

Matrox Imaging Library

version 6.1

User Guide

Manual no. 10513-301-0610

March 1, 2000

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

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

PC/104-Plus™ is a trademark of the PC/104 Consortium.

CompactPCI™ is a trademark of PCI Industrial Computer Manufacturers' Group.

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

Texas Instruments is a trademark of Texas Instruments Incorporated.

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: Getting started19

The MIL package	20
MIL and the Intel MMX™/SSE™ technologies	24
System requirements	25
Getting started	26
Installation	27
Building an application	29
Distributing your MIL application	33
Distributing MIL run-time DLL files with your software	33
Obtaining a run-time license	34

Chapter 2: Allocating an image buffer and grabbing images37

Getting started	38
Allocating and displaying an image buffer	39
Grabbing images	42

Chapter 3: Image processing45

Image processing	46
The MIL package	47
Steps to performing a typical application.	48
A typical application	48

Chapter 4: Improving your images 53

Image quality	54
Techniques to improve images	55
Averaging an input sequence	56
Applying spatial filters	57
Opening and closing.	58
Basic geometrical transform.	59

Chapter 5: Image manipulation 61

Image manipulation	62
Image statistics	62
Generating a histogram	63
Finding the image extremes	65
Projecting an image to one dimension	65
Thresholding your images	66
Binarizing	66
Clipping.	68
Histogram equalization.	69
Accentuating edges	70
Edge enhancers	71
Edge detection.	71
Arithmetic with images.	74
Combining images	74
Mapping an image	76
Erosion and dilation.	77
Distance transform	81
Labeling.	82

Chapter 6: Advanced image processing85

Advanced image processing	86
Custom spatial filters	86
Custom morphological operations	90
Erosion and dilation	91
Thinning and thickening	96
Matching	98
Searching for hits or misses	98
Connectivity mapping	99
Fast Fourier Transform	100
Watershed transformations	104
Using watersheds to separate touching objects	105
Using watersheds to separate objects from their background	106
Minimum variation between extrema	107
Using marker images	108
Style of the watershed lines	109
Skipping the last level	110
Polar-to-rectangular and rectangular-to-polar transform	111
Warping	113
First-order polynomial warpings	114
Using LUTs to perform a warping	114
Interpolation modes	116
Points outside the source buffer	117
Discrete Cosine Transform	118

Chapter 7: Calibration. 119

Introduction.	120
Types of distortions	121
Steps to getting results in real-world units . .	122
Transforming coordinates or results	122
Physically correcting an image.	122
Automatically getting results in real-world units	124
Calibrating your imaging setup	126
Real-world grid	126
List of coordinates	128
Calibration modes	129
Coordinate systems and camera position . . .	130
Multiple fields of view.	133
Single camera fixed on a manipulator: Relative camera position example	133
Single camera and moveable object: Relative coordinate system example.	134
Several cameras and fixed object: Relative coordinate system example.	135
Processing calibrated images	136

Chapter 8: Blob analysis 137

Blob analysis	138
MIL and blob analysis	139
Steps to performing blob analysis.	139
A simple blob analysis example	142
Blob reconstruction	145

Chapter 9: Setting up for blob analysis147

Identifying blobs	148
Adjusting blob analysis processing controls. .	150
Controlling the image lattice	150
The pixel aspect ratio	151
Setting the Blob identification mode	152
Selecting blobs	153

Chapter 10: Analyzing the blobs155

Making feature extractions	156
The area and perimeter	158
Dimensions.	160
Determining the shape	162
Finding the blob location.	166
Moments.	169
Location, length and number of runs.	170

Chapter 11: Pattern matching171

Pattern matching	172
Simple alignment techniques.	173
Vertical and horizontal alignment	173
Angular alignment	177

<i>Chapter 12: Models, searches, and search parameters</i>	<i>181</i>
Performing a search	182
Rotation	188
Setting the angle of search	190
Determining the rotation tolerance of a model	191
Masking the model	192
Search parameters	194
Specifying the number of matches	194
Setting the acceptance level	194
Setting the certainty level	195
Redefining the model's reference position	196
Selecting the search region	197
Positional accuracy	198
Setting the speed parameter	199
Preprocess the search model	199
Speeding up the search	200
Choose the appropriate model	200
Adjust the search speed parameter	200
Effectively choose the search region and search angle	201
Searching for multiple models at the same time	201

The pattern matching algorithm (for advanced users)	202
Normalized Correlation	202
Hierarchical Search	204
Search Heuristics	206
Sub-pixel accuracy	206
<hr/>	
<i>Chapter 13: Optical character recognition</i>	<i>209</i>
The MIL OCR module	210
Steps to reading or verifying a string in an image	211
A typical application	213
Using fonts	216
Calibrating fonts	217
Setting character constraints	219
Setting processing controls	220
Managing fonts	222
Saving and restoring a font	222
Inverting a font	222
Inquiring about a font	222
Visualizing a font	222
Creating custom fonts	224
Speeding up the read or verification operation.	228

Chapter 14: DataMatrix and bar codes 229

Introduction	230
General steps	231
Controlling read operations	231
Controlling write operations	234

Chapter 15: Measurements 235

The measurement module	236
Markers	237
A multiple marker	237
Steps to finding and obtaining measurements of markers	238
A measurement example	241
Measurement box.	244
Search algorithm	247
Marker characteristics	248
Edge markers: fundamental characteristics .	249
Edge markers: advanced characteristics. . .	251
Stripe markers: fundamental characteristics	256
Stripe markers: advanced characteristics . .	259
Multiple marker characteristics	261
Measurements between two markers	262
A measurement example	264

Chapter 16: Specifying and managing your data buffers	269
Data buffers	270
Target system	271
Specifying the dimensions of a data buffer. . .	271
Data type and depth	272
Attribute	273
Manipulating and controlling certain data buffer areas.	277
Child buffers.	277
Copying specific buffer areas	278
Managing data buffers	279
Controlling how color image buffers are stored	281
RGB buffers	282
Binary buffers	284
YUV buffers.	284
YUV16 Packed	285
YUV9 Planar	286
YUV12 Planar	286
YUV16 Planar	287
YUV24 Planar	288
Child YUV buffers.	288
Accessing a MIL buffer directly	289
Mapping a data buffer to user-allocated memory.	290
Pixel conventions	293

Chapter 17: Lookup tables 295

Lookup tables	296
LUTs and data buffers	297
Loading and generating data into LUTs	297
Generating data directly into the LUT buffer	297
Loading LUTs with precalculated data . . .	298
Using LUTs	299
Processing using LUTs	299
Displaying using LUTs	299
LUTs and digitizers	300

Chapter 18: Displaying an image 301

Displaying an image	302
Display configuration	303
Single-screen configuration	303
Dual-screen configuration	303
Multi-head display configuration	304
Display modes and the display window	305
Displaying in windowed-mode	305
Displaying in non-windowed mode	305
Display size and depth	306
Displaying buffers of different data depths	306
Removing a buffer from the display	307
Displaying multiple buffers	308
Panning, scrolling, and zooming	311

Annotating the displayed image non-destructively	312
Using GDI annotations	314
Displaying an image in a user-defined window	317
Using the MdispSelectWindow() function.	317
LUTs and changing the displayed colors or gray levels.	321
Changing the default LUT values	322
Different display architectures in windowed mode.	324
Underlay display architecture	325
Overlay/regular display architecture.	326
DirectDraw underlay-surface display architecture	327
Advanced controls for windowed mode	328
Display types in windowed mode.	328
Zoom types in windowed mode	329
Controlling how the LUT buffer is loaded into the Windows palette.	330
Controlling how the logical palette is loaded into the physical output LUTs.	330
<hr/>	
<i>Chapter 19: Generating graphics</i>	<i>333</i>
MIL and graphics	334
Preparing for graphics.	334
Drawing graphics	336
Writing text	338

Chapter 20 : Grabbing with your digitizer. 339

Cameras and input devices	340
The data format	341
The digitizer number	342
Multiple cameras	342
Number of frames or fields	343
Grabbing to the display	344
Live and pseudo-live continuous grabs. . .	344
Live transfer to the display.	345
Pseudo-live transfers to the display	345
Window occlusion	347
Reference levels, lookup tables, and scaling	349
Black and white reference levels	349
Color image reference levels.	350
Mapping grabbed data through a LUT . . .	350
Scaling	351
Optimizing application performance when grabbing	352
Grab mode	352
Double buffering	353
Multiple buffering	355
Grabbing a sequence of frames in real-time	356
Grabbing with triggers and exposures	356
Asynchronous reset mode	357
Triggers and exposures	358
Software triggers	361

Auto-focusing	361
Search strategies	362
<hr/>	
<i>Chapter 21: Color</i>	367
Dealing with color	368
Grabbing.	368
Displaying.	370
Processing.	371
Saving and loading color images	374
<hr/>	
<i>Chapter 22 : JPEG compression</i>	375
Introduction	376
General steps	377
Controlling a JPEG compression	378
JPEG lossless	378
JPEG lossy	379
Using your own table	380
Restart markers	381
<hr/>	
<i>Chapter 23: Data manipulation with multiple systems</i>	383
Data manipulation with multiple systems	384
<hr/>	
<i>Chapter 24: Using MIL with multi-processing and under multi-thread systems</i>	385
Multi-processing	386
Multi-threading.	387
MIL and multi-threading.	388

Chapter 25: Using MIL with Native Mode

Functions 397

 Integrating native functions with MIL code . . 398

 Portability 398

 Signaling MIL about Native Mode use. . . 398

 A native mode example. 399

Index

Product Support

Note: For detailed information about MIL functions, see **MIL Command Reference**.

Note: For information about using MIL with your specific board, see **MIL/MIL-Lite Board-Specific Notes**.



Chapter 1: Getting started

This chapter presents the features of the Matrox Imaging Library package. It also explains the installation process, how to run a Matrox Imaging Library application program, and how to distribute your MIL application.

The MIL package

The Matrox Imaging Library (MIL) package is a hardware-independent, modular 32-bit imaging library. It has an extensive set of commands for image processing and specialized operations such as blob analysis, calibration, bar and 2D code reading/writing, measurement, pattern recognition, and optical character recognition operations. It also supports a basic graphics set. In general, MIL can manipulate binary, grayscale, or color images.



The package has been designed for fast application development and ease of use. It has a completely transparent management system and entails virtual, rather than physical, data object manipulation, allowing for platform-independent applications. This means that a MIL application can run on any VESA-compatible VGA board or Matrox imaging board under different environments (that is, Windows 98/NT/2000). MIL uses the notion of systems to identify boards, and more than one board can be controlled by a single application program. MIL is capable of running solely with the Host CPU, but can take advantage of specialized accelerated Matrox hardware if it is available and is more efficient.

Image acquisition

Images can be loaded from disk or acquired from the wide range of supported input devices (if hardware permits) and can be stored in your platform's storage area. Sequences of images can also be loaded and saved in .avi format.

Compression

MIL allows you to compress and decompress images. MIL can compress images using the JPEG lossless or JPEG lossy algorithm.

Image processing capabilities

You can smooth, accentuate, qualify, or modify selected features of an image using MIL's processing capabilities. These capabilities include point-to-point, statistical, spatial filtering, morphological, Fast Fourier transform, as well as geometric operations which include warping and polar-to-rectangular transformations.

Graphics capabilities

You can annotate or alter images using the basic graphics tools in MIL. MIL has commands to write text, as well as basic graphics commands to draw rectangles, arcs, lines, and dots.

Blob analysis capabilities

The MIL blob analysis capabilities allow you to identify and measure connected regions (commonly known as blobs or objects) within an image.

The blob analysis module can measure a wide assortment of blob features, such as the blob area, perimeter, Feret diameter at a given angle, minimum bounding box, and compactness. It can also be used to perform some image processing operations such as reconstructing or eliminating blobs.

Measurement capabilities

The MIL measurement module allows you to find sets of image characteristics or "markers" in an image, based on differences in pixel intensities. Upon finding a marker, the module returns the marker's spatial reference position and measures such features as its width and angle. The module can also take measurements between two markers.

Pattern recognition capabilities

The MIL pattern recognition capabilities can help solve machine vision problems such as alignment, measurement, and inspection of objects. These capabilities include finding:

- The coordinates of a pattern (referred to as a model) in a target image.
- The number of occurrences of a model in a target image.

In some cases, the orientation of the model and/or the target image can play a factor in the search operation. MIL includes a number of options to deal with such cases, each optimized for different image and background conditions. For example, you can find the orientation of a target image or you can search for occurrences of a model at any angle, in the target image.

Optical character recognition capabilities

The MIL optical character recognition (OCR) module provides a powerful and easy to use function set for reading and verifying character strings in grayscale images, providing results such as quality scores and validity flags.

Bar code capabilities

MIL allows you to read and write 2D codes, as well as several types of bar codes.

Calibration capabilities

MIL's calibration module consists of a set of functions that allow you to map pixel coordinates to real-world coordinates. This mapping can be used to get results from other MIL modules in real-world units. The mapping can also be used to physically correct an image's distortions. Calibration mappings can compensate for non-uniform aspect ratio, rotation, perspective foreshortening, and other more complex distortions.

Creating your own MIL functions

If the available MIL operations do not provide the required functionality or do not make use of some board-specific feature, you can use the MIL Developer's Toolkit to directly access your target system's driver functions through native mode and/or to create your own pseudo-MIL functions. Note, although entering native mode can be useful, you should be aware that the resulting application will not be portable to other Matrox platforms supported by the MIL package. The MIL Developer's Toolkit is described in the *Matrox Imaging Library Command Reference* manual.

MIL objects

MIL handles physical objects (systems, digitizers, displays, and data buffers) as virtual objects. These virtual objects must be allocated before you can manipulate them and must be released when they are no longer required. For simple applications, you seldom need to allocate these objects individually, since those set up by default (*MappAllocDefault()*) generally meet your application needs.

Image pixel depth

The MIL package can:

- Grab up to 16-bit grayscale images, or color images.
- Process 1, 8, 16, and 32-bit integer or floating point images.
- Process color images depending on the operation. Each band of a color image is processed individually, one after the other. Statistical, blob analysis, measurement, pattern matching, optical character recognition, and code operations do not support color processing.
- Display 1, 8, or 16-bit grayscale or color images (if the platform supports it).

*MIL documentation's
word usage*

All the MIL documentation uses the words *function* and *command* interchangeably, since most of the commands in MIL are C functions. *Digitizer* and *frame grabber* are also used interchangeably. Finally, in general, *host* refers to the principal CPU in one's computer while *system* refers to your Matrox imaging board and its associated resources.

Command descriptions

Descriptions of the individual commands are found in the *Matrox Imaging Library Command Reference* and the *MIL/MIL-Lite Board Specific Notes* manuals.

MIL and the Intel MMX™/SSE™ technologies

MIL's processing operations have been optimized, in assembly language, to take advantage of Intel MMX™ acceleration and Streaming SIMD Extensions (SSE).

MMX™

Intel MMX™ Technology, an extension to the Intel architecture, is designed specifically to accelerate multimedia (and multimedia-like) applications. Intel MMX™ Technology is built to handle computation-intensive algorithms that perform repetitive operations on small data types (such as 8-bit pixels). The technology covers several areas, such as basic arithmetic operations, logical operations, shift operations, comparison operations, and data transfer instructions. These instructions use a SIMD model that allows the processor to perform a single calculation simultaneously on 2, 4, or 8 data elements by packing multiple operands (8-bit, 16-bit, or 32-bit values) into a single 64-bit register and performing processing functions on them in parallel. On a x86 compatible processor with Intel MMX™ Technology, MIL operations can execute, typically, 4 times faster than on a regular x86 processor. Some operations benefit even more from the MMX™ acceleration (for example, a thinning operation can be up to 16 times faster).

SSE™

Streaming SIMD extensions accelerate performance of floating point operations and include additional integer and cacheability instructions that significantly enhance performance.

System requirements

MIL is available as a set of DLLs under Windows NT/98/2000.

The following system requirements should be respected in order to ensure that MIL operates properly:

- Computer with an x86 compatible processor.
- Windows 98, Windows NT 4.0, or Windows 2000.
- Minimum of 32 Mbytes RAM.
- Minimum of 100 Mbytes free hard disk space.
- Display adaptor (optional).
- Matrox frame grabber (optional).

Supported compilers

The MIL CD includes MIL libraries that support the Microsoft Visual C++ 6.0 (service pack 3) compiler under Windows NT 4.0 (service pack 6), Windows 98 SE, and Windows 2000. The CD also includes ActiveMIL ActiveX controls for Microsoft Visual Basic 6.0 (service pack 3) and Microsoft Visual C++ 6.0 (service pack 3) RAD tools. The service pack indicated in parantheses denotes the actual platform used for testing.

Getting started

You are probably anxious to start using MIL. However, before you start, we recommend that you follow these steps:

- Fill out and mail in your registration card. This ensures that you are on our mailing list and will receive any information on product updates and promotions.
- Install MIL on your hard disk using the installation details (described later in this chapter). Upon completion, the *read.me* file, in the `\MIL` (or user-specified) directory, specifies the location of all MIL files and how to compile the MIL program examples. See the `\MIL\Doc` directory for additional documentation.
- Compile and run our sample program *mstart.exe*, in the examples directory, to test the installation.
- Review the *milsetup.h* file to make sure that the default setup configuration matches your system configuration.

Note, the defaults are not automatically installed on your system; a call to *MappAllocDefault()* initializes the system with these defaults. For simplicity, most examples use the default system and default display buffer. Upon installation, the default image buffer is monochrome if the input device is monochrome and color if the input device is color. Most examples expect the default image buffer to be monochrome. As you progress in the manual, you are shown how to set up your own buffers and select other system configurations. You can then return to a given example and replace portions of the code to meet your requirements.

Installation

In addition to your MIL CD, you will require a hardware key (a two-sided, 25-pin connector) for development of applications. The key allows you to code, debug, and run your applications. To redistribute your MIL applications, see the *Redistributing your MIL application* section at the end of this chapter.

To install your MIL software:

1. Attach the hardware key to the parallel port of your computer. If another device such as a printer is attached to the parallel port, disconnect it, attach the hardware key, and then attach the printer connector to the other end of the hardware key. Note, the printer need not be turned on.
2. Place the installation CD in an appropriate drive. The *setup.exe* program will run automatically.

During installation, you will be asked a number of questions, such as:

- The drive and directory on which to install the program.
- Your target operating system and compiler.
- The type of Matrox hardware installed in your computer (for example, Matrox Corona).
- The digitizer and display format to load into the default setup file, *milsetup.h*.
- The amount of DMA linear non-paged memory to reserve for grab buffers. The amount of reserved DMA memory also establishes the amount of remaining RAM available to your operating system.

After installation, read the *read.me* file in the `\MIL` (or user-specified) directory to determine where MIL files are located and how to compile and run the MIL examples. Note that the installation program also installs Matrox Intellicam (your digitizer configuration program) and the MIL Configuration utility.

MIL Configuration utility

The MIL Configuration utility, located in your Matrox Imaging\MILConfig directory, provides licensing, DMA configuration, and system information tools. For example, if you need to change the amount of reserved memory or if you change the amount of physical memory in your computer, you can change the amount of DMA memory assigned or RAM available to your system at any time by running the MIL Configuration utility (alternatively, you can adjust the memory by uninstalling and reinstalling MIL). Should you require technical support, use the MIL Configuration's System Info property page to generate a *.txt* file that contains all the necessary system information required for basic troubleshooting; this file can then be forwarded to your Matrox technical support representative. You can also use the MIL Configuration's Licensing property page to generate run-time licenses; this is discussed later in this chapter in the section, *Obtaining a run-time license*.

If MIL is run without the hardware key, a temporary evaluation license is assigned to your computer, allowing use of MIL for 30 days. Each time you run MIL, a dialog box appears indicating the number of days until the evaluation license expires. Once this time period has elapsed, MIL will not run unless a hardware key is attached.

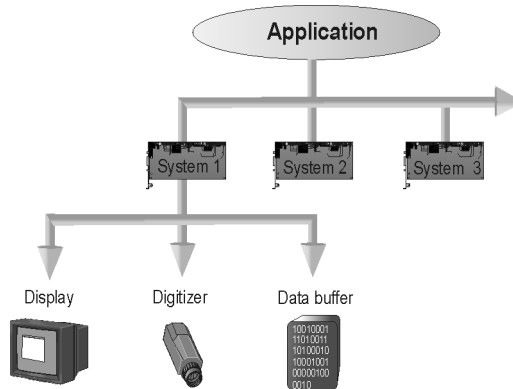
Note that MIL's 30-day evaluation license can only be installed once. Any attempt to tamper with the PC's calendar, before the date of expiry, will disable MIL. In that event, MIL can only be re-used once a hardware key is obtained.

Building an application

Initialization

At the beginning of each application, you must:

1. Allocate your MIL application. This creates a control and execution environment for your imaging application.
2. Allocate your systems. This opens communication channels and initializes the systems (or hardware resources). Once Host communication has been established with a system, you can allocate its memory resources, display, and input capabilities.



Note, systems can have many data buffers, displays, and digitizers. Processing can be done between many systems.

If the required system is the one specified in the *milsetup.h* file, you can use the *MappAllocDefault()* macro (also specified in *milsetup.h*) to allocate the default application, system, image buffer, display, and digitizer. Use *MappFreeDefault()* to free the application, devices, and memory resources that were allocated with *MappAllocDefault()*, when they are no longer required.

Alternatively, you can use *MappAlloc()*, *MsysAlloc()*, *MbufAllocColor()*, *MdispAlloc()*, and *MdigAlloc()* to perform the above-mentioned operations, respectively. In this case, when allocated memory resources, displays, and digitizers are no longer required, free them using *MbufFree()*, *MdispFree()*, and *MdigFree()*, respectively. At the end of each application, free the system using *MsysFree()*, and then free the application using *MappFree()*.

❖ Note, for information about functionality and hardware limitations specific to your target system, refer to the *MIL/MIL-Lite Board-specific notes* manual.

Multiple systems

Note, you can allocate more than one system and then use their identifiers to access their devices and memory resources. Any operation involving more than one system will be performed by the most appropriate one. By default, if none of these systems is more appropriate than the Host, the Host is used to perform the operation.

The default image buffer

If a color digitizer configuration format (DCF) was specified upon installation, the default image buffer is defined as a color buffer (RGB) in the *milsetup.h* file. Note, most examples in this manual assume that the default image buffer is a monochrome buffer. You will have to modify the examples appropriately in order to run them with color defaults. For more details on dealing with color, see Chapter 21.

When allocating the default image buffer and the default display, the image buffer is given a displayable attribute and set to the same size as the allocated display (in single-screen mode, the default display is the same as that of the image capture-size specified in the DCF). This buffer is then cleared and displayed.

Error reporting

You can enable or disable error reporting to the Host screen, using *MappControl()*. By default, error reporting is enabled. If you disable error reporting, you can still determine the success of a particular command or a sequence of commands, using *MappGetError()*. In addition, you can assign a user-defined function to handle the event of a MIL error using *MappHookFunction()*.

Compiling and linking

To compile a MIL application program, you must include the *mil.h* header file, in addition to the required standard C include files. After you have compiled your application program, you will have to link it with the appropriate libraries or import libraries for your operating system, compiler, and target board. The MIL libraries are located in the *MATROX IMAGING (OR USER-SPECIFIED) \MIL \LIBRARY \WINNT \MSC \DLL* directory.

MIL Libraries		Board Libraries	
Library	Description	Library	Description
mil.lib	Core library	mil1394.lib	Matrox Meteor-II/1394 library.
milblob.lib	Blob Analysis module library.	milgen.lib	Matrox Genesis library.
milcal.lib	Calibration module library.	milmet2.lib	Matrox Meteor-II/Standard/MultiChannel library.
milocr.lib	Character Recognition module library.	milmet2D.lib	Matrox Meteor-II/Digitizer
milpat.lib	Pattern Matching module library.	milorion.lib	Matrox Orion library.
milcode.lib	Code module library.	milpul.lib	Matrox Pulsar library.
milim.lib	Image Processing module library.		
milvga.lib	VGA library.		
milmeas.lib	Measurement module library.		

For more details, refer to the *read.me* file in the *\MIL \EXAMPLES* (or user-specified) directory.

Testing installation

We have provided a sample program, *mstart.c*, that allows you to test the installation process and become familiar with running a MIL application. This test program allocates the

application, opens communication with the default target system, displays a welcoming message, pauses, and frees the system resources.

```

/* File name: mstart.c
 * Synopsis: This program displays a welcoming message to the user.
 */

#include <stdio.h>
#include <mil.h>

void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
          MilSystem,      /* System identifier. */
          MilDisplay,     /* Display identifier. */
          MilImage;       /* Image buffer identifier. */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                    M_NULL, &MilImage);

    /* Print a string in the image buffer. */
    MgraText(M_DEFAULT, MilImage, 176L, 210L, " ..... ");
    MgraText(M_DEFAULT, MilImage, 176L, 235L, " Welcome to MIL !!! ");
    MgraText(M_DEFAULT, MilImage, 176L, 260L, " ..... ");

    /* Print a message on the Host screen. */
    printf("\n");
    printf("\nWelcome to MIL !!!\n was printed.\n\n");
    printf("Press <Enter> to end.\n");
    getchar();

    /* Free defaults. */
    MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

Communicating properly?

During application development, you can use *mstart.c* to ensure that the software is communicating properly with the target system. To make sure your frame grabber is working properly with your camera, use Intellicam.

Examples in general

Throughout this manual, examples have been provided to simplify concepts and get you started quickly. The source listing of these examples can be found on disk. Refer to the *readme* file in the `\MIL\EXAMPLES` (or user-specified) directory to determine how to compile these examples.

In addition, some systems cannot run some of the examples because they don't have the hardware capability or enough memory. You should skip these examples or modify them.

Distributing your MIL application

To distribute your MIL application, you will have to distribute MIL run-time DLL files with your application and ensure that the MIL run-time licenses are installed on the target computers.

Distributing MIL run-time DLL files with your software

If the target computers (on which you want to install the MIL run-time DLLs) are immediately accessible, you can install the run-time DLLs directly from the MIL CD. To do so, run the MIL setup program and choose the redistribution option.

Alternatively, to distribute your MIL run-time DLLs, you can have your setup program call MIL's redistribution setup program. There are two ways to distribute the MIL run-time DLLs along with your application:

- Normal distribution
- Silent distribution

Normal distribution

A normal distribution prompts the user with MIL dialog boxes for setup information. To distribute MIL run-time DLL files with your software using this method:

1. Copy the `\REDIST` directory from the MIL CD to the path from which you will burn your software CD.
2. Adjust your installation program so that it calls the `\REDIST\MATROX\SETUP.EXE` file with the parameter, `REDISTRIBUTION`.

The setup.exe file installs the required run-time MIL DLL files on your client's system.

Silent distribution

A silent distribution does not prompt the user for any information; instead, it uses a response file to provide the necessary setup parameters for the intended computer. A silent distribution is often wanted when including MIL within your application and you do not wish to have any Matrox Imaging setup dialog boxes appear.

To distribute the MIL run-time DLLs using a silent distribution:

1. Follow the steps for a normal MIL redistribution of your application.
2. Create a response file that provides the setup questions with all the answers necessary to install MIL according to the target computers. The response file's format parameters and error codes can be found in the *Redist.txt* file.
3. Call the *redist.exe* program with the additional 'RESPONSEFILE = "<filename>" -s' parameter to specify the name and the location of your response file. For example:

```
D:\Redist\Matrox\redist.exe REDISTRIBUTION RESPONSEFILE="D:\Redist\Matrox\response.txt" -s
```

Obtaining a run-time license

You require a MIL run-time license for each MIL application that is distributed. You can call Matrox or your distributor to obtain a license, or to buy a license generator kit to generate run-time licenses yourself. Depending on the kit you purchase, a license kit can generate up to a 100 run-time licenses. Note that a run-time license does not allow development or debugging of MIL applications. Run-time license hardware keys are also available.

Obtaining a run-time license from Matrox or your distributor

You can obtain a license as follows:

1. Install your MIL application on the target PC. An evaluation license is assigned to your system, allowing use of MIL for 30 days.

Each time your application runs MIL, a dialog box appears indicating the number of days until the evaluation license expires and an option to obtain a run-time license is displayed.

2. Select the **Get license** option. A computer code, unique to your machine, is displayed and you are prompted for a license number.

Alternatively, you can run the MIL Configuration utility, located in your Matrox Imaging\MILConfig directory, and generate a license using the Licensing property page.

3. Call Matrox or your distributor and give them this computer code to obtain your permanent run-time license number.
4. Enter the license number.
 - ❖ Note that MIL 30-day evaluation licenses can only be used once. Any attempt to tamper with the PC's calendar, before the date of expiry, will disable the MIL evaluation license. In that event, MIL can only be re-used once a license is obtained.

Using a license kit

The second method to acquire a run-time license requires that you obtain a license kit. A license kit consists of a license meter key (a two-sided 25-pin connector) and a MIL CD. This method allows you to generate run-time licenses for machines using their unique computer codes. You can generate a license with the license kit using one of the following methods:

1. Attach the license meter key to the parallel port of your PC.
2. Run MIL setup. The **Matrox Imaging Master Setup** dialog box appears.
3. Under Redistribution, choose the **MIL License Generator** option. This installs the Matrox License Generator application on your PC.
4. Exit setup.
5. Now, run the Matrox License Generator.
6. Type the computer code for which to generate a license.
7. Click the **Get license** button in the next dialog box. This is the run-time license number that must be entered in your client's machine. Please record the license number for future reference.

Or,

1. Attach the license meter key to the parallel port of your PC.
2. Run your application which incorporates MIL.
Alternatively, you can run the MIL Configuration utility.

3. A dialog box appears indicating that your evaluation license has expired: choose the **Get license** option.
4. Upon consenting to the Matrox Imaging Library Licensing Agreement, click the **Get license** button in the next dialog box. Please record the license number for future reference.

Note that each time a license is generated, a number is deducted from the license meter key until the key is empty.

Using a MIL run-time hardware key for the redistribution license

You can use a MIL run-time hardware key instead of a software license for your target system. To have MIL run-time DLLs recognize it, you must install the driver for the hardware key.

To install the hardware key driver, in addition to your regular redistribution setup, use one of the following procedures:

- **For Windows NT/2000:** Call the `\MATROX\DRIVERS\SUPERPRO\WINNT\SETUPX86.EXE` and select the Install Sentinel Driver function.
- **For Windows 98:** Call the `\MATROX\DRIVERS\SUPERPRO\WIN95\SETUPW9X.EXE` and select Install Sentinel Driver function.

For more information about Sentinel driver installation, refer to the *Readme.txt* file in the `\MATROX\DRIVERS\SUPERPRO` directory.

Note, if you are performing a silent distribution using a MIL run-time hardware key for the redistribution license, set the response file parameter `SUPERPRO` to `INSTALL`, and the Sentinel hardware key driver will be installed with the redistribution of MIL.

Chapter 2: Allocating an image buffer and grabbing images

This chapter shows you how to allocate an image buffer and the basics to start grabbing images.

Getting started

After having run the *mstart.c* program to ensure that you have installed MIL properly, you are ready to grab and display an image. This chapter covers how to allocate and display a monochrome image buffer and the basics to start grabbing.

Note, most of our examples that grab data assume that the system has a monochrome digitizer. They also assume that the input device (camera) is monochrome and is connected to the default input channel of this digitizer (defaults are defined in the *milsetup.h* file).

In addition, the examples assume that the default image buffer is monochrome.

If you have specified a color digitizer input format upon installation, the default digitizer and image buffer will be set to color accordingly (a color image buffer is an image buffer with multiple color bands rather than a monochrome buffer), and therefore will not be appropriate for most examples. To run the examples using the color defaults, you will have to modify some examples appropriately.

Later in this manual, we discuss changing the current input channel, how to specify a different digitizer format, and how to allocate different types of image buffer. With that knowledge, you can return to this chapter and modify the examples. Chapter 21 discusses dealing with color in detail.

Allocating and displaying an image buffer

Allocating an image buffer

Image buffers are storage areas that can hold image data so that it can be displayed, manipulated, grabbed, and/or analyzed. For simple operations, you will find it sufficient to use the default image buffer that can be allocated during application initialization with the *MappAllocDefault()* macro. However, for some operations, you will need to allocate another buffer. For example, if you require that the image data resulting from an operation does not overwrite the source data, you will need two separate image buffers.

You allocate a monochrome image buffer, using *MbufAlloc2d()*. This command requires that you specify:

- The system on which to allocate the buffer.
- The image buffer's size in x and y dimensions.
- The depth of the buffer: 1-, 8-, 16-, or 32-bit buffers.
- The image buffer's data type. Signed, unsigned, and floating-point buffers are all supported by MIL.
- The image buffer's intended use. You can allocate an image buffer to have a combination of uses. It can be used as the source or destination buffer for a processing operation (M_PROC), a buffer in which to store acquired data (M_GRAB), and/or a displayable buffer (M_DISP). This type of information determines where the buffer is allocated in physical memory.

Displaying an image buffer



Especially during application development, it is useful to display the image buffer that you are manipulating. You must first allocate a MIL display on the target system, using *MdispAlloc()* (or *MappAllocDefault()*). If you have allocated a displayable buffer (M_DISP), display it in this display, using *MdispSelect()* and stop displaying it using *MdispDeselect()*. Note, however, that the image buffer and the display must be allocated on the same system.

The following example shows you how to allocate and display an image buffer. Upon completion, it leaves the buffer contents on the display so that you can analyze it. You can modify the example and remove it from the display upon exit by calling *MdispDeselect()* before freeing the image buffer.

```

/* File name: mdisplay.c
 * Synopsis: This program allocates a displayable image buffer, clears its
 *           contents, draws a filled circle, and then displays the buffer.
 *           It also checks whether the allocation was successful, using
 *           the MIL error reporting mechanism.
 */

#include <stdio.h>
#include <stdlib.h>
#include <mil.h>

#define IMAGE_DEPTH 8L

void main(void)
{
    MIL_ID  MilApplication, /* Application identifier. */
           MilSystem,      /* System identifier.      */
           MilDisplay,     /* Display identifier.     */
           MilImage;       /* Image buffer identifier.*/
    long    ErrorCode;      /* Error code value.       */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                     M_NULL, M_NULL);

    /* Allocate a two-dimensional image buffer with the same dimensions as the
     * displayable screen, in which to perform graphic operations.
     */
    MbufAlloc2d(MilSystem, M_DEF_IMAGE_SIZE_X_MIN, M_DEF_IMAGE_SIZE_Y_MIN,
                M_DEF_IMAGE_TYPE, M_IMAGE+M_DISP, &MilImage);

    (cont. ...)

```

```

/* Check the error status code set by the allocation command. If there
 * was no error, draw and display a circle, otherwise print an error
 * message and exit.
 */
MappGetError(M_CURRENT, &ErrorCode);
if (ErrorCode == M_NULL)
{
    /* Clear buffer and draw a circle. */
    MbufClear(MilImage, 0L);
    MgraColor(M_DEFAULT, 255L);
    MgraArcFill(M_DEFAULT, MilImage, 256L, 240L, 100L, 100L, 0.0, 360.0);

    /* Display the image buffer. */
    MdispSelect(MilDisplay, MilImage);

    /* Print a message. */
    printf("A circle was drawn in the displayed image buffer.\n");
    printf("Press <Enter> to end.\n");
    getchar();

    /* Release image buffer. */
    MbufFree(MilImage);
}
else
{
    /* Print an error message. */
    printf("Error: Image buffer allocation failed.\n");
    printf("Press <Enter> to end.\n");
    getchar();
}

/* Release defaults. */
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);
}

```

In this example, we also showed how to determine the success of a buffer allocation. Subsequent examples will not perform explicit error checking; instead, errors will be returned automatically to the screen.

Note, if you allocated the default buffer (*MappAllocDefault()*), this buffer would be cleared and displayed by default.

Displaying multiple buffers

With MIL, you can also display multiple buffers. This is discussed later in the manual, in *Chapter 18: Displaying an image*.

Grabbing images

Grabbing an image



Many applications depend on the ability to grab an image for later analysis or inspection. With MIL, you use an allocated digitizer to grab from an input device (typically a video camera). To allocate your digitizer, use *MdigAlloc()* or *MappAllocDefault()*. This configures the camera interface on the digitizer so it can accept input from the input device. With a call to *MdigGrab()*, you can then grab into a grab image buffer (M_GRAB).

The following example shows you how to grab an image from the default camera.

```
/* File name: mgrab.c
 * Synopsis: This program grabs an image from the camera.
 */

#include <stdio.h>
#include <mil.h>

void main(void)
{
    MIL_ID  MilApplication,    /* Application identifier.    */
           MilSystem,         /* System identifier.         */
           MilDisplay,        /* Display identifier.        */
           MilDigitizer,      /* Digitizer identifier.      */
           MilImage;          /* Image buffer identifier.   */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                     &MilDigitizer, &MilImage);

    /* Grab an image. */
    MdigGrab(MilDigitizer, MilImage);

    /* Report what has happened to the Host screen. */
    printf("An image has been grabbed.\n");
    printf("Press <Enter> to end.\n");
    getchar();

    /* Release defaults. */
    MappFreeDefault(MilApplication, MilSystem, MilDisplay, MilDigitizer,
                   MilImage);
}
```

Allocate the grab image buffer on the same system, and of the same data format type, as the digitizer. For color input devices, use color image buffers (see *Chapter 21: Color*).

By default, when *MdigGrab()* is issued, it grabs a complete frame of data. Use *MdigControl()* to control the number of frames or fields grabbed by *MdigGrab()*. To control the digitizer, see *Chapter 20: Input devices and digitizers*.

*Continuous grabbing
and adjusting your
camera*

When adjusting and focusing your camera, grabbing a single frame at a time can be tedious. MIL features a continuous grab function, *MdigGrabContinuous()*, that grabs image frames into the specified buffer until you issue *MdigHalt()*.

This is discussed in greater detail in *Chapter 20: Input devices and digitizers*. The following example is of adjusting a camera using a continuous grab.

```
/* File name: mfocus.c
 * Synopsis: This program allows you to adjust your camera by grabbing
 *           continuously until a key is pressed.
 */

#include <stdio.h>
#include <mil.h>

void main(void)
{
    MIL_ID  MilApplication, /* Application identifier. */
           MilSystem,      /* System identifier.      */
           MilDisplay,     /* Display identifier.     */
           MilDigitizer,   /* Digitizer identifier.   */
           MilImage;       /* Image buffer identifier.*/

    /* Allocate defaults. */
    MappAllocDefault(M.SETUP, &MilApplication, &MilSystem, &MilDisplay,
                    &MilDigitizer, &MilImage);

    /* Grab continuously. */
    MdigGrabContinuous(MilDigitizer, MilImage);
```

(cont. ...)

```

/* When a key is pressed, halt. */
printf("Continuous grab in progress. Adjust your camera and\n");
printf("press <Enter> to stop grabbing.\n");
getchar();

/* Stop continuous grab. */
MdigHalt(MilDigitizer);

/* Pause to show the result. */
printf("\nDisplaying the last grabbed image.\n");
printf("Press <Enter> to end.\n");
getchar();

/* Release defaults. */
MappFreeDefault(MilApplication, MilSystem, MilDisplay,
                MilDigitizer, MilImage);
}

```

If your camera supports remote lens adjustment, you can use *MdigFocus()* to automatically adjust the lens motor of your camera to achieve optimum focus in your images. See the Auto-focusing section in *Chapter 20: Input devices and digitizers*.

Chapter 3: Image processing

This chapter describes the steps to performing a typical application with the MIL image processing module.

Image processing

Pictures, or images, are important sources of information for interpretation and analysis. These might be images of a building undergoing renovations, a planet's surface transmitted from a spacecraft, plant cells magnified with a microscope, or electronic circuitry. Human analysis of these images or objects presents inherent difficulties: the visual inspection process is time-consuming and subject to inconsistent interpretations and assessments. Computers, on the other hand, are ideal for performing these tasks.

In order for computers to process images, the images must be numerically represented. This process is known as *image digitization*.

Once images are represented digitally, computers can reliably automate the extraction of useful information through the use of digital image processing. Digital image processing performs various types of image enhancements, distortion corrections, and measurements.

The MIL package

MIL provides a comprehensive set of image processing operations. There are two main types of image processing operations:

- Those that enhance or transform an image.
- Those that analyze an image (that is, generate a numeric or graphic report that relates specific image information).

MIL supports such operations as:

- Point-to-point operations. These operations include constant thresholding, image comparison, image subtraction, and image mapping. They compute each pixel result as a function of the pixel value at a corresponding location in either one or two source images.
- Statistical operations. These extract statistical information from a given image, such as the minimum or maximum image pixel value or a histogram. They condense a frame of pixels into a smaller, more functional set of values for analysis.
- Spatial filtering operations. These operations are also known as convolution. They include operations that can enhance and smooth images, accentuate image edges, and remove 'noise' from an image. Most of these operations compute results based on an underlying neighborhood process: the weighted sum of a pixel value and its neighbors' values.
- Morphological operations. These operations include erosion, dilation, opening, and closing of images. They compute new values according to geometric relationships and matches to known patterns in the input image.

Steps to performing a typical application

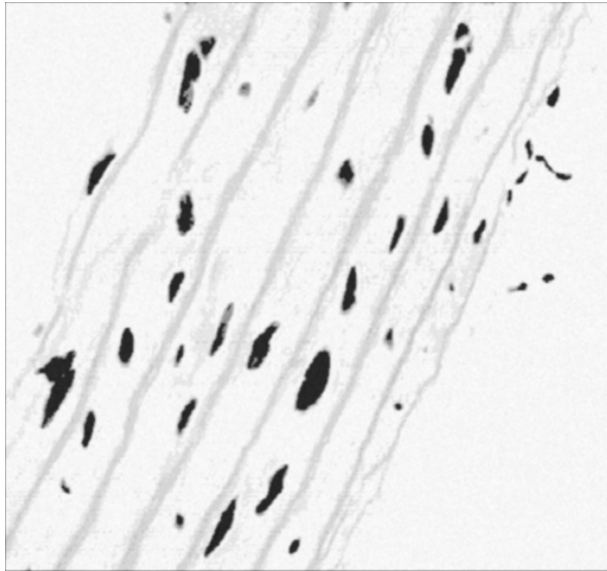
We have described the broad set of operations included in the MIL image processing module. Most applications do not require all of these operations.

Image processing pertains to more than one field and no single application program can solve the problems associated with each one of these fields. Therefore, this section describes what we believe to be a typical problem, where the solution makes use of most of the supported operations. It also outlines the steps to take to implement this solution.

A typical application

In analyzing an image of a tissue sample, you might want to know the number of cell nuclei that are larger than a certain size, indicating an abnormality. This involves the following image processing steps:

1. Grab or load an image of a magnified tissue sample.
2. Smooth the image to remove noise produced during the grab.
3. Binarize the image so that the cell nuclei or particles and the background have different values: represent particles in white and the background in black. This will allow you, later, to label each particle with a unique number.
4. Perform an opening operation to remove small particles from the image.



5. Label each particle with a unique consecutive number starting with the label 1.
6. Calculate and read the extreme value of the image. This value also corresponds to the largest label. Since the image particles are labeled with consecutive unique numbers, the largest valued particle is also labeled with a number that corresponds to the number of particles in the image.

How to encode these steps

The following sample program (mcount.c) shows you how to encode these steps, using an existing image of a tissue sample (cell.mim).

```

/* File name: mcount.c
 * Synopsis: This program loads an image of a tissue sample and determines
 *the number of cell nuclei which are larger than a certain size.
 */

#include <stdio.h>
#include <mil.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE      "cell.mim"
#define IMAGE_WIDTH     512L
#define IMAGE_HEIGHT    480L
#define IMAGE_THRESHOLD_VALUE 128L

/* Small particle radius (in pixels). */
#define SMALL_PARTICLE_RADIUS 2L

void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
        MilSystem, /* System identifier. */
        MilDisplay, /* Display identifier. */
        MilImage, /* Image buffer identifier. */
        MilSubImage, /* Sub-image buffer identifier. */
        ExtrResult; /* Extreme result buffer identifier. */
    long MaxLabelNumber; /* Highest label value. */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                    M_NULL, &MilImage);

    /* Restrict the region to be processed to the image size. */
    MbufChild2d(MilImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage);

    /* Load source image into an image buffer. */
    MbufLoad(IMAGE_FILE, MilSubImage);

    /* Pause to show the original image. */
    printf("This program counts the number of large ");
    printf("particles in the displayed image.\n");
    printf("Press <Enter> to continue.\n");
    getchar();

    (cont. ...)

```

```

/* Smooth the image to remove noise. */
MimConvolve(MilSubImage, MilSubImage, M_SMOOTH);

/* Binarize the image so that particles are represented
 * in white and the background in black. */
MimBinarize(MilSubImage, MilSubImage, M_LESS_OR_EQUAL,
            IMAGE_THRESHOLD_VALUE, M_NULL);

/* Remove small particles. */
MimOpen(MilSubImage, MilSubImage, SMALL_PARTICLE_RADIUS, M_BINARY);

/* Pause to show the remaining particle(s). */
printf("\n");
printf("These particles have been extracted from the original image.\n");
printf("Press <Enter> to continue.\n");
getchar();

/* Label image in place. */
MimLabel(MilSubImage, MilSubImage, M_DEFAULT);

/* The largest label value corresponds to the extreme value of the image.
 */
MimAllocResult(MilSystem, 1L, M_EXTREME_LIST, &ExtrResult);
MimFindExtreme(MilSubImage, ExtrResult, M_MAX_VALUE);
MimGetResult(ExtrResult, M_VALUE, &MaxLabelNumber);

/* Multiply the labeling result to augment the gray level of the particles
 */
if (MaxLabelNumber)
    MimArith(MilSubImage, 255L/MaxLabelNumber, MilSubImage, M_MULT_CONST);

/* Print results. */
printf("\n");
printf("There were %ld large particle(s) in the original image.\n",
        MaxLabelNumber);

printf("Press <Enter> to end.");
getchar();

/* Free all allocations. */
MimFree(ExtrResult);
MbufFree(MilSubImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

Chapter 4: Improving your images

This chapter describes different ways to improve your images, using the MIL image processing module.

Image quality

Prior to manipulating and extracting information from an image, many applications require that you obtain the best possible digital representation of it. Several factors affect the quality of an image. These include:

- **Random noise.** There are two main types of random noise:
 - **Gaussian noise.** When this type of noise is present, the exact value of any given pixel is different for each grabbed image; this type of noise adds to or subtracts from the actual pixel value.
 - **Salt-and-pepper noise** (also known as impulse or shot noise). This type of noise introduces pixels of arbitrary values (usually high-frequency values) that are generally noticeable because they are completely unrelated to the neighboring pixels.

Random noise can be caused, for example, by the camera or digitizer because electronic devices tend to generate a certain amount of noise. If the images were transmitted, the distance between the sending and the receiving devices also magnifies the random noise problem because of interference.

- **Systematic noise.** Unlike random noise, this type of noise can be predicted, appearing as a group of pixels that should not be part of the actual image. This can be caused, for example, by the camera or digitizer or by uneven lighting. If the image was magnified, microscopic dust particles, on either the object or a camera lens, can appear to be part of the image.
- **Distortions.** Distortions appear as geometric transforms of the actual image. These can be caused, for example, by the position of the camera relative to the object (not perpendicular), the curvature in the optical lenses, or a non-unity aspect ratio of an acquisition device.

Techniques to improve images

Most interference problems cannot be adjusted very easily at the source; therefore, preprocessing will probably be required to improve the image as much as possible, without affecting the information that you are seeking. There are several techniques that you can use to improve your image:

- Grab the object of interest several times, averaging each image frame with the previous. This technique is generally effective on Gaussian random noise.
- Apply a low-pass spatial filter to your image to reduce Gaussian random noise and systematic noise with small scale variations. This technique replaces each pixel with a weighted sum of its neighborhood.
- Apply a median filter to your image to reduce salt-and-pepper noise. This technique replaces each pixel with the median pixel value of its neighborhood.
- Perform a morphological opening operation to remove small particles and break isthmuses between objects in your image.
- Perform a morphological closing operation to remove small holes in objects.
- Make sure that the type of camera you allocate digitizes the image with *square* pixels (that is, a 1:1 aspect ratio), to reduce object-shape distortions. If this is not possible or does not correct the problem, you can resize the image, using *MimResize()*.

Averaging an input sequence

An effective technique to remove random noise is to average a grabbed sequence of the same target image. For instance, Gaussian noise affects the value of any given pixel for each grabbed frame, adding to or subtracting from the actual pixel value. Therefore, over several image acquisitions, this noise averages out to zero. As a rule of thumb, Gaussian noise is generally reduced by the square root of the number of grabbed frames.

*Frame averaging
with MIL*

With MIL, you can average an input sequence, using either one of the following methods:

- Adding all input frames and then dividing the result by a specified weight factor. To use this method, use *MimArith()*.
- Adding weighted input frames to a weighted accumulator buffer ($I_{acc} = aI_{in} + (1 - a)I_{acc}$ or $I_{acc} = a(I_{in} - I_{acc}) + I_{acc}$). To use this method, use *MimArithMultiple()* with `M_WEIGHTED_AVERAGE`.

The latter approach also acts as a temporal filter if the input is changing. This allows you to filter out moving objects from a constant background.

If you do not want to lose any frames in your sequence, you can use a method called double buffering, a technique whereby which you can grab data into one buffer while another buffer is being processed. For an example on how to implement double buffering, see *mdbproc.c* file in MIL's example directory.

Applying spatial filters

Spatial filtering provides an effective method to reduce noise. Spatial filtering operations determine each pixel's value based on its neighborhood values. They allow images to be separated into high-frequency and low-frequency components. There are two main types of spatial filters that can remove noise: low-pass filters and rank filters.

Low-pass spatial filters

Low-pass spatial filters are effective in reducing Gaussian random noise (and high-frequency systematic noise), provided that the noise frequency is not too close to the spatial frequency of significant image data. These filters replace each pixel with a weighted sum of each pixel's neighborhood. Note, these filters have a side-effect of selectively smoothing your image and removing edge information.

You can apply low-pass filters with the *MimConvolve()* image processing command. The weights applied to each neighborhood are specified in a data buffer called a *kernel*. MIL provides a predefined low-pass filter called `M_SMOOTH`, that you will find satisfies most applications.

In the previous chapter, we looked at a cell-analysis application that determined the number of cell nuclei in a tissue sample. Since Gaussian noise is generally introduced when digitizing, we performed a smoothing operation prior to performing the analysis.

Rank filters

Rank-filter operations are more suitable for removing salt-and-pepper type noise since they replace each pixel with a pixel in its neighborhood rather than a weighted sum of its neighborhood. The weighted sum generally creates a blotchy effect around each noise pixel.

You can perform a rank-filter operation, using *MimRank()*. In most cases, it is best to use a rank that is half of the number of elements in the neighborhood. This effectively replaces each pixel with the median of the neighborhood and is therefore called a *median filter*. To perform a median filter, set the *MimRank()* rank parameter to `M_MEDIAN`. You will find that the median filter will most often suit your application needs.

Opening and closing

Another way of improving the image might be to remove, for example, small particles that have been introduced by dust, or holes in objects. These tasks can generally be accomplished with an opening or closing operation, respectively.

Opening and closing operations determine each pixel's value according to its geometric relationship with neighborhood pixels, and as such are part of a larger group of operations known as *morphological operations*.

Removing small particles

Besides removing small particles, opening operations also break isthmuses or connections between touching objects. MIL provides the *MimOpen()* command to perform a basic opening operation on 3 by 3 neighborhoods taking all neighborhood pixels into account.

Filling holes

Closing operations are very useful in filling holes in objects; however in doing so, they also connect close objects, as shown below. *MimClose()* performs a standard 3 by 3 closing operation taking all neighborhood pixels into account.



Note, opening and closing operations work best on binary images.

Since opening is the result of eroding and then dilating an image, and closing is the result of dilating and then eroding an image, you can also customize an opening or closing operation, using *MimErode()* and *MimDilate()*. Erosion and dilation are discussed fully in *Chapter 5: Image manipulation*.

Basic geometrical transform

Image distortions can affect application results. For example, in a medical application that analyzes blood cells, if the camera does not have a one-to-one aspect ratio and no correction is performed, the cells appear distorted and elongated, and incorrect interpretations might result. Rotating such an image causes even more serious object distortion.

To resolve distortion problems, the MIL image processing module offers basic, as well as advanced, geometric functions. Since the advanced geometric functions (*MimPolarTransform()* and *MimWarp()*) are slower than the basic geometric functions, they should only be used when the required transform cannot be performed using a basic geometric function. The advanced geometric functions are discussed in Chapter 6.

Resizing an image

The *MimResize()* function resizes an image along the horizontal and/or vertical axis. This can help resolve aspect-ratio problems. If both the horizontal and vertical resizing factors are set to the same value, this function can reduce or magnify an image to an appropriate size.

Rotating an image

In some instances, the orientation of an image can also cause erroneous conclusions. When an object is rotated from its original position, you can realign it in memory by the required angle, using *MimRotate()*.

Translating an image

MimTranslate() displaces an image by a specified number of pixels in the x and/or y direction, with sub-pixel accuracy.

Flipping an image

MimFlip() flips an image horizontally (left to right) or vertically (top to bottom). Note that flipping horizontally allows you to get a mirror copy of the original image.

Interpolation

Geometric functions are performed according to a specified interpolation mode. Interpolation is discussed in Chapter 6.

Chapter 5: Image manipulation

This chapter describes different ways to manipulate your images using the MIL image processing module.

Image manipulation

Once you have improved your image as much as possible, you are ready to start manipulating and extracting information from it. The MIL image processing module offers you several image manipulation operations. Depending on your application, you will need to perform one operation before another in order to extract the required information. This chapter will try to help you determine this order.

Image statistics

Many applications need to obtain some type of image statistic to condense a frame of pixels into a smaller, more functional set of values for analysis. The statistic might be required to perform some subsequent operation and/or might be used to summarize the effect of some image operation. The MIL image processing module offers a variety of functions to extract statistical information from an image. These functions allow you, for example, to:

- Generate the intensity histogram of an image buffer (*MimHistogram()*).
- Find the minimum and maximum values of an image buffer (*MimFindExtreme()*).
- Find the location of certain pixel values (*MimLocateEvent()*).
- Find the number of differences between two image buffers (*MimCountDifference()*).
- Perform an image projection from two dimensions to one dimension (*MimProject()*).

Generating a histogram

A histogram is the intensity distribution of pixel values in an image and is generated by counting the number of times each pixel intensity occurs. This information is very useful for several applications. In particular, it is useful to select a threshold level when binarizing an image (discussed later) and to change the image intensity distribution when trying to increase the image contrast.

You can generate an image histogram, using *MimHistogram()*. This command takes an image buffer and stores the results in a previously allocated histogram result buffer. You allocate the result buffer, using *MimAllocResult()*, specifying its type as `M_HIST_LIST`. Give it enough entries to hold all possible intensities.

You can then read results, using *MimGetResult()*. Once results have been read from the result structure, you can release the structure, using *MimFree()*.

```
/* File name: mhist.c
 * Synopsis: This program loads an image of a tissue sample and generates
 *           the image histogram.
 */
#include <stdio.h>
#include <mil.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE      "cell.mim"
#define IMAGE_WIDTH     512L
#define IMAGE_HEIGHT    480L

/* Number of possible pixel intensities. */
#define NUM_INTENSITIES 256L

void main(void)
{
    MIL_ID MilApplication,          /* Application identifier */
        MilSystem,                /* System identifier. */
        MilDisplay,              /* Display identifier. */
        MilImage,                /* Image buffer identifier. */
        MilSubImage,             /* Sub-image buffer identifier. */
        HistResult;              /* Histogram buffer identifier. */
    long HistVals[NUM_INTENSITIES]; /* Histogram values. */
    short i;                      /* Counter. */

    /* Allocate the default system and image buffer. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                     M_NULL, &MilImage);
```

(cont...)

```

/* Restrict the region to be processed to the image size. */
MbufChild2d(MilImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage);

/* Load source image into an image buffer. */
MbufLoad(IMAGE_FILE, MilSubImage);

/* Allocate a histogram result buffer. */
MimAllocResult(MilSystem, NUM_INTENSITIES, M_HIST_LIST, &HistResult);

/* Generate the histogram. */
MimHistogram(MilSubImage, HistResult);

/* Get the results. */
MimGetResult(HistResult, M_VALUE, HistVals);

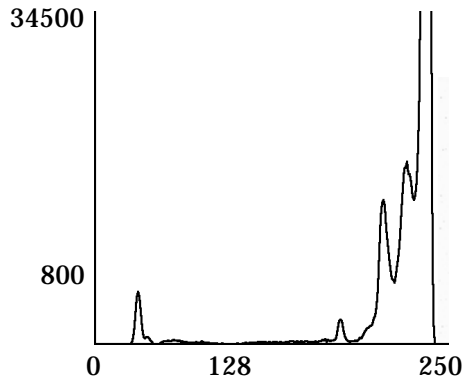
/* Print the results. */
printf("Press <Enter> to print the histogram for the displayed image.\n");
getchar();

for(i=0; i<NUM_INTENSITIES; i++)
{
    printf("%3d: %6ld\n", i, HistVals[i]);
    if((i % 20) == 19)
    {
        printf("\nPress <Enter> to continue.\n");
        getchar();
    }
}

/* Free all allocations. */
MimFree(HistResult);
MbufFree(MilSubImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

You could use the MIL graphics commands to plot the histogram results on a graph (as shown below). The graphics commands (*Mgra...()*) are discussed later in this manual.



The first peak shows the pixel intensities that make up the dark particles in the image, while the other peaks represent the gray and white background pixels.

Finding the image extremes

You can find the minimum and maximum pixel values of your image with *MimFindExtreme()*. Perhaps the most common use for finding the minimum and maximum image pixel values is to fine-tune the black and white reference levels of your digitizer, ensuring full-range digitization.

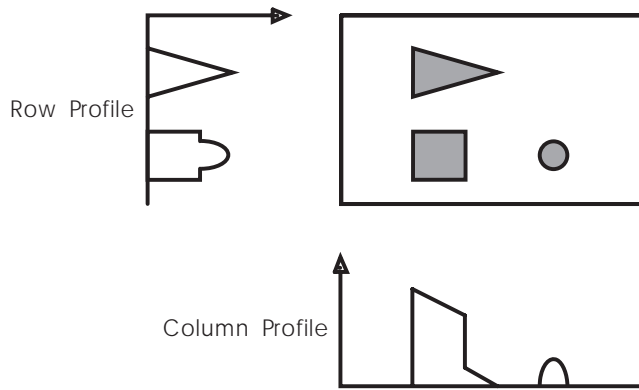
Another use of *MimFindExtreme()* is to find the number of objects in a labeled image. If all objects in an image are labeled with unique consecutive values, using *MimLabel()* (discussed later in this chapter), the largest label value also corresponds to the number of objects in your image.

The *MimFindExtreme()* command stores results in an extreme-value result buffer that should have been previously allocated, using *MimAllocResult()* with the `M_EXTREME_LIST` flag. You can get the resulting values, using *MimGetResult()*, and free the result buffer, using *MimFree()*.

Projecting an image to one dimension

The *MimProject()* command projects an image buffer into a one dimensional buffer, generated by adding all pixel values along each diagonal in the image at the specified angle. This projection is referred to as the pixel value density of each diagonal. The 90 degree projection of the image is known as the row profile, and the 0 degree projection is known as the column profile.

The *MimProject()* command can perform both grayscale and binary image projections. On simple binary images, the projection is useful to detect object locations.



You allocate the result buffer, using *MimAllocResult()* with the `M_PROJ_LIST` flag. You should define a result buffer with as many locations as there are diagonals in the image at the specified angle. You can then get the resulting values, using *MimGetResult()*, and free the result structure, using *MimFree()*.

Thresholding your images

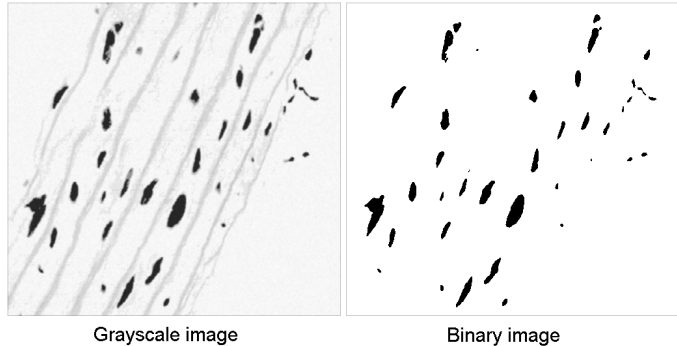
Thresholding images means reducing each pixel to a certain range of values. Some operations can be performed more efficiently on thresholded images. Images with full grayscale levels are useful for some tasks, but have redundant information for others. The MIL package provides two thresholding methods:

- Binarizing, using the *MimBinarize()* command.
- Clipping, using the *MimClip()* command.

Binarizing

A binarizing operation reduces an image to two grayscale values: 0 and the maximum value in the image (for example, 255 if the image is 8-bit). Binary images are useful when trying to identify geometrical patterns and objects in your image since they are not cluttered with shading information. For example,

in our cell application in *Chapter 3*, we were concerned with the number of dark particles in the image and not with the actual gray levels of the dark particles. Therefore, we binarized the image to distinguish dark particles from the background.



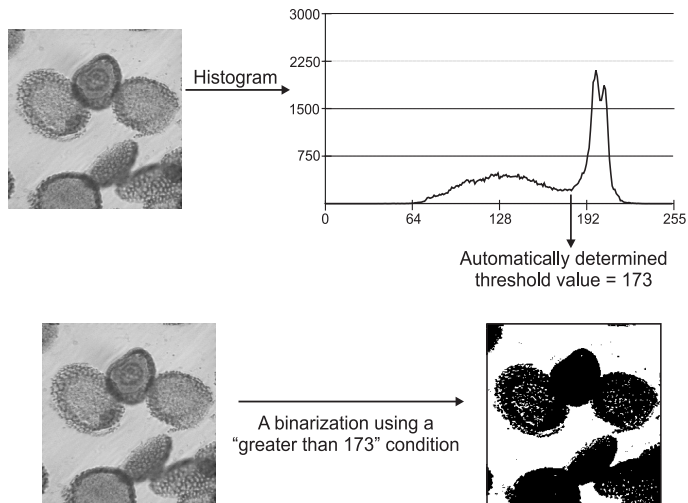
A binarizing operation is performed by comparing each pixel value in the image against one or two specified threshold values (for example, whether each pixel value is above one of the threshold values, or within the range of the two threshold values). Pixels that meet the specified condition are set to the maximum value in the image while other pixels are set to 0.

When using *MimBinarize()*, it is important to select a threshold value that preserves the required information. For example, in our cell application, an inappropriate threshold value might have changed fewer or more image pixels into background pixels, resulting in fewer or more particles than actually exist.

Determining threshold value from histogram

MimBinarize() can automatically determine the threshold value from the source image's characteristics. Specifically, a histogram of the source image is internally generated, then the threshold value is set to the minimum value between the two

most statistically important peaks in the histogram, on the assumption that these peaks represent the object and the background.



Clipping

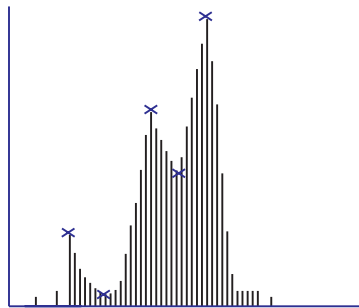
Clipping changes the image data less dramatically than binarizing. It changes the data to include only the range of pixel values in which you are interested. *MimClip()* takes a condition with at most two threshold points and replaces only those pixels that meet the condition with given values. Pixels that do not meet the condition are unaffected.

This can be useful to change data from one data type to another. For example, if you have a 16-bit result, but most of the pixels are less than 256, you could clip the result into an 8-bit buffer, and set all the pixels that are too big to the largest possible value, that is, 255.

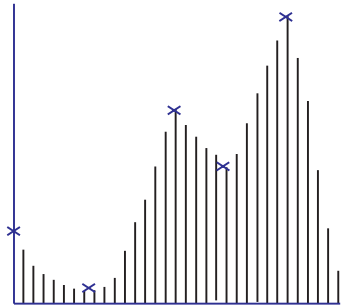
```
MimClip(ImageBuf16, ImageBuf8, M_GREATER, 255L, M_NULL, 255, M_NULL);
```


Histogram equalization

A histogram equalization can be performed to obtain a more uniform distribution of the grayscale values in your image. For example, if the intensity distribution of an image results in a clump in one area of the grayscale, there might be objects that are not easily distinguished because of their similarity in color. You might want to adjust the image's intensity distribution to solve this problem by giving it a more uniform (M_UNIFORM) distribution, using *MimHistogramEqualize()*.



ORIGINAL HIST OGRAM

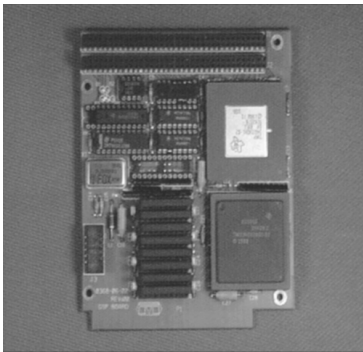


MORE UNIF ORM DIST RIBUTION

The *MimHistogramEqualize()* command first generates a histogram of the source image buffer. The histogram and a selected density function are then used to calculate a transformation LUT. If the destination buffer is an image, the transformation LUT is applied to the source buffer to produce the destination image. If the destination buffer is a LUT, the transformation LUT is copied into the destination LUT that could be used to enhance the source image, either permanently (with *MimLutMap()*) or upon display (with *MdispLut()*). The transformation LUT can also be applied directly to images as they are being grabbed by first associating the LUT buffer with the device, using *MdigLut()* and then grabbing the image.

Accentuating edges

Many applications accentuate the edges surrounding the various image objects and features to increase the quality of the image or to limit some other operation on the image. For example, finding edges of objects and features in an image can be used to highlight defects in a smooth object (as in the circuit board image below). In general, edges can be distinguished by the sharp frequency changes between two or more adjacent pixels.



- Horizontal edges are created when horizontally connected pixels have values that are different from those immediately above or below them.
- Vertical edges are created when vertically connected pixels have values that are different from those immediately to the left or right of them.
- Oblique edges are created from a combination of horizontal and vertical components.

Edge operations

There are two main types of edge operations:

- One that enhances edges to generate higher image contrast.
- One that extracts (detects) edges from the image.

Both these edge operations are types of convolutions (or neighborhood operations that replace each pixel with a weighted sum of each pixel's neighborhood). The weights applied to each neighborhood determine the type of operation that is performed. For example, certain weights produce a horizontal edge detection, others produce a vertical one. The weights are specified in a data buffer called a kernel.

Edge enhancers

You can perform an edge enhancement operation, using *MimConvolve()* with the appropriate kernel. After this operation, the amplified edges accentuate all objects in such a way as to cause the eye to see an increase in detail, generally attributable to greater picture resolution. However, this operation might not produce good results for further processing because when you enhance edges, you also enhance noise pixels.

Two predefined edge enhancers are provided by MIL:

- M_SHARPEN
- M_SHARPEN2

You can try both these kernels to see which best suits your application needs. The second kernel tends to produce more enhanced or sharpened edges.

Note, you obtain approximately the same result as M_SHARPEN by performing a Laplacian edge detection operation on the image and adding the found edges to the original picture.

Edge detection

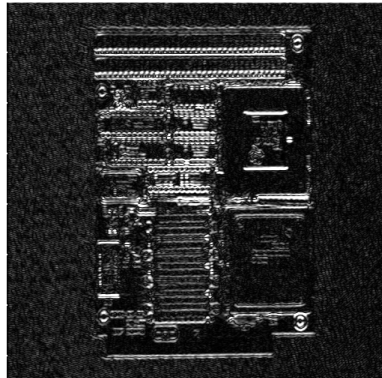
You can perform a multitude of edge detection operations, using *MimConvolve()*. This command offers predefined kernels for most common operations. Each offers some advantage over the others and should be chosen in function of your application.

- Horizontal edge detection (M_HORIZ_EDGE)
- Vertical edge detection (M_VERT_EDGE)
- Laplacian edge detection #1 (M_LAPLACIAN_EDGE)
- Laplacian edge detection #2 (M_LAPLACIAN_EDGE2)
- Compass gradient #1 (M_EDGE_DETECT)
- Compass gradient #2 (M_EDGE_DETECT2)

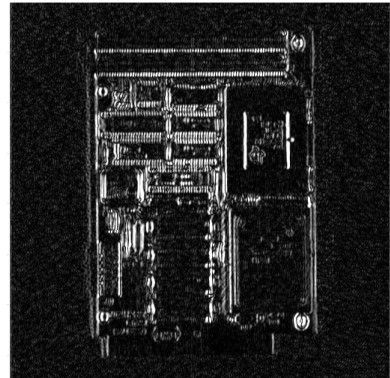
*Horizontal and vertical
edge detection*

Finding the horizontal and vertical edges in the image can be useful to enhance edges in a certain direction and remove those in another.

To extract the horizontal or vertical edges from an image, use *MimConvolve()* with the M_HORIZ_EDGE or M_VERT_EDGE predefined kernel, respectively.



Horizontal edge detection

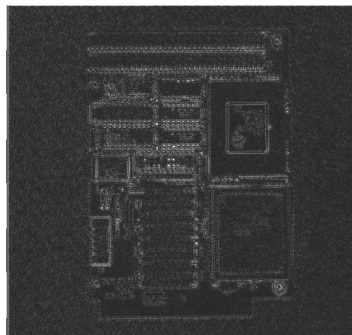


Vertical edge detection

*Laplacian edge
detection*

The Laplacian operations place emphasis on the maximum values, or peaks, within the image. This is why, once this operation has been performed, the edge representation of the image generally looks very similar to the actual image.

To extract the Laplacian edges from an image, use *MimConvolve()* with the M_LAPLACIAN_EDGE or M_LAPLACIAN_EDGE2 predefined kernel.

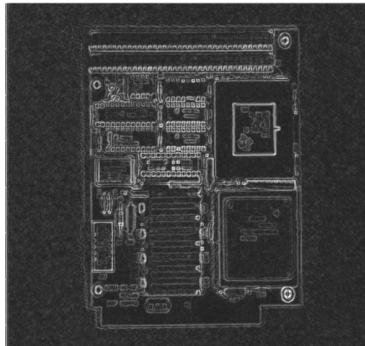


Laplacian edge detection

Compass gradient edge detection

When you perform a compass gradient edge detection operation, edges are determined from the rate of change between pixel values in the image, without regard to the direction of the edges. The resulting image contains only positive values.

You perform this operation, using *MimConvolve()* with the M_EDGE_DETECT or M_EDGE_DETECT2 predefined kernel.



Compass gradient edge detection

Arithmetic with images

It is often very useful to perform arithmetic operations on images. These operations apply the specified operator on individual pixel values in a source image or on pixels at corresponding locations in two source images. These operations, whose results do not depend on neighboring values, are known as *point-to-point operations*.

Besides arithmetic operations, the MIL image processing module includes several other point to point operations: logical, comparative, shifting, or absolute value operations.

Combining images

You can apply most of the above point-to-point operations, using *MimArith()*:

- You can add, subtract, multiply, divide, AND, NAND, OR, XOR, NOR, or XNOR two images or an image and a constant.
- You can NOT, negate, take the absolute value, or simply copy the image into the result buffer.
- You can copy a constant to the entire result buffer.

For example, for a surveillance application, it is more efficient to extract the constant background from the grabbed image and display only changes in the image. The following example shows how this can be done.

```

/* File name: msurvey.c
 * Synopsis: This program grabs an image of the expected constant dark
 *           background, and then subtracts this background image from
 *           subsequent grabbed images.
 */

#include <stdio.h>
#include <conio.h>
#include <mil.h>
#define CAMERA_SIZE_BIT 8L

void main(void)
{
    MIL_ID MilApplication,          /* Application identifier. */
    MilSystem,                     /* System identifier. */
    MilDisplay,                    /* Display identifier. */
    MilCamera,                     /* Camera identifier. */
    MilImage,                      /* Image buffer identifier. */
    GrabImage,                     /* Grab image buffer identifier. */
    BackgroundImage;              /* Background image buffer identifier. */
    long CamSizeX,                 /* Camera width variable. */
    CamSizeY;                      /* Camera height variable. */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                     &MilCamera, &MilImage);

    /* Reads camera X, Y and depth dimensions. */
    MdigInquire(MilCamera, M_SIZE_X, &CamSizeX);
    MdigInquire(MilCamera, M_SIZE_Y, &CamSizeY);

    /* Allocate a second image buffer to store the background image. */
    MbufAlloc2d(M_DEFAULT, CamSizeX, CamSizeY, CAMERA_SIZE_BIT +
                M_UNSIGNED, M_IMAGE+M_PROC, &BackgroundImage);

    /* Allocate a third image buffer to grab the changing image. */
    MbufAlloc2d(MilSystem, CamSizeX, CamSizeY, CAMERA_SIZE_BIT +
                M_UNSIGNED, M_IMAGE+M_PROC+M_GRAB, &GrabImage);

    /* Grab the background image in the display buffer. */
    MdigGrabContinuous(MilCamera, MilImage);

    /* When a key is pressed, halt. */
    printf("Point your camera at a constant dark ");
    printf("background and adjust the focus.\n");
    printf("Press <Enter> to continue.\n");
    getchar();
    MdigHalt(MilCamera);

    (cont. ...)

```

```

/* Copy the displayed buffer into the background buffer. */
MbufCopy(MilImage, BackgroundImage);

/* When a key is pressed, halt. */
printf("Continuous subtraction in progress...\n\n");
printf("Keeping your camera in the same position, create motion\n");
printf("with a bright object in front of the background.\n");
printf("Press <Enter> to stop operation and end.\n");

/* Grab and subtract background in loop. */
while (!kbhit())
{
    MdigGrab(MilCamera, GrabImage);
    MimArith(GrabImage, BackgroundImage, MilImage, M_SUB_ABS);
}
getch();

/* Release defaults and image. */
MbufFree(GrabImage);
MbufFree(BackgroundImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, MilCamera,
                MilImage);
}

```

Mapping an image

You can perform complex operations (such as scaling and logarithms) on an image buffer, using *MimLutMap()*. This function performs the operation simply by mapping the source image buffer through a specified lookup table (LUT) and storing results in the specified destination image buffer.

You allocate a LUT buffer, using *MbufAlloc1d()*, specifying the buffer attribute as *M_LUT*. You can assign mapping values to it by copying data from a Host generated buffer (for example, an array) into it, using *MbufPut1d()*. You can also generate data directly into a LUT buffer according to a specified function, using *MgenLutFunction()*. If you simply want to invert the image or set the image to a constant, you can alternatively use *MgenLutRamp()* to generate an inverse ramp.

Erosion and dilation

Especially during cell analysis, it can be important to know the growth stages of cell particles. Using the image processing erosion and dilation operations, you can view the possible growth stages of these particles.

- Erosion operations peel off layers from objects or particles, removing extraneous pixels and small particles from the image.
- Dilation operations add layers to objects or particles, enlarging any particle. Dilation can return eroded particles to their original size (but not necessarily to their exact original shape).

Erosion and dilation are neighborhood operations that determine each pixel's value according to its geometric relationship with neighborhood pixels, and as such, are part of a group of operations known as *morphological operations*.

They are also the basic operations used to perform the opening and closing operations discussed in the previous chapter.

Note, zero pixels are considered background, while non-zero pixels are considered foreground and part of objects.

Basic erosion

You can perform a basic erosion operation on 3 by 3 neighborhoods, using *MimErode()*.

- If the erosion mode is set to M_BINARY, any pixel whose **neighborhood** is not completely white (any non-zero pixel is considered white) is changed to black (0 is considered black).
- If the erosion mode is set to M_GRAYSCALE, each pixel is replaced with the minimum value in its neighborhood.

You can use the iteration parameter of *MimErode()* to perform an erosion on larger neighborhoods. Iterating the erosion is the equivalent to performing an erosion on a $(1 + 2*i)$ by $(1 + (2*i))$ neighborhood where i is the number of iterations. For example, two iterations of a 3x3 erosion is equivalent to a 5x5 erosion, and three iterations is equivalent to a 7x7 erosion.

Basic dilation

You can perform a basic dilation operation on 3 by 3 neighborhoods, using *MimDilate()*.

- If the dilation mode is set to M_BINARY, any pixel that has one or more white pixels (any non-zero pixel is considered white) in its neighborhood is set to white (0xff in an 8-bit image).
- If the dilation mode is set to M_GRAYSCALE, each pixel is replaced with the maximum value in its neighborhood.

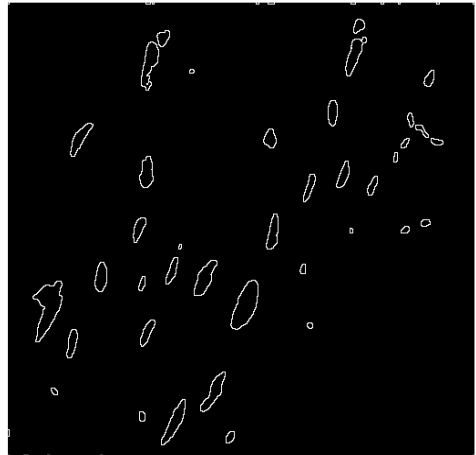
The *MimDilate()* command is similar to the *MimErode()* command in that iterating it will effectively cause a dilation on larger neighborhoods. Iterating the dilation is the equivalent to performing a dilation on a $(1 + (2*i))$ by $(1 + (2*i))$ neighborhood where i is the number of iterations. For example, two iterations of a 3x3 dilation is equivalent to a 5x5 dilation, and three iterations is equivalent to a 7x7 dilation.

An example...

You can use erosion or dilation to find the perimeter of objects. Erode or dilate a binary image and 'XOR' the result with the original image, using *MimArith()*.



Binary image



Exoskeletons of objects

The following example shows how to obtain the exoskeletons of objects in an image.

```

/* File name: mperim.c
 * Synopsis: This program finds the exoskeletons (perimeters) of
 *           dark objects in an image.
 */

#include <stdio.h>
#include <mil.h>

/* Source MIL image file specifications. */
#define IMAGE_FILE           "cell.mim"
#define IMAGE_WIDTH          512L
#define IMAGE_HEIGHT         480L
#define IMAGE_DEPTH          8L
#define IMAGE_THRESHOLD_VALUE 128L

/* Small particle radius (in pixels). */
#define SMALL_PARTICLE_RADIUS 2L
void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem,             /* System identifier. */
    MilDisplay,            /* Display identifier. */
    MilImage,              /* Image buffer identifier. */
    BinImage,              /* Binary image buffer identifier. */
    DilBinImage;           /* Dilated binary image buffer identifier. */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                     M_NULL, &MilImage);

    /* Allocate 2 binary image buffers for fast processing. */
    MbufAlloc2d(M_DEFAULT, IMAGE_WIDTH, IMAGE_HEIGHT,
                1*M_UNSIGNED, M_IMAGE+M_PROC, &BinImage);
    MbufAlloc2d(M_DEFAULT, IMAGE_WIDTH, IMAGE_HEIGHT,
                1*M_UNSIGNED, M_IMAGE+M_PROC, &DilBinImage);

    /* Load source image into an image buffer. */
    MbufLoad(IMAGE_FILE, MilImage);

    /* Pause to show the original image. */
    printf("This program finds the exoskeletons of\n");
    printf("the particles in the displayed image.\n");
    printf("Press <Enter> to continue.\n");
    getchar();

    (cont. ...)

```

```

/* Binarize the image. */
MimBinarize(MilImage, BinImage, M_LESS OR EQUAL,
            IMAGE_THRESHOLD_VALUE, M_NULL);

/* Remove small particles. */
MimOpen(BinImage, BinImage, SMALL_PARTICLE_RADIUS, M_BINARY);

/* Dilate image (adds one pixel around all objects). */
MimDilate(BinImage, DilBinImage, 1L, M_BINARY);

/* XOR the dilated image with the original image. */
MimArith(BinImage, DilBinImage, BinImage, M_XOR);

/* Convert the binary image to a visible grayscale image (0-0xFF). */
MimBinarize(BinImage, MilImage, M_GREATER, 0, M_NULL);

/* Pause to show the resulting image. */
printf("\nExoskeletons of the object's perimeters are being displayed.\n");
printf("Press <Enter> to end.\n");
getchar();

/* Free all allocations. */

MbufFree(BinImage);
MbufFree(DilBinImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

The Chamfer 3-4 transform

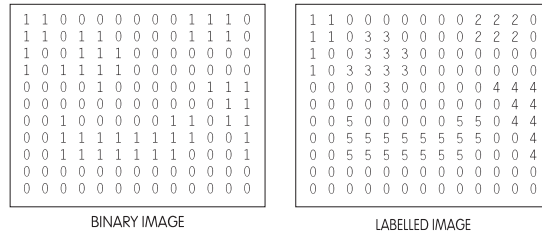
The Chamfer 3-4 transform (`M_CHAMFER_3_4`), like the Chessboard transform, determines the minimum distance using horizontal, vertical, or diagonal steps. However, horizontal and vertical steps are counted as 3 and diagonal steps as 4. This allows the transform to better approximate the true (Euclidean) distance between two pixels. However, it requires that the destination buffer be large enough to hold a number at least three times the maximum distance from a foreground to a background pixel. For example, an 8-bit buffer (255 max) can be used for a maximum distance of 85 pixels and a 16-bit buffer (65535 max) for a maximum distance of 21845 pixels.

Labeling

You can label objects or particles (known as blobs) in an image with *MimLabel()*. Labeling is useful for several operations:

- Identifying and distinguishing blobs.
- Finding the area of a blob. Once a blob is labeled, you find the area by generating a histogram and noting the number of pixels associated with that label value.
- Counting the number of blobs in the image. The label number assigned to the last blob is also the number of blobs in the image (assuming there are fewer blobs than possible labels).
- Using the result as a source for a conditional copy to eliminate some blobs (*MimClip()*).

The *MimLabel()* command numerically identifies each blob in the specified image. Each non-zero pixel within a blob is given the same numerical value, and blobs within an image are given consecutive values.



You can specify that the operation is performed using one of two types of connectivity modes:

- **M_4_CONNECTED:** If two pixels touch on the vertical or horizontal, they are considered part of the same blob.
- **M_8_CONNECTED:** If two pixels touch on the vertical, horizontal, or diagonal, they are considered part of the same blob.

To distinguish between touching blobs, separate the blobs by performing an erosion operation before the labeling operation.

Chapter 6: Advanced image processing

This chapter describes different advanced image processing techniques.

Advanced image processing

Besides the image processing functions discussed in previous chapters, MIL contains more advanced image processing functions. These advanced functions, among other things, allow you to remove noise, separate objects from their background, and correct image distortions. They include neighborhood operations using custom structuring elements or kernels, frequency transforms, watershed transforms, and warpings.

Custom spatial filters

Spatial filtering operations include operations that can enhance and smooth images, accentuate image edges, and remove ‘noise’ from the image.

Spatial filters are operations that compute results based on an underlying neighborhood process: the weighted sum of a pixel value and its neighbors’ values. The weights are known as the kernel values. These kernel values determine the type of spatial filter. For example, applying the following kernel results in a sharpening of the image:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Whereas, applying the following kernel smooths an image (it also increases the intensity of the image by a factor of 16, so you will need to normalize the convolution result):

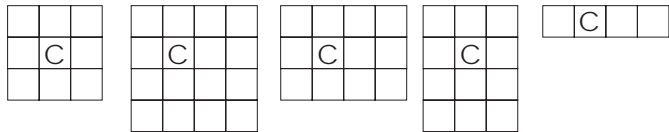
$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

If the predefined kernels provided by *MimConvolve()* do not meet your requirements, you can create your own spatial filtering operation by providing your own kernel.

Defining your own kernel

To define your own kernel:

1. Allocate a kernel buffer (`M_KERNEL`), using `MbufAlloc2d()`. The dimensions of the kernel determine the size of the neighborhood that is used in the operation. The result of the operation is stored in the destination buffer at the location corresponding to the kernel's center pixel. When the kernel has an even number of rows and/or columns, the center pixel is considered to be the top-left pixel of the central elements in the neighborhood.

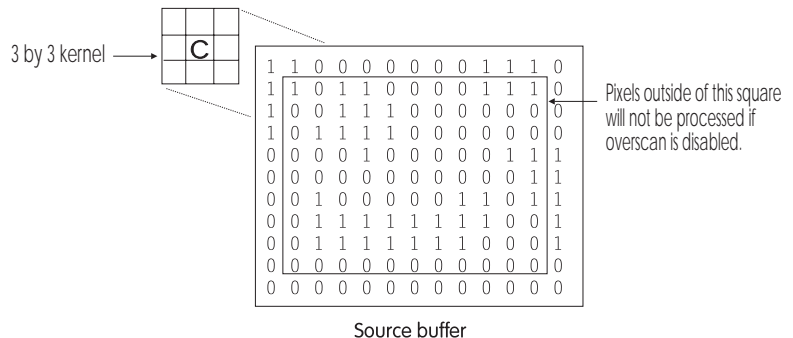


Examples of neighborhoods and their center pixel.

2. Load the kernel values into this kernel buffer, using `MbufPut()` or `MbufPut2d()`.

You can modify the default operation flags associated with custom kernels, using `MbufControlNeighborhood()`. These operation flags determine how the convolution operation will be handled. You can control:

- How the operation handles the borders (overscan) of the source buffer. If overscan is disabled, the bordering pixels of the source image are not processed if additional processing time is implicated. For example, if you are using a 3 by 3 kernel with a normal center pixel, and overscan is disabled, the pixels on the borders of the source buffer are not processed if processing time can be saved.



To process the bordering pixels, specify an overscan. A transparent overscan uses the parent buffer to provide the overscan pixels needed for the border calculation. Note, if the parent buffer is not available, a mirror overscan is performed. A mirror overscan specifies that the overscan pixels will be a mirror copy of the source buffer's bordering pixels. A replacement overscan allows you to specify a specific value to use for the overscan pixel values during processing.

- Whether or not the absolute value of the result is taken.
- The division (normalization) factor to apply to the result.
- Whether or not to saturate the result.
- The position of the center pixel.

An example...

The following is an example of a spatial filtering operation using a custom 3 by 3 kernel.

```
/* File name: mconvol.c
 * Synopsis: This program loads an image and then does a
 *           3x3 custom convolution (smoothing) on it.
 */

#include <stdio.h>
#include <mil.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE    "wafer.mim"
#define IMAGE_WIDTH   512L
#define IMAGE_HEIGHT  480L

/* Kernel informations. */
#define KERNEL_WIDTH   3L
#define KERNEL_HEIGHT  3L
#define KERNEL_DEPTH   8L

/* Average kernel information data definition. */
unsigned char KernelData[KERNEL_HEIGHT][KERNEL_WIDTH] =
    { {1, 2, 1},
      {2, 4, 2},
      {1, 2, 1}
    };
```

(cont.)

```

void main(void)
{
    MIL_ID MilApplication,    /* Application identifier.      */
    MilSystem,               /* System identifier.           */
    MilDisplay,              /* Display identifier.          */
    MilImage,                /* Image buffer identifier.     */
    MilSubImage,             /* Sub-image buffer identifier. */
    MilKernel;               /* Custom kernel identifier.    */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                     M_NULL, &MilImage);

    /* Restrict the region to be processed to the image size. */
    MbufChild2d(MilImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage);

    /* Load source image into an image buffer. */
    MbufLoad(IMAGE_FILE, MilSubImage);

    /* Pause to show the original image. */
    printf("This program does a convolution on the displayed image.\n");
    printf("It uses a custom smoothing kernel.\n");
    printf("Press <Enter> to continue.\n");
    getchar();

    /* Allocate a MIL kernel. */
    MbufAlloc2d(MilSystem, KERNEL_HEIGHT, KERNEL_WIDTH,
                KERNEL_DEPTH+M_UNSIGNED, M_KERNEL, &MilKernel);

    /* Put the custom data in it. */
    MbufPut(MilKernel, KernelData);

    /* Set a normalization (divide) factor to have a kernel with a sum equal
     * to one. */
    MbufControlNeighborhood(MilKernel, M_NORMALIZATION_FACTOR, 16L);

    /* Convolute the image using the kernel. */
    MimConvolve(MilSubImage, MilSubImage, MilKernel);

    /* Pause to show the result. */
    printf("\n");
    printf("The original image was smoothed using a custom kernel.\n");
    printf("Press <Enter> to terminate.\n");
    getchar();

    /* Free all allocations. */
    MbufFree(MilKernel);
    MbufFree(MilSubImage);
    MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

Custom morphological operations

Morphological operations are neighborhood operations that compute new values according to geometric relationships and matches of known patterns in the input image. The *MimMorphic()* command supports different types of morphological operations:

- Erosion
- Dilation
- Thinning
- Thickening
- Matching
- Hit or miss transformation

Different geometric relationships for each of these operations are specified, using a structuring element.

Defining your own structuring element

To define your own structuring element:

1. Allocate a structuring element buffer (`M_STRUCT_ELEMENT`) , using *MbufAlloc2d()*. The dimensions of the structuring element determine the size of the neighborhood that is used in the operation. The result of the operation is stored in the destination buffer at the location that corresponds to the structuring element's center pixel. When the structuring element has an even number of rows and/or columns, the center pixel is considered to be the top-left pixel of the central elements in the neighborhood (see custom spatial filters).
2. Load the structuring element values into this buffer, using *MbufPut()* or *MbufPut2d()*. Give the structuring element values according to the morphological operation that is to be performed. For binary and some grayscale operations, the structuring element values must be 0, 1, or `M_DONT_CARE` (the latter means that the corresponding neighbors are not considered in the comparison) . For other grayscale operations, any structuring element value can be used, including `M_DONT_CARE`.

For custom structuring elements, you can use *MbufControlNeighborhood()* to control how the operation handles the borders (overscan) of the source buffer (see custom spatial filters) and the position of the neighborhood's center pixel.

Erosion and dilation

Two fundamental morphological operations are erosion and dilation. These functions allow you to view the possible growth stages of an object in the foreground (non-zero pixels) of an image.

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0	0	0
1	0	0	0	1	1	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0	0	0	0
1	0	0	1	1	1	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	1	1	0	0
0	0	0	0	1	0	0	0	1	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0
0	1	1	0	1	0	0	0	1	1	1	0
0	1	1	0	1	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0

Original image

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Eroded image

0	0	0	1	1	1	1	0	0	0	0	0
1	1	0	1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0

Dilated image

1	1	1
1	1	1
1	1	1

Structuring element

Note: In these images, pixels represented with the value 1, actually have the maximum buffer value (for example, they have the value 0xffff in a 16-bit image).

There are two versions of erosion and dilation:

■ Erosion (M_ERODE):

- Binary erosion: If the structuring element does not match the corresponding neighborhood values exactly, the center pixel is set to zero; otherwise, it remains unchanged. In effect, binary erosion peels off layers of objects.
- Grayscale erosion: Subtracts each structuring element value from the corresponding pixel value in the neighborhood, and then replaces the center pixel of the neighborhood with the minimum value from the resulting neighborhood values.

■ Dilation (M_DILATE):

- Binary dilation: If any of the structuring element values match the corresponding neighborhood values, the center pixel is set to the maximum value of the buffer (e.g. 0xff for an 8-bit buffer); otherwise, it remains unchanged. In effect, binary dilation adds layers to the objects.
- Grayscale dilation: Adds each structuring element value to the corresponding pixel value in the neighborhood, and then replaces the center pixel of the neighborhood with the maximum value from the resulting neighborhood values.

Note, in binary mode, erosion of the white pixels is the same as dilation of the black pixels.

If the processing mode is set to M_BINARY, a binary erosion or dilation is performed and all non-zero pixels are considered as 1's; otherwise, the grayscale version of these operations is performed.

Use *MblobReconstruct()* to perform a conditional dilation.

Using standard erosion
and dilation

MIL also supports *MimErode()* and *MimDilate()*, commands specialized in performing the most standard form of erosion and dilation operation. These operations use the following structuring element when performing in binary mode:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

And use the following structuring element in grayscale mode:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

In other words, these commands execute a simple 3 by 3 minimum or maximum operation without adding or subtracting anything from the pixel.

For example, to perform the most standard dilation operation on a source image buffer, use *MimDilate()* with the processing mode set to *M_BINARY*, or use *MimMorphic()* with a 3 x 3 structuring element of ones and the processing mode set to *M_BINARY*. Note, in general the standard version is faster.

An example

The following example shows how to define your own structuring element. It demonstrates, on an image with rounded objects, the difference between performing the standard opening operation, *MimOpen()*, and performing a custom opening with a circular type structuring element. Note, the latter preserves the original shape of the objects better than the square structuring element of the standard erosion.

```

/* File name: mopen.c
 * Synopsis: This program loads an image of a tissue sample and then
 *           performs opening operations on it using two methods. */

#include <stdio.h>
#include <mil.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE           "cell.mim"
#define IMAGE_WIDTH          240L
#define IMAGE_HEIGHT         240L
#define IMAGE_DEPTH          8L
#define IMAGE_THRESHOLD_VALUE 128L

/* Structuring element information. */
#define STRUCT_ELEM_WIDTH    5L
#define STRUCT_ELEM_HEIGHT   5L
#define STRUCT_ELEM_DEPTH    32L

/* Small particle radius (in pixels). */
#define SMALL_PARTICLE_RADIUS 2L

void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem,             /* System identifier. */
    MilDisplay,            /* Display identifier. */
    MilImage,              /* Image buffer identifier. */
    BinImage,              /* Binary Image buffer identifier. */
    MilSubImage0,          /* Sub-image buffer identifier for original image. */
    MilSubImage1,          /* Sub-image buffer identifier for binarization. */
    MilSubImage2,          /* Sub-image buffer identifier for common open. */
    MilSubImage3;          /* Sub-image buffer identifier for customized open */
    MIL_ID StructElem;     /* Structuring element buffer. */

    (cont. ...)
```

```

/* Structuring element data definition. */
long StructArray[STRUCT_ELEM_HEIGHT][STRUCT_ELEM_WIDTH] =
{ {M_DONT_CARE, M_DONT_CARE, 1, M_DONT_CARE, M_DONT_CARE},
  {M_DONT_CARE, 1, 1, 1, M_DONT_CARE},
  {1, 1, 1, 1, 1},
  {M_DONT_CARE, 1, 1, 1, M_DONT_CARE},
  {M_DONT_CARE, M_DONT_CARE, 1, M_DONT_CARE, M_DONT_CARE} };

/* Allocate defaults. */
MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
M_NULL, &MilImage);

/* Allocate a binary image buffer for fast processing. */
MbufAlloc2d(M_DEFAULT, IMAGE_WIDTH, IMAGE_HEIGHT,
1*M_UNSIGNED, M_IMAGE+M_PROC, &BinImage);

/* Define four processing buffers in the display buffer, restricting the
 * regions to be processed to the top left corner of the original image. */
MbufChild2d(MilImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage0);
MbufChild2d(MilImage, IMAGE_WIDTH, 0L, IMAGE_WIDTH, IMAGE_HEIGHT,
&MilSubImage1);
MbufChild2d(MilImage, 0L, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_HEIGHT,
&MilSubImage2);
MbufChild2d(MilImage, IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_WIDTH,
IMAGE_HEIGHT, &MilSubImage3);

/* Load source image into image buffer for original image. */
MbufLoad(IMAGE_FILE, MilSubImage0);

/* Allocate a structuring element. */
MbufAlloc2d(MilSystem, STRUCT_ELEM_WIDTH, STRUCT_ELEM_HEIGHT,
STRUCT_ELEM_DEPTH + M_SIGNED, M_STRUCT_ELEMENT, &StructElem);

/* Load buffer with data. */
MbufPut2d(StructElem, 0L, 0L, STRUCT_ELEM_WIDTH,
STRUCT_ELEM_HEIGHT, StructArray);

/* Pause to show the original image. */
printf("This program does the opening of an image using two different\n");
printf("structuring elements.\nPress <Enter> to continue.\n");
getchar();

/* Smooth the image to remove noise. */
MimConvolve(MilSubImage0, MilSubImage0, M_SMOOTH);

```

(cont. ...)

```

/* Binarize the image so that particles are represented in white and
 * the background in black, placing result in subimage1 on the display. */
MimBinarize(MilSubImage0, MilSubImage1, M_LESS_OR_EQUAL,
            IMAGE_THRESHOLD_VALUE, M_NULL);

/* Copy the binarized image to a binary buffer for fast processing */
MbufCopy(MilSubImage1, BinImage);

/* Opening using common method, placing result in subimage2 on the
 * display. */
MimOpen(BinImage, MilSubImage2, SMALL_PARTICLE_RADIUS,
        M_BINARY);

/* Opening (Erode and Dilate) using customized method, placing
 * result in subimage3 on the display. */
MimMorphic(BinImage, BinImage, StructElem, M_ERODE,
            SMALL_PARTICLE_RADIUS/2, M_BINARY);
MimMorphic(BinImage, MilSubImage3, StructElem, M_DILATE,
            SMALL_PARTICLE_RADIUS/2, M_BINARY);

/* Pause to show the opened particle(s). */
printf("The top right image is the binarized source image. When \n");
printf("opening it using the standard method, the bottom left image \n");
printf("results, whereas when opening it using the customized method,\n");
printf("the bottom right image results and best preserves the original\n");
printf("shape of the objects.\nPress <Enter> to end.\n");
getchar();

/* Free structuring element buffer. */
MbufFree(StructElem);

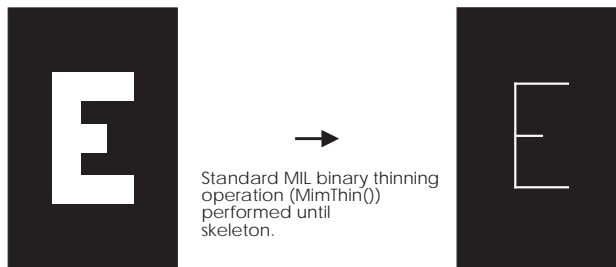
/* Free the allocated buffers. */
MbufFree(MilSubImage0);
MbufFree(MilSubImage1);
MbufFree(MilSubImage2);
MbufFree(MilSubImage3);
MbufFree(BinImage);

/* Release defaults. */
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

Thinning and thickening

You can reduce or enlarge objects in the foreground (non-zero pixels) of an image, using operations based on a rigid match of the pixel's neighborhood and the structuring element. Using a thickening operation, you can enlarge the object and perform such operations as a convex hull. Using a thinning operation, you can reduce objects and perform such operations as finding their skeleton.



You can perform a thinning or thickening operation with a specified structuring element, using *MimMorphic()*. These operations are typically performed several times, using a different structuring element so that the required pattern is sought in each direction.

You can also perform standard thinning or thickening operations with *MimThin()* or *MimThick()*, respectively.

There are two versions of thinning and thickening:

Thinning objects

■ Thinning (M_THIN):

- Binary thinning: This operation replaces the center pixel by the value zero if a pixel's neighborhood matches the structuring element exactly. However, if the neighborhood does not match, the pixel value remains unchanged.
- Grayscale thinning:
 - if $\text{MAX}(0) < \text{center pixel} \leq \text{MIN}(1)$
 - center pixel = $\text{MAX}(0)$
 - else
 - center pixel is unchanged

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

Thickening objects

■ Thickening (M_THICK):

- Binary thickening: This operation replaces the center pixel by the maximum value of the buffer (for example, 0xff for an 8-bit buffer) if the pixel's neighborhood matches the structuring element exactly. However, if the neighborhood does not match, the pixel value remains 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 structuring element, and $\text{MIN}(1)$ is the minimum of all pixels in the neighborhood that correspond to one in the structuring element.

Both versions of thinning and thickening take structuring elements containing only 0's, 1's, and 'don't care' values.

If the processing mode is set to `M_BINARY`, a binary thinning or thickening is performed, otherwise the grayscale version of these operations is performed.

Matching

Matching allows you to determine the degree of similarity between certain areas of the image and a pattern (specified by a structuring element). The operation takes a binary or grayscale source image and produces a corresponding grayscale image, wherein the value of each pixel is equal to the total number of matches between the neighborhood of the source image's corresponding pixel and the structuring element values.

Searching for hits or misses

You can determine which pixels have neighborhoods that match a pattern exactly by performing a 'hit or miss' operation. When the neighborhood of a source image's pixel matches the pattern exactly, the value of the corresponding pixel in the destination image is the maximum value of the buffer (e.g. 0xff for an 8-bit buffer). When the neighborhood does not match exactly, the pixel value is zero.

Connectivity mapping

In some cases, an image must undergo several passes with different structuring elements. This can be very time-consuming. To perform such operations more efficiently, you should consider the connectivity (or cellular) mapping command, *MimConnectMap()*. This command reduces a serial operation to a parallel operation.

The *MimConnectMap()* command calculates a connectivity code for each pixel in a binary source image and then maps these codes through the specified 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:

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

The pixels are connected and mapped as follows:

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

$$\text{Result} = \text{LUTMAP}(\text{connectivity code})$$

Program the LUT with values that would result if the required structuring elements were applied. As each connectivity code has 9 bits, you should supply a LUT buffer with at least 512 (2 to the power of 9) entries.

Fast Fourier Transform

A Fast Fourier Transform (FFT) is used to identify any consistent spatial patterns in an image (which can be caused, for example, by systematic noise). MIL can perform one or two dimensional FFTs using ***MimTransform()***. For 1-D transforms, each row or column is treated as a 1-D signal. This method of transform separates a one dimensional signal into a set of sine and cosine waves of different frequencies. For a two-dimensional signal (image), it can be interpreted as the decomposition of an image into a set of 2-D patterns. The composition of these waves make up the original waveform.

The forward Fourier transform is defined as:

$$F(u, v) = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x, y) \exp \frac{(2\pi i x u)}{N} \exp \frac{(2\pi i y v)}{M}$$

Where u and v are coordinates in the frequency domain and x and y are coordinates in the spatial domain. A forward FFT yields a real (R) and an imaginary (I) component of the image in a frequency domain (spectrum).

The reverse Fourier transform is defined as:

$$f(x, y) = \frac{1}{nm} \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} F(u, v) \exp \frac{(-2\pi i x u)}{N} \exp \frac{(-2\pi i y v)}{M}$$

where x and y are the coordinates in the spatial domain and u and v are coordinates in the frequency domain.

Magnitude and phase

For a more visual understanding of the FFT results, you can calculate the phase and magnitude, also using the ***MimTransform()*** function.

The magnitude is calculated as $\sqrt{R^2 + I^2}$, where R and I are real

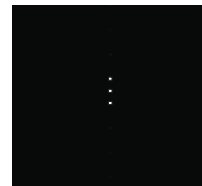
and imaginary components of the image, respectively.

MimTransform() uses the flag `M_MAGNITUDE` to obtain this value.

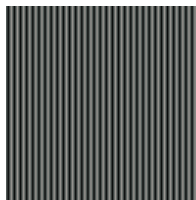
The following figures show single-frequency images and their magnitude. Because single-frequency images contain only one spatial frequency component, their corresponding frequency images appear as a single point of brightness with their associated negative-frequency mirrors. Note that the points in the frequency domain appear in the direction of the pattern. The distance between the points and the center (DC component) represents the frequency of the pattern.



Vertical low frequency image.



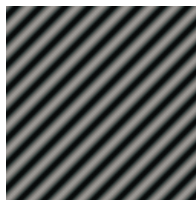
The image in frequency domain.



Horizontal high frequency image.



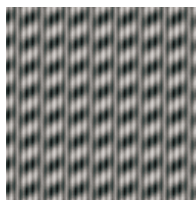
The image in frequency domain.



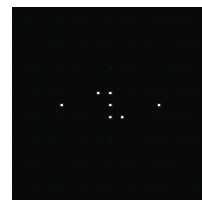
Diagonal low frequency image.



The image in frequency domain.



A combination of the three frequencies.



The image in frequency domain.

The spatial shift of each pattern in the image (in degrees) is called the phase. It is calculated using the formula $\text{atan}(I/R)$. Use the flag `M_PHASE` to obtain the phase.

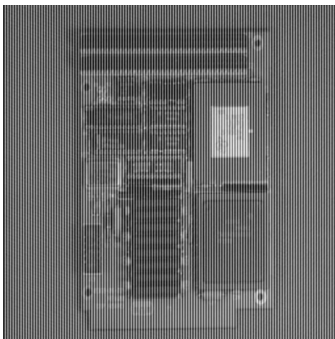
Filtering an image

To filter constant spatial patterns using FFTs:

1. Perform a forward transform (`M_FORWARD`) calculating the magnitude (`M_MAGNITUDE`) of the image. Scale the image within displayable range using `M_LOG_SCALE` (this applies the formula, $c \log[1 + |F(u, v)|]$).
2. Find the frequency components representing the noise and design a mask to remove these components.
3. Once the mask is designed, perform a simple transform to obtain the real and imaginary components, this time without calculating the magnitude.
4. Apply the mask to both the real and imaginary components of the image in the frequency domain.
5. Finally, perform a reverse transform to obtain a filtered image.

If you know the frequency of the noise pattern and have designed the mask, you need only perform steps 3 to 5.

Image (with noise pattern)



Frequency domain

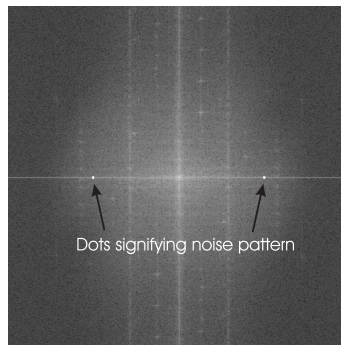
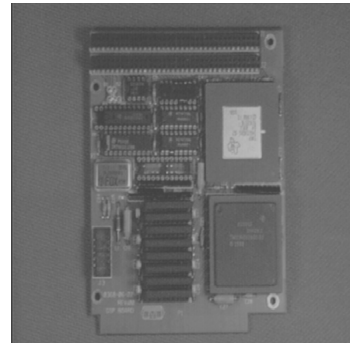


Image (noise pattern removed)



Following is an excerpt from the example *mfft.c*. It performs an FFT on an image with a vertical noise pattern. A forward transform is performed to obtain the real and imaginary components of the image. The values of locations corresponding to the noise pattern are set to 0. Finally, a reverse transform is performed to obtain a spatial image without the noise pattern.

```

/* File name: mfft.c
 * Synopsis: This program uses the Fast Fourier Transform to filter an image.
 */
void main(void)
{
    .
    .
    .
    .

    /* Compute the Fast Fourier Transform of the image. */
    MimTransform(MilSubImage00, M_NULL, MilTransformReal,
                MilTransformIm, M_FFT, M_FORWARD+M_CENTER);

    /* Filter the image in the frequency domain. */
    MbufPut2d(MilTransformReal, 63, 127, 1, 1, &ZeroVal);
    MbufPut2d(MilTransformIm,   63, 127, 1, 1, &ZeroVal);
    MbufPut2d(MilTransformReal, 191, 127, 1, 1, &ZeroVal);
    MbufPut2d(MilTransformIm,   191, 127, 1, 1, &ZeroVal);

    /* Recover the image in the spatial domain. */
    MimTransform(MilTransformReal, MilTransformIm,
                MilSubImage01, M_NULL, M_FFT, M_REVERSE+M_CENTER);
    .
    .
    .
}

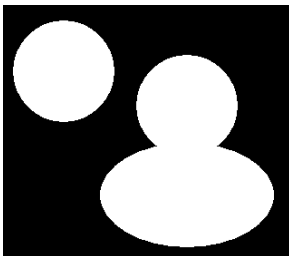
```

Watershed transformations

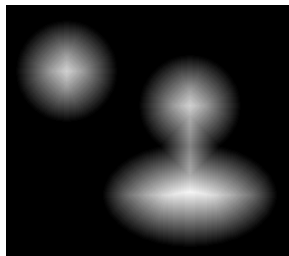
You can perform watershed transformations using *MimWatershed()*. A watershed transformation is generally used in conjunction with other processing operations to segment images, that is, to separate objects from their background and/or from each other.

To understand what a watershed transformation is, it is useful to think of an image as a topographic surface. In other words, the value of each pixel represents a certain height, with the lowest pixel value (the darkest pixel) representing the point of lowest elevation and the highest pixel value (the brightest pixel) representing the point of highest elevation. A *minimum* in the image is defined as a pixel or a set of connected pixels that is lower in value (or elevation) than all its neighboring pixels. A *maximum* is a pixel or a set of connected pixels which is higher in value (elevation) than all its neighboring pixels. (Pixels are connected if they are vertically, horizontally, or diagonally adjacent). A *catchment basin* refers to a minimum or maximum's zone of influence. For example, for a minimum, a catchment basin refers to the set of pixels which, if a drop of water were to fall from one of these pixels, it would eventually reach that minimum.

MimWatershed() labels an image's catchment basins and/or builds dividing lines between the catchment basins. These dividing lines are known as the *watershed lines* of the image. Note that catchment basins can be determined from the image's minima or its maxima.



An image with three objects.



A distance transformation produces a maximum in each object.



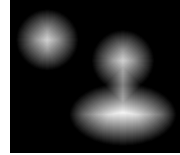
The watershed lines and labelled catchment basins of the resulting image.

Using watersheds to separate touching objects

You can use *MimWatershed()* in conjunction with *MimDistance()* and *MimArith()* to separate touching objects in a binary image.



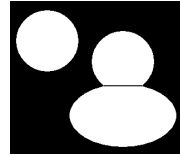
a) Touching objects in a binary image.



b) A distance transformation of the image. Note that there is a maximum in each object. In addition, the touching objects have touching zones of influence.



c) A watershed transformation of the resulting image, showing only watershed lines. Several options were set to prevent watershed lines from extending into the background and to force the lines to be straight.



d) An AND operation between the result of the watershed transformation and the original image results in the above.

To summarize:

1. Perform a distance transformation on the image. This will result in a grayscale image with a maximum in each object.
2. Perform a watershed transformation on the resulting image. Note that:
 - Catchment basins must be determined from the image's maxima rather than its minima since *MimDistance()* produces a maximum in each object.
 - The transform must show only watershed lines. To save time, you can prevent watershed lines from extending into the background. You can also specify that the watershed lines be straight. (These options are discussed in more detail later.)

- You must specify the minimum variation in gray levels between extrema that is required to produce a new catchment basin (this is discussed in more detail later). In general, when separating touching objects in a binary image, a low value (2) is usually sufficient.
3. Perform an AND operation between the original image and the result of step 2, using *MimArith()*.

Using watersheds to separate objects from their background

MimWatershed() can be used in conjunction with other processing operations to separate objects from their background. For example, if the objects have well-defined edges, an edge detection will produce a maximum along the edges of each object. These maxima will define each object as a catchment basin since they produce a minimum in each object. A watershed transformation will then label the catchment basins, effectively segmenting the image.



An image with well-defined edges.



An edge detection performed on the image.



A watershed transformation of the resulting image, showing labelled catchment basins.

To summarize:

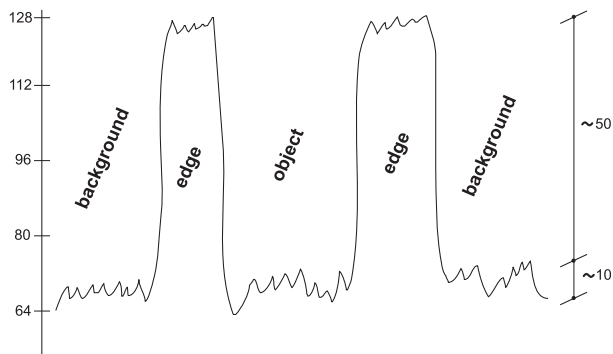
1. Perform an edge detection on the image.
2. Determine, through some analysis of the resulting image, the minimum variation in gray levels between extrema that is required to produce a new catchment basin (this is discussed in the next section).
3. Perform a watershed transformation on the resulting image. You must specify that catchment basins be determined from the image's minima. In addition, the transform should only show labelled catchment basins.

Minimum variation between extrema

A typical image contains a lot of unwanted extrema, often due to noise. If catchment basins were determined from each extremum, the transform would segment various noise areas, resulting in over-segmentation. The **MinimumVariation** parameter of *MimWatershed()* allows you to prevent such over-segmentation while still separating objects from their background.

The **MinimumVariation** parameter specifies the minimum variation in gray levels between extrema that is required to produce a new catchment basin. In other words, a new catchment basin will be determined from an extremum only when the difference in gray-levels between it and its closest extrema is greater than the value specified by the **MinimumVariation** parameter.

The following shows the line profile across an object (after an edge detection was performed on the image). In this case, extrema in the background (as well as within the object) have a maximum gray-level variation of about 10. The minimum gray-level variation between the background and the edges is about 50. In this case, therefore, the **MinimumVariation** parameter should be set to a value somewhere between 10 and 50, for example, 30. Note that, if it is set above 50, the object will not be separated from the background since its extrema will not produce a new catchment basin.



The default value for the **MinimumVariation** parameter is 1, which means that each extremum produces a catchment basin.

Using marker images

If you are able to approximate the location of your objects in an image (either through some pre-processing or through some previous knowledge of the image), you might want catchment basins determined from a separate image (known as a *marker image*), instead of from extrema in the source image. In this case, each group of touching pixels with the value zero in the marker image (known as a *marker*) produces a catchment basin in the corresponding area of the source image. Specifically, each marker in the marker image forces a minimum in the corresponding area of the source image. Pixels in the marker image are considered touching if they are vertically, horizontally, or diagonally adjacent, that is, if they are “8-connected”.

If you use a marker image, there is no need to determine what value to set the **MinimumVariation** parameter in order to properly segment the image, since you mark off the extrema in a separate image. Marker images are also useful in preventing over-segmentation since you control not only the location of the extrema but also the number of extrema. However, if you cannot locate your objects in the image, you should not be using a marker image.

Note that catchment basins can be determined from markers in the marker image as well as from extrema in the source image. In this case, supply a marker image to *MimWatershed()* and also specify the minimum variation in gray-levels in the source image required to produce a new catchment basin.

Style of the watershed lines

Watershed lines can be *8-connected* or *4-connected* (set the **ControlFlag** parameter of *MimWatershed()* to `M_4_CONNECTED` or `M_8_CONNECTED`). In addition, they can be traced exactly or forced to be straight (set **ControlFlag** to `M_REGULAR` or `M_STRAIGHT_WATERSHED`).

*8-connected vs.
4-connected*

8-connected watershed lines consist of pixels that are horizontally, vertically, or diagonally touching. 4-connected watershed lines consist of pixels that are just horizontally and/or vertically touching. 8-connected watershed lines can separate 4-connected blobs, that is, blobs whose pixels can touch horizontally or vertically. 4-connected watershed lines are required to separate 8-connected blobs, that is, blobs whose pixels can touch horizontally, vertically, or diagonally.



A white blob on a black background.



If the blob is only 4-connected, the above 8-connected watershed line will separate it. However, if the blob is 8-connected, the above watershed line is not sufficient, since the blob is still diagonally touching.



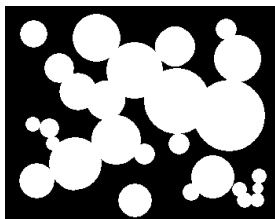
If the blob is 8-connected, the above 4-connected watershed line will separate it.

❖ MIL's blob analysis module allows you to define blobs as either 4- or 8-connected.

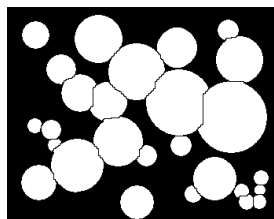
Note that 4-connected watershed lines can also separate 4-connected blobs but result in over-separation.

Exact vs. straight

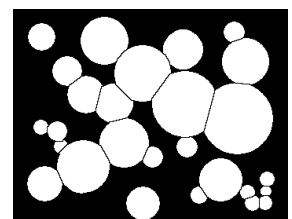
For visual purposes, watershed lines can be traced exactly or forced to be straight.



Original image.



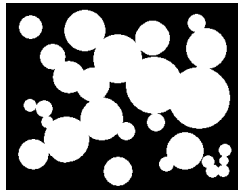
Exactly-traced watershed lines.



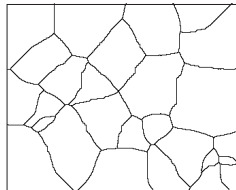
Straight watershed lines.

Skipping the last level

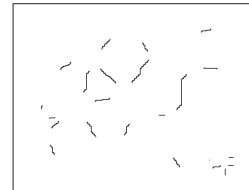
When you perform *MimWatershed()*, you can skip the last intensity level of the transformation (by setting the **ControlFlag** parameter to `M_SKIP_LAST_LEVEL`). In other words, you can prevent an extremum's zone of influence from extending beyond $L_{\max} - 1$ (for a minimum) or $L_{\min} - 1$ (for a maximum), where L_{\max} is the maximum gray-level in the image and L_{\min} is the minimum gray-level. In effect, this prevents the background in the image from being processed, resulting in quicker processing times.



Original image.



A watershed transform when the last level of processing is not skipped.



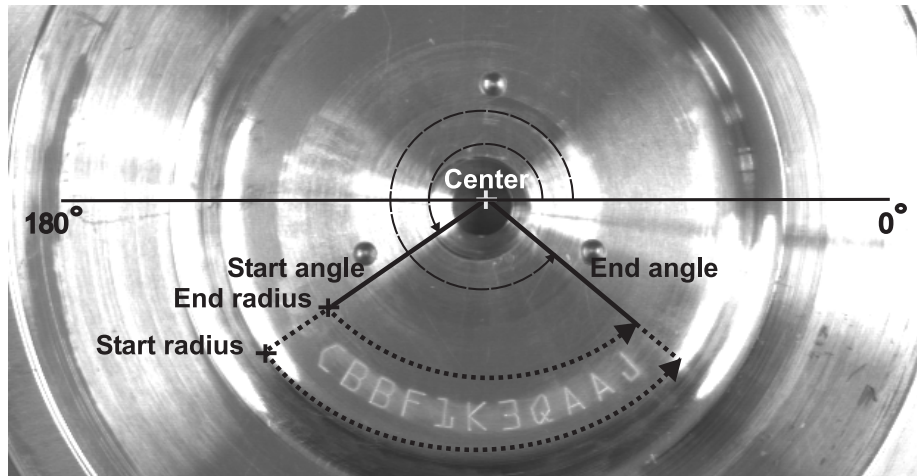
A watershed transform when the last level is skipped. The only watershed lines are those between touching objects.

This option should be used when separating touching objects since, in this case, watershed lines in the background are unnecessary.

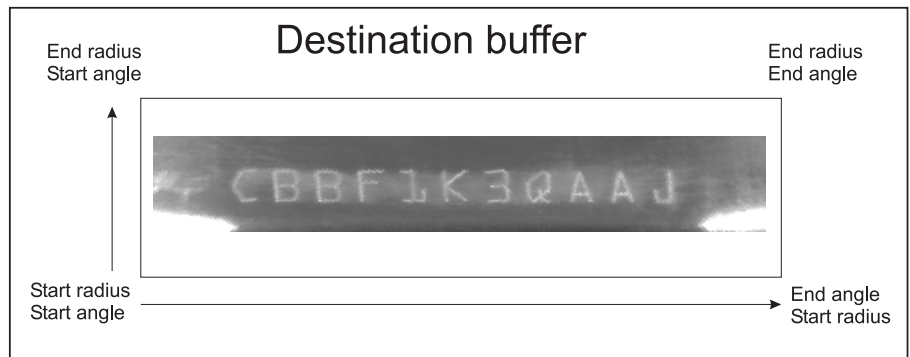
Polar-to-rectangular and rectangular-to-polar transform

Polar-to-rectangular and rectangular-to-polar transform allows conversion of polar coordinates to cartesian coordinates and vice versa. With MIL, you can perform rectangular-to-polar or polar-to-rectangular transforms, using the **MimPolarTransform()** function.

Following is an example of a rectangular-to-polar transform. The dotted line defines the borders of the zone of interest:



The result will be mapped to the destination buffer as shown below:



For a rectangular-to-polar transform, the borders of the zone of interest are defined by specifying the center, the start and end radius, and the start and end angle in a source buffer. The function scans the specified zone from the start angle to the end angle. In our example, since the start angle is less than the end angle, the direction of the scan is counter clockwise. The increment in angle is determined by the length (in pixels) of the outside arc, calculated as follows:

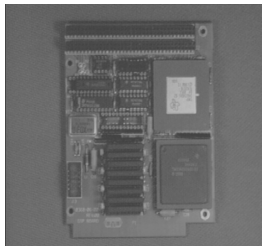
$$\Delta angle = \frac{(endangle - startangle)}{arclength}$$

The valid range of angle is from -360 to 360 degrees and the maximum span of the angle must not exceed 360 degrees. These values are then mapped to a destination buffer.

A polar-to-rectangular transform performs the reverse of the transform described above. It takes a source buffer and maps it to a destination buffer. The center, start angle, end angle, start radius, and end radius parameters are used to specify the position of the contents of the source buffer in the destination buffer.

Warping

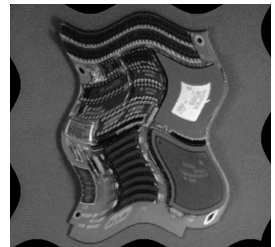
In addition to functions which perform specific geometric transforms (*MimFlip()*, *MimResize()*, *MimRotate()*, *MimTranslate()*, and *MimPolarTransform()*), MIL includes a more general geometric function, *MimWarp()*. It can perform any of the specific transforms, as well as complex warpings. Such warpings could be used, for example, to correct geometric distortions.



Source image



Possible transforms using *MimWarp()*



MimWarp() performs a warping by first associating each pixel position of the destination buffer, (x_d, y_d) , with a specific point (not necessarily a pixel) in the source buffer, (x_s, y_s) . The pixel value at (x_d, y_d) is then determined from an interpolation around its associated source point. Destination pixels can be associated with source points through a first-order polynomial mapping or through look-up tables (LUTs).

Note that the functions which perform specific transforms are faster than *MimWarp()*. You should only use *MimWarp()* when the required transform cannot be otherwise performed.

❖ Geometric distortions can also be resolved using the calibration module. See Chapter 7 for details.

Example

A warping example, *mwarp.c*, can be found in your examples directory.

First-order polynomial warpings

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:

$$x_s = a_0x_d + a_1y_d + a_2$$

$$y_s = b_0x_d + b_1y_d + b_2$$

Generating
coefficients

The coefficients ($a_0...a_2, b_0...b_2$) required to produce a first-order polynomial warping can be automatically generated using *MgenWarpParameter()* or can be user-supplied. When using *MgenWarpParameter()*, 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.

To combine coefficients, you need to use separate calls to *MgenWarpParameter()*. For example, to generate coefficients for a rotation and translation, you need to call *MgenWarpParameter()* twice, using the output buffer of the first call as the input buffer of the second call. After all coefficients are generated, pass the coefficient buffer to *MimWarp()*.

Using LUTs to perform a warping

When you perform a warping using LUTs, x_s is determined from (x_d, y_d) through one LUT and y_s is determined from (x_d, y_d) through another LUT. In other words,

$$x_s = \text{LUT}_x[x_d, y_d]$$

$$y_s = \text{LUT}_y[x_d, y_d]$$

Since x_s and y_s can be arbitrarily mapped, you can perform any type of warping when using LUTs.

The LUTs that you pass to *MimWarp()* can be user-supplied or, for a 3x3 matrix-defined warping, can be automatically generated using *MgenWarpParameter()*.

3x3 matrix-defined warping

A 3x3 matrix-defined warping is performed 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) , according to the following equation:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ c_0 & c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix}$$

where

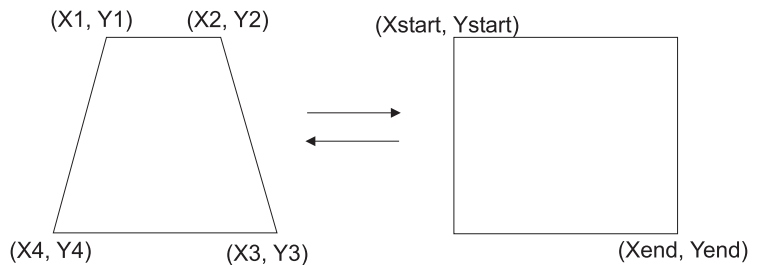
$$x_s = \frac{x}{w} = \frac{a_0 x_d + a_1 y_d + a_2}{c_0 x_d + c_1 y_d + c_2}$$

$$y_s = \frac{y}{w} = \frac{b_0 x_d + b_1 y_d + b_2}{c_0 x_d + c_1 y_d + c_2}$$

To perform a 3x3 matrix-defined warping, supply the 3x3 coefficients $(a_0...a_2, b_0...b_2, c_0...c_2)$ to *MgenWarpParameter()*, which will generate the LUTs required by *MimWarp()*.

Perspective warping

A 3x3 matrix-defined warping can produce perspective transformations that map an arbitrary quadrilateral onto a rectangle or that map a rectangle onto an arbitrary quadrilateral.



To produce such a perspective transformation, specify the coordinates of the above points; *MgenWarpParameter()* will generate the required 3x3 coefficients $(a_0...a_2, b_0...b_2, c_0...c_2)$. You then call *MgenWarpParameter()* again, having it generate

the LUTs from the 3x3 coefficients. Alternatively, if you do not need to save the 3x3 coefficients, you can have the LUTs generated on the first call to *MgenWarpParameter()*.

After the LUTs are generated, pass them to *MimWarp()*.

First-order polynomial warping

If c_0 and c_1 are set to 0 in the equation for a 3x3 matrix-defined warping and c_2 is set to 1, the equation reduces to a first-order polynomial warping. Therefore, you could perform a first-order polynomial warping by having *MgenWarpParameter()* generate the LUTs from the $(a_0...a_2, b_0...b_2, 0\ 0\ 1)$ coefficients, then passing the LUTs to *MimWarp()*. Depending on your system, this might be faster.

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 interpolation modes are available:

- 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.

Points outside the source buffer

Sometimes, the point associated with a destination pixel will fall outside the source buffer. In such cases, the new value for the destination pixel can be determined in one of the following ways:

- You can use pixels from the source buffer's ancestor buffer. If the source buffer is not a child buffer or if the point falls outside the ancestor buffer, the destination pixel will be left as is.
- You can just leave the destination pixel as is.
- You can set the destination pixel to 0.

In general, you should use pixels from the source buffer's ancestor buffer when the source buffer is a child buffer. This will ensure that the pixels you use are related to the source buffer. If the source buffer is not a child buffer, use one of the other options.

Note that you can set the destination pixel to a value other than 0 by first clearing the destination buffer to that value.

Discrete Cosine Transform

Discrete Cosine Transform (DCT) is mainly used for image JPEG lossy compression. MIL can perform DCT using **MimTransform()**. For a one dimensional signal, this method separates the signal into a set of cosine waves of different frequency. For a two-dimensional signal (image), it can be interpreted as the decomposition of an image into a set of 2D cosine patterns. The composition of these waves make up the original waveform.

The forward DCT is defined as:

$$C(u, v) = \frac{K(u)}{2} \frac{K(v)}{2} \sum_{x=0}^7 \sum_{y=0}^7 f(x, y) \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right]$$

where u and v are coordinates in the frequency domain,

$$K(u) = \frac{1}{\sqrt{2}} \text{ for } u = 0 \text{ and } K(u) = 1 \text{ for } u > 0$$

$$\text{and } K(v) = \frac{1}{\sqrt{2}} \text{ for } v = 0 \text{ and } K(v) = 1 \text{ for } v > 0$$

The reverse DCT is defined as:

$$f(x, y) = \sum_{v=0}^7 \frac{K(v)}{2} \sum_{u=0}^7 \frac{K(u)}{2} C(u, v) \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right]$$

where x and y are coordinates in the spatial domain.

Frequency 0, also called the DC component, is plotted in the top-left corner of the spectrum. All other components in the spectrum are called AC components. A DCT concentrates the low frequency components of the image in the first few coefficients (top left-corner) of the spectrum. MIL divides the image into independent blocks of 8x8 pixels and performs the transform on each individual block.

Centering of the spectrum is not supported in MIL.

Chapter 7: Calibration

This chapter describes how to use MIL's calibration module.

Introduction

MIL's calibration module (*Mcal...()*) consists of a set of functions that allow you to map pixel coordinates to real-world coordinates. This mapping can be used to get results from other MIL modules in real-world units. The mapping can also be used to physically correct an image's distortions.

By getting results in real-world units, you automatically compensate for any distortions in an image. Therefore, you can get accurate results despite an image's distortions.

Calibration

Defining the pixel-to-world mapping is known as *calibration*. A *calibration object* is used to hold the defined mapping, as well as certain control settings.

Once you have created your calibration object, you can:

- Use it to transform pixel coordinates or results to their real-world equivalents.
- Use it to physically correct an image.
- Use it to automatically get results from other MIL modules in real-world units. The modules that can return results in real-world units are:
 - *Mblob...()*
 - *Mcode...()*
 - *Mim...()*
 - *Mmeas...()*
 - *Mocr...()*
 - *Mpat...()*
- ❖ Note that a few results are always returned in pixel units. If a result can be returned in either real-world or pixel units, it will be stated in the command description.

Types of distortions

You can use calibration if you have one or more of the following types of distortion:

- **Non-unity aspect ratio distortion:** Present when the X and Y axis have two different scale factors. This is evident, for example, if you know that the object in your image should be round and it appears as an ellipse. This type of distortion is often a side effect of the sampling rate used by some older digitizers.
- **Rotation distortion:** Present when the camera is perpendicular to the object grabbed in the image, but not aligned with the object's axes.
- **Perspective distortion:** Present when the camera is not perpendicular to the object grabbed in the image. Objects that are further away from the camera appear proportionally smaller than the same size objects closer to the camera.
- **Other spatial distortions:** Complex distortions, such as pin cushion and barrel-type distortions, fall in this category. These distortions can be compensated for by using a large number of small sections in the mapping function. If the number of sections used is big enough and the corresponding area covered in each is small enough, the mapping in each area can be approximated with a linear interpolation function.

Steps to getting results in real-world units

To get results in real-world units:

1. Allocate a calibration object, using *McalAlloc()*.
2. Calibrate your imaging setup, using either *McalGrid()* or *McalList()*.
3. Do one of the following:
 - To transform pixel coordinates or results to their real-world equivalents, use *McalTransformCoordinate()* or *McalTransformResult()*.
 - To physically correct an image, use *McalTransformImage()*.
 - To automatically get results from other MIL modules in real-world units, associate the calibration object to an image or digitizer, using *McalAssociate()*.

Transforming coordinates or results

*Transforming
coordinates*

You can use *McalTransformCoordinate()* to convert coordinates from their pixel to their world values (or vice-versa).

Transforming results

You can use *McalTransformResult()* to convert a specific non-positional result (a length, angle, or area) from its pixel to its world value (or vice-versa). Note, however, that this function uses the average pixel size to perform the conversion; results will be more accurate if you first correct the image.

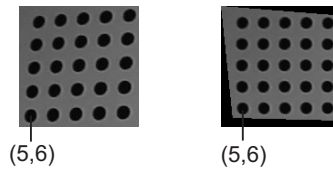
Physically correcting an image

Corrected image

You can use *McalTransformImage()* to physically correct and remove certain types of distortions in an image. An image that has been physically corrected using the calibration module is known as a *corrected image*. When a corrected image is used within a MIL module, results can be returned in real-world units. As is expected, the image features in the destination image have the same world coordinates as they had in the source image, despite the fact their pixel coordinates have changed.

For example, after calibrating the source image below, the world coordinates of the bottom-left circle are (5,6). When you correct your image, the world coordinates of the bottom-left circle in the corrected image are also (5,6), even though it is evident that the pixel coordinates are different for the bottom-left circle in the source and corrected images.

Source image Corrected image



Note that images are physically corrected using a geometric warping.

Accelerating through a cache

By default, a cache is used to physically correct an image. The first time a calibration object is used to transform an image, this cache fills up with information relevant to the transformation. On subsequent transformations with the calibration object, the information in the cache can significantly accelerate the transform. However, if you need to save memory, you can disable this cache, using the `M_TRANSFORM_CACHE` setting of `McalControl()`. (The cache consists of two 32-bit buffers with the same size as the destination buffer of `McalTransformImage()`).

❖ The information in the cache is flushed whenever the size of the source or destination buffers of `McalTransformImage()` changes, or whenever the angle of the relative coordinate system of the calibration object changes. Coordinate systems are discussed later in this chapter.

Automatically getting results in real-world units

Associating

To automatically get results from other MIL modules in real-world units, associate the calibration object to an image or digitizer, using *McalAssociate()*.

Disassociating

To disassociate a calibration object from an image or digitizer, you also use *McalAssociate()*.

Calibrated image

An image with an associated calibration object is known as a *calibrated image*. A calibrated image still appears distorted because it has not been physically corrected. However, when it is used within a MIL module, results from this module can be returned in real-world units.

Note that the calibration object gets associated to the image, not to the buffer containing the image. To understand the implications this has on processing, refer to the *Processing calibrated images* section in this chapter.

Also note that, if you save a calibrated image to file, the calibration settings do not get saved.

Calibrated digitizer

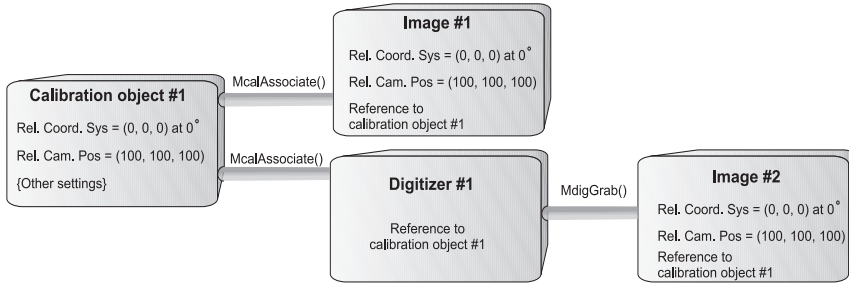
A digitizer with an associated calibration object is known as a *calibrated digitizer*. When you grab an image with a calibrated digitizer, the calibration object currently associated to the digitizer gets associated to the grabbed image. Therefore, the grabbed image becomes a calibrated image.

It is recommended to associate a particular calibration object to only one digitizer and adjust the parameters (if necessary) of each subsequent calibration object so that they are in the same world-coordinate system.

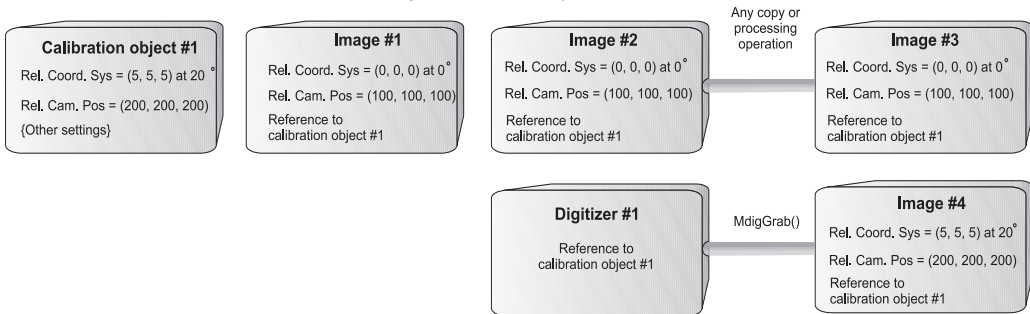
Associating to image vs. digitizer

When you associate a calibration object to an image (either with a call to *McalAssociate()* or a grab with a calibrated digitizer), the image receives a copy of the calibration object's current relative coordinate system and current relative camera position and a reference to the calibration object for all other settings. This means that, if you change the relative coordinate system or relative camera position after association, the change will not affect the image. (The relative coordinate system and relative camera position are discussed later). If you change any

other setting of the calibration object after association, the change will affect the calibrated image. When you associate a calibration object to a digitizer, the digitizer only receives a reference to the calibration object.



State after some changes to calibration object #1:



Child buffers

When a calibration object is associated to an image that contains child images, the child images are automatically calibrated. In addition, their offsets to the parent image are taken into account when returning real-world results.

Returned results

When a calibrated image is used within a MIL module, results can be returned in pixel units or in real-world units. To specify whether results should be in pixel or real-world units, use the `M_OUTPUT_COORDINATE_SYSTEM` setting of `McalControl()`. By default, results are in real-world units. While results can be returned in pixel or real-world units, any value which you pass to a MIL function must be in pixel units.

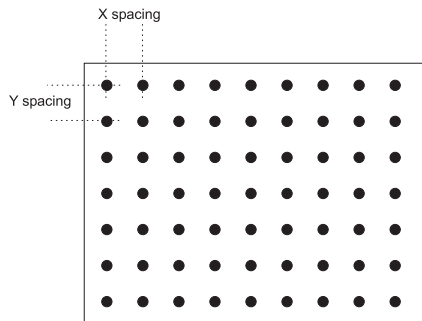
❖ A few results are always returned in pixel units. If a result can be returned in either real-world or pixel units, it will be stated in the command description.

Calibrating your imaging setup

To calibrate your imaging setup, you can use an image of a user-defined grid of circles or you can use a list of pixel and real-world coordinates. To use a grid, call *McalGrid()*. To use a list of coordinates, call *McalList()*.

Real-world grid

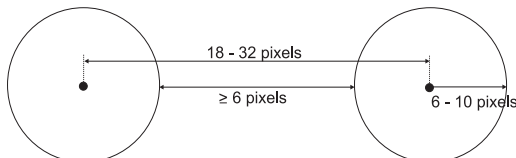
McalGrid() determines the pixel-to-world mapping from an image of a user-defined grid of circles and the world description of this grid. The world description includes the number of rows and columns, as well as the center-to-center distance between these rows and columns, in real-world units.



*General rules for
constructing a grid*

McalGrid() can create a pixel-to-world mapping from almost any grid of circles. However, to create an accurate (sub-pixel) mapping, your physical grid should meet the following guidelines (at the working resolution):

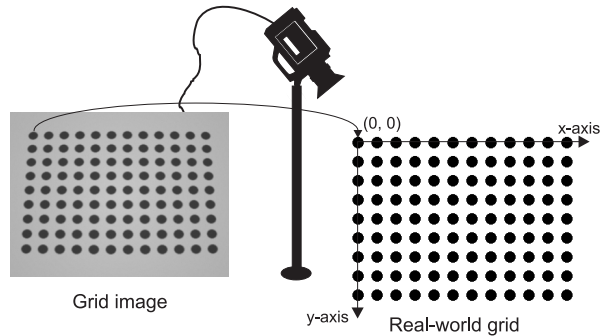
- The radius of the grid's circles should range between 6 and 10 pixels.
- The center-to-center distance between the grid's circles should range from 18 to 32 pixels (22 pixels recommended).
- The minimum distance between the edges of the circles should be 6 pixels.



- The grid should be large enough to cover the area of the image from which you want real-world results (the working area).
- The grid image should have high contrast.

The world

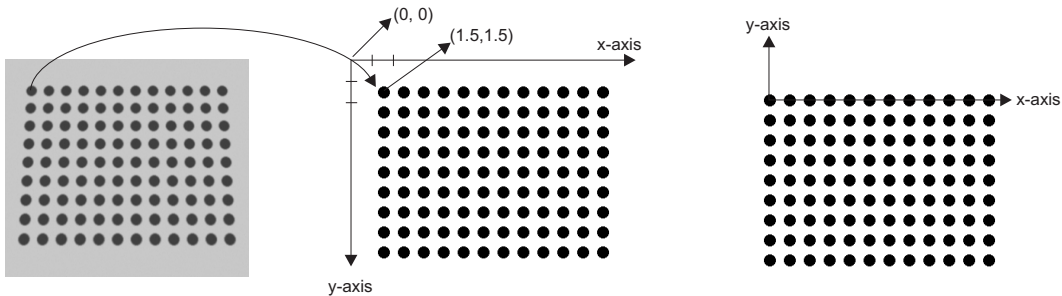
By default, the circle in the top-left corner of the grid image is associated to the origin, $(0, 0)$, of the *real world coordinate system*, the first column of circles is aligned with its Y-axis, and the first row of circles is aligned with its X-axis. This is because, in general, you know where that first circle is in the real-world, so you need results with respect to that position (the top-left pixel, for example, is generally not a known position in the real-world).



Offset and Y-axis

If necessary, you can associate the top-left circle of the grid image to a different position within the real-world coordinate system. The origin of the real-world coordinate system does not have to be within the field-of-view. Note that this offset is specified in real-world units.

You can have the positive Y-axis oriented 90° counter-clockwise with respect to the positive X-axis (by default, the calibration module assumes it is oriented 90° clockwise).



You can associate the top-left circle of the grid image to a different position within the world. In this case, an offset of (1.5,1.5) was used.

You can have the positive y-axis orientated counter-clockwise with respect to the positive x-axis.

Each circle's coordinates

After you call *McalGrid()*, you can inquire about the pixel coordinates and associated real-world coordinates of each circle in the grid using *McalInquire()* with *M_CALIBRATION_IMAGE_POINTS_X/Y* and *M_CALIBRATION_WORLD_POINTS_X/Y*. This will return the coordinates of the center of the grid's circles.

List of coordinates

McalList() uses a list of pixel coordinates and their associated real-world coordinates to define the pixel-to-world mapping. The more coordinates you specify, the more accurate the mapping. *McalList()* can be used when you explicitly know the real-world coordinates for a given set of pixel coordinates. The specified pixel coordinates should cover the area of the image from which you want real-world coordinates (the working area).

In the case of perspective distortion, knowing the world coordinates of 4 points in the image gives sufficient information to create a mapping function. To create a good mapping for a radial distortion requires a larger number of coordinates (for example, more than 30) distributed over the image.

Calibration modes

When you use *McalGrid()* or *McalList()*, you also have to specify the calibration mode. MIL supports the following calibration modes:

- Piecewise linear interpolation.
- Perspective transformation.

Piecewise linear interpolation

In general, you should use the piecewise linear interpolation mode. This mode can compensate for any kind of distortion. It is very accurate for points located inside the working area. However, it is less accurate for points outside the working area. The piecewise linear interpolation mode fits a piecewise linear interpolation function to the set of image coordinates and their real-world equivalents.

Perspective transformation

The perspective transformation mode can compensate for rotation, translation, scale, and perspective distortions. For such distortions, the perspective transformation mode is accurate for points inside and outside the working area. This mode cannot compensate for non-linear distortions such as lens distortions. The perspective transformation mode best fits a global perspective transformation function to the set of image coordinates and their real-world equivalents.

Coordinate systems and camera position

By default, after calibration, real-world positional results will be given within the *absolute world coordinate system*, rather than the *image coordinate system*. You can specify the position of your camera in the absolute world coordinate system. In addition, you can get results relative to some object by moving and/or orienting the *relative world coordinate system*.

Image coordinate system

The image coordinate system is the coordinate system used to locate and/or measure objects in an uncalibrated image. Its unit of measure is pixels. Its origin, (0, 0), is the middle of the image's top-left pixel. Its Y-axis is aligned with the first column of pixels and its X-axis is aligned with the first row of pixels.

Absolute world coordinate system

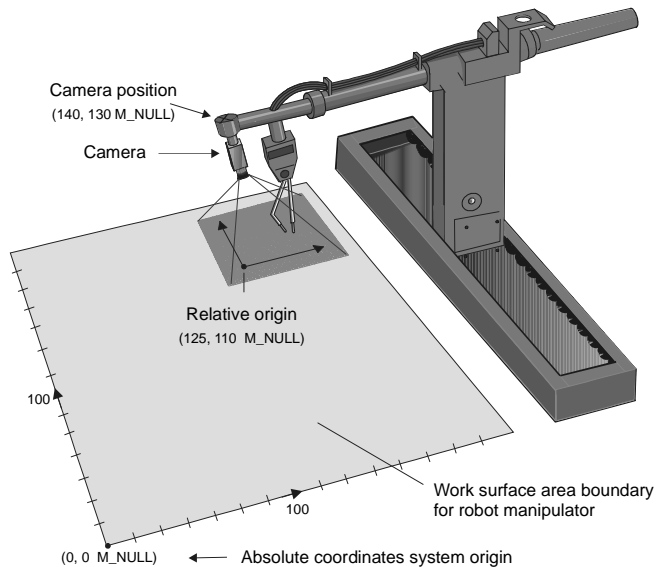
The absolute world coordinate system is the coordinate system used to locate and/or measure objects in the real world. It is implicitly defined from the calibration points when calibrating the imaging setup. Its unit of measure is user-defined (mm, cm, inches, etc.). Calibration relates the image coordinate system to the absolute world coordinate system.

Note that, when using a grid of circles to calibrate your imaging setup, the X-axis of the absolute world coordinate system is aligned with the first row of circles, and the Y-axis is aligned with the first column of circles.

For the sake of simplicity, the absolute world coordinate system will be known as the *absolute coordinate system*.

Camera position

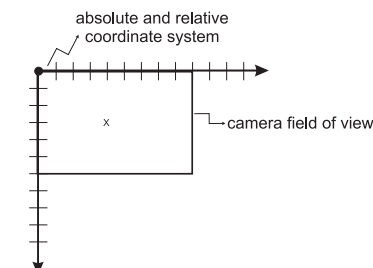
The camera position can be any arbitrary point that moves with the camera; it does not have to be the actual camera position. For example in the following diagram, the camera position is assigned not to the actual camera, but to the arm of the robotic manipulator.



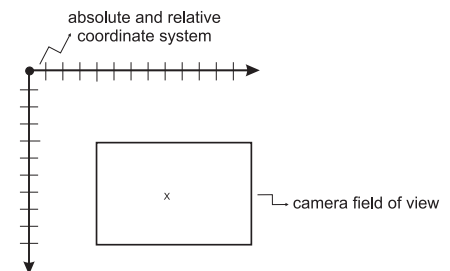
Relative camera position

The relative camera position refers to the position of your camera relative to the absolute coordinate system. Adjusting the relative camera position can be useful when analyzing an object that cannot fit in a single image (see the *Multiple fields of view* section for details).

The relative camera position affects positional results taken from a calibrated image, as shown below.



In this case, $M_CAMERA_POSITION_X$ and $M_CAMERA_POSITION_Y = 0$. The center of the image is mapped to (4, 3).

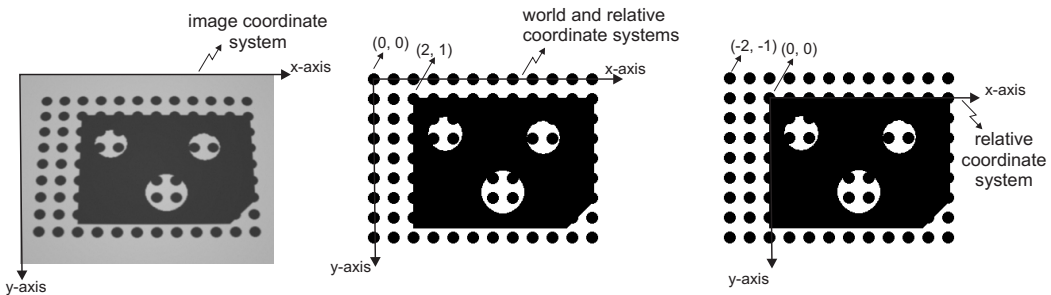


In this case, $M_CAMERA_POSITION_X = 4$ and $M_CAMERA_POSITION_Y = 4$. The center of the image is mapped to (8, 7).

To adjust the relative camera position, use the `M_CAMERA_POSITION_X` and `M_CAMERA_POSITION_Y` controls of `McalControl()`. Note however, that the Z position of the camera cannot be changed when adjusting the relative camera position, and should be set to `M_NULL`.

Relative world coordinate system

By default, the relative world coordinate system is aligned with the absolute coordinate system. However, to get results relative to some object, it can be moved anywhere within the absolute coordinate system and rotated by any angle. Its unit of measure is the same as the absolute coordinate system. For the sake of simplicity, the relative world coordinate system will be known as the *relative coordinate system*.



Note that this image is for illustrative purposes only. In general, the object should not be placed over a grid because if the grid's circles and the object are not differentiated when performing the processing operation, then erroneous results will be returned.

To move and/or rotate the relative coordinate system, use `McalRelativeOrigin()`. Once you change the origin and/or orientation of the relative coordinate system, world coordinates will be returned in this relative coordinate system.

Note that, when you physically correct an image (using `McalTransformImage()`), the image is transformed such that the relative coordinate system is aligned with the image coordinate system.

Multiple fields of view

The calibration module makes it possible to measure the length between various points on an object, even when the object is not entirely within the camera's field-of-view. The camera's field-of-view refers to the largest world region visible in an image at a given resolution.

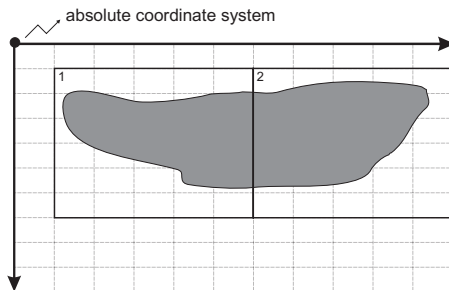
The approach to analyzing a large object involves acquiring images of the various parts of the object, getting real-world coordinates from each image, and then calculating the distance between the coordinates. Three types of applications are considered:

- A single camera is fixed on a manipulator and the manipulator is moved to different positions to acquire the different images.
- A single camera is fixed to a location and the object is moved to different positions to acquire the different images.
- Several cameras are used to acquire the different images.

Single camera fixed on a manipulator: Relative camera position example

When a single camera is fixed on a manipulator, part of the object is grabbed, the camera is moved to a different position, a different part of the object is grabbed, and the process repeats until the entire object has been grabbed. For the coordinates from each image to be in the same absolute coordinate system, the relative camera position has to be updated each time the

camera is moved. To change the relative camera position, use the `M_CAMERA_POSITION_X` and `M_CAMERA_POSITION_Y` controls of `McalControl()`.



1 = first field of view

2 = second field of view

For the first grab:

`M_CAMERA_POSITION_X = 1`

`M_CAMERA_POSITION_Y = 1`

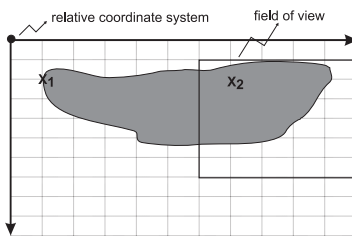
For the second grab:

`M_CAMERA_POSITION_X = 6`

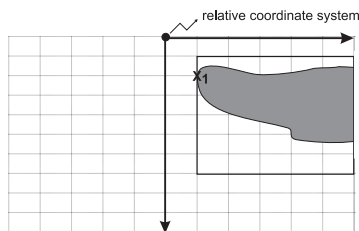
`M_CAMERA_POSITION_Y = 1`

Single camera and moveable object: Relative coordinate system example

When a single camera is fixed to a location, part of the object is grabbed, the object is moved to a different position, another part of the object is grabbed, and the process repeats until the entire object has been grabbed. For the coordinates from each image to be in the same coordinate system, the relative coordinate system must be moved each time the object is moved. To move the relative coordinate system, use `McalRelativeOrigin()`.



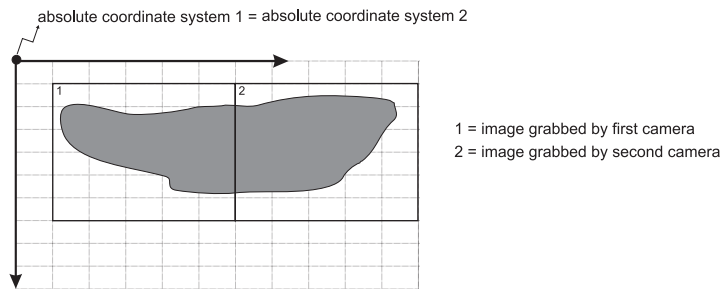
To measure the distance between x_1 and x_2 , the relative coordinate system must be moved along with the object. In this first grab, position x_2 maps to (7, 2).



The object and relative coordinate system are moved 5 world units to the right. Position x_1 maps to (1, 2) so the distance between x_1 and x_2 is 6 world units. Note that, if the relative coordinate system were not moved, x_1 would also have mapped to (1, 2).

Several cameras and fixed object: Relative coordinate system example

For the coordinates from each image to be in the same absolute coordinate system when several cameras are used, the calibration object used to calibrate each camera must use the same absolute coordinate system. When using *McalGrid()*, you must use an offset to relate the origin of each grid to the same absolute coordinate system. When using *McalList()*, the real-world coordinates used in each mapping must be from the same absolute coordinate system.



Processing calibrated images

The calibration object is associated with the image and not the buffer. This has certain implications on processing operations. Depending on the type of operation performed on the calibrated image, the destination image is associated with the following calibration object:

- When performing point-to-point or neighborhood processing operations, the destination image is associated with the same calibration as the source image.

If the operation uses more than one source image, a calibration object gets associated to the destination image only if all source images have the same calibration object; otherwise, no calibration object gets associated to the destination image.

- Geometrical functions (*MimFlip()*, *MimPolarTransform()*, *MimResize()*, *MimRotate()*, *MimTranslate()*, and *MimWarp()*) always result in an uncalibrated image, even if the source image is calibrated.
- The function *MbufClear()* always results in an uncalibrated image.
- Functions for which the source data is always uncalibrated always produce uncalibrated images. These functions include, for example *MbufImport...()* and *MbufLoad()*.

Chapter 8: Blob analysis

This chapter describes the basic steps to extract connected regions of pixels (blobs) within an image.

Blob analysis

Blobs?

Blob analysis allows you to identify connected regions of pixels within an image, then calculate selected features of those regions. The regions are commonly known as **blobs**.

Blobs are areas of touching pixels that are in the same logical pixel state. This pixel state is called the foreground state, while the alternate state is called the background state. Typically, the background has the value zero and the foreground is everything else (although some control is generally provided to reverse the sense).

Feature extraction



In many applications, we are interested only in blobs whose features satisfy certain criteria. Since computation is time-consuming, blob analysis is often performed as an elimination process whereby only blobs of interest are considered in further analysis. The steps involved in feature extraction are:

1. Analyze an image and exclude or delete blobs that don't meet determined criteria.
2. Analyze the remaining blobs to extract further features and determine their criteria.

Repeat these steps, as necessary, until you have all the blob measurement results you need.

Reducing the raw data to just a few feature measurements generally produces more comprehensible and useful results.

MIL and blob analysis

The MIL package includes a blob analysis module that can extract a wide assortment of blob features, such as the blob area, perimeter, Feret diameter at a given angle, minimum bounding box, and compactness.

Identifier image

MIL uses a user-specified blob identifier image to discriminate between blobs and the background. Controls are provided to allow you to specify how this identifier image is interpreted (which pixels are part of which blob). Blobs are considered to consist of either zero or non-zero pixels, depending on the foreground control setting. The non-zero pixels can either have any value or must be set to the maximum value of the buffer (for example, 0xff for an 8-bit image), depending on the identifier type (grayscale or binary). In addition, MIL provides controls to take into account such blob image information as the pixel aspect ratio.

For binary feature extractions, such as those that pertain to the overall shape of the blob, the blob identifier image is used for both identification and computation. For grayscale extractions (e.g. the mean pixel value in a blob), you must also provide a grayscale image whose pixel values will be used in computation.

Supported buffers

With MIL, you can perform blob analysis on 1-bit, 8-bit or 16-bit unsigned buffers.

Steps to performing blob analysis

Although there are a multitude of features that can be calculated and used during the elimination or the analysis process, the following is a series of steps that you will typically perform:

1. Grab or load an image that was captured under the best possible conditions to minimize the amount of preprocessing required.

2. If necessary, reduce the amount of noise in the image. (Noise makes the next step more difficult.)
3. Segment the image so that blobs are separated from the background and from each other. Typically, this involves binarizing the image so that the background is in one state (zero or non-zero) and the blob pixels are in the other state. This image is known as the blob identifier image. If you plan to perform grayscale calculations, you will need the original grayscale image as well.
4. If necessary, preprocess the blob identifier image. If there are too many noise particles, calculation time will be increased. An opening operation (for non-zero blobs) or a closing operation (for zero blobs) will remove most of the noise particles without affecting real blobs significantly. You might also need to separate touching blobs at this stage (or they will be counted as a single blob).
5. Allocate a buffer for blob analysis results, using *MblobAllocResult()*.
6. If necessary, adjust default blob analysis controls to fit your application, using *MblobControl()*. You can control the pixel aspect ratio, when to consider two pixels touching (along horizontal and vertical only or also along the diagonal), which values in the identifier image represent a blob (zero or non-zero), and whether or not non-zero pixels in the identifier image can have any value or must be set to the maximum value of the buffer (grayscale or binary). For example, the maximum value of an 8-bit buffer is 0xff.
7. Allocate a feature list, using *MblobAllocFeatureList()*. This list is used to specify the features that should be calculated. By default, this feature list is empty; no features are selected.

8. Calculate the required features and analyze the results. This involves the following:
 - ❑ Adding the required features into the feature list so that they will be calculated. Typically, you will use *MblobSelectFeature()* to perform this operation. However, when calculating moments or Feret diameters, you might need to use the more general feature selectors, *MblobSelectMoment()* or *MblobSelectFeret()*, respectively.
 - ❑ Calculating results for the selected features, using *MblobCalculate()*. For this command, you will have to specify the blob identifier image that will be used to identify the blobs and calculate binary features, and (optionally) the grayscale image that will be used to calculate grayscale features.
 - ❑ If necessary, excluding or deleting blobs that do not meet the criteria, using *MblobSelect()*. Results for the excluded or deleted blobs will not be returned. Excluded blobs will be ignored in future calculations, while deleted blobs will be removed from the blob analysis result buffer altogether.
 - ❑ Getting the number of blobs currently included, using *MblobGetNumber()*, and retrieving the results from the blob result buffer, using *MblobGetResult()*. Note, *MblobGetResultSingle()* obtains results for a single blob, while *MblobGetLabel()* and *MblobGetRuns()* can be used to obtain more specific results.

You can repeat this step until you obtain all required results for the blobs of interest. Note, the process of excluding or deleting unwanted blobs and then calculating more features is the preferred method if you have many unwanted blobs. If this is not the case, it is often faster to calculate all the required features for all the blobs with a single call to *MblobCalculate()*, and then exclude or delete unwanted blob results afterwards.

A simple blob analysis example

We have provided an example that counts the number of blobs in an image and then marks their centers of gravity. Note, the binarizing step produces a considerable number of spurious blobs and holes, so some processing is performed to clean up the blob identifier image before doing any calculations.

```

/* File name: mblob.c
 * Synopsis: This program loads an image of some nuts, bolts and
 *           washers, determines the number of each of these and marks their
 *           center of gravity. */

#include <stdio.h>
#include <mil.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE           M_IMAGE_PATH"bolts.mim"
#define IMAGE_WIDTH          512L
#define IMAGE_HEIGHT         480L
#define IMAGE_THRESHOLD_VALUE 24L

/* Maximum number of blobs. */
#define MAX_BLOBS            100L

/* Minimum and maximum area of blobs. */
#define MIN_BLOB_AREA        50L
#define MAX_BLOB_AREA        50000L

/* Radius of the smallest particles to keep. */
#define MIN_BLOB_RADIUS      3L

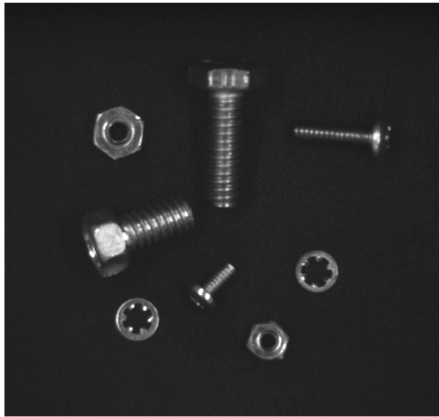
/* Minimum hole compactness corresponding to a washer. */
#define MIN_COMPACTNESS      1.5

/* Size and color of the cross used to mark centers of gravity. */
#define CROSS_SIZE           20L
#define CROSS_COLOR          250L

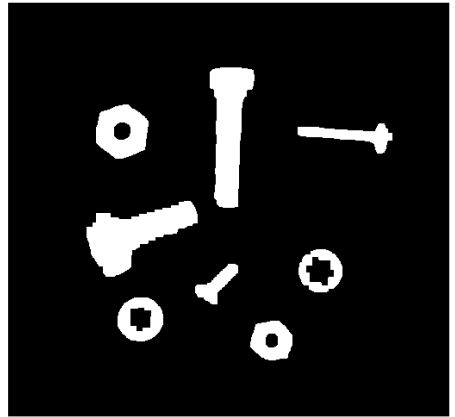
/* Utility functions prototype */
void DrawCross(MIL_ID ImageId, double CenterX, double CenterY, long Color);

void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
          MilSystem,       /* System identifier. */
          MilDisplay,      /* Display identifier. */
          MilImage,        /* Image buffer identifier. */
          BinImage,        /* Binary image buffer identifier. */
          BlobResult,      /* Blob result buffer identifier. */
          FeatureList;     /* Feature list identifier. */
    long TotalBlobs, /* Total number of blobs. */
          ...
          CogX[MAX_BLOBS], /* X coordinate of center of gravity. */
          CogY[MAX_BLOBS], /* Y coordinate of center of gravity. */
          n;               /* Counter. */
    (cont. ...)

```



Original image

Binarized version
after noise has been removed

```

/* Allocate defaults */
MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
M_NULL, &MilImage);

/* Allocate a binary image buffer for fast processing. */
MbufAlloc2d(M_DEFAULT, IMAGE_WIDTH, IMAGE_HEIGHT,
1*M_UNSIGNED, M_IMAGE+M_PROC, &BinImage);

/* Load source image into image buffer. */
MbufLoad(IMAGE_FILE, MilImage);

/* Pause to show the original image. */
printf("This program determines the number of objects in the\n");
printf("displayed image and marks the center of gravity of each.\n");
printf("Press <Enter> to continue.\n");
getchar();

/* Binarize image. */
MimBinarize(MilImage, BinImage, M_GREATER_OR_EQUAL,
IMAGE_THRESHOLD_VALUE, M_NULL);

/* Remove small particles and then remove small holes. */
MimOpen(BinImage, BinImage, SMALL_PARTICLE_RADIUS, M_BINARY);
MimClose(BinImage, BinImage, SMALL_PARTICLE_RADIUS, M_BINARY);

/* Allocate a feature list. */
MblobAllocFeatureList(MilSystem, &FeatureList);

(cont. ...)

```

```

/* Enable the area feature to select blobs of interest
 * and the COG feature to mark their center of gravity.
 */
MblobSelectFeature(FeatureList, M_AREA);
MblobSelectFeature(FeatureList, M_CENTER_OF_GRAVITY);

/* Allocate a blob result buffer. */
MblobAllocResult(MilSystem, &BlobResult);

/* Calculate selected features for each blob. */
MblobCalculate(BinImage, M_NULL, FeatureList, BlobResult);

/* Exclude blobs whose area is too small. */
MblobSelect(BlobResult, M_EXCLUDE, M_AREA, M_LESS_OR_EQUAL,
            MIN_BLOB_AREA, M_NULL);

/* Get the total number of selected blobs. */
MblobGetNumber(BlobResult, &TotalBlobs);
printf("\nThere are %ld objects in the image.\n", TotalBlobs);

/* Check for array overflow. */
if(TotalBlobs > MAX_BLOBS)
{
    printf("Error: too many blobs.\n");
}
else
{
    /* Get the results. */
    MblobGetResult(BlobResult, M_CENTER_OF_GRAVITY_X+M_TYPE_LONG, CogX);
    MblobGetResult(BlobResult, M_CENTER_OF_GRAVITY_Y+M_TYPE_LONG, CogY);

    /* Draw gray cross at the center of gravity of each blob. */
    for(n=0; n < TotalBlobs; n++)
    {
        DrawCross(MilImage, CogX[n], CogY[n], CROSS_COLOR);
    }

    printf("and their centers of gravity have been marked.\n\n");
}
...

/* Print results. */
printf("\nThere are: %ld bolts\n", TotalBlobs-BlobsWithHoles);
printf("          %ld nuts\n", BlobsWithHoles - BlobsWithRoughHoles);
printf("          %ld washers\n\n", BlobsWithRoughHoles);
printf("Press <Enter> to end.\n");
getchar();

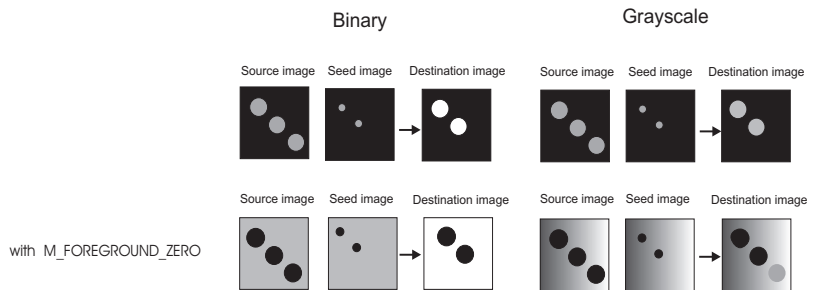
/* Free all allocations. */
MblobFree(BlobResult);
MblobFree(FeatureList);
MbufFree(BinImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

Blob reconstruction

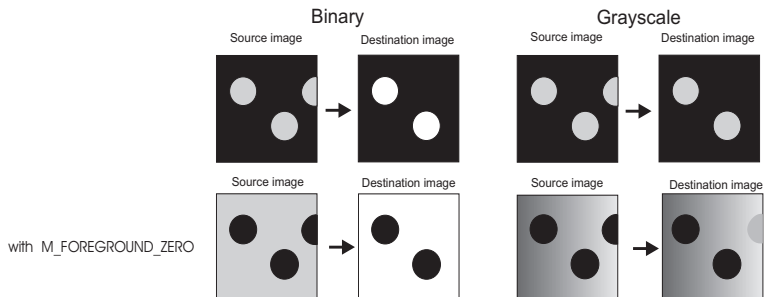
Although the blob analysis module is used mainly for blob feature calculation purposes, some of the *Mblob...()* commands can be used to perform blob image reconstruction. An example of this is the *MblobReconstruct()* command. It can:

- Reconstruct blobs from a seed image (that is, copy in the destination buffer only those blobs that have a corresponding seed in the seed buffer):



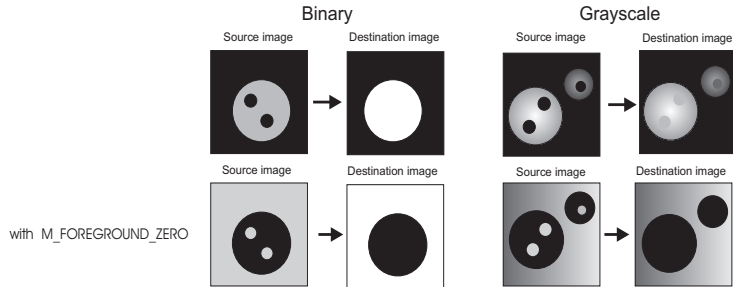
Note that, for images with grayscale backgrounds, blobs in the source image which are not seeded are filled with the average grayscale value of the background.

- Delete blobs that touch a border of the image:



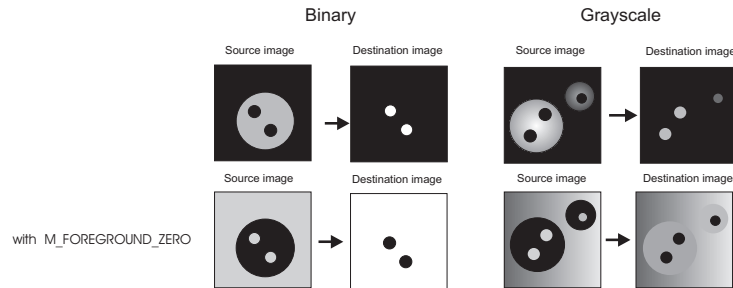
Note that, for images with grayscale backgrounds, the border blobs are filled with the average grayscale value of the background.

■ Fill holes in blobs:



Note that, for images with grayscale blobs, the holes are filled with the average grayscale value of the blob.

■ Extract holes from blobs:



Note that, for images with grayscale backgrounds, the blobs are filled with the average grayscale value of the background.

Finally, the analysis and selection tools (for example, *MblobSelect()* in conjunction with *MblobLabel()* and *MblobFill()*) can also perform other types of image reconstruction.

Chapter 9: Setting up for blob analysis

This chapter describes how to set up for blob analysis. It discusses setting the controls for the blob identifier image, and excluding blobs from calculations.

Identifying blobs

The MIL blob analysis capabilities allow you to identify and extract features of connected regions of pixels (commonly known as blobs) within an image. MIL requires a user-specified blob identifier image in order to determine which pixels belong to which blob in the original image. Blob features involving overall shape are extracted directly from the identifier image. Features that use the actual pixel values of the blob also require the original image.

The MIL blob analysis module considers touching foreground pixels in the blob identifier image to be part of the same blob. Consequently, what is easily identifiable by the human eye as several distinct but touching blobs is interpreted by MIL as a single blob. In addition, any part of a blob that is in the background pixel state, because of lighting or reflection, is considered as background during analysis.

To reduce preprocessing, the blob identifier image should be acquired under the best possible circumstances. This means ensuring that blobs do not overlap and, if possible, don't touch. It also means ensuring the best possible lighting and using a background with a gray level that is very distinct from the gray level of the blobs. If noise is a problem, you might also need to filter the image after acquisition (for example, using a median filter or a convolution with `M_SMOOTH`).

Segmenting the blob image

Once the best possible image is acquired and most noise is filtered out, you must separate the different blobs from the background. Segmentation can be done in two ways:

- Binarize the image, using *MimBinarize()* so that background pixels are represented as zero values and blob pixels are represented as another value.

- Clip all background pixels to zero, while retaining the original values of blob pixels, using *MimClip()*. This method has the advantage of not needing a separate buffer to hold the binary image, but you will not see the result of the segmentation as clearly. The first method is usually better.

If simple segmentation is not possible due to poor lighting or blobs with the same gray level as parts of the background, you must develop a segmentation algorithm appropriate to your particular image.

Preprocessing

Producing the blob identifier image frequently creates some spurious blobs or holes (for example, due to noise or lighting). Such noise blobs make it harder to interpret blob analysis results. If you have many noise blobs, you should probably preprocess the image before using it as an identifier. An opening operation (for non-zero valued blobs or holes) or a closing operation (for zero valued blobs or holes) will remove most noise without significantly affecting real features.

If blobs are touching, you might try eroding the image a few times to break them apart.

Note, preprocessing the blob identifier image might affect the accuracy of calculations because of the slight change in blob shape. If this is a problem, perform the calculations on all the blobs, including those that are actually introduced by noise, then use the results to filter out the noise. Note, however, that this method increases the memory required and might increase the calculation time.

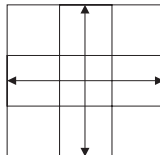
Adjusting blob analysis processing controls

Before performing any blob analysis calculations, you should ensure the correct interpretation of the blob identifier image. Use *MblobControl()* to control how certain aspects of the blob identifier image are interpreted, for example:

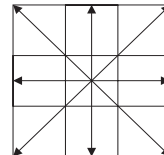
- Which pixel values are considered to be in the foreground (M_FOREGROUND_VALUE).
- Whether two pixels touching at their corners are considered part of the same blob, by appropriately defining the image lattice (M_LATTICE).
- Whether non-zero pixels can have any value or must be set to the maximum value of the buffer; for example, 0xff for an 8-bit buffer (M_IDENTIFIER_TYPE).
- The pixel aspect ratio of the image (M_PIXEL_ASPECT_RATIO).
- Whether to produce separate results for each blob or for groups of blobs (M_BLOB_IDENTIFICATION).
- How many Feret angles are considered when calculating a Feret feature (M_NUMBER_OF_FERETS). Typically, the default value will be appropriate.

Controlling the image lattice

MIL represents images using a square lattice and considers adjacent pixels along the vertical or horizontal axis as touching. However, you can control whether two diagonally adjacent pixels are considered touching.

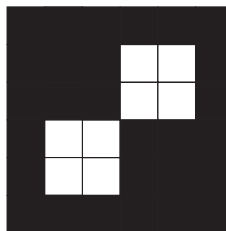


Lattice with 4 neighbor connections
(M_4_CONNECTED)



Lattice with 8 neighbor connections
(M_8_CONNECTED)

Use *MblobControl()* to specify how the blob identifier image lattice should be interpreted. For example, the following is considered one blob if the lattice is set to M_8_CONNECTED, but two blobs if set to M_4_CONNECTED.



The pixel aspect ratio

*Pixel's relation to
real distance*

When acquiring an image of a scene, each pixel represents some real distance both in width and in height. Ideally, this distance is the same in both directions, producing square pixels and allowing for simple feature calculations. However, after digitization, it is quite common for a pixel to represent a different distance in each direction. The ratio of the pixel's width to its height is called the **pixel aspect ratio**. For example, a pixel of equal width and height has a pixel aspect ratio of 1.0.

Note that if you have a calibrated image, feature results are returned in calibrated units.

*Adjusting the aspect
ratio*

In blob analysis, the pixel aspect ratio directly affects feature extractions. For example, all circular blobs are stretched or squashed if the pixels are not exactly square. In this case, you have two alternatives:

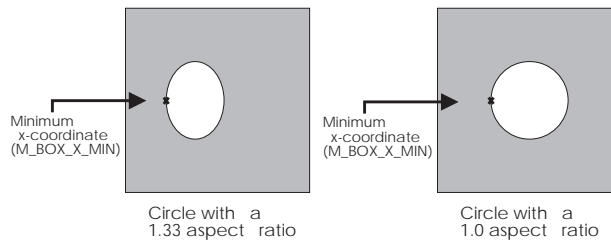
- You can adjust your image, using *MimResize()*, and then make the required blob analysis feature extractions.
- You can have calculations take the actual aspect ratio into consideration without modifying the image, by specifying the ratio, using *MblobControl()*. However, any feature derived from multiple Feret diameters cannot take the pixel aspect ratio into account accurately, and you will get better results by actually resizing your image. The same is true of the general Feret diameter.

In both cases, the actual aspect ratio can be calculated using a simple procedure. Grab an image of a true circle or square and extract the `M_FERET_X` and `M_FERET_Y` features with the default pixel aspect ratio of 1.0. The relationship between these features represents the actual pixel aspect ratio to be used in calculations (M_FERET_Y / M_FERET_X).

Note, if your image has other types of distortions, you can use *MimWarp()* to adjust the image.

*Positions and
the pixel aspect ratio*

Note, all results are affected by the pixel aspect ratio, including those that are just positions within the image. For example, to mark `M_BOX_X_MIN` on an image with a graphics command, you must take the aspect ratio into account (in this case by dividing the returned result by the aspect ratio).



Setting the Blob identification mode

Using *MblobControl()*, you can control how blobs in the blob identifier image are treated during calculations. This depends on the blob identification mode setting:

- Individually (`M_INDIVIDUAL`)
- All blobs grouped together (`M_WHOLE_IMAGE`)
- Different blobs with the same label grouped together (`M_LABELED`)

Note, this mode does not change how blobs are identified (regions of connected foreground pixels); rather, it determines whether results are combined into groups.

Results for each blob

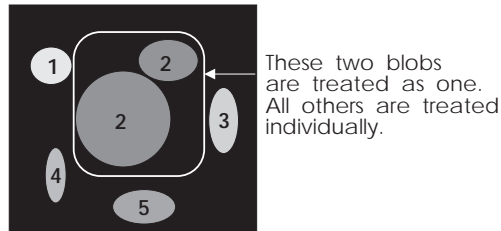
When using the blob analysis package, you usually want to make feature calculations on each blob. For example, if you want to find the area of each cell in a tissue image, set the blob identification mode to `M_INDIVIDUAL`.

*Results for
blobs grouped as one*

Sometimes, however, you need calculations based on the entire image rather than individual blobs. For example, you might want to calculate the area of all the copper in a rock sample image. MIL simplifies your task by allowing you to treat all foreground pixels together by setting the blob identification mode to `M_WHOLE_IMAGE`. Blobs in an image are treated as one blob and features are calculated for this grouped blob.

*Results for blobs
grouped by label value*

Blob identification mode `M_LABELED` allows you to do joint calculations on blobs with the same label value. When using labeled mode, ensure that each blob in the identifier image has a uniform pixel value. This value is the `M_LABEL_VALUE` result for that blob, and determines the grouping of the blobs.



Selecting blobs

Once all blobs are clearly identifiable by the blob analysis package, you are ready to perform calculations. However, in some cases, you will not want to make time-consuming feature extractions for every single blob in the blob identifier image. For example, you probably do not want to calculate features for blobs that are touching the edges of the image or that are noise artifacts. Often, you cannot preprocess these blobs out of your image without losing too much information.

Selecting blobs

The MIL blob analysis package has a command, *MblobSelect()*, for such cases. This allows you to select (on the basis of calculations already made) a subset of blobs for which to make further calculations and get results. This command is generally used in one of two ways:

- If you don't have too many unwanted blobs, it is usually faster to calculate all required features for all blobs. Then, prior to getting results, use *MblobSelect()* to exclude or delete results obtained for blobs that do not meet your criteria.
- If you have many unwanted blobs, you might save time and memory by first calculating, for all blobs, only those features that allow you to distinguish between relevant and unwanted blobs. Exclude from future calculations (or delete altogether from the blob analysis result buffer) blobs that do not meet your criteria, using *MblobSelect()*. Then, calculate all required features for remaining blobs.

If you cannot exclude or delete many blobs using the second method, use the first.

You can make as many calls as necessary to *MblobSelect()* and *MblobCalculate()* in order to arrive at the right set of results. However, you must always give the same identifier and grayscale buffers to *MblobCalculate()* during this procedure. If you give different buffers or change the existing buffers in any way (for example, if you use *MblobFill()* to erase blobs from the identifier image), all current results in the result buffer will be discarded the next time you call *MblobCalculate()*. In addition, all selected features will be re-calculated for all blobs in the new identifier image. This means that you will have to restart the selection procedure. If you intend to calculate grayscale features during your analysis, you must include the grayscale image before starting your calculations.

Chapter 10: Analyzing the blobs

This chapter discusses some of the more commonly used features available for extraction with the MIL blob analysis module. It also discusses some basic concepts of these features.

Making feature extractions

Calculating features

The MIL blob analysis module can calculate a variety of different blob measurements or features, such as the area, perimeter, Feret diameter, and center of gravity of each selected blob. Although the *MblobCalculate()* command initiates the actual calculations, it is the specified feature list that determines which calculations will be performed.

When you first allocate the feature list with *MblobAllocFeatureList()*, no features are selected for calculation. You generally use the *MblobSelectFeature()* command to add features to this feature list. You can, however, use the more specialized *MblobSelectMoment()* or *MblobSelectFeret()* commands to select a specific moment or Feret diameter, respectively.

Binary and grayscale features

The blob analysis module supports both binary and grayscale features. When selecting a binary feature, all calculations are performed using only the blob identifier image. When grayscale features are selected, you must also provide the *MblobCalculate()* command with a grayscale image. The blob identifier image will identify the blobs, and the grayscale image will supply the actual blob pixel values.

Selecting features

When you call *MblobCalculate()*, the identifier image is scanned to locate blobs, and any selected features are calculated. Even if only a few features are selected, the overhead of scanning the image can be considerable. Therefore, it is usually more efficient to select many features and make one call to *MblobCalculate()*, rather than to select and calculate one feature at a time. Note, features that have already been calculated for the specified images will not be recalculated if you call *MblobCalculate()* again, unless any parameters of the calculation have changed.

Which features to calculate

There are several considerations when selecting features:

- Before selecting a feature for calculation, you should take the blob shapes into consideration. Some features are more appropriate for certain blob shapes than for others. For example, some should be used for round blobs rather than long, thin ones, and vice versa. The *MblobSelectFeature()* command provides this information.
- When trying to distinguish between two similar blobs, selection of certain features, rather than some other features that might also seem appropriate, might reveal a more notable difference.
- If two features allow you to come to the same conclusion, it is recommended that you select the one that is calculated more quickly. For example, features derived from multiple Feret diameters tend to calculate relatively slowly, and grayscale calculations are considerably longer than binary ones.

Note, for a visual representation of blobs that meet (or don't meet) certain criteria, call *MblobFill()* or *MblobLabel()* after calculating some features and calling *MblobSelect()*. These commands fill blobs with their own label values (*MblobLabel()*) or with a user specified value (*MblobFill()*).

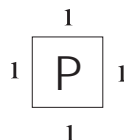
Sorting results

The results obtained from *MblobGetResult()* can be sorted in ascending or descending order, by a maximum of three features assigned as sorting keys. To specify a feature as a sorting key, you can add *M_SORT#_UP* or *M_SORT#_DOWN* to the features selected with the following blob functions: *MblobSelectFeature()*, *MblobSelectMoment()*, *MblobSelectFeret()*. Assign the numbers 1, 2, or 3 to the # to indicate the sorting precedence of the feature(s).

The area and perimeter

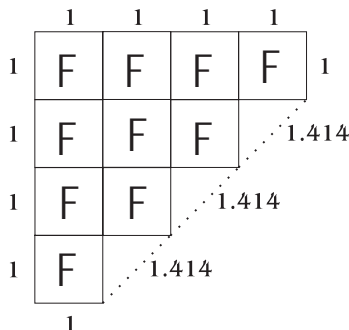
The pixel

Each pixel in your image represents a real width and height (for example, in millimeters). However, all results from the blob analysis commands (that represent a distance or area) are expressed in raw (uncalibrated) pixel units. You are left with the task of converting these results to actual physical units. This task is made easier if the width and height of the pixels are the same (that is, the pixel aspect ratio (width/height) equals 1.0). In this case, each pixel (P) is represented as follows:



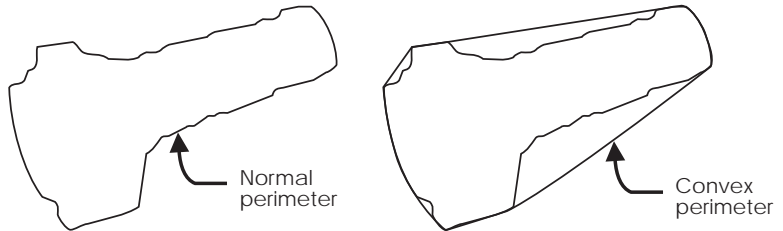
The area and perimeter

A pixel ratio of 1.0 implies that the area (M_AREA) of a single pixel blob is equal to 1 and the perimeter (M_PERIMETER) is equal to 4. When calculating the area and perimeter of a larger blob, the area would then equal the number of pixels in the blob (excluding holes), and the perimeter would equal the total number of pixel sides along the blob edges (including the edges of holes). Note, an allowance is made for the staircase effect that occurs in a digital image when representing diagonals and curves. For example, in the following blob (where F represents foreground pixels), the area is 10 and the perimeter is 14.242.



The convex Perimeter

You can also calculate an approximation of the convex perimeter (M_CONVEX_PERIMETER) of the blobs. The convex perimeter is the perimeter of the convex hull (see below).



This feature is derived by taking the diameter of the blob (Feret diameter) at different angles. You can adjust the number of Feret diameters used with the *MblobControl()* command. The greater the number of Feret diameters used, the more accurate the approximation.

The aspect ratio

When the pixel aspect ratio has been set to anything other than 1.0, using *MblobControl()*, the aspect ratio is applied to the pixel width during calculations. Each pixel is now represented as:

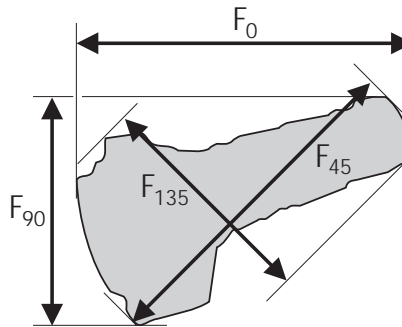
$$\begin{array}{c} \text{asp. ratio} \\ 1 \left[\begin{array}{c} \text{P} \\ \text{P} \end{array} \right] 1 \\ \text{asp. ratio} \end{array}$$

This affects all calculated features as if you had actually stretched the image (from the top-left corner in the x-direction only), by a factor equal to the pixel aspect ratio. We can no longer say that results are in "pixel" units. In fact, results are really in units of "pixel height" since the height is not affected by the aspect ratio.

Dimensions

The Feret diameter

Besides the area and perimeter, you might need to determine the dimension of the blobs. Since blobs are not typically rectangular in shape, you will probably have to take the length (or diameter) of the blobs at various angles from the horizontal axis. This is actually one of the many definitions of the blob length, called the Feret diameter. Several Feret diameters are illustrated below. Note, the angle at which the Feret diameter is taken (relative to the horizontal axis) is specified as a subscript to the F.



Calculating different Feret diameters

With MIL, you can calculate the Feret diameter at a specified angle (`M_GENERAL_FERET`) by adding it to the feature list, using `MblobSelectFeret()`. To add the Feret diameter at 0° (horizontal Feret diameter) and 90° (vertical Feret diameter) to the feature list, you can also use `MblobSelectFeature()` (`M_FERET_X` and `M_FERET_Y`, respectively).

You can automatically determine the minimum, maximum, and average Feret diameters of the blob by adding the `M_FERET_MIN_DIAMETER`, `M_FERET_MAX_DIAMETER`, and `M_FERET_MEAN_DIAMETER` features, respectively, to the feature list, using `MblobSelectFeature()`. These diameters will be determined by testing the diameter of the blobs at several angles. Increasing the number of angles that are tested increases the accuracy of the results, but also increases processing time.

You can use the *MblobControl()* command to change the default number of angles (M_NUMBER_OF_FERETS value); these angles will start at 0° and increase in increments of $180^\circ/(\text{number of Feret diameters})$.

Note, the maximum Feret diameter is not very sensitive to the number of angles; using 8 angles usually produces an accurate result. The minimum diameter, however, can be inaccurate for long thin blobs unless many angles are used.

The angles at which the minimum and maximum Feret diameter were found can be determined by adding the M_FERET_MIN_ANGLE and M_FERET_MAX_ANGLE to the feature list, using *MblobSelectFeature()*.

You can determine the ratio of the maximum to minimum Feret diameter by adding M_FERET_ELONGATION feature to the feature list, using the above command.

Dimensions of long thin blobs

Although the Feret diameters provide a good approximation of the blob size, these features are not very good for long, thin blobs, even when using the maximum number of angles (M_MAX_FERETS). For these, the following features, available with *MblobSelectFeature()*, might provide better results:

- M_LENGTH: an extraction of the true length of a blob.
- M_BREADTH: an extraction of the true breadth of a blob.
- M_ELONGATION: the ratio of the length to the breadth.

These features are derived from the area and perimeter, using the assumption that the blob area is equal to the [length x breadth] and the perimeter is equal to $2(\text{length} + \text{breadth})$. These relations only hold if the length and breadth are constant throughout a blob. However, long, thin blobs generally satisfy this assumption, even if they are not straight.

Note, since these features use only the area and perimeter, they are faster to calculate than Feret features.

Determining the shape

Other useful features during classification are those that give you information about the blob shape. Two blobs can have similar sizes but different shapes because of a different number of holes, curves, or edges.



For example, in the illustration above, the blobs have similar sizes, but can be distinguished by the shape of their holes. If you treat the holes as the actual blobs (set non-zero pixels as foreground pixels and zero pixels as background pixels), you can extract the differences in shape of the holes.

Compactness and roughness

Two features that can qualify the shape of these holes are:

- Compactness (M_COMPACTNESS)
- Roughness (M_ROUGHNESS)

The compactness is a measure of how close all particles in the blob are from one another. It is derived from the perimeter and area. A circular blob is most compact and is defined to have a compactness measure of 1.0 (the minimum); more convoluted shapes have larger values.

The roughness is a measure of the unevenness or irregularity of a blob's surface. It is a ratio of the perimeter to the convex perimeter of a blob. Smooth convex blobs have a roughness of 1.0, whereas rough blobs have a higher value because their true perimeter is bigger than their convex perimeter.

Although either of these features can be used in the classification process, compactness is faster to calculate since it is derived using only the area and perimeter.

For example

In the example below, we calculate the number of bolts, nuts, and washers in an image (diagram found in *Chapter 8*), and distinguish between the nuts and washers by analyzing the compactness of their holes.

```

/* File name: mblob.c
 * Synopsis: This program loads an image of some nuts, bolts and
 *           washers, determines the number of each of these and marks
 *           their center of gravity. */

#include <stdio.h>
#include <mil.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE           M IMAGE_PATH bolts.mim"
#define IMAGE_WIDTH          512L
#define IMAGE_HEIGHT         480L
#define IMAGE_THRESHOLD_VALUE 24L

/* Maximum number of blobs. */
#define MAX_BLOBS            100L

/* Minimum and maximum area of blobs. */
#define MIN_BLOB_AREA        50L
#define MAX_BLOB_AREA        50000L

/* Radius of the smallest particles to keep. */
#define MIN_BLOB_RADIUS      3L

/* Minimum hole compactness corresponding to a washer. */
#define MIN_COMPACTNESS      1.5

/* Size and color of the cross used to mark centers of gravity. */
#define CROSS_SIZE            20L
#define CROSS_COLOR           250L

/* Utility functions prototype */
void DrawCross(MIL_ID ImageId, double CenterX, double CenterY, long Color);

void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem,             /* System identifier. */
    MilDisplay,            /* Display identifier. */
    MilImage,              /* Image buffer identifier. */
    BinImage,              /* Binary image buffer identifier. */
    BlobResult,            /* Blob result buffer identifier. */
    FeatureList;           /* Feature list identifier. */
    long TotalBlobs,       /* Total number of blobs. */
    BlobsWithHoles,        /* Number of blobs with holes. */
    BlobsWithRoughHoles;   /* Number of blobs with rough holes. */
    CogX[MAX_BLOBS],       /* X coordinate of center of gravity. */
    CogY[MAX_BLOBS],       /* Y coordinate of center of gravity. */
    n;                     /* Counter. */
}

```

(cont. ...)

```

/* Allocate defaults */
MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                 M_NULL, &MilImage);
/* Allocate a binary image buffer for fast processing. */
MbufAlloc2d(M_DEFAULT, IMAGE_WIDTH, IMAGE_HEIGHT,
            1*M_UNSIGNED, M_IMAGE+M_PROC, &BinImage);

/* Load blob image into image buffer. */
MbufLoad(IMAGE_FILE, MilImage);

/* Pause to show the original image. */
printf("This program determines the number of nuts\n");
printf("washers, and bolts in the displayed image.\n");
printf("Press <Enter> to continue.\n");
getchar();

/* Binarize the image. */
MimBinarize(MilImage, BinImage, M_GREATER_OR_EQUAL,
            IMAGE_THRESHOLD_VALUE, M_NULL);

/* Remove small particles and then remove small holes. */
MimOpen(BinImage, BinImage, MIN_BLOB_RADIUS, M_BINARY);
MimClose(BinImage, BinImage, MIN_BLOB_RADIUS, M_BINARY);

/* Allocate a feature list. */
MblobAllocFeatureList(MilSystem, &FeatureList);

/* Enable the area feature to select blobs of interest
 * and the COG feature to mark their center of gravity. */
MblobSelectFeature(FeatureList, M_AREA);
MblobSelectFeature(FeatureList, M_CENTER_OF_GRAVITY);

/* Allocate a buffer for the results. */
MblobAllocResult(MilSystem, &BlobResult);

/* Calculate selected feature for each blob. */
MblobCalculate(BinImage, M_NULL, FeatureList, BlobResult);

/* Exclude blobs whose area is too small. */
MblobSelect(BlobResult, M_EXCLUDE, M_AREA, M_LESS_OR_EQUAL,
            MIN_BLOB_AREA, M_NULL);

/* Get the total number of selected blobs. */
MblobGetNumber(BlobResult, &TotalBlobs);
printf("\nThere are %ld objects in the image.\n", TotalBlobs);

/* Check for array overflow. */
if(TotalBlobs > MAX_BLOBS)
{
    printf("Error: too many blobs.\n");
}
else
{
    /* Get the results. */
    MblobGetResult(BlobResult, M_CENTER_OF_GRAVITY_X+M_TYPE_LONG, CogX);
    MblobGetResult(BlobResult, M_CENTER_OF_GRAVITY_Y+M_TYPE_LONG, CogY);

    (cont. ...)

```



```

    /* Draw gray cross at the center of gravity of each blob. */
    for(n=0; n < TotalBlobs; n++)
    {
        DrawCross(MilImage, CogX[n], CogY[n], CROSS_COLOR);
    }
    printf("and their centers of gravity have been marked.\n\n");
}

/* Reverse what is considered to be the background so that
 * holes are seen as being blobs. */
MblobControl(BlobResult, M_FOREGROUND_VALUE, M_ZERO);

/* Add a feature to distinguish between types of holes. Since area
 * has already been added to the feature list, and the processing
 * mode has been changed, all blobs will be re-included and the area
 * of holes will be calculated automatically. */
MblobSelectFeature(FeatureList, M_COMPACTNESS);

/* Calculate selected features for each blob. */
MblobCalculate(BinImage, M_NULL, FeatureList, BlobResult);

/* Exclude small holes and large (the area around objects) holes. */
MblobSelect(BlobResult, M_EXCLUDE, M_AREA, M_OUT_RANGE,
            MIN_BLOB_AREA, MAX_BLOB_AREA);

/* Get the number of blobs with holes. */
MblobGetNumber(BlobResult, &BlobsWithHoles);

/* Exclude blobs whose holes are compact (i.e. nuts). */
MblobSelect(BlobResult, M_EXCLUDE, M_COMPACTNESS,
            M_LESS_OR_EQUAL, MIN_COMPACTNESS, M_NULL);

/* Get the number of blobs with holes which are NOT compact. */
MblobGetNumber(BlobResult, &BlobsWithRoughHoles);

/* Print results. */
printf("\nThere are: %ld bolts\n", TotalBlobs-BlobsWithHoles);
printf("           %ld nuts\n", BlobsWithHoles - BlobsWithRoughHoles);
printf("           %ld washers\n", BlobsWithRoughHoles);
printf("Press <Enter> to end.\n");
getchar();

/* Free all allocations. */
MblobFree(BlobResult);
MblobFree(FeatureList);
MbufFree(BinImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

Holes

In some cases, you can also distinguish between blobs by determining the number of holes that they have (`M_NUMBER_OF_HOLES`). For example, you could distinguish between bolts and nuts in the *bolts.mim* image by counting blob holes. However, this is not a very robust measure, as a single noise pixel in a bolt blob would count as a hole.

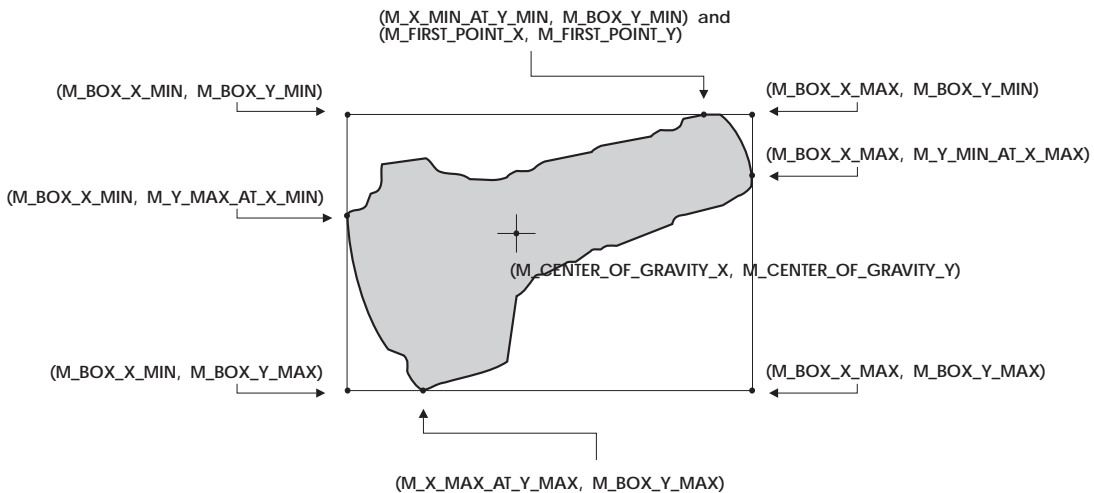
Finding the blob location

Finding the location of blobs in an image can sometimes be more useful than finding their shape or size. For example, if a robotic arm needs to pick up several items regardless of their type, it can use their location in an acquired image to determine their actual physical position.

You can also use the blob location to determine if a blob touches the image borders. If there are any such blobs, you might want to adjust the camera's field of view so that all items are completely represented in the image, or you might want to exclude these blobs.

Blob points

You can determine the following blob points by adding them to the feature list:



The center of gravity can be calculated in binary or grayscale mode. To calculate the latter, you must provide *MblobCalculate()* with a grayscale image.

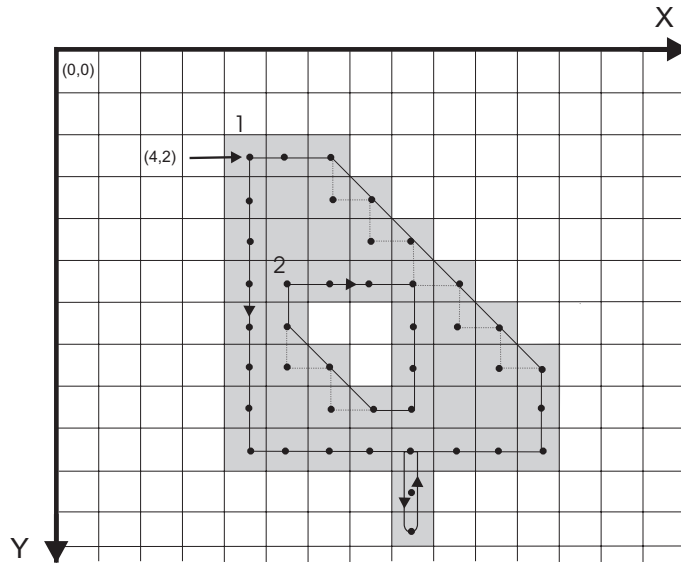
Chained pixels

You can obtain the coordinates of pixels bordering blobs or delimiting holes in blobs, in a counterclockwise or clockwise direction respectively. These pixels are referred to as chained pixels (M_CHAINS).

You can use the chained pixel coordinates to create a chain code. A chain code is a directional code that records an object's boundary as a discrete set of vectors, where each vector points to the next pixel in the chain.

Chained pixels always form a closed chain. This implies that the starting pixel in the chain is also the closing one. If your blob has regions which are 1 pixel wide, these pixels are chained twice, once in the forward direction and then in the opposite direction.

In the diagram below, the thick lines illustrate pixels which are chained twice. The diagram also illustrates chained pixels of a blob in an 8 and 4-connected lattice, where the solid lines illustrate chained pixels in an 8-connected lattice, and the dotted lines illustrate how chained pixels deviate in a 4-connected lattice. Also, note that the blob's outermost chain is identified as index 1. Chains that delimit holes in blobs are identified by subsequent indexes.



The `M_CHAINS` feature calculates four separate chain features. This includes `M_NUMBER_OF_CHAINED_PIXELS` which calculates the total number of chained pixels for each blob or a specified blob; the `M_CHAIN_INDEX` feature which assigns an index to each chained pixel, for every chain within in a blob; and the `M_CHAIN_X` and `M_CHAIN_Y` features which calculate the x and y coordinates of all chained pixels within a blob.

When retrieving results for chain features, you should retrieve results for the number of chained pixels (`M_NUMBER_OF_CHAINED_PIXELS`) first. Retrieving results for this feature allows you to allocate an array which is large enough for the other chain results. Thus, to find the results for a single chain, check the `M_CHAIN_INDEX` array for the appropriate chain indices, and retrieve the x and y results from the corresponding elements in the `M_CHAIN_X` and `M_CHAIN_Y` arrays.

For the blob shown on the previous page, the following arrays would result:

M_CHAIN_INDEX	M_CHAIN_X	M_CHAIN_Y
1	4	2
1	4	3
1	4	4
1	4	5
:	:	:
2	5	5
2	6	5
2	7	5
2	8	5

Moments

Using the blob analysis module, you can also calculate the moments used to find the center of gravity, as well as other grayscale or binary moments. The *MblobSelectMoment()* command allows you to add any moment to the feature list, whereas *MblobSelectFeature()* allows you to add only the more common moments.

You can calculate either central or ordinary moments. Central moments use coordinates that are relative to the center of gravity of the blob, and therefore are independent of a blob's position within the image, whereas, ordinary moments are affected by the blob position because they use coordinates relative to the top left corner of the image.

Finding the label value

The blob analysis module automatically calculates label values for included blobs when a call to *MblobCalculate()* is made. You can obtain a label value for a single blob with a call to *MblobGetLabel()*, by specifying the blob's coordinate. A label value can be useful to obtain calculation results for a single blob with *MblobGetResultSingle()* or *MblobGetRuns()*.

Location, length and number of runs

A run is defined as a horizontal string of consecutive foreground pixels. The blob analysis module can be used to obtain the total number of runs (`M_NUMBER_OF_RUNS`) for each blob. Results are obtained with *MblobGetResult()* or *MblobGetResultSingle()*.

To obtain the length and coordinate of each run in a specific blob, use the *MblobGetRuns()* command. The runs that make up each blob can be used to calculate features that are not supported directly by MIL.

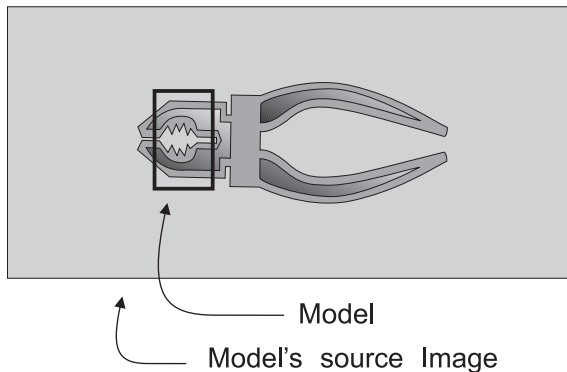
Chapter 11: Pattern matching

This chapter explains quick techniques to perform alignment operations, using the MIL pattern matching (recognition) module.

Pattern matching

The MIL package includes a pattern matching module that uses normalized grayscale correlation to help solve machine vision problems such as alignment, measurement, and inspection of objects. The module also provides quick techniques to find horizontal, vertical, and angular displacement of most images.

The main function of the pattern matching module is to search for occurrences of a pattern in an image. MIL refers to the pattern for which you are searching as the *search model* and the image from which it is extracted as the *model's source image*.



The image being searched is called the *target image*.

This chapter describes how these techniques can be applied to different types of targets. The next chapter looks at defining a search model, finding the occurrences of this model in the target image, and understanding the search algorithm.

With MIL, you can only perform pattern matching operations on 8-bit grayscale unsigned buffers.

Simple alignment techniques

Vertical and horizontal alignment

MIL can find the vertical and horizontal displacement of a target image by comparing the location of a unique model, taken from an aligned source image, with its actual location in the target image. A unique model can be chosen from any location in the aligned image as long as the model is known to appear in a shifted target image.

Allocating the model

You can **automatically** allocate a unique model, using *MpatAllocAutomodel()*. This function allocates the best unique search model for a given image.

Preprocessing the model

Once the model is defined, you must use *MpatPreprocModel()* to train the system to find the model in the most efficient manner. This function analyzes the model and determines which shortcuts can be safely used during the search.

Finding the model in the target image

Now, you are ready to find the model in the target image. Allocate a pattern matching result buffer, using *MpatAllocResult()*, and then call *MpatFindModel()* to find the model in the target image.

By comparing the model's coordinates in the model's source image with those in the target image, you obtain the vertical and horizontal displacement of the target image.

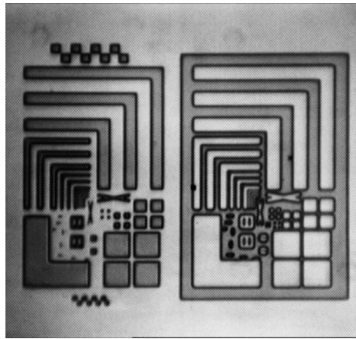
Δ Important

The coordinates resulting from a search return the **reference position** of the model, relative to the top-left corner of the target image. To find the equivalent coordinates in the model's source image, use *MpatInquire()* with `M_ORIGINAL_X` and `M_ORIGINAL_Y`.

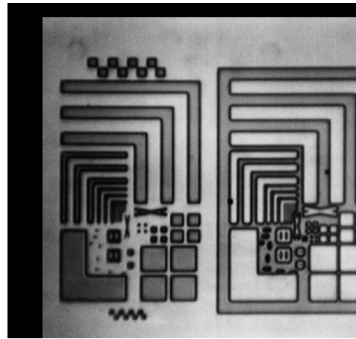
Note, once the model is defined, you can perform the search operation on an unlimited number of target images.

A wafer alignment example

The following sample program finds the vertical and horizontal displacement of a wafer image.



Model's source image



Target image

```

/* File name: mshift.c
 * Synopsis: This program finds the horizontal and vertical
 *           displacement of a wafer image.
 */

#include <stdio.h>
#include <mil.h>

/* Source and target images file specifications. */
#define MODEL_IMAGE_FILE M_IMAGE_PATH"wafer.mim"
#define TARGET_IMAGE_FILE M_IMAGE_PATH"shfwafer.mim"
#define IMAGE_WIDTH      512L
#define IMAGE_HEIGHT      480L

/* Model width, height, maximum displacement, initial position */
#define MODEL_WIDTH      64L
#define MODEL_HEIGHT      64L
#define MODEL_MAX_DISPLACE 64L

void main(void)
{
    MIL_ID MilApplication,          /* Application identifier. */
    MilSystem,                     /* System identifier.      */
    MilDisplay,                    /* Display identifier.     */
    MilImage,                      /* Image buffer identifier.*/
    MilSubImage,                   /* Sub-image buffer identifier.*/
    Model,                         /* Model identifier.       */
    Result;                        /* Result buffer identifier.*/
    long PosX, PosY;               /* Model position.        */
    long AllocError;               /* Allocation error variable.*/
    double OrgX=0.0, OrgY=0.0;     /* Original center of model.*/
    double x=0.0, y=0.0, Score=0.0; /* Result variables.      */

```

(cont. ...)

```

/* Allocate defaults. */
MappAllocDefault(M_SETUP, &MilApplication, &MilSystem,
                 &MilDisplay, M_NULL, &MilImage);

/* Load model image into an image buffer. */
MbufLoad(MODEL_IMAGE_FILE, MilImage);

/* Restrict the region to be processed to the bottom right corner
 * of the image. */
MbufChild2d(MilImage, IMAGE_WIDTH/2, IMAGE_HEIGHT/2,
            IMAGE_WIDTH/2, IMAGE_HEIGHT/2, &MilSubImage);

/* Announce the automatic model definition. */
printf("A model is being automatically defined in the source image, ");
printf("please wait...\n\n");

/* Automatically allocate normalized grayscale type model. */
MpatAllocAutoModel(MilSystem, MilSubImage, MODEL_WIDTH, MODEL_HEIGHT,
                  MODEL_MAX_DISPLACE, MODEL_MAX_DISPLACE, M_NORMALIZED,
                  M_DEFAULT, &Model);

/* Check for a successful model allocation. */
MappGetError(M_CURRENT, &AllocError);
if (!AllocError)
{
    MpatInquire(Model, M_ALLOC_OFFSET_X+M_TYPE_LONG, &PosX);
    MpatInquire(Model, M_ALLOC_OFFSET_Y+M_TYPE_LONG, &PosY);
    MpatInquire(Model, M_ORIGINAL_X, &OrgX);
    MpatInquire(Model, M_ORIGINAL_Y, &OrgY);
    /* Draw box around model. */
    MgraRect(M_DEFAULT, MilSubImage, PosX - 1, PosY - 1,
             PosX + MODEL_WIDTH, PosY + MODEL_HEIGHT);
    printf("Model successfully defined as shown on the\n");
    printf("displayed image.\n");
    printf("Press <Enter> to continue.\n");
    getchar();

    /* Load target image into an image buffer. */
    MbufLoad(TARGET_IMAGE_FILE, MilImage);

    /* Allocate result. */
    MpatAllocResult(MilSystem, 1L, &Result);

    /* Find model. */
    MpatFindModel(MilSubImage, Model, Result);
    /* If one model was found above the acceptance threshold set. */
    if (MpatGetNumber(Result, M_NULL) == 1L)
    {
        /* Get results. */
        MpatGetResult(Result, M_POSITION_X, &x);
        MpatGetResult(Result, M_POSITION_Y, &y);
        MpatGetResult(Result, M_SCORE, &Score);
    }
}

```

(cont. ...)

```

        /* Draw a box around occurrence. */
        MgraRect(M_DEFAULT, MilSubImage,
            (long)(x + 0.5) - (MODEL_WIDTH/2) - 1,
            (long)(y + 0.5) - (MODEL_HEIGHT/2) - 1,
            (long)(x + 0.5) + (MODEL_WIDTH/2),
            (long)(y + 0.5) + (MODEL_HEIGHT/2));

        /* Analyze and print results. */
        printf("A misaligned version of the source image was loaded.\n\n");
        printf("Image was found to be offset by %.2f in X, and %.2f in\n",
            x - OrgX, y - OrgY);

        printf("Model match score is %.1f percent.\n", Score);
        printf("Press <Enter> to end.\n");
        getchar();

    }
    else
    {
        printf("Error: Pattern not found properly.\n");
        printf("Press <Enter> to end.\n");
        getchar();
    }
    /* Free result buffer and model. */
    MpatFree(Result);
    MpatFree(Model);

}
else
{
    printf("Error: Automatic model definition failed.\n");
    printf("Press <Enter> to end.\n");
    getchar();
}

/* Free child image and defaults. */
MbufFree(MilSubImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

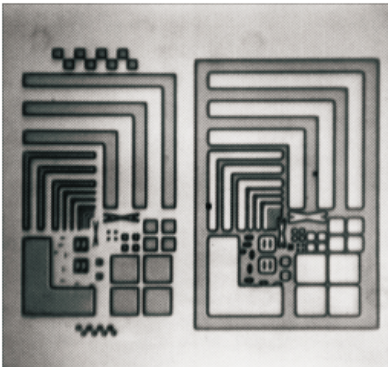
By executing *mshift.c*, you will find that *shfwafer.mim* is shifted by approximately 50 pixels horizontally and 20 pixels vertically.

Angular alignment

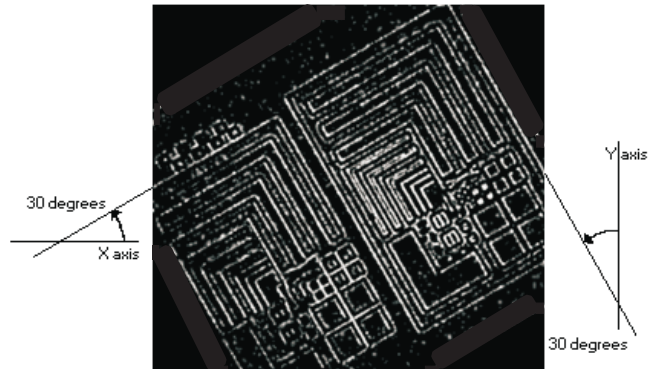
You can find the angular displacement of a target image, or of an object in that image, in a number of ways. The choice of method will depend on whether whole-image or object orientation is required, the shape and distinctiveness of the object, the complexity of the image background, and the degree of angular accuracy that is required. In this chapter, we will discuss basic methods to determine the orientation of an image and the orientation of a model in an image.

Whole-image orientation

You can quickly determine, with MIL, the orientation of an image based on the dominant edges in the image and their angular displacement from the image frame. The image can have either uni-directional dominant edges (such as parallel stripes) or bi-directional perpendicular dominant edges. The *MpatFindOrientation()* function is designed for images with smooth edges, usually obtained when grabbing an image with a camera. It will not work well on an artificially generated image unless the lines and edges are anti-aliased.



typical image



edge detection on a 30 degree rotated image

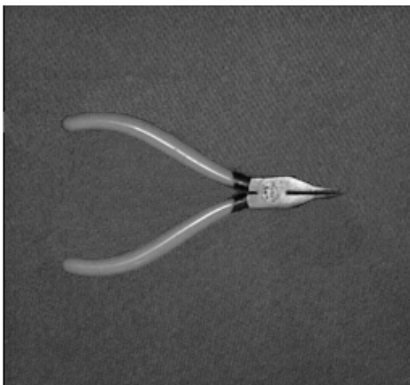
Note that if an image does not have dominant edges, its orientation cannot be well defined. In addition, if the image's background contains edges, the orientation of these edges might be found instead.

Whole-image orientation can be determined by following these steps:

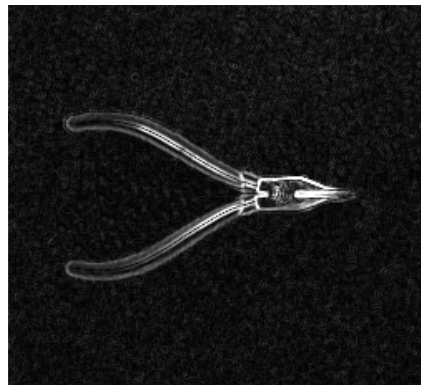
1. Ensure that the target image contains predominant edges.
2. Allocate a pattern matching result buffer, using *MpatAllocResult()*.
3. Call *MpatFindOrientation()*, specifying no model identifier (M_NULL), the identifier of the target image buffer, the appropriate result range for the type of target image, and the identifier of the result buffer. For images with uni-directional predominant edges, the result range should be set to M_RESULT_RANGE_180. Alternatively, for images with bi-directional edges, the result range should be set to M_RESULT_RANGE_45 or M_RESULT_RANGE_90.
4. Call *MpatGetResult()* to get the angle of orientation, returned as a value in the specified result range.

Object orientation

The orientation of a single large object on a smooth uniform background can be found by defining it as an M_ORIENTATION type model and searching for the general contours of the object in a target image. The model should be created from an image with a uniform background, so that the contours of the object can be properly defined, thereby making the operation more effective.



Typical image



Contours are well defined

Orientation of a model can be found by following these steps:

1. Ensure that the typical image contains a unique object on a smooth uniform background.
2. Create an M_ORIENTATION model of the unique object, using *MpatAllocModel()*.
3. Preprocess the model using *MpatPreprocModel()*.
4. Allocate a pattern matching result buffer, using *MpatAllocResult()*.
5. Call *MpatFindOrientation()*, specifying the identifier of an M_ORIENTATION model type, the identifier of the target image buffer, the appropriate result range for the image, and the identifier of the buffer in which to store the results. For model-orientation searches, use a 360° range to include all the rotational possibilities of the model.
6. Call *MpatGetResult()* to get the angle of orientation from the result buffer.

Chapter 12: Models, searches, and search parameters

This chapter explains how to define search models and parameters to perform and optimize more complex pattern matching operations.

Performing a search

In the previous chapter, we discussed some quick techniques to determine the alignment of a target image. This chapter looks at defining a search model and finding the occurrences of this model in the target image to help solve machine vision problems such as ones listed below:

- In machine guidance applications, mechanical devices need to be informed of the location of parts to be picked. Therefore, the search model must be specific to the part in question; it cannot be for an arbitrary location.
- When performing an alignment using gauging techniques, the location of two or more points of reference (fiducial marks) is required. To perform this process, you must define your own model to uniquely identify the reference points.

Steps to performing a search

The steps involved in performing a search with a model are as follows:

1. Load or grab a model's source image.
2. Define the model from the model's source image.
3. Optionally, specify a range for angular search.
4. Optionally, set the model's "don't care" pixels to exclude certain pixels from the search process.
5. Specify the model's search parameters (search constraints).
6. Preprocess the model.
7. Allocate a result buffer.
8. Grab a target image. Optionally, process it to improve its quality.
9. Find the model in the target image.
10. Read the search results.

In general, the first seven steps are performed once, while steps 8 through 10 are repeated as required. Note, in practice, models are usually saved on disk, using *MpatSave()*; therefore steps 1 through 6 are often replaced by a single step that restores a saved model from disk, using *MpatRestore()*.

*Load the model's
source image*

Load the image from which to extract the model. This image must be of the best quality possible. If your images tend to be noisy, try to clean the image, using the image processing techniques discussed previously in this manual.

Define the model

Use *MpatAllocModel()* to define which portion of this image is to be used as your model, or use *MpatAllocAutoModel()* to have MIL automatically generate the model for you (see previous chapter). Upon allocation, the model is extracted from the selected region in the model's source image buffer and stored into a non-displayable model buffer. The model's source image buffer is then no longer needed. To view the portion of the image from which the model was extracted, use *MpatCopy()*.

When allocating a model, you must specify its size. Generally, relative to the target image, small models take longer to find than larger ones, although very large models can also be time-consuming.

Specify search angle

You can set the angular search limits for the specified model, using *MpatSetAngle()*. By default, the angle of search is 0°. However, you can enable and specify a rotational range of up to 360°, as well as the required precision of the resulting angle and the interpolation mode used for the rotated model. These settings can influence the speed of the search significantly. The accuracy of the search can also be influenced.

*Set model's "don't care"
pixels*

You can set pixels in the model to the "don't care" state, using *MpatSetDontCare()*. These pixels will not be considered when finding occurrences of the model in a target image. Note that setting don't care pixels also affects the speed of the search.

Specify model parameters

When search models are allocated (whether automatically or manually), they are assigned a set of default search parameters (search constraints). Some of the parameters can be changed. For instance, you can limit the search to a certain region of the target image (*MpatSetPosition()*), restrict the number of occurrences to find (*MpatSetNumber()*), and set the level of acceptance (*MpatSetAcceptance()*).

Preprocess the model

The preprocessing stage uses the known model, together with a typical (optional) target image, to decide on the optimal search strategy for subsequent search operations.

Allocate a result buffer

Before performing the search, you must allocate a result buffer, using *MpatAllocResult()*. This buffer is used to store the result values for subsequent search operations. You can delete the result buffer, using *MpatFree()*.

Acquire the new target image

Once the model is defined and the model's search parameters meet your application needs, the target image should be loaded from disk, or acquired from the input device into an image buffer.

Find the model

You can now search in the target image for the coordinates of model occurrences, using *MpatFindModel()*. The search is performed according to the defined model parameters.

You can also search for several models of the same size and search region in the same image, using *MpatFindMultipleModel()*. This function finds occurrences of the specified models in the given image and returns the position of each occurrence for each model or of the best matches from the group of models. Note that in the former case, you have to allocate and specify a result buffer for each model that is being sought. In the latter case, you have to allocate and specify a single result buffer. If you have to search for several different models, this is more efficient.

Read the search results

To read results, use *MpatGetNumber()* and *MpatGetResult()* to get, respectively, the number of model occurrences found in the target, and the required results.

The following sample program (*msearch.c*) shows how to define a model and then find this model in a target image. It also demonstrates the sub-pixel accuracy of *MpatFindModel()*.

```

/* File name: msearch.c
 * Synopsis: This program defines a model and then searches for it
 *           in a shifted version of the image. The model is saved
 *           on disk for future use.
 */

#include <stdio.h>
#include <mil.h>

/* Source image file specifications. */
#define IMAGE_FILE    M_IMAGE_PATH"board.mim"

/* Image shifting values. */
#define SHIFT_X       4.25
#define SHIFT_Y       7.25

/* Model position and size. */
#define MODEL_XPOS     304L
#define MODEL_YPOS     126L
#define MODEL_WIDTH    86L
#define MODEL_HEIGHT   100L

/* Minimum match score to determine acceptability of model (default). */
#define MODEL_MIN_MATCH_SCORE 70.0

/* Minimum accuracy for the search. */
#define MODEL_MIN_ACCURACY 0.1

/* File in which to save model. */
#define MODEL_FILE     M_IMAGE_PATH"chip.mmo"

/* Absolute value macro. */
#define absolute(x) ((x) < 0.0) ? -(x) : (x)

void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem,             /* System identifier. */
    MilDisplay,            /* Display identifier. */
    MilImage,              /* Image buffer identifier. */
    Model,                 /* Model identifier. */
    Result;                /* Result identifier. */
    double XOrg=0.0, YOrg=0.0; /* Model original positions. */
    double x=0.0, y=0.0;      /* Model positions. */
    double Score=0.0;         /* Model correlation score. */

    (cont. ...)

```

```

/* Allocate defaults. */
MappAllocDefault(M_SETUP, &MilApplication, &MilSystem,
                 &MilDisplay, M_NULL, M_NULL);

/* Restore source image into an automatically allocated image buffer. */
MbufRestore(IMAGE_FILE, MilSystem, &MilImage);

/* Display the image buffer. */
MdispSelect(MilDisplay, MilImage);

/* Allocate a normalized grayscale model. */
MpatAllocModel(MilSystem, MilImage, MODEL_XPOS, MODEL_YPOS,
               MODEL_WIDTH, MODEL_HEIGHT, M_NORMALIZED, &Model);

/* Set the search accuracy to high. */
MpatSetAccuracy(Model, M_HIGH);

/* Set the search model speed to M_HIGH. */
MpatSetSpeed(Model, M_HIGH);

/* Preprocess the model. */
MpatPreprocModel(MilImage, Model, M_DEFAULT);

/* Draw a box around the model in the model image. */
MgraRect(M_DEFAULT, MilImage,
          MODEL_XPOS - 2,
          MODEL_YPOS - 2,
          MODEL_XPOS + MODEL_WIDTH + 1,
          MODEL_YPOS + MODEL_HEIGHT + 1);

/* Pause to show the original image and model position. */
printf("A model was defined in the source image.\n");
printf("Press <Enter> to continue.\n");
getchar();

/* Translate the image on a subpixel level. */
MimTranslate(MilImage, MilImage, SHIFT_X, SHIFT_Y, M_DEFAULT);
printf("Source image was shifted by %.2f in X and %.2f in Y.\n\n",
       SHIFT_X, SHIFT_Y);

/* Allocate result buffer. */
MpatAllocResult(MilSystem, 1L, &Result);

/* Find the model in the target buffer. */
MpatFindModel(MilImage, Model, Result);

```

(cont. ...)

```

/* If one model was found above the acceptance threshold. */
if (MpatGetNumber(Result, M_NULL) == 1L)
{
    /* Read results and draw a box around model occurrence. */
    MpatGetResult(Result, M_POSITION_X, &x);
    MpatGetResult(Result, M_POSITION_Y, &y);
    MpatGetResult(Result, M_SCORE, &Score);
    MgraRect(M_DEFAULT, MilImage,
        (long)(x + 0.5) - (MODEL_WIDTH / 2),
        (long)(y + 0.5) - (MODEL_HEIGHT / 2),
        (long)(x + 0.5) + (MODEL_WIDTH / 2) - 1,
        (long)(y + 0.5) + (MODEL_HEIGHT / 2) - 1);

    /* Pause to show the shifted image and print out the difference
    * to confirm the sub-pixel accuracy.
    */
    MpatInquire(Model, M_ORIGINAL_X, &XOrg);
    MpatInquire(Model, M_ORIGINAL_Y, &YOrg);
    printf("The model was found to be shifted by %.2f in X and %.2f in Y.\n",
        x - XOrg, y - YOrg);
    printf("Model match score is %.1f percent.\n\n", Score);

    /* Save model to disk for future use if verification was successful. */
    if (
        (absolute((x - XOrg) - SHIFT_X) <= MODEL_MIN_ACCURACY) &&
        (absolute((y - YOrg) - SHIFT_Y) <= MODEL_MIN_ACCURACY) &&
        (Score >= MODEL_MIN_MATCH_SCORE)
    )
    {
        MpatSave(MODEL_FILE, Model);
        printf("Model was saved on disk as '%s'.\n", MODEL_FILE);
    }
    else
    {
        printf("Model verification error, model was not saved!\n");
    }
}
else
{
    printf("Model was not found, model was not saved!\n");
}

/* Wait for a key press to continue. */
printf("Press <Enter> to end.\n");
getchar();

/* Free all allocations. */
MpatFree(Result);
MpatFree(Model);
MbufFree(MilImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);
}

```

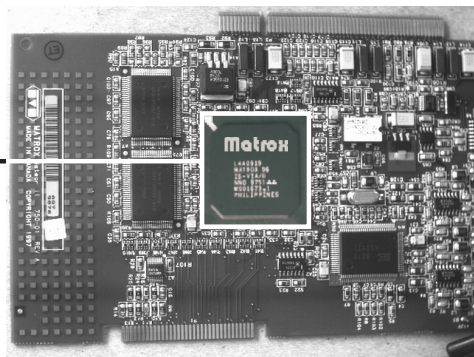
Rotation

Besides `M_ORIENTATION` model searches made with `MpatFindOrientation()` (discussed in the previous chapter), there are two ways to search for a model that can appear at different angles using the `MpatFindModel()` function:

- Search for rotated versions of the model.
- Search for models taken from the same region in rotated images.

Target image

Model



Model



An `M_NORMALIZED` model
internally rotated

■ = resulting `M_DONT_CARE` pixels



An `M_NORMALIZED+`
`M_CIRCULAR_OVERSCAN`
model internally rotated

The following describes how to create the models to perform these types of searches.

Creating rotated versions of the models

To implement the first type of search, allocate an `M_NORMALIZED` model with `MpatAllocModel()`. Then, enable and specify the angular range in which it can appear with `MpatSetAngle()`. When you call `MpatPreprocess()`, it will internally create different models by rotating the original model at the required angles, assigning don't care pixels to regions that do not have corresponding data in the original model.

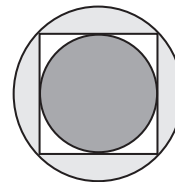
This method should only be used when the pixels surrounding the model follow no predictable pattern (for example, when searching for loose nuts and bolts lying on a conveyor with an inconsistent background).

Extracting models at different rotations

To implement the second type of search, first allocate an `M_NORMALIZED + M_CIRCULAR_OVERSCAN` model with `MpatAllocModel()`; this will extract the model as well as circular overscan data from the model's source image (specifically, MIL extracts the region enclosed by a circle which circumscribes the model). Second, enable and specify the angular range in which the model can appear with `MpatSetAngle()`. When you call `MpatPreprocess()`, it will extract different orientations of the model from the overscanned model.

It is recommended that the model be as square as possible: the longer the rectangle, the smaller the number of consistent central pixels in every model. Therefore, this type of model should only be used when the model's distinct features lie in the center of the region, so that they are included in all models when rotated.

- ☐ Model
- ☐ Circular overscan data
- ☐ Consistent central pixels



As mentioned, a larger region than the one defined will be fetched from the model's source image. Therefore, the model must not be extracted from a region too close to the edge of the model's source image.

The pixels surrounding the model should be relevant to the positioning of the pattern (that is, the model should appear in the target image with the same overscan data). An example is the image of an integrated circuit.

Both methods find the position and match score of the model in a target image.

Finally, it should be noted that MIL's implementation of *MpatFindModel()* with a `M_CIRCULAR_OVERSCAN` type model is significantly faster than that of a `M_NORMALIZED` model when performing an angular search.

Setting the angle of search

By default, the angle of search is 0° . However, you can specify a rotational range of up to 360° , as well as the required precision of the resulting angle and the interpolation mode used for the rotated model. These settings can influence the speed of the search significantly. The accuracy of the search can also be influenced.

When an angular range has been specified with *MpatSetAngle()*, *MpatPreprocess()* creates a model for every x degrees within the range, where x is determined by the specified tolerance (`M_SEARCH_ANGLE_TOLERANCE`). Tolerance defines the full range of degrees within which the pattern in the target image can be rotated from a model at a specific angle and still meet the acceptance level.

After the approximate location is found, MIL fine-tunes the search, according to the specified accuracy (`M_SEARCH_ANGLE_ACCURACY`). To be effective, you must set the degree of accuracy to a value smaller than that of the rotation tolerance.

When searching within a range of angles, you should use as narrow a range as possible, since the operation can take a long time to perform. Note that the model is rotated according to the interpolation mode

(M_SEARCH_ANGLE_INTERPOLATION_MODE).

Determining the rotation tolerance of a model

Every model has its own particular rotation tolerance. This tolerance is dependent on the individual model characteristics and surrounding target image features. To determine the rotation tolerance of a model:

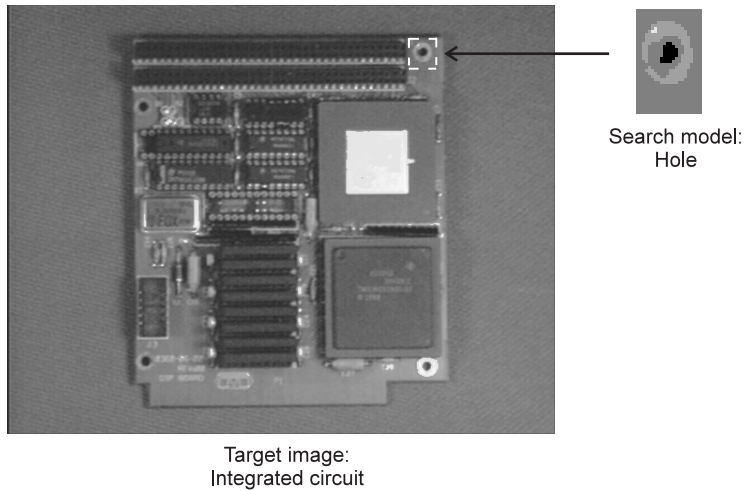
1. Set the search angle of a model to the same angle as the sought for pattern in a sample target image. However, set the positive and negative delta values to zero since you want to test by how much a pattern in an image can be rotated and still correlate with a model at a specific angle.
2. Use the *MimRotate()* function to rotate the image in very small, positive increments (for example, 0.5 degrees), and perform a *MpatFindModel()* operation at every angle. Make sure that the image's center of rotation is the same as that of the model, otherwise the resulting tolerance will not be accurate. Note, when rotating the image, always set the angle from the image's original position to avoid interpolating the image more than once. Check the results for the greatest angle that produces an acceptable score.
3. Repeat steps 1 through 3, rotating the image in negative increments.
4. Take the minimum of the absolute value of these angles. Double this angle and set it as the rotation tolerance for the angular search.

Masking the model

Often your search model contains regions that you need MIL to ignore when searching for the model in the target image. These regions might be noise pixels or simply regions that have nothing to do with what you are searching for.

An example

For instance, in a machine guidance application, a mechanical device might need to know where mounting holes are located on a circuit board so that screws can be properly inserted. In this case, a mounting hole would be the search model and the circuit board would be the target image.



In the above image, the search model contains too much of the actual board; it might not match holes in different areas of the board.

In such cases, parts of the search model (any model type other than `M_ORIENTATION`) can be masked by setting pixel values in certain regions to "don't cares". MIL then ignores these regions when searching for occurrences of the model.

In our example, you would need to mask the edges of the search model, as follows:



The dark region around the hole should be masked

The unmasked region of the search model now more closely resembles the pattern for which to search; it is more circular in shape and contains little of the actual board.

To create a mask:

1. View the portion of the image from which the model was extracted, using *MpatCopy()*.
2. Clear the pixels that should be set to "don't care", using the appropriate *Mgra...()* function.
3. Call *MpatSetDontCare()*, specifying the image along with the foreground color used to draw the "don't care" pixels. This function sets the model's "don't care" pixels.
4. To view the mask, call *MpatCopy()* again; this time specify if you want to view the masked image or only the mask.

When you change the "don't care" pixels of a model, you should preprocess the search model again.

Search parameters

Once a model is defined (whether manually or automatically), it is assigned set of default search parameters (search constraints). Some of these parameters can be changed. You can change these parameters:

- The number of occurrences to find.
- The threshold for acceptance and certainty.
- The model's reference position.
- The region to search in the target image.
- The positional accuracy.
- The search speed.

Specifying the number of matches

You can specify how many matches to try to find, using *MpatSetNumber()*. If all you need is one good match, set it to one (the default value) and avoid unnecessary searches for further matches. If a correlation has a match score above the certainty level, it is automatically considered an occurrence (default 80%), the remaining occurrences will be the best of those above the acceptance level.

Setting the acceptance level

The level at which the correlation (match score) between the model and the pattern in the image is considered a match is called the acceptance level.

You can set the acceptance level for the specified model, using *MpatSetAcceptance()*. If the correlation between the target image and the model is less than this level, they are not considered a match. A perfect match is 100%, a typical match is 80% or higher (depending on the image), and no correlation is 0%. If your images have considerable noise and/or distortion, you might have to set the level below the default value of 70%.

*Typical match has an
80 to 100% correlation*

However, keep in mind that such poor-quality images increase the chance of false matches and will probably increase the search time.

Note, perfect matches are generally unobtainable because of noise introduced when grabbing images.

When you ask for a specific number of matches (using *MpatSetNumber()*, the *MpatFindModel()* command might not find that number; you should always call *MpatGetNumber()* to see how many occurrences were actually found. When multiple results are found, they are returned in decreasing order of match score (that is, best match first).

Setting the 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 very good match and can stop searching the rest of the image for a better one. 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, would 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 a certainty level that is reasonable for your needs. 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 target 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.

Symmetrical models fall into this category. At certain angles symmetrical models might seem like an occurrence in the

target image, but if the search was completed, a match with a higher score would be found. Use *MpatSetCertainty()* to set the certainty level.

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.

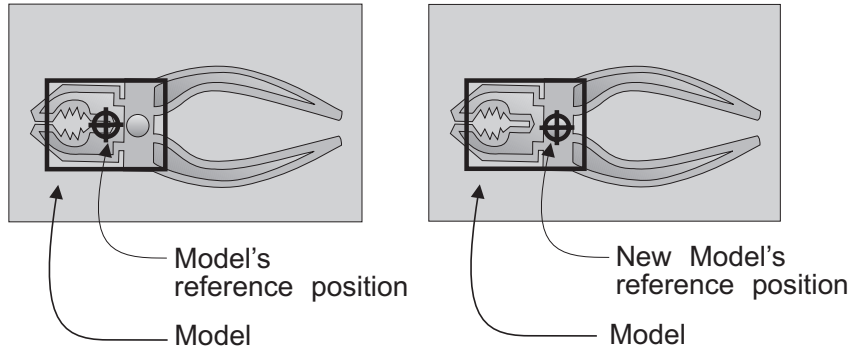
Another common case is a pattern that usually produces very good scores (say above 80%), but occasionally a degraded image produces a much lower score (say 50%). Obviously, you must set the acceptance level to 50%; otherwise you will never get a match in the degraded image. But what value is appropriate for the certainty level? If you set it to 50%, you take the 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.

Redefining the model's reference position

The coordinates returned by *MpatGetResults()*, after a call to *MpatFindModel()* are the coordinates of the model's reference position (in pixel or real-world units, depending on whether the imaging setup is calibrated; see *Chapter 7 Calibration*). By default, this reference position is defined to be at the geometric center. Note that, when using pixel units, results are returned relative to the top-left corner of the target image.

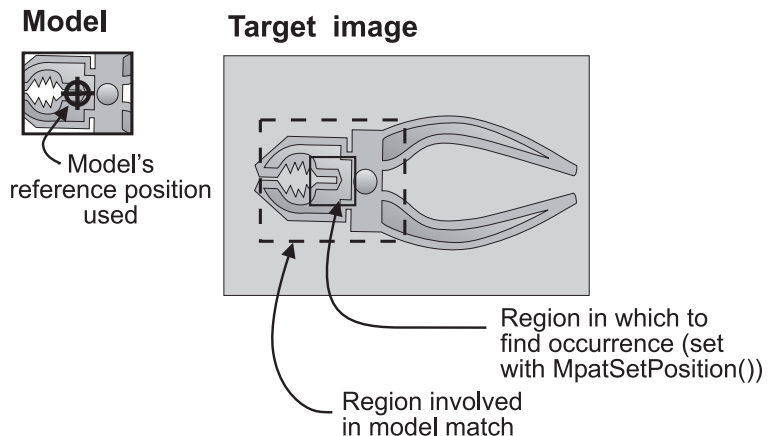
If there is a particular spot from which you would like results returned, you can change the model's reference position, using *MpatSetCenter()*. For example, if your model has a hole and you want to find results with respect to this hole, change the

reference position of the model accordingly. Note that you can define the reference position to be outside of the model's boundary.



Selecting the search region

Instead of searching the entire region of an image, you can limit the search region with *MpatSetPosition()*. This function specifies the region in which to find the model's reference position. Therefore, the search region can even be smaller than the model. If you have redefined the model's reference position (with *MpatSetCenter()*), make sure that the search region defined by *MpatSetPosition()* covers this new reference position and takes into account the angular search range of the model.



In general, you should not use a child buffer to delimit the search region to a portion of an image; this might cause the search routine to have border or edge effects and be less accurate (the routine does not assume that there is valid data outside of the buffer).

Search time is roughly proportional to the region searched; always set the search region to the minimum required when speed is a consideration.

Positional accuracy

You can set the positional accuracy for your search. Use *MpatSetAccuracy()* to set the required positional accuracy. It can be set to:

- M_LOW (typically ± 0.20 pixel)
- M_MEDIUM (typically ± 0.10 pixel)
- M_HIGH (typically ± 0.05 pixel)

Note, the actual precision achieved is dependent on the quality of the model and of the image (the tolerances listed above are typical for high-quality, low-noise images).

A less precise positional accuracy will speed up the search. Positional accuracy is also slightly affected by the search speed parameter (*MpatSetSpeed()*).

Setting the speed parameter

You can specify the algorithm's search speed, using *MpatSetSpeed()*. As you increase the speed, the robustness of the search operation (the likelihood of finding a model) decreases. Search speed is discussed at greater length in the section, "Speeding up the search".

Preprocess the search model

Once you are ready to search for the allocated model (either manually or automatically), you must preprocess the model. The preprocessing stage uses the known model to decide on the optimal search strategy for subsequent search operations. Preprocessing should be performed after all search constraints have been set. Use the *MpatPreprocModel()* function to preprocess the model.

MpatPreprocModel() has a parameter that allows you to specify a typical target image. Providing a typical image is optional; you can set this parameter to `M_NULL`. If you provide this image, it helps *MpatPreprocModel()* improve the search's robustness and optimize the strategy for subsequent search operations. You should only specify a typical image if the model will always appear on the same type of background.

If you save the model to disk, the model's preprocessing changes are stored with the model. Upon restoration, the model need not be preprocessed.

Speeding up the search

To ensure the fastest possible search, you should:

- Choose an appropriate model.
- Set the search speed parameter to the highest possible setting for your application.
- Set the search region to the minimum required. Search time is roughly proportional to the region searched, so don't search the whole image if you don't need to.
- Search the smallest range of angles required.
- Select the lowest positional accuracy that you need.
- Set the certainty level to the lowest reasonable value (so that the search can stop as soon as a good match is found).
- Search for multiple models at the same time, if possible.

Choose the appropriate model

The size of a model affects the search speed. In general, small models take longer to find than larger ones, although very large models can also be time-consuming. In general, the optimal size is approximately 128 x 128 pixels if you are searching a large region (for example, most of the image). Small models are found quickly when the search region is not too large.

Adjust the search speed parameter

The model has a search speed parameter that is used to set the speed of the search. As you increase the speed, the robustness of the search operation (the likelihood of finding a model) decreases. When you call *MpatPreprocModel()*, MIL analyzes the pattern in the model, and determines what shortcuts are appropriate; only shortcuts that are considered safe for a particular model are taken. This also means that higher search speeds might not be any faster for certain models, depending on the particular pattern. Higher search speeds reduce the positional accuracy very slightly.

You adjust the search speed parameter setting, using *MpatSetSpeed()*. This command has five settings:

- M_VERY_HIGH
- M_HIGH
- M_MEDIUM
- M_LOW
- M_VERY_LOW

As expected, the M_VERY_HIGH and M_HIGH speed settings allow the search to take all possible shortcuts, performing the search as fast as possible. Higher speed settings are recommended when searching on a good quality image or when using a simple model. Note, the search might have a lower tolerance for rotation when using this setting.

The M_MEDIUM speed setting is the default setting and is recommended for medium quality images or more complex models. A search with this setting is capable of withstanding up to approximately 5 degrees of rotation for typical models.

Use the M_LOW or the M_VERY_LOW speed settings only if the image quality is particularly poor and you have encountered problems at higher speeds. The speed parameter is discussed further in the algorithm description at the end of this chapter.

Effectively choose the search region and search angle

You can improve performance by not searching the whole image unnecessarily. Search time is roughly proportional to the region searched; set the search region to the minimum required, using *MpatSetPosition()*. You can also improve performance by selecting the lowest positional accuracy. In addition, for an angular search, select lowest angular accuracy (*MpatSetAngle()* with M_SEARCH_ANGLE_ACCURACY) and range required, in combination with the highest tolerance possible.

Searching for multiple models at the same time

When searching for multiple models of the same size and search region, it is more efficient to call the *MpatFindMultipleModel()* function instead of calling *MpatFindModel()* once for each model.

The pattern matching algorithm (for advanced users)

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 (see *Chapter 5: Image manipulation*). 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, the model does not 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"):

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

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.

In our case, 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 *MpatGetResult()* is actually:

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

Note, 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:

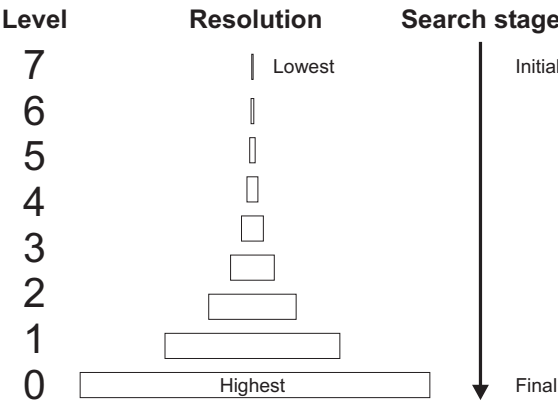
$$\sum I, \sum I^2, \sum IM$$

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

On a typical PC, the multiplications alone account for most of the computation time. A typical application might need to find a 128x128-pixel model in a 512x512-pixel image. In such a case, the total number of multiplications needed for an exhaustive search is $2 \times 512^2 \times 128^2$, or over 8 billion. On a typical PC, this would take a few minutes, much more than the 5 msec or so the search actually takes. Clearly, *MpatFindModel()* does much more than simply evaluate the correlation function at every pixel in the search region 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 the resolution of an image or model is 512x512 at level 1, then at level 2 it is 256x256, at level 3 it 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 resolutions 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 required, force a specific starting level, using *MpatSetSearchParameter()* with `M_FIRST_LEVEL`.

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

Note that the pyramidal representation of the buffer is generated each time *MpatFindModel()* or *MpatFindMultipleModel()* is called. However, you can save the pyramidal representation of the buffer (generated when *MpatFindModel()* or *MpatFindMultipleModel()* is called) in the result buffer, using *MpatSetSearchParameter()* with `M_TARGET_CACHING`. This pyramidal representation is re-used by consecutive calls to *MpatFindModel()* and *MpatFindMultipleModel()* as long as the same result buffer is used and the image, search region, and model size are not modified.

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 region. On most models, match peaks (pixel locations where the surrounding image is most similar to the model, and correlation results are largest) are several pixels wide. These can be found without evaluating the correlation function everywhere.

MpatPreprocModel() analyzes 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, an exhaustive initial search is performed. The search algorithm tends to be conservative; if necessary, force fast peak finding, using *MpatSetSearchParameter()* with `M_FAST_FIND`.

Using *MpatSetSearchParameter()* with `M_EXTRA_CANDIDATES`, you can set the number of extra peaks to consider. Normally, the search algorithm considers only a limited number of (best) scores as possible candidates to a match when proceeding at the most sub-sampled stage. You can add robustness to the algorithm, by considering more candidates, without compromising too heavily on search speed. In addition, you can use *MpatSetSearchParameter()* with `M_COARSE_SEARCH_ACCEPTANCE` to set a minimum match score, valid at all levels except the last level, to be considered for an occurrence of the model. This ensures that possible models are not discarded at lower levels, yet maintains the required certainty during the final level.

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 a high search speed, using *MpatSetSpeed()*. For most models, this has little effect on the score or accuracy, but does increase speed.

Sub-pixel accuracy

The highest match score occurs at only one point, 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 surface is fitted to the match scores around the peak

and, from the equation of the surface, the exact peak position is calculated. The surface 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 and precision used to hold the correlation result, is about 0.025 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 0.05 pixels is achieved for low-noise images. These numbers apply if you select high-search accuracy, using *MpatSetAccuracy()*, in which case the search always proceeds to resolution level 0.

If you select medium accuracy (the default), the search might stop at resolution level 1, and hence the accuracy is half of what can be attained at level 0. Selecting low accuracy might cause the search to stop at level 2, so the accuracy is reduced by an additional factor of two (to about 0.2 pixel).

Chapter 13: Optical character recognition

This chapter presents the features of the optical character recognition (OCR) module.

The MIL OCR module

Many types of industries require the analysis of character strings in images. For example, the semiconductor industry requires serial numbers printed on wafers to be read for tracking purposes. The pharmaceutical industry requires analysis of medicine bottle labels to ensure, for example, that expiry dates are properly printed.

The MIL optical character recognition (OCR) module provides a powerful and easy to use function set for reading and verifying character strings in 8-bit grayscale images, providing results such as quality scores and validity flags. The OCR module can read and verify mechanically generated, uniformly spaced, character strings of known lengths. The module is especially designed to operate on character strings in degraded images and can tolerate up to 10 degrees of rotation in the target string. The OCR module can also be used in conjunction with other MIL functions to develop hardware independent OCR programs for machine vision applications.

The module loads a grayscale representation of the font characters from a font file. Each character in the string to be read (target string) is compared to each character representation in this font. The representation with the closest match is chosen. You can adjust the value at which this match will be considered a success by setting the acceptance level. The result of a read or verify operation will yield a string of characters with the closest match, a confidence score for each character and its position in the target string, and a validity flag for each character and for the whole string.

Two predefined font files are provided to read and verify semiconductor wafer serial numbers of standard SEMI font character types. For applications requiring other font types, the OCR module supports the creation of custom fonts.

The module also supports user-specified character constraints, processing controls, and the automatic calibration of fonts. To ensure recognition accuracy, checksum calculations are performed when analyzing standard SEMI font character strings, and user-defined validation functions can be specified.

Matrox READER

Included with the OCR module is Matrox READER, a Windows utility giving you interactive access to all of the OCR functions. Matrox READER can be used, for example, for OCR experimentation, font calibration, control setting and operation timing. See the *read.me* file in the utility's directory for more details.

Matrox READER includes the MIL command interpreter, MILINTER. MILINTER can be used to access MIL functions while in Matrox READER, and can also be used to create custom scripts of MIL functions. For information on MILINTER, see the *Matrox Intellicam User Guide*.

Steps to reading or verifying a string in an image

The basic steps to read or verify a string in an image are:

1. Load or create a character font.
2. Calibrate the font to match the target image's character size and spacing.
3. Specify font character constraints and processing controls.
4. Allocate a result buffer.
5. Acquire or load a target image. Optionally, limit the area to be read and/or preprocess the image to improve its quality. Note that OCR can only process unsigned 8-bit buffers.
6. Read or verify the string in the target image.
7. Read the results.

In general, the first four steps are performed once, while steps 5 to 7 are repeated as desired. Note that you can save the font calibration results, character constraints, and processing controls with the character font. Therefore, steps 1 to 3 can be replaced by a single step which loads a font from disk.

Load or create a character font

You can load an existing font from disk using *MocrRestoreFont()* or you can create a custom font using *MocrCopyFont()* or *MocrImportFont()*.

Calibrate the font

Use *MocrCalibrateFont()* to determine the width, height, and spacing of characters in the target image, or manually program them using *MocrControl()*.

Specifying constraints and controls

You can use *MocrSetConstraint()* to specify the type of characters (alphabetic, numeric, or other) that should appear at a specific position in a string.

You can use *MocrControl()* to modify processing controls, such as the speed of the algorithm.

Allocate a result buffer

You must allocate a result buffer using *MocrAllocResult()*. This buffer will be used to store subsequent read or verify result values.

Acquire and pre-process a new target image

Once the previous steps have been performed, the target image should be loaded from disk or acquired from the input device into an image buffer. You have the option of limiting the area to be read and of removing noise to improve the target image prior to performing the OCR operation.

Read or verify the string

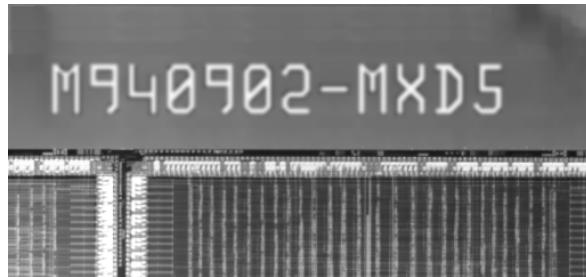
You can now read or verify the string in the target image using *MocrReadString()* or *MocrVerifyString()*. These operations are performed according to the defined character constraints and processing controls.

Read the results

You can obtain the OCR results using *MocrGetResult()*. You can determine whether or not the string or characters read were valid. In addition, you can obtain the characters read, their confidence scores, and their individual positions.

A typical application

The following example demonstrates how to read the serial number in an image of a semiconductor wafer, using the OCR functions in conjunction with other MIL functions. The serial number, printed on the wafer, is a standard SEMI font character string containing a checksum.



```

/* File name: mocrread.c
 * Synopsis: This program calibrates an OCR font (semi-font) and uses it to
 *           read the string present in the image. The string read is then
 *           printed to the screen and the calibrated font is saved to disk.
 */

#include <stdio.h>
#include <string.h>
#include <mil.h>

/* Target image character specifications. */
#define CHAR_IMAGE_FILE "ocrsemil.mim"
#define CHAR_SIZE_X_MIN 22.0
#define CHAR_SIZE_X_MAX 23.0
#define CHAR_SIZE_X_STEP 0.50
#define CHAR_SIZE_Y_MIN 43.0
#define CHAR_SIZE_Y_MAX 44.0
#define CHAR_SIZE_Y_STEP 0.50

/* Target reading specifications. */
#define READ_REGION_POS_X 30L
#define READ_REGION_POS_Y 40L
#define READ_REGION_WIDTH 420L
#define READ_REGION_HEIGHT 70L
#define READ_SCORE_MIN 50.0

(cont. ...)
```

```

/* Font file names. */
#define FONT_FILE_IN    "semil292.mfo"
#define FONT_FILE_OUT   "semicali.mfo"

/* Length of the string to read (null terminated) */
#define STRING_LENGTH    13L

/* Drawing color for the resulting string */
#define STRING_DRAWING_COLOR  255L

void main(void)
{
    MIL_ID MilApplication,          /* Application identifier. */
    MilSystem,                     /* System identifier. */
    MilDisplay,                    /* Display identifier. */
    MilImage,                      /* Image buffer identifier. */
    MilSubImage,                   /* Sub-image buffer identifier. */
    OcrFont,                       /* OCR font identifier. */
    OcrResult;                     /* OCR result buffer identifier. */
    char  String[STRING_LENGTH];   /* Array of characters to read. */
    double Score;                  /* Reading score. */

    /* Allocate defaults */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                     M_NULL, &MilImage);

    /* Load source image into image buffer. */
    MbufLoad(CHAR_IMAGE_FILE, MilImage);

    /* Restrict the region of the image in which to read the string.*/
    MbufChild2d(MilImage, READ_REGION_POS_X, READ_REGION_POS_Y,
                READ_REGION_WIDTH, READ_REGION_HEIGHT, &MilSubImage);

    /* Restore the OCR character font from disk. */
    MocrRestoreFont(FONT_FILE_IN, M_RESTORE, MilSystem, &OcrFont);

    /* Pause to show the original image and ask for the calibration string. */
    printf("The OCR font will be calibrated using the displayed image.\n");
    printf("Type the string present in the image followed by <Enter>.\n");
    scanf("%s",String);
    getchar();
   strupr(String);
    printf("\nCalibrating font...\n\n");

    /* Calibrate the OCR font. */
    MocrCalibrateFont(MilSubImage, OcrFont, String, CHAR_SIZE_X_MIN,
                     CHAR_SIZE_X_MAX, CHAR_SIZE_X_STEP,
                     CHAR_SIZE_Y_MIN, CHAR_SIZE_Y_MAX,
                     CHAR_SIZE_Y_STEP, M_DEFAULT);

    (cont. ...)

```

```

/* Set the user-specific character constraints for each string position */
MocrSetConstraint(OcrFont, 0, M_LETTER, M_NULL); /* A to Z only */
MocrSetConstraint(OcrFont, 1, M_DIGIT, "9"); /* 9 only */
MocrSetConstraint(OcrFont, 2, M_DIGIT, M_NULL); /* 0 to 9 only */
MocrSetConstraint(OcrFont, 3, M_DIGIT, M_NULL); /* 0 to 9 only */
MocrSetConstraint(OcrFont, 4, M_DIGIT, M_NULL); /* 0 to 9 only */
MocrSetConstraint(OcrFont, 5, M_DIGIT, M_NULL); /* 0 to 9 only */
MocrSetConstraint(OcrFont, 6, M_DIGIT, M_NULL); /* 0 to 9 only */
MocrSetConstraint(OcrFont, 7, M_DEFAULT, "-"); /* . only */
MocrSetConstraint(OcrFont, 8, M_LETTER, "M"); /* Monly */
MocrSetConstraint(OcrFont, 9, M_LETTER, "X"); /* X only */
MocrSetConstraint(OcrFont, 10, M_LETTER, "ABCDEFGH"); /* SEMI checksum */
MocrSetConstraint(OcrFont, 11, M_DIGIT, "01234567"); /* SEMI checksum */

/* Pause to signal the following read operation. */
printf("The string present in the displayed image will be read and\n");
printf("the result will be printed.\nPress <Enter> to continue.\n");
getchar();

/* Allocate an OCR result buffer. */
MocrAllocResult(MilSystem, M_DEFAULT, &OcrResult);

/* Read the string. */
MocrReadString(MilSubImage, OcrFont, OcrResult);

/* Get the string and its reading score. */
MocrGetResult(OcrResult, M_STRING, String);
MocrGetResult(OcrResult, M_SCORE, &Score);

/* Print the result. */
printf("\nThe string read is: \"%s\" (score: %.1f%%).\n\n", String, Score);

/* Draw the string under the reading region. */
MgraFont(M_DEFAULT, M_FONT_DEFAULT_LARGE);
MgraColor(M_DEFAULT, STRING_DRAWING_COLOR);
MgraText(M_DEFAULT, MilImage, READ_REGION_POS_X+(READ_REGION_WIDTH/4),
        READ_REGION_POS_Y+READ_REGION_HEIGHT, String);

/* Save the calibrated font if the reading score was sufficient. */
if (Score > READ_SCORE_MIN)
{
    MocrSaveFont(FONT_FILE_OUT, M_SAVE, OcrFont);
    printf("Read successful, calibrated OCR font was saved to disk.\n");
}
else
{
    printf("Error: Read score too low, calibrated OCR font not saved.\n");
}
printf("Press <Enter> to end.\n");
getchar();

/* Free all allocations. */
MocrFree(OcrFont);
MocrFree(OcrResult);
MbufFree(MilSubImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

Using fonts

To read or verify character strings in images, the OCR module must know about the font type and dimensions of the target string. The module uses fonts (or typesets) to specify the style, size, and spacing of characters in the images to be read or verified.

The module provides two predefined font files, *semi1292.mfo* and *semi1388.mfo*, for analyzing SEMI font character strings in a target image. If your application requires font types other than the standard SEMI fonts, you can create custom font files (see *Creating custom fonts* at the end of this chapter).

A font file contains information such as:

- The grayscale representations of the characters.
- Codes identifying each character (usually the ASCII codes for the characters).
- The number of characters in the font file.
- Character dimensions.
- The maximum number of characters in the target image string to read or verify.
- Control parameters, such as target image character size and spacing, and character constraints.

The above information can be modified and saved to meet specific application requirements.

Calibrating fonts

Before character strings can be read or verified in target images, fonts must be calibrated to the size and spacing of characters in the target images. The *MocrCalibrateFont()* function uses a sample reference image to obtain these values. The sample image must be representative of all target images to be analyzed: the sample image's characters must be equal in type, size, and spacing to those of the target images. The character string in the sample image, as well as in target images, must be of known length, aligned, uniformly spaced, and cannot be rotated more than 10 degrees.

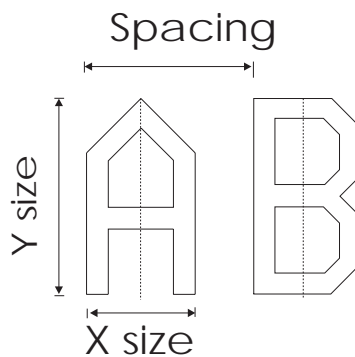
Generally, a font is calibrated once, at the beginning of an application. If your target image characters' size and/or spacing change, you should calibrate the font again. If the font is not calibrated, *MocrReadString()* or *MocrVerifyString()* might not be able to find the string in the target image because of the difference in size between the font characters and the target image characters.

Calibration values can be stored in the font file as part of the font's information set. You can use *MocrSaveFont()* to save the calibration values with the font or *MocrModifyFont()* to actually modify the font's character representation to match the characters in the sample target image.

*Target character
dimensions*

If *MocrCalibrateFont()* function is inappropriate, you can use the *MocrControl()* function to manually set the width (M_TARGET_CHAR_SIZE_X), the height (M_TARGET_CHAR_SIZE_Y), and spacing (M_TARGET_CHAR_SPACING) of target image characters with sub-pixel accuracy.

These values must be very precise so as not to affect reading performance and reliability. The following diagram illustrates spacing and size of sample characters.



Setting character constraints

The read operation compares each character in the target image to each character in the font to find the best match. You might know beforehand that certain characters (or types of characters) should appear at specific positions in the string. If this is the case, you can speed up and increase the robustness of the read operation by restricting the comparison to only those characters in the font. The following types of constraints can be set using *MocrSetConstraint()*:

- One or many digits (M_DIGIT): ASCII codes 48 to 57.
- One or many letters (M_LETTER): ASCII codes 65 to 90 and 97 to 122.
- One or many uppercase letters (M_LETTER+M_UPPERCASE): ASCII characters 65 to 90.
- One or many lowercase letters (M_LETTER+M_LOWERCASE): ASCII characters 97 to 122.
- Specific list of mixed character types (for example, A,1,b,2), including special characters and punctuations (for example, &, -, ...).
- Default (all characters in the font).

The constraints are stored with the font as part of its information set and can be inquired, using *MocrInquire()*.

The following is a portion of the *mocrfont.c* example found at the end of this chapter. It demonstrates how character constraints are set. For example, the character in the first position should be the letter K, the character in the second position should be any upper or lowercase letter.

```
/* Set character constraints for each position of the string to read. */
MocrSetConstraint(OcrFont, 0, M_LETTER, "K"); /* Must be K. */
MocrSetConstraint(OcrFont, 1, M_LETTER, M_NULL); /* Any letter. */
MocrSetConstraint(OcrFont, 2, M_DIGIT, M_NULL); /* Any digit. */
MocrSetConstraint(OcrFont, 3, M_DIGIT, "12"); /* Must be 1 or 2. */
MocrSetConstraint(OcrFont, 4, M_DIGIT, M_NULL); /* Any digit. */
MocrSetConstraint(OcrFont, 5, M_DEFAULT, M_NULL); /* Any character. */
```

Setting processing controls

Before reading or verifying an image, you should ensure that the processing controls associated with the font are appropriate for your application using *MocrInquire()*. The following processing controls are associated with a font and their values can be changed using *MocrControl()*:

- Length of target string.
- Target character dimensions.
- Acceptance levels.
- Symbol for unrecognized characters.
- Characters to erase from the font.
- Contrast enhancement and string location.
- Robustness of the read algorithm.

Acceptance levels

You can set the acceptance level of a successful read/verify operation for an entire string or for each of its characters.

- Setting levels for each character (M_CHAR_ACCEPTANCE).

If the correspondence (also known as the match score) between a character in an image and a character in a specified font is less than the specified acceptance level, that character is considered invalid and the associated validity flag for that character is set to false. A perfect match is 100%, a typical match is 60%, and no correlation is 0%. If your images have a lot of noise or distortion, you might have to set a low acceptance level. However, keep in mind that poor-quality images increase the chance of false readings and will probably increase the reading time.

Note also that perfect matches are generally unobtainable because of noise obtained during image acquisition.

- Setting the acceptance level for the entire string of characters (M_STRING_ACCEPTANCE).

The match score for the entire string is determined by taking the average of the match scores of all characters in that string. If this average passes the acceptance level set for the string, the string is considered valid and its validity flag is set to true.

Unrecognized characters

You can specify the symbol for unrecognized (or invalid) characters (`M_CHAR_INVALID`). If a target image character's match score does not reach the specified acceptance level, you can force a specified symbol to be returned at that position in the string. If no symbol is specified (default), the character with the closest match will be returned. For example:

Target string: "HELLO"

Invalid character: "*"

Acceptance threshold: 30% 30% 30% 30% 30%

Confidence scores: 70% 15% 53% 24% 80%

Result: H*L*O

Since the confidence scores of the characters in the second and fourth positions are less than the specified acceptance level, these characters are replaced by asterisks in the result.

Characters to erase from font

If you no longer require certain character representations in your font file, they can be deleted from the font, using the `M_CHAR_ERASE` option.

Image enhancement and string location

By default, the read and verify operations first clean the target image (filter and perform contrast enhancement) and then locate the target string. You can skip the image enhancement step if your images don't have too much noise and have good contrast. You can skip the location step if you have created a child buffer which surrounds the string very closely, without touching it. By skipping these steps, you can save a significant amount of time.

Robustness of the read algorithm

You can set the speed and reliability of the algorithm by setting the robustness factor. For instance, noisy images might require a high level of reliability. The robustness factor can be adjusted such that the algorithm is slower but more reliable.

Managing fonts

You can save, restore, modify, and inquire about fonts, as well as display their grayscale representations.

Saving and restoring a font

You can use *MocrSaveFont()* to save a font to disk. You can save all of the font's associated data or only its associated processing controls or character constraints. You can restore a previously saved font, using *MocrRestoreFont()*. This function restores all data associated with the font or restores only processing controls or character constraints.

Inverting a font

The *MocrModifyFont()* function can be used to invert the grayscale representations of the font characters to match that of the target image characters.

Inquiring about a font

You can inquire about character constraints, processing controls, and other information associated with a font using the *MocrInquire()* function.

Visualizing a font

It might be necessary at some point during application development to display the grayscale character representations of your font. To do so, use *MocrCopyFont()* to copy the grayscale representations to a displayable image buffer.

For example...

The following example shows you how to inquire about a font and how to display its grayscale character representations.

```

/* File name: mocrview.c
 * Synopsis: This program shows how to visualize the characters of a font.
 */

#include <stdio.h>
#include <mil.h>

/* Target font file name. */
#define FONT_FILE_NAME "semi1292.mfo"

void main(void)
{
    MIL_ID MilApplication,          /* Application identifier.          */
    MilSystem,                     /* System identifier.              */
    MilDisplay,                    /* Display identifier.             */
    MilImage,                      /* Image buffer identifier.        */
    OcrFont;                       /* OCR font identifier.           */
    double CharNumber,             /* Number of characters in font    */
    CharBoxSizeX, CharBoxSizeY;    /* Size of character's box in font */

    /* Allocate defaults */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                     M_NULL, &MilImage);

    /* Restore the OCR character font from disk. */
    MocrRestoreFont(FONT_FILE_NAME, M_RESTORE, MilSystem, &OcrFont);

    /* Inquire the OCR font character number and dimensions. */
    MocrInquire(OcrFont, M_CHAR_NUMBER, &CharNumber);
    MocrInquire(OcrFont, M_CHAR_BOX_SIZE_X, &CharBoxSizeX);
    MocrInquire(OcrFont, M_CHAR_BOX_SIZE_Y, &CharBoxSizeY);

    /* Verify that all the character representations fits in the target image */
    if (CharNumber <
        ((MbufInquire(MilImage, M_SIZE_X, M_NULL) / CharBoxSizeX) *
         (MbufInquire(MilImage, M_SIZE_Y, M_NULL) / CharBoxSizeY))
    )
    {
        /* Display the font representation. */
        MocrCopyFont(MilImage, OcrFont, M_COPY_FROM_FONT+M_ALL_CHAR, M_NULL);

        /* Pause to show the result. */
        printf("The font characters have been copied to the displayed image.\n");
        printf("Press <Enter> to end.\n");
        getchar();
    }
    else
    {
        printf("Error: Target image too small to copy the font characters.\n");
    }

    /* Free all allocations. */
    MocrFree(OcrFont);
    MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

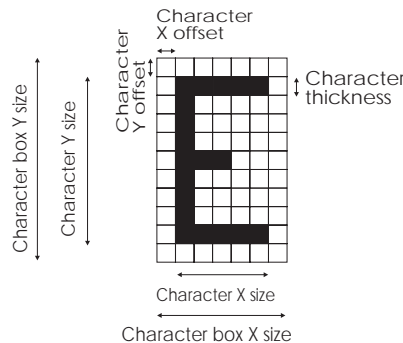
Creating custom fonts

While the standard SEMI font is frequently used for wafer analysis in the semiconductor industry, other applications requiring different font types can be addressed by creating custom fonts.

To create a custom font:

1. Allocate a font buffer, using *MocrAllocFont()*.
2. Grab or create the grayscale character representations of the font in a MIL image buffer and then copy them from the image buffer to a MIL font buffer, using *MocrCopyFont()*. Alternatively, you can import grayscale character representations from a text file or an image file (in either a TIFF or MIL format) into a MIL font buffer using *MocrImportFont()*.

Note that the font buffer must have been previously allocated using *MocrAllocFont()*. When allocating a font buffer, you must specify the dimensions of its character representations and their character boxes. The following is an example of a character in its character box and the dimensions that you will have to specify. Values are to be specified in pixels. Each square in the grid represents one pixel.



Once copied or imported into a font buffer, the font information can be saved on disk using *MocrSaveFont()*, and then later restored using *MocrRestoreFont()*.

For example...

The following example demonstrates how a custom font is created to verify lot numbers printed on medicine bottle labels. Character constraints have been set for each of the six positions in the string to be read. To simplify the example, the font's character representations have been drawn using the MIL graphics module instead of being grabbed.



```

/* File name: mocrfont.c
 * Synopsis: This program create a custom OCR font, set its constraints
 *           and uses it to read a string drawn in the image. The string
 *           read is then printed to the screen and the calibrated font
 *           is saved to disk.
 */

#include <stdio.h>
#include <mil.h>

/* Typical reading specifications. */
#define STRING_TO_READ      "KF419N"
#define STRING_SCORE_MIN    50.0
#define STRING_SCALE        2.0

/* Font specifications. */
#define FONT_CHAR_LIST      "FKLMN0123456789"
#define FONT_CHAR_NUM        15L
#define FONT_CHAR_BOX_SIZE_X 16L
#define FONT_CHAR_BOX_SIZE_Y 32L
#define FONT_CHAR_OFFSET_X   1L
#define FONT_CHAR_OFFSET_Y   6L
#define FONT_CHAR_SIZE_X     14L
#define FONT_CHAR_SIZE_Y     18L
#define FONT_CHAR_THICKNESS   4L
#define FONT_NUM_CHAR_TO_READ 6L
#define FONT_CHAR_FOREGROUND M_FOREGROUND_WHITE
#define FONT_FILE_NAME       "custom.mfo"

(cont. ...)

```

```

/* Work region specifications. */
#define WORK_REGION_POS_X 20L
#define WORK_REGION_POS_Y 80L
#define WORK_REGION_WIDTH
(long)((FONT_CHAR_BOX_SIZE_X*FONT_CHAR_NUM+1) \
    *STRING_SCALE)

#define WORK_REGION_HEIGHT
(long)((FONT_CHAR_BOX_SIZE_Y*2)*STRING_SCALE)

/* Image background color. */
#define BACKGROUND_COLOR 128

void main(void)
{
    MIL_ID MilApplication,          /* Application identifier. */
    MilSystem,                     /* System identifier. */
    MilDisplay,                    /* Display identifier. */
    MilImage,                      /* Image buffer identifier. */
    MilSubImage,                   /* Sub-image buffer identifier. */
    OcrFont,                       /* OCR font identifier. */
    OcrResult;                     /* OCR result buffer identifier. */
    double Score;                  /* Reading score. */
    char String[FONT_NUM_CHAR_TO_READ+1]; /* Array for the read characters. */

    /* Allocate defaults */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
        M_NULL, &MilImage);

    /* Restrict the source image to the work region. */
    MbufChild2d(MilImage, WORK_REGION_POS_X, WORK_REGION_POS_Y, WORK_REGION_WIDTH,
        WORK_REGION_HEIGHT, &MilSubImage);

    /* Draw a representation of all the characters of the new font to create. */
    MbufClear(MilSubImage, 0);
    MgraFont(M_DEFAULT, M_FONT_DEFAULT_LARGE);
    MgraText(M_DEFAULT, MilSubImage, 1, 0, FONT_CHAR_LIST);

    /* Pause to show the characters of the font. */
    printf("A custom OCR font will be created from the characters drawn\n");
    printf("in the displayed image.\nPress <Enter> to continue.\n");
    getchar();

    /* Allocate a new empty OCR font. */
    MocrAllocFont(MilSystem, M_DEFAULT, FONT_CHAR_NUM,
        FONT_CHAR_BOX_SIZE_X, FONT_CHAR_BOX_SIZE_Y,
        FONT_CHAR_OFFSET_X, FONT_CHAR_OFFSET_Y,
        FONT_CHAR_SIZE_X, FONT_CHAR_SIZE_Y,
        FONT_CHAR_THICKNESS, FONT_NUM_CHAR_TO_READ,
        FONT_CHAR_FOREGROUND, &OcrFont);

    /* Copy the character representation to the font. */
    MocrCopyFont(MilSubImage, OcrFont, M_COPY_TO_FONT, FONT_CHAR_LIST);

    (cont. ...)

```

```

/* Set character constraints for each position of the string to read. */
MocrSetConstraint(OcrFont, 0, M_LETTER, "K"); /* Must be K. */
MocrSetConstraint(OcrFont, 1, M_LETTER, M_NULL); /* Any letter. */
MocrSetConstraint(OcrFont, 2, M_DIGIT, M_NULL); /* Any digit. */
MocrSetConstraint(OcrFont, 3, M_DIGIT, "12"); /* Must be 1 or 2 */
MocrSetConstraint(OcrFont, 4, M_DIGIT, M_NULL); /* Any digit. */
MocrSetConstraint(OcrFont, 5, M_DEFAULT, M_NULL); /* Any character */

/* Set the target image character scale for the font manually. */
MocrControl(OcrFont, M_TARGET_CHAR_SIZE_X, FONT_CHAR_SIZE_X*STRING_SCALE);
MocrControl(OcrFont, M_TARGET_CHAR_SIZE_Y, FONT_CHAR_SIZE_Y*STRING_SCALE);
MocrControl(OcrFont, M_TARGET_CHAR_SPACING, FONT_CHAR_BOX_SIZE_X*STRING_SCALE);

/* Draw a typical string respecting the font and its constraints,
 * but at a bigger scale.
 */
MbufClear(MilSubImage, 0);
MgraFontScale(M_DEFAULT, STRING_SCALE, STRING_SCALE);
MgraText(M_DEFAULT, MilSubImage, 0, 0, STRING_TO_READ);

/* Pause to show the string to read. */
printf("\nA typical string with a bigger scale will be read using\n");
printf("the new custom font and the result will be printed.\n");
printf("Press <Enter> to continue.\n");
getchar();

/* Allocate an OCR result buffer. */
MocrAllocResult(MilSystem, M_DEFAULT, &OcrResult);

/* Read the string. */
MocrReadString(MilImage, OcrFont, OcrResult);
/* Get the string and its reading score. */
MocrGetResult(OcrResult, M_STRING, String);
MocrGetResult(OcrResult, M_SCORE, &Score);

/* Print the result. */
printf("The string read is: \"%s\" (score: %.1f%%).\n\n", String, Score);

/* Save the custom font if the reading score was sufficient. */
if (Score > STRING_SCORE_MIN)
{
    MocrSaveFont(FONT_FILE_NAME, M_SAVE, OcrFont);
    printf("Read successful, calibrated OCR font was saved.\n");
}
else
{
    printf("Error: Read score too low, custom OCR font not saved.\n");
}
printf("Press <Enter> to end.\n");
getchar();

/* Free all allocations. */
MocrFree(OcrFont);
MocrFree(OcrResult);
MbufFree(MilSubImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

Speeding up the read or verification operation

To ensure the fastest possible read or verify operation:

- Reduce the area in the target image to be read or verified by creating a child buffer around the target string using *MbufChild...()*; the search time is roughly proportional to the area searched.
- Set appropriate character constraints using *MocrSetConstraint()*. You can speed up the process by limiting the number of character representations to be compared.
- Set the processing controls (using *MocrControl()*) to skip the contrast enhancement and/or the string location step.
- Adjust the robustness factor of the read algorithm (using *MocrControl()*) according to the quality of your image.

Chapter 14: DataMatrix and bar codes

This chapter describes how to read and write 2D and bar codes.

Introduction

MIL allows you to read and write 2-D codes, such as DataMatrix and PDF417, as well as several types of bar codes.



A DataMatrix code (left) and two types of bar codes

A read operation searches for a specified type of code in an image and decodes it. The decoded string can then be used to identify the object in the image.

A write operation encodes a null-terminated string into an image using the specified type of coding scheme. The resulting image can then be scaled or rotated, if necessary, and then printed on an object (using a hardware printing device) in order to label the object.

More information

For technical information about DataMatrix, PDF417, or bar codes, see the *AIM International Symbology Specification - DataMatrix*, *AIM International Symbology Specification - PDF417*, or *The Bar Code Book*, Roger C. Palmer, Helmers Publishing, United States, 1995.

General steps

To perform a read or write operation:

1. Allocate a code object, using *McodeAlloc()*. A code object specifies the type of code to read or write and how to perform the operation.
2. If necessary, change control settings of the code object, using *McodeControl()*.
3. To perform a read operation, use *McodeRead()*. To perform a write operation, use *McodeWrite()*.
4. Retrieve results, using *McodeGetResult()*.
5. Free the memory allocated to the code object, using *McodeFree()*.

Controlling read operations

You can set the following controls of a read operation using *McodeControl()*:

- The type of encoding scheme (M_ENCODING) and the type of error correction (M_ERROR_CORRECTION).
- The cell size of the code (M_CELL_SIZE_MIN and M_CELL_SIZE_MAX), as well as the number of cells in the X or Y direction of a 2-D code (M_CELL_NUMBER_X or M_CELL_NUMBER_Y).
- The color (black or white) of the code (M_FOREGROUND_VALUE).
- The search angle (M_SEARCH_ANGLE, M_SEARCH_ANGLE_DELTA_NEG, and M_SEARCH_ANGLE_DELTA_POS).
- The search speed (M_SPEED).
- The threshold value (M_THRESHOLD).
- The size of the string to search (M_STRING_SIZE).

In general, except for the encoding scheme and error correction, you should only change a control if you are having problems finding the specified code or if the operation is too slow for your application. There are some code types however, for which it is essential to explicitly set their controls.

Encoding scheme and error correction

For each supported code, you must specify the encoding scheme (M_ENCODING) and error correction method (M_ERROR_CORRECTION). There are several encoding schemes and error correction types available. For a full list, see the description of *McodeControl()* in the *MIL Command Reference*.

Note that error correction allows MIL to detect and correct errors.

Cell size

The cell size is the size, in pixels, of a unit of code. For the 2D codes, it is the width, in pixels, used to encode one bit of data. For a bar code, it is the pixel width of the smallest bar of the code. During a read operation, the cell size of the code must be within the range defined by M_CELL_SIZE_MIN and M_CELL_SIZE_MAX. If it is not, the code will not be found.

Number of cells in x and y

By default, the operation searches for 2D code with any number of cells in the X and Y direction. For the PDF417 code, you must specify the number of cells in the X and Y direction (M_CELL_NUMBER_X and M_CELL_NUMBER_Y). If you know the number of cells in the X and/or Y direction of the DataMatrix code, you can specify this, to increase the speed of the operation.

String size

By default, the operation searches for a string of any size. However, if you know the exact size of the string, you might want to specify this (M_STRING_SIZE), to increase the robustness of the operation. Note, for the BC412 code, you must specify this parameter.

Search angle

In a read operation, the specified code is sought for within the range of angles defined by (M_SEARCH_ANGLE - M_SEARCH_ANGLE_DELTA_NEG) to (M_SEARCH_ANGLE + M_SEARCH_ANGLE_DELTA_POS). By default, the search is performed at $0 \pm 5^\circ$. You should change the search angle if the code appears rotated in the image. You

should increase the angular range if you are unsure of the code's exact orientation. Note, however, that when searching for a bar code, the operation speed might decrease as you increase the range of angles. For the DataMatrix code, the angular range does not affect the speed of the operation.

Search speed

A read operation can be performed at several speeds (M_SPEED+M_VERY_LOW, M_LOW, M_MEDIUM, M_HIGH, or M_VERY_HIGH). The faster the search speed, the less robust the operation. In general, the larger and more clearly defined the code, the better chance it has of being found at a speed higher than the default. If you are having problems finding the code, you might want to search at a speed lower than the default.

Threshold value

In a read operation, the source image is internally binarized so as to separate the code from the background. By default, the threshold value is chosen automatically from the source image's histogram (the two highest peaks in the histogram are located and the threshold value is set to the minimum value between these peaks). The automatically chosen threshold value is suitable in most cases. However, if you feel that a different value would result in a better separation (and therefore in a more efficient operation), you can specify this value (M_THRESHOLD).

Note that, if the background is both darker and lighter than the code, a simple binarization will not separate the code from the background. In this case, you should process the image before performing the read operation so that the background is either darker or lighter than the code.

Controlling write operations

You can set the following controls of a write operation using *McodeControl()*:

- The type of encoding scheme (M_ENCODING) and the type of error correction (M_ERROR_CORRECTION). Note that error correction allows a read operation to determine whether the encoding produced any errors. For a full list of supported encoding schemes and error correction types, see the description of *McodeControl()* in the *MIL Command Reference*.
- The cell size of the code (M_CELL_SIZE_MIN). The cell size is the size, in pixels, of a unit of code. For 2D codes, it is the width, in pixels, used to encode one bit of data. For a bar code, it is the pixel width of the smallest bar of the code. Instead of specifying a cell size, M_CELL_SIZE_MIN can be set to M_DEFAULT, in which case the code is resized so as to just fit into the destination image of the operation.
- The number of cells (M_CELL_NUMBER_X and M_CELL_NUMBER_Y). If set to M_ANY, the minimum number of cells possible will be used to perform the write operation.
- The color (black or white) in which to write the code (M_FOREGROUND_VALUE).

Size of destination image

The destination image of the write operation should be large enough to hold the encoded string. For a given code object and string, you can inquire about the minimum buffer size required by first calling *McodeWrite()* with its image buffer parameter set to M_NULL and then using the M_WRITE_SIZE_X and M_WRITE_SIZE_Y result types of *McodeGetResult()*.

Chapter 15: Measurements

This chapter describes the MIL measurement module and the steps to follow to take measurements.

The measurement module

The MIL measurement module allows you to find sets of image characteristics or "markers" in an image, based on differences in pixel intensities. Upon finding a marker, the module returns the marker's spatial reference position and measures such features as its width and angle. The module can also take measurements between two markers.

The measurement module can be used, for example, to measure the width of pins protruding from chips or printed circuit boards (PCBs) and to measure the distance between each pin.

The measurement module relies on a one-dimensional analysis. As such, it has several advantages over the pattern matching module when locating relatively simple image characteristics; it is independent of lighting, more tolerant of slight differences, and much faster.

You specify the approximate location and other characteristics of a marker to help locate the marker in an image. The more precisely the marker characteristics are defined, the more likely it is to be distinguished from similar aspects of the image.

The measurement module can operate on 8-bit or 16-bit unsigned grayscale buffers. Measurements are made with sub-pixel accuracy and results can be returned in pixels or real-world units (*see Chapter 7: Calibration*).

This chapter discusses finding and obtaining measurements of markers, how to define and set marker characteristics, and then the steps that are generally followed when taking measurements between markers.

Markers

To take any type of measurement with the MIL measurement module, you must first define your markers, using *MmeasSetMarker()*. The marker contains the image characteristics to search for in the target image.

There are three types of markers that can be used in measurement operations:

- **Point marker.** A marker consisting of a single point or multiple points and generally used as a reference position in calculations involving two markers. Point markers cannot be searched for but can be placed manually at the required location as a reference marker. Positional results from a previous pattern matching or blob analysis operation on the image can also be declared as point markers.
- **Edge marker.** A marker consisting of an edge or multiple edges. Edges are sharp changes between two or more adjacent pixels.
- **Stripe marker.** A marker consisting of a stripe (two edges) or multiple stripes. Stripe marker edges do not have to be parallel.



A multiple marker

The measurement module allows you to define a multiple edge, stripe, or point marker, so that you can search for multiple instances of the same image characteristics. Using a multiple marker in a measurement operation makes it possible to take global measurements, then compare them for conformity. For example, a multiple marker could be used to verify if a series of presumed-identical pins are actually of equal width or if the

spacing between the pins is within established limits. Note that a multiple marker is considered to be only one marker that has a specified number of instances of the same characteristics.

Steps to finding and obtaining measurements of markers

Although there are many types of measurements that can be taken with the MIL measurement module, the series of steps outlined below are usually followed to find and obtain measurements of markers:

1. Allocate an edge or stripe marker. For a multiple marker, set the number of occurrences of the edge or stripe.
2. Set the processing area, generally referred to as the measurement box.
3. Set the marker's characteristics.
4. Optionally, set the measurement control settings.
5. Grab or load a target image. Optionally, preprocess it to improve its quality.
6. Find the marker in the target image and calculate measurements.
7. Read the results.

In general, the first four steps are performed once, while steps 5 to 7 are repeated as required. Note, you can avoid step 1 to 4 by saving an initialized marker on disk and restoring it when needed.

Allocating or restoring a marker

Allocate a new marker, using *MmeasAllocMarker()* or restore a previously saved marker from disk, using *MmeasRestoreMarker()*. You can allocate an edge, stripe, or point marker, depending on your application needs; if a multiple marker is needed, specify the number of occurrences of the edge or stripe. When a marker is no longer required, you should free the memory associated with it, using *MmeasFree()*. Store a marker to disk using *MmeasSaveMarker()*. The save

command stores all of the marker's associated parameter settings, such as the marker's typical position, width, and contrast.

Setting the marker's measurement box

Before you can search for a marker, you must define the area in which to perform the search. This area is known as the measurement box. The measurement box is stored as a characteristic of the marker and is set with *MmeasSetMarker()*. Subsequent search operations for this marker will only be performed in the area defined by the box.

Setting the marker's characteristics and processing area

The marker's characteristics, such as its approximate width, orientation, and position, must also be defined using *MmeasSetMarker()*.

You can inquire about the current settings of a marker's characteristics, using *MmeasInquire()*.

Specifying the measurement control settings

When performing an *MmeasFindMarker()* operation, you must specify a measurement context, either the default one or one allocated using *MmeasAllocContext()*. Measurement context settings, such as the pixel aspect ratio, control the behavior of measurement operations. You can modify these settings using *MmeasControl()*.

When a measurement context is no longer required, it should be freed, using *MmeasFree()*.

Acquiring and pre-processing a target image

The target image can be either loaded from disk or acquired from an input device and placed into an image buffer. You can preprocess the target image to remove noise and improve the image prior to performing the measurement operation. Note, however, that preprocessing operations such as filtering often result in a slight shifting of the location position of edges.

Therefore, if precise edge location and measurement are crucial to your application, preprocessing operations should be kept to a minimum.

Finding the marker and taking measurements

Use *MmeasFindMarker()* to find a marker. The *MmeasFindMarker()* function can measure all calculable edge and stripe characteristics, such as the width, orientation, or contrast. The function's parameters allow you to specify which measurements to take (the default setting performs all measurements).

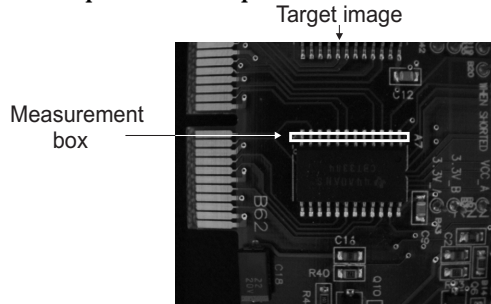
Reading results

You can obtain results using *MmeasGetResult()*. Results from a *MmeasFindMarker()* operation are stored directly with the marker rather than in a result buffer. All positional results are relative to the top-left pixel in the target image or the origin of the coordinate system of a calibrated image (recall that the pixel reference position is its center). Note that results do not overwrite specified marker characteristics.

For a multiple marker, the *MmeasFindMarker()* function takes the required measurements for all the located edges or stripes. The number of edges or stripes found can be obtained using `M_NUMBER` as the result type with the *MmeasGetResult()* function. Global results, such as the maximum, minimum, mean, or standard deviation of any characteristic of the result group can be returned.

A measurement example

The following example demonstrates how to use the measurement module to find the positions, widths, and angles of the pins on a chip.



```

/* File name: MmeasMul.c
 * Synopsis: This program measures the positions, widths and angles of
 *           the pins of a chip.
 */

/* Regular includes. */
#include <stdio.h>
#include <mil.h>
#include <math.h>

/* Source MIL image file specification. */
#define IMAGE_FILE           M_IMAGE_PATH "chip.bmp"

/* Processing region specification */
#define MEAS_BOX_WIDTH      230
#define MEAS_BOX_HEIGHT    10
#define MEAS_BOX_POS_X     220
#define MEAS_BOX_POS_Y     167

/* Target stripe specifications. */
#define STRIPE_ORIENTATION  M_VERTICAL
#define STRIPE_POLARITY_LEFT M_POSITIVE
#define STRIPE_POLARITY_RIGHT M_NEGATIVE
#define STRIPE_NUMBER       12

/* Size and color of the cross and the circle. */
#define CROSS_SIZE          15L
#define CROSS_COLOR         240L
#define CIRCLE_SIZE         15L

/* Utility functions prototypes */
void DrawCross(MIL_ID ImageId, double CenterX, double CenterY, long Color);

/* Main application function */
void main(void)

(cont...)

```

```

{
    MIL_ID MilApplication,          /* Application identifier, */
    MilSystem,                     /* System identifier,      */
    MilDisplay,                    /* Display identifier,     */
    MilImage,                      /* Image buffer identifier, */
    StripeMarker;                  /* Stripe marker identifier, */
    double StripeCenterX[STRIPE_NUMBER], /* Stripe X center positions, */
    StripeCenterY[STRIPE_NUMBER], /* Stripe Y center positions, */
    MeanAngle,                     /* Stripe mean angle,      */
    MeanWidth,                     /* Stripe mean width,      */
    MeanSpacing;                   /* Stripe mean spacing,    */
    long NumberFound;              /* Number of stripes found, */
    long i;                        /* Index.                  */

    /* Allocate defaults */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem,
                    &MilDisplay, M_NULL, M_NULL);

    /* Restore source image into an automatically allocated image buffer. */
    MbufRestore(IMAGE_FILE, MilSystem, &MilImage);

    /* Display the image buffer. */
    MdispSelect(MilDisplay, MilImage);

    /* Draw the contour of the measurement box */
    MgraRect(M_DEFAULT,
            MilImage,
            MEAS_BOX_POS_X-1, MEAS_BOX_POS_Y-1,
            MEAS_BOX_POS_X+MEAS_BOX_WIDTH+1, MEAS_BOX_POS_Y+MEAS_BOX_HEIGHT+1);

    /* Pause to show the original image. */
    printf("This program will determine the positions of each pin of the chip.\n");
    printf("Press <Enter> to continue.\n");
    getchar();

    /* Read the source image again to remove previously drawn rectangle */
    MbufLoad(IMAGE_FILE, MilImage);

    /* Allocate a stripe marker */
    MmeasAllocMarker(M_DEFAULT, M_STRIPE, M_DEFAULT, &StripeMarker);

    /* Stripe specifications */
    MmeasSetMarker(StripeMarker, M_NUMBER,
                    STRIPE_NUMBER, M_NULL);
    MmeasSetMarker(StripeMarker, M_POLARITY,
                    STRIPE_POLARITY_LEFT, STRIPE_POLARITY_RIGHT);
    MmeasSetMarker(StripeMarker, M_ORIENTATION,
                    STRIPE_ORIENTATION, M_NULL);

    /* Specify the search box size. */
    MmeasSetMarker(StripeMarker, M_BOX_ORIGIN,
                    MEAS_BOX_POS_X, MEAS_BOX_POS_Y);
    MmeasSetMarker(StripeMarker, M_BOX_SIZE,
                    MEAS_BOX_WIDTH, MEAS_BOX_HEIGHT);

    (cont...)

```

```

/* Find the stripe and measure its width and angle. */
MmeasFindMarker(M_DEFAULT, MilImage, StripeMarker, M_POSITION + M_ANGLE +
M_WIDTH);

/* Get the number of stripes found*/
MmeasGetResult(StripeMarker, M_NUMBER + M_TYPE_LONG, &NumberFound, M_NULL);

/* Get the stripe position, width and angle. */
MmeasGetResult(StripeMarker, M_POSITION, StripeCenterX,
StripeCenterY);
MmeasGetResult(StripeMarker, M_ANGLE + M_MEAN, &MeanAngle, M_NULL);
MmeasGetResult(StripeMarker, M_WIDTH + M_MEAN, &MeanWidth, M_NULL);
MmeasGetResult(StripeMarker, M_SPACING + M_MEAN, &MeanSpacing, M_NULL);

/* Draw a cross on the center of each stripe found. */
for(i=0; i<NumberFound; i++)
{
    DrawCross(MilImage, StripeCenterX[i], StripeCenterY[i], CROSS_COLOR);
}

/* Draw the contour of the measurement box */
MgraRect(M_DEFAULT,
MilImage,
MEAS_BOX_POS_X-1, MEAS_BOX_POS_Y-1,
MEAS_BOX_POS_X+MEAS_BOX_WIDTH+1, MEAS_BOX_POS_Y+MEAS_BOX_HEIGHT+1);

/* Print the results. */
printf("The center of each pin found have been marked.\n");
printf("The statistics of the pins are:\n");
printf("Average angle : %5.2f\n", MeanAngle);
printf("Average width : %5.2f\n", MeanWidth);
printf("Average spacing : %5.2f\n", MeanSpacing);
printf("Press <Enter> to end.\n");
getchar();

/* Free all allocations. */
MmeasFree(StripeMarker);
MbufFree(MilImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);
}

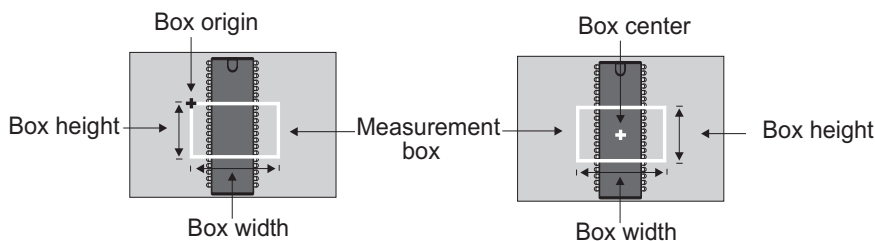
/* Draw a cross at the specified position. */
void DrawCross(MIL_ID ImageId, double CenterX, double CenterY, long Color)
{
    MgraColor(M_DEFAULT, Color);
    MgraLine(M_DEFAULT, ImageId,
(long)(CenterX+0.5)-(CROSS_SIZE/2), (long)(CenterY+0.5),
(long)(CenterX+0.5)+(CROSS_SIZE/2), (long)(CenterY+0.5));
    MgraLine(M_DEFAULT, ImageId,
(long)(CenterX+0.5), (long)(CenterY+0.5)-CROSS_SIZE,
(long)(CenterX+0.5), (long)(CenterY+0.5)+CROSS_SIZE);
}

```

Measurement box

The marker's measurement box indicates the area of the target image in which to search for the marker. Proper placement of the measurement box is essential to the success of any *MmeasFindMarker()* search. The default setting of the measurement box is the whole image. The measurement box should be limited to as small an area containing the marker as possible, in order to ensure the success of the operation, especially when using a highly detailed or complex target image. In addition, by limiting the processing region, you can accelerate the find marker operation.

There are two ways to specify the measurement box. To specify the box's position, you can set either the origin (M_BOX_ORIGIN) or the center (M_BOX_CENTER) coordinates. You must also set the box's width and height (M_BOX_SIZE).

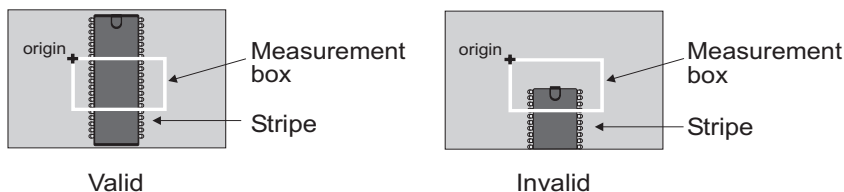


Note that the origin is always the top-left corner of the unrotated box and all width and height values are positive.

Obtaining valid results

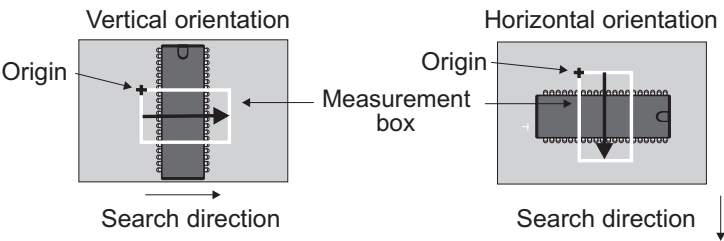
To obtain valid results the edge or stripe must enter and leave by opposite sides of the box. The illustration below is an example of valid and invalid measurement box definitions.

Stripe marker with vertical orientation



Orientation

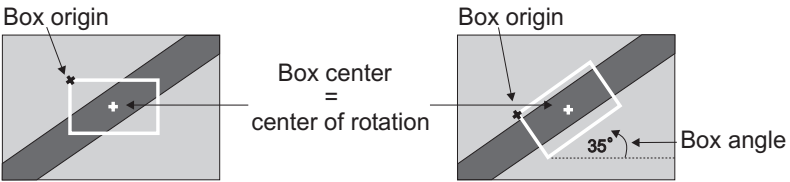
The orientation (M_ORIENTATION) specifies the angle of the edge or stripe in relation to that of the measurement box and can be set to either M_VERTICAL or M_HORIZONTAL. More importantly, the orientation determines the direction in which the search will proceed.



These settings can tolerate a certain amount of rotation. The amount is determined by the target image, placement of the measurement box, and the marker characteristic settings. The greater the degree of rotation, the greater the chance of not finding the marker and miscalculation of characteristics, such as edge strength and width. If the rotation is too great and you cannot find the marker, the marker's measurement box should be defined with an angle.

Setting the measurement box's angle

You can set the measurement box angle (M_BOX_ANGLE) to approximately the same angle as the marker. The angle is in a counter-clockwise direction relative to the positive X-axis and can be any value from 0 to 360 degrees. When an angle is specified, the center of rotation used is the center of the measurement box. To modify this default center of rotation, use *MmeasSetMarker* (... ,M_BOX_ANGLE_REFERENCE,...).



Multiple-angle search for a marker

You can also rotate the measurement box to search within a specified range of angles. To perform a multiple-angle search for a marker, enable multiple-angle search, using

MmeasSetMarker (... , M_BOX_ANGLE_MODE, ...). This function also includes control settings that allow you to specify the range of angles to be searched (M_BOX_ANGLE_DELTA_NEG and M_BOX_ANGLE_DELTA_POS), the degree of rotation tolerance at any given angle (M_BOX_ANGLE_TOLERANCE), the interpolation method (M_BOX_ANGLE_INTERPOLATION_MODE), and the required degree of accuracy for the resulting marker (M_BOX_ANGLE_ACCURACY).

The range of angles is searched in step angle increments determined by the specified rotation tolerance. The marker's rotation tolerance is the full range of degrees within which a marker can be rotated from a measurement box that is at a specific angle and still be found. Once the approximate location of the edge is found, the degree of accuracy controls the number of fine-tuned searches that are performed. To be effective, you must set the degree of accuracy to a value smaller than that of the rotation tolerance.

Determining the rotation tolerance of a marker

Every marker has its own particular rotation tolerance. This tolerance is dependent on the individual marker characteristics and surrounding image features. To determine the rotation tolerance of a marker, simulate the rotation of the marker by rotating the target image, so that you can determine by how much the marker can be offset from the measurement box and still be found. That is:

1. Make sure the measurement box is in the proper position and the marker is located within the measurement box at close to the required angle. Since the measurement box must remain in a set position, the positive and negative deltas must be zero (that is, you can set the angle of the box, but do not perform an angular search).
2. Use the *MimRotate()* function to rotate the image in very small increments (for example, 0.5 degrees), in the positive direction, and perform a *FindMarker()* operation at every angle. Make sure that the image's center of rotation is the same as that of the measurement box, otherwise the resulting tolerance will not be accurate. Note, when

rotating the image, always set the angle from the image's original position to avoid interpolating the image more than once.

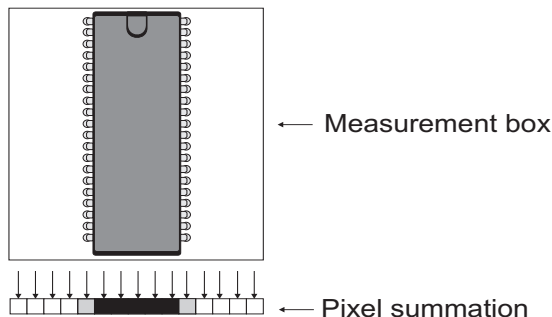
3. Check the results for the greatest angle that produces an acceptable score.
4. Repeat steps 1 through 3, rotating the image in the negative direction.
5. Take the minimum of the absolute value of these angles. Double this angle and set it as the rotation tolerance for the angular search.

Searching at any angle

Alternatively, you can set the measurement box angle to M_ANY to allow MIL to analyze the contents of the measurement box and determine the angle. However, it should be noted that the processing time will be greatly increased when using this technique.

Search algorithm

The MIL measurement module projects the two-dimensional measurement box into a one-dimensional line (that is, it takes the box's profile). The pixel summation is performed horizontally or vertically, depending on the measurement box's origin and the orientation of the marker. Each sum represents the intensity of all the pixels in that column.



To locate each edge, an edge filter is then applied to the profile. The edge filter first finds the edge value of each profile value. The edge value is the difference between one profile value and

the next. The greater the difference, the larger the edge value. The filter rejects as possible markers any edges with edge values below the edge threshold value.

The filter then finds the marker by scoring each possible edge based on geometric constraints that you specify, giving each characteristic a specific weight, or degree of importance. The edge(s) with the highest score is returned as the marker.

Marker characteristics

Associated with a marker is a set of parameters specifying the characteristics of the marker. The *MmeasFindMarker()* function searches through the target image to find the edge or stripe that best matches the characteristics (parameter settings) of the specified marker. The more precisely defined your marker characteristics, the more likely the find routine will have success in distinguishing it from similar aspects of the image.

Marker characteristics are set to their default values upon allocation of the marker (see the *MmeasAllocMarker()* command reference description for the default values). These values can be modified at any time, using *MmeasSetMarker()*.

The important characteristics to set when searching for a marker are the measurement box, the polarity, the contrast, and for stripe markers, the width. Fundamental and advanced characteristics are discussed for edge, stripe, and multiple markers in the following sections. Note, all characteristics can be set to *M_ANY* if the value is unknown or not a criteria, unless otherwise specified in the *MIL Command Reference*.

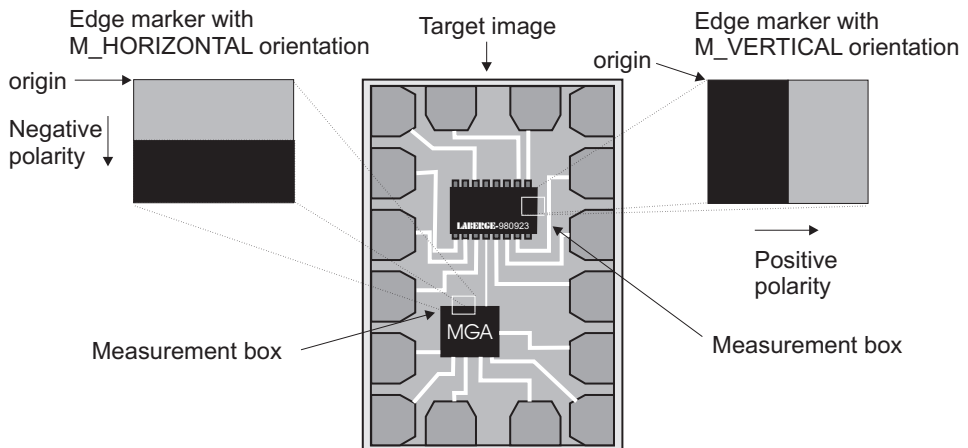
Edge markers: fundamental characteristics

This section describes the fundamental characteristics of edge markers that are set using *MmeasSetMarker()* or returned as measurement results.

Polarity

Polarity of an edge

The polarity (M_POLARITY) of an edge describes whether an edge is rising or falling. A rising edge denotes a rise in grayscale values and a positive (M_POSITIVE) polarity. A falling edge denotes a decrease in grayscale values and a negative (M_NEGATIVE) polarity. When setting the polarity of a marker it is important to keep in mind the direction of the search, which is performed horizontally or vertically, depending on the measurement box's origin and the orientation of the marker (see the *Measurement box* section).



Position and position variation

Position

The marker's position is defined as the X and Y coordinates of the marker's center (the center of the portion of the marker located within the measurement box). These coordinates are relative to the top-left pixel of the image and are used as the default reference position when using *MmeasCalculate()* to calculate measurements.

When several edges have similar characteristics within the same measurement box, then you can use the position characteristic to specify the approximate X and/or Y coordinates (M_POSITION, M_POSITION_X, M_POSITION_Y) at which to find the required marker's center.

You can also specify a tolerance for these coordinates (M_POSITION_VARIATION).

The position must be located within the measurement box (taking into account the measurement box's angle or angular range), otherwise an error is generated.

Contrast and contrast variation

Contrast

You can indicate the typical difference in grayscale values between an edge and its background (M_CONTRAST). Contrast is useful in distinguishing between several different edges, particularly when the required edge does not have the largest edge strength (described later), or when the edge is at an angle.

You can also specify a tolerance for the contrast (M_CONTRAST_VARIATION).

Length

Length

You can measure the length (M_LENGTH) of an edge marker. The length being measured is restricted to the portion of the marker contained within the measurement box. Note, the length of an edge marker cannot be set, but can be returned with *MmeasGetResult()*.

Line equation

Line equation

You can calculate the line equation (M_LINE_EQUATION) of the mean line following an edge marker: $Y = MX + B$, where M denotes the slope of the line and B denotes the Y-intercept.

Note, the line equation of an edge marker cannot be set, but can be returned with *MmeasGetResult()*. The slope of the line equation (M_LINE_EQUATION_SLOPE) and the Y intercept of the line equation (M_LINE_EQUATION_INTERCEPT) can also be returned separately.

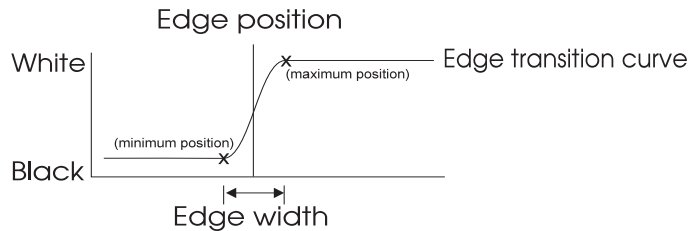
Edge markers: advanced characteristics

This section describes the advanced characteristics of edge markers that are set using *MmeasSetMarker()* or returned as measurement results.

Width

Width of an edge marker

Edges are usually gradual shifts in grayscale values over several pixels. The smoother the image, the more gradual the change. The width of an edge can be seen as a measure (in pixels) of this gradual shift in grayscale values. The diagram below illustrates a profile of an edge where the gradual transition from black to white can be seen.



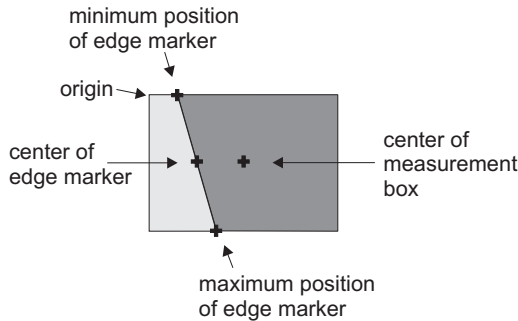
The measurement module calculates the marker's position to be in the middle of this width. The position variation is equivalent to half the edge's width. Note, the more an edge is at an angle the greater the stretching or distortion of its actual width.

The width of an edge marker cannot be set, but can be found with *MmeasFindMarker()*.

Minimum/maximum position

Minimum/maximum position

The minimum and maximum positions (M_POSITION_MIN and M_POSITION_MAX) indicate the start and end positions of the marker. The minimum position is always on the side of the measurement box adjacent to the origin, regardless of how the box is rotated.



The minimum and maximum positions of an edge cannot be set but can be found with *MmeasGetResult()*.

Edge strength

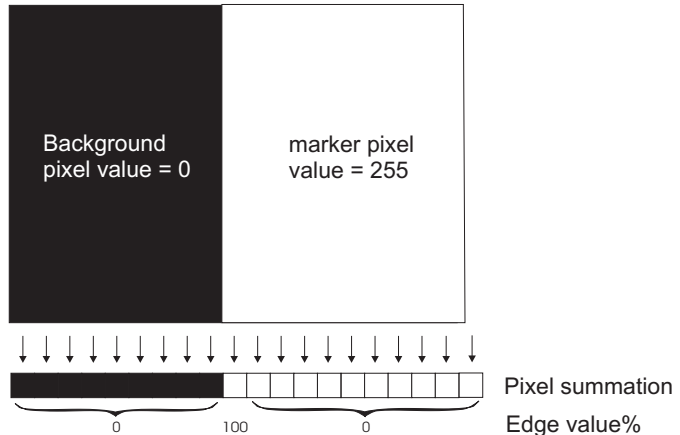
A marker's edge strength (M_EDGE_STRENGTH) is the minimum/maximum edge value along the width of the edge (depending on the polarity of the edge). The edge value is represented as a normalized percentage of the maximum pixel value possible for the specific image buffer. The sign of the edge value represents the polarity of the edge. For example, for a measurement box with a vertical orientation, the equation for an edge value is:

$$\text{edge value\%} = \frac{\Delta \text{ adjacent profile values}}{(\text{box height}) (2^{\text{buffer depth}} - 1)} * 100$$

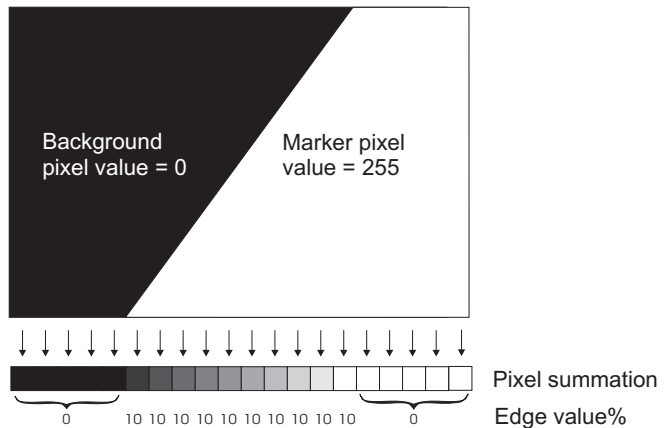
For instance, in an 8-bit image buffer the maximum possible pixel value is 255. Therefore, a 50% edge strength in this buffer represents a maximum difference in average adjacent profile values of 128 and a rising edge. The larger the absolute edge value, the greater the edge strength. The default setting finds the marker with the largest edge strength.

You can also specify a tolerance for the edge strength (M_EDGE_STRENGTH_VARIATION).

The edge in the diagram below has an edge strength of 100%, the maximum edge value possible since the edge is completely vertical and has an edge width of one pixel.



However, if the edge is at the following angle, the edge profiles are distributed over ten profiles, resulting in a much lower edge strength of 10%:



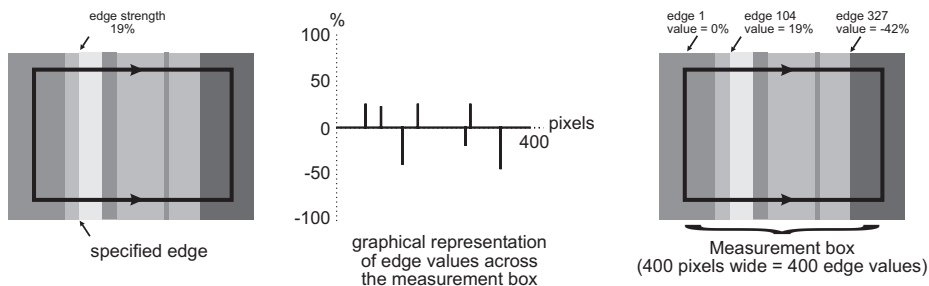
When the edge strength is not very high, you should specify the contrast. By using the contrast characteristic, changes in consecutive edge values are summed, from where the edge starts and ceases (that is, from one region of zero edge value to another), allowing the marker to be located. In general, it is recommended to set the contrast rather than to specify an edge strength, since the edge strength is very dependent on lighting.

Edge threshold

The edge threshold is the edge value beneath which a grayscale variation is not considered an edge and is set with `M_EDGE_THRESHOLD`.

Determining the strength of the required edge

To determine the edge strength of the required marker, use `M_BOX_EDGE_VALUES`. This calculates the edge values for every profile value of the measurement box. The illustration below shows a specific edge strength measurement, a graphical representation of box edge value measurement, and a sampling of these results.

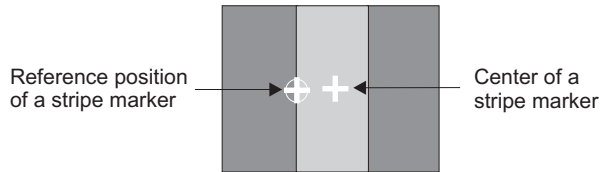


Notice that the measurement box is 400 pixels wide so there are 400 edge values that are returned.

Marker reference

The marker reference position (`M_MARKER_REFERENCE`) defines the position from which calculations between two markers are taken. By default, the reference position is set to the center position of the marker. You can, however, move the reference position by specifying x and y offsets relative to the center of the marker. The marker reference position is only used with the `MmeasCalculate()` function; `MmeasFindMarker()`

always returns the marker's actual center. For example, the reference position for the edge marker in the diagram below is set to the left of the marker's actual edge.



Weight factors

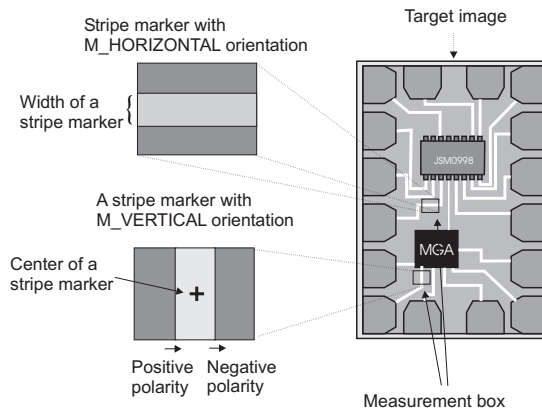
When searching for a marker, the relative importance (weight) assigned to each of the marker characteristics is crucial to the robustness of the operation. By default, 50% of the search weight is assigned to the edge strength; the remaining 50% is equally divided among all characteristics that can have a weight factor and that are set to a value other than `M_ANY` (the value used to flag an "ignore" state). This makes the edge strength by far the most important characteristic in the search.

You can override this default by adding `M_WEIGHT_FACTOR` to certain of the marker characteristics (see *MmeasSetMarker()* in the *MIL Command Reference* manual for the list of applicable characteristics). However, to better control the search, it is recommended when specifying weight factors that you assign a weight factor to all the enabled characteristics which support weight factors to a total of 100%.

For example, in a case where you must distinguish between two edge markers of different contrast, you can specify the typical contrast of the marker to be found. If you specify only this characteristic, the default search algorithm will assign a 50% weight to the edge strength and the remaining 50% to the contrast. However, if the edge you want to ignore has the higher edge strength, the desired edge might not be found. In this case, specifying the weights as 30% for the edge strength and 70% for the contrast will give precedence to the edge with the best match to the specified contrast.

Stripe markers: fundamental characteristics

A stripe marker is simply a marker with two edges, therefore the discussion of edge characteristics applies to each of the stripe marker's edges. However, certain characteristics have special attributes applicable only to stripe markers.



Polarity

The edges of a stripe marker can have opposite (M_OPPOSITE) polarities or can have similar (M_SAME) polarities.

Contrast and contrast variation

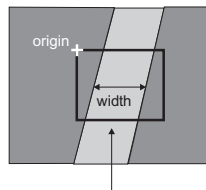
The contrast for a stripe marker requires two values, one for each of the stripe's edges. The contrast of the second edge can be set to M_SAME if both edges of the stripe have approximately the same contrast. Both values can be set to M_ANY if the contrast is unknown (default). The contrast variation for a stripe marker should be the maximum of the contrast variations of both its edges.

Width and width variation

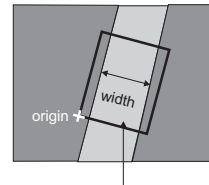
To help find a stripe marker, you can specify the typical distance between both of its edges (`M_WIDTH`) in pixels. You can also specify by how many pixels the width of a stripe marker might vary (`M_WIDTH_VARIATION`). This value should be equivalent to the maximum amount of variation.

Width of a stripe marker

The width of a stripe marker is the average distance in pixels between its edges. Note, if the marker measurement box (processing region) is at an angle, the width is measured according to the orientation of the box, as shown below.



Measurement box with a vertical orientation



Measurement box with a horizontal orientation at a 75 degree angle

Position

The position (`M_POSITION`) of a stripe marker is considered the center between the two edges of the stripe. This is the default reference position. The reference position of a marker can be moved with `MmeasSetMarker(M_MARKER_REFERENCE)`.

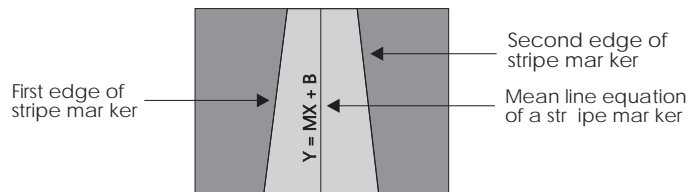
Length

The length of a stripe marker is measured along the mean line between its exterior edges. The length of either of its edges can also be measured. The length being measured is restricted to the portion of the marker contained within the measurement box. Note, the length of a stripe marker cannot be set, but can be returned with `MmeasGetResult()`.

Line equation

You can calculate the equation of the mean line following an stripe marker: $Y = MX + B$, where M denotes the slope of the line and B denotes the Y-intercept.

Line equations can be calculated for a stripe, or each edge of a stripe marker. The line equation of a stripe marker is the mean of the line equations of its edges.



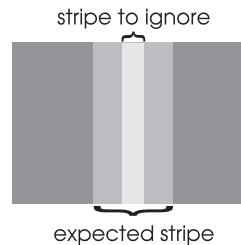
Note, the line equation of a stripe marker cannot be set, but can be returned with *MmeasGetResult()*. The slope of the line equation (M_LINE_EQUATION_SLOPE) and the Y intercept of the line equation (M_LINE_EQUATION_INTERCEPT) can also be returned separately.

Stripe markers: advanced characteristics

Inside edge and inside-edge variation

Inside edge

To help find a stripe marker, you can specify the typical number of edges located between the external edges of the stripe marker you are defining. For example, in the following illustration, the two stripes share the same position since their centers coincide.



To find the larger stripe without having to determine and specify its width, specify 2 as the number of inside edges of the stripe marker (`M_EDGE_INSIDE`). To find the smaller stripe, specify 0 (the default, `M_ANY`, ignores the possibility of any inside edges). The identification of inside edges is based only on the edge threshold setting (`M_EDGE_THRESHOLD`). The number of such edges found can also be returned using `M_EDGE_INSIDE` as a result type.

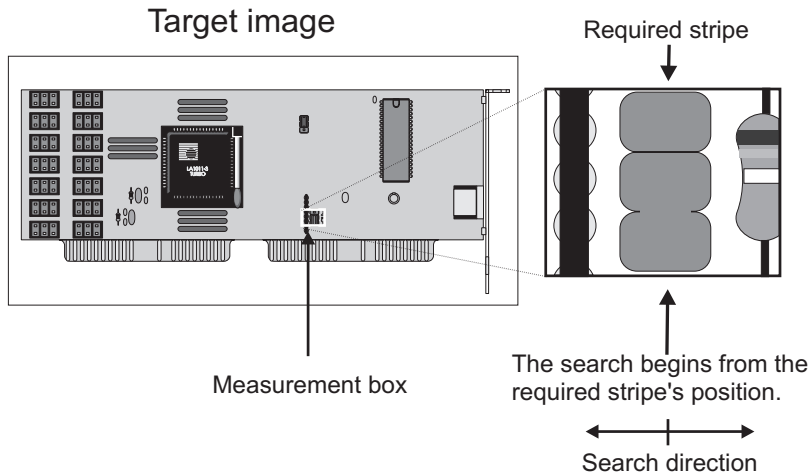
Inside edge variation

You can also specify the tolerance in the number of inside edges of a stripe (`M_EDGE_INSIDE_VARIATION`). Note, this tolerance should be in increments of two if stripes are contained within stripes, since two edges are recognized for each stripe.

Position inside stripe

If necessary when defining a stripe marker, you can specify if the X and Y coordinates of `M_POSITION` must be located inside or outside of the stripe (`M_POSITION_INSIDE_STRIPE`). If the position is defined as being within the stripe, the search algorithm is modified so that the search proceeds outwards in both directions from that point, making the operation faster and more robust. `M_POSITION_INSIDE_STRIPE` is useful when the required stripe does not have the greatest edge strength or

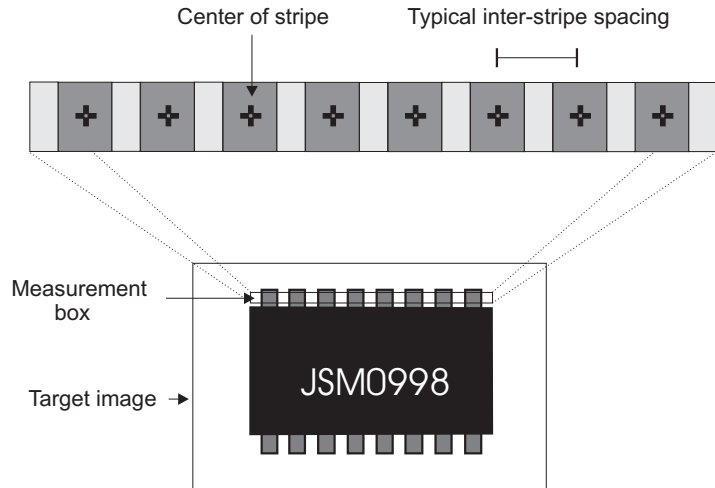
when it is difficult to set the measurement box without clipping other similar image characteristics. If defined as outside, no stripe including the position is considered.



Since other aspects of the image can have similar features or stronger edges, the required stripe might not be found. In order to successfully locate the stripe, specify its `M_POSITION` and then set `M_POSITION_INSIDE_STRIPE` to `M_YES` so that the search begins from the marker's approximate position, allowing the marker to be found.

Multiple marker characteristics

To use a multiple marker, only a few additional *MmeasSetMarker()* parameters need to be set. The marker edge or stripe characteristics are set just as with any marker, except that `M_POSITION` and its weight factor is ignored.



Specify the number of edges or stripes (`M_NUMBER`) to be found (default is 1) and the typical spacing between them if a regular pattern is expected (`M_SPACING`). Unless a minimum number (`M_NUMBER_MIN`) is specified, no results will be returned if the number of edges or stripes found falls below `M_NUMBER`. If the exact number of edges or stripes is unknown then `M_NUMBER` can be set to `M_ALL`.

When the `M_SPACING` setting is enabled, an initial search is performed to find all edges or stripes which best conform to the marker characteristics. This group of edges or stripes are then inspected to ensure that the spacing constraints are met. The marker's `M_SPACING` can be set to `M_SAME`, which takes the average spacing of all located edges or stripes and then applies this spacing as a criteria for determining the marker's actual edges or stripes.

`M_WIDTH` can also be set to `M_SAME`, meaning that the average width is applied as a constraint to all located stripes.

Measurements between two markers

The *MmeasCalculate()* function performs calculations between two markers' reference positions. The default setting of the *MmeasCalculate()* function performs all measurements; distance, angle, and line equation. For a stripe marker the center position is used as the reference position from which to take measurements.

Steps to taking measurements between two markers

The series of steps outlined below are usually followed to take measurements between two markers:

1. Allocate a result buffer, using *MmeasAllocResult()*.
2. Allocate, define, and find each marker with the *MmeasFindMarker()* function (see the steps previously outlined in the *Steps to finding and obtaining measurements of markers* section).
3. Call *MmeasCalculate()*, specifying which markers are to be used as the first and second reference positions and which measurements to take.
4. Read the results, using *MmeasGetResult()*.

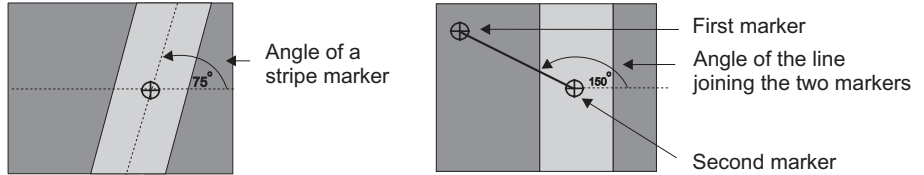
Calculating with multiple markers

If both markers are multiple markers, then calculations are made using the edges or stripes of the first marker and the corresponding edges or stripes in the second marker. The number of calculations is limited to the smallest number of results held in either marker (that is, if a marker contains only one edge or stripe, then only one calculation is performed, regardless of the number of edges or stripes contained in the other marker).

With a multiple marker, results for each calculation will be held in an array. Note that the array which you pass to *MmeasGetResult()* must be large enough to hold the result for each edge or stripe. If necessary, *MmeasGetResultSingle()* can be used to retrieve a single result from the array.

Angle

You can calculate the angle of a line joining two markers as shown below.

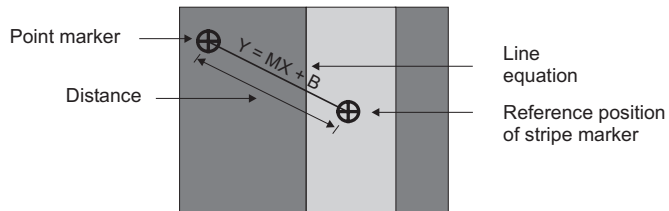


The angle is measured in a counter-clockwise direction relative to the positive x-axis, and can be a value from 0 to 360 degrees.

Line equation and distance

The line equation and distance can be calculated for the line joining two markers (see diagram below).

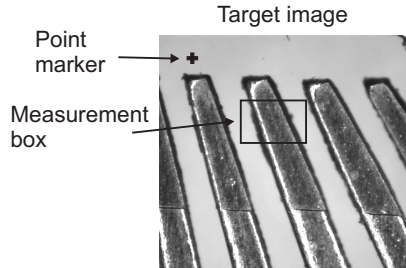
The distance between two markers is the distance between both of their reference positions, as illustrated in the diagram below.



The horizontal or vertical distance between two markers can also be calculated by using `M_DISTANCE_X` or `M_DISTANCE_Y` as the result type with `MmeasGetResult()`.

A measurement example

The following example illustrates the steps to take to find a stripe in an image, and measure its position, width, and angle. It also demonstrates how to perform calculations using a point marker as a reference point.



```

/* File name: mmeas.c
 * Synopsis: This program measures the position, width and angle of
 *           a stripe in an image, and marks its center and edges.
 *           It also calculates the length and angle of a line going
 *           from a reference point in the image to the center of
 *           the located stripe.
 */

#include <stdio.h>
#include <mil.h>

/* Source MIL image file specification. */
#define IMAGE_FILE          M_IMAGE_PATH"lead.mim"

/* Processing region specification */
#define MEAS_BOX_WIDTH      128
#define MEAS_BOX_HEIGHT    100
#define MEAS_BOX_POS_X     166
#define MEAS_BOX_POS_Y     130

/* Target stripe typical specifications. */
#define STRIPE_POLARITY_LEFT    M_POSITIVE
#define STRIPE_POLARITY_RIGHT  M_NEGATIVE
#define STRIPE_WIDTH           45L
#define STRIPE_WIDTH_VARIATION 10L

/* Reference point specification */
#define REFERENCE_POS_X       60L
#define REFERENCE_POS_Y       45L

```

(cont.)

```

/* Allocate a stripe marker */
MmeasAllocMarker(MilSystem, M_STRIPE, M_DEFAULT, &StripeMarker);

/* Specify the stripe approximative definition */
MmeasSetMarker(StripeMarker, M_POLARITY, STRIPE_POLARITY_LEFT,
STRIPE_POLARITY_RIGHT);
MmeasSetMarker(StripeMarker, M_WIDTH, STRIPE_WIDTH, M_NULL);
MmeasSetMarker(StripeMarker, M_WIDTH_VARIATION, STRIPE_WIDTH_VARIATION,
M_NULL);
MmeasSetMarker(StripeMarker, M_BOX_ANGLE_MODE, M_ENABLE, M_NULL);

/* Specify the search box size. */
MmeasSetMarker(StripeMarker, M_BOX_ORIGIN,
MEAS_BOX_POS_X, MEAS_BOX_POS_Y);
MmeasSetMarker(StripeMarker, M_BOX_SIZE,
MEAS_BOX_WIDTH, MEAS_BOX_HEIGHT);

/* Find the stripe and measure its width and angle. */
MmeasFindMarker(M_DEFAULT, MilImage, StripeMarker, M_DEFAULT);

/* Get the stripe position, width and angle. */
MmeasGetResult(StripeMarker, M_POSITION, &StripeCenterX,
&StripeCenterY);
MmeasGetResult(StripeMarker, M_POSITION+M_EDGE_FIRST,
&StripeFirstEdgeX,
&StripeFirstEdgeY);
MmeasGetResult(StripeMarker, M_POSITION+M_EDGE_SECOND,
&StripeSecondEdgeX,
&StripeSecondEdgeY);
MmeasGetResult(StripeMarker, M_WIDTH, &StripeWidth, M_NULL);
MmeasGetResult(StripeMarker, M_ANGLE, &StripeAngle, M_NULL);

/* Draw the contour of the measurement box */
MmeasGetResult(StripeMarker, M_BOX_CORNER_TOP_LEFT +M_TYPE_LONG, &Box00X,
&Box00Y);
MmeasGetResult(StripeMarker, M_BOX_CORNER_TOP_RIGHT +M_TYPE_LONG, &Box10X,
&Box10Y);
MmeasGetResult(StripeMarker, M_BOX_CORNER_BOTTOM_LEFT +M_TYPE_LONG, &Box01X,
&Box01Y);
MmeasGetResult(StripeMarker, M_BOX_CORNER_BOTTOM_RIGHT+M_TYPE_LONG, &Box11X,
&Box11Y);
MgrrLine(M_DEFAULT, MilImage, Box00X, Box00Y, Box10X, Box10Y);
MgrrLine(M_DEFAULT, MilImage, Box10X, Box10Y, Box11X, Box11Y);
MgrrLine(M_DEFAULT, MilImage, Box11X, Box11Y, Box01X, Box01Y);
MgrrLine(M_DEFAULT, MilImage, Box01X, Box01Y, Box00X, Box00Y);

/* Draw a cross on the center, left and right edge of the found stripe.*/
DrawCross(MilImage, StripeCenterX, StripeCenterY, CROSS_COLOR);
DrawCross(MilImage, StripeFirstEdgeX, StripeFirstEdgeY, CROSS_COLOR);
DrawCross(MilImage, StripeSecondEdgeX, StripeSecondEdgeY, CROSS_COLOR);

```

(cont.)

```

/* Print the result. */
printf("The stripe in the image is at position %.2f,%.2f and\n",
StripeCenterX, StripeCenterY);
printf("is %.2f pixels wide with an angle of %.2f degrees.\n",
StripeWidth, StripeAngle);
printf("Its center and edges have been marked.\n\n");
printf("Press <Enter> to continue.\n");
getchar();

/* Allocate a point marker */
MmeasAllocMarker(MilSystem, M_POINT, M_DEFAULT, &PointMarker);

/* Specify the reference point position in the image.*/
MmeasSetMarker(PointMarker, M_POSITION, REFERENCE_POS_X, REFERENCE_POS_Y);

/* Draw reference point position in the image */
DrawMark(MilImage, REFERENCE_POS_X, REFERENCE_POS_Y, MARK_COLOR);

/* Allocate a calculation result buffer */
MmeasAllocResult(MilSystem, M_CALCULATE, &CalcResult);

/* Calculate the distance and angle between the point and the stripe. */
MmeasCalculate(M_DEFAULT, PointMarker, StripeMarker, CalcResult,
M_DISTANCE+M_ANGLE);

/* Get the distance and angle. */
MmeasGetResult(CalcResult, M_DISTANCE, &ReferenceDistance, M_NULL);
MmeasGetResult(CalcResult, M_ANGLE, &ReferenceAngle, M_NULL);

/* Draw a cross on the reference point and a line from that point
 * to the center of the found stripe.
 */
DrawCross(MilImage, REFERENCE_POS_X, REFERENCE_POS_Y, CROSS_COLOR);
MgraLine(M_DEFAULT, MilImage, REFERENCE_POS_X, REFERENCE_POS_Y,
(long)(StripeCenterX+0.5),(long)(StripeCenterY+0.5));

/* Print the result. */
printf("The distance and angle from the drawn reference point are\n");
printf("%.2f pixels and %.2f degrees.\n", ReferenceDistance,
ReferenceAngle);
printf("\nPress <Enter> to end.\n");
getchar();

/* Free all allocations. */
MmeasFree(CalcResult);
MmeasFree(PointMarker);
MmeasFree(StripeMarker);
MbufFree(MilImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);
}

```

(cont.)

```

/* Draw a cross at the specified position. */
void DrawCross(MIL_ID ImageId, double CenterX, double CenterY, long Color)
{
    MgraColor(M_DEFAULT, Color);
    MgraLine(M_DEFAULT, ImageId,
        (long)(CenterX+.5)-(CROSS_SIZE/2), (long)(CenterY+.5),
        (long)(CenterX+.5)+(CROSS_SIZE/2), (long)(CenterY+.5));
    MgraLine(M_DEFAULT, ImageId,
        (long)(CenterX+.5), (long)(CenterY+.5)-(CROSS_SIZE/2),
        (long)(CenterX+.5), (long)(CenterY+.5)+(CROSS_SIZE/2));
}

/* Draw a reference mark at the specified position. */
void DrawMark(MIL_ID ImageId, double CenterX, double CenterY, long Color)
{
    MgraColor(M_DEFAULT, Color);
    MgraArc(M_DEFAULT, ImageId, (long)(CenterX+.5), (long)(CenterY+.5),
        CROSS_SIZE, CROSS_SIZE, 0.0, 360.0);
    MgraLine(M_DEFAULT, ImageId, (long)(CenterX+.5)-(CROSS_SIZE),
        (long)(CenterY+.5), (long)(CenterX+.5)+(CROSS_SIZE), (long)(CenterY+.5));
    MgraLine(M_DEFAULT, ImageId, (long)(CenterX+.5),
        (long)(CenterY+.5)-(CROSS_SIZE), (long)(CenterX+.5),
        (long)(CenterY+.5)+(CROSS_SIZE));
}

```

Chapter 16: Specifying and managing your data buffers

This chapter discusses data buffers in detail. It shows you how to allocate and manage data buffers, and how to restrict an operation to a portion of a data buffer by using child buffers. It shows you how YUV buffers are stored, how to create a user-defined buffer, and how MIL defines the pixel reference position.

Data buffers

Data buffers

In this manual, the term *data buffer* is used loosely to refer to the most general type of data buffer (storage area) that is allocated by the MIL package and operated on by most MIL functions. For example, a data buffer can be a buffer for image data or one for lookup table (LUT) data. Besides data buffers, there are also other buffers (for example, result buffers), which are specific to a particular group of functions. These types of buffers are discussed in the chapters describing their related functions.

Allocating data buffers

All data buffers must be allocated before a function can access them. You can allocate a monochrome buffer using *MbufAlloc1d()*, *MbufAlloc2d()*, or *MbufAllocColor()*. You allocate a color buffer using *MbufAllocColor()*.

When allocating a data buffer, you must specify its:

- Target system.
- Dimensions.
- Data type and depth.
- Attribute.

Controlling specific parts

You can manipulate or control specific parts of data buffers by allocating and using child buffers. A child buffer is a subset of the parent buffer (a specific area of the parent buffer). Although any change made to the child buffer data affects the parent buffer, the buffer is considered a data buffer in its own right; wherever the parent buffer can be used, you can use the child buffer instead to affect only a part of the buffer. All results are returned relative to the child buffer coordinates rather than the parent buffer.

Target system

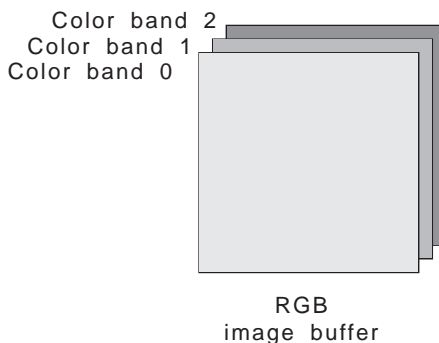
A data buffer is allocated on the specified system. If the `M_DEFAULT_HOST` system is specified, the default Host system of the current MIL application will be used. If `M_DEFAULT` is specified, MIL will select the most appropriate system on which to allocate the data buffer (it can be the default Host system or any currently allocated system).

In addition, any operation involving one or more buffers will be performed by the most appropriate system that is associated with one of the buffers. By default, if none of these systems is more appropriate than the Host, the Host is used to perform the operation.

Specifying the dimensions of a data buffer

Data buffers can have up to three dimensions: an x, y, and color band dimension. Most data buffers have an x dimension (for example, LUT buffers) or an x and y dimension (for example, monochrome image buffers). The color-band dimension has been provided to allow you to store data for each color component used to represent an image; when allocating color buffers, each band will be of the same data depth and type.

Once you finish using a data buffer, you should release its memory space, using *MbufFree()*.



Certain MIL functions support manipulating multi-band image buffers. See *Chapter 21: Color* for details on handling color image buffers.

Data type and depth

Data type and depth

The data depth of a buffer indicates the number of bits per band in the buffer (1, 8, 16, 32). The data type of a buffer indicates how its data is internally represented (that is, whether the data is considered signed, unsigned, or floating-point). Supported combinations are: 1-bit packed binary; 8-, 16-, and 32-bit integer (signed and unsigned); and 32-bit floating-point. If a function can only operate on data buffers of certain depths, this is explicitly stated in the command's description, otherwise the function can be used with any combination of data buffers (the *MIL Command Reference* manual).

Packed binary buffers

The packed binary data format represents each pixel by a single bit, in a state of 0 or 1. Therefore, 8 pixels can be packed in a single byte (known as an 8-bit data unit); that is, in a format eight times smaller than an 8-bit image.

Processing done directly on a packed binary buffer is very fast and efficient. Many MIL functions support accelerated processing using packed binary buffers. General processing functions which do not support packed binary buffers directly, automatically convert the data into a suitable data type buffer, perform the operation, and re-convert the resulting buffer to packed binary.

For efficiency, when possible, you should store binary data in packed binary buffers (rather than, for example, 8-bit integer buffers with only the values 0 and 0xFF). General processing functions that are optimized for packed binary buffers are noted as such in the *MIL Command Reference* manual.

Integer and floating-point buffers

In general, the fewer bits per pixel in a buffer, the faster an operation can be performed on the buffer. Packed binary buffers are the fastest to process. When you need to use integer buffers, use 8 bits per pixel when possible, 16 bits if necessary, and 32

bits as a last resort. When you need non-integer values, extra precision, or a greater dynamic range, you can use floating-point data buffers.

Attribute

Buffer type and usage

The data buffer attribute indicates the buffer type and its intended usage. MIL uses this information to determine the most appropriate location in physical memory in which to allocate the buffer, and how to handle the buffer. A data buffer can be one of the following types:

- M_IMAGE (image buffer).
- M_LUT (lookup table buffer).
- M_KERNEL (kernel buffer for convolution functions).
- M_STRUCT_ELEMENT (structuring element buffer for morphology functions).

Allocating an image buffer

When allocating an image buffer (M_IMAGE), you must give more information about its intended usage. An image buffer can be any combination of the following:

- A buffer that can be displayed (M_DISP).
- A buffer that can be processed (M_PROC).
- A buffer in which data can be grabbed (M_GRAB).
- A buffer in which data is stored in a compressed format (M_COMPRESS).

For example, to allocate an image buffer that can be displayed and used for processing, its attribute should be given as:

M_IMAGE + M_DISP + M_PROC

In general, buffers are allocated in Host memory instead of on-board memory by default. This is because on-board memory is limited in size and Host memory can be accessed much faster than on-board memory. However, if the system has an on-board processor, the buffer is allocated on-board by default. These defaults can be overridden by using the *MbufAlloc...()* M_ON_BOARD and M_OFF_BOARD attributes.

Grab buffers

Buffers with an attribute of `M_GRAB` are allocated in DMA memory, which is physically contiguous and always present. This is also known as non-paged memory. An advantage to non-paged memory is that a bus mastering device can write to it without the help of the CPU.

If a system does not support grab buffers (for example, `M_HOST_SYSTEM`), you could still allocate a buffer on such a system in physically contiguous and always present memory by giving it an `M_NON_PAGED` attribute instead.

Displayable buffers

When a displayable buffer is allocated and selected for display (*MbufAlloc...*() with M_DISP, and then *MdispSelect()*), two buffers are maintained internally: one in Host memory for processing purposes, the other in a frame buffer surface (maintained directly or through a DIB) for display purposes (not necessarily the same size). When the Host buffer is modified, its associated buffer in the frame buffer surface is automatically updated. When displaying a buffer, both the buffer and the display must have been allocated on the same system.

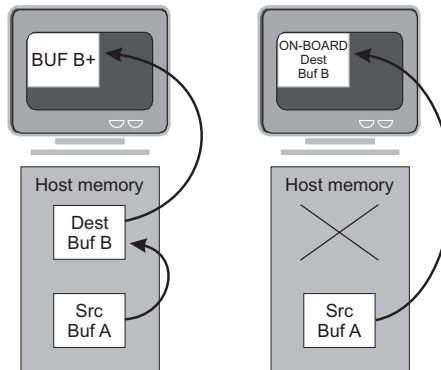
When grabbing a single frame into a displayable buffer, MIL grabs into the Host memory version of the buffer and then updates the display of the buffer. When grabbing continuously, the grab is made directly to the frame buffer surface and then at the end of the grab, the Host buffer is updated.

Overriding the default allocation sequence

On boards with a display section, you can override the default buffer allocation sequence and force allocation only in the frame buffer surface using the *MbufAlloc...*() M_ON_BOARD attribute. In general, the buffer is allocated in the non-displayable area of the frame buffer surface. If you are in a non-windowed mode and the M_DISP attribute is specified, the buffer will be in the displayable area. Note you can allocate only one M_DISP+M_ON_BOARD buffer and one M_OVR+M_ON_BOARD buffer unless stated in the *MIL/MIL-Lite Board Specific Notes* manual.

Overriding the default allocation sequence is useful when allocating a displayable buffer under any non-windowed mode. If you are not using the displayable buffer for processing or are

only using it as a destination, storing the buffer on-board will avoid the extra copy operation to the display without the penalty of slowing down processing.



Even if it is not in the displayed area of the frame buffer, the image buffer depth and display depth must be the same.

Internal format of the buffer

It is also possible to force the internal representation of a data buffer using internal storage format specifiers, such as `M_PACKED` or `M_PLANAR`, which force the data buffer to be in a packed or planar format, respectively. Refer to *MbufAllocColor()* for a complete list of internal format specifiers.

Insufficient memory

If there is insufficient memory of the appropriate type to allocate a buffer with the specified attributes, the function generates an error and does not allocate the buffer.

Inappropriate data buffer usage

If you try to use a data buffer in a situation that is not appropriate for its allocated attribute, an error message is generated and the operation is not performed. For example, if you try to display a buffer without an `M_DISP` attribute with *MdispSelect()*, an error message will be generated.

Manipulating and controlling certain data buffer areas

You can manipulate or control specific parts of a data buffer by creating a child buffer within it or by copying specific parts of it to another buffer.

Child buffers

Child buffers are subsets of parent buffers

A child buffer is a subset (or region of interest) of a given data buffer (known as the parent buffer). Child buffers occupy a specific area of the parent buffer. Since this area is part of the same physical space as the parent buffer, changes made to the child buffer affect the parent buffer and vice versa.

Allocating child buffers

The child buffer is considered a data buffer in its own right. Like its parent buffer, a child buffer must be allocated so that it can be associated with an identifier and recognized as an entity by the MIL package. Allocate a monochrome child buffer using *MbufChild1d()* or *MbufChild2d()*. To allocate a child buffer consisting of only one of the color bands of a multi-band image buffer, use *MbufChildColor()* or *MbufChildColor2d()*. Note, as a subset of the parent buffer, a child buffer cannot exceed the bounds of its parent in any dimension. For example, a color buffer cannot be created from a monochrome parent buffer.

A child buffer takes on the same attributes and type as the parent buffer. In general, any operation that can be performed on the parent buffer can also be performed on the child buffer.

Allocate a child buffer by specifying its size and offset with respect to each of the parent buffer dimensions. After, when using the child image buffer, any specified or returned coordinates are relative to the child's top-left corner.

As with any MIL data buffer, once you have finished using a child data buffer, you must delete it, using *MbufFree()*.

One major benefit of the child buffer is being able to handle several buffers simultaneously, in contexts where normally only one buffer can be handled. For example, when using MIL in non-windowed mode, you can only display one buffer at a time. However, you might want to display the source and destination buffer of an operation simultaneously. You can get around this situation by allocating a displayable image buffer as large as the display, then allocating two child buffers from this buffer. You can then use one as the source data buffer and one as the destination. When the parent buffer is selected on the display (*MdispSelect()*), both the source and the destination child buffers can be seen.

Copying specific buffer areas

As an alternative to using a child buffer, you can restrict operations to specific areas or bits of a **buffer** (child or parent) by copying the required portions to another buffer. You can copy data from any type of data buffer to another using any of the following functions. For example:

- Copy an image buffer to another buffer at the specified offset, using *MbufCopyClip()*. Data that falls outside of the destination buffer will be automatically clipped.
- Copy specific non-sequential areas to another buffer based on a conditional buffer, using *MbufCopyCond()*. Source buffer data is copied to the destination buffer if corresponding data in the specified conditional buffer satisfies the copy condition. Other data in the destination buffer is left unaffected.
- Copy specific non-consecutive bits to another buffer based on a mask, using *MbufCopyMask()*. Only destination bits that correspond to non-zero bits in the mask are modified with source bits.
- Copy a single band of a multi-color band buffer to or from a single-band buffer, using *MbufCopyColor()* or *MbufCopyColor2d()*. This allows you to operate on a single color band of a buffer.

If the source buffer depth is greater than that of the destination, the most significant bits are truncated when the data is copied into the destination. If the source and destination buffers are signed and the destination depth is greater than that of the source, the source data is sign-extended when it is copied into the destination.

MbufCopy() copies the entire buffer into another buffer, while the other commands copy only portions of a buffer.

Managing data buffers

Besides the copy functions discussed in the previous section, MIL provides several other data buffer management functions. These allow you to transfer data between an array and a buffer, load data into a buffer (or a sequence of buffers), and save a buffer (or a sequence of buffers) to disk.

Putting and retrieving data

You can put data from an array into a data buffer, using *MbufPut()*, *MbufPut1d()*, *MbufPut2d()*, *MbufPutColor()*, or *MbufPutColor2d()*. *MbufPut()* puts data in the entire buffer, while *MbufPutColor()* or *MbufPutColor2d()* put data into one or all color bands of a multi-band buffer. The other two commands allow you to put data in a selected area of a monochrome buffer, respectively.

In addition, you can retrieve data from a data buffer and place it into an array, using *MbufGet()*, *MbufGet1d()*, *MbufGet2d()*, *MbufGetColor()*, or *MbufGetColor2d()*. *MbufGet()* gets data from the entire buffer, while *MbufGetColor()* or *MbufGetColor2d()* get data from one or all bands of a multi-band buffer. The other two commands, like their ‘put in buffer’ counterparts, allow you to get data from a selected area of a monochrome, respectively.

❖ Note that you can also access the contents of a MIL buffer from an array by using *MbufInquire()*. Inquire the Host address of the buffer, and then using a pointer access the buffer as an array. This is discussed in more detail later.

Loading a data buffer

You can load data, using one of two methods:

- Load data into an automatically allocated MIL data buffer, using *MbufImport()* with `M_RESTORE`, or using *MbufRestore()*.
- Load data into a previously allocated MIL data buffer, using *MbufImport()* with `M_LOAD` or using *MbufLoad()*.

These commands internally handle the opening and closing of the file. With *MbufImport()*, you can specify the file's format. *MbufLoad()* and *MbufRestore()* will read the data in the file to determine the format, therefore they might take more time to return a result.

Saving a data buffer

You can save a data buffer to disk, using *MbufExport()* or *MbufSave()*. *MbufExport()* is the most general of these commands and can save data in any MIL-supported file format. *MbufSave()* can only save data in an `M_MIL` file format.

These functions internally handle opening and closing the file. If the given file name already exists, the file will be overwritten.

Loading and saving a sequence of data buffers

You can import or export a sequence of image buffers to a file using *MbufImportSequence()* or *MbufExportSequence()*, respectively. The available file formats are: standard AVI DIB format, MJPEG format, and proprietary AVI MIL format.

Controlling how color image buffers are stored

A color image buffer's internal representation can be either in a planar or packed format. When allocating the buffer, if its attribute is also set to `M_PLANAR`, the pixels are stored in planes (for example, RRR GGG BBB). When allocating the buffer, if its attribute is set to `M_PACKED`, each pixel is stored as one unit containing all its components (for example, RGB RGB RGB).

MIL automatically selects the most appropriate format, according to the specified intended usage attribute. If an image buffer is allocated in one format, and a general processing function requiring another format is called, the function will automatically convert the data to the required format and re-convert it back to its original format upon completion. To change a buffer's default internal storage format, change the internal storage part of the attribute parameter for *MbufAllocColor()*. Note that it might be slower to process buffers with `M_PACKED` attributes.

In general, packed formats are mostly used for display purposes; when selecting a buffer's attribute as `M_DISP`, the default internal representation is usually packed. This configuration allows for faster transfers to display sections that handle packed data (for example, VGA). However, if the display section of your board has dedicated red, green, and blue frame buffer planes, the buffer is allocated in planar format.

Planar formats are generally preferred for processing. Here, the buffer stores each pixel as three component planes (for example, RRR, GGG, BBB). Processing is done on each of the components separately.

When allocating an image buffer with more than one attribute, for example, `M_DISP` and `M_PROC`, the buffer's internal storage requirements for the display will take precedence over other attributes.

See the *MIL/MIL-Lite Board-Specific Notes* manual to determine which formats are supported on your board.

RGB buffers

By default, MIL allocates color image buffers in an RGB color format. The pixels are internally stored in little-endian order, that is, they are stored in memory from their least-significant to the most significant bytes. The definitions of the RGB formats that are available are shown here. The corresponding MIL constant is shown in brackets beside the common format name.

RGB data formats

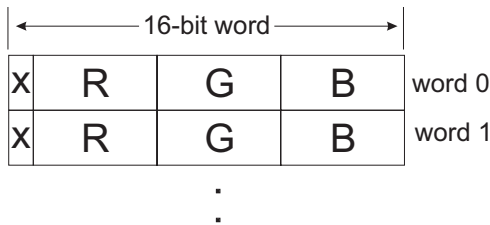
BGR24 packed (M_BGR24+M_PACKED) is a format whereby each pixel is internally stored as three consecutive bytes in little-endian order, that is:

Byte 0	B
Byte 1	G
Byte 2	R
Byte 3	B
Byte 4	G
Byte 5	R
	⋮

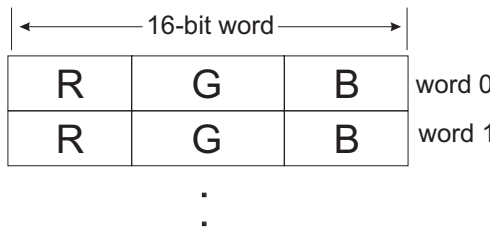
BGR32 packed (M_BGR32+M_PACKED) is a format whereby each pixel is internally stored as four consecutive bytes, in little-endian order. The most-significant byte is a "don't care" byte, as shown below:

Byte 0	B
Byte 1	G
Byte 2	R
Byte 3	X
Byte 4	B
Byte 5	G
Byte 6	R
Byte 7	X
	⋮

RGB15 packed (M_RGB15+M_PACKED) is a format whereby each pixel is internally stored as a 16-bit word with a 5-bit blue value (least significant), a 5-bit green value, a 5-bit red value, and a "don't care" bit (most significant), in little-endian order, as shown below. Note that when accessing an M_RGB15+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits of each band are set to 0.



RGB16 packed (M_RGB16+M_PACKED) is a format whereby each pixel is internally stored as a 16-bit word with a 5-bit blue value (least significant), a 6-bit green value, and a 5-bit red value (most significant), in little-endian order, as shown below. Note that when accessing an M_RGB16+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits of each band are set to 0.



RGB planar are formats whereby the color components of all the pixels are stored contiguously: (RRR..., BBB..., GGG...).

Binary buffers

Binary buffers have a different internal storage format than other types of buffers: eight pixels are stored in one byte. The leftmost pixel of an image is the least significant bit that is stored in memory.

YUV buffers

YUV is a compressed format in which Y is the grayscale component (luminance) and U and V are the color components. MIL supports grabbing, loading, or saving images in a YUV color format.

Although any general processing operation can be performed on YUV buffers, allocating them for processing purposes is not recommended because MIL is configured to process RGB color data only. However, MIL will automatically convert YUV buffer data to RGB for all general processing operations (including conversion for display), and re-convert it to YUV upon completion.

All YUV formats are supported even on the Host system. However, only some systems support grabbing into YUV buffers. See the *MIL/MIL-Lite Board-Specific Notes* manual to determine if grabbing into YUV buffers is supported on your system.

YUV buffers must be allocated as 3-band 8-bit buffers, however, the actual number of bits per pixel will differ depending on the YUV format selected.

The supported YUV formats are:

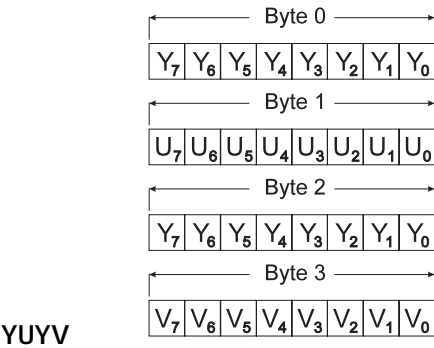
- YUV16 Packed
- YUV9 Planar
- YUV12 Planar
- YUV16 Planar

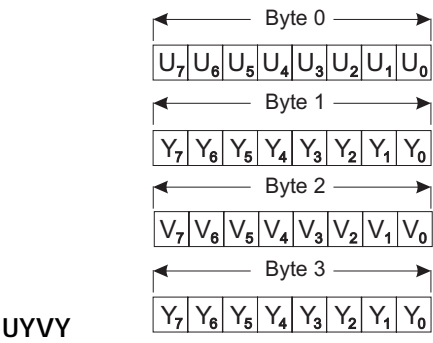
YUV16 Packed

YUV16 Packed or YUV 4:2:2 (M_YUV16+M_PACKED) is an interleaved data format. Although each pixel has a corresponding one byte Y (luminance component), each pair of pixels share the same one byte U (chrominance U) and the same one byte V (chrominance V). Since a pair (two pixels) is represented by 4 bytes, each pixel has an average of 16 bits per pixel.

The YUV16 packed data format has two available formats: YUYV and UYVY. The only difference between these two YUV formats is the ordering of data in the buffer. Certain digitizer boards grab data in exclusively YUYV or UYVY packed data format. In addition, certain display adapters are optimized to handle YUYV format (or can only handle the YUYV format in their underlay).

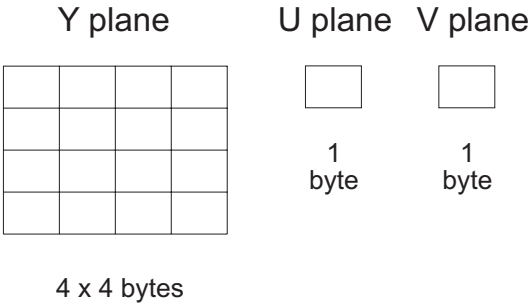
When you allocate an M_YUV16+M_PACKED buffer, MIL allocates the buffer in the format that is most suitable for the selected platform and the specified buffer attributes. You can, however, force a format using the M_YUV16_YUYV or M_YUV16_UYVY control types. When the buffer has an M_GRAB attribute, forcing an inappropriate format generates an error. When the buffer has an M_DISP attribute, if you force the buffer in the other YUV format, then CPU intervention is required to perform the automatic conversion. See the *MIL/MIL-Lite Board Specific Notes* for supported data formats.





YUV9 Planar

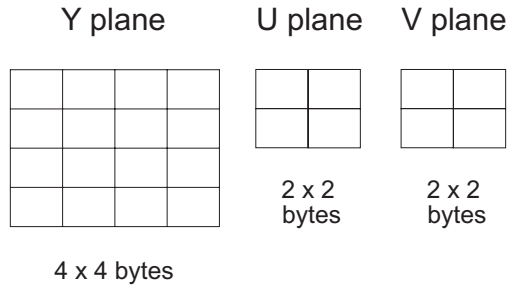
YUV9 Planar (M_YUV9+M_PLANAR) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 16 pixels share the same one byte of U (chrominance U) and the same one byte of V (chrominance V). Since the 16 pixels are represented by 18 bytes, each pixel has an average 9 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 1 byte each of U and V.



YUV12 Planar

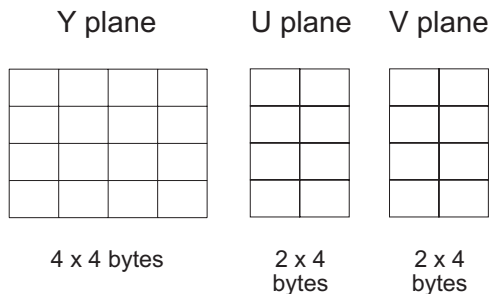
YUV12 Planar (M_YUV12+M_PLANAR) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 4 pixels share the same one byte of U (chrominance U) and the same one byte of V

(chrominance V). Since the 16 pixels are represented by 24 bytes, each pixel has an average of 12 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 4 bytes each of U and V.



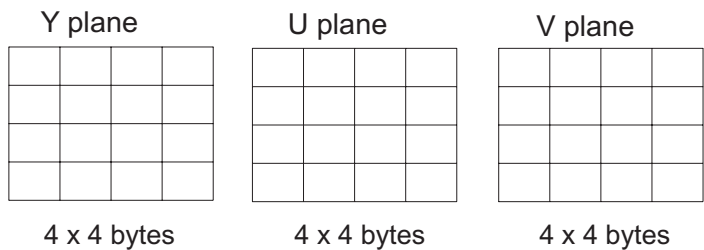
YUV16 Planar

YUV16 Planar (M_YUV16+M_PLANAR) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 2 pixels share the same 1 byte of U (chrominance U) and the same 1 byte of V (chrominance V). Since the 16 pixels are represented by 32 bytes, each pixel has an average 16 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 8 bytes each of U and V.



YUV24 Planar

YUV24 Planar (M_YUV24+M_PLANAR) is an uncompressed planar format whose components have a depth of one byte and are of equal size. Each pixel has a corresponding 1 byte Y (luminance) component, 1 byte U component (chrominance U), and 1 byte V component (chrominance V). Since the 16 pixels are represented by 48 bytes, each pixel has an average 24 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 16 bytes each of U and V.



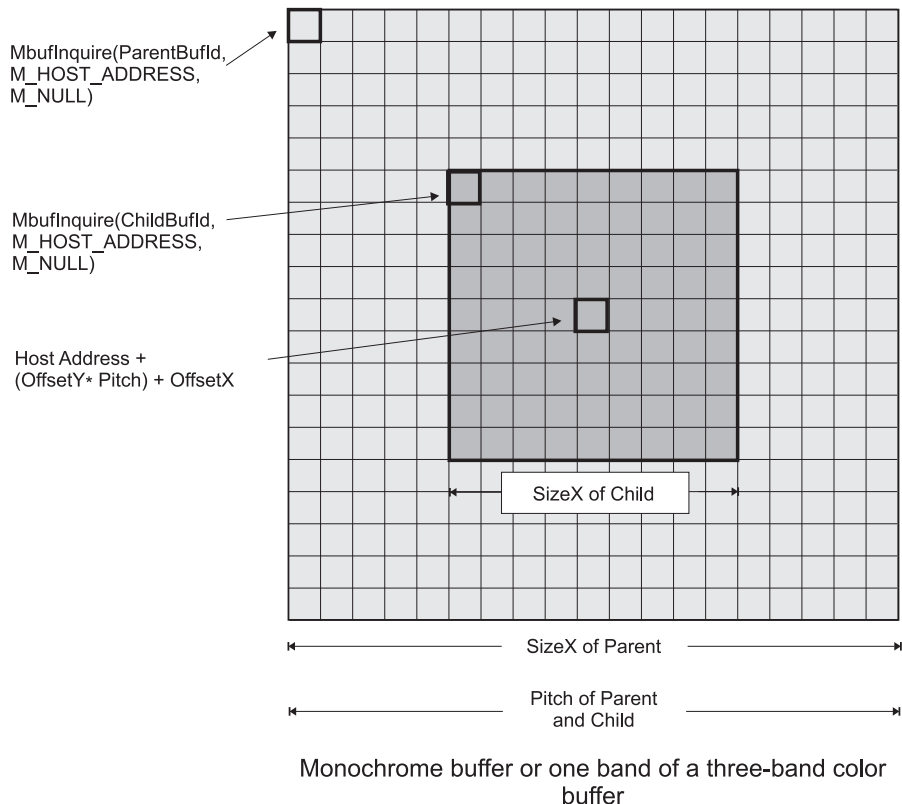
Child YUV buffers

You can create child buffers from YUV buffers in the same way as RGB child buffers. When creating YUV child buffers, MIL will keep the proportions of the U and V bands with respect to the Y band. For example, if your YUV9 Planar Y band is a size of 256 x 256 pixels, the U and V bands will be 1/4 the size of the Y band in each dimension (width and height): 64 x 64 pixels, which is 1/16 the size of the Y band. If a child buffer is 16 x 16 pixels, then the U and V bands will be 4 x 4 pixels. In other words, the 4 x 4 U and V bands (16 pixels) is 1/16 the size of the Y band (256 pixels).

Accessing a MIL buffer directly

If needed, a MIL buffer's contents can be accessed directly. For instance, if you want to calculate the average value of the pixels of your image, you could create a custom algorithm. The algorithm could be applied directly to the buffer without having to copy the contents of the MIL buffer into a user-allocated array (*MbufAlloc()*) by using *MbufGet()* and *MbufPut()*. To do so would be more efficient and might improve the performance of the custom algorithm.

In order to access the MIL buffer directly, the buffer's address and pitch must be known. Once you know this, you will be able to access them directly for optimum performance.



Address

The address of a parent or child buffer can be returned using *MbufInquire()*. Selecting `M_HOST_ADDRESS` will return a logical address, while `M_PHYSICAL_ADDRESS` will return a physical address. In either case, the first address of the buffer you are specifying will be the top left-most pixel in the image. Knowing the pitch and the depth of the buffer will tell you the address of the following row.

Pitch

The pitch of a buffer is the number of units between the beginnings of any two adjacent lines of the buffer's data and can be measured in pixels or bytes. Note that in some instances, the pitch in bytes will be more accurate than in pixels. If the last pixel falls outside of a 32-bit boundary (required by Windows), the start of the next row will be located at the beginning of the next 32-bit boundary; this process is called internal padding. When measuring the pitch in pixels, the padding can be counted as "extra" pixels, depending on the depth of the pixels. This will result in an inaccurate pitch.

Mapping a data buffer to user-allocated memory

Instead of allocating new memory to a buffer using *MbufAlloc...()*, you can create a buffer from the memory at a specified location, using *MbufCreate2d()* to create a monochrome data buffer and *MbufCreateColor()* to create a color data buffer. In these cases, MIL does not allocate any memory; it uses the memory that you provide.

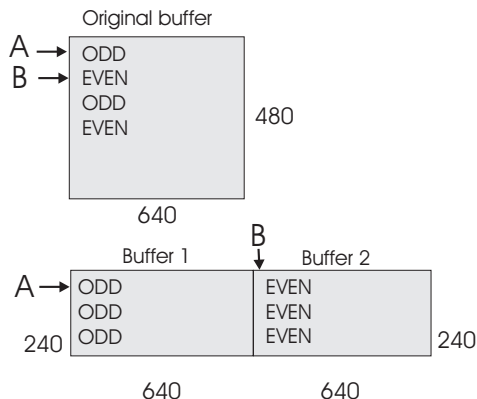
When creating a buffer with *MbufCreateColor()*, you must pass an array of pointers to the addresses of the data. For packed color buffers, you must pass an array of one pointer; for planar buffers, you must pass an array with the same number of pointers as the number of bands in the buffer. When creating a buffer with *MbufCreate2d()*, you must pass the address of the data. The address(es) can be either logical or physical. If you want to use the buffer for grabbing, the address(es) must be physical (grab buffers must be allocated in physically contiguous and always present memory, that is, non-paged). The *MbufCreate...()* functions must be used with caution because, when using physical addresses, these functions

provide direct manipulation of any of your PC's memory mapped devices; when using logical addresses, memory protection errors could result.

You can use *MbufInquire()* with the `M_HOST_ADDRESS` or `M_PHYSICAL_ADDRESS` control type to determine the Host's logical address or the physical address of a buffer's data, respectively. Note that the physical address is not necessarily an address in Host memory. It could be an address in on-board memory. If an on-board buffer is mapped to the Host, you can use the *MbufInquire()* function with the `M_HOST_ADDRESS` control type to determine the Host address to which it is mapped.

There are several instances when memory mapping is useful. A particularly useful instance is when processing and displaying an interlaced grab in a time critical application. In this case, you could use a displayable buffer to store and display the grabbed data. Then, to process each field as it is grabbed, you could use a buffer that is mapped to the odd field of the displayable buffer (Buffer 1) and a buffer that is mapped to the even field (Buffer 2).

Create Buffers 1 and 2 as follows:

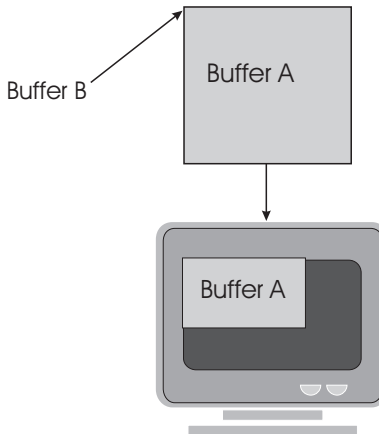


■ Buffer 1: (Odd field)

- Size = 640 x 240 (i.e., half height)
- Pitch = 1280 (i.e., to skip to the next field)
- Address = Address A (i.e., first pixel of the first row)

- Buffer 2: (Even field)
 - Size = 640 x 240 (i.e., half height)
 - Pitch = 1280 (i.e., to skip to the next field)
 - Address = Address B (i.e., first pixel of the second row)

In general, MIL automatically issues a display update after a displayed buffer has been modified. However, if a buffer selected on the display is modified using a mapped buffer, its display is not updated until you notify it of the change using *MbufControl(...M_MODIFIED...)*.



See *Chapter 26: Data Manipulation with multiple systems* for another instance where creating buffers is useful.

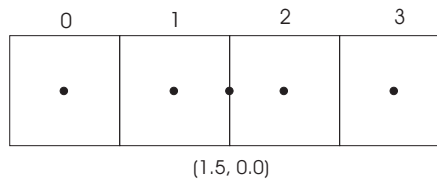
Pixel conventions

The center of a pixel is important for all MIL functions which return positional results with subpixel accuracy. The reference position of a pixel is its center, and the resulting subpixel coordinates are with respect to the pixel's center.

With this in mind, the coordinates of the center of an image can always be found using the following formula:

$$\left(\frac{\text{Width} - 1}{2}, \frac{\text{Height} - 1}{2} \right)$$

For example, the following image contains 4 pixels. If the formula is applied, the center of the image is found at (1.5, 0).

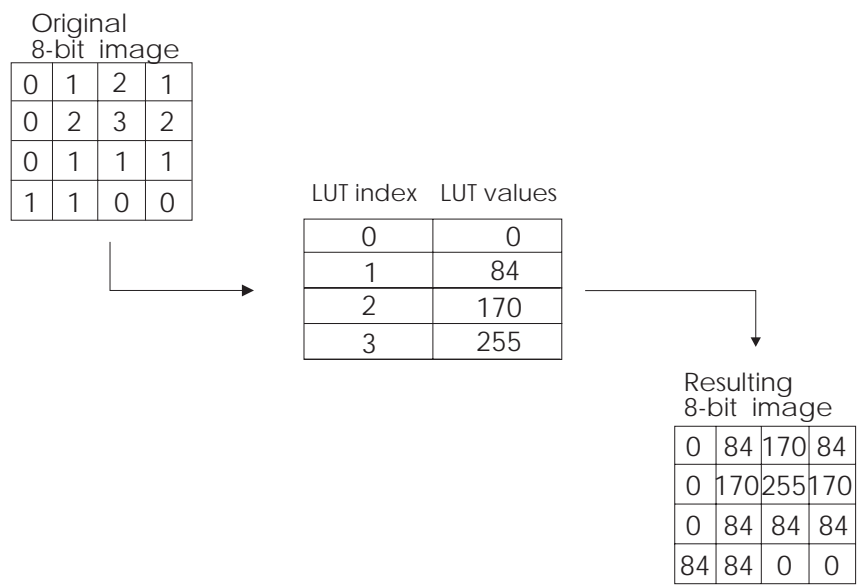


Chapter 17: Lookup tables

This chapter describes lookup tables (LUTs). It shows you how to generate and modify them and briefly discusses how to use them.

Lookup tables

Lookup tables (LUTs) are collections of memory locations that are used to map data to pre-calculated values. They can easily reduce a multi-step or complex operation to a single-step LUT mapping.



You can map an image buffer through a LUT, using *MimLutMap()*. If the hardware system permits, you can also use LUTs to precondition input data at acquisition time, before it is stored in an image buffer. LUTs can also be used (hardware system permitting) to adjust the color contrast and intensity of an image upon display, without affecting the actual data.

LUTs and data buffers

LUT buffers

The MIL package represents LUTs as LUT data buffers. As with any other data buffer, LUT buffers must be allocated before they are used. A LUT buffer can be loaded, stored, or copied to another buffer (not necessarily to another LUT buffer) or to disk. You can also allocate child LUT buffers. When a LUT buffer is no longer required, you should free its memory space, using *MbufFree()*.

Allocating LUT buffers

LUT buffers are typically one-dimensional data buffers created with *MbufAlloc1d()* (single row). However, you can allocate a color RGB LUT, using *MbufAllocColor()*. In this case, set the number of bands to 3 (for RGB), the y-dimension to 1, and the x-dimension to have enough entries to represent the full range of possible values of the image buffer.

Loading and generating data into LUTs

With MIL, you can generate data directly into a LUT buffer or calculate the data and then load it in a LUT buffer.

Generating data directly into the LUT buffer

Direct LUT data generation

You can generate general data directly into a LUT buffer, using *MgenLutRamp()* or *MgenLutFunction()*.

The *MgenLutRamp()* command generates a value for each LUT index within the specified index range. The index range together with the start and end values determine the increment.

The increment

If the increment is positive, *MgenLutRamp()* generates a ramp. If the increment is negative, the command generates an inverse ramp. If the increment is equal to zero, it loads the entire LUT range with the given start value.

The *MgenLutFunction()* command generates data within the specified LUT buffer area according to a specified function. The functions available are: M_LOG, M_EXP, M_SIN, M_COS, M_TAN, and M_QUAD. The LUT index and the start x value are used as the x value in the equation.

The *MimHistogramEqualize()* command can be used to create a LUT for intensity correction.

Color LUTs

When generating data in a color LUT buffer, each row of each color band is loaded with the same data. For example, for an RGB LUT, the red, green, and blue bands of the LUT are loaded with the same data.

To load each color band with different data, you would have to generate the data into three separate one-dimensional LUT buffers, then copy each buffer to the appropriate color band of the color LUT buffer, using *MbufCopyColor()*.

Loading LUTs with precalculated data

More complex LUTs

There are several ways to generate more complex LUTs. Most of these, however, involve pre-calculating the data, then loading it into the LUT buffer:

- Calculate data, using your Host system, and then load it into the LUT, using *MbufPut()*, *MbufPut1d()*, or *MbufPutColor()*.
- Generate data into another data buffer, using MIL commands other than *MgenLutRamp()* (for example, using the *MimArith()* command and perhaps the histogram of the image), then copy the data to the LUT buffer, using *MbufCopy()* or *MbufCopyColor()*.
- Load previously saved LUT data from disk to the LUT buffer (*MbufLoad()*). Note, when loading data from disk, there should be enough data for each dimension of the LUT buffer.
- Restore a previously saved LUT, using *MbufRestore()*. Note, this command actually performs the LUT buffer allocation.

Using LUTs

In MIL, LUTs can be used in different circumstances:

- when performing certain processing operations
- when displaying data (if supported by hardware)
- when acquiring data from a digitizer (if supported by hardware)

In each of these cases, if you want only a certain portion or palette of the LUT to be used, allocate the palette as a child buffer, and then specify the child LUT buffer identifier instead of its parent.

Refer to the documentation accompanying your target system device to determine under what circumstances it supports LUTs.

Processing using LUTs

LUT buffer parameter

A LUT buffer identifier parameter is included in all commands that process using LUTs.

Displaying using LUTs

When you want to map a displayable image buffer through a LUT prior to displaying it, you need to associate the LUT buffer with the display, using *MdispLut()*. If this feature is supported by the hardware, it allows you to adjust the color contrast and intensity upon display without affecting the actual image data in memory.

The LUT buffer must match the pixel depth, and should either have the same number of color bands as the display or have a single color band. In the case of a single band, the same data is loaded into each of the display color LUTs.

Monochromatic effect

If you associate a one-band LUT buffer with a display, the same data is loaded in each output channel LUT, and the same data is routed to each output channel LUT. This produces a monochromatic effect when displaying a single-band image.

Pseudo-color effect

If you associate a three-band color LUT buffer (RGB) with a display, each LUT buffer color band is loaded in the corresponding output channel LUT. When displaying a single-band image, the same data is sent to each LUT. This produces a pseudo-color effect on the display .

True color effect

As mentioned above, if you associate a one-band LUT buffer with a display, the same LUT buffer data is loaded in each of the available output channel LUTs upon display. Although the same LUT values are used, you obtain a true color effect upon display of a color image because, typically, each image color band does not contain the same data. You generally want this image and LUT configuration when performing gamma correction to compensate for your monitor.

Finally, as is expected, associating a three-band color LUT with a display creates a true-color effect upon display of a color image.

Displaying image buffers with an associated LUT is further discussed in *Chapter 18: Displaying an image*.

LUTs and digitizers

Associating a LUT to a digitizer

Using MIL, you can map data from a digitizer through LUTs during image acquisition (if the device supports a LUT) . This requires that you associate the LUT to the digitizer, using *MdigLut()*. The LUT buffer must match the pixel depth of the device. In addition, it should either have the same number of color bands as the digitizer or have a single color band.

Chapter 18: Displaying an image

This chapter discusses the display of image buffers, in detail. It shows you how to display several images simultaneously, and discusses some of the special effects that can be applied to a displayable image buffer.

Displaying an image

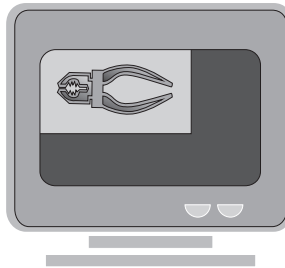
MIL is platform independent. If your system is not using an imaging board with a display section, MIL will use your VGA for display purposes.

Displayable image buffers

To display an image buffer, the buffer must have been allocated with a displayable attribute (`M_DISP`). In addition, a display must have been allocated, using *MdispAlloc()* or *MappAllocDefault()*. Both the buffer and the display must have been allocated on the same system.

Selecting a buffer for display

Once the buffer and the display have been allocated, use *MdispSelect()* to select the image buffer for display. The buffer is displayed at the top-left corner of the screen or in a dedicated window. If the specified image buffer is smaller in size than the display, the border outside the buffer is blanked out. If the specified image buffer is larger in size than the display, the right and bottom part of the buffer, the part that exceeds the display size, is not displayed.



If you want to display only one band of a three-band color buffer, you must first allocate a two-dimensional displayable image buffer and copy the required band into it using *MbufCopyColor()*. You can then display this buffer.

Frame buffers

This manual uses the term frame buffer to refer to display memory. The number of available frame buffer surfaces depends on the system you are using. Matrox imaging boards that have a display section typically have two frame buffer surfaces: a dedicated or dynamically allocated main (underlay)

surface and an overlay (VGA) surface. Separate VGA boards typically have only one frame buffer surface, a VGA frame buffer.

Display configuration

MIL supports various display configurations which use combinations of imaging boards with display sections, separate VGAs, and multiple screens. Some of these configurations might not be supported on your system, therefore it is important that you are aware of your system's hardware restrictions when allocating a display in MIL.

Single-screen configuration

The single-screen configuration is a display configuration in which a single board is used both as a VGA to display the user interface (for example, the Windows desktop) and for the display of images. Both the user interface and images are viewed on a single screen. When using an imaging board with a display section in this configuration, the VGA controls the overlay (VGA) frame buffer.

This configuration is supported on systems using an imaging board with a display section and those which use a separate VGA. In other words, this configuration is supported on all systems.

Dual-screen configuration

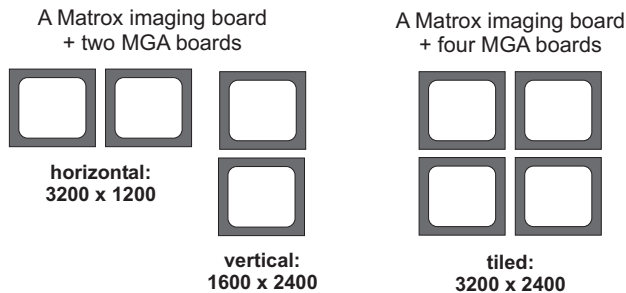
The dual-screen configuration is a display configuration that consists of a separate VGA board for main user-interface display (for example, the Windows desktop) and a Matrox imaging board for image display. This configuration is only supported on systems using an imaging board with a display section.

In this configuration, you can override the default and have images displayed on the same screen as your Windows desktop, so that the display is essentially running in a single-screen configuration. To do so, set the initialization flag for *MdispAlloc()* to *M_WINDOWED*; this operation is discussed later.

- ❖ To configure your Matrox imaging board with a display section in dual-screen mode, see the installation manual of your board to install the board appropriately.

Multi-head display configuration

If you are running Windows NT, you can run in a multi-head display configuration. This configuration is a multi-board configuration that uses a combination of Matrox imaging boards and/or Matrox MGA boards (up to 4 boards). A multi-head display configuration creates one large Windows desktop across multiple screens, in a horizontal, vertical, or tiled fashion.



To run a multi-head display, click on **List All Modes...** in your Windows Display utility (Control Panel) and choose a dual-sized desktop area (for example, 3200x1200) from the list of available resolutions. Note that in a multi-head display configuration, your monitor settings should be compatible with your least-capable monitor.

Display modes and the display window

There are two display modes available, depending on your system's configuration:

- **Windowed mode** (M_WINDOWED).
- **Non-windowed mode** (M_NON_WINDOWED).

You must select one of these modes when allocating a display with *MdispAlloc()*. These modes are described below.

Displaying in windowed-mode

A windowed-mode display (M_WINDOWED) is displayed in its own window. The window is tracked and updated with the image buffer selected on the display; that is, if the window moves or is occluded, the window is updated with the image buffer accordingly.

In windowed mode, multiple windowed-mode displays can be allocated and selected for display; therefore, the display device number should always be set to M_DEFAULT.

This mode is the default allocation mode in a single-screen configuration (M_DEFAULT). If your board has a display section and you are using it in a dual-screen configuration, you can still choose not to use it, and display an image, even a live grabbed image, in windowed mode. In this case, the display is on your Windows desktop.

In windowed-mode, MIL does not communicate directly with the VGA, but uses the normal Windows mechanisms (Windows API functions and extensions) to display images. In other words, it allocates image buffers in a Windows Device Independent BITMAP (DIB or DirectDraw surface) and loads LUT buffers into Windows logical palettes (refer to the Microsoft SDK Programming Guide for information on Windows DIBs, DirectDraw surface, and logical palettes).

Displaying in non-windowed mode

A non-windowed mode (M_NON_WINDOWED) display has no window associated with it. You are responsible for moving and tracking this type of display, if required.

The buffer, selected on the display, is displayed at the top-left corner of the screen. On boards with two dedicated frame buffers, this buffer is actually displayed from the main (underlay) frame buffer surface, and is only visible wherever the overlay (VGA) is set to the keying color (by default, 0).

In this mode, only one MIL display can be allocated and selected for display. This is the default configuration in dual-screen mode.

Display size and depth

In a single-screen configuration, you determine the display format (size and depth) of the overlay frame buffers using the Windows Display utility (Control Panel); in this case, the display format of the MIL display has no effect and should be set to M_DEFAULT. In dual-screen configuration, the display format or video configuration format (VCF) of the selected display determines the display format of the frame buffers.

If the display section has two dedicated frame buffers, a main (underlay) frame buffer and an overlay (VGA) frame buffer, both surfaces are configured to the same size.

In windowed mode, when you select a buffer to a display, Windows will create a display of the same size as the buffer, unless such a display cannot fit in the Windows desktop.

In non-windowed mode, MIL will create a display of the same size as the display format of the frame buffers.

Displaying buffers of different data depths

Displayable buffers usually have a depth of 8 bits (or 3-band 8 bits in the case of color images). If you are in windowed mode, you can display images of other depths (for example, 1-bit or 16-bit images). By using *MdispControl()* with the M_VIEW_MODE control type, you can control the way such buffers are actually being displayed.

The M_VIEW_MODE control type provides three modes of displaying non 8-bit images:

- The M_BIT_SHIFT setting will bit shift the pixel values of the image by the specified number of bits upon updating the display.

- The `M_AUTO_SCALE` setting remaps the pixel values to the display such that the minimum and maximum values in the image (not the full range of the buffer) are set to 0 and 255, respectively. If the image buffer contains a single value, its corresponding displayed value is determined by linearly re-mapping the full range of the buffer (for example, 0 to 64K) to 0 through 255.
- The `M_MULTI_BYTES` setting is primarily useful when grabbing from a multi-tap camera. This setting displays each byte of the image in separate display pixels. For instance, each pixel of a 16-bit image will occupy two consecutive display pixels; each pixel of a 32-bit image will occupy four consecutive display pixels.

The default display mode (`M_DEFAULT`) will automatically select the appropriate mode, depending on the image depth.

Removing a buffer from the display

After displaying an image buffer, you can remove it from the display and close the associated window (in windowed mode) or leave the display blank (in non-windowed mode), using *MdispDeselect()*. To display a different image buffer, you are not required to remove the current buffer from the display; selecting another buffer for display automatically updates the display with the new buffer.

You can only remove the entire image buffer from the display. That is, when displaying a parent buffer, you cannot remove one of its child buffers from the display.

Once you have finished using a display, you should free it, using *MdispFree()*. If a displayed buffer is freed, the buffer is either automatically removed from the display (in windowed mode) or is left blank (in non-windowed mode).

Displaying multiple buffers

MdispSelect() only allows you to view one buffer at a time in a display. However, in windowed mode, you can use many displays to view more than one buffer at a time. In non-windowed mode, you can view more than one buffer at a time using child buffers. For example, you can display the source and destination buffers of an operation, using the following steps:

1. Allocate a large displayable buffer (*MbufAlloc2d()* or *MbufAllocColor()*). This buffer will be known as the parent buffer.
2. Allocate two non-overlapping child buffers within it (*MbufChild2d()* or *MbufChildColor()*).
3. Select the parent buffer for display (*MdispSelect()*).
4. Use one of the child buffers as the source image buffer and the other as a destination image buffer of the operation.

An example...

The following portion of MIL code shows how to display multiple buffers in a single display. The required portion of the cell image, *cell.mim*, is loaded into a child of a displayable buffer and then used as the source of a binarizing operation. The result is stored in another child of the same displayable buffer.


```

/* File name: mmultdis.c
 * Synopsis: This program shows how to display more than one
 *           image buffer at a time on a single display. It
 *           allocates a displayable image buffer, allocates
 *           two child buffers from it, and then uses the child
 *           buffers as the source and destination of a binarizing
 *           operation.
 *
 *           Note: The display will be zoomed if the system's
 *           display supports it.
 */

#include <stdio.h>
#include <stdlib.h>
#include <mil.h>

/* MIL image file name. */
#define IMAGE_FILE          M_IMAGE_PATH "cell.mim"

/* MIL image file specifications. */
#define IMAGE_WIDTH         128L
#define IMAGE_HEIGHT        240L
#define IMAGE_TYPE          8L+M_UNSIGNED
#define IMAGE_THRESHOLD_VALUE 128L
#define ZOOM_VALUE          2L

void main(void)
{
    MIL_ID MilApplication, /* Application identifier.          */
    MilSystem,             /* System identifier.          */
    MilDisplay,            /* Display identifier.         */
    MilParentImage,        /* Image buffer identifier.    */
    MilSrcImage,           /* Source image buffer identifier. */
    MilDstImage;           /* Destination image buffer identifier. */

    /* Allocate the defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem,
                     &MilDisplay, M_NULL, M_NULL);

    /* Allocate a display image buffer. */
    MbufAlloc2d(MilSystem, IMAGE_WIDTH*2, IMAGE_HEIGHT,
                IMAGE_TYPE, M_IMAGE+M_DISP+M_PROC, &MilParentImage);

    /* Allocate two child buffers from the displayable parent buffer. */
    MbufChild2d(MilParentImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT,
                &MilSrcImage);
    MbufChild2d(MilParentImage, IMAGE_WIDTH, 0L, IMAGE_WIDTH, IMAGE_HEIGHT,
                &MilDstImage);

    /* Clear the parent buffer. */
    MbufClear(MilParentImage, 0L);

    /* Display the parent buffer. */
    MdispSelect(MilDisplay, MilParentImage);

```

(cont...)

```

/* Load the entire source image into the source child buffer. */
MbufLoad(IMAGE_FILE, MilSrcImage);

/* Binarize the source child buffer into the destination child buffer. */
MimBinarize(MilSrcImage, MilDstImage, M_GREATER_OR_EQUAL,
            IMAGE_THRESHOLD_VALUE, M_NULL);

/* Report on the Host screen what is being displayed. */
printf("A binarizing operation was performed on the child buffer on the\n");
printf("left side of the display, and the result is being displayed in\n");
printf("the child buffer on the right side of the display.\n");
printf("Press <Enter> to continue.\n\n");
getchar();

/* Report on the Host display what is being displayed. */
printf("Display zoomed by %ld in X and Y (if supported).\n", ZOOM_VALUE);

/* Zoom both child buffers by zooming the display. */
MdispZoom(MilDisplay, ZOOM_VALUE, ZOOM_VALUE);

/* Wait for a key */
printf("Press <Enter> to end.\n");
getchar();

/* Close the display. */
MdispDeselect(MilDisplay, MILParentImage);

/* Free all allocations. */
MbufFree(MilDstImage);
MbufFree(MilSrcImage);
MbufFree(MilParentImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);
}

```

Panning, scrolling, and zooming

At times, your image buffer might be larger than the display, or have details that are too fine or too small to see. Display effects can be associated with the display to view specific parts of the image. These effects are panning, scrolling, and zooming. Note that these are only display effects; they do not affect the content of the image buffer.

Panning and scrolling

Panning and scrolling displace an image horizontally or vertically, respectively, on the display. You can pan and scroll your image to display the appropriate location at the top-left corner of the window (in windowed mode) or screen (in non-windowed mode), using *MdispPan()*. Note, in non-windowed mode, to display the image at another location on the display, you must create a large displayable image buffer, display it, and then allocate and use a child buffer at the required location on the display.

Zooming

Zooming is the horizontal and/or vertical replication of each pixel by the given integer factor. You can zoom the display by an integer factor using *MdispZoom()*. Note that zooming by a large factor might cause a "blocky" effect. In windowed mode, you can also reduce the size of an image on the display. To do so, pass a negative zoom factor to *MdispZoom()*; this functionality is not supported in non-windowed mode.

In the *mmultdis.c* example, the source and destination image buffer dimensions are rather small, so the parent buffer is zoomed by a factor of 2. This is achieved with the line following the binarizing operation:

```
MdispZoom(MilDisplay, ZOOM_VALUE, ZOOM_VALUE);
```

Annotating the displayed image non-destructively

In windowed mode

In windowed mode, you can annotate the displayed image non-destructively using MIL's overlay-display mechanism.

To make use of this functionality in windowed mode, do the following:

1. Enable the overlay-display mechanism, using the following function call:

```
MdispControl(DisplayID, M_WINDOW_OVR_WRITE, M_ENABLE)
```

2. Select a buffer to the display:

```
MdispSelect(DisplayID, ImageBufId)
```

Since the overlay-display mechanism is enabled, this will not only display the selected image, but it will associate a temporary overlay buffer with the display. This overlay buffer will annotate the underlying image with an effect called keying, which makes portions of the overlay buffer transparent so that underlying areas of the displayed image show through. Therefore, anything that you draw in this buffer will annotate the image selected to the display. Note that when you select another image to the display, another temporary overlay buffer is created.

3. To access the overlay buffer, use the following call to determine the identifier of the buffer:

```
MdispInquire(DisplayID, M_WINDOW_OVR_BUF_ID, &OverlayBufferID)
```

This overlay buffer will have the same number of bands as the buffer selected to the display.

4. Draw into the display's overlay buffer with the appropriate graphics function (*Mgra...()*). For example, to write text in the overlay buffer, use *MgraText()*.

In non-windowed mode

To make use of this functionality in non-windowed mode, follow the same steps. However, in this mode, when the overlay-display mechanism is enabled, the display is associated with a temporary overlay buffer immediately. This overlay

buffer is also the same size as the display. When selecting another buffer to the display, the overlay buffer remains the same.

Using the overlay

If available, whenever it is most efficient, both the underlay and overlay frame buffer surfaces are used to annotate the displayed image.

If your board does not have two frame buffer surfaces, a simulated version of the overlay effect is produced. This means that the display update will be slower and a continuous grab operation will appear only in pseudo-live, due to the additional operation needed to combine the grabbed image with the overlay buffer. However combined, the actual buffer selected on the display is not overwritten by the content of the overlay buffer.

Keying

When allocating a display (*MdispAlloc()*), keying is automatically enabled, if required, and the keying color is automatically set to a default color (generally appropriate).

If required, select another keying color with *MdispOverlayKey()*. If you are using an 8-bit display resolution (256 colors), you can set the color to a value between 0 and 255. If you are using a non 8-bit display resolution (15-bit, 16-bit, 24-bit, or 32-bit), call the macro *M_RGB888* and specify the RGB value, for example, as follows:

```
MdispOverlayKey(..., M_RGB888(20,32,24), ...).
```

When the overlay buffer is created, it is cleared to the effective keying color. If the keying color is changed after the overlay buffer is created, it will not be cleared.

The following portion of MIL code shows the enabling of the overlay on the display, the inquiring of the overlay buffer identifier, and the display of text in the overlay (see also, *mdispovr.c*). It assumes that an image has been selected to the display.

```

/* Enable overlay effects on top of the display buffer */
MdispControl(MilDisplay, M_WINDOW_OVR_WRITE, M_ENABLE);

/* Inquire the identifier of the overlay buffer associated with the
 * displayed buffer.
 */
MdispInquire(MilDisplay, M_OVR_BUF_ID, &MilOverlayImage);

/* Print a string in the overlay buffer to appear over the displayed
 * image buffer.
 */
MgraText(M_DEFAULT, MilOverlayImage, 0, 0, " - MIL Overlay Text - ");

```

*Forcing the display in
the overlay*

Although not commonly done, if using an imaging board that has a display section, you can allocate a MIL display that is only in the overlay (VGA) frame buffer surface, if the display is allocated with the M_OVR attribute. When in non-windowed mode and using an imaging board with a display section, your buffer must be allocated with an M_IMAGE+M_OVR+... attribute before it can be selected to an M_OVR display.

Using GDI annotations

If the display has been selected, you can also annotate the displayed buffer using Windows GDI annotations. Use one of the following methods:

- Allocate a Windows display device context (DC) for drawing in the displayed image buffer. To do so, use *MbufControl()* with M_WINDOW_DC_ALLOC. Inquire the identifier of this context using *MbufInquire()* with M_WINDOW_DC. Then, use this DC with Windows GDI function calls.

The buffer which you are annotating must be internally stored in M_DIB or M_DDRAW format, and cannot be a child buffer.

You can create a DC for either the image buffer or the overlay buffer of the display. Note that if you create a DC for the image buffer and then draw using this DC, drawing will be destructive (that is, the data of the image buffer is actually changed).

When either buffer is changed, signal MIL by calling *MbufControl*(..., M_MODIFIED,...).

This method avoids a flickering display when drawing.

- Inquire the display's window handle using *MdispInquire()* with M_WINDOW_HANDLE. Pass the window handle to the Windows *GetDC()* function to get a Windows display device context (DC). Then, paint the annotations with GDI functions from a function hooked to the display update event (*MdispHookFunction()*), that is, paint each time the MIL display is modified.

Note that drawing using this method is non-destructive (that is, the actual data of the image buffer is not changed).

The following portion of MIL code shows the creation of the device context of the overlay buffer, the inquiring of the device context, and the drawing and writing in the overlay buffer (see also, *mdispovr.c*).

```

HDC    hCustomDC;
HPEN   hpen, hpenOld;
char   chText[80];

/* Create a device context to draw in the overlay buffer with GDI. */
MbufControl(MilOverlayImage, M_WINDOW_DC_ALLOC, M_DEFAULT);

/* Inquire the device context. */
hCustomDC = ((HDC)MbufInquire(MilOverlayImage, M_WINDOW_DC, M_NULL));
if (hCustomDC)
{
    /* Create a blue pen. */
    hpen=CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
    hpenOld = SelectObject(hCustomDC,hpen);

    /* Draw a cross in the overlay buffer. */
    MoveToEx(hCustomDC,0,ImageHeight/2,NULL);
    LineTo(hCustomDC,ImageWidth,ImageHeight/2);
    MoveToEx(hCustomDC,ImageWidth/2,0,NULL);
    LineTo(hCustomDC,ImageWidth/2,ImageHeight);

    /* Write text in the overlay buffer. */
    strcpy(chText, "GDI Overlay Text ");
    SetTextColor(hCustomDC,RGB(0, 0, 255));
    TextOut(hCustomDC,ImageWidth*3/18,ImageHeight*4/6, chText,
            strlen(chText));
    SetTextColor(hCustomDC,RGB(255, 0, 0));
    TextOut(hCustomDC,ImageWidth*12/18,ImageHeight*4/6, chText,
            strlen(chText));

    /* Deselect and destroy the blue pen. */
    SelectObject(hCustomDC,hpenOld);
    DeleteObject(hpen);
}

/* Delete created device context. */
MbufControl(MilOverlayImage, M_WINDOW_DC_FREE, M_DEFAULT);

/* Signal MIL that the overlay buffer was modified. */
MbufControl(MilOverlayImage, M_MODIFIED, M_DEFAULT);

```

Displaying an image in a user-defined window

Selecting a buffer into a specific display window

Under Windows, you can display a specific image buffer in a user-defined window, using *MdispSelectWindow()*. For best results, the display must have the same resolution as the image buffer depth. The window must be created with the Windows API functions. If the defined window is of different dimension than the image buffer, any excess window area will be left untouched or any excess image area will be cropped.

By default, under Windows, images are displayed in the default window, using *MdispSelect()*. This function dynamically creates a window in the Windows desktop for the specified display, if the display is not already selected. The created window respects any window control that has been associated with the display using an *Mdisp...()* function.

Using the *MdispSelectWindow()* function

The *MdispSelectWindow()* function is similar to *MdispSelect()*, except that it allows you to specify a handle to a user-defined window, rather than displaying into a MIL created window. This window is automatically refreshed when the display is modified (for example, when the image data is modified). You can use *MdispDeselect()* to deselect the image from the display.

An example...

The following portion of MIL code from the *mwindisp.c* example shows you how to display an image in a user-defined window, grab into such a window, and remove the image from the display.

```
/* File name: mwindisp.c
 *
 * Synopsis: This program displays a welcoming message in a user-
 *           defined window and grabs into it (if supported). It uses
 *           the MIL system and the MdispSelectWindow() function
 *           to display the MIL buffer in a user created client window.
 *
 *           Use MdispDeselect() to remove the selected image buffer
 *           from the display.
 */

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <windows.h>
#include <mil.h>
#include <mwinmil.h>
#include <wingdi.h>

#define BUFFERSIZE_X      640
#define BUFFERSIZE_Y      480
#define BUFFERSIZE_BAND   1
#define MAX_PATH_NAME_LEN 256

/* Prototypes */
void MilApplication(HWND UserWindowHandle);
void MilApplicationPaint(HWND UserWindowHandle);
/*****
 */
 * Name:      MilApplication()
 *
 * synopsis:  This function is the core of the MIL application that
 *            will be executed when the "Start" menu item of this
 *            Windows program will be selected. See WinMain() below
 *            for the program entry point.
 *
 *            It will use MIL to display a welcoming message in the
 *            specified user window and to grab in it if it is supported
 *            by the target system.
 */
```

(cont...)

```

void MilApplication(HWND UserWindowHandle)
{
    /* MIL variables */
    MIL_ID MilApplication, /* MIL Application identifier. */
    MilSystem, /* MIL System identifier. */
    MilDisplay, /* MIL Display identifier. */
    MilDigitizer, /* MIL Digitizer identifier. */
    MilImage; /* MIL Image buffer identifier. */

    long BufSizeX;
    long BufSizeY;
    long BufSizeBand;

    /* Allocate a MIL application. */
    MappAlloc(M_DEFAULT, &MilApplication);

    /* Allocate a MIL system. */
    MsysAlloc(M_DEF_SYSTEM_TYPE, M_DEV0, M_DEFAULT, &MilSystem);

    /* Allocate a MIL display. */
    MdispAlloc(MilSystem, M_DEV0, M_DEF_DISPLAY_FORMAT, M_DEFAULT,
    &MilDisplay);

    /* Allocate a MIL digitizer if supported and sets the target image size.*/
    if (MsysInquire(MilSystem, M_DIGITIZER_NUM, M_NULL) > 0)
    {
        MdigAlloc(MilSystem, M_DEV0, M_DEF_DIGITIZER_FORMAT, M_DEFAULT,
        &MilDigitizer);
        MdigInquire(MilDigitizer, M_SIZE_X, &BufSizeX);
        MdigInquire(MilDigitizer, M_SIZE_Y, &BufSizeY);
        MdigInquire(MilDigitizer, M_SIZE_BAND, &BufSizeBand);
    }
    else
    {
        MilDigitizer = M_NULL;
        BufSizeX = BUFFERSIZE_X;
        BufSizeY = BUFFERSIZE_Y;
        BufSizeBand = BUFFERSIZE_BAND;
    }

    /* Do not allow example to run in dual-screen mode */
    if (MdispInquire(MilDisplay, M_DISPLAY_MODE, M_NULL) != M_WINDOWED)
    {
        MessageBox(0, "This example does not run in dual-screen mode.",
        "MIL application example",
        MB_APPLMODAL | MB_ICONEXCLAMATION );
        goto end;
    }
}

```

(cont...)

```

/* Allocate a MIL buffer. */
MbufAllocColor(MilSystem, BufSizeBand, BufSizeX, BufSizeY, 8+M_UNSIGNED,
(MilDigitizer? M_IMAGE+M_DISP+M_GRAB : M_IMAGE+M_DISP), &MilImage);

/* Clear the buffer */
MbufClear(MilImage,0);

/* Select the MIL buffer to be displayed in the user specified window */
MdispSelectWindow(MilDisplay, MilImage, UserWindowHandle);

/* Print a string in the image buffer using MIL.
   Note: After a MIL command writing in a MIL buffer, the display
   will automatically update the window given to MdispSelectWindow().
*/

MgraText(M_DEFAULT, MilImage, (BufSizeX/8)*3, BufSizeY/2,
" ..... ");
MgraText(M_DEFAULT, MilImage, (BufSizeX/8)*3, BufSizeY/2+25,
" Welcome to MIL !!! ");
MgraText(M_DEFAULT, MilImage, (BufSizeX/8)*3, BufSizeY/2+50,
" ..... ");

/* Windows code to open a message box to wait a key. */
MessageBox(0, "" "Welcome to MIL !!!" " was printed",
"MIL application example",
MB_APPLMODAL | MB_ICONEXCLAMATION );

/* Grab in the user window if supported by the system. */
if (MilDigitizer)
{
    /* Grab continuously. */
    MdigGrabContinuous(MilDigitizer, MilImage);

    /* Windows code to open a message box to wait a key. */
    MessageBox(0, "Continuous grab in progress",
"MIL application example",
MB_APPLMODAL | MB_ICONEXCLAMATION );

    /* Stop continuous grab. */
    MdigHalt(MilDigitizer);
}

/* Deselect the MIL buffer from the display. */
MdispDeselect(MilDisplay, MilImage);

/* Free allocated objects. */
MbufFree(MilImage);

end:

MdispFree(MilDisplay);
if (MilDigitizer)
    MdigFree(MilDigitizer);
MsysFree(MilSystem);
MappFree(MilApplication);
}

```

LUTs and changing the displayed colors or gray levels

In general, when displaying in a 256-color display resolution, images are mapped through physical output LUTs on display. These LUTs are available and programmable so that you can achieve the best display effect for your images since not all colors are available in this resolution. Note that when displaying in a non-256-color display resolution, MIL can simulate a display LUT in software for 8-bit images.

How images are mapped through the physical output LUTs

By default in windowed mode, when displaying 8-bit images, MIL tries to use the image's pixel values to address the physical output LUTs. When displaying color images, MIL will search the physical output LUTs for the entry that best matches the color of the image's pixels being displayed. It then creates a translation table for the image. This table is used to convert each pixel value upon display to a value that will address the appropriate physical output LUT entry.

In non-windowed mode, MIL uses the image's pixel values to address the physical output LUTs.

Palette versus physical output LUTs

The actual programming of the physical output LUTs is handled by MIL in one of two ways. In windowed mode, MIL indirectly programs the physical output LUTs through the use of a Windows palette. In non-windowed mode, MIL programs the physical output LUTs directly.

Default palette settings in windowed mode

By default in windowed mode, MIL provides a good default logical palette for the realization of the physical output LUTs (*MdispLut(..., M_DEFAULT, ...)*). MIL takes into consideration the displayed image, the Windows display driver used, and the VGA physical output LUT capabilities, and produces the best "portability versus visual quality" compromise possible.

Default physical output LUT settings in non-windowed mode

By default in non-windowed mode, MIL generates a ramp in the physical output LUTs, which uses the full range of available intensities (*MdispLut(..., M_DEFAULT, ...)*). This type of mapping is also referred to as a pass-through LUT mapping (or transparent LUT mapping).

Changing the default LUT values

In general, if the default LUT values are not appropriate for your application, you can change the LUT values to control the displayed colors or gray levels of an image. Some situations that might require special display effects are:

- When displaying monochrome images, you might want to view the images with each gray intensity in a different color. For example, you can associate specific colors to ranges of temperatures obtained by an infra-red camera.
- When displaying monochrome images, you might want to invert the image values. For example, when grabbing a film negative, you can display the film as it will be printed.
- In windowed mode, when displaying color images under a 256-color display driver resolution, you might want to reduce the loss of color resolution. For example, when displaying a color image with many shades of red, you might want to select a LUT so that all shades of the image are represented.

To change the LUT values, associate a pseudo-color or custom LUT buffer to the display with *MdispLut()* or to the displayed image buffer with *MbufControl()*. Please note that LUT buffers used for display have the following restrictions:

- If the LUT buffer values are changed while the image is selected on the display, the changes will not take effect.
- The LUT buffer will not be used when displaying a 3-band 8-bit image under a non-8-bit display resolution.
- A LUT buffer cannot be associated to a display that belongs to a system using an imaging board with an on-board display section, unless that display has been allocated with the *M_OVR* initialization parameter.
- The LUT buffer must have one or three bands. Note that the number of LUT buffer entries must be the same as the maximum number of intensities that can be represented in the displayed buffer. In other words, if you want to invert an 8-bit grayscale image (that is, an image that can have 256 intensities), your LUT must also have 256 entries.

You can use *MdispInquire()* to obtain information about the physical output LUTs of a display.

Associating a pseudo-color LUT

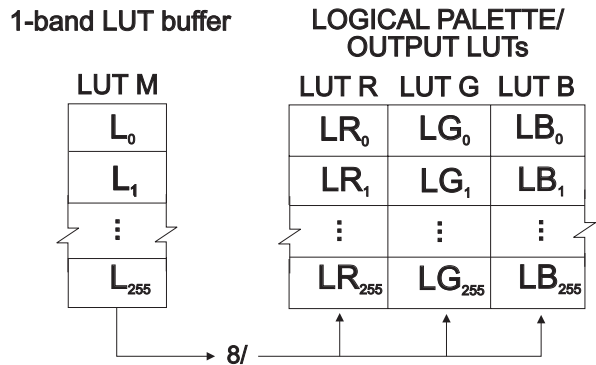
To view an 8-bit image buffer with each gray intensity in a different color, associate the default pseudo-color LUT buffer (M_PSEUDO) with the display of the image. In windowed mode, the data is loaded in each component of the logical palette. In non-windowed mode, the data is loaded into the physical output LUTs of the display.

A 1-band custom LUT buffer

To invert the values of an 8-bit image on display, you would need physical output LUTs that map each value to the maximum pixel value minus the current pixel value. To do so:

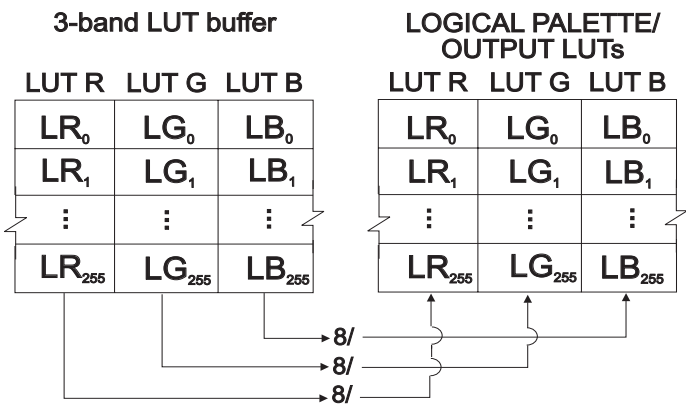
1. Allocate a one-band LUT buffer (*MbufAlloc1d()*).
2. Generate the data into the buffer, using *MgenLutRamp()* or load the data into it, using *MbufPut()*. The depth of the LUT buffer data must be 8-bits.
3. Associate the LUT buffer with the required display using *MdispLut()*, or to a particular image with *MbufControl()*.

If you associate a one-band LUT buffer with the display or buffer in windowed mode, the same data is loaded into each component of the logical palette. In non-windowed mode, the same data is loaded in each available display physical output LUT.



A 3-band custom LUT buffer

To reduce the loss of color resolution when displaying an image with a specific range of colors, you would need physical output LUTs that contain all the required colors so that when Windows creates a translation table for the image, most colors are mapped to their exact values. Follow the steps for a 1-band LUT buffer, except allocate and load a 3-band LUT buffer instead. When in windowed mode, each band of the LUT buffer is loaded into its corresponding component of the logical palette. In non-windowed mode, each band is loaded in a different display physical output LUT (if a different LUT is available for each display channel).



Different display architectures in windowed mode

Although all MIL display windows have a similar appearance, MIL uses one of three different architectures to make an image buffer visible through a display window. The particular architecture determines the behavior of the display window under specific circumstances. For example, the display architecture will determine how a continuous grab operation will behave when overlapped by another window. Similarly, the display architecture will determine whether or not the Host CPU is used to overlay graphical annotations or images on top of the buffer selected for display.

The three display architectures are listed and discussed below.

- Underlay display
- Overlay/ regular display
- DirectDraw underlay-surface display
- ❖ The type of display architecture that can be used depends on the available hardware. Keep in mind that MIL automatically selects the most appropriate display architecture when a display is allocated (*MdispAlloc()*) with the default (M_DEFAULT) initialization flag.

Underlay display architecture

Under the underlay display architecture, the Windows desktop sits in a dedicated overlay frame buffer surface, whereas the selected MIL buffer(s) sits in a dedicated underlay frame buffer surface. The data of the buffer, selected to a windowed display, is visible through a special hardware keying mechanism.

An underlay display architecture is used only when the video frame buffer is physically split into a main (underlay) and overlay frame buffer. The display resolution sets the size of the overlay and the underlay frame buffer surfaces that are used for on-screen purposes. Accordingly, the amount of video memory required is twice that of the current display resolution. This display architecture is used only when it is more efficient to do so.

The underlay display architecture is used only on Matrox Imaging frame grabbers such as Matrox Corona, Matrox Genesis, and Matrox Pulsar frame grabbers.

Under this display architecture, there are a number of important features:

- *MdigGrabContinuous()* is always performed live with no Host CPU intervention, irrespective of overlapping of the display windows.
- Graphics and video overlay on top of the selected buffer is done with no Host CPU intervention.
- *MdispLut()* is usually not supported.

- *MdispZoom()* is not accelerated by the hardware, which means that it is emulated by the software.
- The underlay surface data-format usually follows the Windows display resolution. This means that the underlay surface data-format is typically in RGB format.

MIL will choose the most appropriate display architecture at the time of display allocation (*MdispAlloc()*). Still, you can use the `M_WINDOWED + M_UND` initialization flag to force this dedicated underlay display architecture, provided that you have the appropriate hardware.

Overlay/regular display architecture

Under a overlay/regular display architecture, image buffers are allocated in a Windows Device Independent Bitmap (DIB or Direct Draw surface) and then Windows handles their display. In addition, LUT buffers are loaded into logical Windows palettes.

An overlay/regular display architecture uses the regular Windows mechanisms (Windows API function and extensions) to display images. In this case, the video frame buffer does not have to meet any special conditions.

Under this display architecture, there are a number of important features:

- *MdigGrabContinuous()* is performed live with no Host CPU intervention when (1) the display format is supported by the frame grabber, (2) the display window is not overlapped by another window, and (3) when there is no overlay. Otherwise, when these conditions are not met, MIL automatically switches to a pseudo-live grab, which uses Host CPU to emulate the grab operation to the display.
- Graphics and video overlay on top of the selected buffer is emulated by the software. Consequently, the graphics overlay makes use of the Host CPU.
- *MdispLut()* is supported. Therefore, it is possible to perform a continuous grab operation in pseudo-color.
- *MdispZoom()* is not accelerated by the hardware, which means that it is emulated by the software.

Although MIL will choose the most appropriate display architecture, at the time of display allocation (*MdispAlloc()*), you can use the `M_WINDOWED + M_OVR` initialization flag to force this overlay display architecture.

DirectDraw underlay-surface display architecture

The DirectDraw underlay display architecture is similar to the underlay display architecture described above. The display resolution sets the size of the overlay frame buffer surface. However, under the DirectDraw underlay-surface display architecture, the VGA display adapter dynamically allocates an underlay surface that is of the same size as the buffer selected to the display. Accordingly, the amount of video memory used for on-screen purposes depends on the size and depth of the image buffer selected for display.

A DirectDraw underlay-surface display architecture can only be used when the VGA display adapter can dynamically allocate an underlay surface, and is only used when it is more efficient to do so.

The DirectDraw underlay-surface display architecture is currently available when using a Matrox frame grabber with a Matrox G200 or Matrox G400 graphics controller such as the Matrox Orion frame grabber. The Cyrix processor's companion chip on the Matrox 4Sight imaging platform also supports the DirectDraw underlay-surface display architecture.

Under this DirectDraw underlay surface display architecture, there are a number of important features:

- *MdigGrabContinuous()* is always performed live with no Host CPU intervention, irrespective of overlapping of the display windows.
- Graphics and video overlay on top of the selected buffer is done with no Host CPU intervention.
- *MdispLut()* is not supported.
- *MdispZoom()* is accelerated by the hardware, which means that there is no Host CPU intervention.

- The underlay surface data format is usually YUV. Typically, this is an advantage because it allows true color images to be displayed even in 256 colors display resolution (if there are no hardware restrictions which apply).

MIL will choose the most appropriate display architecture at the time of display allocation (*MdispAlloc()*). Still, you can use the `M_WINDOWED + M_DDRAW_UND` initialization flag to force this DirectDraw underlay-surface display architecture, provided that you have the appropriate hardware.

Advanced controls for windowed mode

Display types in windowed mode

In windowed mode, when allocating a display (*MdispAlloc()*), you can specify how image buffers are displayed in a 256-color display resolution. There are three types of display initialization:

- **Enhanced** (`M_DISPLAY_ENHANCED`, `M_DISPLAY_8_ENHANCED`, `M_DISPLAY_24_ENHANCED`)
- **Basic with optimization** (`M_DISPLAY_BASIC`, `M_DISPLAY_8_BASIC`, `M_DISPLAY_24_BASIC`)
- **Basic without optimization** (`M_DISPLAY_WINDOWS`, `M_DISPLAY_24_WINDOWS`)

Select both an `M_DISPLAY_8_XXX` and `M_DISPLAY_24_XXX` display initialization to independently control the display of 8-bit and 3-band 8-bit images.

Enhanced

When using an enhanced initialization, the MIL display calls the Microsoft Video for Windows **DrawDIBDraw()** function to display image buffers. This function's use of dithering particularly improves the display of 3-band 8-bit images under 256-color display resolution.

Note, with enhanced initializations, the actual display color values are selected, on a best-match basis, from the logical palette's available display colors. Therefore, effects such as those of an inverse LUT are not possible. This is the default display initialization for an 8-bit 3-band image.

Basic with optimization

When using a basic with optimization initialization, the MIL display calls the Windows API **StretchDIBits()**, **StretchBlt()**, or **DirectDrawBlt()** function to display image buffers. When 8-bit images are displayed, the pixel values are used, as much as possible, to index the physical output LUTs. When 3-band 8-bit images are displayed in an 256-color display resolution, the display uses an algorithm optimized for speed. This algorithm converts 24 bits to 8 bits by taking the most-significant bits of each component: 3 bits each are taken from the red and green components, and 2 bits from the blue. This produces an 8-bit DIB with 3:3:2 RGB values for display; it is these values that are used to address the physical output LUTs. This is the best possible combination when you are not aware of the color content of the image buffer.

Basic without optimization

When using a basic without optimization initialization, the MIL display calls the Windows API **StretchDIBits()**, **StretchBlt()** or **DirectDrawBlt()** function to display image buffers; however no optimization for speed is done when displaying a 3-band 8-bit image in a 256-color display resolution. The display will display such images (color images) on a best-match basis and display 8-bit images using their pixel values to address the physical output LUTs.

This display initialization can result in slow display performance.

Zoom types in windowed mode

In windowed mode, when allocating a display (*MdispAlloc()*), you can specify how image buffers are zoomed in a 256-color display resolution. There are two types of zoom initialization:

- **Enhanced** (M_ZOOM_ENHANCED)
- **Basic** (M_ZOOM_BASIC)

Enhanced

When using an enhanced zoom initialization, the **DrawDIBDraw()** function is called to perform a zoom. Although zooming might be a little slower than using the basic initialization option, it does not alter the dithering quality,

providing a better quality zoom. This option is the default and is only available when an `M_DISPLAY_XXX_ENHANCED` display initialization is used.

Basic

When using a basic zoom initialization, Windows (Windows API functions) is called to perform the zoom. Note, if an `M_DISPLAY_XXX_ENHANCED` display initialization is used, this zoom might alter the quality of the **DrawDIBDraw()** dithering.

Controlling how the LUT buffer is loaded into the Windows palette

When calling *MdispLut()*, MIL will copy the data of each band of the LUT buffer to the corresponding component of the logical palette, without modification. To obtain good results, the specified color values must be carefully selected to provide the best color match upon image display. If the specified values closely match the RGB values that occur frequently in the image to be displayed, very good results can be obtained.

Controlling how the logical palette is loaded into the physical output LUTs

When in windowed mode, you can control how Windows loads the logical palette into the physical output LUTs. If you want the Windows palette manager to use palette optimization when realizing the physical output LUT values from the logical palette, use *MdispControl()* with the `M_WINDOW_PALETTE_NOCOLLAPSE` control type.

The `M_WINDOW_PALETTE_NOCOLLAPSE` control type can be used to control palette optimization in one of two ways:

- The Windows palette manager realizes the physical output LUTs with the best color usage of the logical palette (`M_DISABLE`); this is the default setting.
- The Windows palette manager realizes the physical output LUTs by loading the logical palette "as is" (`M_ENABLE`).

Optimizing the physical output LUTs

When setting the `M_WINDOW_PALETTE_NOCOLLAPSE` control type to `M_DISABLE`, the Windows palette manager attempts the best color usage of the logical palette when realizing the

physical output LUTs. The palette manager tries to map colors from the logical palette into the currently realized physical output LUTs to reduce the number of requested new entries. This reduces the chance of a color occurring more than once in the physical output LUTs.

Realizing the physical output LUTs without modification

When setting the `M_WINDOW_PALETTE_NOCOLLAPSE` control type to `M_ENABLE`, the Windows palette manager loads each component of the logical palette "as is" into the corresponding physical output LUT. This can result in a color occurring more than once in the physical output LUT.

Chapter 19: Generating graphics

This chapter describes the graphics commands that are available with MIL. These consist of drawing and text-writing commands.

MIL and graphics

The MIL package supports basic drawing and text commands that are useful in typical image processing or machine vision applications. These commands could be used, for example, to create a conditional buffer or to annotate an image.

Preparing for graphics

There are two requirements for graphics operations:

- An image buffer in which to perform the operation.
- A set of graphics parameters, referred to as a graphics context, with which to perform the operation.

Graphics context

Allocate a graphics context, using *MgraAlloc()*. Upon allocation, each of the graphics parameters of the graphics context is set to the default (refer to the *MgraAlloc()* command reference description for the defaults). You can change these parameter settings according to your needs.

Different graphics contexts can coexist. Use their identifier to specify which to use or change.

Once a graphics context is no longer required, it should be freed, using *MgraFree()*.

When a MIL application is created, using *MappAlloc()* or *MappAllocDefault()*, a default graphics context is automatically created. It can be used as a normal graphics context by specifying `M_DEFAULT` as the graphics context identifier. Since `M_DEFAULT` is simply another graphics context, you can change its parameter settings according to your needs.

Graphics parameters

There are two basic parameters that apply to graphic objects:

1. **Background color.** This determines the background color of textual graphic objects. The default background color value is zero (typically corresponds to black). You can change this color, using *MgraBackColor()*.
2. **Foreground color.** This determines the color in which graphic objects are drawn or written. The default foreground color value is the highest positive buffer value (typically corresponds to white). You can change this color, using *MgraColor()*.

Selecting colors

A grayscale value can be any integer or floating-point number. If the given value exceeds the range of the possible values that can be stored in each band of the destination buffer, the least significant bits of the value are used.

Clearing the buffer

Once you are satisfied with the graphics parameters, you should determine whether you need to clear the graphics image buffer prior to drawing or writing to it. You can use *MgraClear()* or *MbufClear()* to clear the buffer to a specific color.

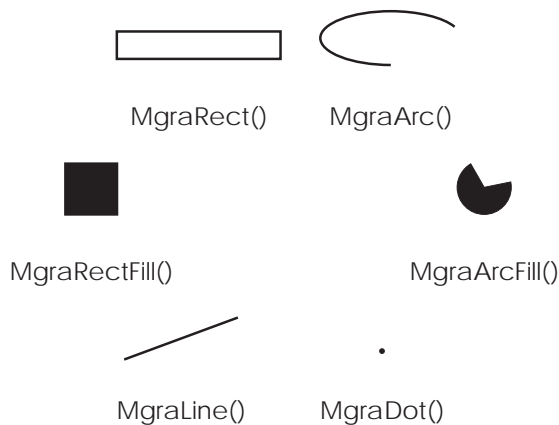
Drawing graphics

With the MIL package, you can draw:

- lines (*MgraLine()*)
- rectangles (*MgraRect()* and *MgraRectFill()*)
- arcs, circles, and ellipses (*MgraArc()* and *MgraArcFill()*)
- dots (*MgraDot()*)

Using *MgraLine()*, *MgraRect()*, *MgraArc()*, or *MgraDot()*, you can draw the outline of most required shapes. The outlines are drawn one pixel wide.

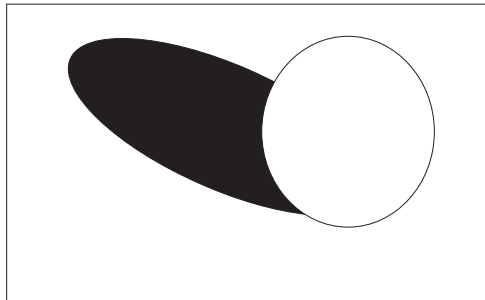
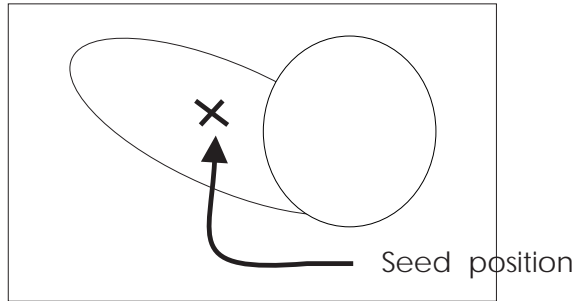
In addition, the MIL package includes *MgraRectFill()* and *MgraArcFill()* so you can draw solid rectangles and arcs.



If you need complex filled-in shapes, draw the outline of the shape and use *MgraFill()* to fill it.

Filling shapes

MgraFill() performs a boundary-type seed fill. It fills an area of the target buffer with the current foreground color, starting from the specified seed position. Filling occurs on adjacent pixels of the same value as the original seed pixel.



Note, any drawing is clipped outside the boundaries of the buffer.

Writing text

You can also write text in the drawing area, using *MgraText()*. This command writes a null-terminated (`\0`) ASCII string at the specified position in a given buffer, using the foreground and background color and current font of the specified graphics context.

When specifying the location at which to write the string, give the top-left corner coordinates of the first character in the string.

(*x*, *y*)



G	o	o	d		m	o	r	n	i	n	g	!		
---	---	---	---	--	---	---	---	---	---	---	---	---	--	--

Although the graphics context specifies a default character font and size, you can change the font and size of this context, using *MgraFont()* and *MgraFontScale()*, respectively. *MgraFont()* provides a set of predefined fonts from which to choose.

Chapter 20 : Grabbing with your digitizer

This chapter discusses the cameras supported with MIL and the control of your digitizers, including the fine-tuning of the input and auto-focusing.

Cameras and input devices

The MIL package supports input from any type of input device supported by the digitizer. Data grabbed from an input device with the digitizer using *MdigGrab()* or *MdigGrabContinuous()*, is stored into an image buffer. For color cameras, you must use color image buffers, with the same number of bands as the incoming data. Note, since most input devices are cameras, they will hereafter be referred to as such.

For a digitizer to be recognized by MIL, it must be allocated on the target system, using *MdigAlloc()* (or *MappAllocDefault()*). The allocation sets up the digitizer to match your camera's data format and to access the active input channel. Once you have finished using a digitizer, you should free it, using *MdigFree()*.

If you often use the same camera and prefer to use *MappAllocDefault()* to set up and initialize your system, you might want to update the *milsetup.h* file to reflect your camera.

When developing an application, it is recommended that you use a simple camera. Once the application is working, switch to a more sophisticated camera, if necessary. This approach makes debugging much easier.

The data format

MdigAlloc() needs the camera's digitizer configuration format (DCF) to perform the digitizer allocation. The DCF defines such parameters as the input frequency and resolution, and will determine limits when grabbing an image.

MIL provides a number of predefined DCFs for the basic cameras supported by your digitizer. Refer to the *MIL/MIL-Lite Board-specific notes* manual for exact settings. MIL also provides some DCF files that you can load if the predefined DCFs don't suit your needs.

Once a digitizer has been allocated, you can use *MdigInquire()* to inquire about its settings.

If you find a DCF file that is appropriate for your video source, but need to adjust some of the more common settings, you can do so directly, without adjusting the file, using the *Mdig...()* commands. For more specialized adjustments, you can adjust the file itself, using Matrox Intellicam.

If you cannot find an appropriate DCF file or have a non-standard input device that does not appear in our list (such as a strobe or trigger device), you can create your own, also using Matrox Intellicam. For more information on Matrox Intellicam, refer to the *Matrox Intellicam User Guide* manual.

If you cannot develop the required DCF using Matrox Intellicam, you should provide the camera specifications to your Matrox Technical Support Engineer. A suitable customized DCF file can then be developed, if your digitizer supports the camera.

The digitizer number

The device number

In addition to the data format, *MdigAlloc()* requires that you specify the digitizer number. The digitizer number specifies the required digitizer, and its rank with respect to other digitizers of the same type (color or monochrome) residing in the same system. Note, if there is only one digitizer on the specified system, you must specify the digitizer number as M_DEV0 or M_DEFAULT.

Multiple cameras

MIL also supports applications that require input from different cameras. In general, you cannot simultaneously activate two cameras, whether or not they are connected to the same digitizer.

The input channel

Most digitizers have several multiplexed input channels, that is they have several channels but can only grab from one of the channels at a time. In this case, if you have a camera that is not connected to the first channel of its digitizer, you must specify the channel, using *MdigChannel()*.

If there are several cameras of the same data format connected to a digitizer, you only need to allocate a digitizer with the DCF of the first camera and use *MdigChannel()* to switch between the others of the same type.

When using different cameras on the same digitizer, a different DCF must be used for each camera. In general, to switch between cameras of different formats, you have to allocate the digitizer with one format, grab, free the digitizer, and then allocate the digitizer again with the second format. Some systems permit virtual digitizers (for example, Matrox Genesis) so that you can allocate several digitizers, specify a channel for each digitizer, and then grab with the appropriate digitizer, without having to free and re-allocate between switches.

Number of frames or fields

To grab a single frame or field, use *MdigGrab()*. The type of scanning used by your camera determines whether you grab fields or frames. With progressive scanning cameras, frames are grabbed. If your camera uses interlaced scanning, fields are grabbed. By default, if your camera uses interlaced scanning, one call to *MdigGrab()* will grab both the odd and even fields.

To grab a series of continuous frames, use *MdigGrabContinuous()*; this function uses the specified digitizer to continuously acquire frames of data until *MdigHalt()* is called.

Note, when grabbing data with *MdigGrab()*, you can specify how many fields or frames to grab using *MdigControl()*, with `M_GRAB_FIELD_NUM` or `M_GRAB_FRAME_NUM`.

Line-scan cameras

If your target digitizer supports it, you can grab from a line-scan camera in the same way you would, for example, an RS-170 type camera. However, you should be aware of how data from these cameras is stored.

When acquiring data from a line-scan camera, each line of each destination buffer band is filled from top to bottom. The operation will only end once the entire buffer has been filled.

Grabbing to the display

Live and pseudo-live continuous grabs

With MIL, you can grab to a displayable buffer selected on a display. If your system is not using an imaging board with a display section, MIL will use your VGA for display purposes. MIL uses one of two methods to transfer when grabbing:

- **Live grab:** MIL grabs directly to the version of the buffer that is physically allocated in the frame buffers (display memory).
- **Pseudo-live grab:** MIL grabs into the Host memory version of the buffer and then updates the version in the frame buffers (display memory).

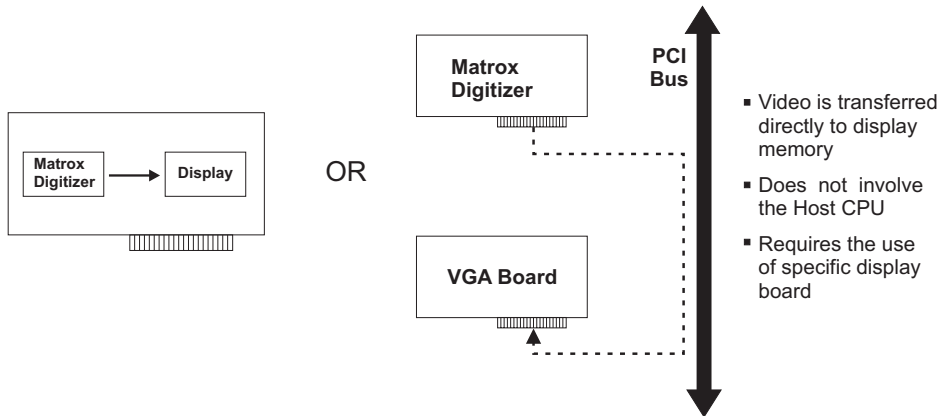
When grabbing, the digitizer (for example, Matrox Meteor-II) always acts as the bus master.

A monoshot grab is always pseudo-live. Grabbing a specific number of frames is also performed pseudo-live (note that under a non-windowed display it is possible to perform a live monoshot grab by allocating your buffer directly on the VGA board with `M_ON_BOARD`).

In general, a continuous grab is live. By default, at the end of the continuous grab, a copy of the last image grabbed is made in the Host memory version of the buffer (or on-board processing memory). This allows the image to be processed. You can override the copy-to-Host behavior, using *MsysControl()* with the `M_LAST_GRAB_IN_TRUE_BUFFER` control type. Note that in this case, the *MdigGrabContinuous()* call will not modify the Host buffer in any way.

Live transfer to the display

The digitizer can generally transfer all grabbed data directly to display memory, when grabbing to an on-board display or when grabbing to a VGA that supports fast linear-memory accesses to its frame buffer.



Pseudo-live transfers to the display

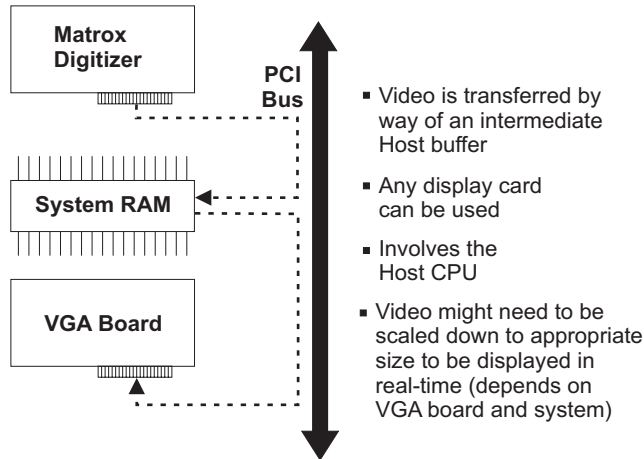
A continuous grab will automatically switch to pseudo-live if:

- Your VGA board does not support fast linear-memory accesses (discussed later in this section).
- The format of the grabbed data is not compatible with your VGA display mode. For example, performing a color grab in 256 color display resolution.
- Your board does not have both an underlay and overlay frame buffer surface and there is a non-rectangular overlap between the display windows on the display device.
- Your board does not have both an underlay and overlay frame buffer surface, DDraw is disabled or you are in multi-head mode, and the grab display window does not have the focus, that is, is not active.
- Your board does not have both an underlay and an overlay frame buffer surface and you are using the display's overlay buffer, that is, have enabled `M_WINDOW_OVR_WRITE` with

MdispControl(). In this case, the grab will be pseudo-live because an additional operation is required to combine a grabbed image with a simulated version of the overlay.

- You are in multi-head mode and the display window occupies more than one screen.

MIL transparently performs pseudo-live grabs:



By default, when a continuous grab switches to pseudo-live, it will transparently double buffer the grab in Host memory. That is, while the digitizer is grabbing one frame into a Host buffer, the display driver performs a blit of the previous frame (stored in the temporary Host buffer) to the frame buffers (VGA display memory). Double-buffering can be disabled using *MsysControl()* with `M_DISPLAY_DOUBLE_BUFFERING`.

Pseudo-live transfers will be real time (that is, full frame rate of 30 for NTSC or 25 fps for PAL) if the CPU transfer from the Host buffer to display memory is fast enough. That is, if the blit is taking at most one frame time length. Blit time is affected by the load of the CPU (for example, the number of process threads and the priorities of other boards). You can reduce the load of the CPU in the pseudo-live grab operation by disabling the double buffering operation. However, when double buffering is disabled, only half of the full frame rate can be achieved.

Multi-head mode

In multi-head mode, note that a continuous grab without overlay can be moved from one screen to another and be displayed live when it has the focus. However, a continuous grab with overlay will only be live on the screen attached to the on-board display section; it will switch to pseudo-live on the other screen(s). In both cases, when the window displaying the grab intersects two screens, the grab is pseudo-live.

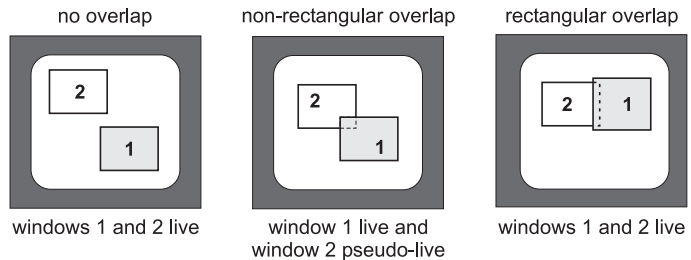
The table below indicates the type of configurations which are supported on particular boards.

Display configuration	Corona	Genesis	Meteor-II	Orion	Pulsar
single-screen or multi-head (windowed mode)	x	x	x	x	x
single- or dual-screen (non-windowed mode)	x	x			x

For more information about your board's transfer capabilities, consult the *MIL/MIL-Lite Board-specific notes* manual.

Window occlusion

When there is no overlap or rectangular overlap of a live grab window, the continuous grab is displayed live. When there is non-rectangular overlap (that is, the displayed portion of the occluded window is no longer rectangular), there is pseudo-live display.



* Window 1 is the active window and window 2 is the grab and display window.

Note that when DDraw is disabled (see *MsysAlloc()*), the continuous grab is displayed live only when the window has the focus (that is, is active).

Using an MGA VGA board

Matrox recommends using Matrox MGA boards for real-time display of video data. Selection of an MGA board depends on your application's requirements. To find out more about display mode resolutions on a particular board, see the *MIL/MIL-Lite Board-specific notes* manual.

Using a VGA board other than MGA board

If your VGA is not an MGA board, you must reconfigure the [Vga] section in the *mil.ini* file.

The following is an example of a *mil.ini* configuration file, describing the Matrox MGA Millennium-II PCI board (contact your VGA board vendor for this information). The Matrox vendor identifier is 102B, the MGA Millennium-II device identifier is 051B, the VGA frame buffer is mapped to an address, offset by 0 from its PCI base address of 0:

```
[Vga]
VgaVendorId=102B
VgaDeviceId=051B
VgaBaseAddressIndex=0
VgaBaseAddressOffset=0
```

Instead of specifying all of the above parameters, you can specify the VGA board's physical address:

```
VgaPhysicalAddress=EF000000
```

If the live grab operation does not have the proper pitch or the proper pixel depth, the following optional entries must be specified:

```
VgaPitch=400
VgaFormat=M_BGR15+M_PACKED
```

❖ All values are hexadecimal.

The default location of the *mil.ini* file is the Windows directory under Microsoft Windows. A different location can be specified using the environment variable, MILINIDIR.

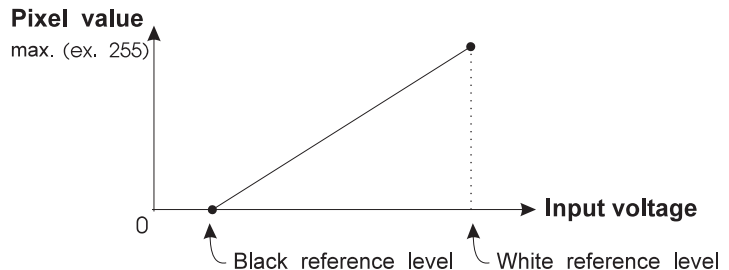
Reference levels, lookup tables, and scaling

MIL provides functions to improve the appearance of a grabbed image on input (if your hardware allows it). You can adjust the brightness and contrast of the images, as well as the hue and saturation for color grabs, by fine-tuning the controls of the analog-to-digital converters in your system. You can also correct and precondition the input data prior to storing it, through scaling, or by mapping it through an input LUT.

Black and white reference levels

When digitizing images, 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 any voltage above the white reference level to the maximum pixel value, and any voltage below the black reference level to a zero pixel value.

Matrox digitizers support fine-tuning of these reference levels. By reducing or increasing either or both the black and white reference levels, you affect the brightness of the image. By reducing one reference level and increasing the other, you affect the contrast of the image.



MIL linearly represents the distance between the minimum and maximum voltages, in which the black reference level can be adjusted (hardware-specific), as units between `M_MIN_LEVEL` and `M_MAX_LEVEL`. The same is done for the white reference level adjustment range. These units are the values by which you can adjust the specified reference level, using *MdigReference()*.

To calculate the value to pass to *MdigReference()*, use the following equation with the appropriate voltages specified in the *MIL/MIL-Lite Board-specific notes* manual for your particular board.

$$\text{Value to pass to } MdigReference() = \left(\frac{\text{Voltage needed} - \text{minimum voltage}}{\text{maximum voltage} - \text{minimum voltage}} \right) (M_MAX_LEVEL - M_MIN_LEVEL)$$

The smallest voltage increment supported by your board can differ such that consecutive reference-level settings might produce the same result.

Note, the new reference level might not take effect until the next grab, at which point, a certain amount of delay might be incurred as the hardware adjusts to the reference-level changes.

Color image reference levels

When grabbing composite color images, *MdigReference()* provides specific control parameters to adjust the levels of contrast, brightness, hue, and saturation. These levels can be set to values from 0 to 255. See the *MIL/MIL-Lite Board-specific notes* manual for your particular board for more details.

Mapping grabbed data through a LUT

You can correct or precondition input data by mapping it through a LUT when grabbing (if the hardware permits). This requires that you copy a LUT buffer to a digitizer's physical input LUT, using *MdigLut()*.

You can copy a LUT buffer that has the same number of color bands as the digitizer's physical input LUTs. If you copy a one-band LUT buffer to a digitizer that has more than one physical input LUT, each of the digitizer's LUTs is loaded with the same LUT buffer data.

In addition, the LUT buffer's number of entries must match the digitizer's input data range.

To revert to the default LUT values, you must copy the default LUT (M_DEFAULT) to the digitizer. For digitizers, the default LUT is one that maps pixels to the same values. This type of LUT is typically referred to as a transparent LUT.

Scaling

The *MdigControl()* function allows you to scale grabbed data horizontally and vertically. If you scale grabbed data, the stored image size is different from the original image by the specified factors in the X and/or Y direction. The scaled image is written in contiguous locations in the image buffer, starting from the top-left corner. For example, if you set both the X and Y scaling factors to 1/2, only one column and one row out of two are written to the image buffer.

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	115	0	0	0	0	0	0	0
0	215	244	0	0	0	0	0	0	0
0	215	243	196	196	0	0	0	0	0
0	215	111	111	87	87	87	87	86	87
0	0	0	111	115	87	87	87	87	0
0	0	0	0	111	111	115	45	0	0
0	0	0	0	0	111	92	92	0	0
0	0	0	0	0	0	111	111	0	0



0	0	0	0	0
0	115	0	0	0
0	243	196	0	0
0	0	115	87	87
0	0	0	92	0

Subsampled image

X subsampling factor = 2

Y subsampling factor = 2

Original image

The X and Y scaling factors are independent. Note, depending on the digitizer and camera used, some scaling factors might not be available.

To disable scaling, set scaling factors to 1.

Optimizing application performance when grabbing

Grab mode

When grabbing data with *MdigGrab()*, you can control the synchronization by setting the *MdigControl()* `M_GRAB_MODE` control type to a value of `M_SYNCHRONOUS`, `M_ASYNCHRONOUS`, or `M_ASYNCHRONOUS_QUEUED` (if supported).

- If the grab mode is set to `M_SYNCHRONOUS`, your application will be synchronized with the end of a grab operation. In other words, your application will wait until the grab has finished before executing the next command.
- If the grab mode is set to `M_ASYNCHRONOUS`, your application will not be synchronized with the end of a grab operation. This option allows other commands to execute while still grabbing. This is a useful option when performing double buffering, a technique whereby you can grab data into one buffer while processing the previously grabbed buffer (discussed below). Note, a call to another *MdigGrab()* before the current grab has finished will cause your application to wait until the current grab has finished.
MdigGrabContinuous() is by definition asynchronous since you must use *MdigHalt()* to stop the grab.
- If your imaging board supports queuing, you can set the grab mode to `M_ASYNCHRONOUS_QUEUED`; if another grab is issued before the first one is finished, the grab will be queued on-board, allowing you to perform other processes while waiting for the next *MdigGrab()* to be executed. Note, you can still force your application to wait until the end of a grab before executing an operation, by calling *MdigGrabWait()*.

Double buffering

Double buffering involves grabbing into one image while processing the previously grabbed image. Double buffering allows you to grab and process concurrently. You must switch the destination of the grab between the two image buffers. In addition, you need to synchronize the grabbing and processing so that:

- You do not process an image 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.

Below is an example of how to perform double buffering:

```

/* ***** */
/* This example does double buffered grab with real time processing. */
/* Note: This assume that the processing operation is shorter than a grab */
/* and that the PC has sufficient bandwidth to support the 2 */
/* operations simultaneously. Also if the target processing buffer */
/* is not on the display, the processing speed is augmented. */
.
.
.
/* Image scale. */
#define IMAGE_SCALE 0.5

/* headers */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <mil.h>

/* Main function. */
void main(void)
{
    MIL_ID    MilApplication;
    MIL_ID    MilSystem      ;
    MIL_ID    MilDigitizer   ;
    MIL_ID    MilDisplay     ;
    MIL_ID    MilImage[2]    ;
    MIL_ID    MilImageDisp   ;

    long      NbProc = 0;

```

(cont...)

```

/* Allocations. */
MappAlloc(M_DEFAULT, &MilApplication);
MsysAlloc(M_DEF_SYSTEM_TYPE, M_DEF_SYSTEM_NUM, M_SETUP, &MilSystem);
MdigAlloc(MilSystem, M_DEFAULT,
M_DEF_DIGITIZER_FORMAT, M_DEFAULT, &MilDigitizer);
MdispAlloc(MilSystem, M_DEFAULT, M_DEF_DISPLAY_FORMAT, M_DEFAULT,
&MilDisplay);

/* Allocate 2 grab buffers. */
MbufAlloc2d(MilSystem,
            (long)(MdigInquire(MilDigitizer, M_SIZE_X, M_NULL)*IMAGE_SCALE),
            (long)(MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL)*IMAGE_SCALE),
            8L+M_UNSIGNED,
            M_IMAGE+M_GRAB+M_PROC, &MilImage[0]);
MbufAlloc2d(MilSystem,
            (long)(MdigInquire(MilDigitizer, M_SIZE_X, M_NULL)*IMAGE_SCALE),
            (long)(MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL)*IMAGE_SCALE),
            8L+M_UNSIGNED,
            M_IMAGE+M_GRAB+M_PROC, &MilImage[1]);

/* Allocate 1 displayable buffer and clear it. */
MbufAlloc2d(MilSystem,
            (long)(MdigInquire(MilDigitizer, M_SIZE_X, M_NULL)*IMAGE_SCALE),
            (long)(MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL)*IMAGE_SCALE),
            8L+M_UNSIGNED,
            M_IMAGE+M_GRAB+M_PROC+M_DISP, &MilImageDisp);
MbufClear(MilImageDisp, 0x0);
.
.
.

/* Put the digitizer in asynchronous mode. */
MdigControl(MilDigitizer, M_GRAB_MODE, M_ASYNCHRONOUS);

/* Grab into the first buffer. */
MdigGrab(MilDigitizer, MilImage[0]);

/* Process one buffer while grabbing the other. */
while( !kbhit() )
{
    /* Grab second buffer while processing first buffer. */
    MdigGrab(MilDigitizer, MilImage[1]);
    .
    .
    .

```

(cont...)

```

/* Process the first buffer already grabbed. */
/* Note: Real time only if PC is fast enough. */
MimConvolve(MilImage[0], MilImageDisp, M_EDGE_DETECT);
.
.
.
/* Grab first buffer while processing second buffer. */
MdigGrab(MilDigitizer, MilImage[0])

/* Process the second buffer already grabbed. */
MimConvolve(MilImage[1], MilImageDisp, M_EDGE_DETECT);
}
.
.
.
/* Free allocations. */
MbufFree(MilImageDisp);
MbufFree(MilImage[0]);
MbufFree(MilImage[1]);
MdispFree(MilDisplay);
MdigFree(MilDigitizer);
MsysFree(MilSystem);
MappFree(MilApplication);
}

```

Multiple buffering

When an occasional frame takes longer to process than the time required to grab, you can use a multiple buffering technique to ensure that all processing is completed without losing any frames. To perform multiple buffering, use the *MdigHookFunction()* when grabbing asynchronously to hook the grab function to certain grab events, such as the start or end of a frame: the hooked function will interrupt the processing to perform the grab, and return to continue processing after the grab is initiated. You can grab into as many buffers as required to ensure that all processing is finished before overwriting a buffer with a new frame.

Note, processing is generally faster if the buffer is not on the display.

Grabbing a sequence of frames in real-time

To grab a sequence of frames in real-time, simply use successive, asynchronous calls to *MdigGrab()* :

```
/* Put digitizer in asynchronous mode */
MdigControl(MilDigitizer, M_GRAB_MODE, M_ASYNCHRONOUS);

/* Grab the sequence. */
for (n=0; n<NbFrames; n++)
{
    /* Grab one buffer at a time. */
    MdigGrab(MilDigitizer, MilImage[n]);
}
```

Note that you must also allocate a buffer for each frame of the sequence. After you have grabbed a sequence, you can use the *MbufExportSequence()* function to export the sequence of image buffers (compressed or un-compressed 8-bit) to an *.avi file. When exporting, you must specify the number of buffers and the frame rate (number of images/second) of the sequence. Note, the MIL identifiers of the image buffers to export must be kept in an array.

Use the *MbufImportSequence()* to import a sequence of images from an *.avi file into separate image buffers. You can import compressed (MJPEG) or un-compressed 8-bit images. You can also choose to import the sequence into automatically allocated buffers or previously allocated buffers.

Grabbing with triggers and exposures

If your Matrox digitizer supports trigger input, this allows you to grab a frame upon the occurrence of an event; that is, nothing is grabbed when you call **MdigGrab()** or **MdigContinuousGrab()**, until a specified event occurs. When grabbing continuously, the digitizer waits for a trigger before grabbing each frame; you must still call **MdigHalt()** after grabbing all required frames.

The camera's digitizer definition format (DCF) file specifies whether or not to perform a triggered grab and exactly how it should be carried out. For example, if the DCF specifies that an

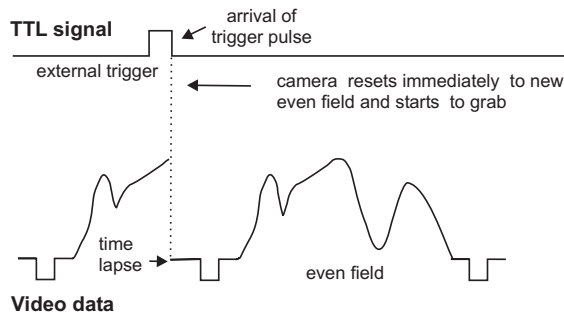
exposure signal should be generated (for the camera) upon the grab trigger event, the actual grab would only be triggered once the active exposure time was over.

You can use MIL commands to override the DCF trigger settings. You can enable/disable whether **MdigGrab()**/**MdigContinuousGrab()** performs a triggered grab using **MdigControl()** with `M_GRAB_TRIGGER`. You can also specify the source and activation mode of the event upon which to grab using **MdigControl()** with `M_GRAB_TRIGGER_SOURCE` and then with `M_GRAB_TRIGGER_MODE`.

Asynchronous reset mode

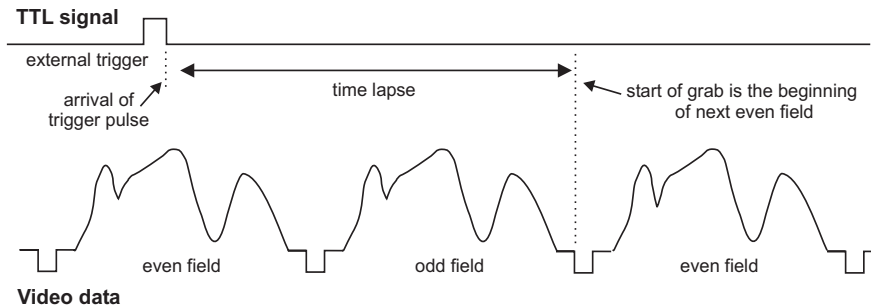
If your digitizer supports asynchronous reset mode, the digitizer resets the camera to begin a new frame when the trigger signal is received.

Asynchronous reset mode



Otherwise, the digitizer waits for the next valid frame (or field) before commencing to grab. The grab activation mode is specified in the DCF file.

Next valid frame (or field) mode



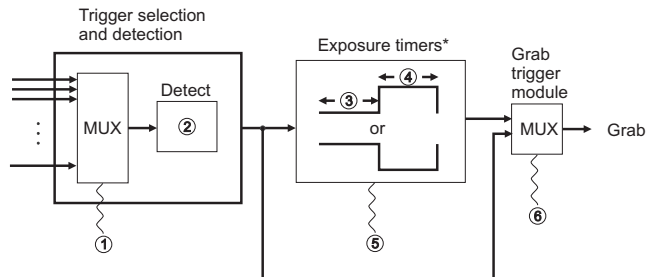
Triggers and exposures

In MIL, there are two methods of grabbing with triggers and exposures: the automatic exposure model and the manual bypass model. They are described in detail in the following diagrams. By default, MIL uses the automatic exposure model. You can change this default using **MdigControl()** with `M_GRAB_EXPOSURE_BYPASS`.

Automatic exposure model

In the automatic exposure model, the digitizer is configured to have the pipeline that is illustrated in the next diagram. (Note that the defines specified in the following illustration are those to be used with the **MdigControl()** function).

(M_GRAB_EXPOSURE_BYPASS set to M_DISABLE or M_DEFAULT)



- ① trigger source (M_GRAB_TRIGGER_SOURCE)
- ② trigger detection method (M_GRAB_TRIGGER_MODE)
- ③ exposure delay (M_GRAB_EXPOSURE_TIME_DELAY)
- ④ exposure time (M_GRAB_EXPOSURE_TIME)
- ⑤ polarity of exposure signal (M_GRAB_EXPOSURE_MODE)
- ⑥ bypass exposure timers if exposure time = 0 (M_GRAB_EXPOSURE_TIME)

* exposure timers will be cascaded automatically (if necessary) to generate one signal that has the required delay and active time

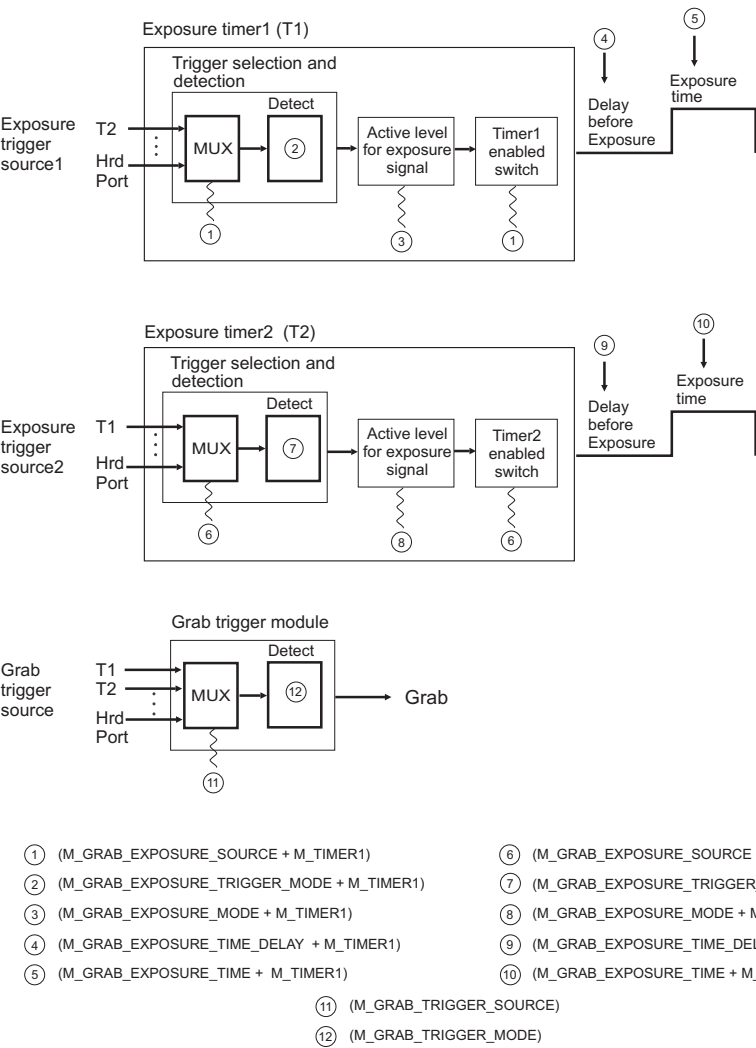
To summarize:

- **MdigControl()** with M_GRAB_TRIGGER_SOURCE selects which signal to use as the source of the trigger (for example, M_HARDWARE_PORT0). **MdigControl()** with M_GRAB_TRIGGER_MODE, selects the trigger detection method (for example, trigger on the rising edge of the signal).
- If the exposure time (**MdigControl()** with M_GRAB_EXPOSURE_TIME) is zero, the trigger sets off the grab trigger module immediately, initiating the actual grab. The exposure timers are bypassed.
- If you set the exposure time to a non-zero value, an exposure signal is generated with an active period equal to the specified exposure time (M_GRAB_EXPOSURE_TIME). The active period occurs after the specified delay (M_GRAB_EXPOSURE_TIME_DELAY). The signal will be generated with the specified polarity (M_GRAB_EXPOSURE_MODE). The end of exposure will trigger the grab trigger module, initiating the actual grab.

Manual exposure
bypass model

In the manual bypass model, you are responsible for enabling and setting-up all the exposure timers and grab trigger connections

Manual exposure bypass model
(M_GRAB_EXPOSURE_BYPASS set to M_ENABLE)



Software triggers

In general, the digitizer's grab trigger module and exposure timers can also be triggered by software (M_SOFTWARE). In this case, following a grab call, nothing is grabbed until you call a specific function (discussed below). Note that in this case, the grab call must be asynchronous (that is, issue the grab with **MdigGrab()** in asynchronous mode or with **MdigGrabContinuous()** or the grab call must be called on a separate thread.

In the automatic exposure model

In the automatic exposure model, issue the software trigger by calling **MdigControl()** with M_GRAB_TRIGGER and M_ACTIVATE. This will trigger the grab if the exposure time is 0, otherwise the call will trigger the exposure signal which in turn will trigger the grab.

In the manual bypass model

In the manual bypass model, to issue a software trigger for the grab trigger module, call **MdigControl()** with M_GRAB_TRIGGER and M_ACTIVATE. To issue a software trigger for one of the exposure timers, call **MdigControl()** with M_GRAB_EXPOSURE+M_TIMER n and M_ACTIVATE.

Note, for a digitizer without an exposure timer, the exposure time is considered to be zero.

Auto-focusing

You can use *MdigFocus()* to automatically adjust the lens motor of your camera to a position that produces optimum focus in your images. This function is primarily useful when your camera's depth of field is limited with respect to the range required by the grabbed object and manual adjustment is not possible.

MdigFocus() determines the optimum focus position by grabbing an image at an initial lens position, analyzing the focus quality of the grabbed image, calling a user-defined function that changes the position of the lens motor, and then grabbing and analyzing another image. The process repeats until the optimum focus position is found.

The focus quality of an image (known as its *focus indicator*) is measured by analyzing its edges. An image with good focus quality (a high focus indicator) has well-defined edges, that is, has a sharp difference in gray-levels between its object edges and its background.

By default, *MdigFocus()* subsamples and filters each grabbed image before analyzing it. This makes it easier to analyze the image. If necessary, you can specify that the subsampling and/or filtering be skipped. Skipping these operations will result in a more accurate analysis of the image's focus quality. It is primarily useful to skip these operations when your images contain fine details since subsampling or filtering can remove these details. Note that subsampling the grabbed images increases the speed of *MdigFocus()*; filtering the grabbed images slows down *MdigFocus()*.

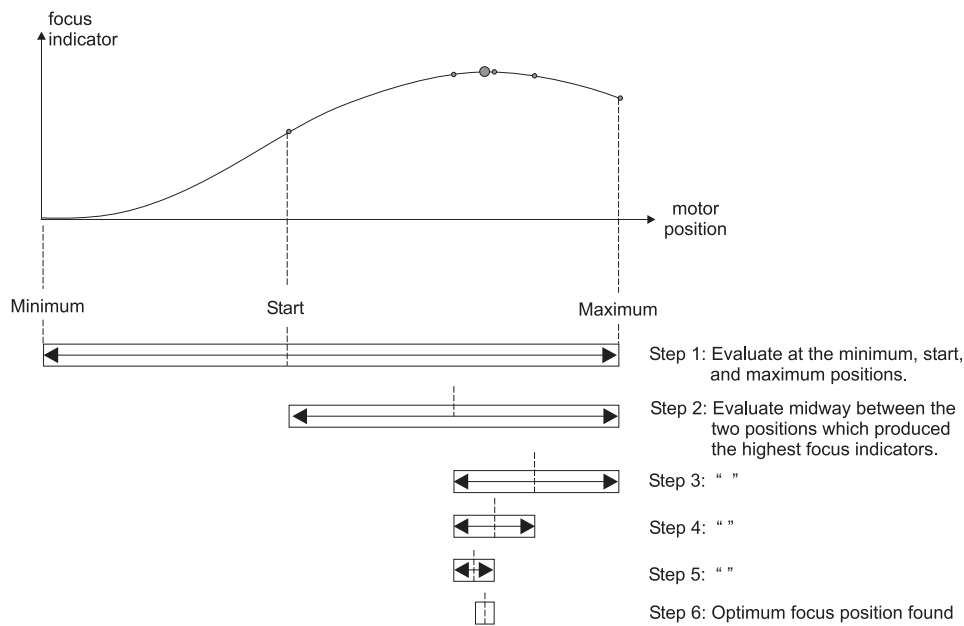
If necessary, you can specify that only a sub-region of the image be analyzed, by passing a child buffer to the function. This is primarily useful if there are objects at different distances within the camera's field of view. In such a case, each object will have a different optimum focus position, so you need to use a child buffer to specify the object on which to focus.

Search strategies

When you perform *MdigFocus()*, you have to specify the minimum, maximum, and starting position of the lens motor. Given these parameters, different strategies can be used to find the optimum focus position. These strategies determine how the position is updated (in which direction and by how much) between grabs. They can affect the speed and accuracy of the operation.

Bisection strategy

The bisection strategy breaks down the given positional range, step-by-step, until it finds the optimum focus position.

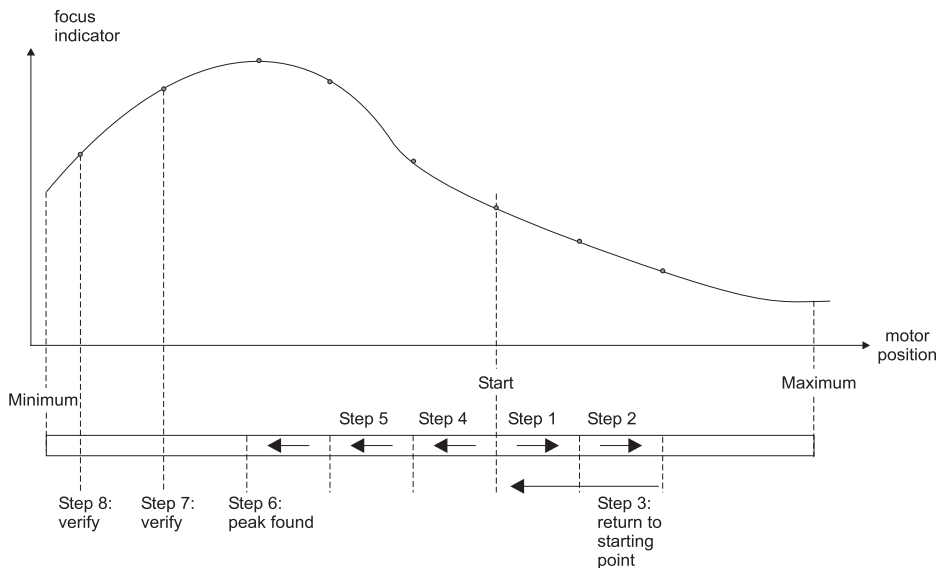


In general, the bisection strategy processes the fewest amount of images. However, it is most sensitive to noise and requires that the lens motor travel the greatest distance.

Refocus strategy

The refocus strategy scans upward or downward until it finds the optimum focus position or until it reaches the minimum or maximum position. While scanning in one direction, if the focus indicator decreases continuously (indicating an out-of-focus condition), the focus position is returned to its starting point and scanning is started in the opposite direction. By default, if a peak in focus indicator values is found, the next two positions

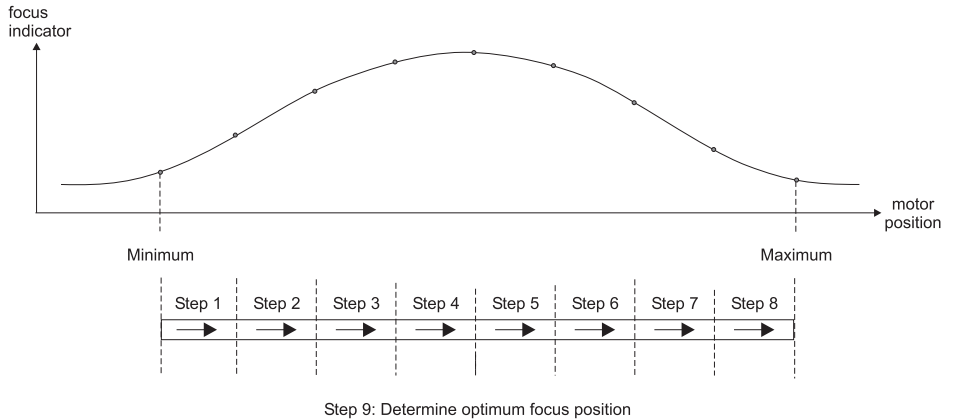
are scanned to make sure the peak is truly the optimum. If necessary, you can change the number of positions used to verify a peak.



The refocus strategy is the best strategy to use when the current focus position is close to optimum.

Scan-All strategy

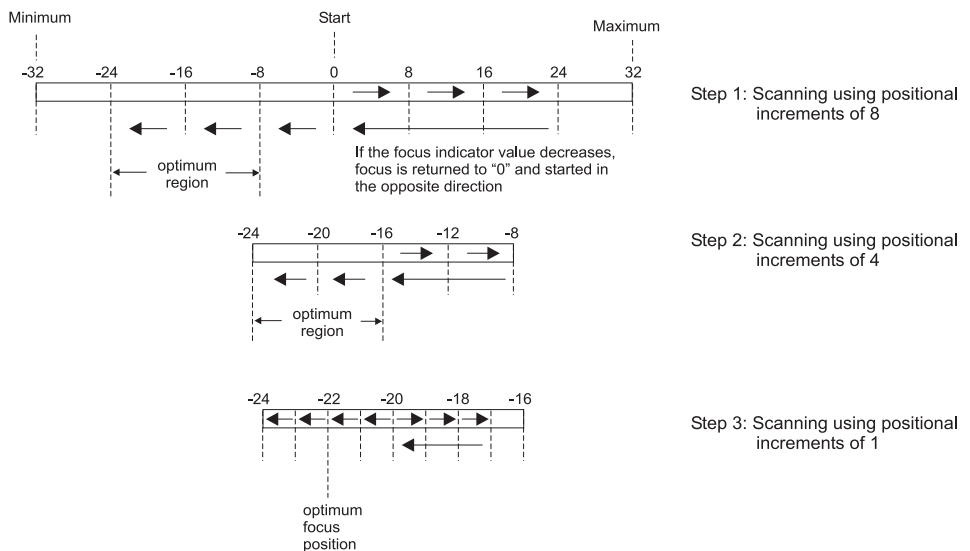
The scan-all strategy scans, by 1, all positions between the minimum and maximum and returns the position which produced the highest focus indicator.



The scan-all strategy is the slowest but most accurate.

Smart-Scan strategy

The smart-scan strategy performs three refocus searches, each with a smaller positional increment. You specify the initial positional increment; the subsequent increments are factors of the initial one. As with the refocus strategy, the default number of positions used to verify a peak is 2 but can be changed.



The smart-scan strategy is a compromise between the speed of a bisection and the accuracy of a scan-all.

*Evaluate the focus
indicator*

Rather than determine the optimum focus position, *MdigFocus()* can be used to simply return the focus indicator value for a given image or for the image grabbed at the current lens position.

Chapter 21: Color

This chapter discusses how to handle objects in color with MIL.

Dealing with color

MIL supports grabbing, displaying, and processing color images.

MIL can represent an object in color with a single color buffer, allocated with *MbufAllocColor()*.

Grabbing

You grab from an input device (typically a camera) into a color image buffer, as you would into a two-dimensional grayscale image buffer, by calling *MdigGrab()* or *MdigGrabContinuous()*.

Before performing a color grab, a digitizer must be allocated, using *MdigAlloc()* (or *MappAllocDefault()*), specifying a color digitization data format. In addition, the digitizer and the image buffer must be allocated on the same system and have compatible dimensions. Once you have finished using the digitizer, you should free it, using *MdigFree()*.

When grabbing from a color digitizer, each color component is transmitted simultaneously. The destination buffer must have the same number of color bands as the digitizer. The data is simultaneously stored in the appropriate component of the image buffer. When grabbing RGB, the red component is stored in the first color band, the green component is stored in the second color band, while the blue component is stored in the third color band.

If the hardware permits, you can control the digitization reference level of each channel, using *MdigReference()*.

❖ Note, upon installation, if you specified a color camera, the default image buffer allocated with *MappAllocDefault()* will be a three-band color image buffer. If you didn't specify a color camera, but would now prefer to use one, you might want to update the *milsetup.h* file to reflect the desired defaults for the allocation of your color camera and a color image buffer.

Note, most examples in this manual assume that the target system has a monochrome digitizer, and that the camera and default image buffer are monochrome. To run the examples using a color digitizer and image buffer, you must modify the code appropriately.

Mapping grabbed data through a LUT

You can also correct or precondition input data by mapping it through a LUT upon acquisition (if the hardware permits). This requires that you associate a LUT buffer with the input device, using *MdigLut()*.

The LUTs that can be associated to a digitizer are either one-dimensional LUT buffers (single rows) or LUT buffers that have the same number of color bands as the digitizer. If you associate a one-dimensional LUT buffer with the digitizer, each of the digitizer's color band input LUTs is loaded with the one-dimensional LUT buffer data. If you associate a multi-band LUT buffer with the digitizer, each of the digitizer's color band LUTs is loaded with its corresponding color band LUT buffer data.

Note, the LUT buffer depth must match the digitizer's pixel depth.

To disassociate the LUT buffer from the digitizer, you need to associate the digitizer with the default LUT, using *M_DEFAULT* as a parameter to *MdigLut()*.

Displaying

You display a color-image buffer as you would a two-dimensional grayscale image buffer. You must first allocate the image buffer with a displayable attribute (`M_DISP`), then select it for display, using *MdispSelect()*. To stop displaying the image buffer and leave the display blank, use *MdispDeselect()*.

Before you can display a buffer, the display must be allocated, using *MdispAlloc()* (or *MappAllocDefault()*). The image buffer and the display must be allocated on the same system and have compatible dimensions.

When you display a color-image buffer (usually RGB), the first band is routed to the first output channel (usually red), the second band is routed to the second output channel (usually green), while the third band is routed to the third output channel (usually blue).

When a display is allocated, a default pass-through LUT (transparent LUT) is loaded into the output LUT(s) (if any). You can change the displayed colors of an image by associating a lookup table (LUT) to the display, using *MdispLut()*.

When you associate a one-color-band LUT buffer with a display that has more than one output LUT, the same LUT buffer data is loaded in each of the available output channel LUTs.

When you associate a multi-band LUT buffer to a display that has multiple output LUTs, each output LUT is loaded with the data of the corresponding LUT buffer color band.

To disassociate the LUT buffer from the display, you need to associate the display with the default LUT, using `M_DEFAULT` as a parameter to *MdispLut()*.

Processing

MIL can process color (multi-band) image buffers by processing each band of an image individually. However, MIL cannot perform statistical analysis, blob analysis or pattern recognition operations on color image buffers.

To process a single band of a color image, you can extract one band, using *MbufCopyColor()* or access it directly, using *MbufChildColor()*. In either case, processing can then be performed on the two-dimensional single band buffer. In the case of *MbufCopyColor()*, after each color band has been extracted and processed, it can be re-inserted into the buffer, using *MbufCopyColor()*. If *MbufChildColor()* was used, the parent buffer (in this case a multi-band buffer) is automatically updated after processing since its child buffer occupies the same physical space in memory.

Using the MIL command *MimConvert()*, you can perform color conversions, such as converting an RGB image into a HLS (Hue, Luminance, and Saturation) image and vice versa. You can also extract the luminance (intensity) from an RGB image or copy the luminance component of an image into a three-band buffer to create a monochromatic (gray) RGB buffer.

An example

The following is an example of color image manipulation and conversion.

```

/* File name: mcolor.c
 * Synopsis: This program allocates a displayable color image buffer,
 *           displays it and loads its contents with a color image. It then
 *           converts it to Hue, Luminance, Saturation (HLS), adds a
 *           certain offset to the luminance component and converts the
 *           image back to Red, Green, Blue (RGB) to display the result.
 */

#include <stdio.h>
#include <stdlib.h>
#include <mil.h>

/* Source MIL image file specifications. */
#define IMAGE_FILE      "bird.mim"
#define IMAGE_WIDTH     256L
#define IMAGE_HEIGHT    240L
#define IMAGE_BAND      3L
#define IMAGE_DEPTH     8L

/* Luminance offset to add and maximum value of the image */
#define IMAGE_LUMINANCE_OFFSET  40L
#define IMAGE_MAX_VALUE        255L

void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem,             /* System identifier. */
    MilDisplay,            /* Display identifier. */
    MilImage,              /* Image buffer identifier. */
    MilSubImage0,          /* Sub-image buffer identifier for original image. */
    MilSubImage1,          /* Sub-image buffer identifier for processed image. */
    MilSubImageLum;        /* Sub-image buffer identifier for luminance. */
    long ImageSizeX,       /* Image width. */
    ImageSizeY;            /* Image height. */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                     M_NULL, M_NULL);
}

```

(cont...)


```

/* Find the best size for the display image depending on the display type. */
if (MdispInquire(MilDisplay,M_DISP_MODE,M_NULL)==M_WINDOWED)
{
    ImageSizeX = IMAGE_WIDTH * 2;
    ImageSizeY = IMAGE_HEIGHT;
}
else
{
    /* The size of the entire display to avoid possible display artifact. */
    ImageSizeX = min(MdispInquire(MilDisplay,M_SIZE_X,M_NULL),
        M_DEF_IMAGE_SIZE_X_MAX);
    ImageSizeY = min(MdispInquire(MilDisplay,M_SIZE_Y,M_NULL),
        M_DEF_IMAGE_SIZE_Y_MAX);
}

/* Allocate a color display image buffer to perform processing in it. */
MbufAllocColor(MilSystem, IMAGE_BAND, ImageSizeX, ImageSizeY,
    IMAGE_DEPTH+M_UNSIGNED, M_IMAGE+M_DISP+M_PROC, &MilImage);

/* Clear the image buffer. */
MbufClear(MilImage, 0L);

/* Display the image buffer. */
MdispSelect(MilDisplay, MilImage);

/* Define 2 processing buffers in the display buffer, restricting the
 * regions to be processed to the source image size. */
MbufChild2d(MilImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage0);
MbufChild2d(MilImage, IMAGE_WIDTH, 0L, IMAGE_WIDTH, IMAGE_HEIGHT,
    &MilSubImage1);

/* Load a color image in subimage0. */
MbufLoad(IMAGE_FILE, MilSubImage0);

/* Print a message. */
printf("A color source image was loaded and will be processed\n");
printf("to increase its luminance.\nPress <Enter> to continue.\n");
getchar();

/* Converts it to Hue, Luminance, Saturation (HLS). */
MimConvert(MilSubImage0, MilSubImage1, M_RGB_TO_HLS);

/* Do a child that map to the luminance component */
MbufChildColor(MilSubImage1, M_LUMINANCE, &MilSubImageLum);

```

(cont. ...)

```

/* Clip the buffer luminance component to avoid later saturation. */
MimClip(MilSubImageLum, MilSubImageLum, M_GREATER,
        IMAGE_MAX_VALUE-IMAGE_LUMINANCE_OFFSET, M_NULL,
        IMAGE_MAX_VALUE-IMAGE_LUMINANCE_OFFSET, M_NULL);

/* Add an offset to the luminance component. */
MimArith(MilSubImageLum, IMAGE_LUMINANCE_OFFSET, MilSubImageLum, M_ADD_CONST);

/* Converts it back to Red, Green, Blue (RGB) for display. */
MimConvert(MilSubImage1, MilSubImage1, M_HLS_TO_RGB);

/* Print a message. */
printf("The color source image in the top left corner was converted\n");
printf("to HLS, the luminance component was augmented and it was\n");
printf("converted back to RGB in the top right corner image.\n");
printf("Press <Enter> to end.\n");
getchar();

/* Release subimages and color image buffer. */
MbufFree(MilSubImageLum);
MbufFree(MilSubImage1);
MbufFree(MilSubImage0);
MbufFree(MilImage);

/* Release defaults. */
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);
}

```

Saving and loading color images

MIL supports the saving and loading of color images from disk in different file formats. See the *MbufSave()*, *MbufLoad()*, *MbufRestore()*, *MbufImport()*, and *MbufExport()* command reference descriptions in the *Matrox Imaging Library Command Reference* manual for more details.

Note, all the MIL data allocation, access, and generation (*Mbuf...()* and *MgenLut...()*) commands can handle color image buffers.

Chapter 22 : JPEG compression

This chapter describes how to compress and decompress images.

Introduction

MIL allows you to compress and decompress images. Compression allows you to store more images in memory than would normally be possible. In addition, it allows images to be transferred more quickly, since it reduces the amount of data that must be transferred. MIL can compress images using the JPEG lossless algorithm or the JPEG lossy algorithm.

❖ Under MIL-Lite, dedicated hardware is required to compress and decompress images.

JPEG lossless

The JPEG lossless algorithm compresses images without any loss of information. Typically, the algorithm compresses images by a factor of 2:1, although a factor of 4:1 can sometimes be achieved. The JPEG lossless algorithm can compress 8- or 16-bit buffers with 1 or 3 bands.

JPEG lossy

The JPEG lossy algorithm compresses images by a variable factor but introduces some loss of information. The higher the compression factor, the more the compression, but the lower the image quality. The JPEG lossy algorithm can compress 8-bit buffers with 1 or 3 bands. To be compatible with most image-viewing software, MIL allows you to store compressed color images in YUV format.

Interlaced JPEG

MIL can perform a JPEG compression such that the image data is stored in separate fields. This is referred to as an *interlaced JPEG compression*. Unless otherwise stated, everything that applies to a JPEG compression also applies to an interlaced JPEG compression.

Control options

MIL allows you to control certain aspects of a compression. Specifically, you can use your own compression tables, although the default tables are suitable for most applications.

*.avi files

You can use *MbufExportSequence()* to export a sequence of image buffers to an audio video interleave (*.avi) file. You can use *MbufImportSequence()* to import a sequence of images from an *.avi file into separate buffers.

General steps

Compression

To compress an image:

1. Allocate a buffer in which to hold the compressed image. Use *MbufAlloc...()*, allocating the buffer with an `M_COMPRESS+CompressionType` attribute.
2. If necessary, change the control settings of the buffer, using *MbufControl()*. Specifically, for a lossy compression, you might want to change the quantization factor, which is one of the factors that determine the amount of compression.
3. If the image to compress is stored in a buffer, use *MbufCopy()* to compress it into the buffer allocated in step 1. If it is stored on file, use *MbufImport()*. Note that, if you want the compressed image stored on file rather than in a buffer, use *MbufExport()* instead of *MbufCopy()*. In this case, there is no need to allocate a destination buffer.

You can also automatically compress your grabbed images. To do so, use *MdigGrab()* with a destination buffer that has an `M_GRAB+M_COMPRESS+CompressionType` attribute.

Decompression

To decompress an image, use *MbufCopy()*, *MbufImport()*, or *MbufExport()*, depending on where the source image is stored (in a buffer or on file) and where you want results written (to a buffer or file). Before the decompression, you should not change any control settings in the source image. This is because, in order for the reconstructed image to match the original, the same controls must be used to decompress. If you do change a control setting, the image data will be lost.

Multi-band buffers and color formats

When you allocate a multi-band buffer for a lossy compression, you can specify that the compressed image be stored in an RGB or YUV format. Note that most image-viewing software display compressed color images in YUV 4:2:2 format. When the chosen format differs from that of the source image, MIL internally converts the source image to the specified format, then performs the compression.

Multi-band buffers and control settings

If you are compressing a multi-band buffer, you can specify different control settings for each band. To do so, create a child buffer from each band, using *MbufChildColor()*, then set controls for each child buffer, using *MbufControl()*.

Alternatively, if you performing a lossy compression on a YUV image, you can use the `xx_LUMINANCE` and `xx_CHROMINANCE` control types. The `xx_LUMINANCE` control type affects the Y band, while `xx_CHROMINANCE` affects the U and V bands.

Application-specific markers

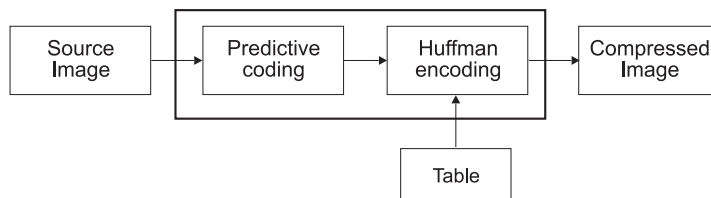
During a compression, MIL adds some application-specific markers to the resulting image. Most other packages will ignore these markers and therefore be able to decompress the file. MIL itself ignores unrecognized markers when it decompresses files.

Controlling a JPEG compression

This section provides a brief overview of the JPEG lossless and lossy algorithms and of the controls you have over these algorithms. In general, you should only change these controls if you are familiar with the algorithm you are using. For detailed information about the JPEG lossless and lossy algorithms, see 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.

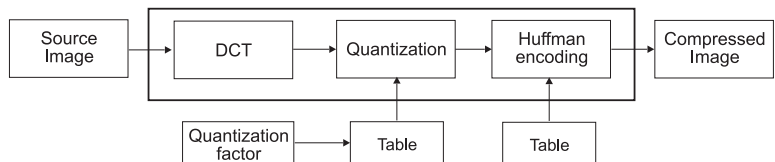
By default, MIL uses the pixel to the left to predict values. This is suitable for most images. However, you can specify that no predicting be done, using *MbufControl()*. In this case, the values after predictive coding will be the same as the original values. This can be useful if you have developed 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; MIL only supports predictor #0 (no predictor) and 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 the difference between 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 DC Huffman table. You can use the default DC Huffman table or you can create your own table. If you want to use your own table, refer to the *Using your own table* section.

JPEG lossy

The JPEG lossy algorithm is outlined below. First, each 8x8 block of the image is represented in its frequency domain through a discrete cosine transform, resulting in 1 DC and 63 AC values. Each block is then quantized and Huffman encoded.



Quantization divides each of the 64 values in a block by a specified value, according to a quantization table. After each block is quantized, Huffman encoding assigns a variable-length "code word" to each value. Each DC value in a block is assigned a code word according to a DC Huffman table. The AC values are assigned a code word according to an AC Huffman table. You can control a JPEG lossy compression by using your own quantization and/or Huffman tables.

Using your own table

For a JPEG lossless compression, you can use your own DC Huffman table. For a JPEG lossy compression, you can use your own quantization, DC Huffman, or AC Huffman table. In order to use your own table:

1. Allocate a buffer with an `M_ARRAY` attribute and of the required size. Huffman tables are one-dimensional, so use *MbufAlloc1d()* to allocate the buffer. Quantization tables are two-dimensional, so use *MbufAlloc2d()*.
2. Transfer the table values to the buffer, using *MbufPut1d()* or *MbufPut2d()*, depending on the type of table.
3. Associate the `M_ARRAY` buffer to the required image buffer, using *MbufControl()*.

Note that you can associate a different table to each band of a multi-band buffer. To do so, create a child buffer from each band, using *MbufChildColor()*, then associate a table to each child buffer. Alternatively, for lossy compressions of YUV images, use the `xx_LUMINANCE` and `xx_CHROMINANCE` control types.

Restart markers

When an image is compressed, MIL 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 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.

By default, MIL 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 *MbufControl()* to change the number of rows or blocks between restart markers.

- ❖ 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 number of rows or blocks between restart markers, especially if you are not transmitting the image over a noisy medium. In fact, if you are sure that the transmission medium is not noisy, you might want to set the restart interval to 0, that is, not use restart markers. This will increase the compression ratio, as well as reduce the time required to decompress the image.

Chapter 23: Data manipulation with multiple systems

Data manipulation with multiple systems

To use multiple Matrox imaging boards, you have to allocate a MIL system for each board.

Processing

To perform a processing operation, your source and destination buffers can be on different systems; MIL will transparently copy buffers to the most efficient of these system, if necessary.

Exchanging data

To exchange data between systems, you can physically copy the data from one system to another. The copy is always performed by the most suitable system. If both systems are of the same type, the copy is always performed by the destination system.

Instead of performing a physical copy using *MbufCopy()*, you can allocate a buffer on one system and use *MbufCreate...()* to access this buffer from another system. *MbufCreate...()* creates a buffer that maps to allocated memory (for example, on the Host or any MIL system); no memory is actually allocated to this newly created buffer.

The second method can be used, for example, to update a buffer (or part of it) with data grabbed from different systems. Note that after writing to the created buffer, you should notify the real buffer that its contents have been changed, by calling *MbufControl()* with *M_MODIFIED*. See *Chapter 16: Specifying and managing your data buffers* for more information about creating data buffers.

Grab and display

To grab, the digitizer and the destination buffer must be allocated on the same MIL system. Similarly, to display a buffer, the display and the buffer must be allocated on the same MIL system.

Systems without an on-board display section use the VGA for display. Therefore, under Windows, such systems will automatically display together on the same screen.

Chapter 24: Using MIL with multi-processing and under multi-thread systems

This chapter describes how MIL handles multi-processing and multi-threading.

Multi-processing

Multi-processing is the ability to execute various processes (applications) simultaneously.

MIL applications are autonomous processes (or executables) designed to execute a complete operation or series of operations. Therefore, they can profit from multi-processing by executing independently, without interference from each other.

In general, when multiple processes are running, no sharing of systems is permitted, except for the Host and VGA. Some particular systems, such as Matrox Genesis, can also be shared.

Systems with multi-processing

Systems that support multiple processes have on-board resources (like processors) that can be shared by different processes. However, if many processes are running at the same time, these processes have to share the available processing time and will not be able to share data.

Systems without multi-processing

Not all systems support multi-processing. For example, a simple frame grabber with only acquisition capability (like the Matrox Meteor-II) cannot ensure either the response time to a command or the independence of a process necessary for multi-processing. Therefore, on such systems MIL will refuse to allocate the system if it is already being used by another process. To use a non-multi-processing system within a multi-processing environment, all processes that need to communicate with the system must do so by sending their requests through a single dedicated process.

Multi-threading

MIL also supports multi-threading. Multi-threading is the ability to perform multiple operations simultaneously in the same process. This is done by creating different threads (execution queues) to ensure sequential execution of operations within the same thread, while allowing simultaneous yet independent execution of other operations in other threads.

Threads within a process share the same data. Therefore, they can communicate and exchange data such as MIL identifiers.

Multi-threading is most appropriate for applications where independent tasks can be done simultaneously but need to share data or to be controlled and synchronized within a main task.

Speed considerations

Multi-threading does not always result in an increase of speed and efficiency. Threads running simultaneously share the same system resources (such as memory) and generally run on the same CPU. This sharing can, in some cases, slow the process. For example, when using a system with multiple CPUs under Windows NT, the threads generally run on separate CPUs and provide more processing power. However, since they share the same memory, operations that are I/O intensive and require only simple processing might not be accelerated.

Alternatives

Most applications do not require the use of multiple threads since there are other ways of multi-tasking. Mechanisms such as asynchronous grab and call-back functions can be used (see *MdigControl()* and *MdigHookFunction()*). Applications resolved by alternative means are often simpler to implement and easier to maintain than multi-threaded applications.

MIL and multi-threading

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). For example, if you have two independent processing tasks that can be performed in parallel, it is often easier to have each controlled by a separate thread.

Thread execution

Under multi-thread operating systems, you can create as many threads as you require. The MIL commands in any thread are executed as follows:

- If the target processor is the Host CPU, processing in each thread is determined by the operating system.
- If the target processor is an on-board processor of a system that supports multi-threading (like the Matrox Genesis), MIL automatically creates, and eventually terminates, an on-board thread for each Host thread that sends commands to the board.

MIL application context

For each new Host thread sending MIL commands, MIL creates a new default MIL application context and initializes it to the state of the main MIL application (the first application allocated with *MappAlloc()*). Its purpose is to handle the context of the new thread, such as error reporting.

You can force the thread to inherit the state of a specific existing MIL application by creating a child MIL application, using *MappChild()*. Although inheriting (upon allocation) the state of the parent application, the child application is subsequently considered a separate application and can be modified independently of the state of its parent.

You can have the thread's application initialized with a reset initial state by allocating a new application, using *MappAlloc()*.

Synchronization

Thread synchronization is generally done by the Host synchronization services (such as Windows NT/2000 and 98 event objects). However, when using a system with an on-board processor, this processor is not synchronized with the Host.

This means that Host threads continue execution without waiting for the execution of the on-board commands to complete. In most cases, this is desirable to make the Host thread available for other tasks. However, for operations that necessitate the completion of a previous command(s) in order to return valid results (for example, *MbufGet()* after an *MdigGrab()*), MIL automatically synchronizes the threads to force the Host to wait for completion of the earlier command(s).

Explicit synchronization might be necessary if commands sharing a common resource or system might conflict with each other. For example, two threads sharing the same image buffer MIL identifier might each try to clear the buffer to a different value. If the threads are not synchronized, these commands might execute at the same time and the buffer could be cleared to either value or even to a combination of the two values. Use the MIL synchronization command, *MappControlThread()*, to control the flow of such commands.

Thread control

Windows NT/2000 and 98 systems are both multi-process and multi-thread. They provide various thread control services, including events (used to synchronize threads).

The MIL *MappControlThread()* command serves as a link between MIL and the operating system. It controls and coordinates both MIL threads and MIL events. It can create and delete a MIL thread, set a thread as the current active thread, set its processing mode, determine its current state, and synchronize its processing by forcing a "wait" state. It can exert similar controls on MIL events. MIL events can be used in addition to, or instead of, the operating system's events.

Error reporting

Some functions in MIL are asynchronous, that is, they queue their command to the hardware and then immediately return control to the Host. For this reason, errors are only reported when detected and not necessarily before the end of the MIL function.

The most common way to check for errors is to use the *MappGetError()* function. This function returns the errors currently detected in a thread.

An example of using multiple threads or systems

Multiple threads

The following example illustrates how multiple threads can be used to perform processing. It also illustrates how to synchronize multiple threads, using events.

```

/* File name: mthread.c
 * Synopsis: This program shows how to use different threads and synchronize
 *           them with MIL. It creates 4 processing threads that are used
 *           to work in 4 different regions of a display buffer.
 *
 *
 * Thread usage:
 *   - The main thread starts a processing thread in each of the 4 different
 *     quarters of a display buffer. The main thread then waits for a key to
 *     be pressed to stop them.
 *   - The top-left and bottom-left threads work in a loop, as follows: the
 *     top-left thread adds a constant to its buffer, then sends an event to
 *     the bottom-left thread. The bottom-left thread waits for the event
 *     from the top-left thread, rotates the top-left buffer, then sends an
 *     event to the top-left thread. When the top-left thread receives the
 *     event, the loop continues.
 *   - The top-right and bottom-right threads work exactly the same way as the
 *     top-left and bottom-left threads, except that the bottom-right thread
 *     performs an edge detection, rather than a rotation.
 *
 * Note that the top and bottom threads (of each half) could be set to do
 * something else while waiting for each other.
 */
/* headers */
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <windows.h>
#include <mil.h>
/* local defines */
#define IMAGE_FILE           "bird.mim"
#define IMAGE_WIDTH          256L
#define IMAGE_HEIGHT         240L
#define STRING_LENGTH_MAX   40
#define STRING_POS_X         10
#define STRING_POS_Y         220
#define STRING_TOP            "0"
#define STRING_BOTTOM        "0"
/* Thread function prototypes */
unsigned long MFTYPE TopThread(void *TParam);
unsigned long MFTYPE BotLeftThread(void *TParam);
unsigned long MFTYPE BotRightThread(void *TParam);

```

(cont....)

```

/* Thread parameters structure */
typedef struct
{
    MIL_ID  SrcImageId;
    MIL_ID  DstImageId;
    MIL_ID  EventSendId;
    MIL_ID  EventWaitId;
    MIL_ID  EventEndId;
    MIL_ID  EventEndBotId;
    long    *NumberOfIterPtr;
    long    *ComVarPtr;
} THREAD_PARAM;

/* Main function: */
void main(void)
{
    MIL_ID  MilApplication,      /* Application identifier.          */
    MilSystem,                  /* System identifier.               */
    MilDisplay,                 /* Display identifier.              */
    MilImage,                   /* Image buffer identifiers.        */
    MilChild,                   /* Child buffer identifiers.        */
    MilTopLeftImage,           /* Top left child image.            */
    MilBotLeftImage,           /* Bottom left child image.         */
    MilTopRightImage,          /* Top right child image.           */
    MilBotRightImage,          /* Bottom right child image.        */
    EventSendTopLeft,          /* Event send by top left thread.   */
    EventSendTopRight,         /* Event send by top right thread.  */
    EventWaitTopLeft,          /* Event waited on by top left thread. */
    EventWaitTopRight,         /* Event waited on by top right thread. */
    EventEndTopLeft,           /* Event used to exit top left thread. */
    EventEndBotLeft,           /* Event used to exit bottom left thread. */
    EventEndTopRight,          /* Event used to exit top right thread. */
    EventEndBotRight;          /* Event used to exit bottom right thread. */
    long  NumberOfTopLeft = 0L, /* Number of top left threads iterations. */
    NumberOfBotLeft = 0L,      /* Number of bottom left threads iterations. */
    NumberOfTopRight = 0L,     /* Number of top right threads iterations. */
    NumberOfBotRight = 0L,     /* Number of bottom right threads iterations. */
    ComVarLeft = 0L,           /* Communication variable for left thread. */
    ComVarRight = 0L;          /* Communication variable for right thread. */
    THREAD_PARAM  TParTopLeft, /* Parameters passed to top left thread. */
    TParBotLeft, /* Parameters passed to bottom left thread. */
    TParTopRight, /* Parameters passed to top right thread. */
    TParBotRight; /* Parameters passed to bottom right thread. */
    HANDLE  ThreadHandle[4]; /* Thread handles. */
    DWORD   ThreadId[4];    /* Thread Ids. */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem,
                     &MilDisplay, M_NULL, &MilImage);

```

(cont. ...)

```

/* Allocate child buffers. */
MbufChild2d(MilImage, 0, 0, IMAGE_WIDTH*2, IMAGE_HEIGHT*2, &MilChild);
MbufChild2d(MilChild, 0, 0, IMAGE_WIDTH, IMAGE_WIDTH, &MilTopLeftImage);
MbufChild2d(MilChild, IMAGE_WIDTH, 0, IMAGE_WIDTH,
            IMAGE_HEIGHT, &MilTopRightImage);
MbufChild2d(MilChild, 0, IMAGE_HEIGHT, IMAGE_WIDTH,
            IMAGE_HEIGHT, &MilBotLeftImage);
MbufChild2d(MilChild, IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_WIDTH,
            IMAGE_HEIGHT, &MilBotRightImage);
MdispSelect(MilDisplay, MilChild);

/* Allocate synchronization events. */
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventSendTopLeft);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventSendTopRight);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventWaitTopLeft);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventWaitTopRight);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventEndTopLeft);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventEndTopRight);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventEndBotLeft);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventEndBotRight);

/* Initialize source buffers. */
MbufLoad(IMAGE_FILE, MilTopLeftImage);
MbufLoad(IMAGE_FILE, MilTopRightImage);

/* Initialize threads parameter structures. */
TParTopLeft.SrcImageId      = MilTopLeftImage;
TParTopLeft.DstImageId      = MilTopLeftImage;
TParTopLeft.EventSendId     = EventSendTopLeft;
TParTopLeft.EventWaitId     = EventWaitTopLeft;
TParTopLeft.EventEndId      = EventEndTopLeft;
TParTopLeft.EventEndBotId   = EventEndBotLeft;
TParTopLeft.NumberOfIterPtr = &NumberOfTopLeft;
TParTopLeft.ComVarPtr       = &ComVarLeft;

TParBotLeft.SrcImageId      = MilTopLeftImage;
TParBotLeft.DstImageId      = MilBotLeftImage;
TParBotLeft.EventSendId     = EventWaitTopLeft;
TParBotLeft.EventWaitId     = EventSendTopLeft;
TParBotLeft.EventEndId      = EventEndBotLeft;
TParBotLeft.EventEndBotId   = M_NULL;
TParBotLeft.NumberOfIterPtr = &NumberOfBotLeft;
TParBotLeft.ComVarPtr       = &ComVarLeft;

TParTopRight.SrcImageId     = MilTopRightImage;
TParTopRight.DstImageId     = MilTopRightImage;
TParTopRight.EventSendId    = EventSendTopRight;
TParTopRight.EventWaitId    = EventWaitTopRight;
TParTopRight.EventEndId     = EventEndTopRight;
TParTopRight.EventEndBotId  = EventEndBotRight;
TParTopRight.NumberOfIterPtr = &NumberOfTopRight;
TParTopRight.ComVarPtr      = &ComVarRight;

```

(cont....)

```

TParBotRight.SrcImageId      = MilTopRightImage;
TParBotRight.DstImageId      = MilBotRightImage;
TParBotRight.EventSendId     = EventWaitTopRight;
TParBotRight.EventWaitId     = EventSendTopRight;
TParBotRight.EventEndId      = EventEndBotRight;
TParBotRight.EventEndBotId    = M_NULL;
TParBotRight.NumberOfIterPtr = &NumberOfBotRight;
TParBotRight.ComVarPtr       = &ComVarRight;

/* Start rotate and edge detect threads. */

ThreadHandle[0] = (HANDLE) _beginthreadex(NULL, 0L, &TopThread,
                                           &TParTopLeft, 0L, &(ThreadId[0]));
ThreadHandle[1] = (HANDLE) _beginthreadex(NULL, 0L, &BotLeftThread,
                                           &TParBotLeft, 0L, &(ThreadId[1]));
ThreadHandle[2] = (HANDLE) _beginthreadex(NULL, 0L, &TopThread,
                                           &TParTopRight, 0L, &(ThreadId[2]));
ThreadHandle[3] = (HANDLE) _beginthreadex(NULL, 0L, &BotRightThread,
                                           &TParBotRight, 0L, &(ThreadId[3]));

/* Send events to trigger operation of top left and top right threads. */
MappControlThread(EventWaitTopLeft, M_EVENT_SET, M_SIGNALED, M_NULL);
MappControlThread(EventWaitTopRight, M_EVENT_SET, M_SIGNALED, M_NULL);

/* Report what has happened to the Host screen. */
printf("Processing done in a loop.\n");
printf("Press <Enter> to continue.\n");
getchar();
/* Make all threads exit. */
MappControlThread(EventEndTopLeft, M_EVENT_SET, M_SIGNALED, M_NULL);
MappControlThread(EventEndTopRight, M_EVENT_SET, M_SIGNALED, M_NULL);

/* Wait until all threads are finished before freeing MIL objects. */
while ((MappControlThread(EventEndTopLeft, M_EVENT_STATE,
                          M_DEFAULT, M_NULL) == M_SIGNALED) ||
       (MappControlThread(EventEndTopRight, M_EVENT_STATE,
                          M_DEFAULT, M_NULL) == M_SIGNALED) )
;
printf("Top left iterations done:   %4ld.\n", NumberOfTopLeft);
printf("Bottom left iterations done: %4ld.\n", NumberOfBotLeft);
printf("Top right iterations done:    %4ld.\n", NumberOfTopRight);
printf("Bottom right iterations done: %4ld.\n", NumberOfBotRight);
printf("Press <Enter> to end.\n");
getchar();

/* Free buffers. */
MappControlThread(EventSendTopLeft, M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventSendTopRight, M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventWaitTopLeft, M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventWaitTopRight, M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventEndTopLeft, M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventEndTopRight, M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventEndBotLeft, M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventEndBotRight, M_EVENT_FREE, M_DEFAULT, M_NULL);

```

(cont...)

```

    MbufFree(MilTopLeftImage);
    MbufFree(MilTopRightImage);
    MbufFree(MilBotLeftImage);
    MbufFree(MilBotRightImage);
    MbufFree(MilChild);
    /* Release defaults. */
    MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

while (!Exit)
{
    /* Wait for event to process. */
    MappControlThread(EventWaitId, M_EVENT_WAIT, M_DEFAULT, M_NULL);

    /* Process. */
    MimArith(SrcImageId, 5L, DstImageId, M_ADD_CONST);

    /* Increment iteration count and draw text. */
    *((((THREAD_PARAM *) TParam)->NumberOfIterPtr))+= 01L;
    ltoa(*(((THREAD_PARAM *) TParam)->NumberOfIterPtr), Text, 10);
    MgraText(M_DEFAULT, DstImageId, STRING_POS_X, STRING_POS_Y, Text);

    /* Modify communication variable. */
    *((((THREAD_PARAM *) TParam)->ComVarPtr)) += 10L;

    /* Check if processing must be terminated. */
    if (MappControlThread(EventEndId, M_EVENT_STATE, M_DEFAULT,
                          M_NULL) == M_SINGALED)
    {
        /* Make bottom thread exit. */
        MappControlThread(EventEndBotId, M_EVENT_SET, M_SINGALED, M_NULL);

        /* Set exit loop flag. */
        Exit=1;
    }
    /* Synchronize main thread with end of processing. */
    MappControlThread(EventSendId, M_EVENT_SET, M_SINGALED, M_NULL);
}

/* Wait before freeing MIL objects that all threads are finished. */
while (MappControlThread(EventEndBotId, M_EVENT_STATE,
                          M_DEFAULT, M_NULL) == M_SINGALED)
;

/* Make sure exit of thread is synchronized with HOST. */
MappControlThread(EventEndId, M_EVENT_SET, M_NOT_SINGALED, M_NULL);
return(1L);
}

```

(cont....)

```

/* Bottom left functions (Rotate): */
/* ..... */
unsigned long MFTYPE BotLeftThread(void *TParam)
{
    MIL_ID SrcImageId   = ((THREAD_PARAM *) TParam)->SrcImageId;
    MIL_ID DstImageId   = ((THREAD_PARAM *) TParam)->DstImageId;
    MIL_ID EventSendId  = ((THREAD_PARAM *) TParam)->EventSendId;
    MIL_ID EventWaitId  = ((THREAD_PARAM *) TParam)->EventWaitId;
    MIL_ID EventEndId   = ((THREAD_PARAM *) TParam)->EventEndId;
    char    Text[STRING_LENGTH_MAX] = STRING_BOTTOM;
    long    Exit=0;

    while (!Exit)
    {
        long i;

        /* Wait for event to process. */
        MappControlThread(EventWaitId, M_EVENT_WAIT, M_DEFAULT, M_NULL);

        /* Process. */
        MimRotate(SrcImageId, DstImageId, *(((THREAD_PARAM *) TParam)->ComVarPtr)%360,
                  M_DEFAULT, M_DEFAULT, M_DEFAULT, M_DEFAULT,
                  M_NEAREST_NEIGHBOR+M_CLEAR);

        /* Increment iteration count and draw text. */
        (*(((THREAD_PARAM *) TParam)->NumberOfIterPtr))+= 01L;
        ltoa(*(((THREAD_PARAM *) TParam)->NumberOfIterPtr), Text, 10);
        MgraText(M_DEFAULT, DstImageId, STRING_POS_X, STRING_POS_Y, Text);

        /* Check if processing must be terminated. */
        if (MappControlThread(EventEndId, M_EVENT_STATE, M_DEFAULT,
                              M_NULL) == M_SIGNED)
        {
            /* Set exit loop flag. */
            Exit=1;
        }

        /* Synchronize main thread with end of processing. */
        MappControlThread(EventSendId, M_EVENT_SET, M_SIGNED, M_NULL);
    }

    /* Make sure that exit of thread is synchronized with HOST. */
    MappControlThread(EventEndId, M_EVENT_SET, M_NOT_SIGNED, M_NULL);
    return(1L);
}

```

(cont....)

```

/* Bottom right function (Edge Detect): */
/* ..... */
unsigned long MFTYPE BotRightThread(void *TParam)
{
    MIL_ID SrcImageId  = ((THREAD_PARAM *) TParam)->SrcImageId;
    MIL_ID DstImageId  = ((THREAD_PARAM *) TParam)->DstImageId;
    MIL_ID EventSendId = ((THREAD_PARAM *) TParam)->EventSendId;
    MIL_ID EventWaitId = ((THREAD_PARAM *) TParam)->EventWaitId;
    MIL_ID EventEndId  = ((THREAD_PARAM *) TParam)->EventEndId;

    char  Text[STRING_LENGTH_MAX] = STRING_BOTTOM;
    long  Exit=0;

    while (!Exit)
    {
        /* Wait for event to process. */
        MappControlThread(EventWaitId, M_EVENT_WAIT, M_DEFAULT, M_NULL);

        /* Process. */
        MimConvolve(SrcImageId, DstImageId, M_EDGE_DETECT);

        /* Increment iteration count and draw text. */
        (*((THREAD_PARAM *) TParam)->NumberOfIterPtr)+= 01L;
        ltoa(*((THREAD_PARAM *) TParam)->NumberOfIterPtr), Text, 10);
        MgraText(M_DEFAULT, DstImageId, STRING_POS_X, STRING_POS_Y, Text);

        /* Check if processing must be terminated. */
        if (MappControlThread(EventEndId, M_EVENT_STATE, M_DEFAULT,
                               M_NULL) == M_SINGALED)
        {
            /* Set exit loop flag. */
            Exit=1;
        }

        /* Synchronize main thread with end of processing. */
        MappControlThread(EventSendId, M_EVENT_SET, M_SINGALED, M_NULL);
    }

    /* Make sure that exit of thread is synchronized with HOST. */
    MappControlThread(EventEndId, M_EVENT_SET, M_NOT_SINGALED, M_NULL);
    return(1L);
}

```

Chapter 25: Using MIL with Native Mode Functions

This chapter covers the use of Native Mode functions with MIL.

Integrating native functions with MIL code

MIL allows you to mix board-specific code (from the native library function set) with its own code. This is useful when you need to access some board-specific functionality that is not supported directly by the MIL function set or to optimize a time-critical piece of code.

When programming in native mode through MIL, you use the same board driver and programmer's kit that are used by regular native mode programmers. The only difference is the need to use certain rules and commands to ensure proper communication between MIL and the native functions. These rules and commands allow you enter and leave native mode from MIL and access MIL for information, such as the object native handle, concerning data objects on the target board.

Portability

You should note that applications containing native mode functions are not portable to other present or future Matrox platforms supported by MIL.

Signaling MIL about Native Mode use

MIL must be signaled when entering and leaving native mode and when MIL objects have been modified while in native mode, using *MsysControl()*. For buffer modification, *MbufControl()* can also be used to signal MIL.

On entering native mode, MIL does not affect the current state of either the board or the environment.

The *M...Inquire()* functions can be used to determine the buffer, digitizer, or display native identifier (handle) required to use the system's native library.

On leaving native mode, MIL assumes that the board is in the same state as when entering. Therefore, you must ensure that you return the board to the proper state before returning control to MIL. Inquiries about the board state must be made using the board's native library inquiry functions.

A native mode example

In this example, we use MIL mixed with Genesis native library code to grab and warp an image.

Code

```

/* File name: mnatgen.c
 * Synopsis: This program shows how to use GENESIS native library
 *           function calls mixed with MIL function calls.
 */

/* general includes */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mil.h>
#include <imapi.h>

/* Operation control defines */
#define ALLOCATE 1
#define PROCESS 2
#define FREE 3

/* Native functions to grab and warp an image. */
void GrabAndWarp(MIL_ID MilSystem, MIL_ID MilDisplay, MIL_ID MilCamera,
                 MIL_ID MilImage, long Operation);

/* Main function: */
void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem,             /* System identifier. */
    MilDisplay,            /* Display identifier. */
    MilCamera,             /* Camera identifier. */
    MilImage;              /* Image buffer identifier. */

    /* Allocate defaults. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                    &MilCamera, &MilImage);

    /* Allocate and initialize work buffers */
    GrabAndWarp(MilSystem, MilDisplay, MilCamera, MilImage, ALLOCATE);

    /* Print a message on the host screen. */
    printf("Native function called in a loop...\n");
    printf("Press <Enter> to end...");
}

```

(cont. ...)

```

/* Grab and warp grabbed image in a loop */
while (!kbhit())
{
    GrabAndWarp(MilSystem, MilDisplay, MilCamera, MilImage, PROCESS);
}

/* Free work buffers */
GrabAndWarp(MilSystem, MilDisplay, MilCamera, MilImage, FREE);

MappFreeDefault(MilApplication, MilSystem, MilDisplay, MilCamera,
                MilImage);
}

/* Native function: */
/* ..... */
void GrabAndWarp(MIL_ID MilSystem, MIL_ID MilDisplay, MIL_ID MilCamera,
                MIL_ID MilImage, long Operation)
{
    /* Warp coefficient and LUT ID variables.
     * (kept in static to avoid warp coefficient calculation at each call.)
     */
    static long NativeWarpBufId      = M_NULL;
    static long NativeWarpLutXBufId  = M_NULL;
    static long NativeWarpLutYBufId  = M_NULL;
    static long NativeGrabBufId      = M_NULL;
    static long NativeWarpResultBufId = M_NULL;

    /* Inquire useful MIL information. */
    long SizeX  = MdigInquire(MilCamera, M_SIZE_X, M_NULL);
    long SizeY  = MdigInquire(MilCamera, M_SIZE_Y, M_NULL);
    long SizeBand = MdigInquire(MilCamera, M_SIZE_BAND, M_NULL);

    /* Miscellaneous local variables */
    double CornerX1 = 0.0;
    double CornerY1 = 0.0;
    double CornerX2 = SizeX - 1.0;
    double CornerY2 = 0.0;
    double CornerX3 = 2.0 * SizeX;
    double CornerY3 = SizeY - 1.0;
    double CornerX4 = -1.0 * SizeX;
    double CornerY4 = SizeY - 1.0;
    long SrcXStart = 0L;
    long SrcYStart = 0L;
    long SrcXEnd   = SizeX - 1L;
    long SrcYEnd   = SizeY - 1L;

```

(cont. ...)

```

/* Inquire Genesis native Id's */
long NativeSysThreadId = MsysInquire(MilSystem, M_NATIVE_THREAD_ID, M_NULL);
long NativeDigCameraId = MdigInquire(MilCamera, M_NATIVE_CAMERA_ID, M_NULL);
long NativeDigControlId = MdigInquire(MilCamera, M_NATIVE_CONTROL_ID,
M_NULL);
long NativeDigId = MdigInquire(MilCamera, M_NATIVE_ID, M_NULL);
long NativeBufId = MbufInquire(MilImage, M_NATIVE_ID, M_NULL);

/* Notify MIL that we are entering native mode. */
MsysControl(MilSystem, M_NATIVE_MODE_ENTER, M_NULL);

/* Do the selected operation.*/
switch (Operation)
{
/* Preallocate grab and warp buffers (done once for speed). */
case ALLOCATE:
{
imBufAlloc(NativeSysThreadId, SizeX, SizeY, SizeBand, IM_UBYTE,
IM_PROC, &NativeGrabBufId);
imBufAlloc(NativeSysThreadId, SizeX, SizeY, SizeBand, IM_UBYTE,
IM_PROC, &NativeWarpResultBufId);
imBufAlloc(NativeSysThreadId, 3L, 3L, 1L, IM_FLOAT, IM_PROC,
&NativeWarpBufId);
imBufAlloc(NativeSysThreadId, SizeX, SizeY, 1L, IM_SHORT, IM_PROC,
&NativeWarpLutXBufId);
imBufAlloc(NativeSysThreadId, SizeX, SizeY, 1L, IM_SHORT, IM_PROC,
&NativeWarpLutYBufId);
if (NativeGrabBufId && NativeWarpResultBufId && NativeWarpBufId &&
NativeWarpLutXBufId && NativeWarpLutYBufId)
{
/* Calculate warp coefficients */
imGenWarp4Corner(NativeSysThreadId, NativeWarpBufId, CornerX1,
CornerY1, CornerX2, CornerY2, CornerX3, CornerY3,
CornerX4, CornerY4, SrcXStart, SrcYStart,
SrcXEnd, SrcYEnd, IM_DEFAULT, 0L);
imGenWarpLutMatrix(NativeSysThreadId, NativeWarpLutXBufId,
NativeWarpLutYBufId,
NativeWarpBufId, 0L, 0L);
}
else
{
printf("Error allocating resources...\n");
}
break;
}
}

```

(cont....)

```

/* Grab and Warp buffer. */
case PROCESS:
{
    /* Process if allocations were successful */
    if (NativeGrabBufId && NativeWarpResultBufId && NativeWarpBufId &&
        NativeWarpLutXBufId &&NativeWarpLutYBufId)
    {
        /* Grab the image */
        imDigGrab(NativeSysThreadId, NativeDigId, NativeDigCameraId,
            NativeGrabBufId, 1L, NativeDigControlId, 0L);

        /* Warp the grabbed image. */
        imIntWarpLut(NativeSysThreadId, NativeGrabBufId, NativeWarpResultBufId,
            NativeWarpLutXBufId, NativeWarpLutYBufId, 0L, 0L);

        /* Copy the result into the display buffer */
        imBufCopy(NativeSysThreadId, NativeWarpResultBufId, NativeBufId, 0L,
            0L);
    }
    break;
}

/* Free grab and warp buffers. */
case FREE:
{
    if (NativeGrabBufId)
        imBufFree(NativeSysThreadId, NativeGrabBufId);
    if (NativeWarpResultBufId)
        imBufFree(NativeSysThreadId, NativeWarpResultBufId);
    if (NativeWarpBufId)
        imBufFree(NativeSysThreadId, NativeWarpBufId);
    if (NativeWarpLutXBufId)
        imBufFree(NativeSysThreadId, NativeWarpLutXBufId);
    if (NativeWarpLutYBufId)
        imBufFree(NativeSysThreadId, NativeWarpLutYBufId);
    break;
}

/* Notify MIL that we leave native mode. */
MsysControl(MilSystem, M_NATIVE_MODE_LEAVE, M_NULL);

/* Notify MIL that the buffer was modified. */
MbufControl(MilImage, M_MODIFIED, M_DEFAULT);
}

```

Index

A

- absolute value
 - image 74
 - result of operation 88
- absolute world coordinate system 130
- AC Huffman table 379
- accentuating edges 70
- acceptance level
 - definition 194
- acquisition
 - attribute 273
 - continuous 43
 - image 42, 340
 - input LUT 369
 - precondition 350
- adding, image 74
- address
 - Host 290
 - logical 290
- alignment
 - angular 177
 - fiducial marks 182
 - image rotation 59
 - vertical and horizontal 173
- allocate
 - application 29
 - buffers 39
 - child buffer 277, 371
 - data buffer 270
 - defaults 30
 - digitizer 42, 340
 - display 302
 - graphics context 334
 - image buffer 37, 273
 - LUT buffer 76, 297–298
 - measurement marker 238
 - multi-band buffer 368
 - OCR result buffer 212
 - pattern matching model, manual 183
 - pattern matching result buffer 184
- allocation error 276

- angle
 - marker 263
 - orientation 178
- angular alignment 177
- annotation
 - image 334
 - non destructive 313
- application
 - building 29
 - child 388
 - processing, typical 48
 - simultaneous processing 386
- application context 388
- arcs, draw 336
- arithmetic operations 74
- aspect ratio 54, 150, 152, 159
 - definition 151
- auto-focusing 361
- average, input data 55–56

B

- background color
 - associate to graphics context 335
- background, blobs 138
- bar codes 230
- bicubic interpolation 116–117
- bilinear interpolation 116–117
- binarize 48, 63, 66, 148
- Binary buffers 284
- binary buffers, packed 272
- binary measurements, blobs 139, 156
- bisection strategy 362–363
- blanking, display 307
- blob analysis 137–138, 147, 155
 - all-blob interpretation 153
 - area 158
 - binary measurements 139, 156
 - blob location 166
 - blob-group interpretation 153
 - calculating with blob runs 170
 - capabilities 21
 - compactness 162
 - controls 140, 150
 - coordinates 152
 - count by label 65
 - example, compactness 163

- example, count blobs 142
- feature list 140–141, 156, 160
- feature selection 138
- foreground 148, 150
- grayscale measurements 139, 156
- module 139
- non-calibration of results 158
- number of holes 165
- perimeter 158
- result buffer 140, 154
- results 152
- roughness 162
- selecting blob features 156
- selecting blobs 153–154
- single-blob interpretation 152
- speed 154, 156–157, 161
- steps to performing 139
- transformation 145
- blob identifier image 150
 - acquisition 148
 - background 138
 - blob group 150, 152
 - definition 140
 - foreground 138, 148
 - identifier type 150
 - interpretation 150
 - labelled 152
 - lattice 150
 - noise 149
 - pixel aspect ratio 150–151, 159
 - preprocess 140, 149
 - purpose 139, 148
 - segmentation 148
- blobs
 - area 82, 158
 - binary, feature extraction 139
 - border-touching 145
 - breadth 161
 - break apart 58, 77, 140, 149
 - calculate features 152
 - center of gravity 167
 - compactness 162
 - convex perimeter 159
 - counting 142
 - definition 21, 138
 - dimensions 160
 - distinguish 140
 - feature extraction 156
 - feature ordering 138
 - features 139, 141, 155–157
 - Feret diameter 150–151, 160
 - Feret diameter, See also Feret diameter 156
 - grayscale, feature extraction 139
 - grouping 152
 - holes 149, 158, 165
 - identifier types 139
 - identifying 148
 - including See blob identifier image 157
 - label 82, 169
 - locate 66
 - location 166
 - moments 156, 169
 - noise 149
 - number of 82
 - perimeter 158
 - reconstruction 145
 - roughness 162
 - runs 170
 - selecting 153–154
 - shape 162
 - sizing 161
 - touching 148, 150
- border
 - search accuracy 198
- border handling
 - neighborhood operations 87, 91
- borders
 - blobs touching 145
- brightness, adjust on input 349
- buffer
 - accessing a 289
 - RGB 282
 - storage format 281–284, 286–287
 - user-allocated 290
- buffers
 - address 290
 - binary 284
 - displayable 275
 - grab 274
 - pitch of 290
 - supported 139
 - YUV 284

C

- calibrating images 120
- calibration 158
- camera
 - acquisition from 42, 340
 - adjusting/focusing 43, 361
 - sophisticated 340
 - specification 341
- catchment basins 104
- cell size of code 231, 234
- cellular mapping 99
- center of gravity 142, 167
- central moments 169
- certainty level
 - definition 195
- Chained pixels 167
- Chamfer 3-4 transform 82
- characters, text 338
- Chessboard transform 81
- child application 388
- child buffers 277
 - allocate 277
 - color 371
 - data buffer attributes 277
 - definition 270
 - dimensions 277
 - display 278, 311
 - display multiple 308
 - inheriting parent features 277
 - LUT 297
 - offset from parent 277
 - returned coordinates 277
 - size 277
- circles, draw 336
- City Block transform 81
- clear
 - display 307
 - graphics image buffer 335
- clipping
 - borders 278
 - graphics 337
 - pixel values 68, 149
- closing operation 47, 55, 58
- codes 230
- coefficients, warping 114
- color
 - handling techniques 367
 - input LUT 369
- color band 271
 - LUT 323
- color images
 - allocate buffer 368
 - allocate child buffer 371
 - color conversion 371
 - copy 371
 - copy single band 278
 - dealing with 368
 - displaying 370
 - grabbing 368
 - loading 374
 - processing 371
 - processing restrictions 371
 - put data in band 279
 - reference levels 350
 - saving 374
- column profile 65
- commands
 - functions 30
 - pseudo-MIL 398
- communication channels 29, 32
- compactness, blob 162
- comparative operations 74
- compass gradient 71, 73
- compiling 31
- complex operations 76
- compressing images 376
- conditional buffer, creating 334
- connectivity
 - code 99
 - mapping 99
- constant thresholding 47
- continuous grab 43
- contrast 349
 - image, adjusting 63, 69, 349
 - marker/background 250
- control
 - areas processed 277
 - neighborhood center 91
 - neighborhood operation 87
- conversion
 - color 371
 - data format 280
- convex perimeter 159

- convex perimeter, find 96
- convolution 70–71
- coordinates
 - child buffer 277
 - measurement marker 247
 - model 183
 - of a pixel 293
 - search result 173
 - text writing 338
- copy
 - bit truncation/extension 279
 - clip, and 278
 - color band 278, 298, 371
 - conditional 278
 - data 278
 - data to LUT 298
 - mask 278
 - model 183
 - specific buffer areas 278
- Corona
 - exposure 356, 359–361
 - automatic model 358
 - bypass model 360
 - triggers 356, 359–361
- counting
 - dark particles 67
 - objects 48, 82, 163
- custom
 - morphological operations 90
 - spatial filters 86
 - structuring element 90
 - window, VGA 317

D

- data allocation and access module 269
- data average 56
- data buffer
 - attributes 273, 276
 - automatically allocated 280
 - child 270, 277
 - clear 335
 - clip border 278
 - color band 271, 368
 - defined 270
 - depth 272
 - dimensions 271

- display 301
- export data 280
- free 271
- get data, put in array 279
- handling 269
- import 280
- incorrect usage 276
- integer 272
- intended usage 273
- location 273
- LUT, see LUT buffer 297
- management 279
- multiple, display 278
- multiple, handling 278
- packed binary 272
- put data 279
- range 272
- restore 280
- save 280
- type 68, 273
- data format, input device 341
- data generation
 - LUT 297
- data type 39
 - changing 68
- data, overwriting 39
- DataMatrix codes 230
- DC Huffman table 379
- dcf files 341
- decompressing images 376
- default graphics context 334
- defaults
 - display 30
 - image buffer 26, 30, 39, 41
 - initializing 26
 - input device 340
 - input LUT 351
- defect highlighting 70
- depth
 - data buffer 272
 - display 306
- destination buffer 39
- device
 - control module 339
- differences
 - image, count 62
- digitization, definition 46

- digitizer
 - allocate 42, 340
 - color data format 368
 - configuration format 341
 - frame averaging 56
 - free 340
 - input channel 342
 - inquire 341
 - LUT 300, 350, 369
 - number 342
 - reference levels 349
- digitizer configuration files 341
- dilation
 - advanced 91
 - basic 47, 77–78
 - binary algorithm 92
 - conditional 92, 145
 - grayscale algorithm 92
 - opening/closing operation 58
- dimensions, blob 160
- DirectDraw 324, 327
- DirectDraw underlay-surface 327
- DirectDraw underlay-surface display 327
- DirectDraw underlay-surface display
 - architecture 327
- displacement
 - search maximum 197
- display 324
 - allocation 29
 - annotating 312
 - Windows GDI 314
 - border handling 302
 - buffer 39, 278
 - clear 307
 - color 322
 - color image 370
 - control module 301
 - default 30
 - dual-screen 303
 - example 313, 318
 - free 318
 - image location 302
 - LUT 299, 322, 370
 - mode
 - non-windowed 305
 - windowed 305
 - monochromatic effect 300
 - monochrome buffer 38
 - multi-head 304
 - multiple buffers 308
 - non 8-bit buffers 306
 - pan 311
 - pseudo-color effect 300
 - psuedo color LUT 323
 - scroll 311
 - single-screen 303
 - size and depth 306
 - true color effect 300
 - user-defined window 317
 - VGA 303
 - VGA system 317
 - Windows, VGA 317
 - zoom 311
- display architectures 324
- display mode
 - windowed 328
- Displayable buffers 275
- distance transforms 81
- distinguishing edges 70
- distortions 54–55, 59, 120
- dividing, image 74
- dontcare, kernel value' 90
- dots, draw 336
- double buffering, definition 352
- DrawDIBDraw()
 - VGA 328–330
- drawing 334, 336
- dual-screen configuration
 - displaying in 303
 - VGA 303
- dynamic range 349

E

edge

- enhance for contrast 70
- enhancers 47, 70–71, 86
- inside stripe marker 259
- markers 237

edge extractors 72

- compass gradient 71, 73
- definition 70
- horizontal 70–71
- Laplacian 71–72
- oblique 70
- predefined kernels 71
- vertical 70–71

encoding type 231, 234

erosion 47

- advanced 91
- basic 77, 90
- binary algorithm 91
- grayscale algorithm 91
- opening/closing operation 58

error correction type 231, 234

error reporting

- automatic 41
- memory, insufficient 276
- message control 30
- thread 388–389

event, locate 62

examples

- blob analysis, compactness 163
- blob analysis, count blobs 142
- change data type 68
- color image manipulation 372
- color, run with 369
- custom structuring element 93
- display in user-defined window 318
- display multiple buffers 308
- display with overlay 313
- extract background 74
- filter with custom kernel 88
- find perimeter of object 78
- general information 26, 32
- grab 42
- image allocation/display 40
- image analysis 48
- installing 26

kernel 86

mblobcnt.c 163

mblobcog.c 142

mcolor.c 372

mconvol.c 88

MIL sample program 32

mmultidis.c 308

mocrfont.c 225

mocrread.c 213

mocrview.c 223

modify for color 38

mopen.c 93, 96

mperim.c 78

msearch.c 185

mstart.c 32

msurvey.c 74

mwindisp.c 318

Native Mode ProgrammersToolkit' 399

OCR, calibrate a font 213

OCR, create custom font 225

OCR, visualize font characters 223

optical character recognition 213, 219

pattern matching, define model and search 185

standard defaults 38

structuring element 92

wafer alignment 174

zooming 311

excluding blobs 154, 158

export data buffer 280

exposure 361

automatic model

Corona 358

bypass model

Corona 360

Corona 356

extreme value, find 49, 62, 65

F

- false matches 195
- feature list, blob analysis 140–141
- Feret diameter
 - angle 161
 - aspect ratio 151
 - average 160
 - blob feature 156
 - convex perimeter 159
 - general 160
 - illustrated 160
 - minimum/maximum 160
 - number of 150
- fiducial marks 182
- field grabbing 343
- field-of-view 133
- file format 280
- files
 - semi1292.mfo 216
 - semi1388.mfo 216
- fill
 - holes 58
- filled-in shapes 337
- filter
 - temporal 56
- find
 - buffer extremes 62
 - marker 240, 248
 - model 184
- first-order polynomial warping 114
- focus indicator 362
- font
 - associate to graphics context 338
 - predefined 338
 - scale 338
 - size 338
- foreground color
 - associate to graphics context 335
 - fill with 337
- foreground, blobs 138, 150
- frame
 - averaging 56
 - grabbing 43, 343
- frame buffer 302
- frame buffer, display 313

- free
 - buffer, data 271
 - buffer, image processing result 65
 - graphics context 334
- full range digitization 65
- function
 - development 399
 - execution success 30
 - user-created 22
- functions
 - commands 30

G

- gamma correction 300
- Gaussian noise 54–57
- generating warping coefficients 114
- geometric transforms 113
- Grab 274
- grab 351
 - color images 368
 - continuous 43
 - data average 55–56
 - data buffer 273, 368
 - example 42
 - fields 343
 - frames 43, 343
 - halt 43
 - image 42, 48, 340
 - mode 352
 - monochrome 42
 - multi-dimensional buffers 368
 - sequence 56
 - synchronization 352
- Grab buffers 274
- GrabAndWarp()
 - example 399
- graphics 334
 - arcs, draw 336
 - boundary type seed fill 337
 - buffer, clear 335
 - capabilities 21
 - circles, draw 336
 - clipping 337
 - dots, draw 336
 - filled elliptic arcs, draw 336
 - filled rectangles, draw 336

- filled-in shapes 336
- lines, draw 336
- module 333
- non-destructive annotation 313
- outline, draw 336
- parameters 335
- plot a histogram 64
- rectangles, draw 336
- text, write 338
- graphics context
 - allocate 334
 - background color, associate 335
 - default 334
 - definition 334
 - font scale, associate 338
 - foreground color, associate 335
 - free 334
 - object parameters 335
 - text font, associate 338
- grayscale images, modules using only 20

H

- halt grabbing 43
- header file 31
- hierarchical search 204
- histogram
 - equalization 69
 - generate 62–63
 - plotting 64
 - statistical operation 47
- hit or miss pattern matching 98
- holes
 - blobs, in 158
 - blobs, to distinguish 165
 - extract 145
 - fill 55, 58, 145
- horizontal edges 70–71
- host
 - communication 29
 - CPU 20
 - default system 271
 - screen 30
 - system 298
- hue 371
- Huffman encoding 378–379

I

- image
 - addition 74
 - analysis 47
 - arithmetic operations 74
 - binarize 66
 - column profile 65
 - comparison 47
 - contrast 63
 - data type 68
 - definition 46
 - differences, count 62
 - dilation 77
 - distortion 55, 59, 194
 - division 74
 - edge enhancement 70
 - enhancement 47
 - erosion 77
 - extreme values 62, 65
 - grabbing 42
 - histogram 62–63
 - histogram equalization 69
 - inversion 76
 - manipulation, advanced 85
 - manipulation, basic 47
 - manipulation, order 62
 - mapping 47, 76
 - multiplication 74
 - noise reduction 47
 - pixel value clipping 68
 - projection 62, 65
 - quality 54–55
 - row profile 65
 - sharpening 47
 - smoothing 47
 - statistics 62
 - subtraction 47, 74
 - thresholding 66
 - transformation 145

- image buffer
 - acquisition 273
 - allocation 37, 39–40, 273
 - color 38, 43
 - conditional 334
 - default 30, 39, 369
 - defined 39
 - destination 39
 - display 40, 273
 - display border 302
 - display multiple 308
 - display position 302
 - free 271
 - map through LUT 296
 - multi-band 371
 - processing 273
 - removing from display 307
 - select for display 302
 - size 39
 - source 39
 - two-dimensional 38–39, 43
 - uses 39
- image coordinate system 130
- image digitization 46
- image processing 46–47
 - application 48
 - capabilities 20
 - module 45, 53, 61
 - operation ordering 62
 - result buffer 63, 65–66
- imIntDistance() 81
- impulse noise 54
- include file 31
- initialization
 - input device 340
 - system 26, 340
- input device
 - brightness 349
 - contrast 349
 - defaults 340
 - frequency 341
 - line-scan 343
 - LUT 350
 - resolution 341
 - subsampling 351
 - using 42

- inquire
 - digitizer 341
 - font, OCR 222
- installation
 - MIL 26
 - test program 31
- integer buffers 272
- Intellicam 341
- intensity
 - correction 298
 - distribution 63, 69
 - histogram 62
 - HLS 371
- interlaced JPEG compression 376
- interpolation modes 116
- isthmuses 55, 58

J

- JPEG compression 376

K

- kernels
 - buffer allocation 87, 273
 - defined 57
 - example, sharpen/smooth 86
 - predefined 57, 71
 - usage 70, 72–73, 86
 - user-defined 87
- keying 313

L

- labelling
 - blobs 48–49, 169
 - blobs, use 65, 82
- Laplacian edge detection 71–72
- lattice, blobs 150
- length
 - measurement marker 250, 257
- lens curvature 54
- lens motor 361
- line equation
 - measurement marker 250, 258
- linear interpolation function 129

- lines
 - draw 336
- line-scan device 343
- link program with library 31
- load
 - color image 374
 - data 279–280
 - LUT data 298
- locating blobs in an image 66, 166
- logical operations 74
- look-up table 321
 - 1-band custom 323
 - 3-band custom 324
 - changing default 322
 - control loading into physical output
 - LUTs 330
 - control loading into Windows
 - palette 328, 330
 - pseudo-color 323
- lossless compression 376
- lossy compression 376
- low-pass spatial filters 55
- luminance
 - HLS 371
- LUT 321
 - 1-band custom 323
 - 3-band custom 324
 - changing default 322
 - control loading into physical output
 - LUTs 330
 - control loading into Windows
 - palette 328, 330
 - pseudo-color 323
- LUT buffer
 - allocation 273, 297
 - child buffer 297
 - color bands 297–298, 369
 - data generation 297–298
 - dimensions 297
 - load 298
 - management 297
 - one-dimensional 297
 - restore 298

- LUTs
 - allocate 76
 - connectivity mapping 99
 - custom 323
 - definition 296
 - display 299, 370
 - display color, change 322
 - general information 295
 - index 297
 - input 349–350, 369
 - input mapping 76, 300
 - intensity correction 298
 - monochromatic effect 300
 - multiple-color-band 324
 - one-color-band 323
 - processing 76, 299
 - pseudo-color effect 300
 - ramp 297
 - transformation 69
 - true-color effect 300
 - usage 299

M

- M_DISP 275
- M_GRAB 274
- machine guidance 166, 182
- MappAlloc() 30, 334, 388
 - example 318
- MappAllocDefault() 26, 29, 39, 41–42, 302, 334, 340, 368–370
 - example 32, 40, 42–43, 50, 74, 79, 88, 93, 142, 185, 213, 223, 225, 308, 399
- MappChild() 388
- MappControl() 30
- MappControlThread() 389
- MappFree() 30
 - example 318
- MappFreeDefault() 29
 - example 32, 40, 42–43, 50, 64, 74, 79, 88, 93, 142, 163, 174, 185, 223, 225, 308, 372, 399
- MappGetError() 30, 389
 - example 40, 174
- MappHookFunction() 30
- mapping 76, 99
- mapping pixels to world 120

- marker
 - allocation 238
 - center 249, 254
 - characteristics 239, 248
 - contrast 250
 - edge 237
 - find 240, 248
 - find, speed 244, 250
 - find, tolerance 245
 - managing 237
 - measure 240
 - measurement box 244–245, 257
 - parameters 239, 248
 - processing area 239
 - reference 249
 - reference position 254
 - region of interest 244
 - stripe 237
 - types 237
- mask, copy 278
- matrix-defined warping 115
- matrix-defined warpings
 - generating LUTs for 115
- Matrox READER 211
- maximum
 - pixel value 47, 65
- MblobAllocFeatureList() 140, 156
 - example 142, 163
- MblobAllocResult() 140
 - example 142, 163
- MblobCalculate() 141, 154, 156, 167
 - example 142, 163
- mblobcog.c 142
- MblobControl() 140, 150–153, 159, 161
 - example 163
- MblobFill() 146, 154, 157
- MblobFree()
 - example 142, 163
- MblobGetLabel() 141, 169
- MblobGetNumber() 141
 - example 142, 163
- MblobGetResult() 141
 - example 142
- MblobGetResultSingle() 141, 169
- MblobGetRuns() 141, 169
- MblobLabel() 146, 157
- MblobReconstruct() 92, 145
- MblobSelect() 141, 154, 157
 - example 142, 163
- MblobSelectFeature() 141, 156–157, 160–161, 169
 - example 142, 163
- MblobSelectFeret() 141, 156, 160
- MblobSelectMoment() 141, 156, 169
- MbufAlloc() 289
- MbufAlloc1d() 76, 270, 297, 323
- MbufAlloc2d() 39, 87, 90, 270, 308
 - example 40, 74, 79, 93, 142, 318
- MbufAllocColor() 30, 270, 281, 297, 368
- MbufChild1d() 277
- MbufChild2d() 277, 308
 - example 50, 88, 93, 185, 213, 225, 308, 372
- MbufChildColor() 277, 371, 378
 - example 372
- MbufClear() 335
 - example 225, 308, 318
- MbufControl 314
- MbufControl() 314, 323, 377–378, 380
 - example 399
- MbufControlNeighborhood() 87, 91
 - example 88
- MbufCopy() 298, 377
 - example 74, 93
- MbufCopyClip() 278
- MbufCopyColor() 278, 298, 371
- MbufCopyCond() 278
- MbufCopyMask() 278
- MbufCreate2d() 290
- MbufCreateColor() 290
- MbufExport() 280, 374, 377
- MbufExportSequence 376
- MbufFree() 30, 271, 277, 297
 - example 50, 64, 74, 79, 88, 93, 142, 163, 174, 185, 225, 308, 372
- MbufGet() 279, 289
- MbufGet1d() 279
- MbufGetColor() 279
- MbufImport() 280, 374, 377
- MbufImportSequence() 376
- MbufInquire() 290–291, 314
 - example 223, 399
- MbufLoad() 280, 298, 374
 - example 50, 79, 93, 142, 185, 213, 308, 372

- MbufPut() 87, 90, 279, 289, 298, 323
 - example 88
- MbufPut1d() 76, 279, 298
- MbufPut2d() 87, 90
 - example 93
- MbufPutColor() 279, 298
- MbufRestore() 280, 298, 374
- MbufSave() 280, 374
- McalAlloc() 122
- McalAssociate() 122
- McalControl() 125
- McalGrid() 122, 126, 135
- McalList() 122, 126, 128, 135
- McalRelativeOrigin() 132
- McalTransformCoordinate() 122
- McalTransformImage() 122
- McalTransformResult() 122
- McodeAlloc() 231
- McodeControl() 231, 234
- McodeFree() 231
- McodeGetResult() 231
- McodeRead() 231
- McodeWrite() 231
- mconvol.c 88
- MdigAlloc() 30, 42, 340–342, 368
 - example 318
- MdigChannel() 342
- MdigControl() 43, 352, 387
- MdigFocus() 361
- MdigFree() 30, 340, 368
 - example 318
- MdigGrab() 43, 343, 352, 368, 377
 - example 42, 74
- MdigGrabContinuous() 43, 368
 - example 43, 74, 318
- MdigGrabWait() 352
- MdigHalt() 43, 352
 - example 43, 74, 318
- MdigHookFunction() 387
- MdigInquire() 341
 - example 74, 399
- MdigLut() 69, 300, 350, 369
- MdigReference() 350, 369
- MdispAlloc() 30, 39, 302–303, 313, 370
 - example 318
- MdispControl() 306, 313
 - example 313
- MdispDeselect() 39–40, 307, 370
 - example 308, 318
- MdispFree() 30, 307
 - example 318
- MdispHookFunction() 315
- MdispInquire() 315, 323
 - example 313
- MdispLut() 69, 299, 323, 370
- MdispOverlayKey() 313
- MdispPan() 311
- MdispSelect() 39, 275, 278, 302, 308, 370
 - example 308, 372
 - VGA 317
- MdispSelectWindow() 317
 - example 318
- MdispZoom() 311
 - example 308
- measurement context
 - free 239
- measurements
 - angle, marker 263
 - capabilities 21
 - context 239
 - control settings 239
 - length, marker 250, 257
 - line equation 250, 258
 - markers, multiple 256
 - markers, using 240
 - module 235
 - position 249
 - preprocess image 240
 - region of interest 244
 - steps 238
 - width, stripe 257
- measurements, blob
 - binary 139
 - compactness 162
 - dimensions 161
 - discriminate, to 138
 - general 156
 - grayscale 139, 156
 - pixel units 159
 - roughness 162
- median filter 57
- memory
 - insufficient 276
 - resources 29–30
- messages, error 30, 41
- MgenLutFunction() 76, 297–298

- MgenLutRamp() 76, 297–298, 323
- MgenWarpParameter() 114
- MgraAlloc() 334
- MgraArc() 336
- MgraArcFill() 336
- MgraBackColor() 335
- MgraClear() 335
- MgraColor() 335
 - example 142
- MgraDot() 336
- MgraFill() 336–337
- MgraFont() 338
 - example 225
- MgraFontScale() 338
 - example 225
- MgraFree() 334
- MgraLine() 336
 - example 142
- MgraRect() 336
 - example 174, 185
- MgraRectFill() 336
- MgraText() 338
 - example 32, 225, 313, 318
- MIL
 - file format 280
 - header file 31
 - include file 31
 - objects 22
 - running application 31
- MIL Configuration utility 27
- MIL modules
 - blob analysis 137, 147, 155
 - display control 301
 - graphics 333
 - I/O device control 339
 - image processing 45, 53, 61
 - measurements 235
 - pattern matching 171, 181
- mil.h 31
- mil.ini
 - Meteor-II 348
- MILINTER 211
- milsetup.h 26, 29–30, 38, 340, 369
- MimAllocResult() 63, 65–66
 - example 50
- MimArith() 74, 105, 298
 - example 50, 74, 79, 372
- MimBinarize() 66, 68, 148
 - example 50, 79, 93, 142, 163, 308
- MimClip() 66, 68, 82, 149
 - example 68, 372
- MimClose() 58
 - example 142, 163
- MimConnectMap() 99
- MimConvert() 371
 - example 372
- MimConvolve() 57, 71–73, 86
 - example 50, 88, 93
- MimCountDifference() 62
- MimDilate() 58, 78, 92–93
 - example 79
- MimDistance() 105
- MimErode() 58, 77–78, 92
- MimFindExtreme() 62, 65
 - example 50
- MimFlip() 113
- MimFree() 63, 65–66
 - example 50, 64
- MimGetResult() 65–66
 - example 50, 64
- MimHistogram() 62–63
- MimHistogramEqualize() 69, 298
- MimLabel() 65, 83
 - example 50, 82
- MimLocateEvent() 62
- MimLutMap() 69, 76, 296
- MimMorphic() 90, 93, 96
 - example 93
- MimOpen() 58, 93
 - example 50, 79, 93, 142, 163
- MimPolarTransform() 113
- MimProject() 62, 65
- MimRank() 57
- MimResize() 55, 59, 113, 151
- MimRotate() 59, 113
- MimThick() 96
- MimThin() 96
- MimTranslate() 113
 - example 185
- MimWarp() 113
- MimWatershed() 104
 - minimum pixel value 47, 65
- MmeasAllocMarker() 238, 248
- MmeasCalculate() 239, 249
- MmeasFindMarker() 239–240

- MmeasFree() 239
- MmeasInquire() 239
- MmeasRestoreMarker() 238
- MmeasSetMarker() 239, 248, 259
- mmultidis.c 308
- MMX Technology, Intel 24
- mnatgen.c 399
- MocrAllocFont() 224
 - example 225
- MocrAllocResult() 212
 - example 225
- MocrCalibrateFont() 217
- MocrControl() 212, 220
 - example 225
- MocrCopyFont() 212, 222, 224
 - example 223, 225
- mocrfont.c 225
- MocrFree()
 - example 223, 225
- MocrGetResult() 212
 - example 225
- MocrImportFont() 212, 224
- MocrInquire() 219–220, 222
 - example 223
- MocrModifyFont() 217, 222
- mocrread.c 213
- MocrReadString() 212, 217
 - example 225
- MocrRestoreFont() 212, 222, 224
 - example 213, 223
- MocrSaveFont() 217, 222, 224
 - example 225
- MocrSetConstraint() 212, 219, 228
 - example 219, 225
- MocrVerifyString() 212, 217
- mocrview.c 223
- model
 - acceptance level 194
 - allocation, automatic 173
 - allocation, manual 183
 - center 173, 197
 - coordinates 173, 183
 - copy to image buffer 183
 - default search parameters 184
 - dontcarepixels' 202
 - find 173, 194
 - load 184
 - number of matches 194
 - positional accuracy 198, 200
 - preprocess 199–200
 - reference point 197
 - rotate 183, 194
 - search parameters 184
 - size 183, 200
 - storage location 183
 - view 183
- moments 156, 169
 - central 169
 - ordinary 169
- monochromatic effect 300
- monochrome image buffer 39
- mopen.c 93
- morphological operations
 - binary 90
 - custom 90
 - erosion/dilation 77
 - grayscale 90
 - standard 47, 58
- MpatAllocAutoModel() 173, 183
- MpatAllocModel() 179, 183
 - example 185
- MpatAllocResult() 173, 178, 184
 - example 174, 185
- MpatCopy() 183
- MpatFindModel() 173, 184–185, 195–196, 205
 - example 174, 185
- MpatFindOrientation() 178–179
- MpatFree() 184
 - example 174, 185
- MpatGetNumber() 174, 184, 195
 - example 174
- MpatGetResult() 178–179, 184, 203
 - example 174, 185
- MpatInquire() 173
 - example 174, 185
- MpatPreprocModel() 173, 199–200, 206
 - example 185
- MpatSave()
 - example 185
- MpatSetAcceptance() 194
- MpatSetAccuracy() 198, 201, 207
 - example 185
- MpatSetCenter() 196–197
- MpatSetCertainty() 196
- MpatSetDontCare() 183

- MpatSetNumber() 194–195
- MpatSetPosition() 174, 197, 201
- MpatSetSearchParameter() 205–206
- MpatSetSpeed() 199, 201, 206
 - example 185
- mstart.c 31
- MsysAlloc() 30
 - example 318
- MsysControl()
 - example 399
- MsysFree() 30
 - example 318
- MsysInquire()
 - example 318, 399
- multi-dimensional buffers 368
- multi-head configuration
 - displaying in 304
- multiple buffers
 - displaying 308
- multiple fields of view 133
- multiplying, image 74
- multi-processing 385–386
 - definition 386
- multi-threading 385, 387
 - definition 387
- MvgaDispDeselectClientArea()
 - VGA 317
- mwindisp.c 318

N

- native mode 397
 - example code 399
 - integrating with MIL 398
 - interface 398
- nearest-neighbor interpolation 116–117
- neighborhood
 - bordering pixels 88
 - center pixel 87–88, 91
 - convolution method 87
 - dontcarepixels' 90
 - operations 47, 55, 57–58, 70–71, 77, 86, 90, 96
 - overscan 87, 91

- noise 47
 - blobs 149
 - Gaussian 54–56
 - impulse 54
 - random 54, 56–57
 - reduction 57, 86, 140, 149
 - salt-and-pepper 54–55, 57
 - shot 54
 - systematic 54, 57
- non 8-bit buffers
 - displaying 306
- normalization factor 88
- normalized grayscale correlation 202
- number of
 - objects 65
- number of cells in code 231

O

- object orientation 178
- oblique edges 70
- open communication 29, 32
- opening operation 47–48, 55, 58
- optical character recognition 210
 - acceptance levels 220
 - allocating result buffer 212
 - calibrating fonts 212, 217
 - character constraints 212, 219
 - character dimensions 218
 - contrast enhancement 221
 - creating fonts 212, 224
 - erasing characters 221
 - examples 213, 219, 223, 225
 - font files 216–217
 - inquiring about fonts 222
 - inverting fonts 222
 - loading fonts 212
 - managing fonts 222
 - match scores 220
 - processing controls 212, 220
 - READER 211
 - reading results 212
 - reading strings 212
 - restoring fonts 222, 224
 - saving fonts 222, 224
 - semi1292.mfo font file 216
 - semi1388.mfo font file 216

- speeding up 221, 228
- steps 211
- string location 221
- target images 212
- unrecognized characters 221
- verifying strings 212
- visualizing fonts 222
- ordinary moments 169
- orientation
 - image 59, 177
 - object 177–178
 - whole-image 177
- overlay
 - example 313
 - simulated 313
 - usage 313
- Overlay/regular display 326
- Overlay/regular display architecture 326
- overscan 117
 - mirror 88
 - pixels 87, 91
 - transparent 88
- overwriting data 39

P

- packed binary buffers 272
- palette
 - image 297
- panning, display 311
- parent buffer 270, 277
 - display 308, 311
- pattern matching
 - algorithm 202
 - alignment 173
 - basic 66
 - capabilities 21
 - example 174
 - module 171, 181
 - morphological 47, 98
 - peaks 206
 - result buffer 173, 184
 - score 195
 - search steps 182
 - usage 172

- perimeter
 - convex 159
 - normal 158
- perspective transformation function 129
- perspective warping 115
- physical memory 39
 - buffer allocation 273
- picth 290
- pixel
 - area 158
 - aspect ratio 54–55, 140, 150–152, 158–159
 - coordinates 293
 - depth 22
 - dontcare' 183, 202
 - height 159
 - location 62, 90
 - perimeter 158
 - real-world units 158
 - units 159
 - value density of diagonals 65
 - value distribution 69
 - value, minimum/maximum 47
 - value, mininum/maximum 349
- pixel reference position 240
- pixel-to-world mapping 120
- plotting histogram 64
- point-to-point operations 47, 74
- portability
 - native mode 398
- position of marker 249
- positional accuracy 198
- predictive coding 378–379
- preprocess
 - input data 350
 - measurement target image 239
 - model 173, 184, 199–200
- processing
 - attribute 273
 - limiting 277
 - single band 371
 - with LUT 299
- profile 65
- program examples 26
- projecting an image 62, 65

- pseudo-color
 - effect 300
- put data
 - array, from 279

Q

- quantization 380

R

- ramp, LUT 297
- random noise 54, 57
- rank filters 57
- read.me 26–27, 31–32
- READER 211
- reading codes 230
- real-world results, obtaining 120
- reconstruct object from seed 145
- rectangles, draw 336
- reference level
 - analog 349
 - black/white 65, 349
 - controls 349
 - input channel 349
- reference point, model 197
- reference position 249
 - measurement marker 254
- refocus strategy 364
- relative camera position 131, 134
- relative coordinate system 130, 134
- remove
 - holes 55
 - small particles 58
- resize image 59
- resolution pyramid 204
- restart markers 381
- restore
 - fonts, OCR 222
 - LUT buffer 298
- result buffer
 - blob analysis 140–141, 152, 154
 - image processing 63, 65–66
 - pattern matching 173, 184
- RGB
 - buffers 282

- rotate
 - image 59
 - model 183, 194
 - tolerance 201
- roughness, blob 162
- row profile 65

S

- salt-and-pepper noise 54–55, 57
- sample program 31
- saturation
 - HLS 371
 - operation result 88
- save
 - color image 374
 - data 279–280
 - fonts, OCR 222
- scale, input 76, 351
- scaling 351
- scan_all strategy 365
- scrolling, display 311
- search
 - acceptance level 194
 - accuracy 198
 - area 201
 - basic steps 182
 - border effects 198
 - certainty level 195
 - coordinates 173
 - heuristics 206
 - hierarchical 204
 - model 184
 - number of matches 194
 - parameters 184, 194, 205
 - region 197
 - results 195
 - robustness 200
 - speed 183–184, 195, 198–201, 204–206
 - subpixel accuracy 206
- segmentation 148
- semi1292.mfo file, OCR 216
- semi1388.mfo file, OCR 216
- sequence averaging 56
- sharpen image 71
- shift
 - operations 74

- shot noise 54
- single-screen configuration
 - displaying in 303
 - VGA 303
- size
 - child buffers 277
 - data buffer 271
 - display 306
 - image buffer 39
 - LUT buffer 297
 - model 200
 - system display 303
 - text character 338
- skeleton, find 96
- smart_scan strategy 365
- Smatch
 - pattern matching 181
- smoothing 47–48, 57, 86
- software triggers
 - Corona 361
- source buffer 39
- spatial filtering operations
 - algorithm 86
 - custom 86
 - edge enhancement/extraction 70
 - low-pass 55, 57
 - median 55
 - rank effect 57
 - usage 47, 57
- speed
 - marker, find 244, 250
 - model size 200
 - multi-threading 387
 - packed binary processing 272
 - search 184, 195, 198–201, 204–206
 - search, high speed 201
 - search, low speed 201
 - search, medium speed 201
- spurious blobs 149
- square pixels 55
- stop grabbing 43
- storage area 39
- stripe
 - markers 237
 - rising/falling 249
- strobe device 341

- structuring elements
 - buffer allocation 90, 273
 - connectivity 99
 - custom 90
 - default 92
 - defining 90
 - example, custom opening 93
 - example, erosion/dilation 92
 - usage 90, 96, 98
- subpixel accuracy 206
- subsampling input 351
- subtracting, image 74
- synchronization
 - of grab 352
 - thread 387–388
- system
 - allocation 29
 - buffers 39
 - configuration 26
 - default 22, 26
 - definition 20
 - device 299, 368, 370
 - display criteria 39
 - grab criteria 43
 - initialization 26, 340
 - multiple 30
 - multi-processing capabilities 386
- systematic noise 54, 57

T

- target system
 - system 22
- test installation program 32
- text
 - character font 338
 - graphics 338
 - support 334
- thickening 96
 - binary algorithm 97
 - grayscale algorithm 97
- thinning 96
 - binary algorithm 97
 - grayscale algorithm 97

- thread
 - application context 388
 - data sharing 387
 - error reporting 388–389
 - multi-threading 385, 390
 - synchronization 388
- thresholding 63, 66
- toolkit
 - Function Developers' 397
- transformation LUT 69
- transformations
 - generating LUTs for 115
- transforming data 280
- trigger device 341
- triggers 359, 361
 - Corona 356, 360
- true color effect 300
- typical application
 - binarizing 48
 - extreme value 49
 - grabbing 48
 - labelling 49
 - opening 48
 - smoothing 48

U

- Underlay display 325
- Underlay display architecture 325
- uniform distribution 69
- user-allocated buffer 290

V

- vertical edges 70–71
- view model 183

W

- warping 113
 - interpolation modes 116
- warpings
 - generating LUTs for 115
- watershed lines 104
- watershed transforms 104
- width
 - stripe 257
- Window occlusion
 - Meteor-II 349
- Windows
 - custom window, VGA 317
- working area 127
- world coordinate system 130
- writing codes 230

Y

- YUV buffers 284

Z

- zoom
 - display 311
 - example 311

Product Assistance Request Form

[illegible]

