompress the image */
    imJpegDecode(Thread, ImageBuf, JpegBuf, 0);

    /* Save the decompressed image */
    imBufSave(Thread, OutFile, IM_TIFF, ImageBuf);

    /* Free the JPEG buffer */
    imJpegFree(Thread, JpegBuf);

    break;
}

default:
    printf("Unsupported function\n");
    break;
}

/* Allocate a full screen display buffer and clear it */
imBufChild(Thread, IM_DISP, 0, 0, IM_ALL, IM_ALL, &DispBuf);
imBufClear(Thread, DispBuf, 0, 0);

/* Copy the uncompressed image to the display */
imBufCopy(Thread, ImageBuf, DispBuf, 0, 0);

/* Wait for everything to finish, then check for errors */
imSyncHost(Thread, 0, IM_COMPLETED);
if (imAppGetError(IM_ERR_MSG_FUNC, Error))
    printf("%s\n", Error);

/* Clean up */
imBufFree(Thread, ImageBuf);
imBufFree(Thread, DispBuf);
imThrFree(Thread);
imDevFree(Device);
}

```

pat.c

```

/*****
 *
 * Demonstrate the use of the PAT module.
 *
 * (Note that if you are running in single screen mode under Windows,
 * you will not see anything in the Genesis image buffer until you
 * enable keying with a separate program).
 *
 *****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "imapi.h"

/* List of supported functions */
#define SAVE      0
#define RESTORE   1

void main(int argc, char **argv)
{
    long Device;          /* Genesis device */
    long Thread;          /* Thread to execute all functions */
    long ImageBuf;        /* Model or target image */
    long DispBuf1;        /* Display buffer */
    long DispBuf2;        /* Display buffer */
    long ScreenBuf;       /* Display buffer full size of screen */
    long Model;           /* Pattern matching model */
    long Result;          /* Pattern matching result buffer */
    long SizeX, SizeY;     /* Image Size */
    long Func = 0;        /* The function to use */
    char Error[IM_ERR_SIZE]; /* String to hold error message */
    char *ImageFile = "board.mim"; /* Name of image file */
    char *ModelFile = "model.mod"; /* Name of model file */
    long ModelOffX = 272, ModelOffY = 112;
    long ModelSizeX = 128, ModelSizeY = 128;

    /* Check arguments */
    if (argc < 2 || *argv[1] == '?')
    {
        printf("Usage: PAT func\n");
        printf("func = %2d Define and save model\n", SAVE);
        printf("      %2d Restore and find model\n", RESTORE);
        exit(1);
    }
    if (argc > 1)
        sscanf(argv[1], "%li", &Func);

    /* Allocate a device and a thread */
    imDevAlloc(0, 0, NULL, IM_DEFAULT, &Device);
    imThrAlloc(Device, 0, &Thread);

```

```

/* Load the image into a processing buffer */
imBufRestore(Thread, ImageFile, IM_TIFF, IM_PROC, &ImageBuf);

/* Allocate two display buffers of the same size */
imBufInquire(Thread, ImageBuf, IM_BUF_SIZE_X, &SizeX);
imBufInquire(Thread, ImageBuf, IM_BUF_SIZE_Y, &SizeY);
imBufChild(Thread, IM_DISP, 0, 0, SizeX, SizeY, &DispBuf1);
imBufChild(Thread, IM_DISP, SizeX, 0, SizeX, SizeY, &DispBuf2);

/* Allocate a full screen display buffer and clear it */
imBufChild(Thread, IM_DISP, 0, 0, IM_ALL, IM_ALL, &ScreenBuf);
imBufClear(Thread, ScreenBuf, 0, 0);

/* Copy the image to the display */
imBufCopy(Thread, ImageBuf, DispBuf1, 0, 0);

/* Perform the selected processing operation */
switch (Func)
{
    case SAVE:
    {
        printf("Define model and save\n");

        /* Allocate a pattern matching model */
        imPatAllocModel(Thread, ImageBuf, ModelOffX,
                        ModelOffY, ModelSizeX,
                        ModelSizeY, IM_NORMALIZED, &Model);

        /* Preprocess the model for a faster search */
        imPatPreprocModel(Thread, 0, Model, IM_DEFAULT, 0);

        /* Select medium speed and high accuracy */
        imPatSetSpeed(Thread, Model, IM_MEDIUM);
        imPatSetAccuracy(Thread, Model, IM_HIGH);

        /* Save the model */
        imPatSave(Thread, ModelFile, Model);

        /* Free the model */
        imPatFree(Thread, Model);

        break;
    }

    case RESTORE:
    {
        printf("Restore model and find in target image\n");

        long TempBuf;           /* Temporary buffer */
        IM_PAT_RESULT_ST Res;    /* All match results */
        IM_PAT_INQUIRE_ST Inq;  /* All model parameters */

        /* Restore the model */
        imPatRestore(Thread, ModelFile, &Model);
    }
}

```

```

/* Inquire all parameters */
imPatInquire(Thread, Model, IM_ALL, &Inq);

/* Allocate a pattern matching result buffer */
imPatAllocResult(Thread, 1, &Result);

/* Search for the model */
imPatFindModel(Thread, ImageBuf, Model, Result, 0);

/* Get all results */
imPatGetResult(Thread, Result, IM_ALL, &Res);

/* Check if a match was found */
if (Res.number == 0)
{
    printf("Model could not be found\n");
}
else
{
    /* Print the match position and score */
    printf("Model found at (%.2f, %.2f) with score of
           %.1f%%\n", Res.position_x, Res.position_y,
           Res.score);

    /* Mark the match position */
    imGraLine(Thread, 0, ImageBuf, (long) (Res.position_x-10),
              (long) (Res.position_y),
              (long) (Res.position_x+10),
              (long) (Res.position_y));
    imGraLine(Thread, 0, ImageBuf, (long) (Res.position_x),
              (long) (Res.position_y-10),
              (long) (Res.position_x),
              (long) (Res.position_y+10));
    imGraRect(Thread, 0, ImageBuf,
              (long) (Res.position_x - Inq.center_x),
              (long) (Res.position_y - Inq.center_y),
              (long) (Res.position_x + Inq.center_x +
              Inq.size_x),
              (long) (Res.position_y - Inq.center_y +
              Inq.size_y));
    imBufCopy(Thread, ImageBuf, DispBuf1, 0, 0);
}

/* Copy the model to a temporary buffer, then to the display */
imBufAlloc2d(Thread, Inq.size_x, Inq.size_y, IM_UBYTE, IM_PROC,
             &TempBuf);
imPatCopy(Thread, Model, TempBuf, IM_DEFAULT, 0);
imBufCopy(Thread, TempBuf, DispBuf2, 0, 0);
imBufFree(Thread, TempBuf);

/* Free the model and result buffer */
imPatFree(Thread, Model);
imPatFree(Thread, Result);

break;
}

```

```
        default:
            printf("Unsupported function\n");
            break;
    }

    /* Wait for everything to finish, then check for errors */
    imSyncHost(Thread, 0, IM_COMPLETED);
    if (imAppGetError(IM_ERR_MSG_FUNC, Error))
        printf("%s\n", Error);

    /* Clean up */
    imBufFree(Thread, ImageBuf);
    imBufFree(Thread, DispBuf1);
    imBufFree(Thread, DispBuf2);
    imBufFree(Thread, ScreenBuf);
    imThrFree(Thread);
    imDevFree(Device);
}
```

process.c

```

/*****
 *
 * Load a TIFF file and perform a variety of processing operations.
 * Most operations will work on either monochrome or colour images.
 *
 * The purpose is simply to illustrate the usage of processing functions
 * that are non-trivial to use for the first time without an example.
 *
 * (Note that if you are running in single screen mode under Windows,
 * you will not see anything in the Genesis image buffer until you
 * enable keying with a separate program).
 *
 *****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "imapi.h"

/* Prototype for error handler function */
void ErrHandler(void *Param);

/* List of supported functions */
#define CONVOLVE 0
#define ROTATE 1
#define MERGE 2
#define LUT 3
#define MORPHIC 4
#define FFT 5
#define BUFMAP 6
#define PLOT 7
#define WARPMATRIX 8
#define PACK 9
#define WARPLUT 10
#define SHADING 11

void main(int argc, char **argv)
{
    long Device;          /* Genesis device */
    long Thread;          /* Thread to execute all functions */
    long SrcBuf;          /* Source buffer (original image) */
    long DstBuf;          /* Destination buffer (processed image) */
    long SrcDispBuf;      /* Display buffer for original image */
    long DstDispBuf;      /* Display buffer for processed image */
    long ScreenBuf;       /* Display buffer full size of screen */
    long SizeX, SizeY;    /* Image size */
    long NumBands;        /* Number of bands in image */
    long Func = 0;        /* The function to use */

```

```

/* Check arguments */
if (argc < 2 || *argv[1] == '?')
{
    printf("Usage: PROCESS file.tif [func]\n");
    printf("func = %2d User-defined convolution\n", CONVOLVE);
    printf("      %2d Image rotation\n", ROTATE);
    printf("      %2d Three-input ALU operation\n", MERGE);
    printf("      %2d LUT generation and mapping\n", LUT);
    printf("      %2d Binary morphology\n", MORPHIC);
    printf("      %2d Fourier Transform\n", FFT);
    printf("      %2d Host access to image buffer\n", BUFMAP);
    printf("      %2d Plot histogram with graphics functions\n",
        PLOT);
    printf("      %2d Matrix-defined warping\n", WARPMATRIX);
    printf("      %2d Non-rectangular ROIs\n", PACK);
    printf("      %2d LUT-defined warping\n", WARPLUT);
    printf("      %2d Shading correction\n", SHADING);
    exit(1);
}
if (argc > 2)
    sscanf(argv[2], "%li", &Func);

/* Establish an error handler */
imAppCatchError(IM_DEFAULT, ErrorHandler, NULL);

/* Allocate a device and a thread */
imDevAlloc(0, 0, NULL, IM_DEFAULT, &Device);
imThrAlloc(Device, 0, &Thread);

/* Load the image into a processing buffer */
imBufRestore(Thread, argv[1], IM_TIFF, IM_PROC, &SrcBuf);

/* Inquire the image size and number of bands */
imBufInquire(Thread, SrcBuf, IM_BUF_SIZE_X, &SizeX);
imBufInquire(Thread, SrcBuf, IM_BUF_SIZE_Y, &SizeY);
imBufInquire(Thread, SrcBuf, IM_BUF_NUM_BANDS, &NumBands);

/* Allocate two display buffers (they may be clipped to fit on the
/* screen) */
imBufChild(Thread, IM_DISP, 0, 0, SizeX, SizeY, &SrcDispBuf);
imBufChild(Thread, IM_DISP, SizeX, 0, SizeX, SizeY, &DstDispBuf);

/* Allocate a full screen display buffer and clear it */
imBufChild(Thread, IM_DISP, 0, 0, IM_ALL, IM_ALL, &ScreenBuf);
imBufClear(Thread, ScreenBuf, 0, 0);

/* Copy the original image to the display */
imBufCopy(Thread, SrcBuf, SrcDispBuf, 0, 0);

/* Clone the image buffer since some functions can't work in-place */
imBufClone(Thread, SrcBuf, IM_PROC, &DstBuf);

```

```

/* Perform the selected processing operation */
switch (Func)
{
    case CONVOLVE:
    {
        printf("User-defined convolution\n");
        long KerBuf;          /* Kernel buffer */
        short KerVals[9] = /* Array of kernel values */
        {
            -1, -1, -1,
            -1,  9, -1,
            -1, -1, -1
        };

        /* Allocate kernel buffer */
        imBufAlloc2d(Thread, 3, 3, IM_SHORT, IM_PROC, &KerBuf);

        /* Set kernel values */
        imBufPut(Thread, KerBuf, KerVals);

        /* Specify absolute value and clip */
        imBufPutField(Thread, KerBuf, IM_KER_ABSOLUTE, IM_ENABLE);
        imBufPutField(Thread, KerBuf, IM_KER_CLIP, IM_ENABLE);

        /* Perform the convolution */
        imIntConvolve(Thread, SrcBuf, DstBuf, KerBuf, 0, 0);

        /* Free the kernel buffer */
        imBufFree(Thread, KerBuf);
        break;
    }

    case ROTATE:
    {
        printf("Image rotation\n");
        long CoefBuf; /* Coefficient buffer (also control buffer) */

        /* Allocate warp coefficient buffer */
        imBufAlloc2d(Thread, 3, 2, IM_FLOAT, IM_PROC, &CoefBuf);

        /* Generate coefficients for 30 degree rotation about centre */
        imGenWarp1stOrder(Thread, CoefBuf, IM_TRANSLATE, -SizeX/2,
                           -SizeY/2, IM_CLEAR, 0);
        imGenWarp1stOrder(Thread, CoefBuf, IM_ROTATE, 30.0, 0.0,
                           IM_NO_CLEAR, 0);
        imGenWarp1stOrder(Thread, CoefBuf, IM_TRANSLATE, SizeX/2,
                           SizeY/2, IM_NO_CLEAR, 0);

        /* Select bilinear interpolation and replace overscan */
        imBufPutField(Thread, CoefBuf, IM_CTL_RESAMPLE, IM_BILINEAR);
        imBufPutField(Thread, CoefBuf, IM_CTL_OVERSCAN, IM_REPLACE);
    }
}

```



```

/* Rotate the image */
imIntWarpPolynomial(Thread, SrcBuf, DstBuf, CoefBuf,
                    CoefBuf, 0);

/* Free the coefficient buffer */
imBufFree(Thread, CoefBuf);

break;
}

case MERGE:
{
    printf("Three-input ALU operation\n");

    long SrcBBuf;        /* Source buffer for ALU B input */
    long SrcCBuf;        /* Source buffer for ALU C input */

    /* Use a negated copy of the original image as source B */
    imBufClone(Thread, SrcBuf, IM_PROC, &SrcBBuf);
    imIntMonadic(Thread, SrcBuf, 255, SrcBBuf, IM_SUB_NEG, 0);

    /* Draw a circular mask in the source C buffer */
    imBufAlloc2d(Thread, SizeX, SizeY, IM_UBYTE, IM_PROC,
                 &SrcCBuf);
    imBufClear(Thread, SrcCBuf, 0, 0);
    imGraArcFill(Thread, 0, SrcCBuf, SizeX/2, SizeY/2,
                 SizeX/3, SizeY/3, 0.0, 360.0);

    /* Copy original image inside the circle, and negated image
    /* outside it */
    imIntTriadic(Thread, SrcBuf, SrcBBuf, SrcCBuf, DstBuf, 0,
                 IM_PP_MERGE, IM_DEFAULT, 0);

    /* Free the temporary buffers */
    imBufFree(Thread, SrcBBuf);
    imBufFree(Thread, SrcCBuf);

    break;
}

case LUT:
{
    printf("Lut mapping\n");
    long LutBuf;          /* LUT buffer */
    double Coef[2] = {255.0, -1.0}; /* Coefficients for
                                     /* inverse ramp */

    /* Allocate LUT */
    imBufAlloc1d(Thread, 256, IM_UBYTE, IM_PROC, &LutBuf);

    /* Generate an inverse ramp */
    imGen1d(Thread, LutBuf, IM_POLYNOMIAL, 0, 255, 2, Coef, 0);

    /* Perform the LUT mapping */
    imIntLutMap(Thread, SrcBuf, DstBuf, LutBuf, 0);

```

```

/* Free the LUT */
imBufFree(Thread, LutBuf);

break;
}

case MORPHIC:
{
    printf("Binary thinning\n");

    long SkelBuf;          /* Buffer for kernel */
    long Bin1Buf, Bin2Buf; /* Binary work buffers */

    /* Define eight 3x3 kernels. "2" means "don't care" */
    short SkelVals[8][9] =
    {
        0, 0, 0, 2, 1, 2, 1, 1, 1,
        0, 2, 1, 0, 1, 1, 0, 2, 1,
        1, 1, 1, 2, 1, 2, 0, 0, 0,
        1, 2, 0, 1, 1, 0, 1, 2, 0,
        2, 1, 2, 1, 1, 0, 2, 0, 0,
        2, 0, 0, 1, 1, 0, 2, 1, 2,
        0, 0, 2, 0, 1, 1, 2, 1, 2,
        2, 1, 2, 0, 1, 1, 0, 0, 2
    };

    /* Allocate an 8-band kernel buffer */
    imBufAlloc(Thread, 3, 3, 8, IM_SHORT, IM_PROC, &SkelBuf);

    /* Allocate binary work buffers */
    imBufAlloc(Thread, SizeX, SizeY, NumBands, IM_BINARY,
                IM_PROC, &Bin1Buf);
    imBufAlloc(Thread, SizeX, SizeY, NumBands, IM_BINARY,
                IM_PROC, &Bin2Buf);

    /* Set kernel values (all eight bands at once) */
    imBufPut(Thread, SkelBuf, SkelVals);

    /* Binarize the image ready for thinning */
    imBinConvert(Thread, SrcBuf, Bin1Buf, IM_GREATER, 128, 0, 0);

    /* Thin to a skeleton using replace overscan */
    imBufPutField(Thread, SkelBuf, IM_CTL_OVERSCAN, IM_REPLACE);
    imBinMorphic(Thread, Bin1Buf, Bin2Buf, SkelBuf, IM_THIN,
                 IM_IDEMPOTENCE, SkelBuf, 0);

    /* Convert back to an 8-bit image for display */
    imBinConvert(Thread, Bin2Buf, DstBuf, IM_DEFAULT, 0, 255, 0);

    /* Free the temporary buffers */
    imBufFree(Thread, SkelBuf);
    imBufFree(Thread, Bin1Buf);
    imBufFree(Thread, Bin2Buf);

    break;
}

```

```

case FFT:
{
    printf("Fourier Transform\n");

    long IntrBuf;    /* Real component in fixed point */
    long IntIBuf;    /* Imaginary component in fixed point */
    long FltRBuf;    /* Real component in floating point */
    long FltIBuf;    /* Imaginary component in floating point */

    /* Allocate 32-bit buffers for FFT
    /* (size must be a power of 2) */
    imBufAlloc2d(Thread, 512, 512, IM_LONG, IM_PROC, &IntrBuf);
    imBufAlloc2d(Thread, 512, 512, IM_LONG, IM_PROC, &IntIBuf);
    imBufAlloc2d(Thread, 512, 512, IM_FLOAT, IM_PROC, &FltRBuf);
    imBufAlloc2d(Thread, 512, 512, IM_FLOAT, IM_PROC, &FltIBuf);

    /* Convert source from 8-bit real to 32-bit
    /* fixed-point complex*/
    imBufClear(Thread, IntrBuf, 0, 0); /*Clear in case bigger
                                        * than source */
    imBufClear(Thread, IntIBuf, 0, 0); /*Imaginary part is 0 */

    /* Add 12 fractional bits for extra precision */
    imIntMonadic(Thread, SrcBuf, 12, IntrBuf, IM_SHIFT, 0);

    /* Set control fields for forward transform */
    imBufPutField(Thread, IntrBuf, IM_CTL_DIRECTION, IM_FORWARD);
    imBufPutField(Thread, IntrBuf, IM_CTL_NORMALIZE, IM_ENABLE);

    /* Perform the FFT (in-place to save memory) */
    imIntFFT(Thread, IntrBuf, IntIBuf, IntrBuf, IntIBuf,
              IntrBuf, 0);

    /* Convert FFT result to floating point
    /* for further processing */
    imFloatConvert(Thread, IntrBuf, FltRBuf, IM_DEFAULT, 0);
    imFloatConvert(Thread, IntIBuf, FltIBuf, IM_DEFAULT, 0);

    /* Set control fields for reverse transform */
    imBufPutField(Thread, IntrBuf, IM_CTL_DIRECTION, IM_REVERSE);
    imBufPutField(Thread, IntrBuf, IM_CTL_NORMALIZE, IM_DISABLE);

    /* Perform the reverse FFT */
    imIntFFT(Thread, IntrBuf, IntIBuf, IntrBuf, IntIBuf,
              IntrBuf, 0);

    /* Remove fractional bits (with rounding for extra
    /* precision) */
    imIntMonadic(Thread, IntrBuf, 1<<11, IntrBuf, IM_ADD, 0);
    imIntMonadic(Thread, IntrBuf, -12, IntrBuf, IM_SHIFT, 0);

    /* Clip real part to 8 bits (imaginary part should be zero) */
    imIntConvert(Thread, IntrBuf, DstBuf, IM_CLIP, 0);

    /* Display real part (it should be the same as
    /* original image) */
    imBufCopy(Thread, DstBuf, DstDispBuf, 0, 0);

```

```

/* Calculate power spectrum of the FFT for display */
imFloatDyadic(Thread, FltRBuf, FltIBuf, FltRBuf,
               IM_SQUARE_ADD, 0);
imFloatUnary(Thread, FltRBuf, FltRBuf, IM_SQRT, 0);

/* Take square root again just to reduce the dynamic range */
imFloatUnary(Thread, FltRBuf, FltRBuf, IM_SQRT, 0);

/* Convert back to 32-bit integer */
imFloatConvert(Thread, FltRBuf, IntrBuf, IM_TRUNCATE, 0);

/* Convert back to 8-bit integer for display */
imIntConvert(Thread, IntrBuf, DstBuf, IM_CLIP, 0);

/* Histogram equalize (just to be sure something is
/* visible) */
imIntHistogramEqualize(Thread, DstBuf, DstBuf, 256,
                       IM_UNIFORM, 0.0, 0, 255,
                       IM_DEFAULT, 0);

/* Free the temporary buffers */
imBufFree(Thread, IntrBuf);
imBufFree(Thread, IntIBuf);
imBufFree(Thread, FltRBuf);
imBufFree(Thread, FltIBuf);

break;
}

case BUFMAP:
{
    printf("Map buffer on host\n");\
    long HistBuf;          /* Histogram result buffer */
    long HistVals[256];    /* Host array to hold histogram
                           * result */
    unsigned char *Address; /* Host address of first pixel in
                           * image */
    long Pitch;            /* Memory pitch of image buffer */
    long NLines;           /* Number of lines mapped in host
                           * memory */
    long MaxVal;           /* Maximum value in histogram */
    long x, y;             /* Loop counters */
    unsigned char *Pointer; /* Pointer for direct access to
                           * buffer */

    /* Allocate histogram result buffer */
    imBufAlloc1d(Thread, 256, IM_LONG, IM_PROC, &HistBuf);

    /* Perform a histogram and read it back to the host */
    imIntHistogram(Thread, SrcBuf, HistBuf, IM_DEFAULT, 0);
    imBufGet(Thread, HistBuf, HistVals);

    /* Find maximum value in histogram */
    imIntFindExtreme(Thread, HistBuf, HistBuf, IM_MAX_PIXEL, 0);
    imBufGetField(Thread, HistBuf, IM_RES_MAX_PIXEL, &MaxVal);
    printf("Maximum value in histogram is %li\n", MaxVal);
}

```

```

/* Map destination buffer into host memory */
imBufMap(Thread, DstBuf, 0, 0, (void **)&Address, &Pitch,
          &NLines);

/* Clear the buffer before drawing */
imBufClear(Thread, DstBuf, 50, 0);

/* Wait for the clear to finish before accessing the buffer
/* directly */
imSyncHost(Thread, 0, IM_COMPLETED);

/* Draw the histogram directly into the buffer */
for (x = 0; x < 256; x++) /* draw in a 256x256 region */
{
    /* Calculate host address of each point to set */
    y = 255 - (HistVals[x] * 255 / MaxVal);
    Pointer = Address + (y * Pitch) + x;

    /* Write directly to the buffer */
    *Pointer = 255;
}

/* Free the histogram result */
imBufFree(Thread, HistBuf);

break;
}

case PLOT:
{
    printf("Draw histogram with imGraPlot()\n");

    long XBuf;          /* Buffer with X values of points to
                        * plot */
    long YBuf;          /* Buffer with Y values of points to
                        * plot */
    long MaxVal;        /* Maximum value in histogram */
    double Coef[2] = {0.0, 1.0}; /* Coefficients for ramp */

    /* Allocate buffers for X and Y values to plot */
    imBufAlloc1d(Thread, 256, IM_LONG, IM_PROC, &XBuf);
    imBufAlloc1d(Thread, 256, IM_LONG, IM_PROC, &YBuf);

    /* Y values come from the histogram */
    imInthHistogram(Thread, SrcBuf, YBuf, IM_DEFAULT, 0);

    /* X values are just sequential numbers */
    imGen1d(Thread, XBuf, IM_POLYNOMIAL, 0, 255, 2, Coef, 0);

    /* Find maximum value in histogram */
    imIntFindExtreme(Thread, YBuf, YBuf, IM_MAX_PIXEL, 0);
    imBufGetField(Thread, YBuf, IM_RES_MAX_PIXEL, &MaxVal);
    printf("Maximum value in histogram is %li\n", MaxVal);
}

```

```

/* Scale the plot to fit image (use XBuf to hold graphic
/* context*/
imBufPutField(Thread, XBuf, IM_GRA_SCALE_Y,
               (double) -(SizeY- 1) / MaxVal);
imBufPutField(Thread, XBuf, IM_GRA_OFFSET_Y, SizeY - 1);

imBufPutField(Thread, XBuf, IM_GRA_SCALE_X,
               (double) SizeX / 256);
imBufPutField(Thread, XBuf, IM_GRA_COLOR, 255);

/* Plot the histogram */
imBufClear(Thread, DstBuf, 0, 0);
imGraRect(Thread, 0, DstBuf, 0, 0, SizeX-1, SizeY-1);
imGraPlot(Thread, XBuf, DstBuf, XBuf, YBuf, 256);
imGraText(Thread, 0, DstBuf, 10, 10, "Histogram");

/* Free the X and Y buffers */
imBufFree(Thread, XBuf);
imBufFree(Thread, YBuf);

break;
}

case WARPMATRIX:
{
    printf("Perspective transform\n");
    long CoefBuf;      /* Warp coefficient buffer */
    long XLutBuf;      /* X address LUT buffer */
    long YLutBuf;      /* Y address LUT buffer */

    /* Allocate warp coefficient and address LUT buffers */
    imBufAlloc2d(Thread, 3, 3, IM_FLOAT, IM_PROC, &CoefBuf);
    imBufAlloc2d(Thread, SizeX, SizeY, IM_SHORT, IM_PROC,
                 &XLutBuf);
    imBufAlloc2d(Thread, SizeX, SizeY, IM_SHORT, IM_PROC,
                 &YLutBuf);

    /* Generate coefficients for perspective transform */
    imGenWarp4Corner(Thread, CoefBuf,
                    0, SizeY/4, SizeX-1, SizeY/4,
                    3*SizeX/4, 3*SizeY/4, SizeX/4, 3*SizeY/4,
                    0, 0, SizeX-1, SizeY-1, IM_DEFAULT, 0);

    /* Generate address LUTs from the coefficients
    /* (use 4 frac.bits) */
    imBufPutField(Thread, XLutBuf, IM_CTL_PRECISION, 4);
    imGenWarpLutMatrix(Thread, XLutBuf, YLutBuf, CoefBuf,
                      XLutBuf, 0);

    /* Select bilinear interpolation */
    imBufPutField(Thread, XLutBuf, IM_CTL_RESAMPLE, IM_BILINEAR);

    /* Warp the image */
    imIntWarpLut(Thread, SrcBuf, DstBuf, XLutBuf, YLutBuf,
                 XLutBuf, 0);

```

```

/* Free the coefficient and LUT buffers */
imBufFree(Thread, CoefBuf);
imBufFree(Thread, XlutBuf);
imBufFree(Thread, YlutBuf);

break;
}

case PACK:
{
    printf("Use imBufPack() to process a non-rectangular ROI\n");

    long TagBuf;          /* Binary tag buffer */
    long ByteTagBuf;       /* 8-bit version of tag buffer */
    long PackedBuf;        /* 1-d buffer big enough to hold tagged
                           * pixels */
    long ROIBuf;           /* 1-d buffer exactly the right size */
    long NumTagged;        /* Number of tagged pixels */

    /* Allocate tag buffer (1-bit and 8-bit versions) */
    imBufAlloc2d(Thread, SizeX, SizeY, IM_BINARY, IM_PROC,
                  &TagBuf);
    imBufAlloc2d(Thread, SizeX, SizeY, IM_UBYTE, IM_PROC,
                  &ByteTagBuf);

    /* Allocate 1-d buffer for packed pixels
    /* (make sure it's big enough) */
    imBufAlloc(Thread, SizeX*SizeY, 1, NumBands, IM_UBYTE,
                IM_PROC, &PackedBuf);

    /* Draw a circular mask in the tag buffer
    /* (must use 8-bit version) */
    imBufClear(Thread, ByteTagBuf, 0, 0);
    imGraArcFill(Thread, 0, ByteTagBuf, SizeX/2, SizeY/2,
                  SizeX/3, SizeY/3, 0.0, 360.0);

    /* Make the real binary tag buffer */
    imBinConvert(Thread, ByteTagBuf, TagBuf, IM_GREATER,
                  0, 0, 0);

    /* Pack the tagged pixels */
    imBufPack(Thread, SrcBuf, TagBuf, PackedBuf, IM_PACK_1, 0);

    /* Find out how many pixels were tagged */
    imBufGetField(Thread, PackedBuf, IM_RES_NUM_PIXELS,
                  &NumTagged);
    printf("Number of tagged pixels is %li\n", NumTagged);

    /* Make a buffer with just those pixels in the
    /* non-rectangular ROI */
    imBufChild(Thread, PackedBuf, 0, 0, NumTagged, 1, &ROIBuf);

    /* Process the non-rectangular ROI */
    imIntMonadic(Thread, ROIBuf, 50, ROIBuf, IM_ADD_SAT, 0);

```

```

/* Unpack the processed pixels for display */
imBufClear(Thread, DstBuf, 0, 0);
imBufPack(Thread, ROIBuf, TagBuf, DstBuf, IM_UNPACK_1, 0);

/* Free the temporary buffers */
imBufFree(Thread, TagBuf);
imBufFree(Thread, ByteTagBuf);
imBufFree(Thread, PackedBuf);
imBufFree(Thread, ROIBuf);
break;
}

case WARPLUT:
{
    printf("LUT-defined warping\n");

    long XLutBuf;          /* X address LUT buffer */
    long YLutBuf;          /* Y address LUT buffer */
    short *XLutVals;       /* Host array to hold X LUT values */
    short *YLutVals;       /* Host array to hold Y LUT values */
    int Ox, Oy;            /* Original pixel coordinates */
    int Wx, Wy;            /* Warped pixel coordinates */

    /* Allocate address LUT buffers */
    imBufAlloc2d(Thread, SizeX, SizeY, IM_SHORT, IM_PROC,
                  &XLutBuf);
    imBufAlloc2d(Thread, SizeX, SizeY, IM_SHORT, IM_PROC,
                  &YLutBuf);

    /* Allocate host memory in which to create the LUTs */
    XLutVals = (short *) malloc(sizeof(short) * SizeX * SizeY);
    YLutVals = (short *) malloc(sizeof(short) * SizeX * SizeY);
    if (XLutVals == NULL || YLutVals == NULL)
    {
        printf("Couldn't allocate host memory\n");
        break;
    }

    /*
     * Calculate the (X,Y) source address for each destination
     * pixel. Use integer address values for nearest neighbour
     * resampling.
     */
    for (Oy = 0; Oy < SizeY; Oy++)
    {
        for (Ox = 0; Ox < SizeX; Ox++)
        {
            /* Flip the image in the X direction */
            Wx = SizeX - 1 - Ox;

            /* Add a sine wave offset in the Y direction */
            Wy = Oy + (int) (20.0 * sin(Ox * 0.03));

```



```

        /* Don't let the address fall outside the
        /* source image */
        if (Wx < 0 || Wx >= SizeX || Wy < 0 || Wy >= SizeY)
        {
            Wx = 0;
            Wy = 0;
        }

        /* Write the (X,Y) address in the LUTs */
        XLutVals[0x + 0y*SizeX] = (short) Wx;
        YLutVals[0x + 0y*SizeX] = (short) Wy;
    }
}

/* Load the address LUT buffers */
imBufPut(Thread, XLutBuf, XLutVals);
imBufPut(Thread, YLutBuf, YLutVals);

/* Warp the image (using the default nearest neighbour mode)*/
imIntWarpLut(Thread, SrcBuf, DstBuf, XLutBuf, YLutBuf, 0, 0);

/* Free the LUT buffers */
imBufFree(Thread, XLutBuf);
imBufFree(Thread, YLutBuf);

/* Free host memory */
free(XLutVals);
free(YLutVals);

break;
}

case SHADING:
{
    printf("Shading correction\n");

    long GainBuf;          /* Per-pixel gain buffer */
    unsigned short *GainVals; /* Host array to hold gain
                             * values */
    long FracBits = 12;    /* No. of fractional bits in
                             * gain values */
    int x, y;              /* Loop counters */
    float DistX, DistY, Dist, Gain; /* Used in gain calculation*/

    /* Allocate gain buffer */
    imBufAlloc2d(Thread, SizeX, SizeY, IM_USHORT, IM_PROC,
                 &GainBuf);

    /* Allocate host memory for gain values */
    GainVals = (unsigned short *)
        malloc(sizeof(unsigned short) * SizeX * SizeY);
    if (GainVals == NULL)
    {
        printf("Couldn't allocate host memory\n");
        break;
    }
}

```

```

/*
 * Calculate the gain value for each pixel. Assume that the
 * lighting is brightest in the centre of the image, and
 * decreases with distance from the centre. To compensate for
 * this the gain values must be biggest at the edges, and
 * smallest in the centre.
 */
for (y = 0; y < SizeY; y++)
{
    for (x = 0; x < SizeX; x++)
    {
        /* Calculate distance from pixel to image centre */
        DistX = (float) (x - SizeX/2);
        DistY = (float) (y - SizeY/2);
        Dist = (float) sqrt(DistX*DistX + DistY*DistY);

        /* Gain is 1.0 at centre, and 2.0 at the edges */
        Gain = (float) (1.0 + (Dist / (SizeX/2)));

        /* Write the gain value as a fixed point integer */
        GainVals[x + y*SizeX] = (unsigned short) (Gain *
            (1 << FracBits) + 0.5);
    }
}

/* Load the gain values into the gain buffer */
imBufPut(Thread, GainBuf, GainVals);

/* Apply the gain correction, and clip any overflows */
imIntGainOffset(Thread, SrcBuf, DstBuf, 0, GainBuf,
    FracBits, 255, IM_CLIP, 0);

/* Free the Gain buffer */
imBufFree(Thread, GainBuf);

/* Free host memory */
free(GainVals);

break;
}

default:
    printf("Unsupported function\n");
    break;
}

/* Copy the processed image to the display */
imBufCopy(Thread, DstBuf, DstDispBuf, 0, 0);

/* Give any errors a chance to be reported */
imSyncHost(Thread, 0, IM_COMPLETED);

```

```

    /* Clean up */
    imBufFree(Thread, SrcDispBuf);
    imBufFree(Thread, DstDispBuf);
    imBufFree(Thread, ScreenBuf);
    imBufFree(Thread, SrcBuf);
    imBufFree(Thread, DstBuf);
    imThrFree(Thread);
    imDevFree(Device);
}

void ErrHandler(void *Param)
{
    char Error[IM_ERR_SIZE];

    /* Param is not used in this case */
    if (Param) ;

    /*
     * Get the error message and print it. Don't reset the error
     * because we only want the first to be printed.
     */
    imAppGetError(IM_ERR_MSG_FUNC, Error);
    printf("%s\n", Error);
}

```

tfilter.c

```

/*****
 *
 * Temporal filtering (in monochrome or colour).
 *
 * This example demonstrates how grab is double-buffered
 * to achieve real-time processing. Asynchronous grab mode is used
 * so that only a single thread is needed.
 *
 *****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <conio.h>

#include "imapi.h"

void main(int argc, char **argv)
{
    long Device;           /* Genesis device */
    long Camera;           /* Camera */
    long Thread;           /* Thread to execute all functions */
    long GrabOSB[2];       /* OSBs used for synchronization */
    long InBuf[2];         /* Double-buffered input buffer */
    long OutBuf;           /* Single output buffer */
    long DispBuf;          /* Display buffer */
    long GrabCtrlBuf;      /* Grab control buffer */
    long VMSrcCtrlBuf;     /* VM source control buffer */
    long VMDstCtrlBuf;     /* VM destination control buffer */
    long SizeX, SizeY;     /* Image Size */
    long NumBands;         /* Number of bands in image */
    long Zoom = 0;         /* Whether to zoom the display */
    long a;                /* Input weight */
    long m = 8;            /* Fractional bits in 'a' */
    long n = 4;            /* Fractional bits in output */
    long OutType = IM_USHORT; /* Output buffer type */
    char Error[IM_ERR_SIZE]; /* Error message */
    long i, frames = 0;
    double time, da = 0.1;

    /* Check arguments */
    if (argc > 1 && argv[1] == '?')
    {
        printf("Usage: TFILTER [-a] [-x] [-y] [-zoom]\n");
        printf("    -a 0.nn\t Input weight (default %.2f)\n", da);
        printf("    -x size\t Image X size\n");
        printf("    -y size\t Image Y size\n");
        printf("    -zoom\t Zoom by 2\n");
        exit(1);
    }
}

```

```

/* Allocate the device and a thread */
imDevAlloc(0, 0, NULL, IM_DEFAULT, &Device);
imThrAlloc(Device, 0, &Thread);

/* Allocate a camera */
imCamAlloc(Thread, NULL, IM_DEFAULT, &Camera);

/* Determine the image size and number of bands */
imCamInquire(Thread, Camera, IM_DIG_SIZE_X, &SizeX);
imCamInquire(Thread, Camera, IM_DIG_SIZE_Y, &SizeY);
imCamInquire(Thread, Camera, IM_DIG_NUM_BANDS, &NumBands);

/* Check if the user specified a different size or weight */
for (i = 1; i < argc; i++)
{
    if (!strcmp(argv[i], "-a"))
        sscanf(argv[i+1], "%lf", &da);
    else if (!strcmp(argv[i], "-x"))
        sscanf(argv[i+1], "%li", &SizeX);
    else if (!strcmp(argv[i], "-y"))
        sscanf(argv[i+1], "%li", &SizeY);
    else if (!strcmp(argv[i], "-zoom"))
        Zoom = 1;
}

/* Convert weight to fixed point */
a = (long) (da * (1 <= m) + 0.5);

/* For colour use an 8-bit output buffer to reduce processing time */
if (NumBands > 1)
{
    n = 0;
    OutType = IM_UBYTE;
}

/* Allocate processing and control buffers */
imBufAlloc(Thread, SizeX, SizeY, NumBands, IM_UBYTE,
            IM_PROC, &InBuf[0]);
imBufAlloc(Thread, SizeX, SizeY, NumBands, IM_UBYTE,
            IM_PROC, &InBuf[1]);
imBufAlloc(Thread, SizeX, SizeY, NumBands, OutType,
            IM_PROC, &OutBuf);
imBufAllocControl(Thread, &GrabCtrlBuf);
imBufAllocControl(Thread, &VMSrcCtrlBuf);
imBufAllocControl(Thread, &VMDstCtrlBuf);

/* Allocate a full-screen display buffer and clear it */
imBufChild(Thread, IM_DISP, 0, 0, IM_ALL, IM_ALL, &DispBuf);
imBufClear(Thread, DispBuf, 0, 0);

/* Allocate OSBs for synchronization */
imSyncAlloc(Thread, &GrabOSB[0]);
imSyncAlloc(Thread, &GrabOSB[1]);

/* Initialize the output to 0 */
imBufClear(Thread, OutBuf, 0, 0);

```

```

/* Select asynchronous grab mode */
imBufPutField(Thread, GrabCtrlBuf, IM_CTL_GRAB_MODE,
              IM_ASYNCHRONOUS);

/* Specify byte extraction on VM transfer if necessary */
if (n > 0)
    imBufPutField(Thread, VMSrcCtrlBuf, IM_CTL_BYTE_EXT, 8 + n);

/* Optionally zoom by 2 on VM transfer */
if (Zoom)
{
    imBufPutField(Thread, VMDstCtrlBuf, IM_CTL_ZOOM_X, 2);
    imBufPutField(Thread, VMDstCtrlBuf, IM_CTL_ZOOM_Y, 2);
}

/* First grab */
time = imSysClock(Thread, 0.0);
imDigGrab(Thread, 0, Camera, InBuf[0], 1, GrabCtrlBuf, GrabOSB[0]);
printf("Press Enter to stop\n");

i = 1;
while (!imAppGetError(IM_ERR_CODE, NULL))
{
    /* Queue next grab into other buffer */
    imDigGrab(Thread, 0, Camera, InBuf[i], 1, GrabCtrlBuf,
              GrabOSB[i]);

    /* Switch buffers */
    i = 1 - i;

    /* Process each frame as soon as the grab completes */
    imSyncHost(Thread, GrabOSB[i], IM_COMPLETED);
    imIntMac2(Thread, InBuf[i], OutBuf, OutBuf, a<n, (1<<m)-a,
              m, 0);

    /* Copy an 8-bit version to the display (optionally zoomed) */
    imBufCopyVM(Thread, OutBuf, DispBuf, VMSrcCtrlBuf,
                VMDstCtrlBuf, 0);

    /* Stop when a key is hit */
    frames++;
    if (kbhit())
        break;
}

/* Calculate processing rate */
time = imSysClock(Thread, time);
if (frames > 0)
    printf("Processing rate = %.1f fps\n", frames / time);

```

```
/* Check for any errors */
if (imAppGetError(IM_ERR_MSG_FUNC + IM_ERR_RESET, Error))
    printf("%s\n", Error);

/* Clean up */
imBufFree(Thread, DispBuf);
imBufFree(Thread, OutBuf);
imBufFree(Thread, InBuf[0]);
imBufFree(Thread, InBuf[1]);
imBufFree(Thread, GrabCtrlBuf);
imBufFree(Thread, VMSrcCtrlBuf);
imBufFree(Thread, VMDstCtrlBuf);
imSyncFree(Thread, GrabOSB[0]);
imSyncFree(Thread, GrabOSB[1]);
imCamFree(Thread, Camera);
imThrFree(Thread);
imDevFree(Device);
}
```


Index

A

- acceptance level
 - default value 336
 - setting 353
- adaptive recursive filtering 294
- allocating
 - blob analysis feature list 44
 - blob analysis result buffer 45
 - buffers 84, 86, 88, 143
 - camera definitions 146
 - child buffers 91, 94
 - control buffers 90
 - digitizer 171
 - display 185
 - display memory (off-screen) 84, 86, 88, 143
 - display memory (on-screen) 91
 - Host memory 84, 86, 88, 143
 - JPEG buffers 315
 - node 167
 - OSB 369
 - pattern matching models 334, 336
 - pattern matching result buffers 338
 - processing memory 84, 86, 88, 143
 - threads 382
- applications
 - checking for errors 25, 28
 - inquiring about 31
 - setting 27
- arc (elliptical), drawing 214, 216
- area, of blobs 73
- arithmetic operations
 - on floating-point buffers 196, 198–200, 202
 - on integer buffers 255, 267, 283, 285, 287, 304
- artifacts (on the display), avoiding 116
- asynchronous functions 24
- asynchronous grabs 181
- auto reset OSBs 370
- axis of symmetry, of blobs 79

B

- BGR packed images, converting to 105, 112, 179
- binarizing 230
- binary template matching 35
- bit planes, protecting 106, 115, 181, 189
- black reference levels, specifying 149
- blob analysis
 - accumulating results 46
 - binary features 46, 73, 78
 - calculating features 46
 - control settings 45, 48, 68
 - copying results 50
 - copying run information 54
 - deleting blobs 71
 - determine number of blobs 59
 - disabling runs 49
 - excluding blobs 71
 - features 73
 - filling blobs 56
 - freeing buffers 57
 - get label values 58
 - grayscale features 46, 77–78
 - grouping results 48
 - inquiring about settings 68
 - labelling blobs 70
 - obtaining results 64
 - re-including blobs 71
 - removing unwanted blobs 56
 - selecting blobs for calculation 71
 - selecting features 73
 - specifying pixel value of blobs 49
 - timeout period 49
 - transfer results to Host 60
 - transfer run information to Host 66
- blob features
 - area 73
 - axis of symmetry 79
 - box coordinates 73
 - breadth 76
 - center of gravity 78
 - compactness 75
 - contact points 75
 - convex perimeter 75
 - elongation 76
 - Euler number 76

- Feret diameters 74–75, 81
- Feret elongation 75
- intercepts 76–77
- label value 73
- length 76
- maximum pixel value 77
- mean pixel value 77
- minimum pixel value 77
- moments 78, 82
- number of holes 75
- perimeter 73
- point coordinates 74
- roughness 76
- runs in a blob 75
- standard deviation of pixel values 77
- sum of pixel values 77
- sum of squares of pixel values 77
- blob.c program 414
- board
 - Genesis-LC 6
 - main 6
 - processor 6
- breadth, of blobs 76
- buffers
 - allocating 84, 86, 88, 143
 - changing pitch 99
 - clearing 97
 - converting between color and grayscale 238
 - converting between integer and binary 32, 230
 - converting between integer and floating-point 194
 - converting between integer types 237
 - converting between RGB and HSL 238
 - copying data 100, 108
 - copying data (with formatting) 103, 110
 - copying fields 102
 - creating 117
 - freeing 121
 - generating data into 204
 - inquiring about 129
 - loading from file 131, 143
 - locking in physical memory 99
 - mapping to Host 132
 - message 170
 - modifying dimensions/type 134
 - packing 136
 - saving to file 145

- transferring from Host 138–140
- transferring to Host 122–124
- unlocking virtual buffers 99
- unpacking 136
- byte, extracting
 - during copy 107, 115
 - during grab 179

C

- C80
 - clock speed 169
 - functions that use 24
 - inquiring about presence 169
- camera definitions
 - allocating 146
 - changing 148
 - duplicating 147
 - freeing 154
 - inquiring about 155
 - saving to file 157
- center of gravity, of blobs 78
- certainty level
 - default value 336
 - setting 356
- Chamfer 3-4 transform 253
- Chessboard transform 253
- child buffers
 - allocating 91, 94
 - moving 95
 - resizing 95
- City Block transform 253
- clock value, reading 379
- CoffLoadOption entry 170
- column profiles 290
- compactness, of blobs 75
- compressing images 321
- connected regions
 - filling 218
 - labelling 274
- connectivity mapping 235
- control buffers, allocating 90
- control fields
 - adding 141
 - copying 102

- modifying 141
- reading 125–127
- removing 142
- converting
 - between buffer types during copy 105, 112–113
 - between buffer types during grab 179
 - between color and grayscale 238
 - between integer and binary 32, 230
 - between integer and floating-point 194
 - between integer types 237
 - between RGB and HSL 238
- convex perimeter, of blobs 75
- convolution operations 241
- coordinates, of blobs 73–75
- copying
 - blob analysis results 50
 - buffers 100, 108
 - buffers (with formatting) 103, 110
 - fields 102
 - from VM device 112
 - over PCI bus 103
 - over VMChannel 110
 - pattern matching models 339
 - run information about a blob 54
 - to/from VM stream 110
- correlations 249
- count differences between images 252
- creating buffers 117

D

- decompressing images 320
- digitizer
 - allocating 171
 - freeing 177
 - inquiring about 184
 - programming 174
- dilations
 - on binary images 35
 - on integer images 258
- display
 - allocating 185
 - freeing 191
 - inquiring about 192
 - scrolling a region 101
 - setting 187

- display artifacts, avoiding 116
- display memory
 - allocating off-screen 84, 86, 88, 143
 - allocating on-screen 91
 - inquiring about 169
- distance transforms 253
- DMA memory 85, 87, 89, 144
- drawing
 - elliptical arcs 214, 216
 - lines 220
 - plots 222
 - rectangles 224, 226

E

- edge detectors/enhancers 242
- elliptical arc, drawing 214, 216
- elongation, of blobs 76
- erosions
 - on binary images 35
 - on integer images 258
- error checking
 - applications 25, 28
 - functions 373
 - threads 387
- error handler
 - disabling 25
 - enabling 25
- Euler number, of blobs 76
- execution time, measuring 379, 381
- Exponential histogram equalization 272
- exposure time, specifying 152
- extracting byte
 - during copy 107, 115
 - during grab 179

F

- fast Fourier transforms 263
- Feret diameters, of blobs 74–75, 81
- Feret elongation, of blobs 75
- FFTs 263
- fields (control)
 - adding 141
 - copying 102

- modifying 141
- reading 125–127
- removing 142
- filling connected regions 218
- filtering, adaptive recursive 294
- find min/max value in image 262
- first.c program 419
- first-order warpings
 - generating coefficients for 206
 - performing 310
- flipping images 265
- Fourier transforms 263
- frame size of grab, setting 150
- free
 - blob analysis feature lists 57
 - blob analysis result buffers 57
 - buffers 121
 - camera definitions 154
 - digitizer 177
 - display 191
 - JPEG buffers 322
 - node 168
 - OSB 372
 - pattern matching models 341
 - pattern matching result buffers 341
 - threads 386
- functions
 - asynchronous 24
 - halting 389
 - nomenclature 10
 - obtaining current state of 375
 - synchronous 24
 - that can run on Genesis-LC 24
 - that support in-place 24
 - that use NOA 24
 - that use parallel processors of 'C80 24

G

- gain and offset correction 267
- gain levels, specifying 149
- genesis.ini file 170
- Genesis-LC 6, 24
- geometric operations
 - 3x3 matrix-defined warping 211
 - first-order warping 310
 - flipping 265

- LUT warping 308
- rotating 265, 310
- shearing 310
- subsampling 297, 299, 310
- translating 310
- zooming 297, 310, 313
- grab.c program 421
- grabbing
 - continuously 178
 - enabling software-triggered grabs 172, 174
 - enabling synchronized grabs 172
 - fields 178
 - frames 178
 - lines 178
 - with trigger 150
- grayscale correlations 249
- grayscale images, converting to RGB 238

H

- halting, functions 389
- hardware triggers, using 150
- histogram equalizations 271
- histograms 269
- hit-or-miss transformations 35
- horizontal edge detector 242
- horizontal shift, between images 348
- horizontal syncs, triggering with 153
- Host
 - allocate memory on 84, 86, 88, 143
 - transfer blob results to 60
 - transfer blob run information to 66
 - transfer buffer data to 122–124
 - transfer data from 138–140
 - transfer JPEG tables from 326
 - transfer JPEG tables to 323
 - transfer search results to 343
- hot spot
 - default coordinates 336
 - setting 355
- HSL images, converting to RGB 238
- Hyper Cube Root histogram equalization 272
- Hyper Log histogram equalization 272

-
- imAppCatchError() 25, 158–159, 165, 365, 367
 - imAppControl() 27
 - imAppGetError() 25, 28
 - imAppInquire() 31
 - imBinConvert() 32, 231
 - imBinMorphic() 35, 261, 303
 - imBinTriadic() 40, 42, 257, 289, 307
 - imBlobAllocFeatureList() 44, 73
 - imBlobAllocResult() 45
 - imBlobCalculate() 44–46, 49–50, 54, 58, 60, 64, 66, 69–71, 73, 81–82
 - imBlobControl() 46, 48, 71, 74, 99
 - imBlobCopyResult() 50, 59, 63
 - imBlobCopyRuns() 49, 54, 67, 69
 - imBlobFill() 49, 56, 69, 71
 - imBlobFree() 57
 - imBlobGetLabel() 49, 54, 58, 64, 66, 69
 - imBlobGetNumber() 53, 59
 - imBlobGetResult() 53–55, 59–60, 67
 - imBlobGetResultSingle() 54–55, 64, 67
 - imBlobGetRuns() 49, 55, 66, 69
 - imBlobInquire() 68, 70
 - imBlobLabel() 49, 69–70
 - imBlobSelect() 48, 56, 59, 71
 - imBlobSelectFeature() 44, 51, 61, 64, 72–73, 81, 83
 - imBlobSelectFeret() 44, 51, 61, 64, 72, 80–81
 - imBlobSelectMoment() 44, 51, 61, 64, 72, 80, 82
 - imBufAlloc() 84, 122–124, 138–140, 324
 - imBufAlloc1d() 86
 - imBufAlloc2d() 88
 - imBufAllocControl() 85, 89–90
 - imBufChild() 85, 89, 91, 94
 - imBufChildBand() 93–94
 - imBufChildMove() 95
 - imBufClear() 97
 - imBufClone() 98
 - imBufControl() 135
 - imBufCopy() 100, 107–108, 116, 122–124, 138–140
 - imBufCopyField() 100, 102, 108, 110, 141
 - imBufCopyPCI() 100, 103, 122–124, 138–140
 - imBufCopyVM() 100, 110
 - imBufCreate() 117
 - imBufFree() 118, 121
 - imBufGet() 63, 92, 122–124
 - imBufGet1d() 122–124
 - imBufGet2d() 122–123
 - imBufGetField() 35, 125–126, 128
 - imBufGetFieldDouble() 125–126, 128
 - imBufGetNextField() 125–127
 - imBufInquire() 129
 - imBufLoad() 131
 - imBufMap() 132
 - imBufModify() 134
 - imBufPack() 136
 - imBufPut() 138–140
 - imBufPut1d() 138–140
 - imBufPut2d() 138–140
 - imBufPutField() 141
 - imBufRemoveField() 142
 - imBufRestore() 131, 143
 - imBufSave() 145
 - imCamAlloc() 146
 - imCamClone() 147
 - imCamControl() 146–148, 174, 176
 - imCamFree() 154
 - imCamInquire() 155, 184
 - imCamSave() 157
 - imDevAlloc() 167, 382
 - imDevFree() 168
 - imDevInquire() 169
 - imDigAlloc() 171
 - imDigCapture() 172
 - imDigControl() 174
 - imDigFree() 177
 - imDigGrab() 178
 - imDigInquire() 184
 - imDispAlloc() 185
 - imDispControl() 101, 187, 192
 - imDispFree() 191
 - imDispInquire() 192
 - imFloatConvert() 194
 - imFloatDyadic() 196, 201
 - imFloatMac1() 198–199
 - imFloatMac2() 198–199
 - imFloatMonadic() 197, 200
 - imFloatUnary() 197, 201–202
 - imGen1d() 204, 278
 - imGenWarp1stOrder() 206, 212
 - imGenWarp4Corner() 209, 212

- imGenWarpLutMatrix() 206, 211
- imGraArc() 214, 217
- imGraArcFill() 215–216, 219
- imGraFill() 218
- imGraLine() 220
- imGraPlot() 221–222
- imGraRect() 224, 227
- imGraRectFill() 219, 225–226
- imGraText() 228
- imIntBinarize() 33, 230, 232
- imIntConnectMap() 235
- imIntConvert() 237
- imIntConvertColor() 238
- imIntConvolve() 241
- imIntCorrelate() 249
- imIntCountDifference() 252
- imIntDistance() 253
- imIntDyadic() 255, 289, 307
- imIntErodeDilate() 258, 293, 303
- imIntFFT() 263
- imIntFindExtreme() 262
- imIntFlip() 265, 312
- imIntGainOffset() 267, 284, 286
- imIntHistogram() 269, 271
- imIntHistogramEqualize() 270–271
- imIntLabel() 274
- imIntLocateEvent() 262, 276
- imIntLutMap() 278
- imIntMac1() 268, 283, 286
- imIntMac2() 268, 284–285, 296
- imIntMonadic() 257, 287, 307
- imIntProject() 290
- imIntRank() 291
- imIntRecFilter() 294
- imIntScale() 297, 300, 312, 314
- imIntSubsample() 298–299, 312, 314
- imIntThickThin() 301
- imIntTriadic() 257, 289, 304
- imIntWarpLut() 211, 308
- imIntWarpPolynomial() 206, 266, 298, 300, 310, 314
- imIntZoom() 298, 300, 312–313
- imJpegAlloc() 315
- imJpegControl() 315–316, 321
- imJpegControlBand() 319, 321
- imJpegDecode() 316, 320
- imJpegEncode() 316, 321, 325
- imJpegFree() 322
- imJpegGetTable() 323
- imJpegInquire() 324
- imJpegPutTable() 321, 326
- imJpegRead() 325, 327–328, 330
- imJpegReadBuf() 328, 330
- imJpegRestore() 325, 327–328, 330
- imJpegSave() 331
- imJpegWrite() 332
- imJpegWriteBuf() 331, 333
- imPatAllocModel() 334, 336
- imPatAllocResult() 338
- imPatCopy() 339
- imPatFindModel() 340, 342–343
- imPatFree() 341
- imPatGetNumber() 342–343
- imPatGetResult() 340, 343, 348
- imPatInquire() 345, 362
- imPatPreprocModel() 347, 349
- imPatRead() 350–351
- imPatRestore() 350–351
- imPatSave() 352
- imPatSetAcceptance() 347, 353
- imPatSetAccuracy() 346, 354
- imPatSetCenter() 346–347, 355
- imPatSetCertainty() 347, 356
- imPatSetDontCare() 349, 357
- imPatSetNumber() 346, 358
- imPatSetPosition() 346, 359
- imPatSetSearchParameter() 347, 360
- imPatSetSpeed() 346, 364
- imPatWrite() 352, 365
- imSyncAlloc() 369
- imSyncControl() 369–370
- imSyncFree() 372
- imSyncGetError() 373
- imSyncHost() 375
- imSyncThread() 377
- imSysClock() 379
- imSysInquire() 380–381
- imThrAlloc() 381
- imThrCancel() 384
- imThrControl() 385
- imThrFree() 386
- imThrGetError() 30, 387
- imThrHalt() 29, 178, 384, 389
- imThrInquire() 390
- imThrNop() 377, 391
- in-place operations 24

input channel, specifying 149

input line 184

inquire about

- applications 31

- blob analysis settings 68

- buffers 129

- C80 clock speed 169

- camera definitions 155

- digitizer 184

- display 192

- input line 184

- JPEG buffers 324

- memory 169

- node 169

- pattern matching models 345

- pattern matching result buffers 345

- presence of 'C80 169

- presence of NOA 169

- presence of VIA 169

- system attributes 380

- threads 390

- timeout period 31

intercepts, of blobs 76–77

J

JPEG compression/decompression

- allocating buffers for 315

- compressing 321

- control settings 316, 319

- decompressing 320

- freeing JPEG buffers 322

- inquiring about JPEG buffers 324

- loading JPEG buffers 330

- modes 316

- multi-band images 319

- predictors 316

- quality factor 316

- reading compressed images 327–328

- restart markers 317

- saving JPEG buffers to file 331

- tables 318–319

- transferring tables 323, 326

- writing JPEG buffers 332–333

jpeg.c program 423

K

keying 187, 192

L

label connected regions 274

label value, of blobs 73

Laplacian edge detector 242

length, of blobs 76

Library modules 10

lines, drawing 220

loading

- buffers from a file 131, 143

- JPEG buffers 330

locate specified pixels 276

logical operations

- on binary images 42

- on integer images 255, 287, 304

LUT mappings 271, 278

- on displayed images 188

- on grabbed images 149

LUT warpings 308

M

main board 6

manual reset OSBs 370

mapping buffer to Host 132

mapping through LUTs 271, 278

- displayed images 188

- grabbed images 149

matrix-defined warpings

- generating LUTs for 211

- performing 211

maximum value in image, find 262

memory

- allocating 84, 86, 88, 143

- inquiring about 169

message buffers 170

minimum value in image, find 262

model (pattern matching)

- allocating 334, 336

- copying 339

- freeing 341

- inquiring about 345

- preprocessing 349
- reading from open file 350
- restoring from file 351
- saving 349, 352
- search parameters of 336
- searching for 340
- setting don't care pixels 357
- writing to open file 365
- modules, of the Library 10
- moments, of blobs 78, 82
- morphological operations
 - on binary images 35
 - on integer images 253, 258, 301
- multiply and accumulate
 - with floating-point buffers 198–199
 - with integer buffers 283, 285

N

- neighborhood operations
 - connectivity mapping 235
 - convolutions 241
 - correlations 249
 - morphological operations 35, 253, 258, 301
 - rank filter operations 291
- NOA
 - functions that use 24
 - inquiring about presence 169
- node
 - allocating 167
 - freeing 168
 - inquiring about 169
- nomenclature of functions 10
- non-paged pool 85, 87, 89, 144
- normalized grayscale correlation 249
- number of holes, in blobs 75
- number of matches
 - default value 336
 - reading 342
 - setting 358

O

- off-screen display memory, allocating 84, 86, 88, 143
- on-screen display memory, allocating 91

OSB

- allocating 369
- freeing 372
- setting 370
- using in auto reset mode 370
- using in manual reset mode 370
- output lines, setting 175
- overlay display memory, inquiring about 169

P

- packing buffers 136
- panning, the display 189
- parallel processors of 'C80, functions that use 24
- pat.c program 426
- PCI bus, copying over 103
- perimeter, of blobs 73
- perspective warpings 209
- pitch, changing 99
- pixel values
 - find min/max in image 262
 - locate 276
 - sum along image columns/rows 290
- plotting 222
- point-to-point operations
 - adaptive recursive (temporal) filtering 294
 - arithmetic operations 196, 200, 202, 255, 287, 304
 - binarizing 230
 - gain and offset correction 267
 - histogram equalizations 271
 - logical operations 42, 255, 287, 304
 - LUT mappings 278
 - multiply and accumulate operations 198–199, 283, 285
- positional accuracy
 - default value 336
 - setting 354
- preprocessing, models 349
- Prewitt edge detector 242
- process.c program 430
- processing memory
 - allocating 84, 86, 88, 143
 - inquiring about 169
- processing non-rectangular regions 136
- processor board 6

- processors (parallel) of 'C80, functions that
 - use 24
- project 2-d images into 1-d 290

R

- rank filter operation 291
- Rayleigh histogram equalization 272
- reading from open file
 - compressed images 327
 - pattern matching models 350
- rectangles, drawing 224, 226
- recursive filtering 294
- reference levels, setting 149
- region-of-interest, allocating 91
- restart markers 317
- restoring from file
 - compressed images 330
 - pattern matching models 351
- RGB
 - converting to grayscale 238
 - converting to HSL 238
- RGB packed, converting to BGR 105, 112, 179
- RGB555/565, expanding to 3x8-bit 106, 114
- Roberts edge detector 243
- rotating
 - images 265, 310
- roughness, of blobs 76
- row profiles 290
- RS-422 drivers, enabling 152, 175
- RS-422 receivers, enabling 151, 175
- runs, in a blob 75

S

- saving to file
 - buffers 145
 - camera definitions 157
 - JPEG buffers 331
 - pattern matching models 349, 352
- scrolling a region of the display 101
- search parameters
 - default values 336
 - setting 340, 353–360, 364
- search region
 - default value 336
 - setting 359

- search speed
 - default value 336
 - setting 364
- Sharpen edge enhancers 243
- shearing images 310
- shift, between images 348
- Smooth operation 243
- Sobel edge detector 242
- software triggers, using 150, 172, 174
- statistical operations
 - column profiles 290
 - count differences between images 252
 - find min/max pixel value in image 262
 - histograms 269
 - locating pixels 276
 - row profiles 290
- subsample
 - copied data 105, 114
 - grabbed data 181
 - images 297, 299, 310
- synchronization channel, using 150
- synchronize
 - the Host with a function 375
 - threads 377
 - with functions that do not accept OSB 391
 - with functions whose OSB ID is unknown 391
- synchronized grabs, enabling 172
- synchronous functions 24
- system, inquire about 380

T

- template matching 35
- text, writing 228
- tfilter.c program 444
- thickening operations
 - on binary images 35
 - on integer images 301
- thinning operations
 - on binary images 35
 - on integer images 301
- threads
 - allocating 382
 - cancelling commands queued to 384
 - checking for errors 387
 - freeing 386

- inquiring about 390
- setting 385
- timeout period
 - inquiring about 31
 - setting 27
- timers, on the grab 150, 152
- timing, execution of a function 379, 381
- transfer
 - blob results to Host 60
 - blob run information to Host 66
 - buffer data to Host 122–124
 - data from Host 138–140
 - JPEG tables from Host 326
 - JPEG tables to Host 323
 - search results to Host 343
- transformations
 - generating coefficients for 206, 209
 - generating LUTs for 211
- translating images 310
- triggered grabs 150
- TTL drivers, enabling 152, 175
- TTL receivers, enabling 151, 175

U

- Uniform histogram equalization 272
- unpacking buffers 136
- user bits 152, 175

V

- vertical edge detector 243
- vertical shift, between images 348
- vertical syncs, triggering with 153
- VIA
 - copy options 105, 112
 - grab options 179
 - inquiring about presence 169
- VM stream, copy to/from 110, 112
- VMChannel, copying over 110

W

- warpings
 - first-order polynomial 310
 - generating coefficients for 206, 209
 - generating LUTs for 211
 - using LUTs 308
- white reference levels, setting 149
- write masks 106, 181, 189
- writing text 228
- writing to open file
 - compressed images 332–333
 - pattern matching models 365

Z

- zoom
 - copied data 106, 115
 - display 189
 - grabbed data 181
 - images 297, 310, 313

Product Support

Product Assistance Request Form

[illegible]

