

Matrox Genesis Native Library

version 2.1

User Guide

Manual no. 10482-301-0210

July 3, 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

Chapter 1: Introduction 13

The Matrox Genesis Native Library	14
The Matrox Genesis imaging boards	16
The Genesis main board	16
The Genesis processor board	19
The Genesis-LC	20
Basic software concepts	21
A word about examples	23

Chapter 2: Getting started 25

Getting started	26
Basic steps	26
A simple example	27
Your resources	29
Allocating resources	29
Freeing allocated resources.	30
Displaying an image	31
Transferring to/from the Host.	32
Grabbing an image	33
Error reporting	35
Synchronization	36
Running multiple applications	38

Chapter 3: Processing functions. 41

General overview.	42
Basic architecture of the 'C80.	43

Data types	44
Converting between types	45
Processing a specific region of an image	47
Rectangular region.	47
Non-rectangular region	47
Point-to-point processing	49
Mixed data types	49
LUT mappings	50
Mapping with a non-interpolated LUT	52
Mapping with an interpolated LUT	55
Histogram equalization	57
Neighborhood processing	58
Spatial filtering operations.	59
Morphological operations.	60
Defining your own kernel.	68
Specifying the overscan pixels	70
Connectivity mapping	71
Geometric processing	72
Flip/rotate.	72
Scale by integer factors	72
Scale by non-integer factors.	73
Color processing	74
Choosing a color space	74
Statistical processing	77
Histograms	78

***Chapter 4: Advanced processing*79**

Three-input arithmetic and logical operations80
Live processing83
Grabbing a sequence of frames in real-time.83
Real-time processing.83
Geometric warpings86
First-order polynomial warpings86
Using a LUT to perform a warping88
Interpolation modes91
Fourier transforms92

***Chapter 5: Blob analysis*95**

Blob analysis96
General steps97
Segmentation99
Adjusting controls101
Pixel aspect ratio102
Grouping results.104
Timeout period105
Features106
Area and perimeter106
Dimensions107
Shape.109
Blob location110

Selecting blobs	111
Transferring or copying results.	112
Transferring or copying runs	115
<hr/>	
<i>Chapter 6: Pattern matching</i>	<i>117</i>
Pattern matching	118
General steps	119
Creating the model.	121
Preprocessing the model.	122
Adjusting search parameters	123
Acceptance level	123
Number of matches	124
Model's hot spot.	125
Search region	125
Positional accuracy	126
Certainty level	126
"Don't care" pixels	127
Search speed	128
Speeding up the search	129
Managing models	130
The pattern matching algorithm.	131
Normalized Correlation	131
Hierarchical Search	133
Search Heuristics	135
Sub-Pixel Accuracy	136

Chapter 7: Compression137

Introduction	138
Run-length encoding and decoding	138
Run-length encoding (compression)	139
Decoding run-length encoded images (decompression)	140
General steps	140
JPEG Compression	143
General steps	144
Controlling JPEG lossless compression	147
Predictive coding.	147
Huffman encoding	148
Encoding a very large image	150
Writing/reading to or from open files.	151
Restart markers	153
JPEG compatibility issues.	154

Chapter 8: Generating graphics157

Graphics.	158
Generating graphics	159
Plotting.	160
Filling.	162
Writing text.	162

Chapter 9: Buffers and buffer fields163

Data buffers	164
Allocating buffers	165

Data type.	166
Memory location	166
Control buffers.	167
Child buffers	169
Copying buffer data	170
Using the advanced copy functions	171
Tag buffers	173
Zooming and subsampling.	174
Extracting bytes.	174
Swapping bytes	174
Reversing the direction of the copy.	175
Expanding RGB555 or 565 formats	175
Write masks.	175
Reducing overhead	176
Specifying a VIA.	176
Writing a rectangular region	177
Avoiding display artifacts.	177
Continuous copying.	177
Copying to/from the VMChannel	178
Transferring buffer data to the Host	179
Mapping a buffer	180
Creating a buffer from memory already allocated	182

Chapter 10: Grabbing images. 183

Grabbing	184
The grab module	185

VIA options of the grab command	186
Number of iterations	187
Synchronizing multiple grabs	187
Grabbing a rectangular region	187
Grabbing a VM stream	188
Reversing the direction of the grab	188
Grab mode	188
Reducing overhead	188
Line interrupts	189
Grabbing to two or more buffers	190
Camera settings	192
Input channel	193
Synchronization channel	197
Gain and reference levels	197
Input LUTs	198
Frame size	200
User bits	201
Triggers	202
Programmable timers	204
Running multiple applications	208
<hr/>	
<i>Chapter 11: Displaying images</i>	209
The display section	210
Grayscale images vs. color images	213
Using the monochrome version	213
Using the color version	214
Using the overlay	215

Keying	217
In single-screen mode	217
In dual-screen mode	218
Panning, scrolling, and zooming	219
Look-up tables	220
Grab and display	221
Using the hardware cursor	222
General steps to using a cursor	222
An example	225
Display memory as extra storage space	227
Running multiple applications	228
<hr/>	
<i>Chapter 12: Error handling</i>	<i>229</i>
Error mechanisms	230
Which error mechanism to use	231
More about application-wide errors	232
Places to check for errors	234
<hr/>	
<i>Chapter 13: Optimizing your application</i>	<i>235</i>
Overview	236
Estimating performance	237
General formula	237
Overheads	238
I/O bound functions	238
Compute bound functions	240
NOA setup overhead	240

Multiprocessing	242
Multiple threads	242
Multiple nodes	245
Programming tips	247
<hr/>	
<i>Appendix A: Glossary</i>	<i>249</i>
<hr/>	
<i>Appendix B: Examples</i>	<i>269</i>
blob.c	270
first.c	275
grab.c	277
jpeg.c	279
pat.c	282
process.c	286
tfilter.c	300
<hr/>	
<i>Index</i>	
<hr/>	
<i>Product Support</i>	

Chapter 1: Introduction

This chapter explains the features of the Matrox Genesis Native Library, and introduces various concepts related to the Genesis boards.

The Matrox Genesis Native Library

The Native Library for Matrox Genesis 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 Matrox Genesis board, as well as for fast application development.

The Matrox Genesis C-callable library (C-binding) that runs on the Host platform is simply a set of small stub functions, one for each function supported by the board. Each function prepares a "message" that consists of an operation code (opcode) and various optional parameters. Messages are sent to the board to perform a specific operation.

*Using MIL versus the
Genesis Native
Library*

In general, we recommend that you first consider using the hardware-independent Matrox Imaging Library (MIL), rather than the Genesis Native Library, to develop your applications. There will, however, be circumstances that will require the Genesis Native Library. These are as follows:

- When MIL does not have the required functionality. For example, MIL does not use some of the more specialized features of the grab module and display section.
- When MIL requires using more calls than the Genesis Native Library to perform the required operation. Some Genesis functions perform several operations with only one call (for example, *imIntTriadic()*). Using these functions can increase the speed of your application.
- When you want to run a particularly complicated, real-time application that requires several operations run in parallel.

- When you want to port your application to the Matrox Genesis local TMS320C80 (MVP) processor (which we will be referring to as the 'C80).
- When you want to develop your application under an environment not supported by MIL (that is, another operating system or compiler).

Using both libraries

MIL allows you to mix native (board-specific) code with its own code. Therefore, if only a portion of your application meets one (or more) of the first three criteria, you can generally use MIL to develop the bulk of the application, then integrate Genesis Native Library function calls where necessary. The main benefit of proceeding in this manner is that it makes your application as portable as possible. If you want to move the application to a different Matrox board, you will only have to change a small portion of code.

The Matrox Genesis imaging boards

The Genesis main board

The Matrox Genesis main board is a single-slot, PCI board with on-board processing, optional grab module, and optional display section. The optional grab module provides real-time acquisition (analog and digital). The optional display section provides high-resolution display for monochrome and color applications, and includes non-destructive overlay graphics. Processing power is scalable by connecting one or more Matrox Genesis main boards or processor boards.

Host interface

The Matrox Genesis main board can be a PCI bus master, and can exchange data with the Host at up to the full PCI bandwidth of 132 MBytes/sec. The actual data transfer rate that can be sustained in practice is host-dependent, but 80 MBytes/sec is attainable on a typical system.

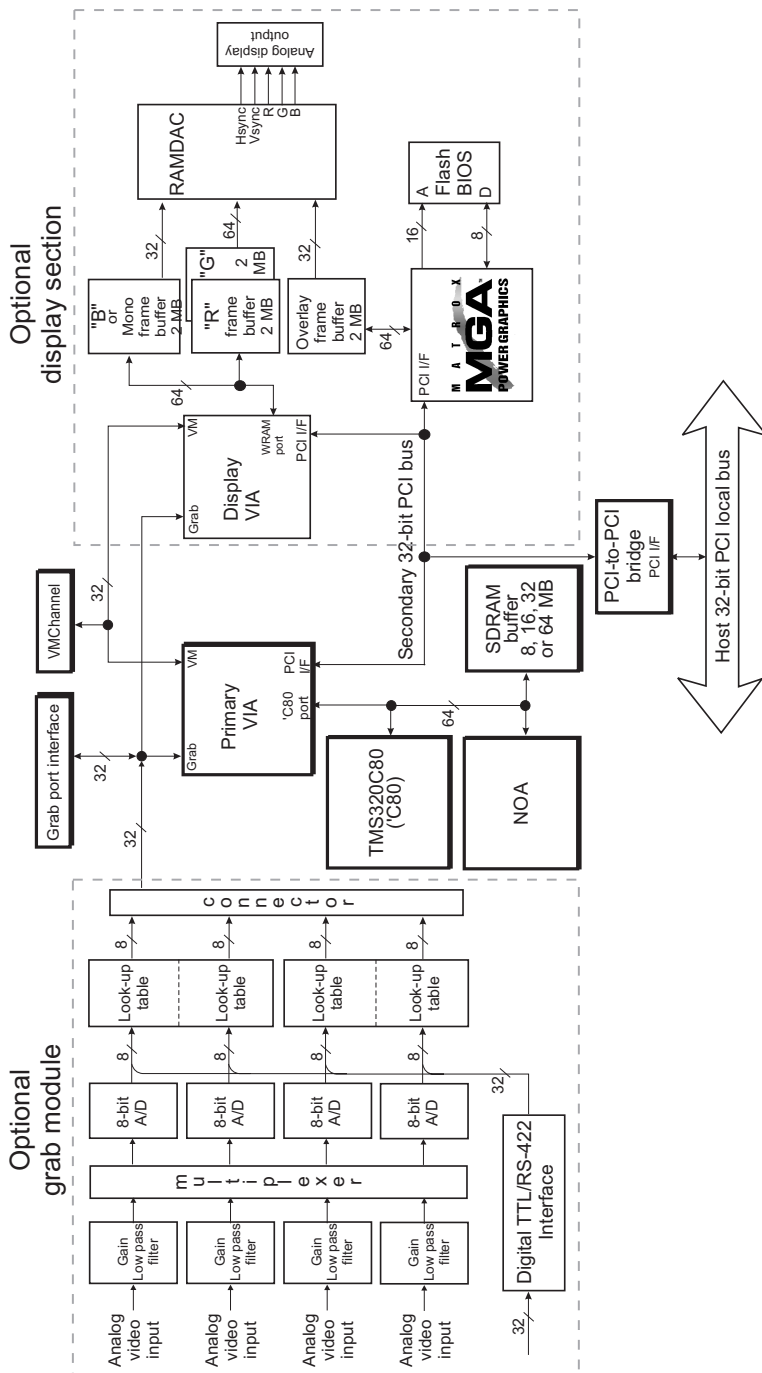
Processing

Processing is performed by the Texas Instruments TMS320C80 (also known as the 'C80) running at 50 MHz. This single-chip, digital signal multiprocessor contains five powerful, fully programmable processors: a RISC master processor (MP) and four parallel processors (PPs). The 'C80 is much more flexible than most custom processing ASICs.

An optional Matrox-designed ASIC (the Neighborhood Operations Accelerator or NOA) can further accelerate neighborhood operations such as convolutions and morphology.

There is up to 64 MBytes of on-board processing memory. Genesis addresses this memory as a single bank of linear memory. This means that there are no inherent hardware restrictions on items like image dimensions and pixel depth. Applications requiring more memory can use Host system RAM and transfer data to/from the board over the PCI bus at very high speed when needed.

Genesis Main Board



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

Display section

The optional display section has an 8-bit overlay frame buffer and either an 8-bit monochrome or a 24-bit true color main (underlay) frame buffer (depending on whether you have the monochrome or color version of the display section). The overlay and main frame buffers can be displayed at a maximum resolution of 1600 x 1200. The main frame buffer will use the same resolution as the overlay. Display memory is physically distinct from processing memory, and transfers between processing and display memory are performed in hardware by a custom chip: the Video Interface ASIC (VIA). To maintain a live display of processed images, transfers can occur in parallel with processing.

Grab module

Images can be grabbed directly into processing memory, display memory, Host memory, or any other memory mapped onto the PCI bus (you can also grab to two or more of these destinations at the same time). Grabbing is totally independent of processing operations, so an image can be acquired while a different one is being processed. This makes real-time operations much more flexible and easier to realize.

Processor boards

Genesis processor boards can be added to increase system performance. A typical processor board (which requires one extra PCI slot) has two 'C80s, each with additional memory, VIA, and optional NOA.

- ❖ A processor board can also have only one 'C80, along with additional memory, VIA, and optional NOA.

Routing data

Matrox Genesis can send image data along various paths.

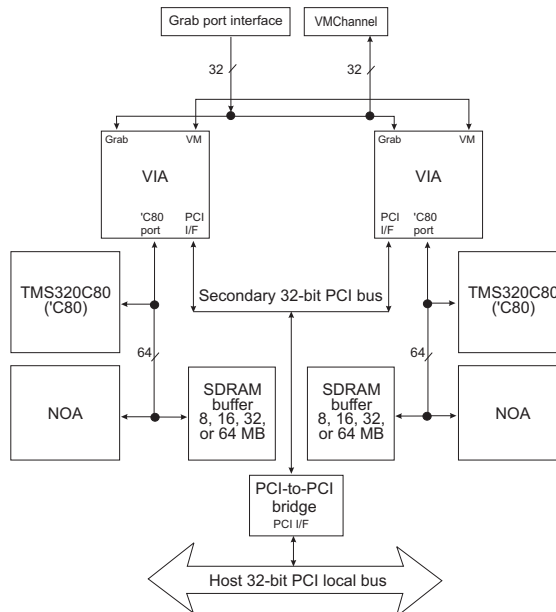
- The grab module broadcasts input data to all the video interface ASICs (VIAs) in the system, and each VIA can write to its local memory bank: processing memory (SDRAM) or display memory (WRAM).
- Using the VMChannel, Genesis can transfer data from any Genesis memory bank in a system to any other Genesis memory bank in the system. This is most commonly used to send images from processing memory to the display. The VMChannel can also transfer data from a Genesis memory bank to an external (non-Matrox) VM device.

- Using the PCI bus, Genesis can copy data between any two Genesis memory banks in a system, between a Genesis memory bank and the Host, and between a Genesis memory bank and any other memory mapped onto the PCI bus. The PCI bus is the only route that can be used to send data to/from the Host. It is also used for general communication between multiple 'C80s, and as the secondary route to transfer processing data between processing nodes.

The Genesis processor board

The Matrox Genesis processor board is basically the main board with no on-board grab module or display section. The processor board is most commonly connected to a Genesis main board, but can be used on its own. It can also be used with any other grab and/or display hardware that can send or receive data over the PCI bus or VMChannel, or that can send data over the grab port. A typical processor board has two 'C80s, each with additional memory, VIA, and optional NOA.

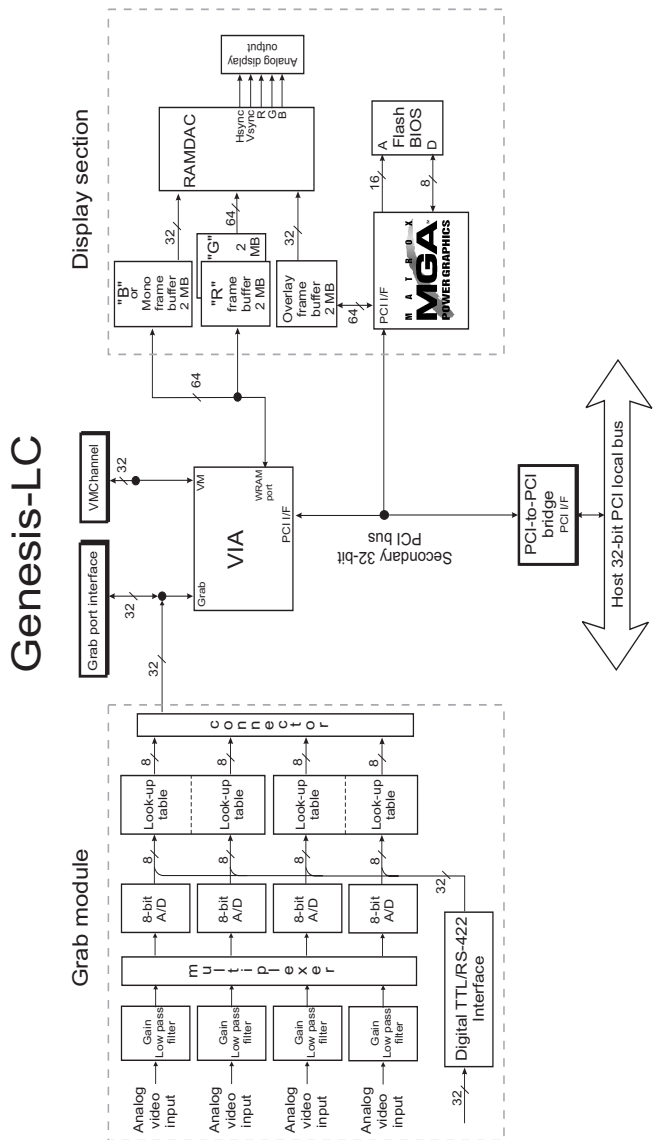
Genesis Processor Board



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

The Genesis-LC

The Matrox Genesis-LC is a low-cost version of the main board. Basically, it is the main board without a processing section. In general, this manual does not explicitly refer to the Genesis-LC because any discussion of the main board also applies to the Genesis-LC, except for discussion of the processing section.



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

Basic software concepts

You need to know very little about Genesis hardware to write simple applications. However, you should not expect to write a highly optimized, real-time application without knowing the basic architecture of the system. Refer to the *Genesis Installation and Hardware Reference* for more information.

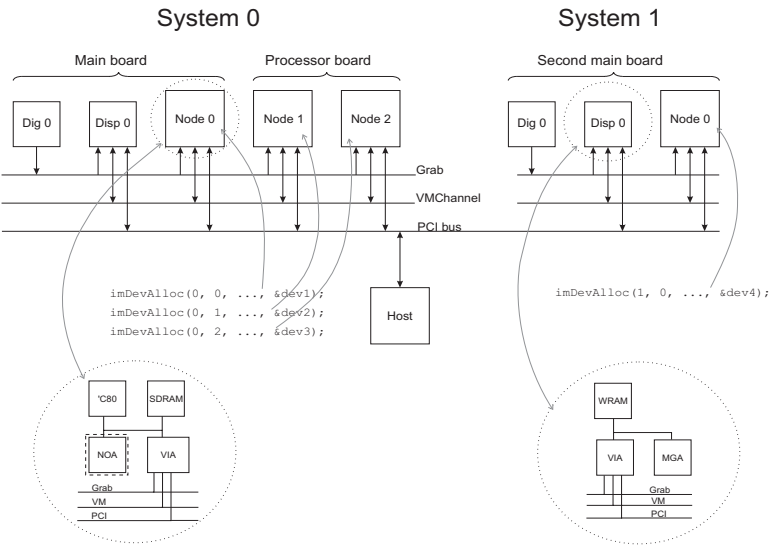
When this manual refers to resources, we mean any digitizers and displays, as well as processing nodes, threads, and buffers that you will allocate and use in your application.

Systems

Genesis uses the concept of "system" to mean 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. Systems are not considered resources as such.

Nodes

Genesis uses the concept of "node" to define the combination of a processor (the 'C80), a video interface ASIC (the VIA), and processing memory. A node can also include a NOA. The first step in any application is to assign a device ID to each "node" that you wish to use. When you do so, the Genesis shell ('C80 code) is downloaded to the node, if it is not already loaded. The Host is then responsible for sending functions to the node.



Threads

A thread, quite simply, is 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. Functions sent to the same thread execute serially (that is, in the order in which they are issued).

Typically, a real-time application has several parts that must run concurrently: acquisition, processing, transfer of data to the Host and/or display, etc. Since each thread carries out its sequence of functions independently of the others, you can allocate several threads to handle a multitasking application.

Note that there are synchronization functions to synchronize threads, when necessary.

Buffers

Buffers are used to hold any type of data, for example, image data, histogram results, LUT values, etc. They can be allocated on-board (in processing or display memory) or on the Host. For more information on buffers, see Chapters 2 and 9.

A word about examples

In an effort to simplify concepts and help you get started quickly, various examples have been provided throughout this manual. The complete source code of these examples can be found in Appendix B. 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 new examples in the \GENESIS\EXAMPLES directory.

All examples have a comment describing the minimum hardware configuration required for them to run. The expression "basic Genesis hardware" refers to one 'C80, one VIA, and processing memory (SDRAM).

The examples that grab data assume that the camera is 8-bit monochrome and that it was specified during installation. The camera should be connected to the default input channel of the digitizer.

Some systems cannot run some of the examples because they do not have the hardware capability or enough memory. You should skip these examples or modify them to suit your particular hardware configuration.

Chapter 2: Getting started

This chapter describes the basics required to create an application.

Getting started

Once you have properly installed the Genesis Native Library, you are ready to grab, process, and display images. This chapter covers the basics of acquisition, processing, transfer, and display. Subsequent chapters look at these topics in greater detail.

Basic steps

Although the main design goal of the Genesis Native Library was the ability to handle demanding real-time applications, it was also designed to be easy to learn and use. After reading this chapter, you should be able to follow most of the examples in this manual and create a simple application.

A typical Genesis application involves the following:

1. Allocate the required resources (processing node, execution threads, image buffers, etc.).
2. Acquire an image (from a file, camera, or other input source).
3. Process the image.
4. Transfer the results to the Host and/or the display.
5. Free the allocated resources.

A simple example

To familiarize you with the functions of the Genesis Native Library, we have included a simple example that writes a message in a buffer, and then displays the contents of this buffer on the screen. The example:

1. Assigns a device ID to the processing node that it uses. This is the first step of any application. You use *imDevAlloc()* to allocate a node on a system.
2. Allocates a thread. You must allocate at least one thread before you can send functions to the board. You use *imThrAlloc()* to allocate a thread.
3. Allocates a full screen display buffer, using *imBufChild()*, then clears this buffer, using *imBufClear()*.
4. Allocates a two-dimensional processing buffer, using *imBufAlloc2d()*, then clears this buffer, using *imBufClear()*.
5. Draws a rectangle (that is the size of the box that will contain the message) in the processing buffer. You use *imGraRect()* to draw a rectangle in a buffer.
6. Writes the required message ("Matrox Genesis") in the rectangle of the buffer, using *imGraText()*.
7. Copies the contents of the processing buffer to the display, using *imBufCopy()*.
8. Frees the allocated resources. Freeing resources makes them available for other applications. The last resources to be freed are the thread and the device, in that order.

The example

Note that this code is part of the *first.c* program and requires the Genesis display section. See Appendix B for the complete *first.c* program.

```

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 */

...

/* 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);

...

/* Clean up */
imBufFree(Thread, DispBuf);
imBufFree(Thread, ProcBuf);
imThrFree(Thread);
imDevFree(Device);

```

Your resources

Before performing any processing, you should allocate all the resources that you will be using with your application.

Note that the allocation functions are synchronous, that is, they do not return control to the Host until they have executed and their newly allocated ID is available. However, most other Genesis Native Library functions are asynchronous, that is, they simply queue their command to the hardware and then immediately return control to the Host.

Allocating resources

Nodes

Since other resources are allocated on nodes, the first resource to allocate is a node. Use *imDevAlloc()* to allocate a node.

Threads

Before you can send any functions to the board, you must allocate at least one thread. Use *imThrAlloc()* to allocate a thread. Functions sent to a thread execute on the node associated with that thread.

Buffers

After allocating nodes and threads, you should allocate the various buffers you will need for your application. A buffer can hold any type of data: image data, histogram results, LUT values, etc. It must be allocated before it can be used by a function. You can allocate buffers on-board (in processing memory or display memory) or on the Host. Use *imBufAlloc()*, *imBufAlloc1d()*, or *imBufAlloc2d()* to allocate a buffer.

Note that the source buffers of a processing function must be in local processing memory, that is, in processing memory on the same node as the thread which will execute the processing function. For maximum efficiency, the destination buffer of a processing function should also be in local processing memory.

Child buffers

When you need to process a rectangular region of a buffer or a specific band of a multi-band buffer, you can use a child buffer. Child buffers are discussed in Chapter 9.

Tag buffers

When you need to process a non-rectangular region of a buffer, you can use a tag buffer. See Chapter 3 for details.

Control buffers

A control buffer refers to a buffer whose control fields are used to 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.

Each control field (or simply "field") holds a single value (integer or floating-point). A field is identified by a unique "tag". The tag itself is just an integer value.

For more information on control buffers, see Chapter 9.

Freeing allocated resources

Once you have finished using a particular resource, you should free it. Use *imDevFree()*, *imThrFree()*, or *imBufFree()*, depending on the resource.

❖ The last two resources to be freed must be the thread and the device, in that order.

It is important to free resources since they are then available to other applications running under a multi-tasking Host operating system, such as Windows NT. Note that several Host applications can use the Genesis system at the same time, provided they do not hoard the board's resources and adhere to certain guidelines (see the *Running multiple applications* section).

Displaying an image

The following is an overview of how to display images on the display section of the Matrox Genesis. You will find more information in Chapter 11.

- ❖ If you have not purchased the display section, you can still display images by transferring them to the Host and using your display hardware. See the next section for information about transferring data to/from the Host.

Display memory

On Matrox Genesis, processing memory and display memory are physically distinct. Although the destination buffer of a processing function can be located in display memory, it is more efficient if it is in the memory directly attached to the 'C80 (that is, the SDRAM) and then copied to the display when necessary. In this case, display memory is being used to hold a second copy of a buffer. Therefore, you should not allocate memory in the main or overlay frame buffer of the display section. Instead, you should use *imBufChild()* to create a child buffer on the screen (at the location you wish to display the image) and then copy the processed buffer to this on-screen child buffer when you need to see it; see the example in the *Getting started* section.

Grayscale and color images

Note that there are two versions of the display section: a monochrome version and a color version. When you set the *Buf* parameter of *imBufChild()* to *IM_DISP*, the resulting on-screen child buffer will automatically have one band on the monochrome version and three bands on the color version.

Copying a buffer to the display

To copy a buffer in processing memory to the display, use *imBufCopy()*. You can also use *imBufCopyVM()* or *imBufCopyPCI()*. *imBufCopyVM()* and *imBufCopyPCI()* are specialized functions that can format data in a variety of ways during the copy; see Chapter 9 for details.

Allocating a display

If you have more than one display in your system, you must allocate each one, so you can later specify to functions that have a "display" parameter which display you want to use. Use *imDispAlloc()* to allocate a display. If you have just one display, there is no need to allocate it because functions that have a "display" parameter will use it by default.

Transferring to/from the Host

Transferring data to/from the Host can be used to read back results (such as histogram results) and to load values into buffers (such as kernel and LUT buffers). It can also be used if you have not purchased the display section of the Genesis and need to use the Host display hardware to view images.

To transfer data from a buffer to the Host, use *imBufGet()*, *imBufGet1d()*, or *imBufGet2d()*. When using one of these functions, you must first allocate Host memory to which to send your data, and specify the Host memory address to which to send it.

To transfer data from the Host, use *imBufPut()*, *imBufPut1d()*, or *imBufPut2d()*. Note that these are synchronous functions because the Host performs the transfer.

❖ If you have allocated a buffer directly on the Host (using *imBufAlloc...()*), it is faster to use *imBufCopy()* to transfer data to/from the Host. See Chapter 9 for details.

An example

The following code transfers histogram results from a processing buffer to the Host. Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```
long HistBuf;          /* Histogram result buffer */
long HistVals[256];    /* Host array to hold histogram result */

...

/* 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);
```

Grabbing an image

With the Genesis Native Library, you can grab images from a wide variety of input sources. The following is an overview of how to grab images with the Genesis Native Library. You will find more information in Chapter 10.

- ❖ Since the most common input source is a video camera, the words input source and camera are used interchangeably in this manual.

Grabbing

To grab an image with the Genesis Native Library, you must first allocate a camera definition that matches your camera type, using *imCamAlloc()*. If you have more than one digitizer in your system, you must also allocate the digitizer with which to grab, using *imDigAlloc()*. You then pass the camera definition identifier and if required, the digitizer identifier, to the grab command (*imDigGrab()*).

Note that the buffer in which to grab can be located anywhere in your system (in processing or display memory on any node). In addition, you can grab a specific number of lines, fields, or frames, or you can continuously grab frames until you call *imThrHalt()*.

The camera definition

The *imCamAlloc()* function takes a file that specifies the parameters of your camera and returns an ID. The camera definition or digitizer configuration file (*.dcf*) includes such information as pixel rate, timings of synchronization signals, channel number, and gain and offset settings. If the camera definition file is given as NULL to *imCamAlloc()*, an ID is returned for the default camera definition file that you specified during software installation.

Note that several predefined camera definition files are available for you to choose from when installing (in the \GENESIS\DCF directory). If none of these match your camera type, you can use Matrox INTELLICAM to create a custom camera definition file.

Grabbing to two or more buffers

With the Genesis Native Library, you can grab to two or more buffers, allocated in different memory banks, at the same time. This can be useful when you want to grab to all nodes in your system, or when you want to simultaneously grab to processing and display memory.

Grab options

When you call *imDigGrab()*, you can specify a number of options (such as zooming and subsampling) through the control buffer passed to this function. Many of these options are particularly useful when grabbing to the display.

An example

The following code continuously grabs frames into a display buffer, until halted by the user. Note that this code is part of the *grab.c* program and requires the Genesis grab module and display section. See Appendix B for the complete *grab.c* program.

```

long Device;          /* Genesis device */
long Thread;          /* Thread to execute all functions */
long DispBuf;         /* Buffer allocated in display memory */
long Camera;          /* Camera */
long SizeX, SizeY;    /* Image Size */

...

/* 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);

/* 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);

```

Error reporting

Most functions in the Genesis Native Library are asynchronous, that is, they queue their command to the hardware and then immediately return control to the Host. As such, the return values of most functions cannot indicate whether the function performed successfully. Synchronous functions could return a meaningful error value, but this would be tedious to check after each call. For these reasons, errors are only reported when requested and not through the return values of functions.

One way to check for errors is to use *imThrGetError()*. This function returns the first error detected in a thread since error information about the thread was last cleared. You can also use *imAppCatchError()* or *imAppGetError()* to check for errors. *imAppCatchError()* 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. *imAppGetError()* returns the first error detected in an application since error information about the application was last cleared.

Note that the error messages provided with the Genesis Native Library functions are function-specific and intended to be self-explanatory.

Places to check for errors

To some degree, the placement of the error checking functions *imAppGetError()* and *imThrGetError()* is application-dependent. However, there are a few places where they should normally be used:

- After the initialization section of the application, where buffers and other resources are usually allocated.
- At the end of the application, just before freeing buffers and other resources.

For more information on error reporting and mechanisms, see Chapter 12.

Synchronization

In the Genesis Native Library, all functions are sent to a specified thread, and execute on the node associated with this thread. Functions sent to the same thread execute serially. Threads execute independently of one another, allowing operations to run in parallel. However, you can use an Operation Status Block (OSB) to synchronize two different threads or to determine the state of a particular asynchronous function.

Operation Status Block

An OSB is a block of memory that you allocate using *imSyncAlloc()*. All asynchronous functions have an "OSB" parameter. You can set this parameter to 0 or you can pass an OSB ID. If you pass an OSB ID, status information will be written about that function in that OSB. You can then use *imSyncHost()* to halt execution on the Host until that function is in a specified state (for example, until that function has completed). You can also use *imSyncThread()* to synchronize two different threads. The *imSyncThread()* function halts execution on one thread until a function in another thread is in a specified state.

Re-using OSBs

An OSB can be re-used with another function after the previous one has completed, and after the OSB has been reset to the waiting state (using *imSyncControl()*). Note that the OSB state is automatically reset after its function completes if the Host or another thread is waiting for it.

An example

The following code grabs and processes in parallel. Since an image must be grabbed before it can be processed, an OSB is associated with the grab command and then *imSyncHost()* is used to ensure that this OSB is in the required state (IM_COMPLETED) before processing. Note that this code is part of the *tfilter.c* program and requires the Genesis grab module. See Appendix B for the complete *tfilter.c* program.

```

long GrabOSB[2];          /* OSBs used for synchronization */
...

/* Allocate OSBs for synchronization */
imSyncAlloc(Thread, &GrabOSB[0]);
imSyncAlloc(Thread, &GrabOSB[1]);

/* Select asynchronous grab mode */
imBufPutField(Thread, GrabCtrlBuf, IM_CTL_GRAB_MODE,
               IM_ASYNCHRONOUS);

/* First grab */
imDigGrab(Thread, 0, Camera, InBuf[0], 1, GrabCtrlBuf, GrabOSB[0]);

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);
}

```

Running multiple applications

It can be very useful to run several applications simultaneously on your Genesis system. For example, if you want to display several images at once, you can write a very simple application that loads and displays just one image, and then run multiple copies of this application simultaneously. If you have a real-time processing application and occasionally need to display the input image, you can write a small, separate application that simply grabs into display memory, then run this application simultaneously with the real-time application (when necessary).

In general, the ability to run several applications simultaneously allows you to write a series of small, simple, re-usable applications, rather than a large, complex application. However, when you run applications simultaneously, you need to follow certain guidelines to ensure that the resources of the Genesis system are shared properly.

General guidelines

The following are some general guidelines you should follow if you run applications simultaneously:

- When you allocate a device (using *imDevAlloc()*), set the *ShellFile* parameter to NULL and the *Mode* parameter to IM_DEFAULT. This will ensure that the device is not initialized more than once, even though it is allocated by each application.
- Write the applications so that they "fail gracefully" (that is, clean up after themselves) if there are not enough resources for them to run. This involves checking for errors. Note that, to reduce the likelihood of failure, you should not allocate more resources than are necessary. In addition, you should free all allocated resources before the application terminates, since they are not freed automatically.
- Avoid queuing more commands to the board than necessary. If an application requires a loop, try to include a synchronous function within the loop (such as *imSyncHost()*). This can prevent the command queue on the board from filling up (refer to Chapter 4 for details).

Other guidelines

For guidelines that deal specifically with the display or digitizer, see Chapter 10 or 11, respectively. Some of these guidelines also ensure that the applications will run regardless of the display mode or camera type.

Chapter 3: Processing functions

This chapter gives an overview of the processing functions available with the Matrox Genesis Native Library.

General overview

The Genesis Native Library includes a wide variety of processing and statistical operations. The functions that can perform these operations are organized according to the data type that they support. Specifically, functions that can perform these operations on packed binary buffers, integer buffers, and floating-point buffers are the *imBin...()* functions, the *imInt...()* functions, and the *imFloat...()* functions, respectively.

In general, all buffers passed to a processing function should have the same size. If they do not, only the area intersected by all the buffers, starting from the top-left corner, will be processed.

Processing operations

Processing operations result in a new image. There are three main types of processing operations:

- Point-to-point
- Neighborhood
- Geometric

Statistical operations

Statistical operations extract information from an image. A histogram is an example of a statistical operation.

Color processing

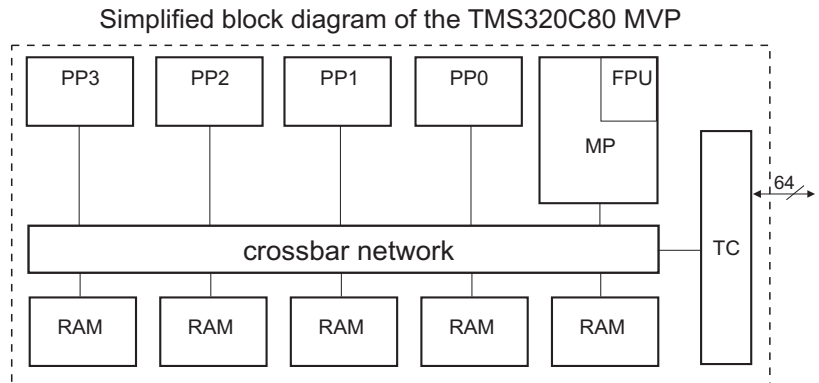
Many processing functions can operate on multi-band (color) buffers. Specifically, if the source and destination buffers have the same number of bands, the function processes corresponding source bands and writes results to the corresponding destination band. If one of the source buffers has only one band while the other(s) have several bands, the function uses that one source band as many times as is needed.

When a function does not support multiple bands, you can create a child buffer for each band, using *imBufChildBand()*, and process the bands individually.

Basic architecture of the 'C80

Most processing and statistical operations on the Genesis are performed by Texas Instrument's 'C80. This is a single-chip multiprocessor device. It includes:

- Four parallel processors (PP0 - 3). These are advanced, 32-bit integer DSPs.
- A 32-bit RISC master processor (MP) with an IEEE-754 floating-point unit (FPU).
- A transfer controller (TC). This manages transfers between on-chip and off-chip RAM.
- A high-speed bus switching network between the processors and on-chip RAM (the crossbar network).



Because it is fully programmable, the 'C80 is much more flexible than custom ASICs or other specialized hardware, so in those few cases where the MIL and Native Library functions are insufficient, you can program the 'C80 directly. To do so, you need to use the optional Genesis Developer's Toolkit, in conjunction with Texas Instruments' TMS320C8x software development tools. Note that the 'C80 is a complex chip, so programming it should not be undertaken lightly, even though it gives you complete access to all features of the board.

Data types

The Genesis Native Library supports the following data types:

- Packed binary
- Integer
- Floating-point

Processing

In general, the fewer bits per pixel in the buffer(s), the faster the processing function. Therefore, when possible, you should use binary buffers for binary data (rather than, say, 8-bit integer buffers with only the values 0 and 0xFF). This will not only speed up binary processing, it will also maximize your storage space (in packed binary format, pixels are packed 1 bit per pixel, that is, in a format eight times smaller than an 8-bit image). You should only use integer buffers for binary data when the required function does not support packed binary buffers.

When using integer buffers, use 8 bits per pixel when possible, 16 bits per pixel if necessary, and 32 bits per pixel only as a last resort. When you need extra precision or greater dynamic range, you can use floating-point buffers.

❖ If you want to display an image, it must be limited to 8 bits per pixel per band. If necessary, convert the image before copying it to the display, or use either *imBufCopyVM()* or *imBufCopyPCI()* to format the image during the copy. Converting between data types is discussed in the next section; *imBufCopyVM()* and *imBufCopyPCI()* are discussed in Chapter 9.

RGB packed

In addition to the above data types, the Genesis Native Library allows you to perform a limited number of operations on data that is in a RGB packed format. In the RGB packed format, there are no color bands; rather, the color components of each pixel are interleaved, as shown below:

RGBRGBRGBRGBRGBRGB ...
RGBRGBRGBRGBRGBRGB ...

The Genesis Native Library does not provide functions to process and/or display the RGB format directly. However, once you have grabbed or loaded your RGB-packed image into a buffer, you can automatically unpack it for display by copying it to a 3-band display buffer, using *imBufCopy()*.

Converting between types

The following functions convert between the supported data types:

- *imBinConvert()*
- *imFloatConvert()*
- *imIntConvert()*

imBinConvert()

Use *imBinConvert()* to convert between integer and packed binary buffers. For integer to binary, the conversion result is 1 when a specified threshold condition is true, and 0 otherwise. For binary to integer, 0s are converted to a specified value, and 1s to a second specified value.

imFloatConvert()

Use *imFloatConvert()* to convert between integer and floating-point buffers. For floating-point to integer, you can round towards zero or round to the nearest integer. Overflows or underflows are set to the maximum or minimum value, respectively, of the destination buffer's data type.

imIntConvert()

Use *imIntConvert()* to convert from one integer data type to another. When converting to a larger pixel size, data is sign-extended. Sign extension propagates the sign bit (most-significant bit) if the source buffer is a signed type; if not, it propagates 0. When converting to a smaller pixel size, the higher bits are discarded.

When you use *imIntConvert()*, you can clip 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).

Processing a specific region of an image

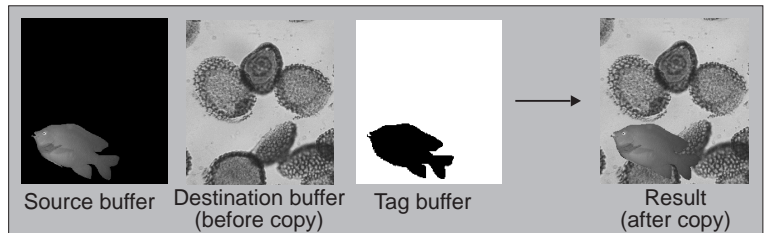
Rectangular region

There will be times when you won't want to process your entire image. To restrict processing to a rectangular region, you can create a child buffer, using *imBufChild()*. For more on child buffers, see Chapter 9.

Non-rectangular region

You can restrict processing to a non-rectangular region by:

- Processing the entire area, then using either *imBufCopyPCI()* or *imBufCopyVM()* with a tag buffer (see below), or using *imIntTriadic()* to merge the source and destination buffers (see Chapter 4). You can only use *imBufCopyPCI()* or *imBufCopyVM()* if you are copying between different memory banks, for example, between processing and display memory. If you are copying within the same memory bank, you must use *imIntTriadic()*.



- Packing the required region using *imBufPack()*, processing the packed buffer, and then, if necessary, unpacking it (also using *imBufPack()*). Packing copies selected pixels of a buffer to a one-dimensional destination buffer. Unpacking copies pixels from a one-dimensional buffer to selected positions in a destination buffer. A tag buffer controls which pixels of the buffer to copy (when packing), or which pixels of the destination buffer to overwrite (when unpacking). If you use the same tag buffer to pack a buffer and then to unpack the resulting buffer, the packed pixels will be written to their original positions.
- ❖ Tag buffers are explained in detail in Chapter 9.

More on packing

Any type of processing can be used on the one-dimensional packed buffer, provided it can be performed on the buffer's data type. However, only point-to-point operations are useful, since these operations do not produce results based on neighborhood information.

Packing vs. copying

In general, it is faster to use *imBufPack()* because only the required region, and not the entire buffer, is processed. However, the time required to pack and unpack might exceed the extra processing time if, for example, you need to process a large region or if you require just one function to perform the required processing.

An example

The following code uses *imBufPack()* to process a non-rectangular region of an image. Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```

long TagBuf;      /* Binary 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 */
imBufAlloc2d(Thread, SizeX, SizeY, IM_BINARY, IM_PROC, &TagBuf);

...

/* Allocate 1-d buffer for packed pixels (make sure it's big enough) */
imBufAlloc(Thread, SizeX*SizeY, 1, NumBands, IM_UBYTE, IM_PROC, &PackedBuf);

/* 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);

/* 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);

```

Point-to-point processing

The Genesis Native Library supports flexible point-to-point operations. A point-to-point operation (unlike a neighborhood operation) does not use a pixel's neighbors when determining the pixel's new values. Examples of point-to-point operations are LUT mappings, arithmetic operations, logical operations, and thresholds.

I/O bound

Point-to-point operations tend to be I/O bound. Therefore, smaller data types are usually processed faster.

In-place supported

In-place operation, but not partially overlapping source and destination buffers, is supported for point-to-point functions.

Mixed data types

As mentioned, not all functions in the Genesis Native Library support all data types. However, the commonly used point-to-point functions *imIntMonadic()* and *imIntDyadic()* do support all integer types, in all combinations.

For these functions, source operands are cast to an internal processing type: by sign extension for signed operands, and by zero extension for unsigned operands. When applicable, saturation is performed on results that overflow or underflow the range of the internal processing type. Final results are cast to the destination type (if necessary) by discarding the high bits.

The internal processing type is chosen as follows:

- Take the smallest type that can represent the full range of both source operands (this is bound to be a signed type if either source is signed). If one operand is constant, the rule still applies.
- If the chosen type has fewer bits than the destination type, promote the type to the same size as the destination. (The destination sign is not taken into account).

Saturation and clipping

For certain operations, *imIntDyadic()* supports both clipping and saturation. Note that clipping is performed when results overflow or underflow the range of the destination buffer; saturation is performed when results overflow or underflow the range of the internal processing type. Therefore, saturation and clipping do not produce the same results when the destination buffer type is different from the internal processing type (for example, when the sources are unsigned 8-bit and the destination is signed 8-bit).

LUT mappings

The Genesis Native Library supports LUT mappings in software, using *imIntLutMap()*. The *imIntLutMap()* function maps an integer image from a source buffer (Src) through a specified look-up table (LUT) and stores the results in a specified destination buffer (Dst). Such mappings can reduce a multi-step or complex operation to a single-step LUT mapping. They can also be used to perform operations not supported by the Genesis Native Library.

Mapping a displayed image

Note that, in single-screen mode, the overlay frame buffer uses all the LUTs in the RAMDAC. Therefore, to map the contents of the main frame buffer in single-screen mode, you must use *imIntLutMap()*. For the details, see Chapter 11.

Generating a LUT

To generate a LUT, allocate a one-dimensional buffer using *imBufAlloc1d()* and then load data into this buffer in one of two ways:

- Use *imGen1d()* (or any other processing function) to generate the data.
- Generate the data on the Host, and then transfer the data to the buffer, using *imBufPut()*.
- ❖ None of the source pixel values can exceed the size of the table; that is, the number of entries in the LUT. For example, if the LUT is size 1024, the maximum source pixel value should be 1023. Pixel values above 1023 will not wrap around; instead, they might cause an invalid memory access.

An example

The following code maps an image through a LUT. Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```
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);
```

LUT performance

The 'C80 performs LUT mappings very efficiently when the LUT fits entirely in on-chip RAM. When the LUT is larger, performance drops and becomes data-dependent, that is, varies with the image being processed. Therefore, to guarantee maximum speed when using *imIntLutMap()*, you should always use the smallest possible LUT. For example, 10-bit data must be stored in a 16-bit buffer, but only requires a LUT of size $2^{10} = 1024$.

A note about notation

In the following sections, the notation *x:y* means that x-bit data is being mapped to y-bit data. For example, a 12:32 mapping means that 12-bit data is being mapped to 32-bit data. Note that, in this case, the required LUT size is:

$$2^{12} * 4 \text{ Bytes} = 16 \text{ KBytes.}$$

$$(\text{number of entries}) * (\text{data depth}) = \text{LUT size}$$

In the equations used to describe LUT mapping operations, **Src** refers to the source buffer, **Lut** refers to the LUT buffer, and **Dst** refers to the destination buffer, respectively, that is used in a LUT mapping operation.

Small LUTs

LUTs smaller than 4 KBytes (for example, 12:8, 11:16, and 10:32) fit entirely in each PP's internal memory. In this case, performance is not data-dependent, that is, performance depends only on the size of the table and on how fast the input data can be accessed.

Medium size LUTs

LUTs bigger than 4 KBytes but smaller than 16 KBytes (for example, 12:32, 13:16, and 14:8) can still fit on chip, but each PP cannot have its own copy of the LUT. In this case, performance is slightly data-dependent. Specifically, execution times increase slightly when the image pixels are concentrated in a small range of values. For most other images, execution times should be fairly constant.

Large LUTs

LUTs bigger than 16 KBytes (for example, 15:8, 16:8, and 16:32) cannot fit entirely on chip. You can process such tables by applying a mapping mode that uses either a non-interpolated LUT or an interpolated LUT.

Mapping with a non-interpolated LUT

A mapping operation that is performed with a non-interpolated LUT does not subsample the LUT buffer, so the actual values in the LUT buffer are used. The *imIntLutMap()* function provides three non-interpolated modes:

- Basic mode
- Shift & mask mode
- Clip mode

Basic mode

The basic non-interpolated mode uses the transfer controller of the C80 to process large LUTs. This occurs when no control fields are added to the LUT buffer to specify a different mode. The operation used is simply:

$$\mathbf{Dst} = \mathbf{Lut}[\mathbf{Src}]$$

In this basic mode, the source pixel is used directly as an index into the LUT and it is your responsibility to make sure that no pixel value in the source buffer exceeds the size of the LUT buffer. For example, if you pass a LUT buffer with 1024 entries, it is assumed that the source buffer contains 10-bit unsigned data, and you should ensure that all pixel values in the source buffer are between 0 and 1023 (inclusive). In this mode, the source buffer can be 8-, 16-, or 32-bit.

Be aware that when the source contains negative values, you must force the unwanted sign bits to 0. For example, the difference of two 8-bit unsigned values lies in the range [-255, +255], but if you then want to apply a LUT with only 512 values, you must first mask (AND) the pixels with 0x01ff (511) to prevent negative numbers from being considered as very large positive values.

For LUTs bigger than 16 KBytes, the function can be quite slow in this basic 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.

The default option

Using the default option, large LUTs are processed using the transfer controller of the 'C80, making the operation strongly data-dependent.

The PP option

Using the PP option, you can increase the speed of the operation for LUTs that are bigger than 16 KByte 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.

Performance drops if this work buffer is too small, so be sure that you have enough memory for a large work buffer. Keep in mind that the results of the mapping operation will be the same as those obtained when using the default option, and the work buffer will not necessarily improve the speed in all cases.

You can have this work buffer automatically allocated, or you can supply it yourself. To specify whether you want an automatically-allocated or user-supplied work buffer, you have to add the IM_CTL_WORK_BUF field to the LUT buffer passed

to *imIntLutMap()*. When you add this field to the LUT buffer, *imIntLutMap()* will use the PP option for LUTs bigger than 16 KBytes; otherwise, it will use the default option.

Default vs. PP option

When you use the default option, performance is strongly data-dependent. Specifically, images tend to the best case when the image pixel values change slowly along a line, and tend to the worst case when the pixel values vary rapidly. Since typical images tend to be closer to the worst case than the best case, it is generally better to use the PP option, which is not very data-dependent. However, for 16:32 mappings, the default option is generally better than the PP option.

Other non-interpolated LUT mapping modes

To increase performance with large LUTs, it is also possible to apply one of the two other types of mapping modes that use a non-interpolated LUT.

Shift & Mask mode

The shift and mask mode is a non-interpolated LUT mode of *imIntLutMap()* that 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). To select this mode, you need to specify the dynamic range of the source pixels with the IM_CTL_INPUT_BITS field. In this mode, the source buffer can be 16- or 32-bit. Note that because of the shift, the source's dynamic range can not be deduced from the size of the LUT buffer. Instead, the shift and mask are deduced from the difference between the number of input bits and the size of the LUT buffer. For example, a LUT mapping with 15-bit source data and a 10-bit LUT (1024 input bits) would lead to a right shift of 5 bits (15 - 10), and a mask value of 0x03ff (1023), which is useful if the source buffer is signed and there are any negative input values. The value given for the IM_CTL_INPUT_BITS field must lie in the range 9-16 for a 16-bit source buffer, and 17-32 for 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:

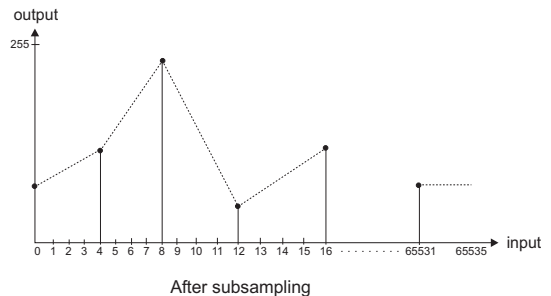
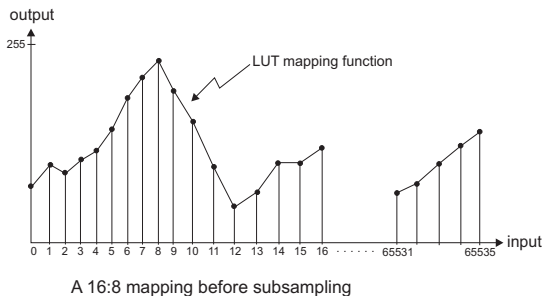
$$\mathbf{Dst} = \mathbf{Lut}[\mathit{clip}(\mathbf{Src})]$$

This is useful when you have large values in the source buffer (values larger than the LUT you want to use). To select this mode, you need to enable clipping (set the IM_CTL_CLIP field of the LUT buffer to IM_ENABLE). 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. Note that when you want this mode, you should not add the IM_CTL_INPUT_BITS field to the LUT buffer as well.

Mapping with an interpolated LUT

Interpolated LUT mappings

If you are performing a 16:8 or 16:32 mapping, you can improve performance by using an interpolated LUT mapping. An interpolated LUT mapping subsamples the LUT buffer to reduce its size to 16 KBytes (a 16:8 LUT is therefore subsampled by 4 and a 16:32 LUT by 16). The output for each 16-bit input is then determined by linearly interpolating between two values of the subsampled LUT.



To specify an interpolated LUT mapping, add the `IM_CTL_RESAMPLE` field to the LUT buffer passed to *imIntLutMap()*. Note that final results will be equal to the actual results when the mapping function varies linearly between the sampled points. Since even a fairly high-order polynomial function varies almost linearly over a range of 16 consecutive input values, final results will usually be very close to the actual results.

Since a 16:32 mapping is usually used to display an image in pseudo-color, a 32-bit destination buffer is always assumed to be in RGBA format, and each of the three 8-bit color bands is interpolated separately.

If necessary, input values from the source buffer can be clipped to avoid reading beyond the end of the subsampled LUT. When clipping is enabled (`IM_CTL_CLIP` is set to `IM_ENABLE`), these large input values are clipped to the maximum value that the subsampled LUT can handle. Similarly, if you want to disable clipping (usually because your input data is already clipped), set the `IM_CTL_CLIP` field to `IM_DISABLE`.

When you perform an interpolated LUT mapping, you might prefer to provide the LUT already subsampled, since it will be quicker to generate and load. However, you then need to specify the size of your input data, since this cannot be inferred from the LUT size (if you provide a LUT of 16K entries, the function would normally assume you have 14-bit data, when in fact you have 16-bit data).

To specify the input data size, use the `IM_CTL_INPUT_BITS` field.

The above descriptions assume that you are using an interpolated LUT (`IM_CTL_RESAMPLE` field is set to `IM_INTERPOLATE`). The `IM_CTL_INPUT_BITS` and `IM_CTL_CLIP` control fields have different meanings when you use a non-interpolated LUT.

Use smaller mappings

Instead of working with large LUTs, you should always try to use several smaller LUT mappings; performance will often be better. For example, a 14:32 mapping is considerably faster if

done as three separate 14:8 mappings. Note that a 14:32 mapping can be used to convert a grayscale image to pseudo-color.

Histogram equalization

Using *imIntHistogramEqualize()*, you can perform a histogram equalization on an integer image, or generate a LUT from a specified histogram equalization operation. 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 from the specified equalization operation; the histogram can be of the source image or user-supplied.

Neighborhood processing

The Genesis Native Library supports a variety of neighborhood operations. A neighborhood operation 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 kernel is used to determine new pixel values.

Compute bound

Neighborhood operations are usually compute-bound, but the type of data can still affect performance. This is because each of the parallel processors of the 'C80 can often process four 8-bit or two 16-bit pixels as quickly as one 32-bit pixel. Speed also depends on the size of your kernel.

In-place not supported

In-place operation, as well as partially overlapping source and destination buffers, are not supported for neighborhood functions, unless otherwise stated.

Predefined kernels vs. custom kernels

In general, you can use your own kernel or a predefined kernel. The predefined kernels will normally execute faster, although they might not always meet your requirements. If you use your own kernel, you can generally control the center pixel of the kernel, as well as other aspects of the operation.

When the NOA is not used (either not present or disabled), nearly all functions support a maximum kernel size of 15x15, except for binary pattern matching, which has no kernel size limit. When using the NOA for binary morphology, the largest kernel supported is 32x32, (again, except for binary pattern matching). However, the NOA can only operate on aligned binary data (an image which starts on a byte boundary and has a width which is a multiple of 16 pixels). If you do pass a non-aligned buffer, the operation will be carried out by the 'C80 instead of the NOA, and the 15x15 maximum kernel size will apply.

Main types

There are two main types of neighborhood operations:

- Spatial filtering (convolution) operations.
- Morphological operations.

Spatial filtering operations

A spatial filtering (convolution) operation determines the new value for a pixel based on the weighted sum of the pixel and the pixel's neighboring values. You perform convolutions using *imIntConvolve()*.

*Using your own
kernel*

If you use your own kernel with *imIntConvolve()*, you can shift, take the absolute value of, and/or clip the results of the convolution.

Note that, if clipping is enabled, it is to the range specified by the IM_CTL_OUTPUT_BITS field (by default, it is to the full range of the destination buffer).

Processing speed

Depending on the kernel values, you might be able to increase the speed at which *imIntConvolve()* is performed by setting its IM_CTL_COMPUTATION field to IM_FAST. This will cause approximations to be made, if possible, so as to increase operation speed. Note, however, that some rounding errors (usually small) can occur if you set this field.

You might also be able to increase operation speed by setting IM_CTL_INPUT_BITS to the number of bits actually in the source buffer. 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 (again, whether operation speed can actually be increased depends on the kernel values).

Note that *imIntConvolve()* runs slower if you shift, take the absolute of, or clip results. You should only use these options if really necessary.

Morphological operations

The Genesis Native Library supports the following morphological operations:

- Erosion.
- Dilation.
- Thinning.
- Thickening.
- Binary template matching.
- Hit-or-miss transformation.
- Distance transform.

Erosion and dilation

There are two types of erosion/dilation operations: binary and grayscale.

Binary erosion and dilation

To perform binary erosion or dilation, use *imBinMorphic()*.

- In binary erosion, if the kernel does not match the neighborhood exactly, the center pixel is set to zero; otherwise, it remains unchanged.
- In binary dilation, if any of the elements of the neighborhood match the corresponding kernel value, the center pixel is set to 1; otherwise, it remains unchanged.

Note that any kernel value other than 0 or 1 is considered a "don't care" value, that is, it is ignored during the operation.

*Grayscale erosion
and dilation*

To perform grayscale erosion or dilation, use *imIntErodeDilate()*. The kernel must be grayscale. A kernel value of IM_DONT_CARE causes the corresponding pixel in the neighborhood to be ignored during the operation.

Grayscale erosions and dilations are done in two steps.

The erosion operation:

1. Subtracts each kernel value from the corresponding pixel value in the neighborhood.
2. Replaces the center pixel of the neighborhood with the minimum value from the resulting neighborhood values.

The dilation operation:

1. Adds each kernel value to the corresponding neighborhood pixel.
2. Replaces the center pixel of the neighborhood with the maximum value from the resulting neighborhood values.

Thinning and thickening

In thinning and thickening operations, the value of the center pixel is determined by whether an exact match of the neighborhood and the kernel is found.

There are two versions of thinning and thickening: binary and grayscale. For either version, any kernel value other than 0 or 1 is considered a "don't care" value, that is, it is ignored during the operation.

Binary thinning and thickening

Use *imBinThin()* to perform a fast binary thinning operation.

- Binary thinning: This operation replaces the center pixel by the value 0 if a pixel's neighborhood matches the kernel exactly. If the neighborhood does not match, the pixel value remains unchanged.

Use *imBinMorphic()* to perform a binary thickening operation.

- Binary thickening: This operation replaces the center pixel by the value 1 if the pixel's neighborhood matches the kernel exactly. If the neighborhood does not match, the pixel value remains unchanged.

❖ Note that it is also possible to use *imBinMorphic()* to perform a binary thinning operation, although processing will be slower.

When performing binary operations using *imBinMorphic()*, the NOA can only directly process buffers which are byte aligned, and have a width which is a multiple of 16 pixels. If a buffer does not respect these restrictions, processing will be slower (either because the NOA will not be used, or an aligned copy of the buffer will be made first). Note that this restriction only applies to child buffers, since buffers are always initially allocated with the proper alignment.

Grayscale thinning and thickening

Use *imIntThickThin()* to perform a grayscale thinning or thickening operation.

- Grayscale thinning:
 - if $\text{MAX}(0) < \text{center pixel} \leq \text{MIN}(1)$
 - center pixel = $\text{MAX}(0)$
 - else
 - center pixel is unchanged.
- Grayscale thickening:
 - if $\text{MAX}(0) \leq \text{center pixel} < \text{MIN}(1)$
 - center pixel = $\text{MIN}(1)$
 - else
 - center pixel is unchanged.

where $\text{MAX}(0)$ is the maximum of all pixels in the neighborhood that correspond to zero in the kernel, and $\text{MIN}(1)$ is the minimum of all pixels in the neighborhood that correspond to one in the kernel.

Multi-band kernels

Since it is common to thin or thicken with a series of different kernels (one applied after the other), you can provide a multi-band kernel to *imBinMorphic()* or *imIntThickThin()*. Each band of the kernel will be applied to the result of the previous one, allowing you to perform a series of operations with one call to the function.

An example

The following code is an example of thinning to skeleton. Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```

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 */
imBinMorphic(Thread, Bin1Buf, Bin2Buf, SkelBuf, IM_THIN,
             IM_IDEMPOTENCE, SkelBuf, 0);

```


Binary template matching

Binary template matching allows you to determine how similar certain areas of a binary image are to a pattern (specified by a kernel). Binary template matching produces resulting pixels that are a count of the number of matches between the neighborhood and kernel values (therefore, the result is a grayscale image). Any kernel value other than 0 or 1 is considered a "don't care" value, that is, it is ignored during the operation.

Use *imBinMorphic()* to perform binary template matching.

Hit-or-miss transformations

A hit-or-miss operation determines which pixels have neighborhoods that match a pattern exactly. When the neighborhood of a source image's pixel matches the pattern exactly, the value of the corresponding pixel in the destination image is set to 1. When the neighborhood does not match exactly, the pixel value is set to 0. Any kernel value other than 0 or 1 is considered a "don't care" value, that is, it is ignored during the operation.

Use *imBinMorphic()* to perform hit-or-miss operations.

Defining your own kernel

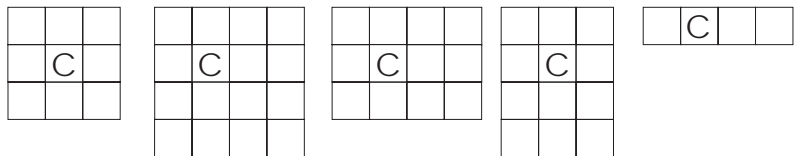
If the predefined kernels do not meet your requirements, you can define your own kernel, as follows:

1. Allocate a kernel buffer using, for example, *imBufAlloc2d()*. The dimensions of the kernel determine the size of the neighborhood used in the operation.

The result of the neighborhood operation is stored in the destination buffer at the location corresponding to the kernel's center pixel. By default, the coordinates of the kernel's center pixel are:

$(\text{int } (\text{Xsize}-1)/2, \text{int } (\text{Ysize}-1)/2)$

where Xsize by Ysize are the dimensions of the kernel buffer. For most functions, however, you can use any pixel of the kernel as the center pixel, by setting the IM_KER_CENTER_X and IM_KER_CENTER_Y fields.



Neighborhoods and their default center pixel

2. Load the kernel values into this kernel buffer, using *imBufPut()* or any processing function that might be appropriate.

An example

The following code performs a convolution with a user-defined kernel. Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```
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);

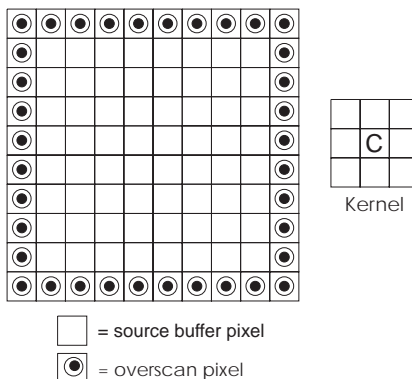
/* Set kernel values */
imBufPut(Thread, KerBuf, KerVals);

/* 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);
```

Specifying the overscan pixels

In a neighborhood operation, the neighborhood of some pixels will fall outside of the source buffer. To determine the new values for these pixels, "extra" source pixels are required. These are known as the overscan pixels of the neighborhood operation.



With the Genesis Native Library, you can use either "transparent overscan" or "replace overscan" to perform a neighborhood operation.

Transparent overscan

If you choose transparent overscan, the pixels of the source buffer's parent buffer are used as the overscan pixels. If the parent buffer can't provide overscan pixels (for example, if the source buffer is not a child buffer or if it touches one of the edges of its parent buffer), then the pixel values used will be undefined (leading to unpredictable results).

Replace overscan

If you choose replace overscan, the overscan pixels are set to a specified constant value.

Transparent vs. replace

In general, use transparent overscan when the source buffer is a child buffer. This will ensure that the overscan pixels are related to the pixels that border the source buffer. Use replace overscan if the source buffer is not a child buffer, if the source buffer touches one of the edges of its parent buffer, or if the source buffer is unrelated to its parent buffer.

Connectivity mapping

The *imIntConnectMap()* function calculates a connectivity code for each pixel in a binary source image and then maps these codes through a LUT buffer.

The connectivity code is obtained by linking the elements of a pixel's 3x3 neighborhood into a string, forming a single 9-bit number. Neighborhood pixels are linked in the following order:

$$\begin{array}{c} n_3 \ n_2 \ n_1 \\ n_4 \ n_8 \ n_0 \\ n_5 \ n_6 \ n_7 \end{array} \quad \text{where } n_i \text{ is either 0 or 1}$$

The pixels are connected and mapped as follows:

$$\text{Connectivity code} = \sum_{i=0}^8 2^i n_i$$

Result = LUTMAP (connectivity code)

Program the LUT with values that produce the desired result for each possible neighborhood configuration. Since each connectivity code has 9 bits, you should supply a LUT buffer with at least $2^9 = 512$ entries.

Support for 32-bit
floating-point

In-place not
supported

Geometric processing

The Genesis Native Library provides basic geometric functions (*imIntFlip()*, *imIntScale()*, *imIntSubsample()*, and *imIntZoom()*), as well as general, more flexible geometric functions (*imIntWarp...()*). The more general functions are discussed in Chapter 4. The basic geometric functions are faster than the general warping functions but cannot produce complex geometrical transforms.

Although the geometric functions are intended for integer images, most can operate on 32-bit floating-point data if no interpolation is specified.

Note that in-place operation is not supported for the geometric functions.

Flip/rotate

The *imIntFlip()* function allows you to:

- Flip horizontally (left to right) or vertically (top to bottom). Note that flipping horizontally allows you to get a mirror copy of your original image.
- Rotate 90, 180, or 270° counter-clockwise.

Scale by integer factors

The *imIntSubsample()* function minifies (subsamples) an image by an integer factor. When no interpolation is specified, the function takes a single sample from each block of the source buffer; when interpolation is specified, the function takes the average value of the block. The size of the block determines the factor by which the source buffer is minified.

The *imIntZoom()* function magnifies (zooms) an image by an integer factor. It replicates each pixel of the source buffer into a block of the same value. An averaging filter can then be applied to the result, thereby producing an interpolated zoom. The size of the block determines the factor by which the source buffer is magnified.

You can also use the *imIntScale()* function to scale an image by integer factors in non-interpolated mode and in interpolated mode. *imIntScale()* has more restrictions than *imIntSubsample()* or *imIntZoom()*; however, it is the fastest way to perform an interpolated scaling by an integer factor (especially for factors of 2, 4, and 8).

Scale by non-integer factors

The *imIntScale()* function can also scale an image by a non-integer factor in interpolated or non-interpolated mode. In non-interpolated mode, the exact scale factor that is specified is used. In interpolated mode, the actual scale factor used might be slightly different from the one requested. This is done in the interest of speed. However, non-integer factors that can be expressed as the ratio of two small integers (such as $1.5 = 3/2$) will also be used exactly as specified. In interpolated mode, the scale factors are used exactly as specified when:

scaling factor = n/m ,

where n and m are integers between 1 and 16 (inclusive).

You can specify the x and y scaling factors yourself, or you can have them automatically chosen such that the re-scaled image will just fill the destination buffer.

Color processing

Many processing functions can operate on multi-band (color) buffers. Specifically, if the source and destination buffers have the same number of bands, the function processes corresponding source bands and writes results to the corresponding destination band. If one of the source buffers has only one band while the other(s) have several bands, the function uses that one source band as many times as is needed.

When a function does not support multiple bands, you can create a child buffer for each band, using *imBufChildBand()*, and process the bands individually.

Choosing a color space

When processing color images, you can take advantage of the extra information available in the color image without greatly increasing the processing time required and without occupying more memory than is necessary. The key is to select the right color space.

The color spaces supported by the Genesis Native Library are RGB and HSL.

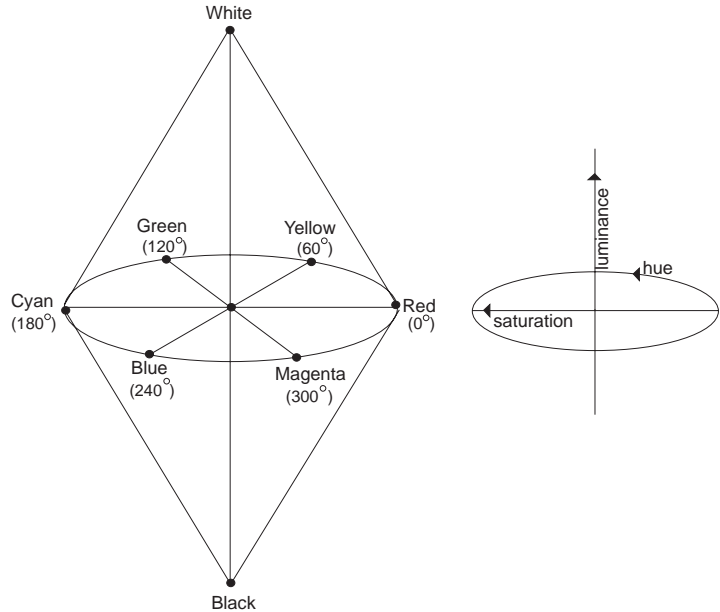
RGB color space

In the RGB color space, color is represented as a combination of red, green, and blue. This color space is generally used for most display hardware since it best matches the three colored phosphors of display monitors. It is also the direct output color space of many cameras and input devices. The color space does have drawbacks, however:

- Processing in this color space is not intuitive. Small changes in one component can have large visible effects.
- In general, the color component values are highly correlated. This redundancy can lead to wasted computation.

HSL color space

In the HSL color space, color is represented as a combination of hue, saturation, and luminance. The hue corresponds to the wavelength of the main color. It is represented as an angular position on a circular color disk. The luminance corresponds to the brightness of the color, while the saturation can be thought of as the measure of color purity or concentration.



The HSL color space is similar to the human way of describing colors. Each color has its own hue value (such as red, orange, or green). Once the hue value is chosen, changes to the saturation or the luminance alter only the color quality, not the basic color.

*Converting between
RGB and HSL*

You can convert between the RGB and HSL color spaces, using `imIntConvertColor()`. For efficiency, when converting from a 3-band (RGB) buffer, you can calculate just the hue (H) component of the HSL color space into a 1-band buffer. You can also use this function to extract the luminance (intensity) from an RGB image, or to copy the luminance component of an image into a three-band buffer, to create a monochromatic (gray) RGB buffer.

In addition, you can perform a custom matrix-defined conversion. To perform an arbitrary matrix-defined color conversion, you must also pass a 3x3 or 3x1 floating point coefficient buffer. For a matrix-defined conversion, the coefficient buffer should be 3x3 if both source and destination are 3-band buffers. 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.

You can control whether clipping is enabled, as well as the data type of input (source) and output (destination) bands, by adding control fields to the coefficient buffer. When IM_CTL_CLIP is set to IM_ENABLED, underflows and overflows are clipped to the output range [0, 255] for unsigned outputs, and [-128, 127] for signed outputs.

It is also possible to set IM_CTL_INPUT_FORMAT and IM_CTL_OUTPUT_FORMAT to IM_UNSIGNED or IM_SIGNED to control the data type of the input and/or output bands, respectively. Note that the first band is always unsigned.

Statistical processing

In addition to the processing operations, the Genesis Native Library provides a variety of statistical operations. For example, you can:

- Take the histogram of an image, using *imIntHistogram()*.
- Locate pixels that satisfy a specified condition, using *imIntLocateEvent()*.
- Count the differences between two images, using *imIntCountDifference()*.
- Find the minimum and/or maximum pixel value in an image, using *imIntFindExtreme()*.

The statistical functions write results either to a field of a buffer (for example, *imIntCountDifference()*, where only one value is returned), or into a buffer itself (for example, *imIntHistogram()*, where multiple values are returned).

Once you have completed your statistical operations and want to read back the results, use *imBufGet()* or *imBufGet1d()* if the results are written into a buffer; use *imBufGetField()* if the result is written into a field of a buffer.

❖ Statistical functions do not support packed binary buffers.

Histograms

When you perform a histogram, speed is dependent on data type. To guarantee maximum speed, the result buffer should be no larger than necessary. For example, 10-bit data must be stored as a 16-bit image, but only requires a result buffer of size $2^{10} = 1024$.

In order to speed up the time required to generate the histogram, you can specify certain options (described below). These options are mainly useful for deep input data. In such a case, the result buffer is normally too large to fit in on-chip RAM and the operation is much slower than normal.

Skip input pixels

You can specify that the histogram be generated using only every x^{th} column and/or every y^{th} row of the source buffer. To do so, use the IM_CTL_SUBSAMP_X and/or IM_CTL_SUBSAMP_Y fields. The histogram will have approximately the same shape but the total number of counts will be less.

Scale data

You can scale the input data, if the number of input bits exceeds the size of the result buffer. For example, if 16-bit data is used with a result buffer of size $2^{10} = 1024$, all input pixels can be right shifted by 6 bits. The operation will be much faster than with a full result buffer (size $2^{16} = 65536$), although there will be some approximations to the shape of the histogram since fewer bins are used.

In order to scale input data when the number of input bits exceeds the size of the result buffer, use the IM_CTL_INPUT_BITS field to indicate the input data size.

Note that, by default, the number of input bits is deduced from the size of the result buffer. For example, when the source buffer is 16-bit and the result buffer is of size 2^{10} , the input data is assumed to be only 10-bit and is therefore not right-shifted. However, if the IM_CTL_INPUT_BITS field is set to 16, input pixels are right shifted by 6 bits.

❖ The IM_CTL_INPUT_BITS field only applies when the source buffer is 16-bit, since no right shift is needed for 8-bit buffers.

Chapter 4: Advanced processing

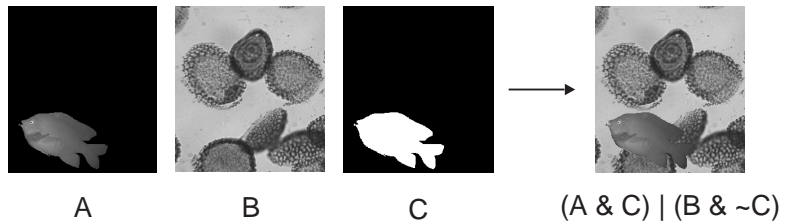
This chapter describes some of the more advanced processing functions of the Matrox Genesis Native Library.

Three-input arithmetic and logical operations

With the Genesis Native Library, you can perform arithmetic and logical operations on up to three input operands, using *imBinTriadic()* or *imIntTriadic()*. Working with three operands rather than two can reduce the number of function calls required to perform an operation. For example, merging operands A and B based on operand C,

$(A \& C) \mid (B \& \sim C)$, can be performed in a single call to *imIntTriadic()* rather than three calls to *imIntDyadic()*.

Note that the operation $(A \& C) \mid (B \& \sim C)$ passes A where C is 0xFFFFFFFF, and passes B where C is 0.



More on the
Triadic functions

Besides providing pre-defined operators, *imBinTriadic()* and *imIntTriadic()* allow you to control the type of arithmetic or logical operation performed by allowing you to specify the actual PP ALU opcode. By specifying opcodes directly, you can perform any type of arithmetic or logical operation that the PP ALU supports. Note, however, that you only need to derive opcodes if you cannot perform the required operation using a pre-defined opcode or using another arithmetic and logical function.

Deriving opcodes for
Triadic functions

The following example outlines how to derive the opcode for a logical operation. Most arithmetic operations exist as pre-defined opcodes; if you ever need to derive the opcode for an arithmetic operation, you can contact Texas Instruments for their *TMS320C80 (MVP) Parallel Processor's User Guide*. (You will need to derive the 32-bit part of the PP opcode that resides in register d0 for the instruction class EALU || ROTATE).

Deriving the proper opcode for a logical operation

Operation: $(A \& C) \mid (B \& \sim C)$ i.e. pass A when C is 1; pass B when C is 0

A	B	C	$(A \& C) \mid (B \& \sim C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

First step:
Derive the result of
the operation for all
combinations of
A, B, and C

	CB = 00	CB = 01	CB = 11	CB = 10
A = 0	0 F0	1 F2	0 F6	0 F4
A = 1	0 F1	1 F3	1 F7	1 F5

Second step:
Represent results
in this table

F7 F6 F5 F4 F3 F2 F1 F0
1 0 1 0 1 1 0 0

10101100 << 19

Last step:
Left-shift by 19 bits

The opcode for $(A \& C) \mid (B \& \sim C)$ is therefore $10101100 \ll 19 = 0x5600000$. The appropriate function calls for *imBinTriadic()* and *imIntTriadic()* would be:

```
imBinTriadic(thr, bufa, bufb, bufc, destbuf, 0x5600000, 0);
```

```
imIntTriadic(thr, bufa, bufb, bufc, destbuf, 0, 0x5600000, IM_DEFAULT, 0);
```

Note, however, that $(A \& C) \mid (B \& \sim C)$ exists as a pre-defined opcode: `IM_PP_MERGE`.

❖ The distinction between arithmetic and logical operations lies in whether the operation produces any carry-outs (for example, $A + B$ produces a carry-out for $1 + 1$). If there are no carry-outs (for any combination of A, B, and C) the operation is logical and its opcode can be derived according to the above steps. If there are carry-outs, the operation is arithmetic and its opcode must be derived using a different method; see the *TMS320C80 (MVP) Parallel Processor's User Guide*.

Live processing

Grabbing a sequence of frames in real-time

To grab a sequence of frames in real-time, simply use successive calls to *imDigGrab()* in the same thread:

```
#define NUMFRAMES 8
long Buf[NUMFRAMES];

/* Allocate a buffer for each frame */
for (i = 0; i < NUMFRAMES; i++)
    imBufAlloc2d(Thread, SizeX, SizeY, IM_UBYTE, IM_PROC, &Buf[i]);

/* Grab the sequence */
for (i = 0; i < NUMFRAMES; i++)
    imDigGrab(Thread, 0, Camera, Buf[i], 1, 0, 0);
```

No frames will be missed as long as you use compatible camera definitions for each call and as long as no other applications are also grabbing to the same memory bank. Frames could be missed if the camera definitions are different, since the digitizer has to be re-programmed for each grab. Frames could also be missed if two or more applications grab to the same memory bank, since each memory bank has just one VIA capable of writing grabbed data to memory.

Real-time processing

To grab and process concurrently on Genesis, you need to allocate two grab input buffers. You then process one buffer while grabbing the next frame into the other buffer. You must switch the destination of the grab between the two input buffers (this is commonly known as double buffering). You also need to synchronize the grabbing and processing so that:

- You do not process a buffer until an entire frame has been grabbed into the buffer.
- You do not grab into a buffer until the previous frame in that buffer has been processed.

There are many ways to implement real-time processing. For example, you could use separate threads for grabbing and processing, and possibly a third thread for synchronization. However, one of the simplest ways is to use a single thread and to use *imDigGrab()* in asynchronous mode.

Asynchronous grab mode

When you use *imDigGrab()* in asynchronous mode, the thread to which *imDigGrab()* is sent will not wait for the grab to complete before continuing to execute. You specify asynchronous mode through the `IM_CTL_GRAB_MODE` field of *imDigGrab()*. Note that, by default, the thread will wait for *imDigGrab()* to complete before continuing.

Using *imDigGrab()* in asynchronous mode allows you to grab and process concurrently using just one thread. This can simplify the application code, as shown below.

```
/* Select asynchronous grab mode */
imBufPutField(Thread, ControlBuf, IM_CTL_GRAB_MODE, IM_ASYNCHRONOUS);

/* Grab the first frame */
imDigGrab(Thread, 0, Camera, InBuf[0], 1, ControlBuf, GrabOSB[0]);
i = 1;

while (!kbhit())
{
    /* Queue the next grab into the other buffer */
    imDigGrab(Thread, 0, Camera, InBuf[i], 1, ControlBuf, GrabOSB[i]);

    /* Switch buffers */
    i = 1 - i;

    /* Process each frame when the grab completes */
    imSynchHost(Thread, GrabOSB[i], IM_COMPLETED);
    imIntLutMap(Thread, InBuf[i], OutBuf, LutBuf, 0);
}
```

Note how the above code satisfies the two synchronization requirements. The first requirement (that processing wait for grabbing) is satisfied by using *imSynchHost()*. The second requirement (that grabbing wait for processing) is automatically satisfied because the grab function is called after the processing function in the same thread (recall that functions in a thread execute serially).

*A note about
command queues*

While the above code processes the same number of frames per second whether *imSyncHost()* or *imSyncThread()* is used, *imSyncHost()* prevents the command queue on the board from filling up. If *imSyncThread()* is used, there is nothing to stop the Host from executing the loop much faster than the board can carry out the processing. Therefore, the command queue on the board will quickly fill up, preventing other applications on the board from running properly. In addition, grabbing will continue even after you exit the loop (for several seconds) because of all the queued commands.

Note also that each command queued to the board uses a small amount of 'C80 time. Therefore, if you queue too many commands, you will significantly slow down any functions that are currently executing. Typically, it is safe to queue a few tens of commands, but hundreds would be too many.

To inquire about the status of a command queue, use *imDevInquire()*.

Geometric warpings

In addition to the basic geometric functions described in Chapter 3, the Genesis Native Library contains geometric functions (*imIntWarp...()*) that allow you to warp an image. Warpings could be used, for example, to correct geometric distortions in an image.

How a warping is performed

The Genesis Native Library performs warpings by associating each pixel position of the destination buffer, (x_d, y_d) , with a specific point in the source buffer, (x_s, y_s) , and then determining the pixel value at (x_d, y_d) from its associated point and from a specified interpolation mode.

First-order polynomial warpings

You can perform first-order polynomial warpings using *imIntWarpPolynomial()*. A first-order polynomial warping is equivalent to linearly translating, rotating, resizing, and/or shearing an image. First-order polynomial warpings are performed by associating points in the source buffer with pixels in the destination buffer according to the following equations:

$$x_s = a_0 + a_1x_d + a_2y_d$$

$$y_s = b_0 + b_1x_d + b_2y_d$$

Generating coefficients

The coefficients ($a_0...a_2, b_0...b_2$) required to produce a polynomial warping can be automatically generated (using *imGenWarp1stOrder()*) or user-supplied. When using *imGenWarp1stOrder()*, you specify how you want the warping performed (for example, by how much you want to rotate and resize an image); the function then generates the coefficients required to produce such a warping (see the following example).

Whether you supply the coefficients yourself or have them generated, they must be placed in a 32-bit floating-point buffer of size 3×2 .

An example

The following code rotates an image by 30° about its center. Since rotations are performed about (0, 0), the image is translated to move its center to (0, 0), rotated, and then translated to its original position. Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```
long CoefBuf; /* Coefficient buffer (also control buffer) */

/* Allocate the warp coefficient buffer */
imBufAlloc2d(Thread, 3, 2, IM_FLOAT, IM_PROC, &CoefBuf);

/* Generate coefficients for a 30° rotation about the center */
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);

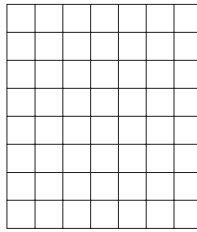
/* Rotate the image */
imIntWarpPolynomial(Thread, SrcBuf, DstBuf, CoefBuf, CoefBuf, 0);
```

Using a LUT to perform a warping

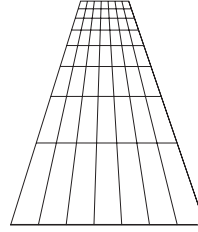
You can perform warpings through look-up tables (LUTs), using *imIntWarpLut()*. Since this function associates points in the source buffer with pixels in the destination buffer through LUTs, it can perform any type of warping. For example, it can perform first-order polynomial warpings and perspective warpings, as well as warpings that arbitrarily map pixels in the destination buffer to points in the source buffer.

Perspective warpings

A perspective warping maps an arbitrary quadrilateral onto a rectangle, in such a way that part of the image seems farther from your plane of view.



Source image



A perspective transformation
of the source image

Perspective warpings are performed by associating points in the source buffer with pixels in the destination buffer according to the following equations:

$$x_s = \frac{x}{w} \quad \text{and} \quad y_s = \frac{y}{w}$$

where

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} c_{00} & c_{10} & c_{20} \\ c_{01} & c_{11} & c_{21} \\ c_{02} & c_{12} & c_{22} \end{bmatrix} \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix}$$

so

$$x_s = \frac{c_{00}x_d + c_{10}y_d + c_{20}}{c_{02}x_d + c_{12}y_d + c_{22}}$$

$$y_s = \frac{c_{01}x_d + c_{11}y_d + c_{21}}{c_{02}x_d + c_{12}y_d + c_{22}}$$

To perform a perspective warping, you must supply the 3x3 coefficients ($c_{00} \dots c_{22}$) to the *imGenWarpLutMatrix()* function. This function generates the LUTs required by *imIntWarpLut()* to perform the warping. The 3x3 coefficients can be user-supplied or automatically generated using *imGenWarp4Corner()*.

Note that, if c_{02} and c_{12} are set to 0 in the equations for a perspective warping, the equations reduce to a first-order polynomial warping. You could therefore perform a first-order polynomial warping by generating the required coefficients using *imGenWarp1stOrder()*, passing the generated coefficients to *imGenWarpLutMatrix()*, and then passing the generated LUTs to *imIntWarpLut()*. However, it is more efficient to use *imIntWarpPolynomial()*.

An example

The following code performs a perspective warping on an image. Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```
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);
```

Another example

The following code warps an image through user-supplied LUTs. Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```

long X LutBuf;      /* X address LUT buffer */
long Y LutBuf;      /* 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);

/*
 * Calculate the (X,Y) source address for each destination pixel.
 * Use integer address values for nearest neighbor 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[Ox + Oy*SizeX] = (short) Wx;
        YLutVals[Ox + Oy*SizeX] = (short) Wy;
    }
}

```

```

/* Load the address LUT buffers */
imBufPut(Thread, XLutBuf, XLutVals);
imBufPut(Thread, YLutBuf, YLutVals);

/* Warp the image (using the default nearest neighbor mode) */
imIntWarpLut(Thread, SrcBuf, DstBuf, XLutBuf, YLutBuf, 0, 0);

```

Interpolation modes

When you perform a warping, pixel positions in the destination buffer, (x_d, y_d) , get associated with specific points in the source buffer, (x_s, y_s) . The destination coordinates have integer values but the source coordinates, in general, do not. Therefore, the pixel value at (x_d, y_d) has to be determined from several source pixels that are near (x_s, y_s) , according to a specified interpolation mode.

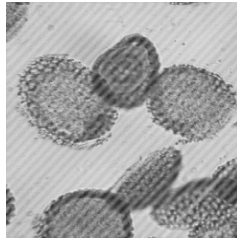
The following are some interpolation modes:

- **Nearest-neighbor.** This mode determines the nearest value to a point, and copies that value into its associated position.
- **Bilinear.** This mode takes a weighted average of the four pixels nearest to the point, and copies that average into its associated position. The pixels closest to the point are given the most weight.
- **Bicubic.** This mode takes a weighted average of the sixteen pixels nearest to the point, and copies that average into its associated position. Again, the pixels closest to the point are given the most weight.

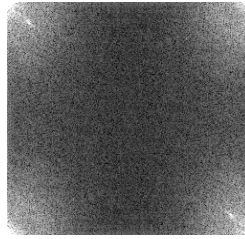
In general, nearest-neighbor interpolation is the fastest to perform, and bicubic interpolation is the slowest. However, nearest-neighbor interpolation produces the least accurate results, and bicubic interpolation produces the most accurate. Bilinear interpolation is often the best compromise between speed and accuracy.

Fourier transforms

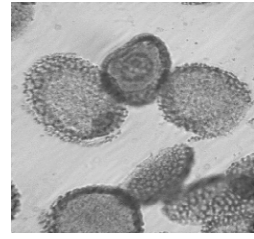
With the Genesis Native Library, you can represent an image in its frequency domain by performing a fast Fourier transform (FFT) on the image, using *imIntFFT()*. In the frequency domain, you can easily locate any constant spatial patterns in an image (which can be caused, for example, by systematic noise). By changing an image's frequency domain representation and then performing an inverse transform (also using *imIntFFT()*), you can emphasize or de-emphasize constant spatial patterns.



An image with constant systematic noise. The spatial frequency of this pattern is quite different from the image's other spatial frequencies. Therefore, in the frequency domain of this image, it should be clearly distinguishable.



The FFT of the image. The bright spots (near the corners) represent the spatial frequencies of the systematic noise.



By removing these spots and then performing an inverse transform, the noise is removed.

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 (this will right-shift results at each stage of the transform so that the dynamic range doesn't get larger); see the following example.

An example

The following code performs an FFT on an image, then performs a reverse transform. Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```

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 the FFT (size must be 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 zero */

/* Add 12 fractional bits for extra precision */
imIntMonadic(Thread, SrcBuf, 12, IntrBuf, IM_SHIFT, 0);

/* Set control fields for a 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);

...

/* 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);

```

Chapter 5: Blob analysis

This chapter describes how to perform blob analysis.

Blob analysis

Blob analysis is a branch of image analysis that allows you to identify connected regions of pixels (blobs) within an image, and then to calculate selected features of those regions. Typical features describe the size, shape, and location of the blobs. Therefore, you can use the results of blob analysis for a variety of purposes, for example, to distinguish between objects in an image, to locate objects, and to measure objects. Along a production line, for instance, you can use results to determine if parts have been manufactured within specified tolerances.

Segmentation

Before you can perform blob analysis, you must segment your image. This is the process of separating blob pixels from the rest of the image. Since the blob module requires an image in which either the blobs or the background have the value zero, the minimum that you must do is threshold the image.

The segmented image is known as the *blob identifier image*. If the background of the blob identifier image has the value zero, touching pixels with non-zero values are considered a blob. If the background has non-zero values, touching pixels with zero values are considered a blob.

Binary vs. grayscale features

Note that most features only depend on the shape of a blob. These are known as binary features. Since the blob module only needs to identify blob pixels from background pixels to calculate binary features, the only image you need to provide to the blob module is the blob identifier image. However, there are certain features (such as the mean pixel value of a blob) which depend on the value of pixels within the blob. These are known as grayscale features. If you plan to calculate grayscale features, you need to provide two images to the blob module: the blob identifier image and the original grayscale image.

General steps

The general steps to perform blob analysis are:

1. Segment your image to produce the blob identifier image. The minimum that you must do is threshold the image so that either the blobs or the background have the value zero.
2. Allocate a blob analysis result buffer, using *imBlobAllocResult()*. A blob result buffer is used to store the results of blob analysis.
3. Adjust blob analysis controls, if the defaults are not suitable, using *imBlobControl()*. If, for example, the blobs have the value zero, you must use *imBlobControl()* to specify this (by default, the background is considered to have the value zero).
4. Allocate a feature list, using *imBlobAllocFeatureList()*. A feature list specifies the features to calculate.
5. Select features for calculation by adding them to the feature list, using *imBlobSelectFeature()*, *imBlobSelectFeret()*, and/or *imBlobSelectMoment()*.
6. Calculate the features, using *imBlobCalculate()*. Note that, if you simply want the number of blobs, you must still perform this step.
7. If necessary, exclude blobs whose results don't meet specified criteria, using *imBlobSelect()*.
8. If necessary, repeat steps 5, 6, and 7, until you have all the results you need.
9. Transfer necessary results to the Host or copy necessary results to an on-board buffer.
10. Free the result buffer and feature list, using *imBlobFree()*.

An example

The following code determines the bounding box of each blob in an image. Note that this code is part of the *blob.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *blob.c* program.

```
long FeatList;          /* Blob feature list */
long Result;            /* Blob result buffer */
long Number;            /* Number of blobs */

/* 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);

/* Calculate selected features */
imBlobCalculate(Thread, IdentBuf, 0, FeatList, Result, IM_CLEAR, 0);

/* Get the number of blobs */
imBlobGetNumber(Thread, Result, &Number);
```

Segmentation

Segmentation is the process of separating blob pixels from background pixels. Since the blob module requires an image in which either the blobs or the background have the value zero, the minimum that you must do is threshold the image (using *imIntBinarize()* or *imBinConvert()*). Typically, this is all you need to do. However, if the gray-levels of the blobs are not different from the gray-levels of the background, you will have to use a more complicated segmentation algorithm. In addition, you will have to process the image if, for example, two blobs are touching (since they will be considered one blob) or if noise has introduced some spurious blobs.

Usually, spurious blobs consist of just a few pixels. You can generally remove these blobs through an erosion followed by a dilation (if your blobs have non-zero values) or through a dilation followed by an erosion (if your blobs have zero values). If you have touching blobs, you can try eroding the image (if your blobs have non-zero values) or dilating the image (if your blobs have zero values), until these blobs are separated. Note that, if your image is packed binary, you perform erosions and dilations using *imBinMorphic()*; otherwise, you use *imIntErodeDilate()*.

If your image has a lot of noise, you might want to filter it before segmenting. For example, you might want to smooth the image (using *imIntConvolve()*) or apply a median filter to it (using *imIntRank()*).

Reducing processing

To reduce processing, your image should be acquired under the best possible conditions. This means that blobs should not overlap and, if possible, not touch. In addition, the background should have a very different gray level from the blobs.

If the above conditions are followed, the acquired image can usually be segmented with a simple threshold.

Processing vs.
excluding

If you need to process your image, it can affect blob calculations because the shape of the remaining blobs might be changed slightly. Instead of processing unwanted blobs out of your image, however, you can first perform calculations on all blobs in the image, and then use *imBlobSelect()* to exclude blobs based on the results. For the details, see the *Selecting blobs* section.

If you have only a few unwanted blobs, it is generally faster to perform calculations on all blobs and then exclude the unwanted ones using *imBlobSelect()*. However, if you have many unwanted blobs, blob analysis will likely be faster if you first process the unwanted blobs out of your image.

Note that which method is more efficient (processing or excluding) ultimately depends on the content of your image.

Adjusting controls

You can control the following aspects of a blob analysis operation using *imBlobControl()*:

- Which pixel values (zero or non-zero) are considered blob pixels (IM_BLOB_FOREGROUND_VALUE). By default, non-zero values are considered blob pixels.
- Whether diagonally adjacent pixels are considered touching (IM_BLOB_LATTICE). Recall that a blob is an area of touching pixels that have the same value. Horizontally and vertically adjacent pixels are considered touching. By default, diagonally adjacent pixels are also considered touching.
- The pixel aspect ratio (IM_BLOB_PIXEL_ASPECT_RATIO). This setting allows you to compensate for pixels that are not square (see the *Pixel aspect ratio* section).
- The number of Feret angles (IM_BLOB_NUMBER_OF_FERETS). This setting is used when a feature requires several Feret diameters to be calculated. Feret diameters are discussed later in the chapter.
- Whether to group results (IM_BLOB_IDENTIFICATION). By default, separate results are produced for each blob. However, you can group the results of certain blobs, or group the results of all blobs (see the *Grouping results* section).
- Whether to save, in the result buffer, a run-length encoded version of the blob identifier image (IM_BLOB_SAVE_RUNS). If you disable the saving of runs, you will reduce the memory required for the result buffer and might also increase the speed of *imBlobCalculate()* slightly. However, you will not be able to use functions which rely on run information (*imBlobCopyRuns()*, *imBlobFill()*, *imBlobGetLabel()*, *imBlobGetRuns()*, *imBlobLabel()*). Note that runs are defined in the *Transferring or copying runs* section.
- The maximum time allowed for *imBlobCalculate()* to process (IM_BLOB_MAX_TIME). By default, *imBlobCalculate()* will run to completion, no matter how long this takes. However, you can specify a maximum processing time (see the *Timeout period* section).

- The amount of processing time that *imBlobCalculate()* uses before it yields to other threads of equal priority (IM_BLOB_TIME_SLICE). By default, *imBlobCalculate()* uses all of the master processor's (MP's) time until it finishes. This means that other threads of equal priority will not be able to execute any new commands until *imBlobCalculate()* has finished (although threads of higher priority will be able to execute commands). If your application requires that *imBlobCalculate()* run in parallel with other commands, you can specify that it use only a certain amount of the MP's time before yielding to other threads of the same priority. For example, if you specify a time slice of 1 ms, then *imBlobCalculate()* will yield to other threads of the same priority every 1 ms. Note that *imBlobCalculate()* will still run to completion, but not as quickly (the performance degradation will depend on how many other threads have commands to execute).

Pixel aspect ratio

In general, an object has the same proportions in real-life as it has in an image. This means that, while blob calculations are performed in pixels, it is relatively easy to interpret results in real-world units. However, if pixels in your image are not square, objects in the image will be distorted, and blob calculations will lead to incorrect interpretations.

To determine whether pixels in your image are square, you can measure the image's *pixel aspect ratio*. The pixel aspect ratio of an image compares the real-world size of a rectangular region in the image, with its size in pixels. Specifically, it is:

$$\frac{\text{\# of pixels in the region's Y direction} / \text{\# of pixels in the region's X direction}}{\text{\# of real-world units in the region's Y direction} / \text{\# of real-world units in the region's X direction}}$$

Note that you can measure the pixel aspect ratio by grabbing an image of a circle or square and then, using the blob analysis module, calculating the Feret diameter of the circle or square at 0° and 90° (IM_BLOB_FERET_X and IM_BLOB_FERET_Y). Since the number of real-world units in the x and y direction of a circle or square are the same, and since the number of pixels in the x and y direction correspond to IM_BLOB_FERET_X and IM_BLOB_FERET_Y, the pixel aspect ratio of the image is IM_BLOB_FERET_Y/IM_BLOB_FERET_X.

If the pixel aspect ratio is 1.0, pixels in the image are square.
 If the pixel aspect ratio is not 1.0, pixels are not square.

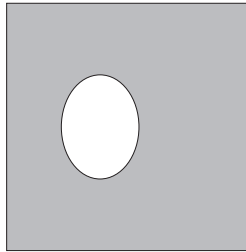


Image of a circle with
a 1.33 aspect ratio.

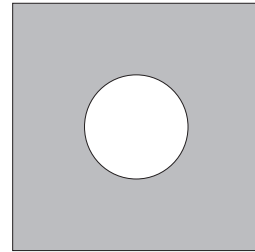


Image of a circle with
a 1.0 aspect ratio.

Non-square pixels

When the pixel aspect ratio is not 1.0, you should first check which camera definition file you are using to grab images. Note, for example, that an RS-170 signal produces non-square pixels when digitized to 512x480 pixels but produces square pixels when digitized to 640x480 pixels.

If changing camera definition files is not an option, you can either:

- Warp the image so that the pixel aspect ratio becomes 1.0. The warping functions are discussed in Chapter 4.
- Specify what the actual pixel aspect ratio is, using *imBlobControl()*. In this case, the pixel aspect ratio is factored in when blob calculations are performed.

In general, it is faster simply to specify the pixel aspect ratio. However, specifying the pixel aspect ratio will only produce correct results if non-square pixels are due to a simple stretching of the image (in the x or y directions). If pixels are not square due to a more complex distortion (such as a perspective distortion), you must warp the image.

Grouping results

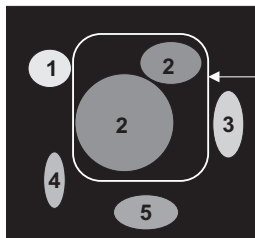
By default, the Genesis Native Library produces separate results for each blob. There might be times, however, when you want to group results. For example, you might want to know the combined area of all blobs in your image, or you might have a situation where a blob is incorrectly considered two or more blobs because it is separated by noise.

Grouping all blobs

To group the results of all blobs, set `IM_BLOB_IDENTIFICATION` to `IM_WHOLE_IMAGE`. In this mode, features are calculated as if all blob pixels are part of the same blob.

Grouping some blobs

To group the results of certain blobs, set `IM_BLOB_IDENTIFICATION` to `IM_LABELLED`. In this mode, results are grouped for those blobs in the blob identifier image that have the same pixel value.



These two blobs are treated as one. All others are treated individually.

Assigning particular label values

By default, *imBlobCalculate()* assigns an arbitrary label value to each blob. Note, however, that you can assign a particular label value by filling each blob with a unique pixel value, using *imGraFill()*, and then setting `IM_BLOB_IDENTIFICATION` to `IM_LABELLED`.

Assigning a particular label value can sometimes make it easier to associate blob results with specific objects in your image.

Timeout period

By default, *imBlobCalculate()* will take as long as necessary to complete. However, you can specify a maximum processing time, using *imBlobControl()*. In this case, *imBlobCalculate()* will timeout when the specified period expires, rather than running to completion.

To determine if *imBlobCalculate()* timed out, call *imBlobInquire()*, setting its **Item** parameter to `IM_BLOB_TIMEOUT`. If *imBlobCalculate()* did time out, results will not be valid.

Setting a maximum processing time can be useful because the execution time of *imBlobCalculate()* can vary widely with the number of blobs in the image. For example, in an inspection application, the number of blobs (defects) is usually very small, so execution is fast. However, if an image contains many defects or was perhaps badly thresholded, processing time can be much longer than normal, possibly causing frames to be missed. You might prefer to set a maximum processing time that would reject such images. For example, assuming that processing normally takes 5 ms for a good image, you might want to abort processing after 15 ms and continue to the next image:

```
/* Set maximum processing time allowed */
imBlobControl(Thread, Result, IM_BLOB_MAX_TIME, 0.015);

/* Perform blob analysis */
imBlobCalculate(Thread, SrcBuf, 0, FeatList, Result, IM_CLEAR, 0);

/* Check whether processing timed out or not */
if (imBlobInquire(Thread, Result, IM_BLOB_TIMEOUT, NULL));
{
    /* Assume that the object is defective */
    ...
}
else
{
    /* Get results as normal */
    imBlobGetNumber(Thread, Result, &Number);
    ...
}
```

Features

This section discusses some of the blob features supported by the Genesis Native Library. For a complete list of features, see the *Genesis Native Library Command Reference*.

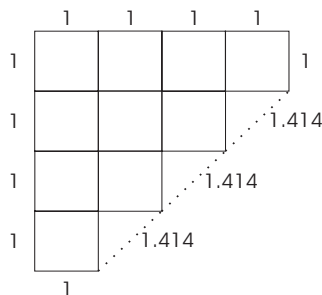
Most features fall into one of four main groups:

- Area and perimeter
- Dimensions
- Shape
- Location

Area and perimeter

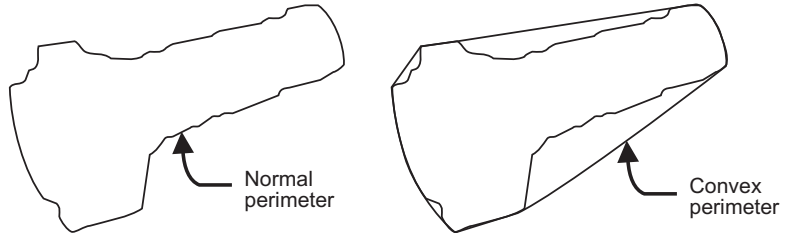
You can calculate the area and perimeter of a blob. Since blob calculations assume a pixel is 1 unit long and 1 unit wide, the area of a single pixel is 1 and the perimeter of a single pixel is 4. The area of a blob is then the number of pixels in that blob, while the perimeter of a blob is the number of pixel sides along that blob.

When calculating the perimeter, an allowance is made for the staircase effect (the "pixel side" of diagonally adjacent pixels is considered to be 1.414, rather than 2). For example, the following blob has a perimeter of 14.242.



Convex perimeter

In addition to the normal perimeter, you can calculate the convex perimeter of a blob. The convex perimeter of a blob is the perimeter of the blob's convex hull.



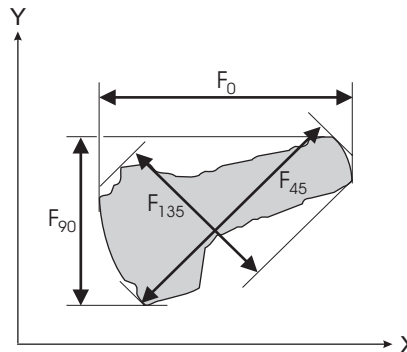
The convex perimeter is derived from several Feret diameters of the blob (Feret diameters are discussed in the next section). You can specify the number of Feret diameters, using *imBlobControl()*. The more Feret diameters used, the more accurate the convex perimeter but the longer the processing time.

Dimensions

The dimensions of a rectangular object are its length and width. Most blobs, however, are not rectangular. Therefore, to get an indication of a blob's dimensions, you need to look at other features (such as the Feret diameter).

Feret diameter

The Feret diameter of a blob is the diameter of the blob at a given angle. Several Feret diameters are shown below. The angle at which the Feret diameter is taken (relative to the horizontal axis) is specified as a subscript to the F.



With the Genesis Native Library, you can determine the minimum and maximum Feret diameter of a blob (`IM_BLOB_FERET_MIN_DIAMETER` and `IM_BLOB_FERET_MAX_DIAMETER`, respectively). The length of a blob can then be defined as its maximum Feret diameter and the width of a blob as its minimum Feret diameter.

Note that the minimum and maximum Feret diameter are determined after checking a specified number of Feret diameters. You specify the number using *imBlobControl()* (the default is 8 and is suitable for most blobs). In general, the more Feret diameters used, the more accurate the calculation, but the longer the processing time.

Different Feret diameters

In addition to the minimum and maximum Feret diameter, you can calculate the Feret diameter at 0° (`IM_BLOB_FERET_X`), at 90° (`IM_BLOB_FERET_Y`), or at a specified angle (using *imBlobSelectFeret()*).

Long, thin blobs

Note that the minimum and maximum Feret diameters of a long, thin blob are not very representative of its dimensions (especially if the blob is curved). For long, thin blobs, the following features are better:

- `IM_BLOB_LENGTH`.
- `IM_BLOB_BREADTH`.

The above features are derived from a blob's area and perimeter, assuming that $\text{area} = \text{length} \times \text{breadth}$ and $\text{perimeter} = 2(\text{length} + \text{breadth})$. These are not valid assumptions for most blob types, although they do hold for long, thin blobs (such blobs have approximately constant breadth along their length).

Shape

When trying to distinguish between blobs, the shape of the blobs can be an important feature. This is because two blobs can have similar sizes but different shapes due to a different number of holes, curves, or edges.

The following features will tell you something about a blob's shape:

- **Compactness.** This is a measure of how close pixels in the blob are to one another. It is derived from the blob's area and perimeter. A circular blob is most compact and is defined as having the minimum compactness value (1.0); more convoluted shapes will have higher values.
- **Roughness.** This is a measure of the unevenness or irregularity of a blob's surface. It is the ratio of the blob's perimeter to its convex perimeter. The minimum roughness value is 1.0 because a blob's perimeter will always be equal to or greater than its convex perimeter. A blob with many jagged edges will have a much higher roughness value because its perimeter will be much larger than its convex perimeter.

Number of holes

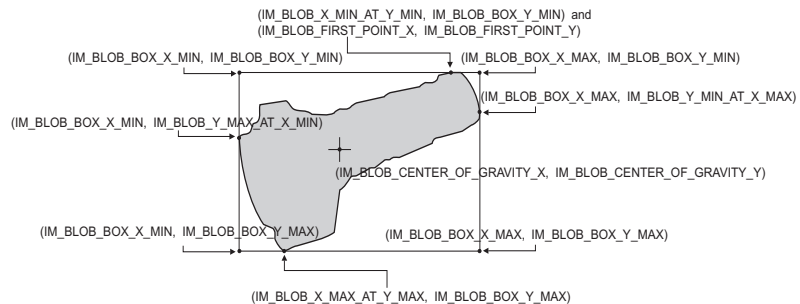
In addition to compactness and roughness, the number of holes in a blob can be useful in distinguishing between blobs. Note, however, that this feature can also be misleading because a hole can be the result of a single noise pixel in the wrong place.

Blob location

The location of a blob can sometimes be more useful than its shape or size. For example, if a robotic arm needs to pick up an object, it can use the location of that object in an image as a guide. You can also use the location of a blob to determine if it touches any image borders. If it does touch any image borders, you might want to adjust the camera's field-of-view or exclude the blob (since certain features, such as the area of the blob, would be misleading).

Blob points

With the Genesis Native Library, you can determine the following blob points:



The center of gravity can be calculated in binary or grayscale mode. The former is determined from the blob identifier image, and the latter from the original grayscale image.

Moments

With the Genesis Native Library, you can calculate the moments used to find the center of gravity, as well as other common moments. If you need to calculate a specific moment, use *imBlobSelectMoment()*.

You can calculate binary and grayscale moments (the former calculated from the blob identifier image and the latter from the original grayscale image). In addition, you can calculate central and ordinary moments. Central moments use coordinates that are relative to the center of gravity of the blob. Ordinary moments use coordinates that are relative to the top-left corner of the image, and are therefore dependent on the blob's position within the image.

Selecting blobs

Sometimes, you will not be interested in all blobs in your blob identifier image. For example, you might have blobs that are actually noise pixels, or you might have blobs that touch the image borders. Unwanted blobs can sometimes be removed by processing the image. However, processing might affect the shape of the relevant blobs in your image and, therefore, the results of blob calculations. In addition, such processing can be time-consuming.

With the Genesis Native Library, you can exclude or delete unwanted blobs using *imBlobSelect()*. Blobs are excluded or deleted based on the result of a specified feature. Therefore, you select the features to calculate, calculate the features, and then use *imBlobSelect()* to exclude or delete unwanted blobs. You then repeat this process until the features and blobs you have selected produce the results you need.

- ❖ If you have many unwanted blobs, you can save time and memory by first calculating only those features that allow you to distinguish between unwanted and relevant blobs. Once unwanted blobs are excluded or deleted, calculate all required features.

Excluding vs. deleting

Note that excluded blobs are simply ignored during subsequent calculations but can be re-included (using *imBlobSelect()*). Deleted blobs are removed completely from the blob result buffer and cannot be re-included. However, deleted blobs are **not** removed from the blob identifier image.

If necessary, you can remove blobs from a blob identifier image using *imBlobFill()*.

Transferring or copying results

Once blob calculations are performed, you can copy results to an on-board buffer using *imBlobCopyResult()*, or transfer results to the Host using *imBlobGetResult()*. These functions can retrieve results for a specific feature or for a predefined group of features. In the latter case, set the *Feature* parameter of *imBlobCopyResult()* or *imBlobGetResult()* to the desired group (for example, IM_BLOB_GROUP1); all results for features in that group will be returned at the same time. Note that retrieving results for a predefined group can reduce the number of function calls required to retrieve results (often, it can reduce it to one function call, since similar features are grouped together). Retrieving results for each feature individually can take a long time. In fact, when there are relatively few blobs and many features, the time to retrieve results can be a significant overhead when compared to the calculation time.

To see which features are in which groups, refer to the *Genesis Native Library Command Reference*. There is some overlap of features between groups. For example, the label value is included in all groups because you might need it no matter what other features you calculate. Also included in each group is the number of blobs. This means that you do not need to call *imBlobGetNumber()* if you are sure you have allocated enough memory for the results.

To save memory and reduce transfer time when retrieving results for a predefined group, features that can easily be derived from others are not included in any group. For example, IM_BLOB_FERET_X is not included, because it is equal to $\text{IM_BLOB_BOX_X_MAX} - \text{IM_BLOB_BOX_X_MIN} + 1$. Note, however, that these features can be retrieved individually or can be determined from their equations (see *imBlobSelectFeature()* in the *Genesis Native Library Command Reference* for the equations needed to derive features).

When you retrieve results for a predefined group, the results are stored in a specific data structure. There is a different data structure for each group; see the *Genesis Native Library Command Reference* for the definitions. 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.

Note that, when you retrieve results for a group of features, only results for features you calculated will be valid. Values for features not calculated will simply be undefined, and no error messages will be generated.

An example

The following code determines the bounding box of each blob in an image, then uses the results to mark each blob. Since the required features are all in the same group, results can be retrieved using a single call to *imBlobGetResult()*.

```
#define MAX_BLOBS 100

/* Array of structures to hold results */
IM_BLOB_GROUP1_ST Group1[MAX_BLOBS];

/* Select all required features */
imBlobSelectFeature(Thread, FeatList, IM_BLOB_BOX, IM_DEFAULT);

/* Calculate selected features */
imBlobCalculate(Thread, IdentBuf, 0, FeatList, Result, IM_CLEAR, 0);

/* Get results all at once */
imBlobGetResult(Thread, Result, IM_BLOB_GROUP1, IM_DEFAULT, Group1);

/* Mark the bounding boxes */
for (i = 0; i < Group1[0].number_of_blobs; 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);
```

Note that, in the above code, the maximum number of blobs is known, so *imBlobGetNumber()* is not needed to determine how much memory to allocate for results. However, if you are not sure what the maximum number of blobs will be, you should call *imBlobGetNumber()* first, then allocate the required memory. See the *blob.c* program in Appendix B.

*Multiple calls to
imBlobGetResult()*

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()* is an asynchronous function while *imBlobGetResult()* is a synchronous function (asynchronous functions have a lower overhead than synchronous functions).

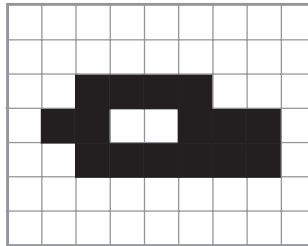
Results for a single blob

If all you need is the result of a specific feature of a single blob, you can use *imBlobGetResultSingle()* instead of *imBlobGetResult()*. The *imBlobGetResultSingle()* function transfers the result of a specified feature of a specified blob.

Transferring or copying runs

You can copy or transfer "run" information about a specified blob using *imBlobCopyRuns()* or *imBlobGetRuns()*. A run is defined as a horizontal sequence of consecutive blob pixels. *imBlobCopyRuns()* and *imBlobGetRuns()* copy or transfer, respectively, the x- and y-coordinate of each run in the blob, as well as the length of each run.

(0, 0)



x-coordinate of each run = 2, 1, 5, 2

y-coordinate of each run = 2, 3, 3, 4

length of each run = 4, 2, 3, 6

* Blob pixels are in black.

Note that run information can be used to calculate features that are not supported by the blob analysis module.

❖ To use *imBlobCopyRuns()* or *imBlobGetRuns()*, you must not disable the saving of runs (using *imBlobControl()*). In addition, you must calculate the total number of runs in a blob (IM_BLOB_NUMBER_OF_RUNS).

Chapter 6: Pattern matching

This chapter describes how to perform a pattern matching operation.

Pattern matching

Pattern matching is a branch of image analysis that allows you to search for a pattern in an image. The pattern for which you are searching is called the *model* and the image being searched is called the *target image*.

Pattern matching can be used in a variety of applications. In machine guidance, for example, it can be used to locate mounting holes on a circuit board, so that a mechanical device can insert screws into these holes (use an image of a typical mounting hole as the model, and an image of a circuit board as the target image). In machine vision, pattern matching can be used to determine the degree by which an object is misaligned (compare the object's coordinates in the target image with its coordinates in a correctly aligned target image).

Note that the pattern matching module operates on 8-bit unsigned buffers.

Comparing images against a pattern

If you simply want to know how similar areas of an image are to a pattern, you can use *imBinMorphic()* (for binary images) or *imIntCorrelate()* (for integer images). These functions do not perform the search algorithm used by the pattern matching module; they simply compare areas of an image to a pattern and write results to a destination buffer. Note that they can take much longer to execute than the pattern matching functions, particularly for large patterns.

For more on *imBinMorphic()* and *imIntCorrelate()*, see the *Genesis Native Library Command Reference*.

General steps

The general steps to perform a pattern matching operation are:

1. Create your model, using *imPatAllocModel()*, or restore a previously saved model from disk, using *imPatRestore()*.
2. Adjust search parameters, if the defaults are not suitable, using the *imPatSet...()* functions.
3. If necessary, preprocess the model, using *imPatPreprocModel()*, so as to increase search speed.
4. Allocate a pattern matching result buffer, using *imPatAllocResult()*. A pattern matching result buffer is used to store the results of a pattern matching operation.
5. Search for the model in a target image, using *imPatFindModel()*. You might want to first process the target image to improve its quality.
6. Transfer necessary results to the Host, using *imPatGetNumber()* and/or *imPatGetResult()*.
7. Repeat steps 5 and 6 for other target images, if necessary.
8. Free the model and the result buffer, using *imPatFree()*.

An example

The following code searches for a model in an image, then transfers results to the Host. The model was allocated previously and saved in a file. Note that this code is part of the *pat.c* program and requires the Genesis display section. See Appendix B for the complete *pat.c* program.

```

long Model;           /* Pattern matching model */
long Result;          /* Pattern matching result buffer */
IM_PAT_RESULT_ST Res; /* All match results */

/* Restore the model */
imPatRestore(Thread, ModelFile, &Model);

/* 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);
}

```

Creating the model

Models are created from rectangular areas of existing images (called *model images*). Before you create your model, you might want to process the model image to improve its quality.

When you create a model, you should keep the following in mind:

- Matches might be missed if the model has a different size or orientation in the target image. Therefore, do not use a model that might appear with a different size or orientation in the target image. Note that a difference in orientation of a few degrees or a size difference of a few percent is normally acceptable.
- False matches can occur if something else in your target image happens to look like your model. In general, a large model has less chance of being confused with something else because more of the model has to match. In addition, large models are faster to find than small models due to the search algorithm used by the pattern matching module (although very large models can also be time-consuming). An efficient model size is approximately 128x128 pixels if you are searching a large area.

Note that the search algorithm is described later in the chapter.

Preprocessing the model

To determine which shortcuts can be safely used during a search, you can preprocess the model. Preprocessing attempts to produce more efficient searches, without affecting results. However, if you have a model that is difficult to find, shortcuts might not be possible and the search will therefore not be faster.

Preprocessing makes the most difference on models with just a few, large features. Models with many small-scale features do not benefit as much. In addition, if you are performing just one search, you might want to skip the preprocessing, since the preprocessing might take more time than it saves.

Typical target images

When you use *imPatPreprocModel()*, you can provide a typical target image on which the model will be used. This can result in further shortcuts and, therefore, in even more efficient searches. However, you should only provide a target image if all target images you will be using have the same type of background. If the target images might have different backgrounds, do not provide one to *imPatPreprocModel()*.

Saving/restoring

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()* (discussed in the next section), the effect of preprocessing is undone; in this case, you will need to preprocess again.

Adjusting search parameters

With the Genesis Native Library, you can control certain aspects of a pattern matching operation, using the *imPatSet...()* functions. You can control:

- The acceptance level, using *imPatSetAcceptance()*.
- The number of matches to find, using *imPatSetNumber()*.
- The search region, using *imPatSetPosition()*.
- The positional accuracy, using *imPatSetAccuracy()*.
- The certainty level, using *imPatSetCertainty()*.
- The model's "don't care" pixels, using *imPatSetDontCare()*.
- The effective center ("hot spot") of a model, using *imPatSetCenter()*.
- The search speed, using *imPatSetSpeed()*.

In addition to the above, you can control the search more precisely using *imPatSetSearchParameter()*. To use this function effectively, however, you need to understand the algorithm used by the pattern matching module; see *The pattern matching algorithm* section for details.

Acceptance level

A search is performed by assigning a match score to each pixel in the target image, based on how closely the model and the region around that pixel match. The acceptance level is the match score above which a match is considered to be found. In other words, if the match score of a pixel is above the acceptance level, there is a match at that position; if the match score is below the acceptance level, there is not a match.

Note that match scores of 100% are generally impossible because of noise. When a region of the target image actually matches the model, the match score will typically be between 80 and 100%. The default acceptance level is therefore slightly below this (70%). If your images have a lot of noise, you might have to set the acceptance level below 70%. Note, however, that if you set the acceptance level too low, false matches might occur.

Number of matches

By default, the search algorithm finds only one match: the one with the highest match score above the acceptance level. If necessary, however, you can specify that the search algorithm find n matches. In this case, the n highest match scores above the acceptance level are returned, in decreasing order of match score. The more matches you require, the longer the search process.

Note that the number of results returned might be less than the number you requested since only matches above the acceptance level are returned. Before you retrieve results using *imPatGetResult()*, you can call *imPatGetNumber()* to determine how much memory is required for the results. The *imPatGetNumber()* function returns the actual 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.

Best reject score

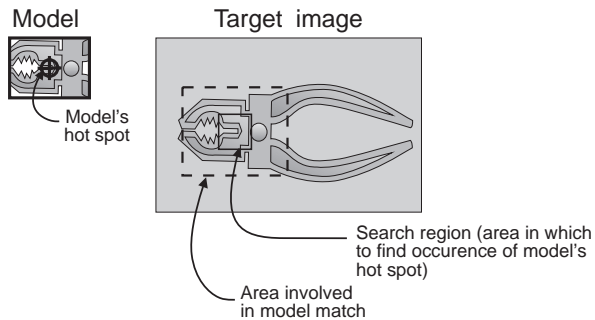
Genesis Native Library keeps track of the largest match score that was rejected during the search. After calling *imPatFindModel()*, you can call *imPatInquire()* with `IM_PAT_BEST_REJECT_SCORE` to get information about the highest match score that was not returned as a match result (either because it was below the acceptance level, or because you did not ask for enough matches). A situation when this inquiry is useful is when you expect only one match, but also want to know if ever there are two (or more) matches. It is faster to search for one match and inquire the best reject score than to search for two (or more) matches.

Model's hot spot

By default, the returned coordinates of a match are the coordinates of the model's center pixel (the model's "hot spot"), relative to the top-left corner of the image. There might be cases, however, when you want the coordinates to refer to something else. For example, if your model has a hole and you want results for this hole, use the *imPatSetCenter()* function to set the model's hot spot to the coordinates of this hole, relative to the top-left corner of the model.

Search region

The search region is the area of the target image in which to search for the model (i.e., the area in which to find the model's hot spot). By default, the search region is the entire target image. To increase search speed, however, you should make the search region as small as possible. If, for example, you know the model's reference point in the target image, the search region can simply be the expected location plus the maximum amount of displacement expected.



- ❖ In general, you should not use child buffers when you want the search region to be smaller than the entire target image. Child buffers can cause misleading results because the search algorithm will not use the area outside the child buffer.

Size of search region

Note that, since the search region is the area in which to find the model's hot spot, the search region can be even smaller than the model (as small as a single pixel).

Search in one direction

To search in only one direction (x or y), set the other dimension of the search area to 1 pixel. Note that, if a search region dimension is 1 pixel, the model will not be found to sub-pixel accuracy in that direction.

Positional accuracy

Once a match with a score above the acceptance level is found, the search algorithm can refine the position of the match to various degrees of accuracy. Specifically, the position can be refined to within ± 0.5 pixels (low accuracy), ± 0.25 pixels (medium accuracy), or ± 0.1 pixels (high accuracy).

The more accuracy you require, the longer the search process.

Certainty level

The certainty level is the match score (usually higher than that of the acceptance level) above which the algorithm can assume that it has found a match and can stop searching the rest of the image for a better score. The certainty level is very important because it can greatly affect the speed of the search. To understand why, you need to know a little about how the search algorithm works.

Since a brute force correlation of the entire model, at every point of the image, might take several minutes, it is not practical. Therefore, the algorithm has to be intelligent. It first performs a rough but quick search to find likely match candidates, then checks out these candidates in more detail to see which are acceptable.

A significant amount of time can be saved if several candidate matches never have to be examined in detail. This can be done by setting an appropriate certainty level. A good level is slightly lower than the expected score. If you absolutely must have the best match in the image, set the level to 100%. This would be necessary if, for example, you expect the image to contain other

patterns that look similar to your model. Unwanted patterns might have a high score, but this will force the search algorithm to ignore them.

Often, you know that the pattern you want is unique in the image, so anything that reaches the acceptance level must be the match you want; therefore, you can set the certainty and acceptance levels to the same value.

Another common case is a pattern that usually produces very good scores (say above 80%), but occasionally a degraded image produces a much lower score (say 50%). Obviously, you must set the acceptance level to 50% or you will never get a match in the degraded image. But what value is appropriate for the certainty level? If you set it to 50%, you take a risk that it will find a false match (above 50%) in a good image before it finds the real match that scores 90%. A better value is about 80%, meaning that most of the time the search will stop as soon as it sees the real match, but in a degraded image (where nothing reaches the certainty level) it will take the extra time to look for the best match that reaches the acceptance level.

"Don't care" pixels

Model pixels that are set to the "don't care" state are ignored during the search process (they do not affect the match score).

Setting model pixels to "don't care" can be useful if your model contains areas that have nothing to do with the pattern for which you are searching. For example, if the required pattern is circular in shape, your model will necessarily contain some unwanted areas of the model image (since models must be rectangular). If these unwanted areas are different in the target image, their presence will affect the match score and could result in matches being missed or false matches being found.

Search speed

You can specify the speed (high, medium, low, very low) at which to perform the search algorithm using *imPatSetSpeed()*. 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.

High speed

You can search at high speed if you have a good quality target image or a simple model. A high-speed search takes all possible shortcuts that were determined by the preprocessing. Searching at high speed is therefore most useful if you preprocessed your model using *imPatPreprocModel()*.

Note that you should not search at high speed if you need the highest possible accuracy (search at medium or low speed instead).

Medium speed

You should search at medium speed (the default setting) if your target images are of medium quality or if your model is complex.

Low speeds

You should search at low or very low speeds only if your target image is of particularly poor quality or if you have encountered problems at higher speeds.

Speeding up the search

The following is a summary of the ways you can speed up a search. Note that most of these were discussed in previous sections of the chapter.

- Search at the highest possible speed. Use *imPatSetSpeed()* to set the search speed.
- Preprocess the model, using *imPatPreprocModel()*.
- Use the smallest possible search region; the search time is roughly proportional to the area searched. Use *imPatSetPosition()* to define the search region.
- Use the lowest possible positional accuracy; the more accuracy you require, the longer the search process. Use *imPatSetAccuracy()* to set the positional accuracy.
- Set the certainty level to the lowest reasonable value (so that the search can stop as soon as a good match is found). Use *imPatSetCertainty()* to set the certainty level.
- Use an efficient model size. Large models are generally faster to find than small models (although very large models can also be time-consuming). An efficient model size is approximately 128x128 pixels if you are searching a large area.
- Use the fastest model allocation algorithm. If model allocation is time critical in your application, you can speed it up by passing one of the following two flag combinations for the Type parameter of *imPatAllocModel()*:
 - IM_NORMALIZED + IM_FAST, or
 - IM_NORMALIZED + IM_VERY_FAST.

Note that IM_FAST allocation leads to some very small differences in the model, but this should not affect pattern matching in most applications. Passing the IM_VERY_FAST flag will allow the fastest possible model allocation, but is a bit more likely to cause problems than IM_FAST.

- ❖ For a discussion of more advanced methods that allow you to speed up the search, refer to *The pattern matching algorithm* section of this chapter.

Saving/restoring

Managing models

You can save a model to disk using *imPatSave()*, as well as restore it from a file using *imPatRestore()*. Note that the model's search parameters are also saved/restored. If the model was preprocessed, the preprocessing changes are also saved/restored.

When you restore a model from file, you might want to inquire about the model's search parameters. You can do so using *imPatInquire()*.

Reading/writing

You can read a model from an open file using *imPatRead()*, as well as write it to an open file using *imPatWrite()*. This can be useful if you want to save/restore several models to/from the same file. As with the save/restore functions, the characteristics of the model are also read/written.

Copying

You can copy a model to an on-board buffer using *imPatCopy()*. This can be useful if you want to view the model. The *imPatCopy()* function can also be used to copy only the model's "don't care" pixels.

Rotating models

You can rotate a model using *imPatAllocRotatedModel()*. This function can rotate a model by 0, 90, 180, or 270°. This can be useful if the model appears at a different angle in the target image.

The pattern matching algorithm

Normalized grayscale correlation is widely used in industry for pattern matching applications. Although in many cases you do not need to know how the search operation is performed, an understanding of the algorithm can sometimes help you pick an optimal search strategy.

Normalized Correlation

The correlation operation can be seen as a form of convolution, where the pattern matching model is analogous to the convolution kernel. In fact, ordinary (un-normalized) correlation is exactly the same as a convolution:

$$r = \sum_{i=1}^{i=N} I_i M_i$$

In other words, for each result, the N pixels of the model are multiplied by the N underlying image pixels, and these products are summed. Note that the model doesn't have to be rectangular, because it can contain "don't care" pixels that are completely ignored during the calculation. When the correlation function is evaluated at every pixel in the target image, the locations where the result is largest are those where the surrounding image is most similar to the model. The search algorithm then has to locate these peaks in the correlation result, and return their positions.

Unfortunately, with ordinary correlation, the result increases if the image gets brighter. In fact, the function reaches a maximum when the image is uniformly white, even though at this point it no longer looks like the model. The solution is to use a more complex, normalized version of the correlation function (the subscripts have been removed for clarity, but the summation is still over the N model pixels that are not "don't cares"):

$$r = \frac{N \sum IM - (\sum I) \sum M}{\sqrt{[N \sum I^2 - (\sum I)^2][N \sum M^2 - (\sum M)^2]}}$$

With this expression, the result is unaffected by linear changes (constant gain and offset) in the image or model pixel values. The result reaches its maximum value of 1 where the image and model match exactly, gives 0 where the model and image are uncorrelated, and is negative where the similarity is less than might be expected by chance.

Normally, we are not interested in negative values, so results are clipped to 0. In addition, we use r^2 instead of r to avoid the slow square-root operation. Finally, the result is converted to a percentage, where 100% represents a perfect match. So, the match score returned by *imPatGetResult()* is actually:

$$Score = \max(r, 0)^2 \times 100\%$$

- ❖ If you are also interested in finding negative versions of the model, you can take the absolute values of the scores, rather than clipping negative scores to 0, using *imPatSetSearchParameter*. In this case, $Score = |r|^2 \times 100\%$.

Note that some of the terms in the normalized correlation function depend only on the model, and hence can be evaluated once and for all when the model is defined. The only terms that need to be calculated during the search are:

$$\sum I, \sum I^2, \sum IM$$

This amounts to two multiplications and three additions for each model pixel.

A typical application might need to find a 128x128-pixel model in a 512x512-pixel image. In such a case, the total number of arithmetic operations needed for an exhaustive search is $5 \times 512^2 \times 128^2$, or over 20 billion. Even on a 'C80, this would take several minutes, much more than the 10 milliseconds or so the search actually takes. Clearly, *imPatFindModel()* does much more than evaluate the correlation function at every pixel in the search area and return the location of the peak scores.

Hierarchical Search

A reliable method of reducing the number of computations is to perform a so-called hierarchical search. Basically, a series of smaller, lower-resolution versions of both the image and the model are produced, and the search begins on a much-reduced scale. This series of sub-sampled images is sometimes called a resolution pyramid, because of the shape formed when the different resolution levels are stacked on top of each other.

Each level of the pyramid is half the size of the previous one, and is produced by applying a low-pass filter before sub-sampling. If level 0 (the original image or model) is 512x512, then level 1 is 256x256, level 2 is 128x128, and so on. Therefore, the higher the level in the pyramid, the lower the resolution of the image and model.

The search starts at low resolution to quickly find likely match candidates. It proceeds to higher and higher resolution to refine the positional accuracy and make sure that the matches found at low resolution actually were occurrences of the model. Because the position is already known from the previous level (to within a pixel or so), the correlation function need be evaluated only at a very small number of locations.

Since each higher level in the pyramid reduces the number of computations by a factor of 16, it is usually desirable to start at as high a level as possible. However, the search algorithm must trade off the reduction in search time against the increased chance of not finding the pattern at very low resolution. Therefore, it chooses a starting level according to the size of the model and the characteristics of the pattern. In the application described earlier (128x128 model and 512x512 image), it might start the search at level 4, which would mean using an 8x8 version of the model and a 32x32 version of the target image. You can, if desired, force a specific starting level, using *imPatSetSearchParameter()*.

The last (lowest) level used is usually determined by the specified positional accuracy; however, you can set this explicitly, using *imPatSetSearchParameter()*.

The logic of a hierarchical search accounts for a seemingly counter-intuitive characteristic of *imPatFindModel()*: large models tend to be found faster than small ones. This is because a small model cannot be sub-sampled as much without losing all detail. Therefore, the search must begin at fairly high resolution (low level), where the relatively large search area results in a longer search time. Thus, small models can only be found quickly in fairly small search areas.

Using the best reject score and best reject level

As mentioned earlier, after calling *imPatFindModel()*, you can call *imPatInquire()* to determine the highest match peak score that was rejected (not returned as a match result) during the search. At the same time, because the peak's match score 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. Both values can be inquired as follows:

```
imPatInquire(ResultBuffer, IM_PAT_BEST_REJECT_SCORE, &RejectScore);
imPatInquire(ResultBuffer, IM_PAT_BEST_REJECT_LEVEL, &RejectLevel);
```

Search Heuristics

Even though performed at very low resolution, the initial search still accounts for most of the computation time if the correlation is performed at every pixel in the search area. For most models, match peaks (pixel locations where the surrounding image is similar to the model and correlation results are largest) are several pixels wide. These can be found without evaluating the correlation function everywhere.

imPatPreprocModel() analyses the shape of the match peak produced by the model, and determines if it is safe to try to find peaks faster. If the pattern produces a very narrow match peak (or the model was not pre-processed), an exhaustive initial search is performed. The search algorithm tends to be conservative; if necessary, force fast peak finding, using *imPatSetSearchParameter()*.

Genesis Native Library also allows you to set the threshold level (in %) for rejecting candidate model peaks at low resolution levels, using *imPatSetSearchParameter()* with `IM_PAT_REJECTION`. Any candidates found below this threshold level will be rejected. 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. The `IM_PAT_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.

At the last (high-resolution) stage of the search, the model is large, so this stage can take a significant amount of time, even though the correlation function is evaluated at only a very few points. To save time, you can select high search speed, using *imPatSetSpeed()*. Only every second model pixel will be used. For most models, this has little effect on the score or accuracy, but does increase speed. However, if accuracy is your primary concern, you should use all model pixels, that is, avoid high speed or force the use of all model pixels with *imPatSetSearchParameter()*.

Sub-Pixel Accuracy

The highest match score occurs at only one pixel position, and drops off around this peak. The exact (sub-pixel) position of the model can be estimated from the match scores found on either side of the peak. A curve is fitted to the match scores around the peak and, from the equation of the curve, the exact peak position is calculated. The curve is also used to improve the estimate of the match score itself, which should be slightly higher at the true peak position than the actual measured value at the nearest whole pixel.

The actual accuracy that can be obtained depends on several factors, including the noise in the image and the particular pattern of the model. However, if these factors are ignored, the absolute limit on accuracy, imposed by the algorithm itself and by the number of bits of precision used to hold the correlation result, is about 0.05 pixels. This is the worst-case error measured in X or Y when an image is artificially shifted by fractions of a pixel. In a real application, accuracy better than 0.1 pixel might be achieved for low-noise images; however, it is better not to rely on more than a 0.1 pixel accuracy. These numbers apply if you select high search accuracy, using *imPatSetAccuracy()*, in which case the search always proceeds to resolution level 0.

If you select medium accuracy (the default), the search may stop at resolution level 1, and hence the accuracy is about half of what can be attained at level 0 (0.25 pixels). Selecting low accuracy may cause the search to stop at level 2, so the accuracy is reduced by an additional factor of two (to about 0.5 pixels).

Chapter 7: Compression

This chapter describes how to compress and decompress images using the run-length encoding module and the JPEG module.

Introduction

Compression allows more images to be stored on-board than would normally be possible. In addition, it reduces the amount of data that must be transferred off-board when images are saved to file, allowing quicker transfers. There are two methods that the Genesis Native Library provides to compress and decompress images: run-length encoding and JPEG compression.

The run-length encoding method can be used to compress images that are originally in a binary format, with 1- or 8-bit pixel depth.

The JPEG compression method can be used to compress images that are originally in a grayscale or color format with 8- or 16-bits per band, and from 1 to 4 bands. Note that the Genesis Native Library does not support all JPEG modes, so you might not be able to decompress a JPEG file that was compressed by another package (refer to the section on JPEG compatibility issues later in this chapter).

Run-length encoding and decoding

The run-length encoding module of the Genesis Native Library allows you to compress and decompress images one at a time. The run-length encoding module performs compression on binary data by encoding information about a run of connected background or foreground pixels in a single byte (8 bits), then storing it into a buffer. A run is a continuous stretch of connected pixels made up of foreground or background pixels.

Supported image types

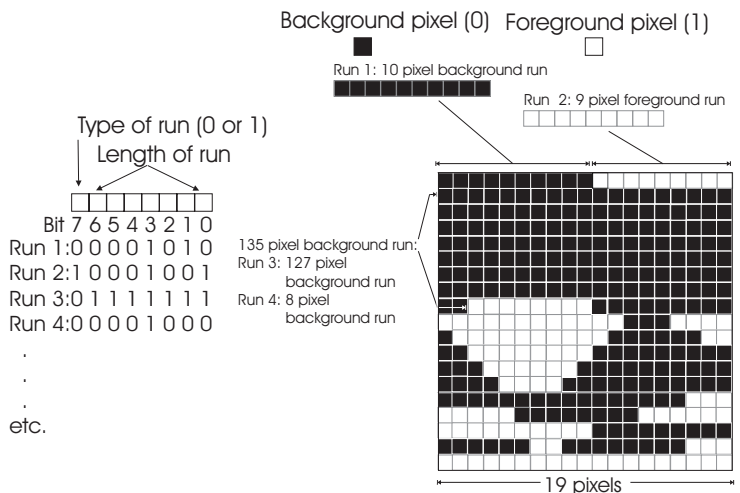
The run-length encoding algorithm can compress any image that is in a binary format. This includes packed binary format (1 bit/pixel), as well as monochrome (8 bits/pixel) image data. For the purpose of run-length encoding, all pixels are treated as either 0 or non-zero. All non-zero pixels are considered to be foreground pixels.

*Mode of operation***Run-length encoding (compression)**

The run-length encoding module performs compression on binary data from an image buffer by storing each continuous run of connected pixels as a single byte (8 bits). The most significant bit (bit 7) is used to indicate the type of run: 1 for a foreground run or 0 for a background run. Each run of pixels is composed exclusively of background or foreground pixels. Pixels that have a value of 0 are considered background pixels. All pixels with a non-zero value are considered foreground pixels.

The lower seven bits (bit 0 to bit 6) are used to indicate the length of the run. Since seven bits are used to indicate the length of a run, the maximum length of a single run is 127. Accordingly, a run of pixels that is larger than 127 is broken into multiple runs. For example, a run of 135 foreground pixels will be broken into two foreground runs: one run of 127 pixels, and the other run of 8 pixels.

It is important to note that a run can span lines. A run of pixels is not broken at the end of a line, but continues on to the starting position of the next line, as long as the type of pixel remains the same (See run #3 in the diagram below).

Run-length encoding byte compression

Keep track of original image dimensions

You should keep a record of the dimensions of the original image, so that a buffer of suitable dimensions can be allocated later for any subsequent decoding. This record is necessary because the compressed buffer does not contain any header information.

Important point to consider...

Since images that are in a packed binary (1 bit/pixel) format are already 8 times smaller than the typical monochrome 8-bit image, using run-length encoding to compress data is only useful if this type of compression would lead to an even greater compression ratio. That is, if the average run length is greater than 8.

Decoding run-length encoded images (decompression)

The run-length module performs decompression of run-length encoded data. Each consecutive byte can be decoded back into the appropriate number of 1- or 8-bit pixels. When saving decoded data in an 8-bit buffer, foreground pixels are set to 255 by default. Since the encoded data does not contain any header information, the destination buffer must be exactly the right size.

General steps

Encoding (compression)

To encode images, you need to follow these steps:

1. Allocate a 1-dimensional, 8-bit buffer that is large enough to hold the compressed data. Don't worry if the compression buffer you allocate is too big because the buffer size necessary to contain the compressed data (in bytes) will be written to the `IM_RLE_SIZE` control field.
2. Run-length encode the image data using `imRleEncode()`.
3. Read the size of the compressed buffer by calling `imBufGetField()` with `IM_RLE_SIZE`.
4. Keep a record of the dimensions of the original image, so that a buffer of suitable dimensions can be allocated later for any subsequent decoding.

One way to do so is to keep a record of the width (SizeX) and height (SizeY) of the original image buffer within the compressed buffer in user-defined fields.

5. If you want to append other image data to the compression buffer, add the required start position control field (IM_RLE_START) to the control buffer. Then, call *imRleEncode()* again. This time, the value written to the IM_RLE_SIZE control field will be that of the last compressed image.

Repeat steps 4 and 5 as often as is required to encode additional image data.

To perform a subsequent operation, such as save, with the encoded data, follow these steps.

1. Allocate a child buffer (from parent compressed buffer) with a buffer size (IM_RLE_SIZE) corresponding to the compressed size of a particular run-length encoded image, and appropriate offset (IM_RLE_START). This child buffer will contain the compressed image data.
2. Use the child buffer to perform any subsequent processing operations or image data transfers.

Decoding (decompression)

To decode run-length encoded images, you need to follow these steps:

1. Allocate a buffer with the original image buffer dimensions.
2. Decode the run-length encoded buffer into an image buffer, using *imRleDecode()*.

Control options

With the Genesis Native Library, you can control certain aspects of the compression/decompression process. This is done using the control fields of a control buffer.

When encoding, you can control the starting position of the next run-length encoded byte within the buffer. This allows you to append images to the same compression buffer.

When decoding the image from a run-length compressed buffer, you have the option to control the starting position (IM_RLE_START in bytes) within the compression buffer to run-length decode. If multiple images are encoded within the

compression buffer, refer to your records on the size of each image in bytes (IM_RLE_SIZE) to determine the appropriate starting position within the buffer. In addition, you have the option to choose the foreground and/or background color.

Example

The following example allocates the appropriate image buffers, uses *imRleEncode()* to compress, and prints the size of the compressed buffer. The compressed buffer is then decompressed using *imRleDecode()*.

```
long SrcBuf, DstBuf;
long CompBuf, CompChild, Control, Size, SizeX, SizeY;

/* Allocate uncompressed buffers of correct size. */
imBufAlloc2d(Thread, SizeX, SizeY, IM_UBYTE, IM_PROC, &SrcBuf);
imBufAlloc2d(Thread, SizeX, SizeY, IM_UBYTE, IM_PROC, &DstBuf);

/* Assume SrcBuf contains a binary image to be run-length encoded. */
...
/* Allocate a 1-D buffer (more than large enough) for the compressed image. */
imBufAlloc1d(Thread, SizeX*SizeY, IM_UBYTE, IM_PROC, &CompBuf);

/* Allocate a control buffer. */
imBufAllocControl(Thread, &Control);

/* Encode(compress) the image data. Then, get and report compressed size. */
imRleEncode(Thread, SrcBuf, CompBuf, Control, 0);
imBufGetField(Thread, Control, IM_RLE_SIZE, &Size);
printf("Compressed size= %i\n", Size);

/* Optional: Save compressed child buffer to disk with user-defined fields
 *          to save original dimensions.
 */
if (Save)
{
    imBufChild(Thread, CompBuf, 0, 0, Size, 1, &CompChild);
    imBufPutField(Thread, CompChild, 1, SizeX);
    imBufPutField(Thread, CompChild, 2, SizeY);
    imBufSave(Thread, FileName, IM_NATIVE, CompChild);
    imBufFree(Thread, CompChild);
}
...

/* Decode (decompress) the compressed buffer. */
imRleDecode(Thread, DstBuf, CompBuf, Control, 0);
```

JPEG Compression

The JPEG module of the Genesis Native Library also allows you to compress and decompress images.

Modes of operation

The module can compress images using the JPEG lossless algorithm or the JPEG lossy algorithm (baseline sequential mode). The JPEG lossless algorithm compresses images without any loss of information. Typically, the JPEG lossless algorithm compresses images by a factor of 2:1, although a factor of 4:1 can sometimes be achieved. The JPEG lossy algorithm introduces some loss of information but compresses images by a variable factor. The higher the specified factor, the more the compression, but the lower the image quality.

Note that lossless mode is the only mode supported by the NOA, so it is the fastest way to do JPEG compression on Matrox Genesis.

Supported types

The JPEG lossless algorithm can compress 8-bit or 16-bit buffers, with up to four bands. The JPEG lossy algorithm can compress 8-bit buffers, with up to four bands.

Control options

With the Genesis Native Library, you can control certain aspects of the compression/decompression. For example, you can use your own compression/decompression tables, or you can compress an image piece by piece if it is too large to be held entirely in memory.

File format and restrictions

When the Genesis Native Library compresses an image, it adds some Genesis-specific markers to the resulting image. Most other packages will ignore these markers and therefore be able to decompress the file. The Genesis Native Library itself ignores unrecognized application-specific markers when it decompresses a file. However, the Genesis Native Library can still decompress a standard JFIF (JPEG File Interchange Format) file that contains just a grayscale image. If the JFIF file contains a multi-band image, the Genesis Native Library can decompress it only if each band is stored separately in the file (if the bands are stored in an interleaved fashion, the Genesis Native Library cannot decompress it).

Compression

General steps

To compress images, you need to follow these steps:

1. Allocate a JPEG buffer, using *imJpegAlloc()*. JPEG buffers are used to store compressed images.
2. Add the required controls to the JPEG buffer, if the defaults do not meet your needs.
3. Compress the image, using *imJpegEncode()*.
4. If necessary, save the compressed image from the JPEG buffer to disk, using *imJpegSave()*. Alternatively, write the data from the JPEG buffer to an ordinary (contiguous) buffer, using *imJpegWriteBuf()*, and then save the data to disk in whatever way is fastest on your system.

Decompression

To decompress images, you need to follow these steps:

1. Load the compressed image into a JPEG buffer, using *imJpegRestore()*, *imJpegRead()*, or *imJpegReadBuf()*. *imJpegRestore()* and *imJpegRead()* load compressed images from files. *imJpegReadBuf()* loads compressed images from ordinary (contiguous) buffers.
2. Decompress the JPEG buffer into an image buffer, using *imJpegDecode()*.

Note that the buffer to which you write the decompressed image should be compatible with the original image buffer (same size, number of bands, and data type). If you do not know what these buffer parameters should be, you can use *imJpegInquire()*. This function inquires about a specified attribute of a JPEG buffer (such as the data type of the original image), and returns the value of this attribute.

Freeing JPEG buffers

Once a JPEG buffer is no longer needed, you should free the memory allocated to it using *imJpegFree()*.

Control options

When a buffer is allocated using *imJpegAlloc()*, the default values for all control settings are stored in that buffer. These default values are suitable for most applications. However, you can change these settings to meet your needs, using *imJpegControl()*, *imJpegControlBand()*, and/or *imJpegPutTable()*. Note that you must use *imJpegControl()* to specify lossless compression (the default is lossy compression).

Some control settings only apply to JPEG lossless compressions and some only apply to JPEG lossy compressions. JPEG lossless controls are discussed in the section *Controlling JPEG lossless compression*.

Note that, when loading a compressed image, all controls that were used to perform the compression are copied 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.

For a complete list of control settings and their default values, see the *Genesis Native Library Command Reference*.

A compression example

The following code compresses (encodes) an image using the lossless mode. Note that this code is part of the *jpeg.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *jpeg.c* program.

```
long JpegBuf;  /* Compressed image */

/* 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);
```

Another compression example

The following code is similar to the previous example, except that it uses *imJpegWriteBuf()*, and then *imBufMap()*, to write the data to disk. Depending on your system, this might be faster than saving the data directly to disk using *imJpegSave()*. Note that the buffer in which to write the compressed image was allocated in Host memory. You could also allocate the buffer in processing memory and then transfer it to the Host, if this is faster. However, you should be aware of certain restrictions (see the *Genesis Native Library Command Reference* for details).

```
unsigned char *Address;
...

/* Allocate a JPEG buffer and select lossless mode */
imJpegAlloc(Thread, 0, &JpegBuf);
imJpegControl(Thread, JpegBuf, IM_JPEG_MODE, IM_LOSSLESS);

/* Compress the image */
imJpegEncode(Thread, ImageBuf, JpegBuf, 0);

/* Allocate a contiguous Host buffer big enough for the result */
imJpegInquire(Thread, JpegBuf, IM_JPEG_SIZE, &Size);
imBufAllocId(Thread, Size, IM_UBYTE, IM_HOST, &HostBuf);

/* Write the compressed image to the buffer */
imJpegWriteBuf(Thread, HostBuf, JpegBuf, 0, 0);

/* Get a pointer to the data in Host memory and write it to disk */
imBufMap(Thread, HostBuf, 0, 0, (void **)&Address, &Dummy, &Dummy);
fwrite(Address, Size, 1, Stream);
```

A decompression example

The following code decompresses (decodes) an image. *imJpegInquire()* is first used to allocate an appropriate buffer in which to save the decompressed image. Note that this code is part of the *jpeg.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *jpeg.c* program.

```
long JpegBuf; /* Compressed image */

/* 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);
```

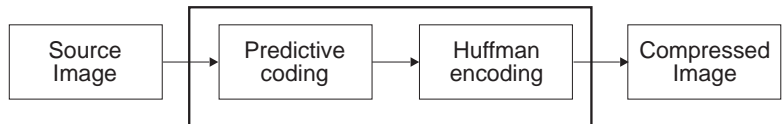
Controlling JPEG lossless compression

This section provides an overview of the JPEG lossless algorithm, and of the controls you have over this algorithm. You should only change these controls if you are familiar with the JPEG lossless algorithm. Changing these controls might allow you to achieve a higher compression ratio than would be possible using the defaults of the Genesis Native Library.

For detailed information about the JPEG algorithm, refer to the *JPEG Technical Specification Revision 8*.

JPEG lossless

The JPEG lossless algorithm is basically a two-step process. First, predictive coding is performed on the image. Then, the result is Huffman encoded.



Predictive coding

Predictive coding is based on the fact that adjacent pixels in an image generally have similar values. Therefore, the value of a pixel can be "predicted" from the values of its neighbor(s). The difference between the original value of the pixel and the predicted value requires fewer bits to store than the original pixel value.

By default, the Genesis Native Library uses the pixel to the left to predict values. This is suitable for most images. However, you can specify that no predicting be done, using *im.JpegControl()*. In this case, the values after predictive coding will be the same as the original values. This can be useful if you have developed your own algorithm to take the place of predictive coding and only need your images Huffman encoded. Note that you must implement your own algorithm to use one of the other "predictors" supported by the JPEG lossless algorithm (the Genesis Native Library only directly supports predictor #1: the "pixel to the left" predictor).

Huffman encoding

After an image has been predictive coded, Huffman encoding assigns a variable-length "code word" to each value. This code is based on the number of bits by which adjacent values differ. By storing the code word, rather than the actual difference value, further compression can be achieved. Values are assigned code words according to a Huffman table.

The Genesis default Huffman table can handle images with up to 16-bits per band (for lossless mode). This same table is used even for 8-bit images and is suitable for most images. However, there are a few JPEG lossless decoders that require a smaller Huffman table for 8-bit images. Refer to the example at the end of this section for a good Huffman table to use on 8-bit images when portability is important.

Using your own table

If you do not want to use the default Huffman table provided, require an optimal compression ratio, and are familiar with the JPEG lossless algorithm, you can use your own Huffman table. If you use your own Huffman table, you first need to represent it by a one-dimensional array. The first 16 numbers in the array should represent the number of code words used for a given code length. For example, if the first 16 numbers are:

0, 1, 5, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0

it means that there are no 1-bit codes, one 2-bit code, five 3-bit codes, one 4-bit code, one 5-bit code, etc.

The next numbers in the array determine which code word is assigned to each value. Specifically, if these numbers are n_0 , n_1 , n_2 , etc., it means that the shortest code word is assigned to a value whenever adjacent values differ by n_0 bits; the next shortest code word is assigned to a value whenever adjacent values differ by n_1 bits; etc. Note that, if these numbers are 0, 1, 2, etc., progressively longer words are assigned to larger differences (this is usually the best sequence, since adjacent values tend to be similar rather than different).

Once you have created the array, you need to transfer it to the required JPEG buffer, using *imJpegPutTable()*. Specify that you are transferring a DC Huffman table (see the following example).

An example

The following code compresses an image with a user-defined Huffman table.

```
long Jpeg;
unsigned char HuffTable[28] =
{ 0, 1, 5, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
};

/* Allocate a JPEG buffer */
imJpegAlloc(Thr, 0, &Jpeg);

/* Put the table into the JPEG buffer */
imJpegPutTable(Thr, Jpeg, IM_JPEG_TABLE_DC, 0, 28, HuffTable);

/* Compress the image and save it to disk */
imJpegEncode(Thr, Image, Jpeg, 0);
imJpegSave(Thr, "lossless.jpg", Jpeg);

/* Free the memory allocated to the JPEG buffer */
imJpegFree(Thr, Jpeg);
```

Encoding a very large image

If an image is too large to be held entirely in memory, you can compress it block by block into a JPEG buffer. This is done by first specifying the number of blocks by which to divide an image, using *imJpegControl()*. You then write a block of the image to an image buffer, compress the image buffer into the same JPEG buffer, and repeat until all blocks have been compressed. Each compressed block is appended to the JPEG buffer, until the entire image has been compressed.

There are some restrictions on the size of each block (except for the last block). For a lossless compression, each block (except the last) must have a Ysize that is a multiple of the IM_JPEG_RESTART_ROWS item of *imJpegControl()* (restart rows are discussed later in this chapter). For a lossy compression, each block (except the last) must have a Xsize that is a multiple of 8 times the IM_JPEG_RESTART_ROWS item, and must have a Ysize that is a multiple of 8.

Note that, if necessary, you can stop a block-by-block compression, using *imJpegControl()*.

An example

The following code compresses an image block by block.

```
long Jpeg;

/* Allocate a JPEG buffer */
imJpegAlloc(Thr, 0, &Jpeg);

/* Specify the number of blocks */
imJpegControl(Thr, Jpeg, IM_JPEG_NUM_BLOCKS, nblocks);

/* Write and compress, one block at a time */
while (nblocks--)
{
    /* Load or grab next block of image */
    ...

    /* Compress the block */
    imJpegEncode(Thr, Image, Jpeg, 0);
}

/* Save the compressed image to a file */
imJpegSave(Thr, "test.jpg", Jpeg);

/* Free the JPEG buffer */
imJpegFree(Thr, Jpeg);
```

Writing/reading to or from open files

With the Genesis Native Library, you can read a compressed image from an open file (using *imJpegRead()*) or write a compressed image to an open file (using *imJpegWrite()*). This can be useful if you are compressing/decompressing several images to/from the same file (see the examples below). This can also be useful if you need to write/read additional information (such as header information) to/from the same file.

Note that, when an image is compressed, the tables used in the compression are also saved to the JPEG buffer (by default). If images are compressed with the same tables and written to the same file, there is no need to save tables after the first image is written. You can disable the saving of tables using *imJpegControl()*.

Writing to the same
file

The following code compresses several images and writes them to the same file. The saving of tables is disabled after the first image is written.

```
FILE *File;
long Jpeg;

/* Open the file for writing */
File = fopen("test.jpg", "wb");

/* Allocate a JPEG buffer */
imJpegAlloc(Thr, 0, &Jpeg);

/* Compress and write the first image */
imJpegEncode(Thr, Image1, Jpeg, 0);
imJpegWrite(Thr, File, Jpeg);

/* Don't save tables with subsequent images */
imJpegControl(Thr, Jpeg, IM_JPEG_SAVE_TABLES, IM_DISABLE);

/* Compress and write subsequent images */
imJpegEncode(Thr, Image2, Jpeg, 0);
imJpegWrite(Thr, File, Jpeg);

imJpegEncode(Thr, Image3, Jpeg, 0);
imJpegWrite(Thr, File, Jpeg);

/* Free the JPEG buffer */
imJpegFree(Thr, Jpeg);

/* Close the file. There is no need to first synchronize since */
/* imJpegWrite() is synchronous */
fclose(File);
```

Reading from the
same file

The following code decompresses several images from the same file. Only one JPEG buffer is needed since, each time *imJpegRead()* is called, 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.

```
FILE *File;
long Jpeg;

/* Open the file for reading */
File = fopen("test.jpg", "rb");

/* Allocate a JPEG buffer */
imJpegAlloc(Thr, 0, &Jpeg);

/* Read and decompress the first image */
imJpegRead(Thr, File, Jpeg);
imJpegDecode(Thr, Image1, Jpeg, 0);

/* Read and decompress subsequent images */
imJpegRead(Thr, File, Jpeg);
imJpegDecode(Thr, Image2, Jpeg, 0);

imJpegRead(Thr, File, Jpeg);
imJpegDecode(Thr, Image3, Jpeg, 0);

/* Free the JPEG buffer */
imJpegFree(Thr, Jpeg);

/* Close the file. There is no need to first synchronize since */
/* imJpegRead() is synchronous */
fclose(File);
```

Restart markers

When an image is compressed, the Genesis Native Library adds restart markers to the bit stream of the compressed image. A restart marker is a special code that signifies that the encoded bit stream has been padded to the next byte boundary before the encoding process was restarted. Restart markers allow the Genesis Native Library to decompress the image using multiple processors. Files that were compressed by a package without restart markers can still be decompressed by the Genesis Native Library, although not as quickly.

By default, the Genesis Native Library places restart markers after a certain number of rows of data have been encoded (for lossless compressions) or after a certain number of 8x8 blocks of data have been encoded (for lossy compressions). If necessary, you can use *imJpegControl()* to specify that restart markers be placed after every *n* rows of data or after every *n* 8x8 blocks of data. This can be useful if you are transmitting the compressed image over a medium that is susceptible to errors. If an error does occur and there are no restart markers, the error will propagate and affect subsequent data. However, if there are restart markers, the error will be confined to the data between markers. Therefore, if you specify that, for example, restart markers be added after every row or after every 8x8 block, an error will only affect one row or one block of the reconstructed image.

- ❖ For a lossy compression with a high compression ratio, too many restart markers can significantly increase the size of the compressed image. In this case, you might want to increase the restart interval, especially if you are not transmitting the image over a noisy medium.

JPEG compatibility issues

You might have problems reading Genesis JPEG files with other software packages, or reading non-Genesis JPEG files with the Genesis Native Library JPEG functions. There are several reasons for this.

Color images

First, Genesis only supports RGB color images saved in planar format (each color band is encoded separately). Many other software packages misinterpret these files and do not decode or display them properly. This occurs for two reasons:

- The three color bands might be recognized and decoded, but they are assumed to be from a YUV color image instead of an RGB image. Accordingly, an unnecessary color space conversion is performed before displaying the image, and the colors come out completely wrong.
- The image is assumed to be monochrome because it is in planar form, and is decoded as a 1-band image (as if it were monochrome).

In addition, when a color image is compressed by another package, it might not be readable by Genesis for two reasons:

- The image might be encoded in interleaved format (the color components are interleaved on a pixel by pixel basis, not by band). In this case, Genesis will report an error that interleaved format is not supported.
- The image might be saved as YUV rather than RGB, possibly with subsampling of the chrominance (UV) components. Neither YUV format nor subsampling is supported by Genesis so, again, an error will be reported.

Compression modes

Matrox Genesis supports both lossy and lossless JPEG compression modes, but many other software packages do not support lossless mode (it is mainly used in medical imaging, where often no loss of information is acceptable). So if you encode an image in lossless mode and save it to disk, you will likely get an error message when trying to open that file with another software package.

Furthermore, Genesis only supports a subset of the many different lossy JPEG compression modes defined in the full JPEG specification. Monochrome images encoded with the baseline DCT method (8 bits per pixel) usually present no problems and are interchangeable between Genesis and other software packages.

Larger files for a given image quality (or lower quality for same file size)

During the encoding process on Matrox Genesis, the Genesis Native Library only provides a default encoding table, but other packages might optimize the tables for a particular image. Therefore, it is possible that compression on Matrox Genesis produces larger files for a given image quality (or produces lower quality when compressing to a file of the same size). However, Genesis does allow you to load custom tables.

Furthermore, after the image is encoded some extra information is added to help speed up the decoding process. This also makes the compressed file slightly larger. However, you can reduce this extra information by increasing the `IM_JPEG_RESTART_ROWS` value using *imJpegControl()*.

JPEG files produced by other packages will not contain the extra information that Genesis needs, so those files cannot be decoded at full speed by Genesis. Note that this limitation only applies to decoding using the 'C80; decoding of JPEG lossless files by the NOA is always done at full speed.

DC Huffman table

And finally, one other known problem concerns the default DC Huffman table. The Genesis default table can handle images with up to 16-bits per band (for lossless mode); this table is used even for 8-bit images. This should not cause any problems, but there are a few lossless decoders that require a smaller Huffman table for 8-bit images. There is an example presented in the *Huffman encoding* subsection of this chapter for defining a custom DC Huffman table. In fact, that table is a good one to use on 8-bit images when portability is important.

Chapter 8: Generating graphics

*This chapter describes the graphics functions available
with the Genesis Native Library.*

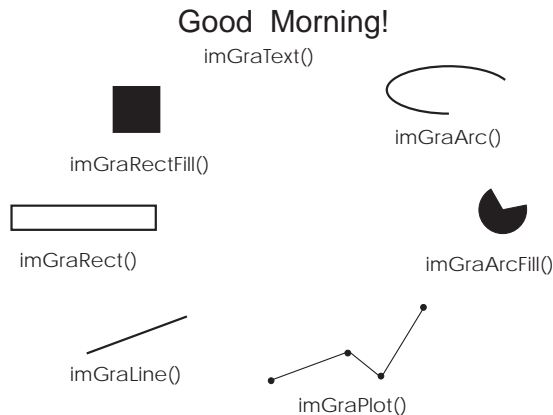
Graphics

The Genesis Native Library contains a set of graphic functions. These functions can draw into any 8-, 16-, or 32-bit integer buffer, and drawing will be clipped to fit the buffer. Note, however, that if you have the display section, you can instead draw into the overlay buffer using the on-board high-performance MGA-2064W graphics accelerator, and the graphics functions provided by the Host operating system. Although these functions are not portable, using the MGA is suitable for graphic-intensive applications, because of the increase in performance.

Available graphics functions

With the Genesis Native Library, you can:

- Draw rectangles, using *imGraRect()* or *imGraRectFill()*.
- Draw arcs, using *imGraArc()* or *imGraArcFill()*.
- Draw lines, using *imGraLine()*.
- Plot a series of points, using *imGraPlot()*.
- Write text, using *imGraText()*.



You can also fill an object, using *imGraFill()*.

Generating graphics

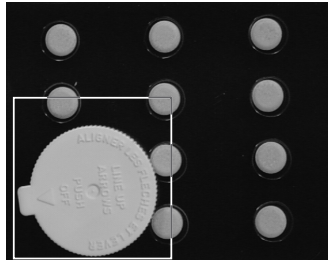
When you generate graphics, you have to specify the drawing color using the `IM_GRA_COLOR` field. The drawing color is the color that is used to draw, plot, write, or fill.

Graphics and color buffers

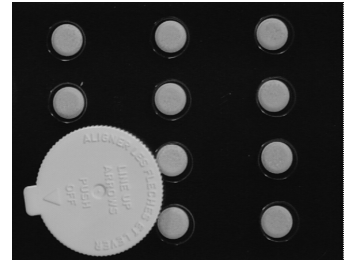
If you are generating graphics into a multi-band buffer, the same drawing color is normally used in all bands. However, for 8-bit buffers with two to four bands, you can specify a different drawing color for each band, using the `IM_GRA_COLOR_MODE` field. In this case, the least-significant byte of the specified color is used for the first band, the next byte is used for the second band, etc.

XOR option

Certain graphics functions give you the option of generating graphics in the drawing color, or in the colors that result from performing an XOR between this color and the pixels of the destination buffer. The latter option is available so that you can later remove the graphic by calling the function again.



A rectangle is drawn using `imGraRect()`, with the XOR option set. The drawing color is `0xFFFFFFFF`.



The rectangle can be removed by again calling `imGraRect()` with the XOR option (other options and parameters should also be the same as they were in the first call).

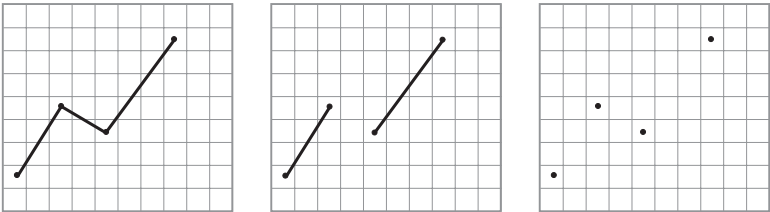
Note that, when using the XOR option, the graphic will usually be most visible if the drawing color is set to `0xFFFFFFFF`.

To use the XOR option, set the `IM_GRA_DRAW_MODE` field.

Plotting

You can plot a series of points using *imGraPlot()*. This can be used, for example, to draw the outline of some object or to plot a histogram.

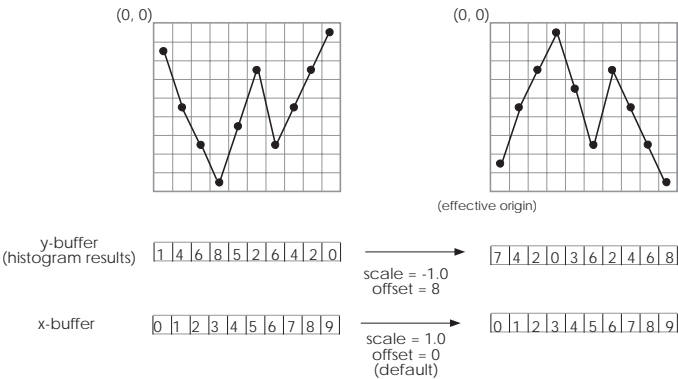
When you use *imGraPlot()*, you need to specify a one-dimensional buffer containing the x-coordinates of the points and another containing the y-coordinates of the points. *imGraPlot()* can connect all the points with a single line, connect each pair of points with a line, or simply draw a dot at each point.



Note that *imGraPlot()* plots a series of lines much faster than separate calls to *imGraLine()*.

Scaling and offsetting points

When you use *imGraPlot()*, you can scale and offset the x and y points by specified factors. This could be used, for example, to plot a histogram with the histogram origin ((0, 0)) at the bottom left of a buffer. To do so, specify a y scale factor of -1.0 and a y offset equal to: the buffer height - 1 (see below).



Plotting a histogram

The following code generates the histogram of an image, and then plots it right-side up (that is, with its origin at the bottom left of a buffer). The buffer in which to plot has dimensions SizeX by SizeY. The y scale ensures that histogram results will fit into this buffer, while the x scale ensures that the entire width of this buffer is used.

Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```

long XBuf;                /* Buffer with X values of points */
long YBuf;                /* Buffer with Y values of points */
long MaxVal;              /* Maximum value in histogram */
double Coef[2] = {0.0, 1.0}; /* Coefficients for ramp */

/* Allocate buffers for the X and Y values to plot */
imBufAlloc1d(Thread, 256, IM_LONG, IM_PROC, &XBuf);
imBufAlloc1d(Thread, 256, IM_LONG, IM_PROC, &YBuf);

/* Y points are the histogram results */
imIntHistogram(Thread, SrcBuf, YBuf, IM_DEFAULT, 0);

/* X points are just sequential numbers */
imGen1d(Thread, XBuf, IM_POLYNOMIAL, 0, 255, 2, Coef, 0);

/* Find the maximum value in the histogram */
imIntFindExtreme(Thread, YBuf, YBuf, IM_MAX_PIXEL, 0);
imBufGetField(Thread, YBuf, IM_RES_MAX_PIXEL, &MaxVal);

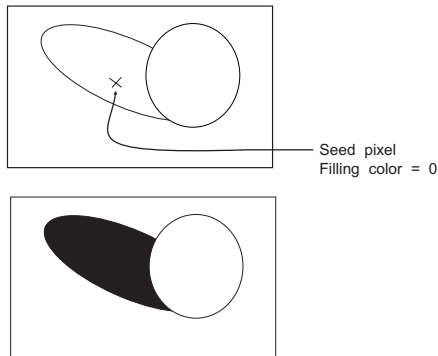
/* Scale plot to fit the image (use XBuf as the control buffer) */
imBufPutField(Thread, XBuf, IM_GRA_SCALE_Y, (double) -(SizeY - 1) / MaxVal);
imBufPutField(Thread, XBuf, IM_GRA_OFFSET_Y, SizeY - 1);
imBufPutField(Thread, XBuf, IM_GRA_SCALE_X, (double) SizeX / 256);
imBufPutField(Thread, XBuf, IM_GRA_COLOR, 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");

```

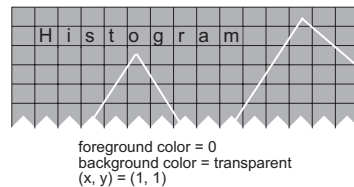
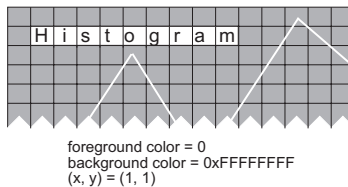
Filling

The *imGraFill()* function fills a connected region with a specified color. A connected region is an area of touching pixels that have the same value (horizontally and vertically adjacent pixels are considered touching; diagonally adjacent pixels are not). To specify the region to fill, you have to specify the position of a pixel within the region (called a seed pixel).



Writing text

When using *imGraText()*, you need to specify the color with which to write the text (called the foreground color), as well as the x and y coordinates at which to start writing (the text is written from the top-left corner of these coordinates). You also need to specify the background against which to write the text. The background can be a specific color or can be transparent (in which case only the strokes of the characters appear).



When writing text, you can use a small, medium, or large version of the default font; you can also scale the size of characters in the x and y directions by specified factors.

Chapter 9: Buffers and buffer fields

This chapter discusses buffers. It shows you how to allocate a buffer, how to use control buffers, child buffers, and tag buffers, and how to copy buffer data.

Data buffers

The processing functions of the Genesis Native Library operate on and store results in buffers. Before processing a buffer, you must allocate it. When allocating a buffer, you must give it a certain width and height, a certain number of bands, and a specific data type. You can allocate buffers on-board (in processing or display memory) or on the Host.

Fields

A buffer can contain control fields. Control fields are used because some functions have so many options that it is more practical to store these options in one place (as control fields in a buffer) than to have these options as parameters of the function. When a buffer is used for this purpose, it is referred to as a *control buffer*.

Child buffers

A child buffer refers to a rectangular region within a buffer or to a specific band of a multi-band buffer. You can therefore allocate a child buffer to restrict processing to a specific region or to a specific band of a buffer.

Copying buffer data

You can copy data between buffers. This is an efficient operation no matter where the source and destination buffers are located.

Tag buffers

When you copy or grab data, you can avoid overwriting regions of the destination buffer by using a tag buffer. A tag buffer specifies which pixels of the destination buffer to leave as they are, and which pixels to overwrite.

Transferring buffer data

You can transfer data between a buffer and an array in Host memory, or between a buffer and a file.

Mapping a buffer

You can use *imBufMap()* to map a buffer into Host memory. This allows you to access a buffer from the Host.

Creating a buffer

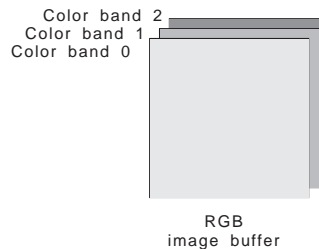
You can use *imBufCreate()* to create a buffer out of memory that has already been allocated. Among other things, this allows you to operate on memory that was not allocated by Genesis (such as memory on another board).

Allocating buffers

You must allocate a buffer before you can use it in a function call. Allocate a buffer using the *imBufAlloc()*, *imBufAlloc2d()*, or *imBufAlloc1d()* function. *imBufAlloc1d()* allocates a one-band buffer with a certain width; *imBufAlloc2d()* allocates a one-band buffer with a certain width and height; while *imBufAlloc()* allocates a buffer with a certain width and/or height, and with one or more bands. (*imBufAlloc()* can therefore be used to allocate any type of buffer; the other functions are provided for ease of use). Once you finish using a buffer, you should free the memory allocated to it, using *imBufFree()*.

Buffers are quite general purpose and can hold a variety of data. For example, image LUTs, convolution kernels, and histogram results are all stored in buffers of the appropriate size and data type.

Note that some processing functions will not operate directly on multi-band buffers. However, you can process a specific component of a color image.



When you allocate a buffer, in addition to specifying its width, height, and number of bands, you must specify:

- Its data type.
- The memory bank in which to allocate the buffer.

Data type

Supported data types are: 1-bit packed binary; 8-, 16-, and 32-bit integer (signed and unsigned); 24-bit packed RGB; and 32- and 64-bit IEEE floating-point. Not all functions can operate on all data types. For example, the functions of the integer processing module can generally operate only on integer-type buffers. When a function does not support a certain data type, you can use *imBinConvert()*, *imFloatConvert()*, or *imIntConvert()* to convert the source buffer to the required type.

Memory location

When you call *imBufAlloc...()*, you can allocate the buffer on-board (in processing or display memory) or on the Host. If you allocate the buffer on-board, it will reside on the same node as the thread which executed the buffer allocation function.

Buffers of a processing function

The source buffers of a processing function carried out by the 'C80 or NOA must be located in processing memory. In addition, they must reside on the same node as the thread which will execute the processing function. That is, these buffers must reside in local processing memory. If these conditions are not met, the results of the processing function will be undefined because you can not read from a non-local buffer. In other words, if you use a non-local buffer as a source buffer in a processing operation, the operation might appear to work, but the results will not be reliable.

The destination buffer of a processing function can be located anywhere, except when the operation uses the NOA. However, for maximum efficiency, it should also be in local processing memory. Therefore, if you want to display the results of a processing function, you should write the results to a buffer in processing memory and then copy that buffer to a display buffer, rather than using a display buffer as the destination of the processing function.

- ❖ If the processing function is iterative, the destination buffer must be in local processing memory because an iterative function also reads from the destination buffer.

When the operation uses the NOA, not only is it inefficient to process buffers that are not in local processing memory (buffers in display memory, Host memory, or processing memory on another node), their use is prohibited.

When a processing operation is carried out by the NOA, the destination buffer must be in local processing memory. This restriction comes about because the NOA can only access buffers that are located on the same node as the thread which will execute the processing function.

The buffer location restrictions discussed above do not apply to the buffer copying functions (*imBufCopy()*, *imBufCopyVM()*, and *imBufCopyPCI()*). In addition, you can use non-processing functions, such as *imBufInquire()* or *imBufGetField()*, on any buffer, regardless of its location.

Note that if you cannot send a processing function to a thread that resides on the same node as the source and destination buffers, you should first copy the buffers to the appropriate node (using *imBufCopy()*).

Control buffers

A control buffer refers to a buffer whose control fields are used to 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.

Fields

Each control field (or simply "field") holds a single value (integer or floating-point). A field is identified by a unique "tag". The tag itself is just an integer value.

Most fields have predefined tags, but you can add your own fields as long as you use tags that don't conflict with those used by the Native Library.

Managing control buffers

A control buffer can store fields that apply to many different functions because a function will only use those fields that apply to it. In addition, you can use any type of buffer as a control buffer. You can even use the buffer that a function is processing as the function's control buffer. Therefore, when you need a control buffer, you can avoid allocating a new buffer by using a previously allocated buffer.

If you do need buffers solely to be used as control buffers, you should allocate these buffers with *imBufAllocControl()*. This function allocates a buffer that can only hold fields, not data, so it can save you memory.

Regardless of which buffers you use as control buffers, you should always set up your control buffers well before you use them, so that, for example, you are not adding fields to a buffer within a time-critical loop (this can be a significant overhead).

Managing fields

You add a new field to a buffer or modify an existing field using *imBufPutField()*. In addition, you remove a field from a buffer using *imBufRemoveField()* and copy fields between buffers using *imBufCopyField()*. You can read the value of a field using *imBufGetField()*, *imBufGetFieldDouble()*, or *imBufGetNextField()*.

Defaults

Most fields have default values, so if a field is not added to a control buffer or if a control buffer is not passed to the function, the function will use these default values. Fields that do not have default values only affect the function if you add them to the control buffer.

Reading results

Some processing functions (specifically those that produce a single value as a result, rather than a whole image) write their result to a field in the destination buffer. You read back the result using *imBufGetField()*.

Child buffers

A child buffer refers to a specified rectangular region of interest within a buffer or to a specific band of a multi-band buffer. The buffer from which a child buffer is created is called its parent buffer.

A child buffer is considered a data buffer in its own right and can be used in the same way as any other buffer. However, no new memory is allocated to a child buffer. Therefore, any changes made to a parent buffer also affects its child buffer(s), and vice-versa.

Child buffers are useful when you want to restrict processing to a portion of a buffer, or when you want to copy/grab into a portion of a buffer.

Child buffers are especially important with display buffers since you usually want to know exactly where on the display the buffer is allocated. In fact, you cannot display an image at all until you have made a child display buffer to hold a copy of the image you want to see.

Allocating child buffers

Allocate child buffers using *imBufChild()* (this allocates a child buffer from a region of a buffer) or *imBufChildBand()* (this allocates a child buffer from a band of a multi-band buffer). You can move and/or resize child buffers, allocated with *imBufChild()*, using the *imBufChildMove()* function. Note that it is more efficient to move an existing child buffer than to free it and allocate a new one.

Copying buffer data

The Genesis hardware allows data to be copied efficiently between buffers, no matter where they are located. You can use one of the following functions:

- *imBufCopy()*. This function uses whatever hardware can perform the copy the fastest.
- *imBufCopyPCI()*. This function uses the PCI bus.
- *imBufCopyVM()*. This function uses the VMChannel.

Note that the above functions copy only image data, and not any buffer fields that might be present. To copy a buffer's fields, use *imBufCopyField()*.

Formatting options

Both *imBufCopyPCI()* and *imBufCopyVM()* can perform a variety of formatting operations on the data as it is copied (such as zooming and packing). However, these functions depend on the physical path involved, so you need to understand your system's architecture to exploit their capabilities. See the section *Using the advanced copy functions*.

More on *imBufCopy()*

When you use *imBufCopy()*, you don't have to worry about the physical path involved because the function will use whatever path is available. *imBufCopy()* is a very important part of the library because it performs the following common functions:

- Copy an image from processing to display memory.
- Copy an image from an on-board buffer to a buffer in Host memory in the fastest possible way (and without tying up the Host CPU during the transfer).
- Copy an image from one processing node to another in a multi-processing system.

In addition to the above, *imBufCopy()* automatically copies all bands of a multi-band buffer, if the source and destination buffers have the same number of bands. If the source buffer is 1-band and the destination is a 3-band display buffer, *imBufCopy()* replicates the source buffer in all 3 bands. This allows you to display a 1-band (grayscale) image when the display is in color mode.

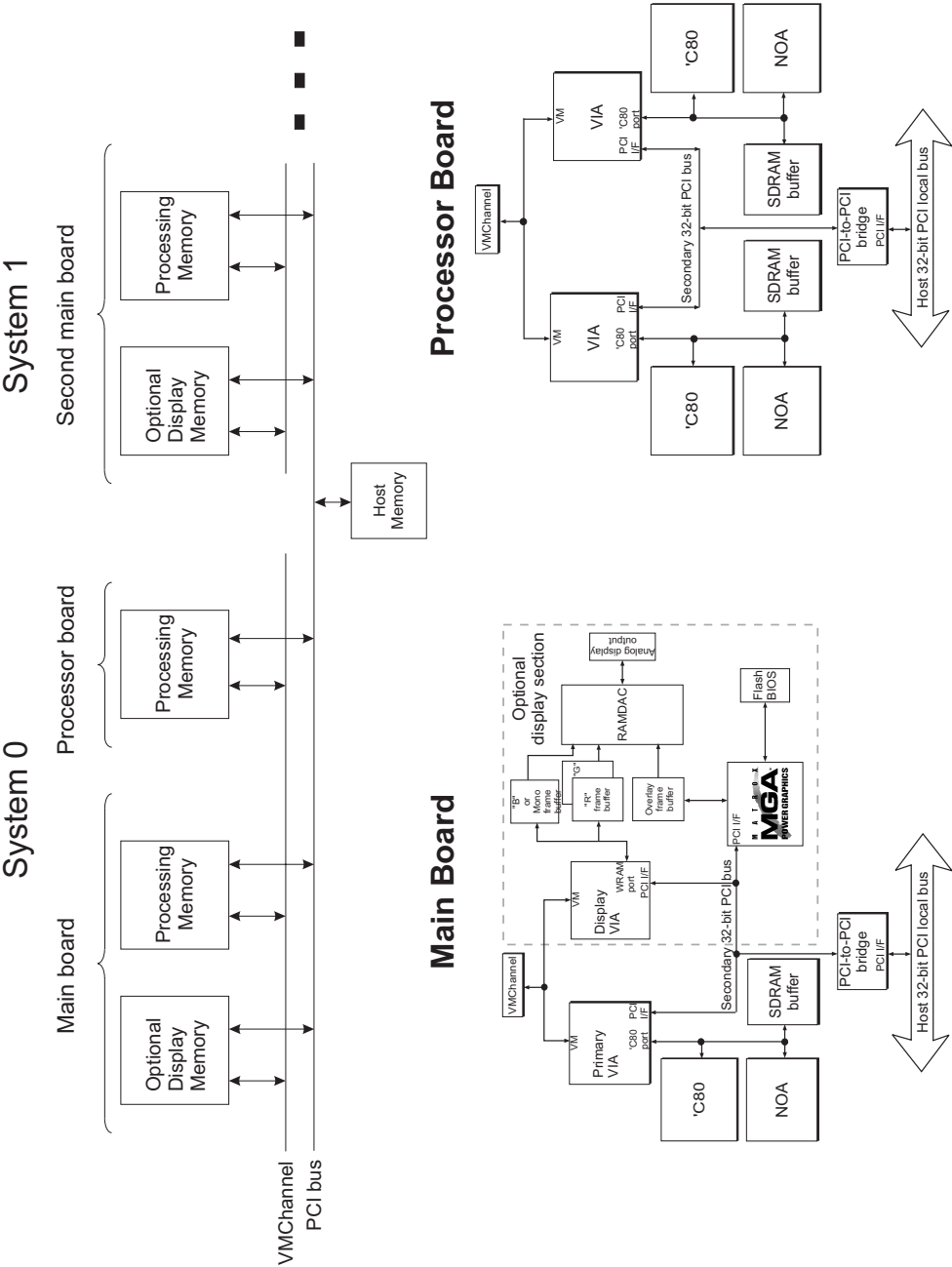
Using the advanced copy functions

This section discusses some of the formatting options available to *imBufCopyPCI()* and *imBufCopyVM()*. Note that, when data is copied over the VMChannel, two (or more) VIAs are involved. When data is copied over the PCI bus, only one VIA is involved. If you use *imBufCopyVM()*, some options only apply to the transmitting VIA, and some only apply to the receiving VIA. If you use *imBufCopyPCI()*, some options only apply if the PCI bus is the source of the data (that is, if the VIA is reading data from the PCI bus and writing it to memory), and some only apply if the PCI bus is the destination (that is, if the VIA is reading data from memory and writing it to the PCI bus).

For a complete list of the options available to *imBufCopyPCI()* and *imBufCopyVM()*, including when these options apply, see the *Genesis Native Library Command Reference*.

If you use *imBufCopyPCI()* or *imBufCopyVM()*, you should be familiar enough with the system architecture to ensure that the copy can actually be performed over the specified path. For example, if you use *imBufCopyPCI()*, you should be sure that the source and destination buffers of the copy are actually connected by the PCI bus. If they are not, the copy will not be performed. You should also be familiar enough with the system architecture to know which is the transmitting VIA, which is the receiving VIA, and whether the PCI bus is the source or destination of the data.

The diagram on the next page indicates the path(s) by which various memory banks are connected. This can be used to determine how two buffers are connected, which is the receiving VIA, etc. Note, for example, that the VMChannel cannot be used to copy between different systems or to the Host. However, it can be used to copy between different boards in the same system.



* For ease of viewing, the grab module and grab ports are not shown.

Tag buffers

When you copy data, you can avoid overwriting regions of the destination buffer by using a tag buffer. A tag buffer specifies which pixels of the destination buffer to leave as they are, and which pixels to overwrite. This is useful when you want to write into a non-rectangular region of a buffer (to write into a rectangular region of a buffer, you could always allocate a child buffer and then copy into that child buffer). Note that using a tag buffer will not affect the speed at which the copy function is performed.

How tag buffers work

When you supply a tag buffer to a copy function (using the `IM_CTL_TAG_BUF` field), a pixel of the destination buffer is overwritten only if its corresponding pixel in the tag buffer has the value 0; if its corresponding pixel has the value 1, it is not overwritten.

Tag buffer requirements

A tag buffer must be of type binary, have the same size as the destination buffer, and be in the same memory bank as the destination buffer. It must have either a single band or the same number of bands as the destination buffer. If the tag buffer is single band and the destination buffer is multi-band, the tag buffer will be applied to each band of the destination buffer.

In addition to the above, the first pixel in the tag buffer must be byte-aligned. Binary buffers are byte-aligned when allocated, so the only way to violate this restriction is to use a child buffer of a binary buffer, with an incorrect alignment. If you use a child buffer as your tag buffer, the *Xstart* parameter of *imBufChild()* should be a multiple of 8 when you create the child buffer.

❖ The byte-alignment restriction does not apply to tag buffers that are used with *imBufPack()*.

Creating tag buffers

In general, you create a tag buffer by processing an integer buffer and then converting that buffer to binary (using *imBinConvert()*). Depending on the region of the destination buffer that you want to protect, there are various processing functions you can use. For example, you can use the graphic functions to draw a required pattern. Note that certain processing functions can be performed directly on binary buffers, so you might be able to avoid processing integer buffers (which would be more efficient).

Zooming and subsampling

You can subsample or zoom the copied data (using the `IM_CTL_SUBSAMP_X`, `IM_CTL_SUBSAMP_Y`, `IM_CTL_ZOOM_X`, and `IM_CTL_ZOOM_Y` fields). Zooming in the x direction replicates each column *n* times; zooming in the y direction replicates each row *n* times. Subsampling in the x direction causes only every *n*th column to be written to the destination buffer (starting with the first column); subsampling in the y direction causes only every *n*th row to be written to the destination buffer (starting with the first row).

Extracting bytes

When the pixel depth is more than 8 bits, you can choose to copy only 8 contiguous bits to the destination buffer (using the `IM_CTL_BYTE_EXT` field). This is mainly useful when you are copying to the display, since the display is limited to 8 bits per pixel. Note that the most significant 8 bits are usually copied.

Swapping bytes

Sometimes, packed color images are stored as BGR instead of RGB (or BGRa instead of RGBa). You can swap the 1st and 3rd bytes of such images (using the `IM_CTL_BYTE_SWAP` field). The packed images must be in 24-bit or 32-bit format. Note that if the destination buffer is 3x8-bit, the copy function will automatically separate the packed color image. Therefore, packed BGR or BGRa images can be correctly displayed by swapping bytes while copying to a 3x8-bit display buffer.

Note that the "a" in BGRa refers to any extraneous data that the packed BGR image might contain.

Reversing the direction of the copy

You can copy the data right to left instead of left to right, and bottom to top instead of top to bottom (using the `IM_CTL_DIR_X` and `IM_CTL_DIR_Y` fields). This can be useful if your optical set-up has delivered images that are horizontally and/or vertically reversed (it could be that a mirror was involved or that parts passed under a line scanner from bottom to top).

Expanding RGB555 or 565 formats

With the Genesis Native Library, you can expand 16-bit color images (either in RGB555 or RGB565 format) to 3x8-bit (using the `IM_CTL_FMTCVR` field). If you want to display such images in color, you must use this option when copying to the display (the display is limited to 8 bits per pixel per band).

Write masks

A write mask can be used to protect certain bit planes of the destination buffer (using the `IM_CTL_WRTMSK` field). This option can only be used if you are copying to the display. It can be useful, for example, if you want to prevent annotation in certain bit planes from being overwritten. Note, however, that Matrox Genesis contains an overlay buffer, so you would normally not annotate the main display buffer. Instead, you would draw in the overlay buffer and enable keying (see Chapter 11 for details).

If you do use a write mask, you must specify the required value using 24 bits. The low 8 bits of this value applies to the red buffer, the next 8 bits to the green buffer, and the high 8 bits to the blue buffer.

Reducing overhead

In order to reduce overhead, you can skip programming of most VIA registers when you perform the copy (use the `IM_CTL_SETUP` field). This is safe if the only difference from the previous copy is the address of the source and/or destination buffer. This is typically the case in a real-time, double-buffered application, where two (or more) buffers are used over and over again, and all other copy parameters stay the same.

Note that, if any copy parameter (except the address of the source or destination buffer) is different, you should not skip programming of any VIA register.

Specifying a VIA

This option only applies to *imBufCopyPCI()*.

When a copy is performed over the PCI bus, only a single VIA is involved. If the source and destination buffers are both on-board, this VIA could either be the one local to the source buffer or the one local to the destination buffer. You can specify which VIA using the `IM_CTL_VIA` field. This can be useful if you are performing several transfers in parallel since, for the transfers to actually occur in parallel, you might need to ensure that specific VIAs be used. Unless you are performing transfers in parallel, however, it normally does not matter which VIA is used.

Writing a rectangular region

This option only applies to *imBufCopyVM()*.

With the Genesis Native Library, you can choose to write a rectangular subset of the source buffer to the destination buffer. You would normally do this by allocating a child buffer from the source buffer, and then copying that child buffer. However, there might be cases when you don't want all of the transmitted data written to the destination buffer. For example, if you have multiple nodes and want each to process a different part of an image, you can use one VIA to transmit the image and several others to receive just a portion of the image.

To define the region to write, use the `IM_CTL_START_X`, `IM_CTL_START_Y`, `IM_CTL_STOP_X`, and `IM_CTL_STOP_Y` fields.

Avoiding display artifacts

This option only applies to *imBufCopyVM()*.

When copying to the display, you can force the copy to be synchronized with updates of the display, in order to avoid visible artifacts such as screen tearing. To do so, use the `IM_CTL_DISPLAY_SYNC` field. Note that, by default, the copy is performed as soon as possible. If you set the `IM_CTL_DISPLAY_SYNC` field, the copy will be delayed, if necessary, in order to avoid visible artifacts.

Continuous copying

This option only applies to *imBufCopyVM()*.

With the Genesis Native Library, you can have the source buffer continuously copied to the destination buffer (using the `IM_CTL_COUNT` field), until you call *imThrHalt()*.

This option could be used to maintain a continuous display of a processing buffer. However, this would be very inefficient, because the buffer would be copied much more frequently than necessary (therefore wasting memory bandwidth) and the VIAs involved in the copy would be prevented from doing any other copies (which could interfere with other applications). It is much better to copy the buffer only when it changes.

Copying to/from the VMChannel

This option only applies to *imBufCopyVM()*.

With the Genesis Native Library, you can copy data to/from a specified VM stream. This is useful when you have multiple nodes and want to broadcast the same image to these nodes. You specify the VM stream using the IM_CTL_STREAM_ID field.

If you are receiving data from an interlaced VM stream, you can specify that the data be written using progressive scanning (set the IM_CTL_ADDR_MODE field to IM_PROGRESSIVE). This can be useful if you are receiving just one field from the stream and want the lines written sequentially into the destination buffer.

❖ You can write from two VM streams to two buffers simultaneously, by using *imDigGrab()* to perform the other transfer.

External VM devices

The IM_CTL_STREAM_ID field can also be used if you have an external (non-Matrox) VM device attached to the VMChannel and want to transfer data to or receive data from this device. You must know the VM stream ID being used by the device and pass this ID to the IM_CTL_STREAM_ID field. Any other formatting options that are specified (such as subsampling and zooming) will still be performed.

Between a buffer
and an array

Transferring buffer data to the Host

You can transfer data from an array in Host memory to a buffer using the *imBufPut()*, *imBufPut1d()*, or *imBufPut2d()* functions. *imBufPut()* writes data to the entire buffer, while *imBufPut1d()* and *imBufPut2d()* write data to a specified block of the buffer. If the buffer has more than one band, they are all written into, one after another. The array should be large enough to fill the buffer and be of the same data type as the buffer.

You can transfer data from a buffer to an array in Host memory using the *imBufGet()*, *imBufGet1d()*, or *imBufGet2d()* functions. *imBufGet()* reads data from the entire buffer, while *imBufGet1d()* and *imBufGet2d()* read data from a specified block of the buffer. If the buffer has more than one band, each band is read, one after another. The array should be large enough to hold the data and be of the same data type as the buffer.

❖ Note the differences between *imBufPut/Get()* and *imBufCopy()*. *imBufCopy()* is the fastest way to transfer data between Host and on-board memory because the VIA drives the transfer without involving the Host CPU. However, *imBufCopy()* can only work with Host buffers allocated with *imBufAlloc...()*. On the other hand, *imBufPut/Get()* can work with any type of Host memory, but are somewhat slower than *imBufCopy()* and tie up the Host CPU (since the Host CPU drives the transfer).

Between a buffer
and a file

You can transfer data from a file to a buffer using the *imBufLoad()* or *imBufRestore()* function. *imBufLoad()* writes data into an existing buffer (the buffer should be large enough to hold the data, and have the same number of bits per pixel as the file); *imBufRestore()* writes data into an automatically allocated buffer.

You can transfer data from a buffer to a file using the *imBufSave()* function.

Mapping a buffer

With the Genesis Native Library, you can map a buffer into Host memory, using *imBufMap()*. This gives you a pointer to the buffer data so that you can access it directly from the Host (see the example below).

In addition to the pointer to the buffer, *imBufMap()* returns:

- The buffer pitch. The pitch is needed to access a specific line of a two-dimensional buffer. For example, the address of a pixel 3 lines down from "addr" would be: (addr + 3*pitch). Note that a buffer's pitch is not necessarily the same as its width in bytes (especially when the buffer is a child buffer).
- The number of consecutive lines mapped. This is equal to: (# of lines in the buffer) - (# of the first line mapped).

An example

The following code draws the results of a histogram into a buffer, from the Host. It performs a histogram on an image, maps the buffer in which to draw into Host memory, and then draws into this buffer from the Host.

Note that this code is part of the *process.c* program and requires only the basic Genesis hardware. See Appendix B for the complete *process.c* program.

```
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 */
unsigned char *Pointer; /* Pointer for direct access to buffer */

/* Allocate histogram result buffer */
imBufAllocId(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);
```

```
/* 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 */
imSyncHost(Thread, 0, IM_COMPLETED);

/* Draw the histogram directly into a 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;
}
```

Creating a buffer from memory already allocated

You can use *imBufCreate()* to create a buffer out of memory that has already been allocated. This memory can be:

- From one or more existing Genesis buffers. Creating a buffer out of Genesis memory can be used to support multiple live grabs on the display (see below for details).
- Contiguous physical memory. Creating a buffer out of contiguous memory is primarily useful when you need to copy a buffer to memory on another board.
- Virtual memory (for example, memory allocated with *malloc()*). Creating a buffer out of virtual memory allows you to use the buffer for DMA transfers. You must lock the buffer in physical memory (using *imBufControl()*) before attempting to copy it. The new buffer can only be used by *imBufCopy()* or *imBufCopyPCI()*, not by any other function.

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

Multiple live grabs on the display

When grabbing from synchronized monochrome cameras (or from a color camera), you can view the input from each monochrome camera (or view each color band) in a separate display buffer. To do so, allocate a child buffer on the display, using *imBufChild()*, for each monochrome camera or color band. Then, use *imBufCreate()* to create a multi-band buffer from these child buffers. When you grab using the ID of the created buffer, the data from each monochrome camera or color band will be displayed in a separate buffer.

Chapter 10: Grabbing images

This chapter discusses how to grab images, and other related topics.

Grabbing

To grab an image with the Genesis Native Library, you must first allocate a camera definition that matches your camera type, using *imCamAlloc()*. If you have more than one digitizer in your system, you must also allocate the digitizer with which to grab, using *imDigAlloc()*. You then pass the camera definition identifier and if required, the digitizer identifier, to the grab command, *imDigGrab()*.

Note that the buffer in which to grab can be located anywhere in your system (in processing or display memory on any node).

Controlling the grab

There are several ways you can control how an image is grabbed. You can:

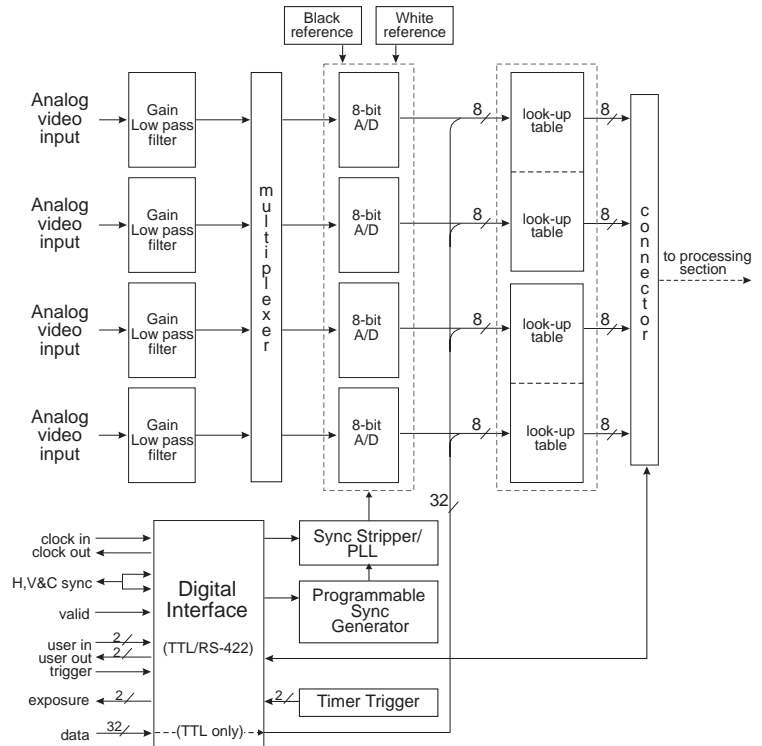
- Specify a number of options (such as zooming and subsampling) through the control buffer passed to *imDigGrab()*. These options are performed by the VIA, as it is the VIA that takes data from the grab port and writes it to memory. These options are therefore independent of the digitizer or the camera type.
- Change the settings of a particular camera definition, using *imCamControl()*.
- Program the digitizer directly, using *imDigControl()*.

imCamControl() vs. *imDigControl()*

imCamControl() and *imDigControl()* basically produce the same results. However, with *imCamControl()*, the digitizer is programmed to a specific camera definition only when a grab is issued with the identifier of that camera definition; *imDigControl()* programs the digitizer directly. Therefore, you should avoid using *imDigControl()* if you want to share the digitizer between several applications.

The grab module

The grab module of the Genesis offers flexible, high-resolution, high-speed acquisition. It features four analog video input channels, for standard or non-standard video, four 8-bit analog to digital converters, a look-up table (LUT) for each channel, and a 32-bit digital interface (TTL/RS-422).



For more details on the grab module, including sampling rates, see the *Genesis Installation and Hardware Reference*.

VIA options of the grab command

When you call *imDigGrab()*, you can specify a number of options through the control buffer passed to this function. All these options are independent of the digitizer section or the camera type; they are carried out by the VIA local to the destination buffer of the grab. Therefore, if you are grabbing to two or more buffers at the same time, you can specify different options for each grab.

Most of the options available to *imDigGrab()* are also available to *imBufCopyVM()* and/or *imBufCopyPCI()*. These common options are:

- Tag buffers.
- Zooming and subsampling.
- Byte extraction.
- Reversing the direction of the grab.
- Write masks.
- Reducing overhead.
- Grabbing a rectangular region.
- Using VM streams.

The above options were discussed in Chapter 9. This section discusses some of these options in relation to *imDigGrab()*, and some of the options that are unique to *imDigGrab()*.

For a complete list of the options available to *imDigGrab()*, see the *Genesis Native Library Command Reference*.

Number of iterations

With the Genesis Native Library, you can grab a specific number of frames, fields, or lines; use the `IM_CTL_COUNT_MODE` field to specify which one. Alternatively, you can continuously grab frames until you call *imThrHalt()*.

Note that, to grab fields, your camera must use interlaced scanning. In that case, you can start grabbing on the next odd field, the next even field, or the very next field (use the `IM_CTL_START_FIELD` field). In addition, you can specify that the data be written using progressive scanning (set the `IM_CTL_ADDR_MODE` field to `IM_PROGRESSIVE`). Progressive scanning can be useful when you want to grab just one field from the camera and you want the lines written sequentially into the destination buffer.

Note that grabbing just one field from an interlaced camera allows you to get a low-resolution image in half the time it would take to grab the whole frame. This can be an important optimization, as long as you do not need the full resolution.

Synchronizing multiple grabs

If you want to grab the same frame to two or more buffers at the same time, you have to set the `IM_CTL_CAPTURE_MODE` field to `IM_SYNCHRONIZED` on each grab; see the *Grabbing to two or more buffers* section for details.

Grabbing a rectangular region

As with the copy functions, you can grab a rectangular region of the image, rather than the entire image. This saves unnecessary memory accesses. Note that this is most commonly used when you have multiple nodes and want each node to grab a different part of the same input frame.

To define the rectangular region, use the `IM_CTL_START_X`, `IM_CTL_START_Y`, `IM_CTL_STOP_X`, and `IM_CTL_STOP_Y` fields.

Grabbing a VM stream

You can use *imDigGrab()* to grab from a VM stream, instead of grabbing from a camera. When you do, all options specified through the control buffer (such as subsampling and zooming) will still be performed. However, the camera settings and the digitizer LUT will not have an effect, since the data is grabbed only through the VIA and does not pass through the grab module.

To specify the VM stream, use the `IM_CTL_STREAM_ID` field.

Reversing the direction of the grab

Reversing the direction of the grab can be particularly useful if your images are reversed and you are grabbing to the display. Without this option, you would need to grab the image to processing memory, process it to correct the problem, and then copy it to the display.

To reverse the direction of the grab, use the `IM_CTL_DIR_X` and `IM_CTL_DIR_Y` fields.

Grab mode

You can execute *imDigGrab()* synchronously or asynchronously. In synchronous mode (the default), the thread to which *imDigGrab()* is sent behaves like any other thread, in that it waits for the grab to complete before continuing to execute. In asynchronous mode, the thread will not wait for the grab to complete before continuing to execute. Asynchronous mode is primarily useful when real-time processing (see Chapter 4 for details).

To specify the grab mode, use the `IM_CTL_GRAB_MODE` field.

Reducing overhead

As with the copy functions, you can skip programming of most VIA registers, in order to reduce overhead. When you grab, you can also skip programming of the grab module, to further reduce overhead. This is safe if you are using the same camera as the previous grab and have not changed any camera parameters with *imCamControl()*.

To specify the necessary setup, use the `IM_CTL_SETUP` field.

Line interrupts

With the Genesis Native Library, you can enable line interrupts during a grab. Line interrupts allow the 'C80 (or the Host in the case of the Genesis-LC) to keep track of when each line of a frame is written to memory. This can be useful when another function needs to wait on the grab command, but only needs to wait for a specific line, rather than the whole frame or field.

To enable line interrupts, use the IM_CTL_LINE field. You can request a single interrupt after a specified line has been grabbed, or you can request continuous interrupts. In the latter case, interrupts will normally be produced as fast as the 'C80 (or the Host) can handle them (after every line, or after every few lines if the line rate is too high). However, when you request continuous interrupts, you can force a specific interval between the interrupts, using the IM_CTL_LINE_INT_STEP field.

Synchronizing with a line

When you enable line interrupts, you must pass an operation status block (OSB) to *imDigGrab()*. The OSB will be updated each time there is an interrupt. You can then wait for a specific line to be grabbed by calling *imSyncThread()* or *imSyncHost()*, as follows

```
imSyncThread(Thread, OSB, IM_LINE_INT + n); /* similarly for imSyncHost() */
```

where *n* specifies the line for which you are waiting. You can also inquire about the current grab line, as follows

```
Line = imSyncHost(Thread, OSB, IM_LINE_INT + IM_INQUIRE);
```

Note that, when you wait for a specific line to be grabbed, the thread (or Host) will be blocked until the grab line count is equal to or greater than the specified line. Since the interrupt will not occur until the end of that line, the data will already be in memory when the thread (or Host) becomes un-blocked.

Interlaced cameras

If your camera uses interlaced scanning and you request a single line interrupt, it normally occurs during the second field of the frame (or the first field if only a single field is grabbed). To receive the interrupt during the first field or during both fields, use the IM_CTL_LINE_INT_FIELD control field.

Grabbing to two or more buffers

With the Genesis Native Library, you can grab to two or more buffers at the same time. The buffers must be in different memory banks, that is, in different nodes or in processing and display memory on the same node.

Grabbing to two or more buffers at the same time can be useful when you want to grab to more than one node in your system, or when you want to simultaneously grab to processing and display memory. Each grab command must be sent to a different thread and must use compatible camera definitions (camera definitions that do not force the digitizer to be re-programmed between grab commands). Note that, if the grab commands were sent to the same thread, they would run sequentially and grab different frames. If the digitizer has to be re-programmed, the same frame cannot be grabbed.

Synchronized capture

To ensure that exactly the same frame is grabbed to two or more buffers at the same time, you must request a synchronized capture, by setting the `IM_CTL_CAPTURE_MODE` field of *imDigGrab()* to `IM_SYNCHRONIZED`. You then explicitly enable the capture, using *imDigCapture()*, when you know that all grabs are ready. For example, the following grabs to two nodes at the same time. A different thread is used for each grab command, and a third thread is used to execute the synchronization commands (since the first two threads are blocked until their grabs have executed).

```
/* 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);
```

Note that the call to *imSyncHost()* ensures that the grab command sent to *Thread2* does not execute until its control buffer is set up. You could also ensure this by using two copies of the control buffer and setting each one in the same thread as it will later be used.

Synchronized capture through triggers

Another way to perform a synchronized capture is to change the camera definition so that it expects a software trigger, set up all the grabs, then either provide the software trigger or reselect the hardware trigger (if the camera normally uses a hardware trigger). However, this method requires different code for different types of cameras (triggered and non-triggered). The method shown in the above code will work regardless of the camera type.

Triggers are discussed in the *Camera settings* section.

Compatible camera definitions

Note that camera definitions do not necessarily have to be the same to be compatible. For example, they can have different input channels if the cameras are physically synchronized, but not if the cameras are unsynchronized. In addition, they can have different gain and reference levels (but not different timing parameters). See the *Camera settings* section for details on input channels, gain and reference levels, and timing parameters.

Different options

Although you must use compatible camera definitions for each grab, you can use different control buffers, since the options specified through the control buffer are performed by the VIA local to the destination buffer of the grab. Therefore, you could, for example, zoom data sent to display memory but not to processing memory.

Camera settings

Once a camera definition is allocated, you can change its settings (such as its channel number and reference levels), using *imCamControl()*. Note that using *imCamControl()* does not change the original camera definition file in the \GENESIS\DCF directory.

Changing camera settings

Using different settings allows different tasks to grab from the same camera, but using the settings appropriate to the task. If necessary, you can make a copy of the camera definition before changing it, using *imCamClone()*. This is useful when you want several identifiers for the same camera, each with different settings.

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.

❖ Even if you are performing a continuous grab, using *imCamControl()* will not affect the digitizer hardware.

Settings

The following sections discuss some of the settings you can change using *imCamControl()*. Note that you should only change a setting if you do not want to use the default value specified in the original camera definition file.

Some of these settings, and their associated descriptions, also apply to *imDigControl()*. For a complete list of settings available to *imCamControl()* and *imDigControl()*, see the *Genesis Native Library Command Reference*.

Input channel

The input channel is the camera channel from which to grab. Monochrome cameras require one channel, while color cameras require three. You specify the input channel(s) by setting the `IM_DIG_CHANNEL` field to the appropriate channel(s): `IM_CHANNEL_0`, `IM_CHANNEL_1`, `IM_CHANNEL_2`, or `IM_CHANNEL_3`. For example, to grab from a single channel, specify the channel number:

```
imCamControl(Thread, Camera, IM_DIG_CHANNEL, IM_CHANNEL_2);
```

To grab from three channels, specify the three channels:

```
imCamControl(Thread, Camera, IM_DIG_CHANNEL,  
              IM_CHANNEL_0+IM_CHANNEL_1+IM_CHANNEL_2);
```

You could also set `IM_DIG_CHANNEL` to `IM_CHANNEL(n)`. In other words, `IM_CTL_CHANNEL(3)` is equivalent to `IM_CTL_CHANNEL_3`. Using `IM_CHANNEL(n)` can be useful if, for example, you are looping through several channels and want the loop counter to specify the appropriate `#define`.

Note that, if you have only one camera connected to the default channel(s) specified in your original camera definition file, you don't need to set `IM_DIG_CHANNEL` before grabbing.

The following are some cases when you will need to set `IM_DIG_CHANNEL` before grabbing:

- When you are grabbing from a monochrome camera that is not connected to the default channel.
- When you are grabbing from a color camera that is not connected to the default channels.
- When you want to grab from multiple channels to different memory banks. For example, you might want to grab each band of a color camera to different memory banks, or grab from several synchronized monochrome cameras to different memory banks. You might also want to grab from several synchronized monochrome cameras to a multi-band buffer. Grabbing from multiple channels is discussed in the following sections.

❖ "Different memory banks" refers to buffers in different nodes or to processing and display memory in the same node.

Settings for each channel

If necessary, 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`:

```
imCamControl(Thread, Camera, IM_DIG_GAIN + IM_CHANNEL_1, 50.0);
```

Note that, when you change a setting without specifying a channel, the setting is changed on all channels currently selected for the camera. This helps to keep the application code independent of the camera type (color or monochrome). It also means you don't have to change settings on channels that you are not using (but which might be being used by another application running on the Genesis system at the same time).

Channel selection on `imDigGrab()`

Once you have selected the required channel(s) and specified their individual settings (if necessary), you are ready to grab. If you are grabbing from only some of the selected channels (perhaps because other nodes are grabbing from the other selected channels), you need to set the `IM_CTL_CHANNEL` field of `imDigGrab()` to specify which one(s); see the next section for examples. If you are grabbing from all the channels that you selected with `imCamControl()` (that is, the number of bands in your destination buffer matches the number of channels selected), you don't need to set `IM_CTL_CHANNEL`.

Grabbing RGB into a packed buffer

When the destination buffer has fewer bands than the number of available channels but a greater pixel depth than the grabbed data, several channels will be packed into each pixel. This allows you to, for example, grab RGB images into a single band 24- or 32-bit buffer. Each pixel will contain a packed RGB value (the fourth byte will be unused in the case of a 32-bit destination buffer).

Grabbing grayscale into a color buffer

When the destination buffer has several bands but only one channel is selected, the data will be replicated in all bands. This allows you to use a monochrome camera in a color application without modifying the color application.

Grabbing from multiple channels to a single memory bank

Each memory bank has just one VIA capable of writing grabbed data to it. Therefore, you cannot simultaneously grab the same frame to different buffers in the same memory bank. However, you can still grab from multiple synchronized cameras simultaneously, by using a single grab command and a multi-band destination buffer. For example, to grab from four monochrome cameras simultaneously:

```
/* Allocate a four-band buffer */
imBufAlloc(Thread, SizeX, SizeY, 4, IM_UBYTE, IM_PROC, &Buf);

/* Select all four channels */
imCamControl(Thread, Camera, IM_DIG_CHANNEL,
              IM_CHANNEL_0+IM_CHANNEL_1+IM_CHANNEL_2+IM_CHANNEL_3);

/* Grab from four channels simultaneously */
imDigGrab(Thread, 0, Camera, Buf, 1, 0, 0);
```

Grabbing from multiple channels to different memory banks

When you want to grab from multiple channels to different memory banks, you must issue a separate grab command for each memory bank involved (since each grab command only programs a single VIA). In addition, you must follow certain rules if you want to ensure that the same frame is grabbed to all nodes; these were discussed in the *Grabbing to two or more buffers* section.

An example

The following code grabs from three channels to different nodes. The three required channels are first selected using the `IM_DIG_CHANNEL` field of `imCamControl()` (this would be necessary if the channels are not the default channels of a color camera or if you are grabbing from several monochrome

cameras). The channel to grab to each node is then set using the `IM_CTL_CHANNEL` field of `imDigGrab()`. For clarity, the code does not include any synchronization to ensure that the same frame is grabbed.

```
imCamAlloc(Thread, NULL, IM_DEFAULT, &Camera);

/* Select all three channels together */
imCamControl(Thread, Camera, IM_DIG_CHANNEL,
             IM_CHANNEL_0+IM_CHANNEL_1+IM_CHANNEL_2);

/* Grab only one of the available channels on each node */
imBufPutField(Thread1, Buf1, IM_CTL_CHANNEL, IM_CHANNEL_0);
imDigGrab(Thread1, 0, Camera, Buf1, 1, Buf1, 0);

imBufPutField(Thread2, Buf2, IM_CTL_CHANNEL, IM_CHANNEL_1);
imDigGrab(Thread2, 0, Camera, Buf2, 1, Buf2, 0);

imBufPutField(Thread3, Buf3, IM_CTL_CHANNEL, IM_CHANNEL_2);
imDigGrab(Thread3, 0, Camera, Buf3, 1, Buf3, 0);
```

Note that the above assumes that *Thread1* and *Buf1* were allocated on one node, *Thread2* and *Buf2* on a second node, and *Thread3* and *Buf3* on a third node.

*An alternative
example*

Instead of setting `IM_CTL_CHANNEL` for each grab, you could grab from multiple channels to different nodes by: allocating a camera definition for each node, selecting a different channel for each camera definition, and then grabbing to a one-band buffer on each node (using the different camera definitions). There is no need to set `IM_CTL_CHANNEL` because only a single channel is selected for each camera (see below).

```
/* Select one channel on each node */
imCamAlloc(Thread1, NULL, IM_DEFAULT, &Camera1);
imCamControl(Thread1, Camera1, IM_DIG_CHANNEL, IM_CHANNEL_0);

imCamAlloc(Thread2, NULL, IM_DEFAULT, &Camera2);
imCamControl(Thread2, Camera2, IM_DIG_CHANNEL, IM_CHANNEL_1);

imCamAlloc(Thread3, NULL, IM_DEFAULT, &Camera3);
imCamControl(Thread3, Camera3, IM_DIG_CHANNEL, IM_CHANNEL_2);

/* Grab a different channel on each node */
imDigGrab(Thread1, 0, Camera1, Buf1, 1, 0, 0);
imDigGrab(Thread2, 0, Camera2, Buf2, 1, 0, 0);
imDigGrab(Thread3, 0, Camera3, Buf3, 1, 0, 0);
```

As with the previous example, the above assumes that *Thread1* and *Buf1* were allocated on one node, *Thread2* and *Buf2* on a second node, *Thread3* and *Buf3* on a third node, and does not include any synchronization to ensure that the same frame is grabbed.

Separate applications

Note that the previous two examples are single applications that use multiple channels. If you have multiple synchronized monochrome cameras and each camera is used by a separate application, each application can run on its own node (using its own channel), without regard to what is happening on the other nodes. In other words, each node can independently execute code such as the following in real-time (no frames will be missed):

```
/* Select the appropriate channel */
imCamAlloc(Thread, NULL, IM_DEFAULT, &Camera);
imCamControl(Thread, Camera, IM_DIG_CHANNEL, IM_CHANNEL(Chan));

/* Optionally adjust settings on that channel */
imCamControl(Thread, Camera, IM_DIG_REF_WHITE, RefWhite);

/* Enter real-time processing loop */
for (;;)
{
    imDigGrab(Thread, 0, Camera, Buf, 1, 0, 0);
    ...
}
```

Synchronization channel

The synchronization channel is the channel carrying the horizontal and vertical synchronization signals from the camera. You specify the synchronization channel through the `IM_DIG_SYNC_CHANNEL` field. The synchronization channel can be a specific channel or the same channel used to grab the data.

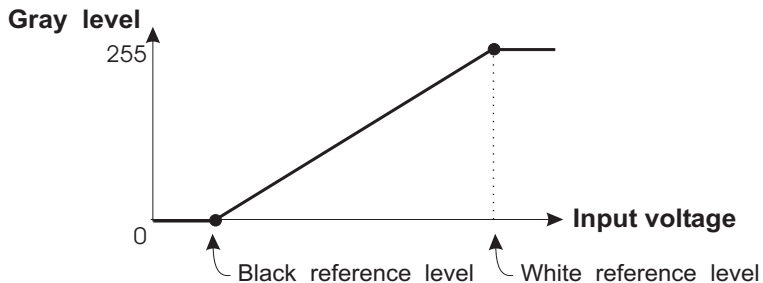
Gain and reference levels

For analog cameras, you can change the black and white reference levels and the gain used by the analog-to-digital converters. You specify the black and white reference levels through the `IM_DIG_REF_BLACK` and `IM_DIG_REF_WHITE` fields, respectively. You specify the gain through the `IM_DIG_GAIN` field. To keep the application code

hardware-independent, you specify a value between 0 and 100% for these fields. The percentage is mapped internally to one of the discrete values supported by the hardware.

Reference levels

The black and white reference levels determine the zero and full-scale levels, respectively, of the input voltage range. The analog-to-digital converters convert voltages above the white reference level to the maximum pixel value, and voltages below the black reference level to a zero pixel value.



Adjusting contrast/brightness

By reducing or increasing either or both the black and white reference levels, you can affect the brightness of the resulting image. By reducing one reference level and increasing the other, you can affect the contrast of the resulting image.

Note that adjusting the brightness and contrast of an image is usually an iterative process. You should generally choose an appropriate gain first, then adjust the reference levels. If you then change the gain, you should readjust the reference levels.

Input LUTs

You can use the LUTs in the grab module to map the data being grabbed. Note, however, that it is generally better to map data in software, after it is grabbed, using *imIntLutMap()*. Using *imIntLutMap()* is more flexible and is not dependent on the digitizer capabilities. In addition, it allows an application to be shared more efficiently with other applications that might also be using the grab module (using the LUTs in the grab module causes the digitizer to be re-programmed, possibly preventing other applications from grabbing the same frame).

For details on *imIntLutMap()*, see Chapter 3.

Using the input LUTs

If you do use the LUT in the grab module, specify the LUT buffer with which to perform the mapping using the `IM_DIG_LUT_BUF` field. You must not modify this buffer until the grab which uses that buffer has started (since the values might not be copied into the hardware until just before the grab starts). If you do modify the buffer, you must again set `IM_DIG_LUT_BUF`.

LUT requirements

The LUT buffer should be one-dimensional and allocated in processing memory. The size of the LUT should match the pixel size of the data being grabbed (for example, if the data being grabbed is 8-bit, the LUT should have 256 entries). In addition, the pixel size of the LUT should match the pixel size of the destination buffer of the grab (that is, if the destination buffer is 8-bit, each LUT entry should also be 8-bit).

Number of bands

A LUT buffer can have one band or multiple bands. If the LUT buffer has one band, the band will be used for all channels being grabbed. If the LUT buffer has the same number of bands as there are channels being grabbed, each band is used for the corresponding channel (band 0 of the LUT buffer is used for the first channel, band 1 for the second channel, etc.)

Note that you can achieve a pseudo-color effect when grabbing grayscale images by using a 3-band LUT buffer and a 3-band destination buffer.

LUT restrictions

There are some restrictions in the way the LUTs in the grab module can be used. These are:

- When grabbing from an analog camera, the LUTs must be 8 bits in and 8 bits out (that is, 256 entries of 8 bits each). In addition, to achieve a pseudo-color effect with an analog camera, the camera must have a sampling rate of 35 MHz or lower and use the first channel.
- When grabbing one or two channels from a digital camera, the LUTs can be up to 13 bits in and 8 or 16 bits out (that is, up to 8K entries of 8 or 16 bits each). When grabbing from more than two channels of a digital camera, the LUTs must be 8 bits in and 8 bits out. To achieve a pseudo-color effect

with a digital camera, the LUT can be up to 13 bits in but must be 8 bits out (for each band). In addition, you must grab from the first channel.

Frame size

You can change the width and height of the grabbed frame using the `IM_DIG_SIZE_X` and `IM_DIG_SIZE_Y` fields, respectively. Adjusting the frame width can be useful if your input source is capable of dynamically changing the number of pixels it sends per line. For certain types of input sources, adjusting the frame width can also be used to compensate for non-square pixels (non-square pixels can lead to incorrect interpretations when analyzing images). Adjusting the frame height can be useful if the number of lines per frame needs to be changed dynamically (in machine inspection, for example, the required number of lines per frame often depends on the object being inspected).

Note that, if you simply want to grab a subset of a frame, you should use the control buffer of *imDigGrab()* rather than set `IM_DIG_SIZE_X` and/or `IM_DIG_SIZE_Y`; see the *VIA options of the grab command* section for details.

Frame width

When you change the frame width for analog cameras that have the pixel clock generated by the Genesis grab module, you change the number of positions at which the analog signal is sampled on every line. Since the line period (the time between horizontal syncs) is fixed, changing the number of positions at which the analog signal is sampled is done by changing the sampling frequency (the pixel clock). This changes the physical distance spanned by each pixel. In this case, the blanking period is automatically adjusted so that only the active portion of each line is still digitized. In other words, the frame width in pixels is changed, but the physical width is not.

Note that, if your input source provides its own pixel clock, and dynamically changes the number of pixels it sends per line, you must specify the new frame width before grabbing a frame with the new size.

Frame height

When you adjust the frame height, you do not change how each line is sampled, only the number of lines grabbed per frame. Adjusting the frame height is most useful for line scan cameras that can be programmed to grab an arbitrary number of lines in order to build up an image.

User bits

The connector on the grab module has several input and output lines not designated for specific purposes; you can use them for anything you want. To do so, use the `IM_DIG_USER_IN` or `IM_DIG_USER_OUT` field, respectively. To select a specific user bit, combine the `#defines` of these fields with `IM_BITx` (for example, `IM_DIG_USER_OUT+IM_BIT1`). User bits can be set or tested through software.

For user outputs, you can use *imCamControl()* or *imDigControl()* to set a specific output low (0) or high (1). If you use *imCamControl()*, the outputs will be set just before the grab, when the rest of the digitizer is programmed. Therefore, the outputs will not interfere with any other grabs that might be in progress. If you use *imDigControl()*, the digitizer will be programmed immediately, interfering with other grabs that might be in progress.

For user inputs, use *imDigInquire()* to read the current hardware value (0 or 1) of the input line.

Note that the grab module supports both TTL and RS422 formats for user inputs and outputs; different pins on the grab connector are used in each case. You must specify which format (and hence which connector pins) you want to use. To specify whether to enable the TTL or RS422 user inputs, set the `IM_DIG_USER_IN_FORMAT` field. To specify whether to enable the TTL or RS422 user outputs, set the `IM_DIG_USER_OUT_FORMAT` field.

Triggers

The grab module has a trigger input that allows you to grab a frame upon occurrence of an event. In other words, when you use a trigger input, nothing will happen when you call *imDigGrab()*, until the event specified in the camera definition file occurs. Triggering can be useful, for example, in an automatic inspection application, where you want to grab a frame only when the part to be inspected is under the camera.

Trigger sources

The event that actually causes a frame to be grabbed is called the trigger source and is specified through the `IM_DIG_TRIG_SOURCE` field of *imCamControl()*. The trigger source can be:

- Software generated. Set `IM_DIG_TRIG_SOURCE` to `IM_SOFTWARE`.
- Hardware generated. Set `IM_DIG_TRIG_SOURCE` to `IM_HARDWARE`.
- Based on the programmable exposure timers of the grab module. Set `IM_DIG_TRIG_SOURCE` to `IM_EXPOSURE`.

Note that, if you want to use the trigger source specified in the original camera definition file, you don't have to set the `IM_DIG_TRIG_SOURCE` field.

Software triggers

When you use a software trigger, a frame is grabbed once you call *imDigCapture()* (after calling *imDigGrab()*):

```
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 */
```

A software trigger can also be enabled by calling *imDigControl()*, setting its `IM_DIG_TRIGGER` field to `IM_ENABLE`. However, *imDigCapture()* is more general; it works even if the grab is triggered indirectly by software (for example, if software is used to start the timer).

Hardware triggers

When you use a hardware trigger, a frame is grabbed once the signal on the specified hardware trigger generates:

- A positive or negative pulse.
- A level-sensitive (low or high) signal.

To select a specific hardware trigger, combine the required trigger with `IM_HARDWARE` (for example, `IM_HARDWARE+IM_TRIGGER2`). If you do not select a specific hardware trigger, the one on the same connector as the camera is used (trigger 1 is the trigger input on the analog connector and trigger 2 is the trigger input on the digital connector).

To specify whether to wait for a positive or negative pulse, or a level-sensitive signal, set the `IM_DIG_TRIG_MODE` field of *imCamControl()*.

To disable a level-sensitive signal, call *imDigControl()*, setting its `IM_DIG_TRIGGER` field to `IM_DISABLE`.

If you are not using the hardware trigger specified in the original camera definition file, you must specify whether you are using the TTL or RS422 trigger input on the grab connector. To enable either the TTL or RS422 trigger, use the `IM_DIG_USER_IN_FORMAT` field.

An example

The following code grabs a frame once a positive pulse is generated on trigger 1.

```
imCamAlloc(Thread, NULL, IM_DEFAULT, &Camera);
imCamControl(Thread, Camera, IM_DIG_TRIG_SOURCE, IM_HARDWARE+IM_TRIGGER1);
imCamControl(Thread, Camera, IM_DIG_TRIG_MODE, IM_RISING_EDGE);
imDigGrab(Thread, 0, Camera, DstBuf, 1, 0, 0); /* waits for
                                                * hardware trigger
                                                */
:
:
```

Note that the above could be used in an automatic inspection application to ensure that a frame is grabbed only when the part to be inspected is under the camera, assuming that:

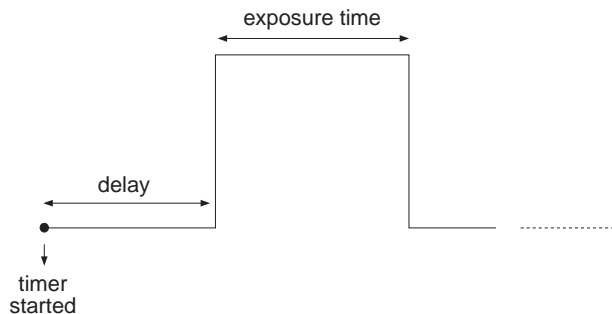
- The signal generated by a part-present sensor is connected to trigger 1.
- The part-present sensor generates a positive pulse when a part is detected.

Programmable timers

When you use one of the timers of the grab module as a trigger, a frame is grabbed a specified amount of time after the timer is started. To select a specific timer (IM_TIMER1, IM_TIMER2, etc.) as a trigger, combine it with IM_EXPOSURE. For example:

```
imCamControl(Thread, Camera, IM_DIG_TRIG_SOURCE, IM_EXPOSURE+IM_TIMER1);
```

Once a timer is started, it produces an output signal after a specified delay. This is known as the exposure signal. The exposure signal remains active for a specified amount of time (known as the exposure time).



The exposure signal can be fed to the camera to control the exposure time (or used for some other purpose, such as firing a strobe light).

Specify the exposure time through the IM_DIG_EXP_TIME field. Specify the delay through the IM_DIG_EXP_DELAY field.

Starting the timer

A timer can be started by a hardware trigger that generates a positive or negative pulse, by software, or by the horizontal or vertical syncs of the camera signals. It can also be started by the exposure signal of another timer. Specify the source of the timer with the IM_DIG_EXP_SOURCE field. Note that a timer cannot be started by a level-sensitive hardware trigger.

If the exposure output was not specified in the original camera definition file, you must specify whether you want to use the TTL or RS422 output pin on the grab connector. To enable either the TTL or RS422 exposure output, use the IM_DIG_USER_OUT_FORMAT field.

From hardware

When you use a hardware trigger, the timer starts once the signal on the specified hardware trigger input generates a positive or negative pulse. By default, the trigger on the same connector as the camera is used (trigger 1 is the trigger input on the analog connector and trigger 2 is the trigger input on the digital connector). If you want to select a specific trigger, combine it with IM_HARDWARE. For example,

```
imCamControl(Thread, Camera, IM_DIG_EXP_SOURCE, IM_HARDWARE+IM_TRIGGER2);
```

To specify whether to wait for a positive or negative pulse, use the IM_DIG_TRIG_MODE field.

From software

When you use software, the timer starts once you call *imDigControl()* with its IM_DIG_EXPOSURE field set to IM_ENABLE.

From horizontal or vertical syncs

Horizontal or vertical syncs are typically used when the camera does not require a trigger, but you still need to output exposure signals for every line or frame.

An example

An application that might need exposure signals is one that uses an asynchronous-reset camera to capture an image of a part along an inspection line. An asynchronous-reset camera is a type of camera that can be reset at any time to force it to start a new frame. It is often used when the time interval between the receipt of a trigger and the capture of an image is critical. A strobe light is often used in such an application to freeze parts at precise moments in time. Assuming that the signal generated by a part-present sensor is connected to a trigger input of the grab module, the sequence of events could be:

1. Once the required pulse (positive or negative) is generated on the trigger, send a signal to reset the camera. The first exposure signal of the grab module can be used for this purpose. Usually, the signal must remain active for some minimum amount of time in order to reset the camera properly. Note that the hardware trigger must be specified as the source that starts the timer.
2. Send a second signal to fire the strobe light. This must also be timed very precisely, and is usually output just after the camera reset signal. The second exposure signal of the grab module can be used for this purpose, and the first exposure signal is specified as the source (so that the delay before firing the strobe is measured relative to the camera reset signal).
3. Capture the first frame after the camera is reset. Since the first exposure signal resets the camera, it is used as the trigger source.

The code is shown below. Note that all the time-critical events are controlled by the hardware of the grab module; software only has to make sure that the grab command is queued before the trigger is received.

```

/* Reset the camera 10μs after the hardware trigger with a pulse of 85μs */
imCamControl(Thread, Camera, IM_DIG_EXP_SOURCE+IM_TIMER1, IM_HARDWARE);
imCamControl(Thread, Camera, IM_DIG_TRIG_MODE, IM_RISING_EDGE);
imCamControl(Thread, Camera, IM_DIG_EXP_MODE+IM_TIMER1, IM_ACTIVE_HIGH);
imCamControl(Thread, Camera, IM_DIG_EXP_DELAY+IM_TIMER1, 10.0E-6);
imCamControl(Thread, Camera, IM_DIG_EXP_TIME+IM_TIMER1, 85.0E-6);

/* Fire the strobe 64μs after resetting the camera */
imCamControl(Thread, Camera, IM_DIG_EXP_SOURCE+IM_TIMER2, IM_TIMER1);
imCamControl(Thread, Camera, IM_DIG_EXP_MODE+IM_TIMER2, IM_ACTIVE_HIGH);
imCamControl(Thread, Camera, IM_DIG_EXP_DELAY+IM_TIMER2, 64.0E-6);
imCamControl(Thread, Camera, IM_DIG_EXP_TIME+IM_TIMER2, 50.0E-6);

/* Trigger the grab only after the camera is reset */
imCamControl(Thread, Camera, IM_DIG_TRIG_SOURCE, IM_EXPOSURE+IM_TIMER1);

/* Wait for hardware trigger then grab */
imDigGrab(Thread, 0, Camera, DstBuf, 1, 0, 0);

```

Running multiple applications

If you are running several applications simultaneously on a Genesis system and if more than one of these applications grab images, there are some specific guidelines you should follow to ensure that the digitizer is shared properly. (Note that general guidelines were given in Chapter 2). These specific guidelines, which also ensure that applications will run unchanged on different Genesis configurations or with different camera types, are:

- If your application is not dependent on a particular camera type, set the *CamFile* parameter of *imCamAlloc()* to NULL so that the camera definition file that you specified during installation is allocated.
- Use *imCamInquire()* to obtain the size of the buffer needed for a grabbed image. You can also use *imCamInquire()* to obtain the number of bands needed for the grabbed image, if your application does not require a specific number of bands. Note, however, that you can always grab a color image into a single-band buffer and a monochrome image into a multi-band buffer. In addition, you do not need to know the number of bands when grabbing into display memory, since most functions which operate on display buffers will transparently work with one or three bands.
- Avoid re-programming the digitizer. This means that, after you allocate a camera definition (using *imCamAlloc()*), avoid using *imCamControl()*. If you use *imCamControl()*, the camera definition might become incompatible with those used by other applications, forcing the digitizer to be re-programmed (if the digitizer has to be re-programmed, the same frame cannot be grabbed by several applications). Note that *imDigControl()* programs the digitizer immediately, so you should never call this function if you want to share the digitizer between several applications.
- Avoid doing a continuous grab unless absolutely necessary. A continuous grab will prevent other applications from grabbing into the same memory bank.

Chapter 11: Displaying images

This chapter describes how to display images, and other related topics.

The display section

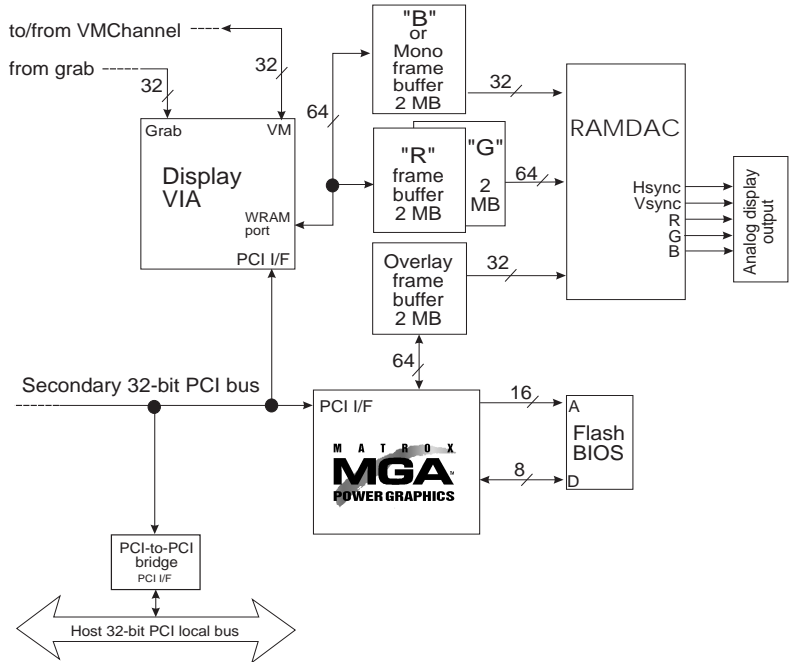
The display section, which is optional on the Genesis main board, consists of two separate frame buffers: a main (underlay) frame buffer and an overlay frame buffer. The overlay frame buffer is powered by the 64-bit Matrox MGA Millennium 2064W graphics accelerator, and interfaces directly with the on-board secondary PCI bus.

The display section has up to 8 MBytes of frame buffer memory: 2 MBytes for the overlay frame buffer and either 2 or 6 MBytes for the main frame buffer (depending on whether you have the monochrome or color version of the display section).

The display section has a maximum resolution of 1600 x 1200, with an 85-Hz non-interlaced refresh rate. It supports the display of captured video in real time.

A 128-bit RAMDAC provides digital-to-analog conversion. The RAMDAC has three 8-bit LUTs that map the contents of the overlay frame buffer. When the overlay is not required, these LUTs can map the contents of the main frame buffer.

Below is a block diagram of the display section. Note that you will find more detail on some of the components in the *Genesis Installation and Hardware Reference*.



Display memory

On Matrox Genesis, processing memory and display memory are physically distinct. Although the destination buffer of a processing function can be located in display memory, it is more efficient if it is in the memory directly attached to the 'C80 (that is, the SDRAM) and then copied to the display when necessary. In this case, display memory is being used to hold a second copy of a buffer. Note that if the processing function uses the NOA, the destination buffer must be located in local processing memory.

To display an image, you should not allocate memory in the main or overlay frame buffer. Instead, you should use *imBufChild()* to create a child buffer on the screen (at the location you wish to display the image) and then copy the processed buffer to this on-screen child buffer when you need to see it.

Grayscale images vs. color images

There are two versions of the display section:

- Monochrome version.
- Color version.

When you set the *Buf* parameter of *imBufChild()* to `IM_DISP`, the resulting on-screen child buffer will automatically have one band if the display is in monochrome mode and three bands if it is in color mode. Note, however, that you rarely need to know how many bands there are, since most functions which operate on display buffers will transparently work with one or three bands.

Using the monochrome version

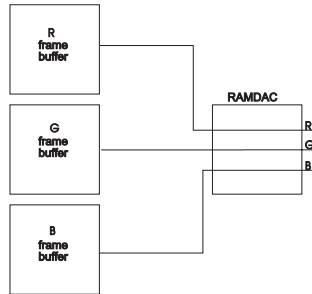
When displaying an image on the monochrome version of the board, the same data is sent to all three color display guns (RGB). As a result, the image is displayed in grayscale.



Note that, if a color application is run on the monochrome version of the board, only one band of the color image can be displayed (the first band). To display this band, allocate an on-screen child buffer (use *imBufChild()*, setting its *Buf* parameter to `IM_DISP`) and then copy the color image to this buffer. The color image will be displayed in grayscale. If you prefer that the color application not be able to run on the monochrome version of the board, try to allocate a 3-band child display buffer (set the *Buf* parameter of *imBufChild()* to `IM_DISP_COLOR`). This will fail on the monochrome version of the board (the buffer identifier will be returned as 0 and an error will be logged).

Using the color version

When displaying an image on the color version of the board, each band of the image is sent to a different color display gun (RGB). As a result, images are displayed in true color.



To display a grayscale image on the color version of the board, you need to copy the image to all three display buffers. This is done automatically if you copy or grab a one-band image into an on-screen child buffer that was allocated by setting the *Buf* parameter of *imBufChild()* to *IM_DISP*. Therefore, there is no need to modify a monochrome application if it is to run on the color version of the board. In addition, you can display grayscale images at the same time as color images.

In specialized applications, the color version of the board can also be used to drive three independent monochrome outputs.

Changing the display mode

On the color version of the board, you can change the display mode to monochrome (using *imDispControl()*). This can be useful if a monochrome application needs the extra display memory (for example, for extra storage or to double buffer the display). Note, however, that you do not need to change the display mode to monochrome to display grayscale images. In addition, if you select monochrome mode, you will interfere with color applications that might also be using the display. Therefore, you should only change the display mode to monochrome if an application has exclusive use of the display and needs the extra display memory.

In monochrome mode, the color version of the board behaves just like the monochrome version, except that you can allocate on-screen child buffers in a specific buffer (red, green, or blue).

Using the overlay

The display section consists of two frame buffer surfaces: the main (underlay) frame buffer and the overlay frame buffer. This allows you to work in either a single-screen or a dual-screen mode, which is determined at installation time.

Single-screen mode

When you are working in single-screen mode, the overlay frame buffer is used by the Matrox MGA 2064W graphics accelerator to display the Host operating system's user interface.

In this mode, you can use the overlay frame buffer to write and/or draw non-destructively over the main frame buffer. If you are working with Windows, use the Windows GDI functions to do so. You should not normally use Genesis functions to access the overlay in single-screen mode since they might interfere with Windows.

Dual-screen mode

When you are working in dual-screen mode, one screen displays the Host operating system's user interface, while the other displays the Genesis frame buffers.

In this mode, you can also use the overlay frame buffer to write and/or draw non-destructively over the main frame buffer. However, this time, you will normally use Genesis functions to access the overlay. As with the main display, the best way to use the overlay with the Genesis functions is to allocate a two-dimensional region of interest (using *imBufChild()*) and then copy the data you want to overlay from a processing buffer to the overlay display buffer. However, if you are drawing only a small amount of annotation into the overlay (with the graphic functions, for example), it might be more efficient to draw it directly into the overlay (that is, give the overlay child buffer as the destination buffer).

Resolution

On Genesis, the Matrox MGA 2064W graphics accelerator only supports 8 bits per pixel for the overlay. In single-screen mode under Windows, use the MGA **Display Properties** utility to configure the overlay frame buffer to the desired 8-bit resolution. Otherwise, use the GENVCFLD utility. The main frame buffer resolution is automatically set to be the same as the overlay resolution. See the *Genesis Installation and Hardware Reference* for details on changing resolution and the GENVCFLD utility.

Note that you should not change the resolution while applications are using the display, since all display buffers will become invalid.

Keying

Keying is an effect that switches between two display sources according to pixel values in one of the sources (that is, according to a keying color). 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. Keying is controlled with the *imDispControl()* function (use the IM_DISP_KEY_MODE field to specify the keying mode and the IM_DISP_KEY_LOW and IM_DISP_KEY_HIGH fields to specify the range of the keying color).

In single-screen mode

By default, only the overlay frame buffer is visible in single-screen mode. To use keying in a way that will not interfere with other applications that might also be using the display, you should first inquire about the current keying mode. If keying is disabled, you should enable it and use a single keying color. If keying is already enabled, you should inquire about the current keying color and use that color as your keying color. For example,

```
if (imDispInquire(Thread, 0, IM_DISP_KEY_MODE, NULL) == IM_KEY_OFF)
{
    KeyVal = 3; /* Choose any system color that's rarely used */
    imBufPutField(Thread, ControlBuf, IM_DISP_KEY_MODE, IM_KEY_IN_RANGE);
    imBufPutField(Thread, ControlBuf, IM_DISP_KEY_LOW, KeyVal);
    imBufPutField(Thread, ControlBuf, IM_DISP_KEY_HIGH, KeyVal);
    imDispControl(Thread, 0, ControlBuf, IM_FRAME);
}
else
    KeyVal = imDispInquire(Thread, 0, IM_DISP_KEY_LOW, NULL);
```

You should then fill, with the keying color, those areas of the overlay that you want transparent (use Windows functions to do so). You should not disable keying when the application terminates, since this will interfere with other applications using the display.

In dual-screen mode

By default, only the main frame buffer is visible in dual-screen mode. If your application is intended to run in dual-screen mode and does not use the overlay buffer, you should not enable keying. This will also make the program useable in single-screen mode, since you can simply run a separate program (such as the GENKEY utility) to enable keying. See the *Genesis Installation and Hardware Reference* for details on the GENKEY utility.

A dual-screen application that does use the overlay will never work well with other applications, especially single-screen applications under Windows. Such a dual-screen application can therefore explicitly enable keying, using any keying mode it requires.

Panning, scrolling, and zooming

You can move the source of your on-screen display by panning, scrolling, or zooming. Use the *imDispControl()* function. Note that panning, scrolling, and zooming are only display effects and do not modify the data in the frame buffers.

You can pan and scroll images in the main frame buffer (this can be useful when the images are larger than the display resolution). You can zoom images (by a factor of 2 or 4), although zooming affects both the main and overlay frame buffers.

*Modify the data
instead*

It is usually better (especially in single-screen mode when several applications might be sharing the display) not to use *imDispControl()* for display effects. Rather, you should modify the data before or during the copy to the display (limited zooming, for example, can be performed by the advanced copy functions). In this way, different effects can be applied to each displayed image. If you want to use zoom factors other than those supported by the copy functions, or if you prefer zoom with interpolation, you can use the appropriate geometric function (*imIntZoom()*, *imIntScale()*, or *imIntWarpPolynomial()*). Note that, if you are using integer factors without interpolation, *imIntZoom()* is the fastest of the three functions. If you are using integer factors with interpolation, *imIntScale()* is the fastest. If you are using non-integer factors, *imIntScale()* is the fastest but has more restrictions than *imIntWarpPolynomial()*.

Look-up tables

The main frame buffer does not have dedicated display LUTs. You can, however, map your data using *imIntLutMap()* and then copy the result to the display. Alternatively, you can use the LUTs in the RAMDAC, if they are not being used to map the overlay frame buffer (to do so, disable the overlay frame buffer using *imDispControl()*). However, using *imIntLutMap()* is more flexible, since each image to be displayed can be mapped in a different way. Note that mapping a color image requires three calls to *imIntLutMap()* (one for each band) while mapping a one-band image requires just one call.

*Displaying in
pseudo-color*

You can display a grayscale image in pseudo-color through three separate mappings of the one-band image (send one mapping to the red display buffer, one to the green display buffer, and one to the blue display buffer). However, it is more efficient to map each value of the one-band image to an RGBa value in a single call to *imIntLutMap()* (use a 32-bit destination buffer and a LUT with 256 entries of 32 bits each). You can then copy the 32-bit buffer directly to the display; the color components will automatically be separated into the correct display buffers.

Grab and display

There will be times when you want to display immediately the images you are grabbing, especially when grabbing continuously. To do so, use *imDigGrab()* with an on-screen child buffer as the destination. Use *imBufChild()* to allocate the on-screen child buffer (to ensure that the same code will work on either the monochrome or color version of the display, set the *Buf* parameter of *imBufChild()* to *IM_DISP*).

When you call *imDigGrab()*, you can specify a number of options through the control buffer passed to this function. Many of these options are particularly useful when grabbing to the display (such as extracting the most-significant byte of images with a pixel depth of more than 8 bits). These options were described in Chapter 10.

*Non-rectangular
windows*

To grab into a non-rectangular window of the display, you have to use tag buffers. Tag buffers were described in Chapter 9.

Using the hardware cursor

The Genesis display section has a hardware cursor that can be displayed and positioned on-screen through the use of Genesis Native Library functions. These functions allow you to, for example, interface a secondary pointing device with the Genesis hardware cursor. The cursor functions should be used only in a dual-screen display configuration because they will interfere with the Windows display driver in a single-screen display configuration.

The Native Library cursor functions are synchronous and executed by the Host, rather than being queued to the Genesis board; therefore, they have no thread parameter. They do, however, have a device parameter to indicate which Genesis board should be used to display the cursor when more than one board is present.

General steps to using a cursor

To use the Genesis hardware cursor, you need to perform the following steps (steps 1, 2, and 7 are only needed when defining a new cursor; these steps can be omitted if you use the default cursor).

1. Allocate a Genesis Native Library cursor by calling *imCurAlloc()*.
2. Define and set the cursor's shape (and hot spot) and color attributes by calling *imCurDefine()* and *imCurSetColor()*, respectively.
3. Load the cursor attributes into the physical hardware and select the software copy of the cursor to the display by calling *imCurSelect()*.
4. Set the cursor's initial display position by calling *imCurSetPosition()*.
5. Enable the hardware cursor, that is, make the hardware cursor visible, by calling *imCurEnable()*.
6. Track and move the cursor using the *imCurGetPosition()* and *imCurSetPosition()* functions.

7. Free the cursor and any cursor-related resources when it is no longer needed by calling *imCurFree()*.

Allocating the cursor

As with other Native Library resources, a cursor must first be allocated. You can allocate any number of cursors with *imCurAlloc()* because cursor allocation just sets up a virtual (software) copy. When a cursor is allocated, its attributes are initially undefined.

The cursor should be freed when no longer needed.

Defining the cursor's shape and hot spot

The cursor's shape and hot spot are specified by calling *imCurDefine()*. The maximum dimensions for a cursor are 64x64 (width x height), and the hot spot is defined relative to the top-left corner (0,0). The hot spot is the coordinate of a specific pixel within the defined cursor to which all cursor positions refer. Valid positions for the hot spot are between (0,0) and (63,63), inclusive.

Instead of defining your own cursor, you can also use the default cursor.

Default cursor

There is a default arrow cursor with its hot spot defined at the tip of the arrow. To use it, you simply have to select it (load it into the hardware); you do not have to allocate or define it. To do so, call *imCurSelect()* and pass 0 as the cursor ID. Then, enable this cursor (make it visible). For example:

```
imCurSelect(Device, 0);
imCurEnable(Device, IM_ENABLE);
```

Defining the cursor's colors

The cursor's on-screen appearance is specified by calling *imCurSetColor()*. The cursor is essentially a pixel image. Each cursor pixel has a 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 calling *imCurSetColor()* and specifying their red, green, and blue color components. For each color component, the valid pixels values range from 0 to 255.

Note that when a cursor smaller than 64x64 is defined, the remaining area will not be visible.

Selecting a cursor to appear on-screen

The cursor attributes that you specify are loaded into the hardware and a cursor is selected to the display, overriding the existing cursor by calling *imCurSelect()*. It is important to note that although you can allocate and set the display properties of an unlimited number of cursors, only one cursor can be loaded into the hardware (selected) at a time.

Once a cursor is selected, the hardware cursor can be made visible or invisible by calling *imCurEnable()*. The cursor appears on-screen as a non-destructive pixel image displayed on top of the underlay and overlay frame buffers, with a defined shape and color scheme. Since the hardware cursor is not always needed as a display effect, the hardware cursor can also be made invisible with *imCurEnable()*. Keep in mind that since the hardware cursor is initially undefined, a cursor should be selected to the display before being enabled.

Moving and tracking the 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. Keep in mind that when you set the cursor's position, the coordinates are those of the hot spot, measured from the top-left corner of the screen.

Note that if you specify a position that is not valid for the current screen resolution, the cursor will not be visible.

You are responsible for tracking the cursor using the *imCurGetPosition()* and *imCurSetPosition()* functions. For example, when the cursor is accessed by simultaneous processes dispatched to multiple threads, you can use *imCurGetPosition()* to read back the current cursor position before executing an operation on a particular thread. Similarly, if you zoom and/or pan the display with *imDispControl()*, a subsequent call to a cursor function will take the zoom factor into account; however, the cursor's current position on the screen will not be not changed automatically. You should reset the cursor position after the display is panned or zoomed by calling *imCurSetPosition()*.

Note that the cursor itself is only zoomed in the X direction when the display is zoomed; due to a hardware restriction, it keeps its size in the Y direction. Despite these limitations, the cursor functions normally when zoomed.

An example

The following example demonstrates how to define, select, and make visible a cross-shaped cursor with dimensions of 32x32 and its hot spot at coordinates (15,15). The color scheme is chosen so that the cursor appears in the form of a red and white cross.

[illegible]

```
/* Allocate and define a 32x32 custom cursor with hot-spot at (15,15). */
imCurAlloc(Device, IM_DEFAULT, &Cursor);
imCurDefine(Device, Cursor, 32, 32, 15, 15, CurData);

/* Set first two colors. */
imCurSetColor(Device, Cursor, 1, 255, 0, 0); /* Red */
imCurSetColor(Device, Cursor, 2, 255, 255, 255); /* White */

/* Select it as the current cursor and make it visible. */
imCurSelect(Device, Cursor);
imCurSetPosition(Device, PosX, PosY);
imCurEnable(Device, IM_ENABLE);
```

The full example is included in the Genesis Native Library
`\EXAMPLES\HOST\MISC\` directory that is on the CD.

Display memory as extra storage space

You can use your off-screen display memory as extra linear storage space for images. This can be useful if you are using a tag buffer during a grab or VM transfer to the display (a tag buffer must be in the same memory bank as the destination buffer).

To allocate off-screen display memory, use *imBufAlloc...()*, setting its *Location* parameter to *IM_DISP*. There are no restrictions on the type or size of buffer that you can allocate in display memory, although there is a limited amount of off-screen memory available.

*Host memory
vs. display memory*

If all you want is extra general-purpose storage space (for example, you are not using tag buffers which must be in display memory), you can use Host memory instead of display memory. Your Host memory is probably not as limited as your display memory, and you can always add more to suit your requirements.

Processing speed

Note that, if you need to process a buffer allocated in display (or Host) memory, you must first copy it to processing memory, since the source buffers of a processing function must be in local processing memory, that is, in processing memory on the same node as the thread which will execute the processing function. For maximum efficiency, the destination buffer of a processing function should also be in local processing memory. Note that the destination buffer of a processing function that uses the NOA must be located in local processing memory.

Displaying the buffer

It is still possible to display an image allocated in a linear format, provided it is 8 bits deep. Note, however, that this is not recommended, since only one such image can be displayed at a time. (Only one image can be displayed because the start address and pitch of the display must be set to be the same as those of the image, and won't necessarily be compatible with other images currently in display memory.) Use *imDispControl()* to display your data and specify the ID of the buffer to display.

Running multiple applications

This section is a summary of the guidelines you should follow if you want to properly share the display between multiple applications at the same time. Note that most of these guidelines were mentioned in previous sections of this chapter and that general guidelines regarding multiple applications were given in Chapter 2.

- When you allocate an on-screen child buffer, always set the *Buf* parameter of *imBufChild()* to IM_DISP. This will allow the application to run on either the monochrome or color version of the display.
- Avoid changing the display mode to monochrome (this is possible if you have the color version of the display section). The only time you should change the display mode to monochrome is when an application has exclusive use of the display and needs the extra display memory.
- Do not use Genesis functions to access the overlay in single-screen mode; this might interfere with Windows.
- Enable keying only after you have inquired about the current keying mode and ensured that keying was disabled.
- Do not disable keying when an application terminates; this will interfere with other applications using keying.
- Do not enable keying in a dual-screen application that does not use the overlay. This will at least make the program useable in single-screen mode, since you can simply run a separate program (such as the GENKEY utility) to enable keying (see the *Genesis Installation and Hardware Reference* for details on the GENKEY utility).
- Do not use *imDispControl()* to pan, scroll, or zoom your on-screen display. If you need these display effects, perform them before or during the copy to the display.
- If you allocate off-screen display memory, use *imBufAlloc...()*, setting its *Location* parameter to IM_DISP.

Chapter 12: Error handling

This chapter describes the various error mechanisms available with the Genesis Native Library.

Genesis and error reporting

Error mechanisms

Error mechanisms

Often, it is important to check that functions have performed successfully (such as after allocating resources). However, most functions in the Native Library are asynchronous; they queue their command to the hardware and then immediately return control to the Host. Therefore, the return values of most functions cannot indicate whether they were performed successfully; but at most whether they were successfully sent to the board. Synchronous functions could return a meaningful error value but these would be tedious to check after every call. For these reasons, errors are only reported when requested, not through the return values of functions.

With the Genesis Native Library, you can:

- Check application-wide errors, using either *imAppGetError()* or *imAppCatchError()*. *imAppGetError()* returns information about the first error detected in the application. *imAppCatchError()* calls a user-defined function once an error in the application is detected.
- Check a specific thread, using *imThrGetError()*. This function returns information about the first error detected in a specific thread.
- Check a specific asynchronous function, using *imSyncGetError()*. This function checks the outcome of a specific function call.

When using *imAppGetError()* or *imThrGetError()*, the error returned is the first to occur since error information about the application or thread was last cleared. You clear error information by specifying an `IM_ERR_RESET` flag when calling these functions.

- ❖ 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. Since functions in a thread execute serially, *imThrGetError()* does not have this problem.

Which error mechanism to use

In general, you should check application-wide errors when you do not expect errors but still need to be sure that none have occurred. However, if your application uses only a single thread, *imThrGetError()* is generally a better alternative to *imAppGetError()* because it requires no explicit synchronization. Then, errors will usually be handled in the following manner:

```
void main()
{
    char error[IM_ERR_SIZE]
    ...
    imBufAlloc2d(thread, 512, 480, IM_UBYTE, IM_PROC, &buf1);
    imBufAlloc2d(thread, 512, 480, IM_UBYTE, IM_PROC, &buf2);
    imDigGrab(thread, 0, cam, buf1, 1, 0, 0);
    imIntConvolve(thread, buf1, buf2, IM_SMOOTH, 0, 0);
    ...

    /* Check for errors */
    if (imThrGetError(thread, IM_ERR_MSG_FUNC, error))
        printf("%s\n", error);
    else
        printf("Completed successfully\n");

    /* Free resources */
    ...
}
```

Note that there are two types of errors that *imAppGetError()* can detect that *imThrGetError()* cannot. These are invalid thread IDs, and errors caused by functions not sent to a thread.

Checking a specific function

You can use *imSyncGetError()* when you want to check the outcome of a specific asynchronous function call. If a function produces an error, it will always be recorded in that function's OSB, and can then be retrieved by *imSyncGetError()*. (*imAppGetError()* and *imThrGetError()* will only report the error if it is the first to occur since error information was last cleared).

More about application-wide errors

You can check application-wide errors using either *imAppGetError()* or *imAppCatchError()*.

Using
imAppGetError()

When you use *imAppGetError()*, information is returned about the first error detected in the application since error information about the application was last cleared. You clear error information about an application by specifying an IM_ERR_RESET flag when calling *imAppGetError()*:

```
imAppGetError(IM_ERR_MSG_FUNC + IM_ERR_RESET, errmsg);
```

When you call *imAppGetError()*, you can retrieve the code of the detected error, the name of the function that caused the detected error, and the message generated by the detected error. When you clear, all these items are simultaneously cleared. This ensures that, at any time, all error items pertain to the same detected error. Note that you should only clear error information when retrieving the last required error item.

Using
imAppCatchError()

When you use *imAppCatchError()*, a user-defined function is called if 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.

imAppCatchError() can be used to establish a standard response to errors. For example, the following shows you how to use *imAppCatchError()* so that error messages are printed whenever an error is detected:


```

...
void myhandler(void);

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

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

void myhandler()
{
    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);
}

```

Places to check for errors

To some degree, the placement of the error checking functions *imAppGetError()* and *imThrGetError()* is application-dependent. However, there are a few places where they should normally be used:

- After the initialization section of the application, where buffers and other resources are usually allocated. Since it is quite possible that some of the allocations failed, it is a good idea to check for errors and, if any allocations did fail, clean up and exit. Since the first error in a sequence of functions is recorded, you only need to make a single call to *imAppGetError()* or *imThrGetError()*.
- At the end of the application, just before freeing buffers and other resources. Note that errors would not usually be expected here if your application has been correctly written. However, during the development phase, it is common to have bugs (for example, bad buffer IDs caused by passing incorrect parameters). Checking for errors during application development can save you debugging time.

A simple application can usually make do with just the second method. Even if some buffer allocations failed early in the application, they will return invalid buffer IDs which will be trapped by subsequent processing functions. Therefore, the error will eventually be reported when you call *imAppGetError()* or *imThrGetError()* at the end of the application.

A more complex application might require extra error checking. For example, an application with a long processing loop should check for errors before entering the loop, to avoid the possibility of looping with, say, a bad buffer ID. You might even want to check for errors within the loop, unless the loop is a very time-critical one.

Chapter 13: Optimizing your application

This chapter describes how to improve the performance of your application.

Overview

With the Genesis Native Library, there are various ways you can improve the performance of your application. Two simple but effective ways, which apply to any application, are:

- **Use the smallest possible data type.** In general, functions run faster when you use a smaller data type. For example, many processing functions run up to twice as fast on 8-bit data than they do on 16-bit data. There can be a much greater difference in speed between binary and other data types, so when possible, you should use binary buffers for binary data (rather than, say, 8-bit buffers with only the values 0 and 0xFF).
- **Choose the best function.** With the Genesis Native Library, there are sometimes different ways of doing the same thing. Therefore, if a function or sequence of functions is too slow for your application, you should consider others; see the *Genesis Native Library Command Reference*.

Multiprocessing

You can also improve performance by multiprocessing (that is, by executing operations in parallel). You can often execute several operations in parallel on a single node. In addition, almost any application can be made to run faster if you have more than one node in your system.

Programming the 'C80

In those few cases where the Native Library does not provide sufficient performance, you can program the 'C80 directly. To do so, you need to use the optional Genesis Developer's Toolkit, in conjunction with Texas Instruments' TMS320C8x software development tools. Note that the 'C80 is a complex chip, so programming it should not be undertaken lightly, even though it gives you complete access to all features of the board.

Estimating performance

To estimate the performance of your application, benchmarks of functions with common data types have been provided. These can be found in the *optimize.doc* file in the \GENESIS\DOC directory. This section describes how to estimate performance for cases not listed in the file. In general, the rules given here apply to all functions; any exceptions are described in the *optimize.doc* file.

Note that, while the rules given here allow you to quickly estimate the performance of a function, it is always more accurate to use *imSysClock()* to measure execution time. For details, see the *Genesis Native Library Command Reference*.

General formula

In general, when adjusting a benchmark to your specific case, you need to subtract the function's overhead from the benchmark, scale the result by an appropriate factor, then add the overhead, i.e.,

$$\text{Performance} = (\text{Benchmark case} - \text{Overhead}) \times \text{Scale} + \text{Overhead}.$$

A function's overhead can be determined from its total execution time and its processing rate (as described in the *Overheads* section).

For all functions, benchmarks have to be scaled according to the new image size involved. For example, if a function with a 0.5 ms overhead takes a total of 2.5 ms for a 512x512 image, then for a 256x256 image, it will take:

$$(2.5 - 0.5) \times (256^2/512^2) + 0.5 = 1.0 \text{ ms.}$$

For a 1024x1024 image, it will take:

$$(2.5 - 0.5) \times (1024^2/512^2) + 0.5 = 8.5 \text{ ms.}$$

In addition to the new image size, benchmarks for I/O bound functions have to be scaled according to the bytes/pixel involved (as described in the *I/O bound functions* section).

Overheads

All functions have a fixed overhead. This means that an image cannot be processed in less time than this overhead, no matter how small the image is. It also means that functions become less efficient when operating on small images, since the overhead becomes a greater part of the total execution time.

A function's overhead can be determined from its total execution time and its processing rate, both of which are listed in the *optimize.doc* file. For example, if a function takes 3.2 ms to process a 512x512 image and has a processing rate of 94 MPixels/sec, its overhead is

$$3.2 \text{ ms} - \frac{512^2 \text{ pixels}}{94 \text{ MPixels/sec}} \approx 0.4 \text{ ms}$$

Overheads might be reduced in the future, so you should always consult the latest version of the *optimize.doc* file when determining a function's overhead.

Note that simple asynchronous functions (such as *imBufPutField()*), and other functions that don't operate on images, have the lowest overhead (currently about 0.2 ms). Synchronous functions (such as *imBufGetField()* and *imBufChild()*) have the highest overhead (currently about 0.6 ms under Windows NT, slightly less under DOS). Processing functions usually have an overhead somewhere in-between (currently about 0.5 ms).

I/O bound functions

A function is I/O bound if its performance is limited by the speed at which it can access data in memory. Note that functions which use the parallel processors (PPs) always work by transferring the image from external memory into on-chip memory, a block at a time. Each block is processed in on-chip memory, then the results are transferred back to external memory. Processing and transferring can overlap, in which case the PPs are not kept waiting for data. However, if the data is processed faster than it can be transferred, the PPs are kept waiting at least part of the time, and the function is said to be I/O bound.

A function can be assumed to be I/O bound if the *optimize.doc* file indicates that it requires a bandwidth of about 300 MBytes/sec.

To estimate the performance of I/O bound functions for cases not listed in the *optimize.doc* file, you need to scale according to the number of bytes/pixel involved, in addition to the image size involved. For example, assume a function with two source buffers and one destination buffer has the following benchmarks when its source and destination buffers are 512x512 pixels large and 8 bits deep.

Time with overhead: 3.2 ms
 Rate without overhead: 94 MPixels/sec
 I/O without overhead: 280 MBytes/sec

The overhead, as calculated in the previous section, is 0.4 ms.

Given the above information, you can estimate the performance for buffers of different sizes and/or depths. For example, if the source and destination buffers are 512x512 but 16-bit, there are twice as many bytes involved, so performance should be $(3.2 - 0.4) \times 2 + 0.4 = 6.0$ ms. If the source and destination buffers are 8-bit but 256x256, there are 1/4 as many pixels involved ($256^2/512^2 = 1/4$), so performance should be $(3.2 - 0.4) \times 1/4 + 0.4 = 1.1$ ms. If the source and destination buffers are 256x256, the source buffers are 8-bit, and the destination buffer is 16-bit, there are 1/4 as many pixels involved and 4/3 as many bytes, so performance should be $(3.2 - 0.4) \times 1/4 \times 4/3 + 0.4 = 1.3$ ms.

Note that, in the benchmark case, the total I/O is 3 bytes/pixel (since there are two source buffers and one destination buffer). To determine the total I/O for other cases, simply scale according to the number of pixels and bytes involved. For example, when the source and destination buffers are 512x512 but 16-bit, the total I/O is $3 \times 2 = 6$ bytes/pixel.

Compute bound functions

A function is said to be compute bound if its performance is limited strictly by the speed at which it can process the data, and not by other factors such as how fast the data can be accessed in memory. Benchmarks for most compute bound functions are listed in the *optimize.doc* file. However, for neighborhood functions that support user-defined kernels, benchmarks are not given for all possible kernel sizes. To estimate performance when your case is not given, scale the appropriate benchmark according to the number of values in your kernel. For example, if the processing rate using a 5x5 kernel (25 kernel values) is 7.65 MPixels/sec, then using a 7x7 kernel (49 kernel values), it should be $7.65 \times 25 / 49 = 3.90$ MPixels/sec.

Note that, for some functions, the performance estimated using the above rule might not always agree with the actual performance. For example, the performance of *imIntConvolve()* also depends on the content of the kernel. Specifically, when kernel values are all the same (or where only the center value is different), performance is faster than when kernel values are completely arbitrary.

NOA setup overhead

It takes quite a long time to set up the NOA for a processing pass, and this overhead can be very significant for small images (it can represent about a 40% overhead even on a 512x512 image when using small kernels). To reduce the NOA set-up time, you can save some or all of the hardware register values in a cache buffer. This is useful when performing an operation with *imBinMorphic()*, *imIntConvolve()*, or *imIntErodeDilate()*. Doing so can reduce processing time for a subsequent call to any of these functions. The first call to one of these functions 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).

You have the option to allocate the cache buffer yourself (this will save a little time on the first call), or have it allocated automatically. Note that if you choose to allocate it yourself, then you need to allocate a one-dimensional, 8-bit buffer of size `IM_CACHE_BUF_SIZE`. Either way, you are responsible for freeing the buffer when you no longer need it.

Example

The following example demonstrates how to reduce NOA setup overhead when using the *imIntConvolve()* command.

```
/* First convolution to save the NOA setup */
imBufPutField(Thread, CtrlBuf, IM_CTL_CACHE_BUF, 0);
imBufPutField(Thread, CtrlBuf, IM_CTL_SETUP, IM_SAVE);
imIntConvolve(Thread, SrcBuf, DstBuf, KerBuf, CtrlBuf, 0);

/* Select fast setup, but allow Src or Dst buffers to be different */
imBufPutField(Thread, CtrlBuf, IM_CTL_SETUP, IM_ADDRESS_ONLY);

/* Subsequent convolutions with this control buffer will be faster */
for (;;)
{
    imIntConvolve(Thread, SrcBuf, DstBuf, KerBuf, CtrlBuf, 0);
    ...
}

/* Free the cache buffer when you no longer need it */
imBufGetField(Thread, CtrlBuf, IM_CTL_CACHE_BUF, &CacheBuf);
imBufFree(Thread, CacheBuf);
```

Multiprocessing

There are several levels of parallelism on a Genesis system. For example, there are multiple processors within a node: a master processor (MP), four parallel processors (PPs), and an optional accelerator (the NOA). In addition, a system can consist of many processing nodes connected by the grab port, VMChannel, and PCI bus.

There are two main reasons why you would want to exploit the parallelism of your system:

- It might be the only way to achieve a certain task (for example, to process an image while maintaining real-time acquisition).
- It might be required to achieve a certain task fast enough. For example, you might need several processing nodes working together to achieve the performance you require.

Multiple threads

For your application to exploit the parallelism of your system, you need to allocate multiple threads on the Genesis board, so that operations can run in parallel. Recall that, in the Genesis Native Library, an operation is sent to a thread, and the operation executes on the node associated with this thread. Threads execute independently of one another, allowing operations to run in parallel, within and/or across nodes.

In general, you should use as many threads as you need to exploit the parallelism of your system, but no more. Using too many threads can make your application less efficient because, among other things, you will need extra synchronization functions.

Multiple threads within a node

When a function is sent to a node to execute, it will normally use as many resources in that node as it needs to run at maximum speed. However, you can use *imThrControl()* to specify that functions sent to a thread use no more than a specified number of PPs, or that they not use the NOA (if available). This can be useful when you want your application to execute several processing threads simultaneously and you have only one node. Note, however, that there is usually no point in executing two functions at the same time if they both use the same processors. For example, if you have allocated two threads on a node and have limited each to two PPs, then two functions will run at the same time, but each at only half speed, so there is no net gain in performance. This is true whether the functions are compute bound (each runs at half speed because it only has half of the processors) or I/O bound (each runs at half speed because it only gets access to memory half of the time).

If two functions use different processors (for example, the PPs and the MP, or the PPs and the NOA), there will still be no advantage to executing them in parallel if both are I/O bound, since both still access memory only half of the time. However, if one or both functions are compute bound, there can be some advantage to running them in parallel.

An application that can benefit from using two threads on the same node is one that processes an image while copying the previously-processed image to the display. If the processing and copy commands were sent to the same thread, they would run serially, so the total execution time would simply be the processing time plus the copy time. If the processing and copy commands were sent to different threads and properly synchronized, the copy command would still only begin when the processing completes, but it would not prevent processing of the next image from starting. Therefore, part or all of the copy time is "hidden", depending on how I/O intensive the processing functions are. For details on implementing such an application, see the examples in the \GENESIS\EXAMPLES directory.

Host threads

When your application contains several distinct parts that you want to run in parallel, it is often easier to design it so that each part is controlled by a separate thread (or task) on the Host. For example, if you have two processing tasks that run in parallel, it is easier to have each controlled by a separate Host thread.

Note that Host threads are only supported by multi-tasking operating systems.

Host threads and synchronization

When you use Host threads and need to synchronize, you can use either the Host synchronization services (such as Windows NT event objects) or the Genesis synchronization functions. Note, however, that the Host synchronization services cannot be used when you want to synchronize on-board events that are running asynchronously with respect to the Host (this is likely to be often).

When using the Genesis synchronization functions, you can refer to the OSB states using the defines `IM_NON_SIGNALLED` and `IM_SIGNALLED`, instead of `IM_WAITING` and `IM_COMPLETED` (to comply with Windows NT conventions).

Multiple nodes

Various Genesis hardware and software features allow you to make maximum use of the multiple nodes in your system. For example:

- Grabbed images are broadcast to all nodes in a system. Each node can take the whole image or only part of it.
- The VMChannel connects all nodes, so that results are easily sent to a specific node for display or further processing.
- Each 'C80 can make random accesses to any other node over the PCI bus. This is normally used for message passing and sharing small amounts of data.
- You can execute functions on any node in the system.

In accordance with the above, there are different ways you can divide an application between nodes:

- Let each node grab a different part of the same input frame, and work only on that (see the *Grabbing part of the same frame* section below).
- Let each node grab and process a complete frame. Each successive frame goes to a different node (see the *Grabbing successive frames* section below).
- Dedicate one node to grabbing, and let it do the first part of the processing before passing the partial results on to the next node in the pipeline.

Any combination of the above methods can also be used. Which method is best depends on the individual application.

Examples

Multi-node examples can be found in the
 \GENESIS\EXAMPLES directory.

Grabbing part of the same frame

Letting each node grab a different part of the same input frame results in the lowest possible latency before an output image is produced. However, it is not suitable for algorithms that need access to a whole image. For example, to count the number of blobs in an image, you should not sub-divide the image and then sum the results (some blobs might be counted more than once).

To grab part of the same input frame to multiple nodes simultaneously, you need to use the `IM_CTL_START_X`, `IM_CTL_START_Y`, `IM_CTL_STOP_X`, and `IM_CTL_STOP_Y` fields of `imDigGrab()`. In addition, you need to synchronize the nodes to ensure that they all grab the same frame. See Chapter 10 for details.

Grabbing successive frames

When each node grabs and processes successive frames, the latency will be longer than when each node grabs a different part of the same input frame. However, performance scales linearly with the number of nodes for almost any algorithm (that is, if you have n nodes, your application will run n times faster using this method).

To grab successive frames to different nodes, you must wait for the start of the previous grab before trying to grab the next frame (if you wait for the end of the previous grab, it will be too late to grab the next frame; if you do not wait at all, you might grab the same frame). For example:

```
/* Grab a frame on the first node */
imDigGrab(Thread1, 0, Camera, Buf1, 1, 0, OSB);

/* Make the second node wait for the grab to start */
imSyncThread(Thread2, OSB, IM_STARTED);

/* The second node grabs the next frame */
imDigGrab(Thread2, 0, Camera, Buf2, 1, 0, 0);
```

Note that the above assumes that *Thread1* and *Buf1* were allocated on the first node, and *Thread2* and *Buf2* on the other node.

Programming tips

The following are some miscellaneous tips that will help your application run at maximum speed:

- Don't use a display buffer as the destination of a processing function; it is much better to use a buffer in processing memory, and then copy that buffer to display memory when results need to be seen.
- Avoid large look-up tables; they are slow. If you need to perform a LUT mapping on 16-bit data, try to use several smaller LUT mappings instead.
- Avoid dividing an image by a constant or another image; this is a slow operation. If you are dividing by a constant, it is better to invert the constant and multiply (making sure to use enough bits to avoid loss of precision).
- Use only those control options that you need; some functions run slower if you select certain options (such as saturation and clipping of results).
- Avoid synchronous functions within time-critical loops; they break the processing pipeline because the Host has to wait for a reply from the board. Since the allocation functions are synchronous, you should allocate all resources at the beginning of your application.
- Don't add fields to a buffer inside a loop. This can cause a significant overhead, especially if you are processing small images.
- If possible, pack small images into a single large buffer and process them all at once (it is more efficient to process large buffers than small ones).
- If your application requires that one or more regions of interest (ROIs) be processed, it might be more efficient to process the whole image than to define and process several ROIs. This is because there is less overhead involved.
- When retrieving blob analysis or pattern matching results, retrieve a group of results rather than retrieving results individually.

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

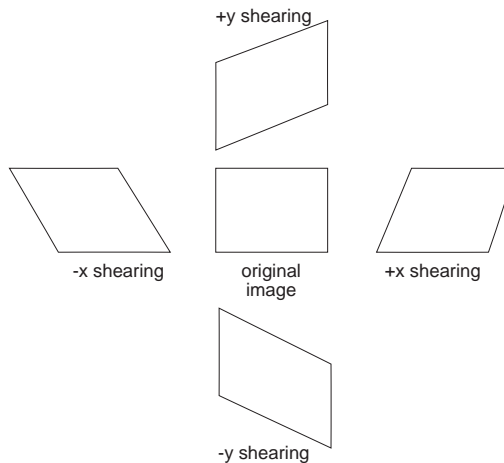
Describes a board whose configuration is designed to include additional modules, if desired. The Genesis main board, for example, can include a display section and/or grab module. In addition, one or more processor boards can be added to increase performance.

■ 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 = %d Load TIFF file, compress, and save JPEG\n", COMPRESS);
        printf("      %d 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 */
    imBufAllocId(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 123–124, 126
- allocating
 - blob analysis feature list 97
 - blob analysis result buffer 97
 - camera definition files 33, 184, 208
 - child buffers 169
 - control buffers 168
 - data buffers 29, 31–32, 165
 - digitizer 184
 - display 31
 - Host memory 166
 - JPEG buffer 140, 144
 - nodes 27, 29, 38
 - off-screen display memory 227–228
 - on-screen child buffers 213, 228
 - operation status block 36
 - pattern matching result buffer 119
 - resources 38, 247
 - threads 27, 29
- analog cameras 199–200
- annotating images 175
- applications
 - checking for errors 230–232, 234
 - color 194, 213–214
 - dividing between nodes 245
 - dual-screen 217–218, 228
 - estimating performance 237
 - latency 246
 - MIL vs Native Library 14
 - monochrome 213–214
 - multiple 38, 208, 228
 - optimizing 236
 - porting 15
 - programming tips 247
 - real-time 22, 38, 83–84, 188, 197
 - single-screen 217–218, 228
 - typical steps 26
- arithmetic and logical operations
 - deriving opcodes for 81
 - divisions 247
 - with three operands 80–81

- asynchronous functions 29, 35–36, 230, 238
- asynchronous grab mode 84, 188
- asynchronous-reset cameras 206

B

- benchmark information 237
- BGR/BGRa images 174
- bicubic interpolation 91
- bilinear interpolation 91
- binary buffers 44–45, 77, 173–174, 236
- binary template matching 60, 65
- bit planes, protecting 175
- blanking period 200
- blob analysis
 - acquiring an image for 99
 - adjusting controls 97, 101
 - assigning label values 104
 - background 96
 - binary features 96
 - blob identifier image 96
 - calculating 97
 - copying results 97, 112
 - defined 96
 - example 98
 - excluding blobs 97
 - feature list 97
 - features 106
 - freeing buffers 97
 - grayscale features 96
 - grouping results 101, 104
 - identifying blobs 101
 - non-square pixels 102–103
 - number of blobs 97, 112–113
 - number of Feret angles 101
 - pixel aspect ratio 101–103
 - result buffer 97, 101
 - runs 101, 115
 - segmentation 97, 99
 - selecting features 97
 - steps 97
 - time slice 102
 - timeout period 101, 105
 - transferring results 97, 247
- blob.c program 270

blobs

- area 106
- assigning label values 104
- breadth 108
- center of gravity 110
- central moments 110
- compactness 109
- convex perimeter 107, 109
- counting 97
- defining 96, 101
- deleting 111
- dimensions 107
- excluding 97, 110–111
- features 106
- Feret diameter 107–108
- filling 104
- grouping 101, 104
- identifying 96, 99
- length 107–108
- location 106, 110
- long, thin 108
- moments 110
- number of holes 109
- number of runs in 115
- ordinary moments 110
- perimeter 106, 109
- processing 111
- removing 111
- separating from image 96, 99
- shape 106, 109
- size 109
- spurious 99
- touching borders 110–111
- touching each other 99
- unwanted 100, 111
- width 107

board

- Genesis-LC 20
- main 16, 18–19, 21, 171
- processor 16, 18–19, 21, 171

brightness, adjusting on grab 198

broadcasting to all nodes 245

buffers

- 1d 165
- 2d 165
- allocating 29, 165
- binary 44–45, 77, 173, 236
- child 29, 42, 47, 74, 169

- color 42, 74, 165, 175
- control 30, 164, 167–168
- converting data type 166
- copying 31, 164, 170
- creating 164, 182
- data types 236
- display 169, 247
- fields 30, 167–168
- freeing 165
- mapping to Host 164, 180
- memory location of 166
- multi-band 42, 74–75, 165, 170
- packed 47, 247
- parent 169
- pitch 180
- RGB packed 45
- single-band 165
- supported data types 166
- tag 29, 47, 164, 173, 186
- transfer to/from a file 179
- transfer to/from the Host 179

byte-aligned buffers 173

C

C80 16, 18–19, 31

- and multiple nodes 245
- block diagram 43
- code 21
- multiple 19
- porting 15
- programming 43, 236

camera definition files

- allocating 33, 184, 208
- changing 184, 192
- compatible 83, 190–191
- creating 33
- that produce non-square pixels 103

camera settings

- frame size 200
- gain levels 198
- input channel 193–194
- LUT 198–199
- reference levels 198
- synchronization channel 197
- triggers 202
- user bits 201

- cameras
 - analog 199–200
 - asynchronous-reset 206
 - color 182, 193
 - digital 200
 - line scan 201
 - monochrome 182, 193–195, 197
- cellular mapping 71
- certainty level 123, 126, 129
- Chamfer 3-4 transform 67
- Chessboard transform 66
- child buffers 29, 42, 47, 74, 169
 - on the display 31, 212–213, 221, 228
- City Block transform 66
- clearing error information 230, 232
- clipping 50, 247
- color buffers
 - copying 75
 - copying to display 175
 - processing 42, 74
- color space
 - HSL 74–75
 - RGB 74–75
- command queues 85
- compressing images
 - achievable ratios 143
 - an example 145, 149–151
 - controlling 145, 147
 - of JFIF files 143
 - overview of algorithm 147
 - saving tables during 151
 - steps 140, 144, 222
 - to open files 151
 - transmitting 153
 - when image is large 150
 - with restart markers 153
 - with your own table 149
- compute bound functions 58, 240, 243
- connected region, filling 162
- connectivity mapping 71
- contiguous physical memory 182
- continuous grab 34, 187, 192, 208
 - an example 34
- contrast, adjusting on grab 198
- control buffers 30, 164, 167–168
- control fields 30, 167–168, 247

- converting
 - between data types 45
 - RGB to HSL 75
- convex perimeter 107
- convolutions 58–59
- copy
 - 16-bit color images 175
 - available functions 170
 - between nodes 170
 - between on-board and Host 170
 - between processing and display 170
 - blob results 97, 112
 - bottom to top 175
 - color images 75
 - continuously 177
 - control fields 168
 - extracting bytes during 174
 - multi-band buffers 75, 170
 - over the PCI bus 170–171
 - over the VMChannel 170–171
 - pattern matching models 130
 - rectangular region 177
 - reducing overhead during 176
 - RGB images 75
 - RGB555/565 images 175
 - right to left 175
 - swapping bytes during 174
 - to non-rectangular regions 173
 - to rectangular regions 169
 - to the display 174–175, 177
 - to/from a VM stream 178
 - with a specific VIA 176
 - with subsampling 174
 - with tag 164, 173
 - with VIA options 171
 - with write masks 175
 - with zoom 174
- counting blobs 97

D

- data paths 18, 171
- data types
 - converting between 45–46
 - mixed 49
 - speed 236
 - supported 44, 166

- decompressing images
 - an example 146, 152
 - controls 145
 - from open files 151
 - restrictions 143
 - steps 141, 144
- development tools 43, 236
- digital cameras 200
- digitizer
 - allocating 184
 - programming 184
- dilation 60–61, 99
- direct memory access 182
- display
 - allocating 31
 - as two-dimensional surface 31, 212
 - block diagram 211
 - buffers 169, 247
 - child buffers 31, 212–213, 221, 228
 - color images 213–214
 - color version of 210, 213–214, 221, 228
 - copying to 174–175
 - dual-screen mode 215–216, 218, 228
 - effects 219, 228
 - grabbing to 221
 - grayscale images 170, 213–214
 - in pseudo-color 220
 - keying 217–218, 228
 - LUTs 210
 - memory 18, 31, 212, 214, 227–228
 - mode 214
 - monochrome version of 210, 213, 221, 228
 - multiple live grabs on 182
 - off-screen buffers 227
 - overview 18, 210
 - panning 219, 228
 - refresh rate 210
 - resolution 210, 216, 219
 - scrolling 219, 228
 - single-screen mode 215–218, 228
 - zooming 219, 228
- display artifacts, avoiding 177
- distance transforms 60, 66
- distortions, geometric 86, 103
- dividing images 247
- DMA transfers 182
- double buffering 83, 176

- drawing
 - an object's outline 160
 - arcs 158
 - lines 158
 - rectangles 158
- dual-screen display mode 215–216, 218, 228

E

- erosion 60–61, 99
- error information, clearing 230, 232
- error messages, printing 232
- error reporting
 - a function 230
 - a thread 230–231
 - and asynchronous functions 230–231
 - application-wide 230–232
 - when to check 35, 234
 - which mechanism to use 231
- estimating performance 237
- event objects 244
- example of
 - an FFT 93
 - basic program 28
 - blob analysis 98
 - blob analysis result transfer 113
 - blob analysis timeout feature 105
 - checking for errors 231
 - compressing images 145, 149–151
 - decompressing images 146, 152
 - defining kernels 69
 - grabbing continuously 34
 - grabbing from multiple channels 196–197
 - grabbing successive frames 83, 246
 - grabbing to two or more buffers simultaneously 190
 - grabbing with timers 207
 - hardware triggers 204
 - keying 217
 - LUT mapping 51
 - LUT warping 90
 - mapping buffer to Host 180
 - pattern matching 120
 - perspective warping 89
 - plotting 161
 - printing error messages 232
 - processing non-rectangular regions 48

- real-time processing 84
- rotating images 87
- software triggers 202
- synchronizing operations 36
- thinning to skeleton 63
- transferring to Host 32

examples, overview 23

exposure signals 204

exposure time 204

extracting bytes

- during copy 174
- during grab 186

F

fast peak finding 135

Feret diameter 107–108

FFTs 92–93

- an example 93

fields (camera), grabbing 187

filling objects 158

first.c program 275

first-order warpings 86, 89

- an example 87
- generating coefficients for 86

flipping images 72

floating-point buffers 45, 72, 166

formatting

- copied data 171
- grabbed data 186

frame buffers 210, 215–217, 219–220

frame size 200

frames, grabbing 187

G

gain levels 198

Genesis Developer's Toolkit 43, 236

Genesis shell 21

Genesis-LC 20

GENKEY utility 218, 228

GENVCFLD utility 216

geometric operations 42

- advanced 72, 86
- basic 72

grab module 18, 185

grab port 19

grab.c program 277

grabbing

- asynchronously 84, 188
- bottom to top 186
- color images 194, 208
- continuously 34, 187, 192, 208, 221
- extracting bytes during 186
- fields 187
- frames 187
- from a specific channel 194
- from interlaced cameras 187
- from multiple channels 195–196
- general steps 184
- line interrupts 189
- lines 187
- low-resolution image 187
- monochrome images 208
- multiple live, on display 182
- part of the same frame 246
- rectangular region 186
- reducing overheads during 188
- right to left 186
- successive frames 83, 246
- to multiple nodes 190
- to non-rectangular regions 221
- to rectangular regions 169
- to the display 208, 221
- to two or more buffers simultaneously 187, 190
- to/from a VM stream 186, 188
- with subsampling 186
- with tag 186
- with VIA options 186
- with write masks 186
- with zoom 186

graphics

- and color buffers 159
- available functions 158
- drawing color 159
- using the MGA 158
- XOR option 159

H

hardware triggers 191, 202–203, 205

- an example 204

- histogram 77
 - plotting right-side up 160–161
 - transferring results 32
- histogram equalization 57
- hit-or-miss transformation 60, 65
- horizontal syncs 197, 200, 205
- Host
 - allocating memory on 32
 - and asynchronous functions 29
 - and command queues 85
 - and synchronous functions 29
 - copying to/from 170
 - halting execution on 36
 - mapping buffers to 164, 180
 - memory 22, 29, 166, 227
 - operating system 30, 158, 215
 - synchronizing with 36, 244
 - threads 244
 - transfer data to/from 19, 26, 31–32, 97, 112, 119, 164, 179
 - transfer rate 16
- hot spot 123, 125
- HSL color space 74–75
- Huffman encoding 147–148

I

- I/O bound functions 49, 237–239, 243
- imAppCatchError() 35, 230, 232
- imAppGetError() 35, 230–232, 234
- imBinConvert() 45, 166, 174
- imBinMorphic() 60, 62–63, 65, 99, 118
- imBinTriadic() 80–82
- imBlobAllocFeatureList() 97
- imBlobAllocResult() 97
- imBlobCalculate() 97, 101, 104–105
- imBlobControl() 97, 101, 103, 105, 107–108, 115
- imBlobCopyResult() 112, 114
- imBlobCopyRuns() 101, 115
- imBlobFill() 101, 111
- imBlobFree() 97
- imBlobGetLabel() 101
- imBlobGetNumber() 112–113
- imBlobGetResult() 112–114
- imBlobGetResultSingle() 112, 114
- imBlobGetRuns() 101, 115

- imBlobInquire() 105
- imBlobLabel() 101
- imBlobSelect() 97, 100, 111
- imBlobSelectFeature() 97, 112
- imBlobSelectFeret() 97, 108
- imBlobSelectMoment() 97, 110
- imBufAlloc() 32, 165, 227–228
- imBufAlloc1d() 50, 165, 227–228
- imBufAlloc2d() 68, 165, 227–228
- imBufAllocControl() 168
- imBufChild() 27, 31, 47, 169, 182, 212–215, 221, 228
- imBufChildBand() 74, 169
- imBufChildMove() 169
- imBufClear() 27
- imBufControl() 182
- imBufCopy() 27, 31–32, 45, 75, 170, 182
- imBufCopyField() 168, 170
- imBufCopyPCI() 31, 44, 47, 170–171, 182, 186
- imBufCopyVM() 31, 44, 47, 170–171, 186
- imBufCreate() 164, 182
- imBufFree() 30, 165, 182
- imBufGet() 32, 77, 114, 179
- imBufGet1d() 32, 77, 179
- imBufGet2d() 32, 179
- imBufGetField() 77, 168
- imBufGetFieldDouble() 168
- imBufGetNextField() 168
- imBufLoad() 179
- imBufMap() 164, 180
- imBufPack() 47–48, 173
- imBufPut() 32, 50, 68, 179
- imBufPut1d() 32, 179
- imBufPut2d() 32, 179
- imBufPutField() 168
- imBufRemoveField() 168
- imBufRestore() 179
- imBufSave() 179
- imCamAlloc() 33, 184
- imCamClone() 192
- imCamControl() 184, 188, 192, 196–197, 201, 208
- imCamInquire() 208
- imDevAlloc() 27, 29, 38
- imDevFree() 30
- imDevInquire() 85
- imDigAlloc() 33, 184

- imDigCapture() 190, 202
- imDigControl() 184, 192, 201, 203, 208
- imDigGrab() 33, 83–84, 178, 186, 196–197, 202, 221
- imDigInquire() 201
- imDispControl() 214, 217, 219–220, 227–228
- imFloatConvert() 45, 166
- imGen1d() 50
- imGenWarp1stOrder() 86
- imGenWarp4Corner() 89
- imGenWarpLutMatrix() 89
- imGraArc() 158
- imGraArcFill() 158
- imGraFill() 162
- imGraLine() 158
- imGraPlot() 158, 160
- imGraRect() 27, 158
- imGraRectFill() 158
- imGraText() 27, 158, 162
- imIntBinarize() 99
- imIntConnectMap() 71
- imIntConvert() 46, 99, 166
- imIntConvertColor() 75
- imIntConvolve() 59, 68, 99
- imIntCorrelate() 118
- imIntCountDifference() 77
- imIntDistance() 66
- imIntDyadic() 49
- imIntErodeDilate() 61, 99
- imIntFFT() 92
- imIntFindExtreme() 77
- imIntFlip() 72
- imIntHistogram() 77
- imIntHistogramEqualization() 57
- imIntLocateEvent() 77
- imIntLutMap() 50–51, 54, 56, 220
- imIntMonadic() 49
- imIntRank() 99
- imIntScale() 73, 219
- imIntSubsample() 72–73
- imIntThickThin() 62–63
- imIntTriadic() 47, 80–82
- imIntWarpLut() 88–89
- imIntWarpPolynomial() 86, 219
- imIntZoom() 72–73, 219
- imJpegAlloc() 140, 144–145
- imJpegControl() 145, 147, 150–151, 153
- imJpegControlBand() 145
- imJpegDecode() 141, 144
- imJpegEncode() 140–141, 144
- imJpegFree() 144
- imJpegInquire() 144
- imJpegPutTable() 145, 149
- imJpegRead() 144, 151
- imJpegReadBuf() 141, 144
- imJpegRestore() 144
- imJpegSave() 144, 146
- imJpegWrite() 151
- imJpegWriteBuf() 141, 144, 146
- imPatAllocModel() 119
- imPatAllocResult() 119
- imPatAllocRotatedModel() 130
- imPatCopy() 130
- imPatFindModel() 119, 133–134
- imPatFree() 119
- imPatGetNumber() 119, 124
- imPatGetResult() 119, 124, 132
- imPatInquire() 130
- imPatPreprocModel() 119, 122, 128–129
- imPatRead() 130
- imPatRestore() 119, 130
- imPatSave() 130
- imPatSetAcceptance() 123
- imPatSetAccuracy() 123, 129, 136
- imPatSetCenter() 123, 125
- imPatSetCertainty() 123, 129
- imPatSetDontCare() 123
- imPatSetNumber() 123–124
- imPatSetPosition() 123, 129
- imPatSetSearchParameter() 123, 132, 134–135
- imPatSetSpeed() 123, 128–129
- imPatWrite() 130
- imSyncAlloc() 36
- imSyncControl() 36
- imSyncGetError() 230–231
- imSyncHost() 36, 38, 84–85, 189, 191
- imSyncThread() 85, 189
- imSysClock() 237
- imThrAlloc() 27, 29
- imThrControl() 243
- imThrFree() 30
- imThrGetError() 35, 230–231, 234
- imThrHalt() 33, 177, 187
- input channels 193–194
- integer buffers 44–46, 158, 174

interlaced scanning 187, 189
interpolated LUT mappings 55–56
interpolation modes 86, 91

J

JPEG compression

- achievable ratios 143
- an example 145, 149–151
- by blocks 150
- controlling 145, 147
- of JFIF files 143
- overview of lossless algorithm 147
- restart markers 153
- saving tables during 151
- steps 140, 144, 222
- to open files 151
- transmitting compressed images 153
- with large images 150
- with your own table 149

jpeg.c program 279

K

kernels

- an example 69
- center coordinates 68
- custom 58–59, 68
- predefined 58

keying 217–218, 228

- an example 217

L

level-sensitive trigger 203

line interrupts 189

line scan cameras 201

live grabs, multiple 182

loops 38, 234, 247

LUT

- in the grab section 198–199
- in the RAMDAC 50, 210, 220
- large vs small 247
- mappings 50–51, 55–56, 220
- warpings 86, 88

M

main board 16, 18–19, 21

- block diagram 16, 18, 171

main frame buffer 210, 215–219

mapping buffer to Host 180

- an example 180

mapping through LUT

- displayed images 220

- grabbed images 198

match peaks 135

match score 123, 128, 132, 136

memory

- display 18, 31, 33, 166, 212, 227–228

- freeing 144

- Host 29, 32, 164, 170, 227

- invalid access 50

- linear 16, 227

- main frame buffer 210

- off-screen 227–228

- operation status block 36

- overlay frame buffer 210

- physical 182

- processing 16, 18, 21, 29, 31, 33, 166,
212, 227

- virtual 182

memory banks

- defined 194

- how connected 171

merging images 47, 80

message passing 245

MGA 210, 215–216

MIL vs Native Library 14–15

model (pattern matching)

- complex 128

- copying 130

- creating 119, 121

- defined 118

- don't care pixels 123, 127

- effective center 123, 125

- efficient 129

- freeing 119

- hot spot 123, 125

- inquiring about 130

- large features 122

- masking 123, 127

- orientation 121

- reading from open file 130
- restoring 119, 122, 130
- rotating 130
- saving 122, 130
- size 121, 129
- small-scale features 122
- writing to open file 130
- model images 121, 127
- monochrome cameras 182, 194–195, 197
- morphological operations 58, 60
- multi-band buffers 42, 74, 165
 - copying 75, 170
 - processing 74
- multiple live grabs, on display 182
- multi-processing 22, 36, 236, 242

N

- Native Library
 - overview 14
 - vs MIL 14–15
- nearest-neighbor interpolation 91
- neighborhood operations 16, 42, 48, 58, 70
- NOA 16, 18, 21, 243
- nodes
 - allocating 27, 29, 38
 - broadcasting to 245
 - defined 21
 - multiple 177–178, 187, 245
- non-square pixels 101–103, 200
- normalized grayscale correlation 131

O

- objects, filling 158
- open files 130, 151
- operating system 30, 158, 215
- operations, synchronizing 36
- optimize.doc file 237–240
- OSB
 - allocating 36
 - and Windows NT 244
- overheads 237–238
 - reducing, when copying 176
 - reducing, when grabbing 188
- overlay frame buffer 210, 215–220, 228
- overscan pixels 70

P

- packed buffers 47, 247
- packed color images 174
- panning the display 219, 228
- parallel processing 22, 36, 236, 242
- parent buffers 169
- part-present sensor 204, 206
- pat.c program 282
- pattern matching
 - acceptance level 123–124
 - adjusting parameters 119, 123
 - algorithm 131
 - certainty level 123, 126, 129
 - controlling search 123
 - copying models 130
 - creating model 119, 121
 - defined 118
 - example 120
 - false matches 121, 127
 - fast peak finding 135
 - masking model 123, 127
 - match peaks 135
 - match score 123
 - model 118, 129
 - number of matches 123–124
 - positional accuracy 123, 126, 129
 - preprocessing 119, 122, 129
 - reading models from open file 130
 - resolution level 133
 - restoring models 119, 122, 130
 - result buffer 119
 - returned coordinates of 123, 125
 - rotating models 130
 - saving models 122, 130
 - search in one direction 126
 - search parameters 119
 - search region 123, 125, 129
 - search speed 123, 128–129
 - searching for model 119
 - speeding up 129
 - steps 119
 - supported buffers 118
 - target image 118, 122
 - transferring results 119, 247
 - uses 118
 - writing models to open file 130

- PCI bus, copying over 19, 170–171, 245
- performance, estimating 237
- perspective warpings 86, 88–89
- physical memory 182
- pitch 180
- pixel aspect ratio 101–103
- pixel clock 200
- plotting 158, 160
 - an example 161
- point-to-point operations 42, 48–49
- polynomial warpings 86, 89
 - an example 87
- positional accuracy 123, 126, 128–129
- predictive coding 147
- preprocessing models 119, 122, 128–129
- printing, error messages 232
- process.c program 286
- processing
 - a band of a buffer 164
 - color buffers 42, 74
 - non-rectangular regions 29, 47–48
 - rectangular regions 29, 47, 164, 169, 247
- processing memory 16, 18, 29, 31, 166, 212, 227
- processing operations 42, 166
- processor board 16, 18–19, 21
 - block diagram 19, 171
- programming
 - 'C80 43, 236
 - digitizer 184
- programming tips 247
- progressive scanning 187
- protecting bit planes 175
- pseudo-color effect, achieving 199, 220

R

- RAMDAC 210
- reading from open files
 - compressed images 151
 - models 130
- reading results 168
- real-time processing 22, 83–84
 - an example 84
- reference levels 198
- refresh rate 210
- replace overscan 70

- resizing images 86
- resolution levels 133
- resolution, of display 210, 216, 219
- resources 21, 29, 38, 247
 - freeing 27, 30
- restart markers 153
- restoring
 - compressed images 145
 - models 122, 130
- reversed images 175, 186, 188
- RGB color space 74–75
- RGB packed buffers 45
- RGB555/565 formats 175
- rotating images 72, 86–87
 - an example 87
- RS422 format 201, 203, 205
- runs in a blob 101, 115

S

- sampling frequency 200
- saturation 49–50, 247
- saving
 - compressed images 144–145, 151
 - models 122, 130
- scaling images 72–73
- screen tearing, avoiding 177
- scrolling the display 219, 228
- SDRAM 18, 31, 212
- search parameters 123
- segmentation 96, 99
- shearing images 86
- single-band buffers 165
- single-screen display mode 215–218, 228
- software triggers 191, 202
 - an example 202
- spatial filtering operations 58–59
- spatial patterns, locating 92
- square pixels 101–103
- statistical operations 42, 77
- subsample
 - copied data 174
 - grabbed data 186
 - images 72–73
- swapping bytes, during a copy 174
- synchronization channel 197

- synchronizing
 - an example 36
 - grabs 190
 - Host threads 244
 - operations 36
 - threads 36
- synchronous functions 29, 32, 35, 230, 238, 247
- systems 21, 171

T

- tag buffers 29, 47, 164, 173–174, 186, 227
- target images 118, 122, 128
- text, writing 158, 162
- tfilter.c program 300
- thickening 60, 62–63
- thinning 60, 62–63
 - an example 63
- threads
 - allocating 27, 29
 - checking for errors 230
 - defined 22
 - multiple 242–243
 - on the Host 244
 - synchronizing 36
- timers, on the grab 202, 204
 - an example 207
- transfer to/from a file, buffer data 179
- transfer to/from the Host
 - achievable rates 16
 - an example 32
 - blob results 97, 112, 247
 - buffer data 19, 32, 138, 164, 179
 - pattern matching results 119, 247
- translating images 86–87
- transmitting compressed images 153
- transparent overscan 70
- triggers
 - hardware 191, 202–203, 205
 - software 191, 202
 - sources 202
 - timers 202, 204
- TTL format 201, 203, 205

U

- user bits 201

V

- vertical syncs 197, 201, 205
- VIA options
 - during a copy 171
 - during a grab 186
- virtual memory 182
- visible artifacts, avoiding 177
- VM device 18, 178, 186
- VM stream 178, 186
- VMChannel
 - and multiple nodes 245
 - copying data over 18, 170–171

W

- warping
 - an example 87
 - first-order 86, 89
 - generating coefficients for 86
 - how performed 86
 - interpolation modes 91
 - perspective 88
 - through a LUT 88, 90
- Windows NT 30, 244
- WRAM 18
- write masks 175, 186
- writing a rectangular region 177, 186
- writing text 158, 162
- writing to open file
 - compressed images 151
 - models 130

Z

- zoom
 - copied data 174
 - display 219, 228
 - grabbed data 186
 - images 72–73, 219

Product Support

Product Assistance Request Form

[illegible]

