

# **Matrox Imaging Library**

version 7

## **User Guide**

Manual no. 10513-301-0710

March 1, 2002

*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 MMX™ 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., 2002. All rights reserved.*

*All rights reserved. Limitation of Liabilities: In no event will Matrox or its suppliers be liable for any indirect, special, incidental, economic, cover or consequential damages arising out of the use of or inability to use the product, user documentation or related technical support, including without limitation, damages or costs relating to the loss of profits, business, goodwill, even if advised of the possibility of such damages. In no event will Matrox and its suppliers' liability exceed the amount paid by you, for the product.*

*Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.*

*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, neither Matrox Electronic Systems Ltd. nor its suppliers assume any responsibility for its use; or for any infringements of patents or other rights of third parties resulting from its use. No license is granted under any patents or patent right of Matrox Electronic Systems Ltd.*

*PRINTED IN CANADA*



# Contents

---

## ***Chapter 1: Getting started . . . . . 19***

The MIL package. . . . .	20
MIL and the Intel MMX/SSE technologies . . . . .	23
System requirements . . . . .	24
Getting started. . . . .	25
Installation. . . . .	25
Building an application. . . . .	28
The MIL user guide. . . . .	32

---

## ***Chapter 2: Allocating an image buffer and grabbing images. . . 33***

Getting started. . . . .	34
Allocating and displaying an image buffer . . . . .	35
Grabbing images . . . . .	38

---

## ***Chapter 3: Image processing. . . . . 41***

Image processing . . . . .	42
The MIL package. . . . .	42
Steps to performing a typical application . . . . .	43
A typical application. . . . .	43

---

## ***Chapter 4: Improving your images . . . . . 47***

Image quality . . . . .	48
Techniques to improve images . . . . .	49
Averaging an input sequence . . . . .	50
Applying spatial filters . . . . .	51
Opening and closing . . . . .	52
Basic geometrical transform . . . . .	53

---

**Chapter 5: Image manipulation . . . . .55**

Image manipulation. . . . .	56
Image statistics. . . . .	56
Generating a histogram . . . . .	56
Finding the image extremes . . . . .	58
Projecting an image to one dimension . . . . .	59
Thresholding your images . . . . .	60
Binarizing. . . . .	60
Clipping. . . . .	62
Histogram equalization . . . . .	62
Accentuating edges . . . . .	63
Edge enhancers. . . . .	64
Edge detection . . . . .	65
Arithmetic with images . . . . .	67
Combining images . . . . .	67
Mapping an image . . . . .	69
Erosion and dilation . . . . .	69
Distance transform . . . . .	73
Labeling. . . . .	75

---

**Chapter 6: Advanced image processing . . . . .77**

Advanced image processing . . . . .	78
Custom spatial filters. . . . .	78
Custom morphological operations. . . . .	82
Erosion and dilation. . . . .	83
Thinning and thickening . . . . .	88
Matching . . . . .	89
Searching for hits or misses. . . . .	90

Connectivity mapping . . . . .	90
Fast Fourier Transform . . . . .	91
Watershed transformations . . . . .	96
Using watersheds to separate touching objects. . . . .	96
Using watersheds to separate objects from their background . . . . .	98
Minimum variation between extrema . . . . .	98
Using marker images. . . . .	99
Style of the watershed lines . . . . .	100
Skipping the last level . . . . .	101
Polar-to-rectangular and rectangular-to-polar transform . . . . .	102
Warping . . . . .	104
First-order polynomial warpings . . . . .	105
Using LUTs to perform a warping. . . . .	105
Interpolation modes . . . . .	107
Points outside the source buffer. . . . .	108
Discrete Cosine Transform . . . . .	108

---

## **Chapter 7: Calibration. . . . . 111**

Introduction. . . . .	112
Types of distortions. . . . .	113
Steps to getting results in real-world units . . . . .	113
Transforming coordinates or results. . . . .	114
Physically correcting an image. . . . .	114
Automatically getting results in real-world units . . . . .	115
Calibrating your imaging setup . . . . .	117
Real-world grid. . . . .	117
List of coordinates. . . . .	119
Calibration modes. . . . .	120

Coordinate systems and camera position . . . . .	120
Multiple fields of view . . . . .	123
Single camera fixed on a manipulator: Relative camera position example . . .	123
Single camera and moveable object: Relative coordinate system example . . .	124
Several cameras and fixed object: Relative coordinate system example. . . . .	125
Processing calibrated images. . . . .	125

---

## ***Chapter 8: Blob analysis . . . . .127***

Blob analysis . . . . .	128
MIL and blob analysis . . . . .	129
Steps to performing blob analysis. . . . .	129
A simple blob analysis example . . . . .	132
Blob reconstruction . . . . .	135

---

## ***Chapter 9: Setting up for blob analysis . . . . .137***

Identifying blobs . . . . .	138
Adjusting blob analysis processing controls . . . . .	139
Controlling the image lattice . . . . .	140
The pixel aspect ratio . . . . .	140
Setting the Blob identification mode . . . . .	142
Selecting blobs. . . . .	143

---

## ***Chapter 10: Analyzing the blobs. . . . .145***

Making feature extractions. . . . .	146
The area and perimeter . . . . .	148
Dimensions . . . . .	150
Determining the shape. . . . .	152

Finding the blob location . . . . .	156
Moments . . . . .	158
Location, length and number of runs . . . . .	159

---

## **Chapter 11: Pattern matching . . . . . 161**

Pattern matching . . . . .	162
Simple alignment techniques . . . . .	163
Vertical and horizontal alignment . . . . .	163
Angular alignment . . . . .	166

---

## **Chapter 12: Models, searches, and search parameters . . . . . 171**

Performing a search with a user-defined model. . . . .	172
Rotation. . . . .	178
Setting the angle of search. . . . .	180
Determining the rotation tolerance of a model . . . . .	180
Masking the model. . . . .	181
Search parameters. . . . .	183
Specifying the number of matches. . . . .	184
Setting the acceptance level. . . . .	184
Setting the certainty level . . . . .	184
Redefining the model's reference position . . . . .	185
Selecting the search region . . . . .	186
Positional accuracy . . . . .	187
Setting the speed parameter . . . . .	187
Preprocess the search model . . . . .	188
Speeding up the search . . . . .	188
Choose the appropriate model . . . . .	189
Adjust the search speed parameter. . . . .	189

Effectively choose the search region and search angle . . . . .	190
Searching for multiple models at the same time . . . . .	190
The pattern matching algorithm (for advanced users) . . . . .	191
Normalized Correlation . . . . .	191
Hierarchical Search. . . . .	192
Search Heuristics . . . . .	194
Sub-pixel accuracy . . . . .	195

---

## **Chapter 13: Geometric Model Finder. . . . .197**

The MIL Geometric Model Finder module. . . . .	198
Steps to performing a search . . . . .	199
Basic concepts . . . . .	200
Defining and adding models to your model finder context . . . . .	201
Model indices and labels. . . . .	201
Drawing the model's active edges . . . . .	202
Extracting edges . . . . .	202
Guidelines for choosing models. . . . .	205
Masking your model . . . . .	208
Determining what is a match. . . . .	209
Interpreting results . . . . .	212
Customizing search settings . . . . .	213
Advanced search settings . . . . .	220
Shared edges. . . . .	226
Global context settings. . . . .	227
Retrieving and analyzing results . . . . .	228
Annotating results . . . . .	228
Calibration . . . . .	229
An example . . . . .	232

---

**Chapter 14: Optical character recognition . . . . . 241**

The MIL OCR module . . . . .	242
Steps to reading or verifying a string in an image . . . . .	243
A typical application. . . . .	245
Using fonts. . . . .	248
Calibrating fonts . . . . .	249
Setting character constraints. . . . .	250
Setting processing controls . . . . .	251
Managing fonts . . . . .	253
Saving and restoring a font . . . . .	253
Inverting a font. . . . .	253
Inquiring about a font. . . . .	253
Visualizing a font . . . . .	253
Creating custom fonts . . . . .	255
Speeding up the read or verification operation . . . . .	259

---

**Chapter 15: 1D and 2D code types . . . . . 261**

Overview of codes . . . . .	262
Using MIL to read and write codes. . . . .	263
Supported code types and noteworthy characteristics . . . . .	265
Anatomy of 1D and 2D codes . . . . .	266
Controlling read and write operations in MIL . . . . .	268
Issues pertaining to both read and write operations. . . . .	268
Read operations - issues to consider . . . . .	269
Write operations - issues to consider . . . . .	272
Retrieving results . . . . .	274

---

**Chapter 16: Measurements. . . . .275**

The measurement module . . . . .	276
Markers . . . . .	277
A multiple marker . . . . .	277
Steps to finding and obtaining measurements of markers. . . . .	278
A measurement example . . . . .	281
Measurement box . . . . .	284
Search algorithm . . . . .	287
Marker characteristics . . . . .	288
Edge markers: fundamental characteristics . . . . .	289
Edge markers: advanced characteristics . . . . .	291
Stripe markers: fundamental characteristics . . . . .	296
Stripe markers: advanced characteristics . . . . .	298
Multiple marker characteristics . . . . .	300
Measurements between two markers . . . . .	301
A measurement example . . . . .	303

---

**Chapter 17: Specifying and managing your data buffers. . . . .309**

Data buffers . . . . .	310
Target system . . . . .	311
Specifying the dimensions of a data buffer . . . . .	311
Data type and depth . . . . .	312
Attribute . . . . .	313
Manipulating and controlling certain data buffer areas . . . . .	316
Child buffers . . . . .	316
Copying specific buffer areas . . . . .	317
Managing data buffers . . . . .	318
Controlling how color image buffers are stored . . . . .	319



RGB buffers . . . . .	320
Binary buffers . . . . .	322
YUV buffers . . . . .	322
YUV16 Packed . . . . .	323
YUV9 Planar . . . . .	324
YUV12 Planar . . . . .	325
YUV16 Planar . . . . .	325
YUV24 Planar . . . . .	326
Child YUV buffers . . . . .	326
Accessing a MIL buffer directly . . . . .	327
Mapping a data buffer to user-allocated memory . . . . .	328
Pixel conventions . . . . .	331
Using buffers with the Bayer color filter . . . . .	332
Using MIL to convert the image . . . . .	333
How the Bayer image gets converted . . . . .	335
White balancing your Bayer images . . . . .	337

---

## **Chapter 18: Lookup tables . . . . . 339**

Lookup tables . . . . .	340
LUTs and data buffers . . . . .	341
Loading and generating data into LUTs . . . . .	341
Generating data directly into the LUT buffer . . . . .	341
Loading LUTs with precalculated data . . . . .	342
Using LUTs . . . . .	343
Processing using LUTs . . . . .	343
Displaying using LUTs . . . . .	343
LUTs and digitizers . . . . .	344

---

**Chapter 19: Displaying an image . . . . .345**

Displaying an image . . . . .	346
Types of displays . . . . .	347
Windowed display . . . . .	347
Auxiliary display. . . . .	348
Display number . . . . .	350
Display size and depth . . . . .	350
Displaying buffers of different data depths . . . . .	351
Removing a buffer from the display . . . . .	353
Displaying multiple buffers . . . . .	354
Panning, scrolling, and zooming . . . . .	357
Annotating the displayed image non-destructively . . . . .	358
Using GDI annotations . . . . .	360
Displaying an image in a user-defined window . . . . .	362
Using MdispSelectWindow() . . . . .	362
Palettes and output LUTs for windowed display (256-color) . . . . .	366
Reference material: Windows palettes and physical output LUTs . . . . .	366
Default palette settings . . . . .	368
Changing the default LUT values . . . . .	369
CPU-assisted display . . . . .	371

---

**Chapter 20: Generating graphics . . . . .373**

MIL and graphics . . . . .	374
Preparing for graphics . . . . .	374
Drawing graphics. . . . .	375
Writing text . . . . .	377

---

**Chapter 21: Grabbing with your digitizer . . . . . 379**

Cameras and input devices . . . . .	380
The data format . . . . .	380
The digitizer number . . . . .	381
Multiple cameras . . . . .	381
Grabbing a single field . . . . .	382
Line-scan cameras. . . . .	382
Grabbing to the display . . . . .	383
Live and pseudo-live continuous grabs . . . . .	383
Live transfer to the display . . . . .	384
Pseudo-live transfers to the display . . . . .	384
Screen Tearing . . . . .	387
Reference levels, lookup tables, and scaling. . . . .	388
Black and white reference levels . . . . .	388
Color image reference levels . . . . .	389
Mapping grabbed data through a LUT . . . . .	390
Scaling . . . . .	390
Optimizing application performance when grabbing . . . . .	391
Grab mode . . . . .	391
Double buffering . . . . .	392
Multiple buffering. . . . .	394
Grabbing a sequence of frames in real-time. . . . .	395
Grabbing with triggers and exposures. . . . .	396
Asynchronous reset mode . . . . .	396
Triggers and exposures . . . . .	397
Software triggers . . . . .	400

Auto-focusing . . . . .	400
Search strategies . . . . .	401

---

**Chapter 22: Color . . . . .407**

Dealing with color . . . . .	408
Grabbing . . . . .	408
Displaying . . . . .	409
Processing . . . . .	410
Saving and loading color images . . . . .	413

---

**Chapter 23: JPEG and JPEG2000 compression . . . . .415**

Introduction . . . . .	416
General steps . . . . .	417
Controlling a JPEG compression . . . . .	419
JPEG lossless . . . . .	419
JPEG lossy . . . . .	420
Restart markers. . . . .	421
JPEG2000 . . . . .	422
Preparation of source image . . . . .	423
Discrete wavelet transform (DWT) . . . . .	424
Quantization . . . . .	426
Decomposition. . . . .	427
Arithmetic encoding. . . . .	428
Post-processing. . . . .	428
Improving results . . . . .	430
Working with tables. . . . .	431
Inquiring values in default tables . . . . .	431
Using your own table . . . . .	432

---

**Chapter 24: Data manipulation with multiple systems . . . . . 433**

Data manipulation with multiple systems. . . . . 434

---

**Chapter 25: Using MIL with multi-processing  
and under multi-thread systems . . . . . 435**

Multi-processing . . . . . 436

Multi-threading . . . . . 437

    MIL and multi-threading . . . . . 437

---

**Chapter 26: Using MIL with Native Mode Functions . . . . . 447**

Integrating native functions with MIL code . . . . . 448

    Portability . . . . . 448

    Signaling MIL about Native Mode use . . . . . 448

A native mode example . . . . . 449

---

**Chapter 27: Distribution and licensing . . . . . 453**

Distribution of MIL-based applications . . . . . 454

Redistributing MIL run-time DLL files and device drivers with your application. . . 454

    Redistributing directly from the MIL CD. . . . . 454

    Redistributing using your own setup program. . . . . 455

Normal redistribution using your custom CD . . . . . 455

Silent redistribution . . . . . 456

    Response file parameters . . . . . 456

    Debugging the response file . . . . . 459

    Important notes for Windows 98/Me users. . . . . 460

    Important notes for Windows NT/2000 users . . . . . 460

Uninstallation . . . . . 460

MIL and MIL-Lite licenses . . . . .	461
Development license . . . . .	461
Temporary license. . . . .	461
Run-time license. . . . .	462
Hardware license-key . . . . .	463
Software license-key . . . . .	464
Hiding the MIL license . . . . .	465
Gencode Utility . . . . .	467

---

## ***Index***

---

## ***Product Support***







**Chapter**

# 1

## **Getting started**

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

## 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/Me/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. A Bayer filter is also included in MIL, which allows you to grab images with cameras using Bayer filters, and then convert them into 3-band color or single-band monochrome images.

### Compression

MIL allows you to compress and decompress images. MIL can compress images using the JPEG and JPEG 2000 algorithms in both lossless and lossy.

### 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. MIL also has functions to draw analysis results.

**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**

MIL provides two modules with pattern recognition capabilities for flexible solutions to machine vision problems such as alignment, measurement, and inspection of objects.

The MIL geometric model finder module provides pattern matching using geometric features. It employs an algorithmic approach which finds models using edge-based geometric features instead of a pixel-to-pixel correlation. Its capabilities include:

- Finding models through a range of angles and scales.
- Finding of occluded models.
- Greater tolerance of adverse lighting conditions.
- Results that provide more detailed information about the nature of the occurrence.

The MIL pattern matching module provides normalized grayscale correlation pattern matching. Its 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.
- The orientation of a target image.

- 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**

---

### **The library**

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

The following system requirements should be respected to ensure that MIL operates properly.

- Computer with Pentium class x86 compatible processor or better.
- Windows 98, Windows Me, Windows NT 4.0, or Windows 2000.
- Minimum of 48 Mbytes RAM for Windows 98/Me/NT, 64 Mbytes RAM for Windows 2000. This does not include DMA or non-paged memory space needed for any of the systems.
- Minimum of 100 Mbytes free hard disk space for a development environment in MIL. Minimum of 25 Mbytes of free hard disk space for a run-time environment in MIL.
- Matrox Imaging frame grabber with a MIL driver for Microsoft Windows 98/Me/NT/2000 (optional).
- graphics controller can be on a Matrox Imaging frame grabber.

### **Supported compilers**

The MIL CD includes MIL libraries that support the Microsoft Visual C++ 6.0 (service pack 5) compiler under Windows NT 4.0 (service pack 6), Windows 98 SE, Windows Me, and Windows 2000. The CD also includes ActiveMIL ActiveX controls for Microsoft Visual Basic 6.0 (service pack 5) and Microsoft Visual C++ 6.0 (service pack 5) RAD tools. The service pack indicated in parentheses 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\DOC` (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 license-key (a two-sided, 25-pin connector) for development of applications. The key allows you to code, debug, and run your applications. MIL supports hardware license-keys for either the parallel or USB ports. To redistribute your MIL applications, see Chapter 27, *Distribution and Licensing*.

To install your MIL software:

1. Attach the development hardware license-key to the parallel or USB port of your computer. If another device is attached to the port, disconnect it, attach the development hardware license-key, and then attach the device's connector to the other end of the hardware license-key. Note, the device 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 development tool.
- Which port you are using for the development hardware license-key: the parallel or the USB port.
- The type of Matrox hardware installed in your computer (for example, Matrox Corona-II). Note that under Windows 98/Me/2000 the boards have to be installed before the Matrox frame grabber drivers are installed.
- Whether to install the MGA drivers. This will only be asked if you have a Matrox Imaging board with a display section or a Matrox graphics board, and the drivers to be installed are newer than the drivers already on your computer.
- The digitizer and display format to load into the default setup file, *milsetup.h*.
- Whether by default, displayed images should be displayed in a window on the Windows desktop or without a window on an auxiliary screen (a non-Windows desktop screen). If the answer is the latter, you will be asked for the *video configuration format (VCF) to use by default*. Auxiliary screens require either two graphics controllers or a DualHead graphics controller that integrates two CRT controllers.
- 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.



It is important to remember that only one copy of MIL can be present on a computer at a time. When installing MIL or MIL-Lite on a computer with a more recent or equally recent version of MIL or MIL-Lite, the set-up program will not install MIL. The application can use the version of MIL already installed on the computer.

Conversely, if the version of MIL or MIL-Lite on the computer is less recent than the application's required version, a decision must be made. Either the version of MIL or MIL-Lite already on the computer must be removed before installing the newer version, otherwise the current version cannot be installed on that computer.

After installation, read the *read.me* file in the `\MIL\DOC` 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.

If MIL is run without the hardware license-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 you purchase a license.

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 license is obtained.

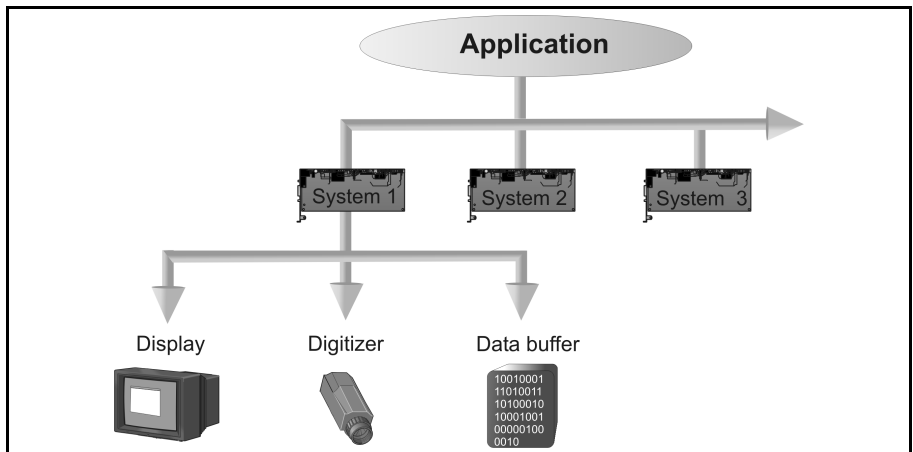
## Building an application

---

### 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 22.

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 (for windowed displays, 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.

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

MIL Libraries	
Library	Description
mil.lib	Core library.
milblob.lib	Blob Analysis module library.
milcal.lib	Calibration module library.
milocr.lib	Character Recognition module library.
milpat.lib	Pattern Matching module library.
milcode.lib	Code module library.
milim.lib	Image Processing module library.
milvga.lib	VGA library.
milmeas.lib	Measurement module library.
milmod.lib	Geometric Model Finder module library.

Board Libraries	
Library	Description
mil1394.lib	Matrox Meteor-II/1394 library.
milcor2.lib	Matrox Corona-II library.
milgen.lib	Matrox Genesis library.
milmet2.lib	Matrox Meteor-II/Standard/Multi-Channel library.
milmet2cl.lib	Matrox Meteor-II/Camera Link library.
milmet2D.lib	Matrox Meteor-II/Digital library.
milorion.lib	Matrox Orion library.

**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 Matrox 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\DOC (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.

## **The MIL user guide**

---

This user guide has been structured to allow you to start processing your images with as little information overload as possible. After reviewing installation and how to build an application, it is expected that you read chapter 2.

The next set of chapters, up to and including chapter 16, deal with enhancement, transformation and analysis operations. It is expected that you read only the chapters that you need; for chapters 4 - 6, you can read only the required sections.

The remaining chapters go into more detail about setting up and managing your images, displays and digitizers, as well as describing other topics that might be useful but not critical to getting started.

**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 22 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 should 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 19: 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 22: 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 21: Grabbing with your digitizer*.

### 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 21: Grabbing with your digitizer*. 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);

    /* 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 21: Grabbing with your digitizer*.

**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**

---

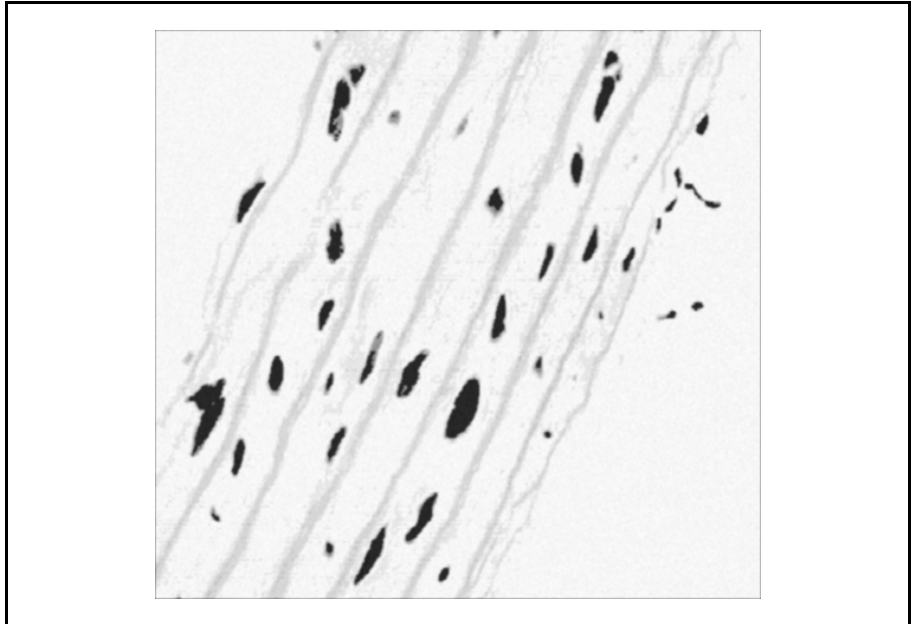
We have described the broad set of operations included in the MIL image processing module. Most applications do not require all of these operations.

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.

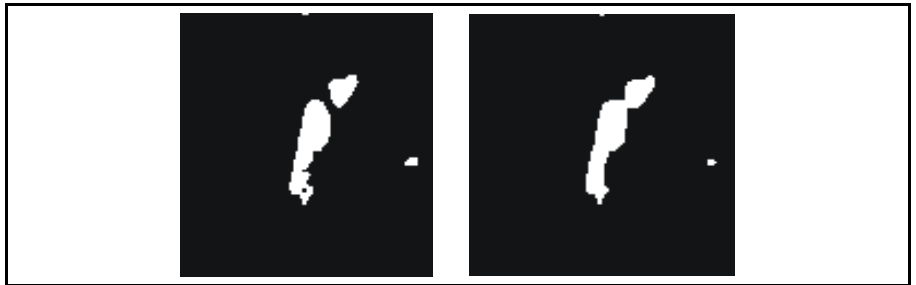
Opening and closing operations determine each pixel's value according to its geometric relationship with neighborhood pixels, and as such are part of a larger group of operations known as *morphological operations*.

### 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 <i>MimResize()</i> 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 <i>MimRotate()</i> .
Translating an image	<i>MimTranslate()</i> displaces an image by a specified number of pixels in the x and/or y direction, with sub-pixel accuracy.
Flipping an image	<i>MimFlip()</i> 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);

    /* 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();

    (cont...)

```

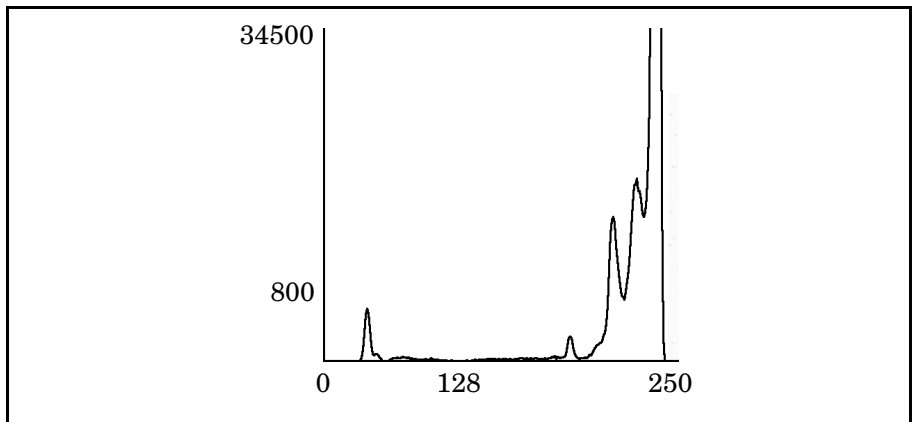
```

    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.



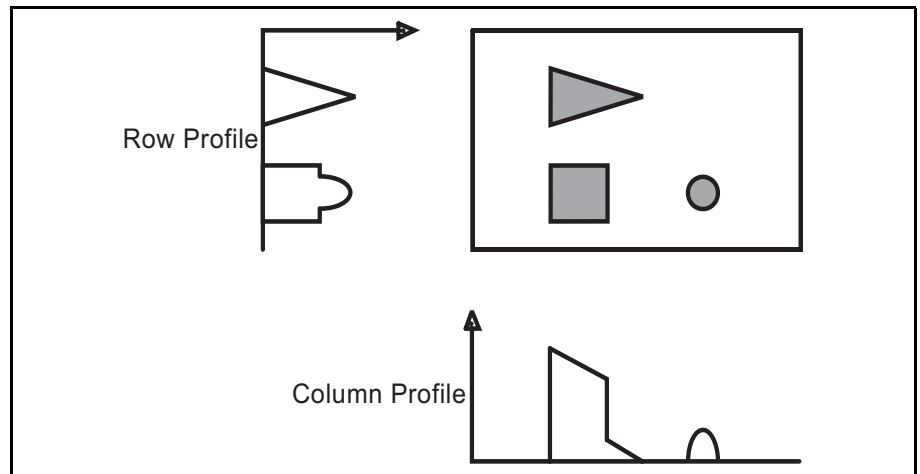
Another use of *MimFindExtreme()* is to find the number of objects in a labeled image. If all objects in an image are labeled with unique consecutive values, using *MimLabel()* (discussed later in this chapter), the largest label value also corresponds to the number of objects in your image.

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

---

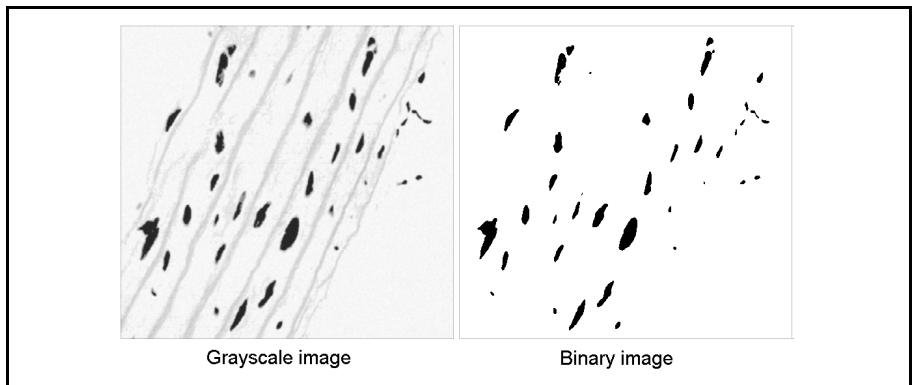
Thresholding images means reducing each pixel to a certain range of values. Some operations can be performed more efficiently on thresholded images. Images with full grayscale levels are useful for some tasks, but have redundant information for others. The MIL package provides two thresholding methods:

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

### Binarizing

A binarizing operation reduces an image to two grayscale values. In general, these values are 0 and the maximum value in the image (for example, 255 if the image is 8-bit). Except in the case of a float buffer in which case they are 0 and 1.

Binary images are useful when trying to identify geometric 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.



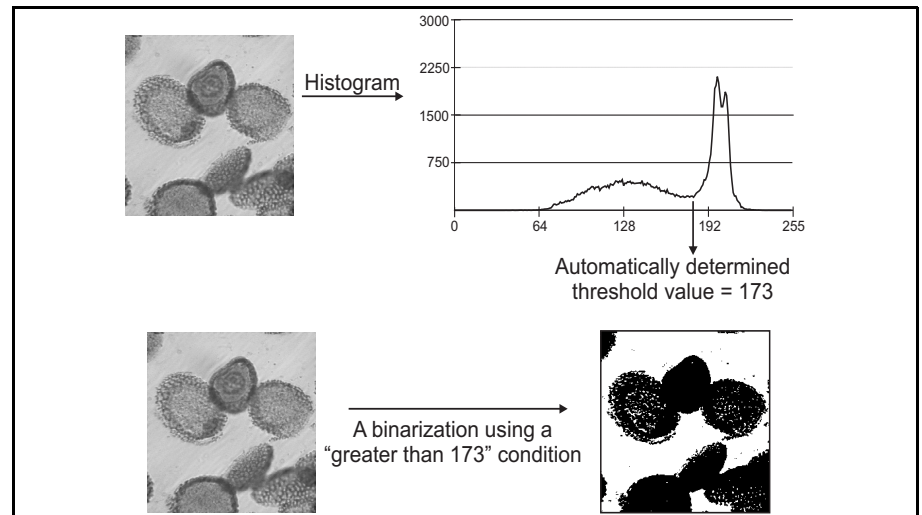
A binarizing operation is performed by comparing each pixel value in the image against one or two specified threshold values (for example, whether each pixel value is above one of the threshold values, or within the range of the two threshold

values). In general, pixels that meet the specified condition are set to the maximum value in the image while other pixels are set to 0. Except in the case of a float buffer, where pixels that meet the specified conditions are set to 1 while others 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. If the histogram contains only two peaks, the threshold value is set to the minimum value between these peaks. If the histogram contains more than two peaks, then the threshold value will typically be set between the two principle peaks, though exceptions exist for pure black and full saturation (0,255).



Clipping

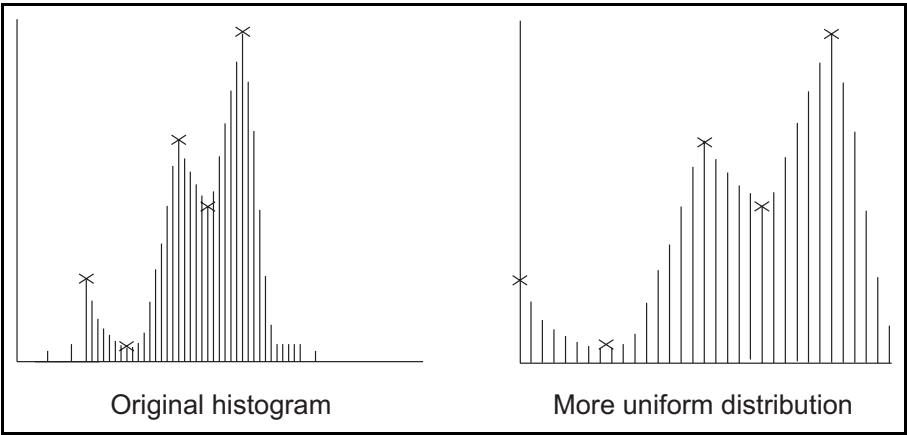
Clipping changes the image data less dramatically than binarizing. It changes the data to include only the range of pixel values in which you are interested. *MimClip()* takes a condition with at most two threshold points and replaces only those pixels that meet the condition with given values. Pixels 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()*.



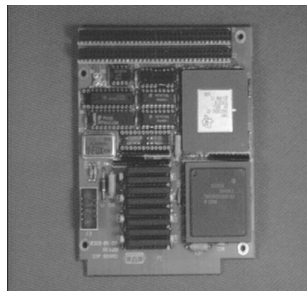
The *MimHistogramEqualize()* command first generates a histogram of the source image buffer. The histogram and a selected density function are then used to calculate a transformation LUT. If the destination buffer is an image, the transformation LUT is applied to the source buffer to produce the destination 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

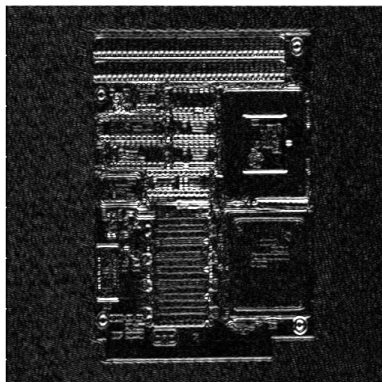
You can perform a multitude of edge detection operations, using *MimConvolve()*. This command offers predefined kernels for most common operations. Each offers some advantage over the others and should be chosen in function of your application.

- 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)

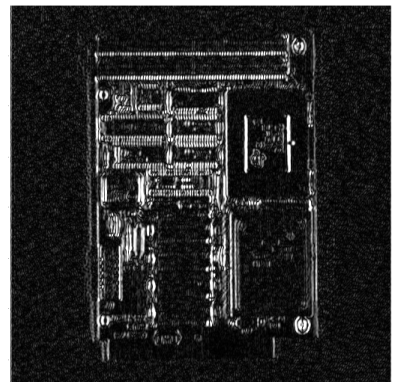
### Horizontal and vertical edge detection

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

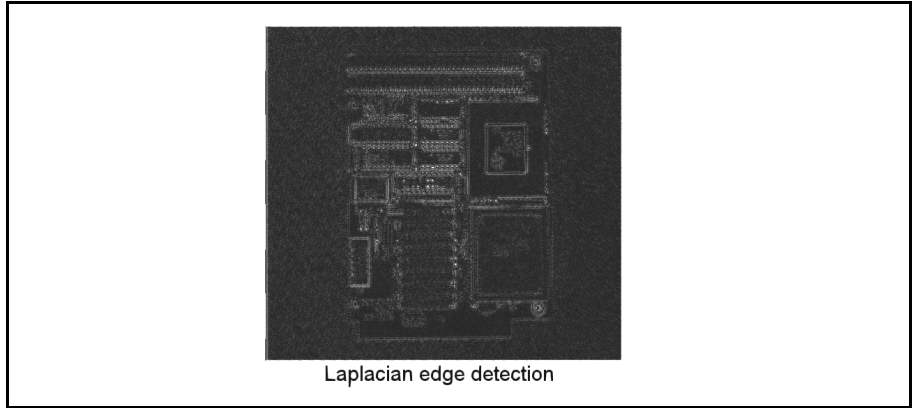


Vertical edge detection

**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.

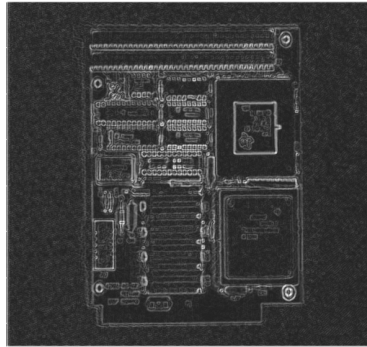


**Compass gradient edge detection**

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**

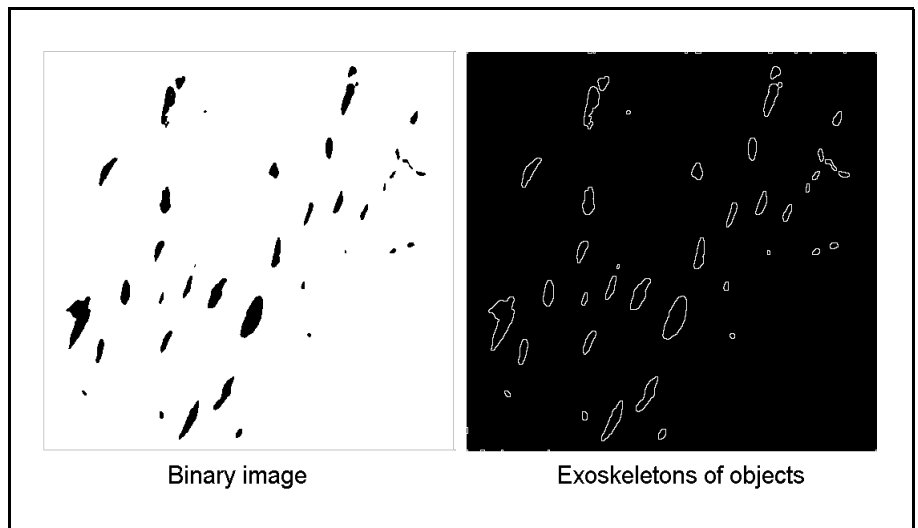
You can perform a basic dilation operation on 3 by 3 neighborhoods, using *MimDilate()*.

- 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()*.



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);
}

```

## Distance transform

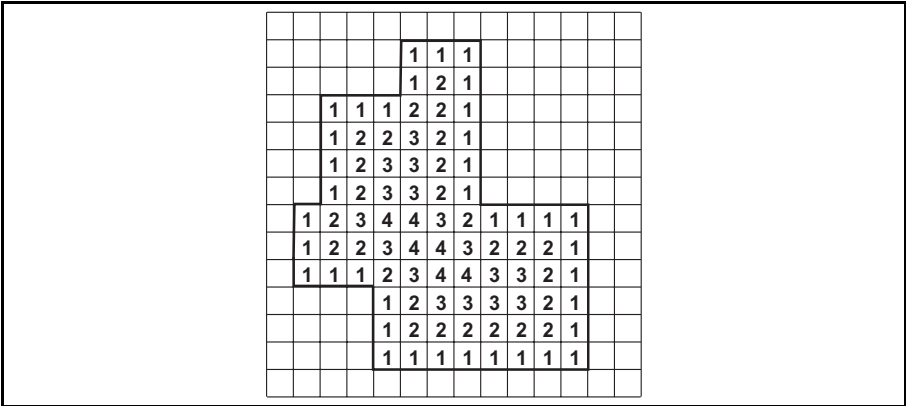
---

You can produce a distance transform using *MimDistance()*. This function determines the minimum distance from each foreground (non-zero) pixel to a background (zero) pixel, and assigns this distance to the foreground pixel. It produces a type of contour mapping of an image's foreground (object) pixels.

You can calculate the minimum distance using one of three transforms.

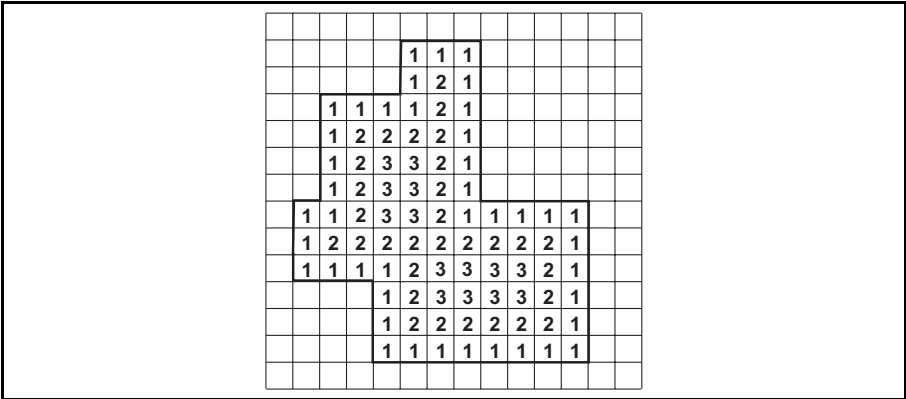
### The City Block transform

The City Block transform (*M\_CITY\_BLOCK*) determines the minimum distance using only horizontal or vertical steps. Each step counts as 1.



The Chessboard transform

The Chessboard transform (M\_CHESSBOARD) determines the minimum distance using horizontal, vertical, or diagonal steps. Each step counts as 1.



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.

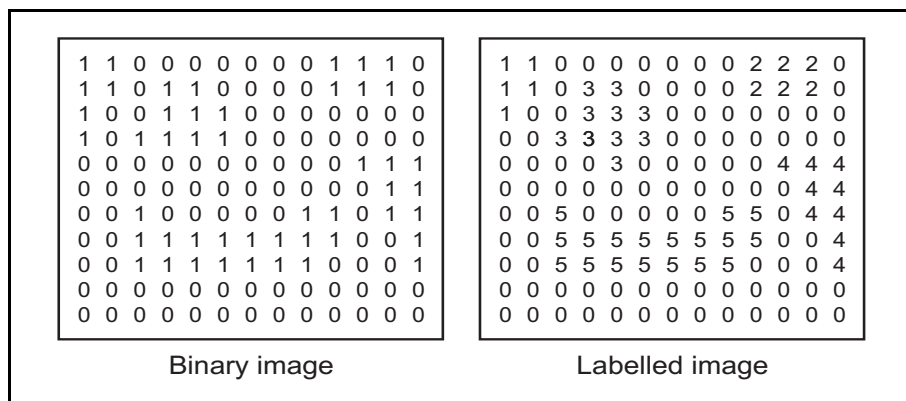


## Labeling

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):

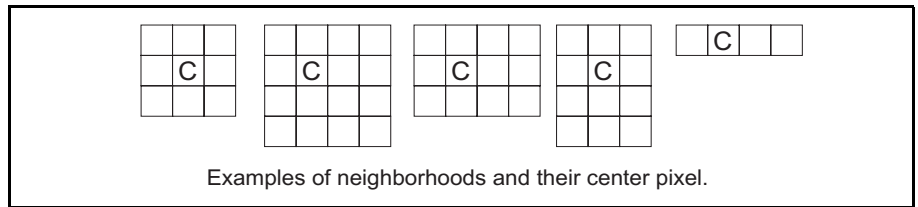
$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

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:

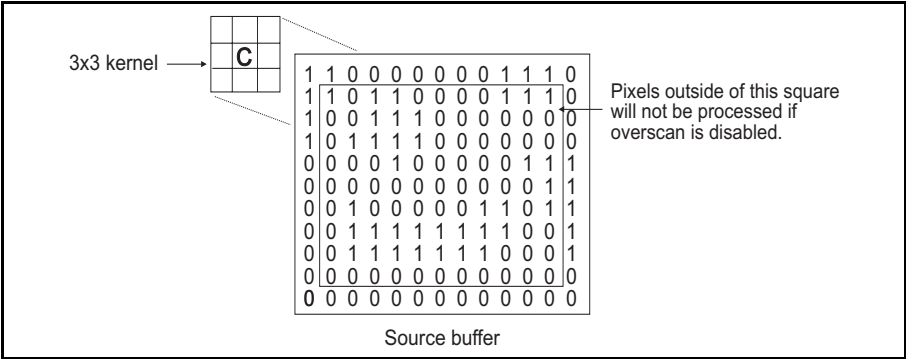
1. Allocate a kernel buffer (`M_KERNEL`), using *MbufAlloc2d()*. The dimensions of the kernel determine the size of the neighborhood that is used in the operation. The result of the operation is stored in the destination buffer at the location corresponding to the kernel's center pixel. When the kernel 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.



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:

- 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.
- 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 needed. 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.

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}
    };

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();

    (cont...)

```

```
/* 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
- 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

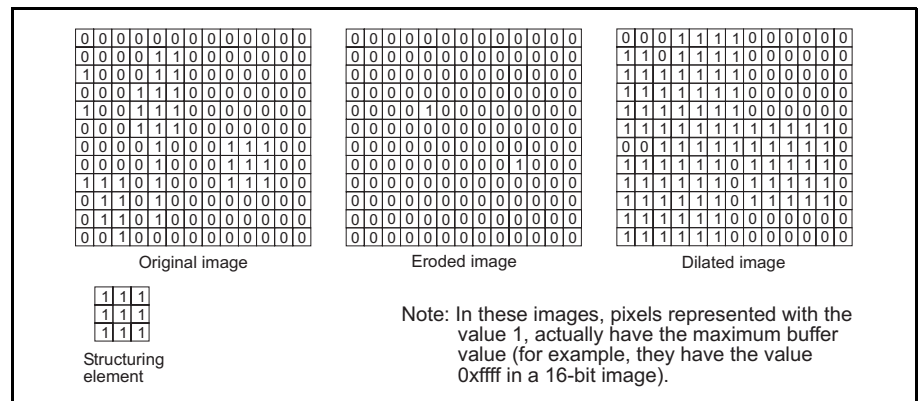
To define your own structuring element:

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.



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. */

    /* 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.
     */

```

(cont...)

```

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);

/* 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);

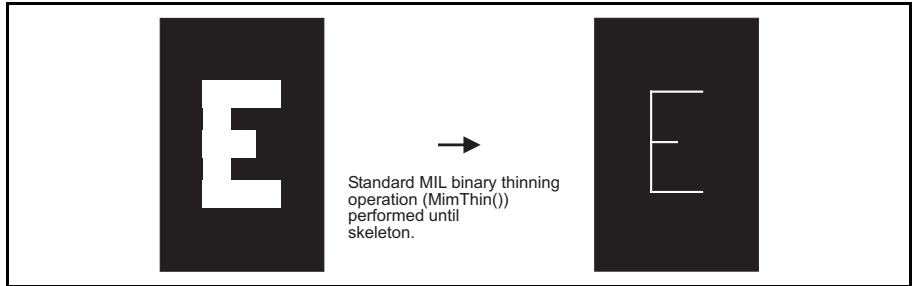
/* Free the allocated buffers. */
MbufFree(MilSubImage0);
MbufFree(MilSubImage1);
MbufFree(MilSubImage2);
MbufFree(MilSubImage3);
MbufFree(BinImage);

/* 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 ( $\text{M\_THICK}$ ):
  - 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  $\text{M\_BINARY}$ , 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). Except in the case of a float buffer, where, if an exact match is found, the result is 1. 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:

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

Result = LUTMAP (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:

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

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:

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

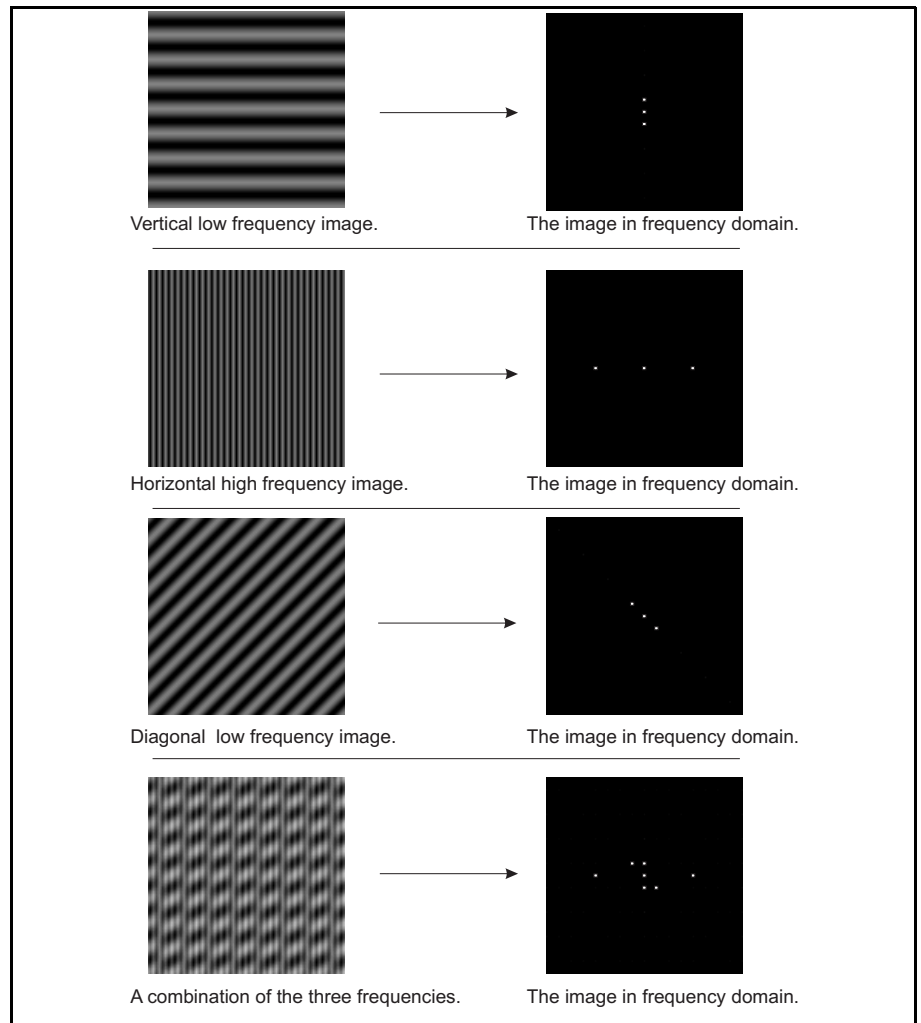
where x and y are the coordinates in the spatial domain and u and v are coordinates in the frequency domain.

**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.



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.

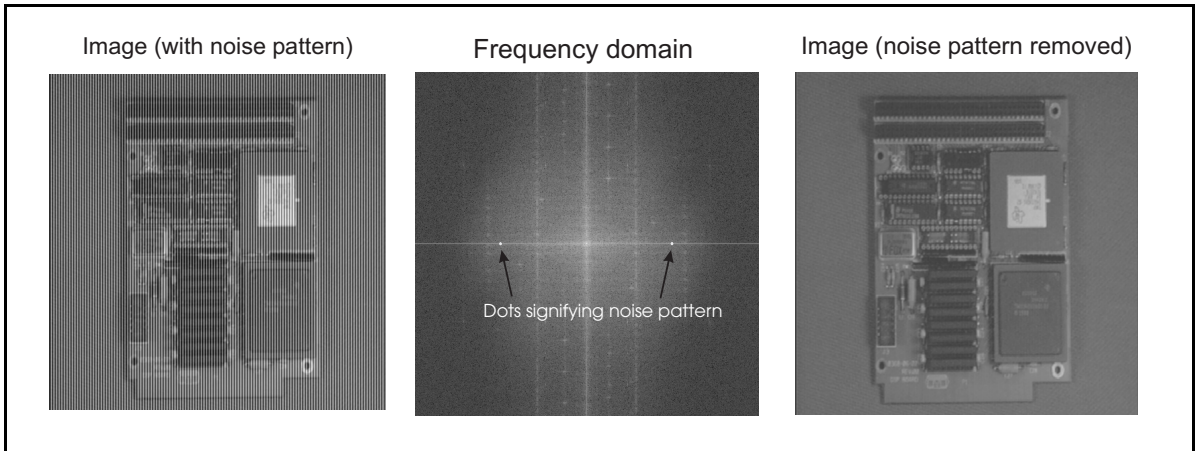
### Filtering an image

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.



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);
    .
    .
    .
}

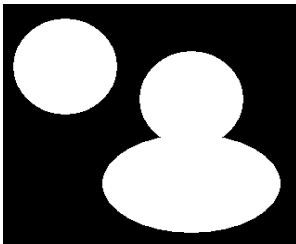
```

## Watershed transformations

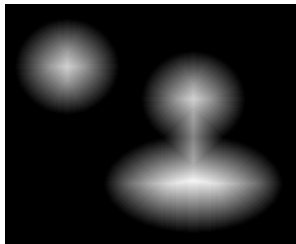
You can perform watershed transformations using *MimWatershed()*. A watershed transformation is generally used in conjunction with other processing operations to segment images, that is, to separate objects from their background and/or from each other.

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.

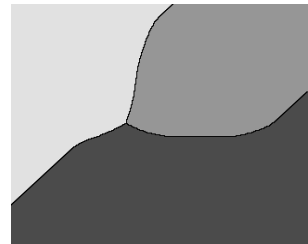
*MimWatershed()* labels an image's catchment basins and/or builds dividing lines between the catchment basins. These dividing lines are known as the *watershed lines* of the image. Note that catchment basins can be determined from the image's minima or its maxima.



An image with three objects.



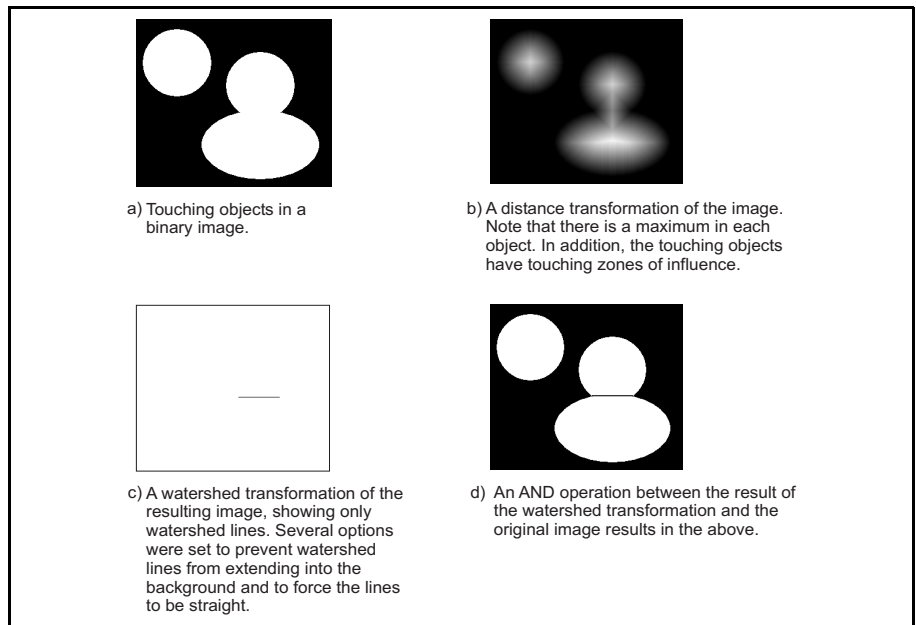
A distance transformation produces a maximum in each object.



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.

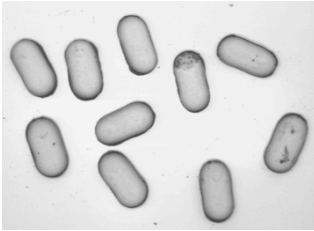


To summarize:

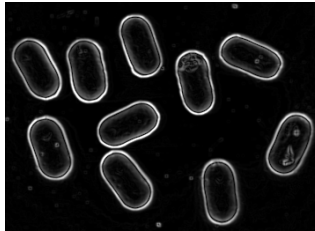
1. Perform a distance transformation on the image. This will result in a grayscale image with a maximum in each object.
2. Perform a watershed transformation on the resulting image. Note that:
  - Catchment basins must be determined from the image's maxima rather than its minima since *MimDistance()* 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.

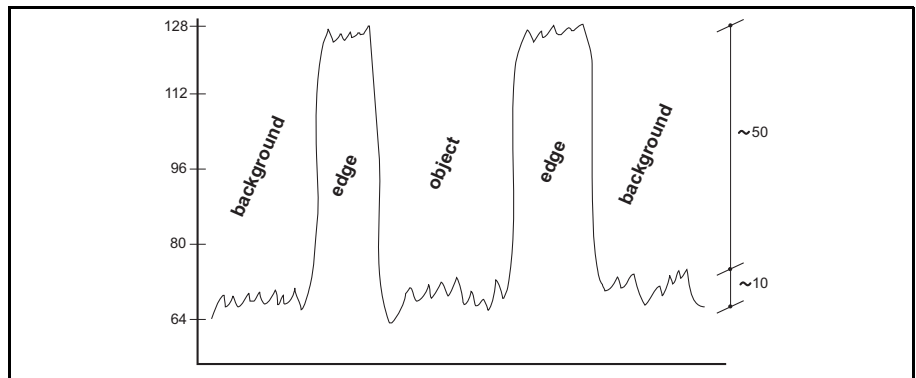
### Minimum variation between extrema

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.

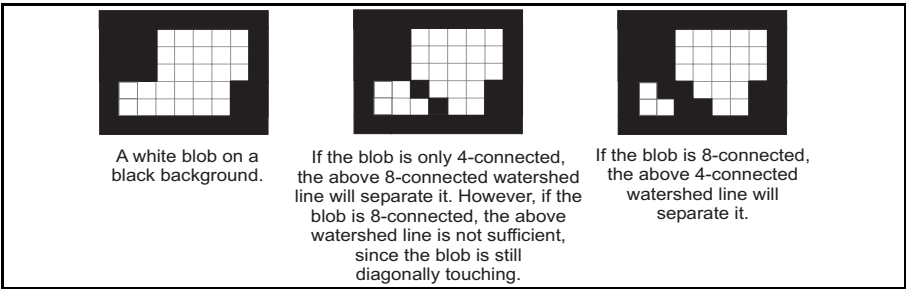
Note that catchment basins can be determined from markers in the marker image as well as from extrema in the source image. In this case, supply a marker image to *MimWatershed()* and also specify the minimum variation in gray-levels in the source image required to produce a new catchment basin.

**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.

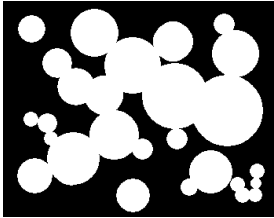


❖ MIL’s blob analysis module allows you to define blobs as either 4- or 8-connected.

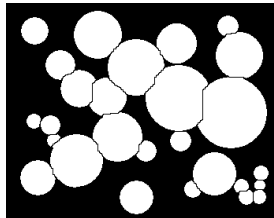
Note that 4-connected watershed lines can also separate 4-connected blobs but result in over-separation.

### Exact vs. straight

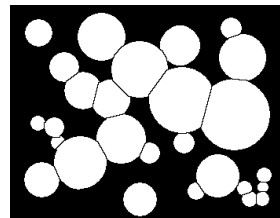
For visual purposes, watershed lines can be traced exactly or forced to be straight.



Original image.



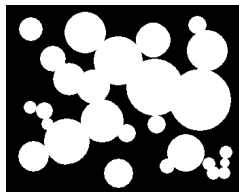
Exactly-traced watershed lines.



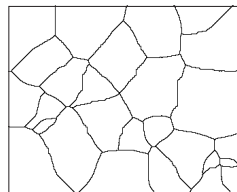
Straight watershed lines.

### Skipping the last level

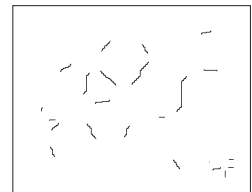
When you perform *MimWatershed()*, you can skip the last intensity level of the transformation (by setting the **ControlFlag** parameter to `M_SKIP_LAST_LEVEL`). In other words, you can prevent an extremum's zone of influence from extending beyond  $L_{\max} - 1$  (for a minimum) or  $L_{\min} - 1$  (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.



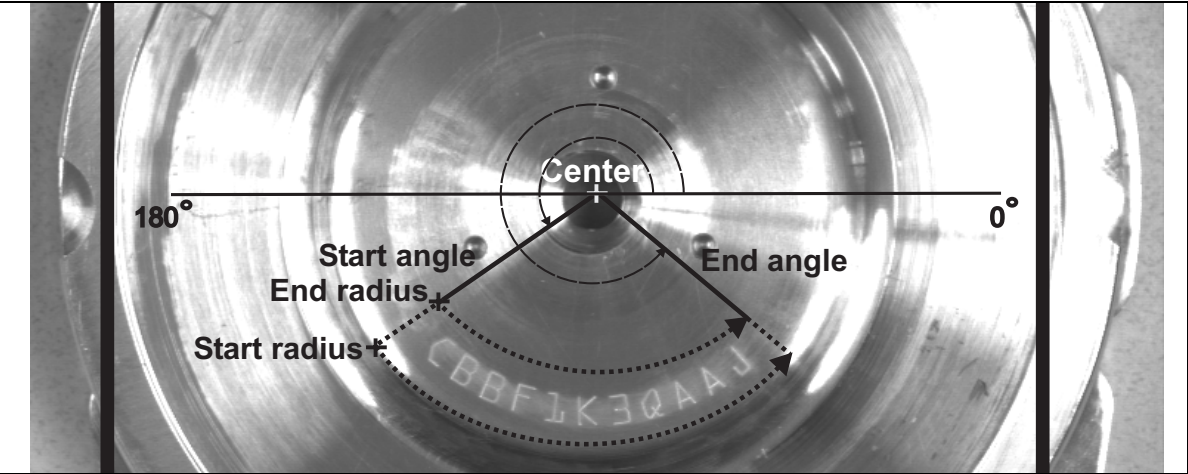
A watershed transform when the last level is skipped. The only watershed lines are those between touching objects.

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:

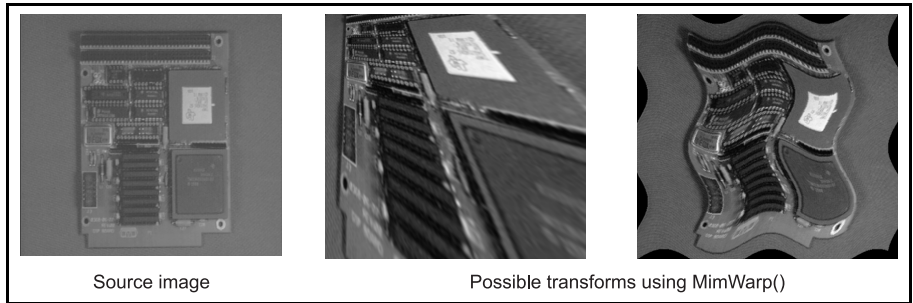
$$\Delta angle = \frac{(end_{angle} - start_{angle})}{arclength}$$

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.



*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

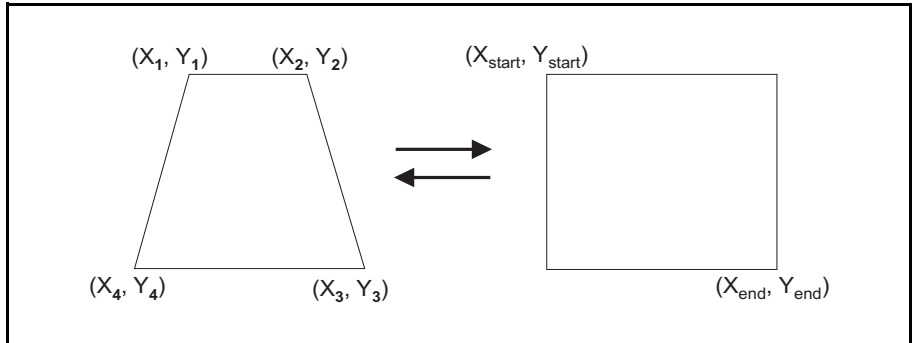
$$x_s = \frac{x}{w} = \frac{a_0 x_d + a_1 y_d + a_2}{c_0 x_d + c_1 y_d + c_2}$$

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

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  $3 \times 3$  coefficients. Alternatively, if you do not need to save the  $3 \times 3$  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 warpings

If  $c_0$  and  $c_1$  are set to 0 in the equation for a  $3 \times 3$  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  $(a_0 \dots a_2, b_0 \dots b_2, 0 \ 0 \ 1)$  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,  $K(u) = \frac{1}{\sqrt{2}}$  for u = 0 and

$K(u) = 1$  for u > 0

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

- *Mmod...()*

- *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()*. These mappings are stored with the calibration object.
3. Do one of the following:
  - To transform pixel coordinates or results to their real-world equivalents, use *McalTransformCoordinate()*, *McalTransformCoordinateList*, or *McalTransformResult()*.
  - To physically correct an image, use *McalTransformImage()*.

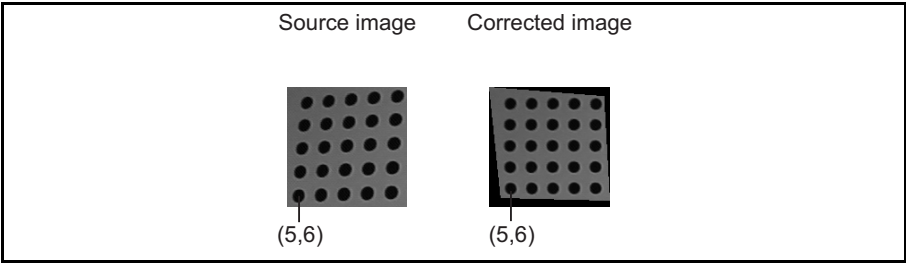
- To automatically get results from other MIL modules in real-world units, associate the calibration object to an image or digitizer, using *McalAssociate()*. The Geometric Model Finder module, however, uses *MmodControl()* with the `M_ASSOCIATED_CALIBRATION` control type to associate calibration objects with individual models.

**Transforming coordinates** You can use *McalTransformCoordinate()* to convert coordinates from their pixel to their world values (or vice-versa). You can also use *McalTransformCoordinateList()* to convert an entire list of coordinates at once.

**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** 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.



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.

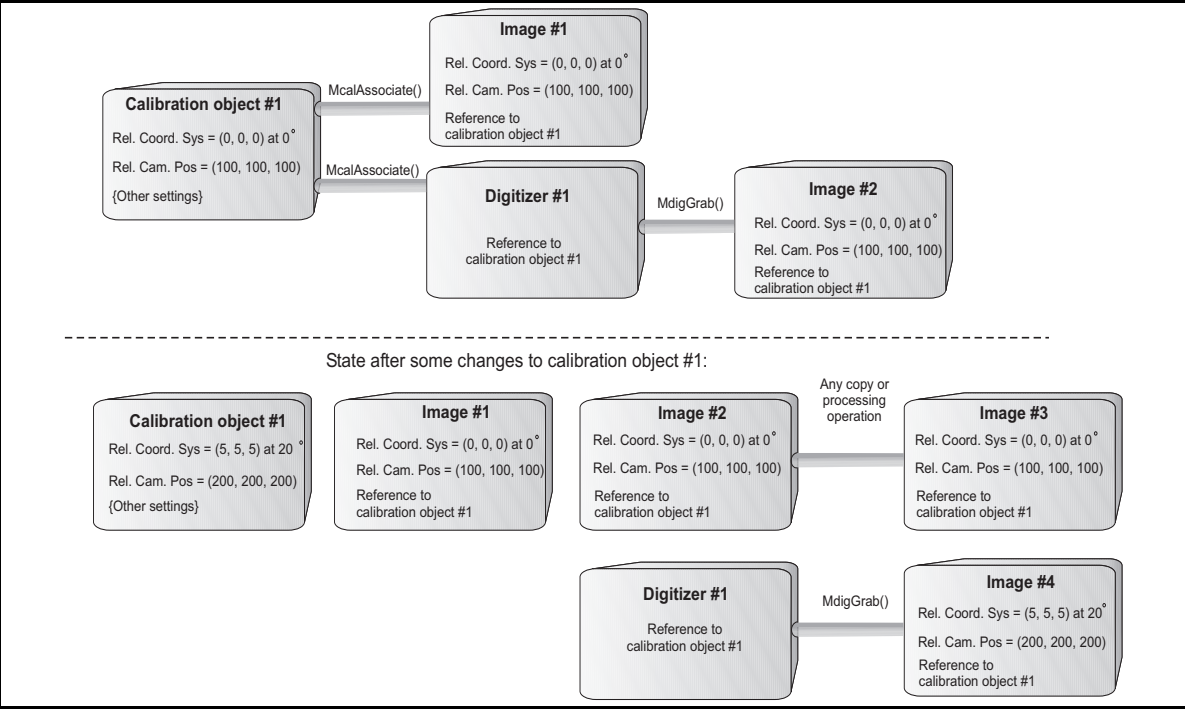
**Calibrated digitizer**

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.



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.

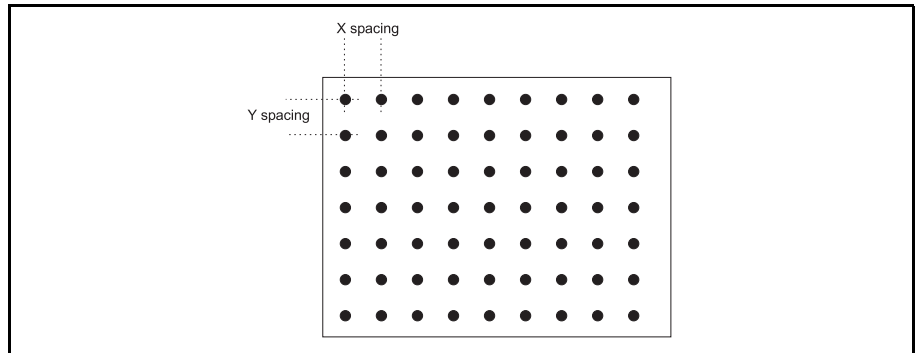
## Calibrating your imaging setup

---

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.

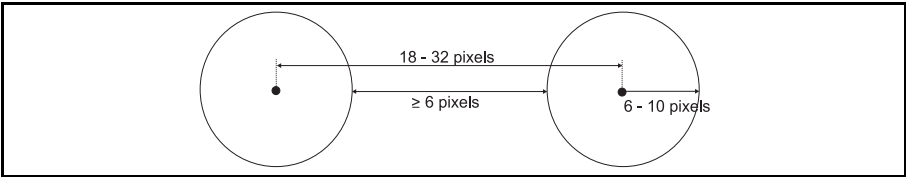


### General rules for constructing a grid

*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).

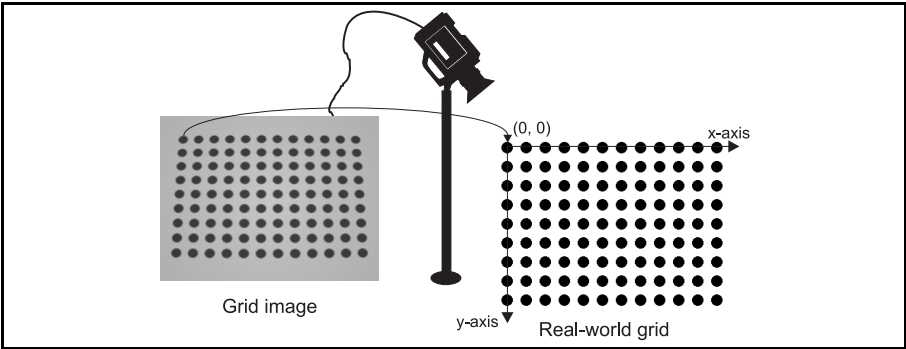
- The minimum distance between the edges of the circles should be 6 pixels.



- 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

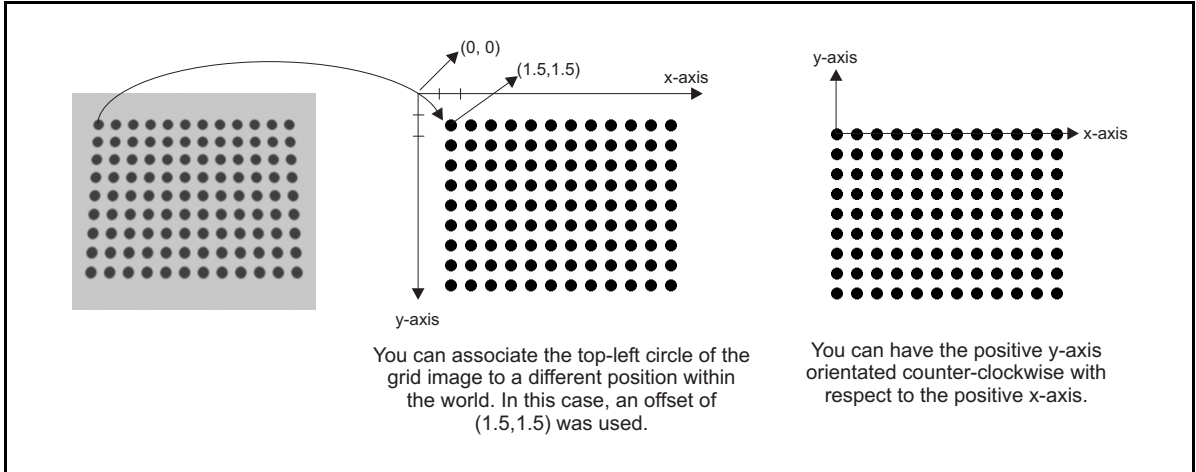
By default, the circle in the top-left corner of the grid image is associated to the origin,  $(0, 0)$ , of the *real world coordinate system*, the first column of circles is aligned with its Y-axis, and the first row of circles is aligned with its X-axis. This is because, in general, you know where 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  $90^\circ$  counter-clockwise with respect to the positive X-axis (by default, the calibration module assumes it is oriented  $90^\circ$  clockwise).



### 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

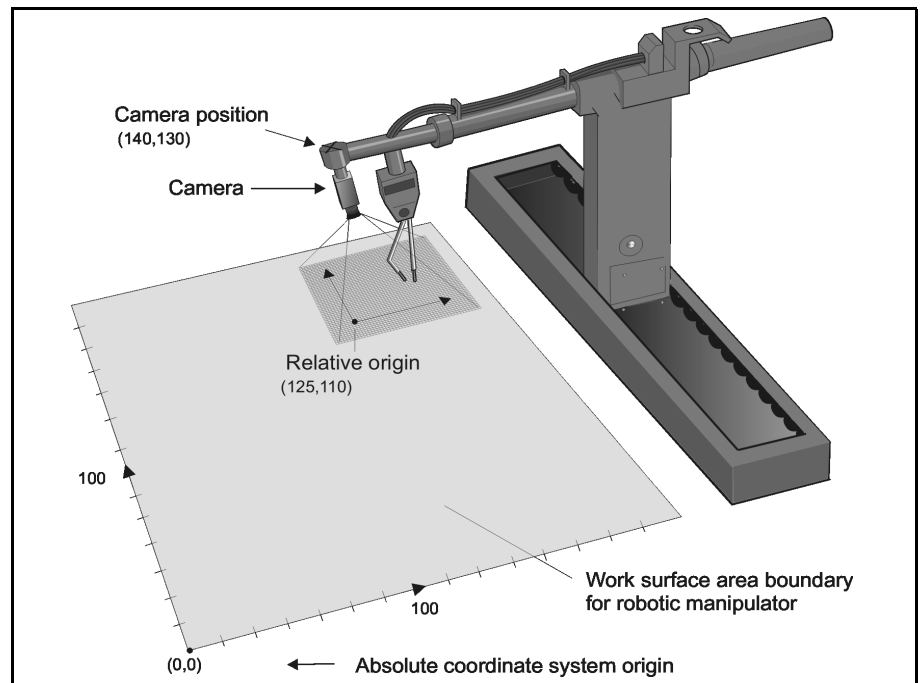
The absolute world coordinate system is the coordinate system used to locate and/or measure objects in the real world. It is implicitly defined from the calibration points when calibrating the imaging setup. Its unit of measure is user-defined (mm, cm, inches, etc.). Calibration relates the image coordinate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

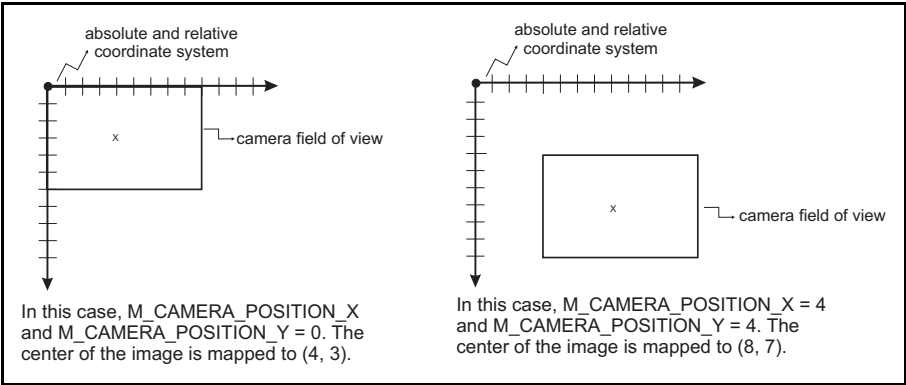
The camera position can be any arbitrary point that moves with the camera; it does not have to be the actual camera position. For example in the following diagram, the camera position is assigned not to the actual camera, but to the arm of the robotic manipulator.



### 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).

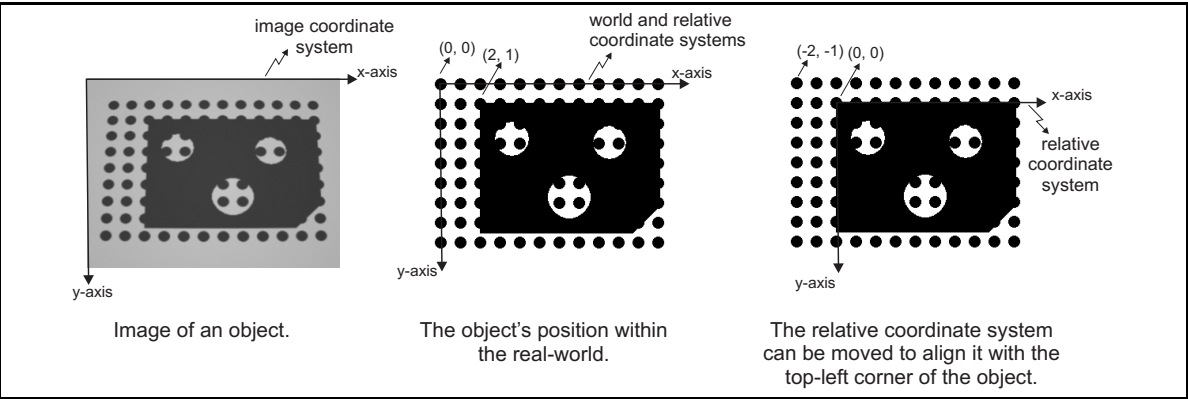
The relative camera position affects positional results taken from a calibrated image, as shown below.



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.

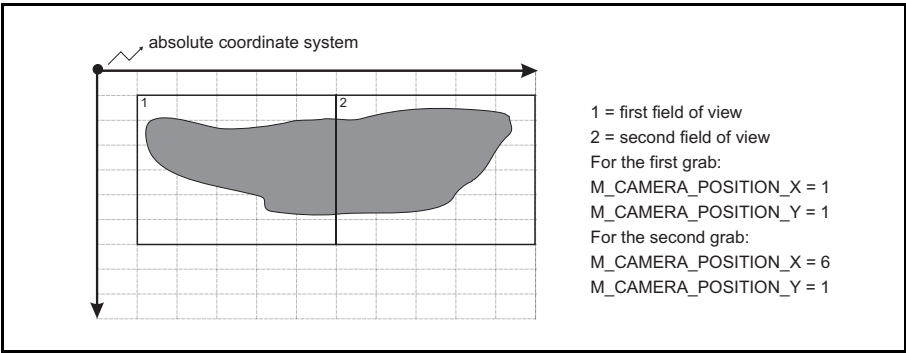
The approach to analyzing a large object involves acquiring images of the various parts of the object, getting real-world coordinates from each image, and then calculating the distance between the coordinates. Three types of applications are considered:

- 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

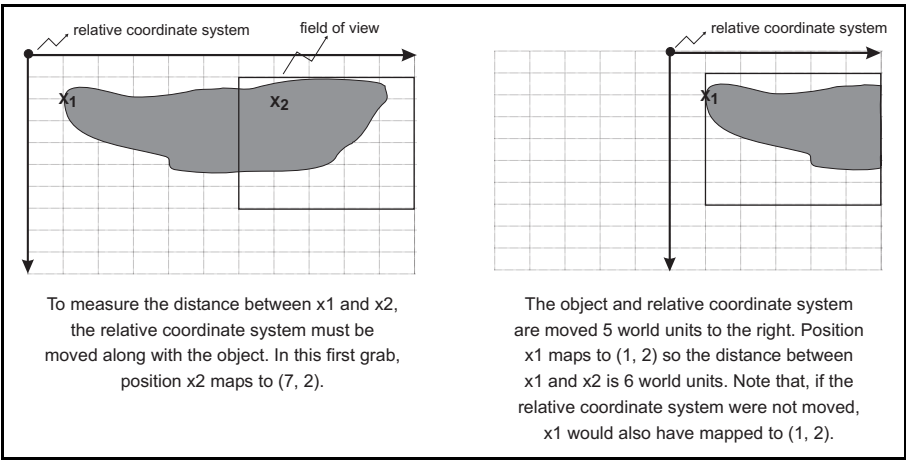
When a single camera is fixed on a manipulator, part of the object is grabbed, the camera is moved to a different position, a different part of the object is grabbed, and the process repeats until the entire object has been grabbed. For the coordinates from each image to be in the same absolute coordinate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



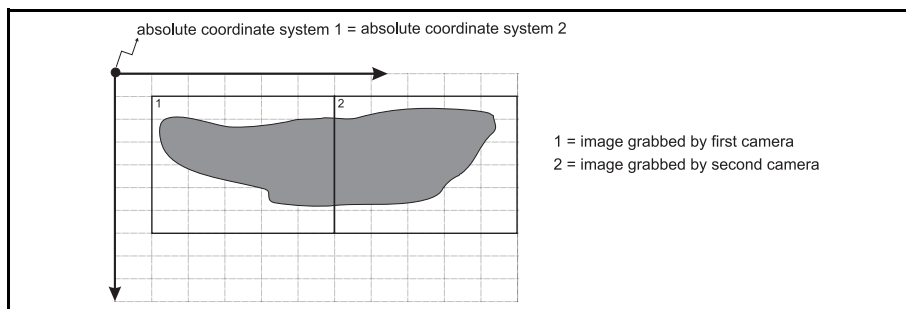
**Single camera and moveable object: Relative coordinate system example**

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()`.



### 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

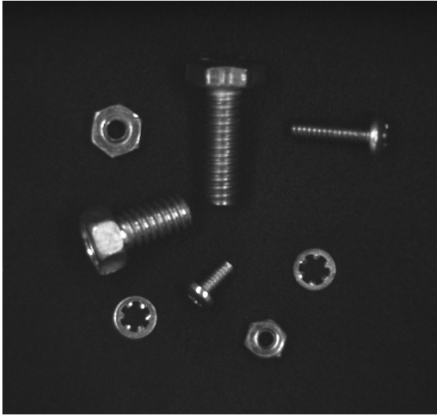
/* Size and color of the cross used to mark centers of gravity. */
#define CROSS_SIZE           20L
#define CROSS_COLOR          250L

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

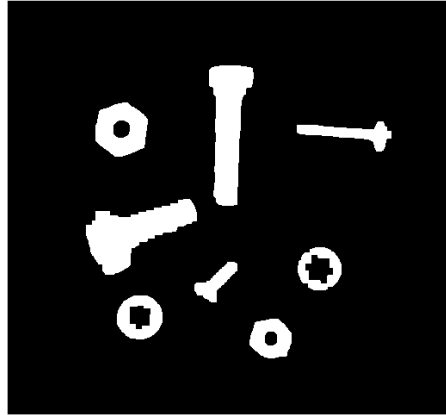
void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem,             /* System identifier. */
    MilDisplay,            /* Display identifier. */
    MilImage,              /* Image buffer identifier. */
    BinImage,              /* Binary image buffer identifier. */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

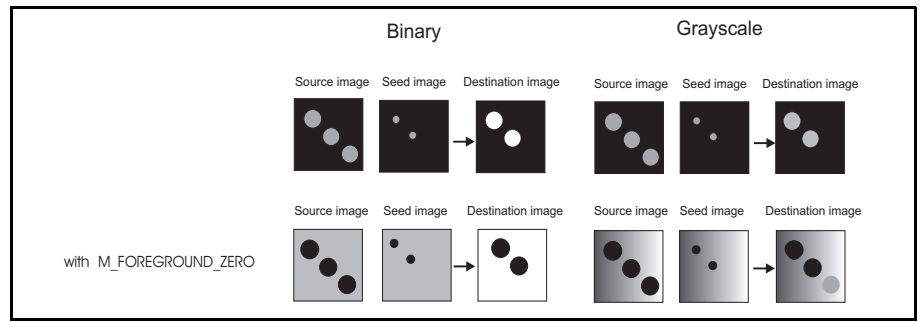
/* Free all allocations. */
MblobFree(BlobResult);
MblobFree(FeatureList);
MbufFree(BinImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

```

## Blob reconstruction

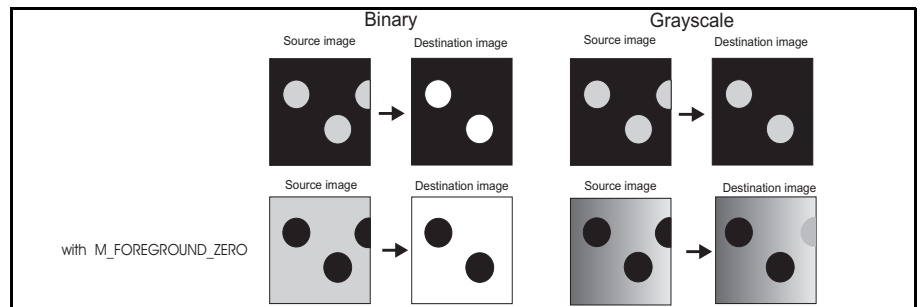
Although the blob analysis module is used mainly for blob feature calculation purposes, some of the *Mblob...()* commands can be used to perform blob image reconstruction. An example of this is the *MblobReconstruct()* command. It can:

- 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):



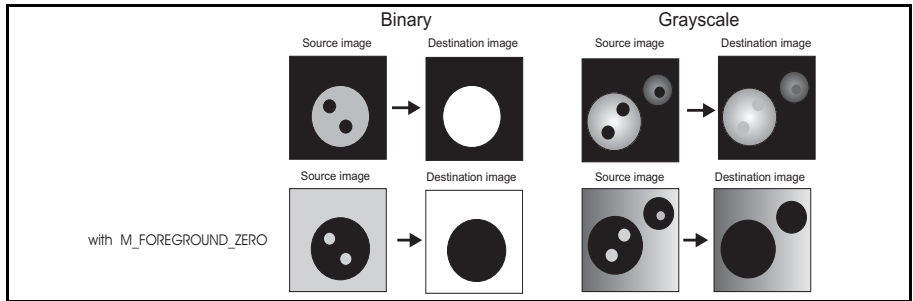
Note that, for images with grayscale backgrounds, blobs in the source image which are not seeded are filled with the average grayscale value of the background.

- Delete blobs that touch a border of the image:



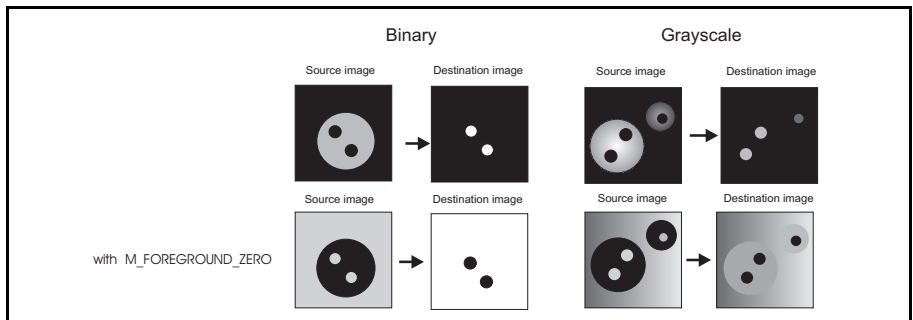
Note that, for images with grayscale backgrounds, the border blobs are filled with the average grayscale value of the background.

- 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

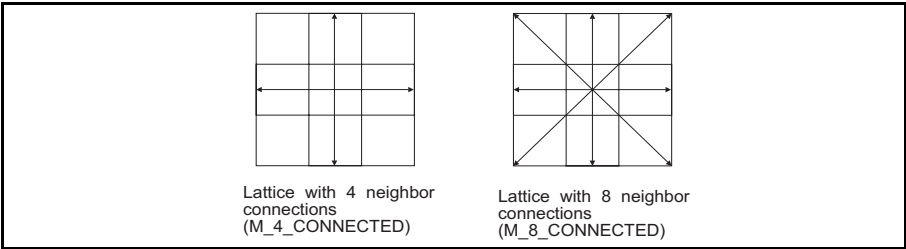
---

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:

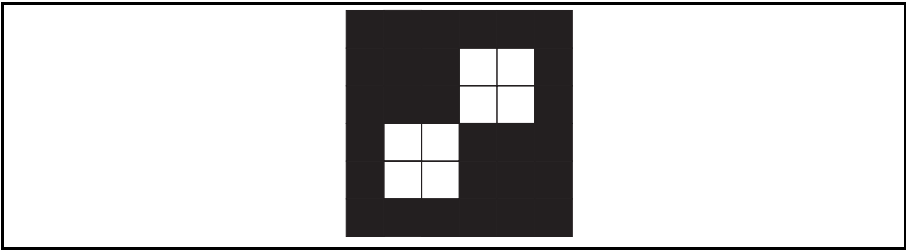
- Which pixel values are considered to be in the foreground (M\_FOREGROUND\_VALUE).
- Whether two pixels touching at their corners are considered part of the same blob, by appropriately defining the image lattice (M\_LATTICE).
- Whether non-zero pixels can have any value or must be set to the maximum 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.



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.



Pixel's relation to  
real distance

The pixel aspect ratio

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:

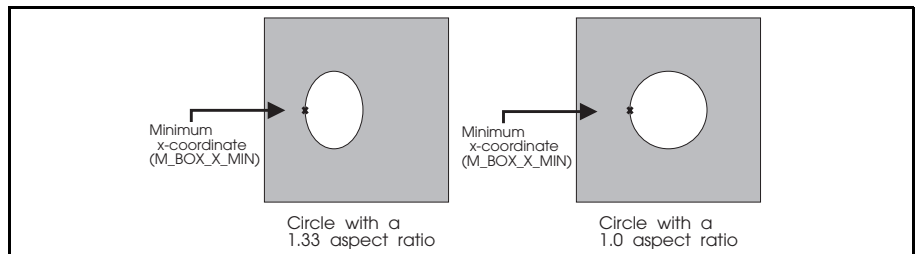
- You can adjust your image, using *MimResize()*, and then make the required blob analysis feature extractions.
- You can have calculations take the actual aspect ratio into consideration without modifying the image, by specifying the ratio, using *MblobControl()*. However, any feature derived from multiple 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**

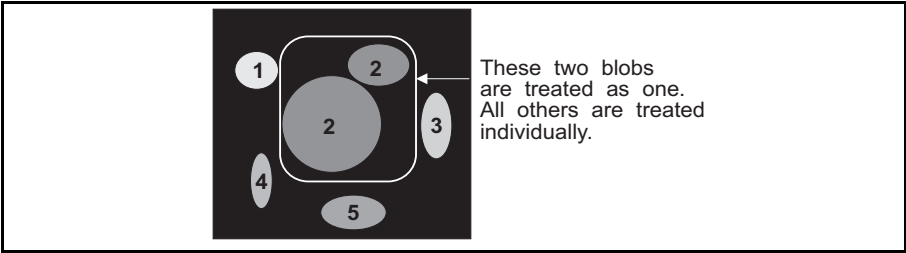
When using the blob analysis package, you usually want to make feature calculations on each blob. For example, if you want to find the area of each cell in a tissue image, set the blob identification mode to M\_INDIVIDUAL.

**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()*).

### Drawing results

You can also use *MblobDraw()* to draw specific features of the results in an image buffer. For example, you can draw the blobs contours, holes, position, minimum and maximum Feret diameters, among other features. You can use a previously allocated graphics context (see Ch. 19 *Generating graphics*) to control the drawing color, or use the default graphics context (M\_DEFAULT). You can draw directly into the image buffer, or annotate the image non-destructively by drawing into its display overlay buffer (see Ch. 18 *Annotating the displayed image non-destructively*).

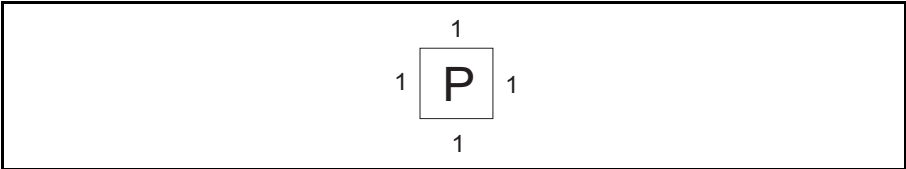
### Sorting results

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

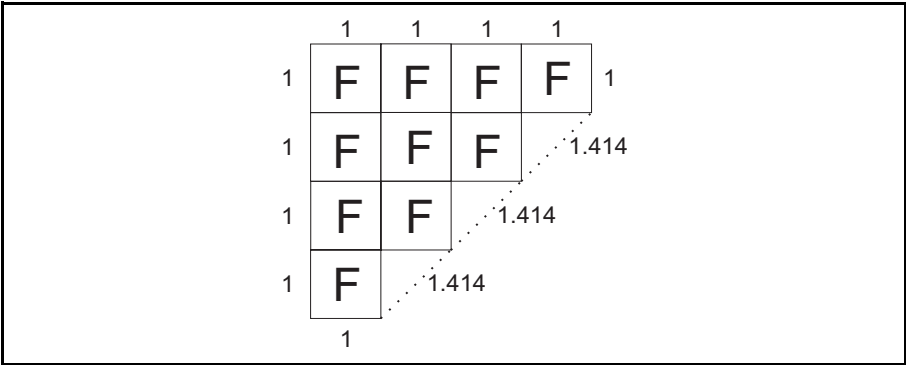
### The pixel

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:



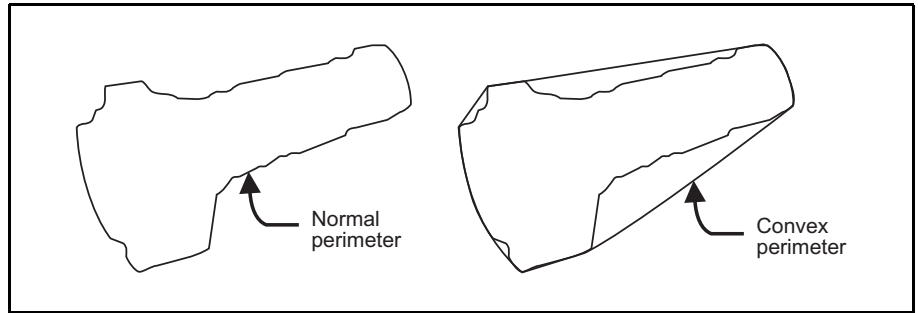
### The area and perimeter

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

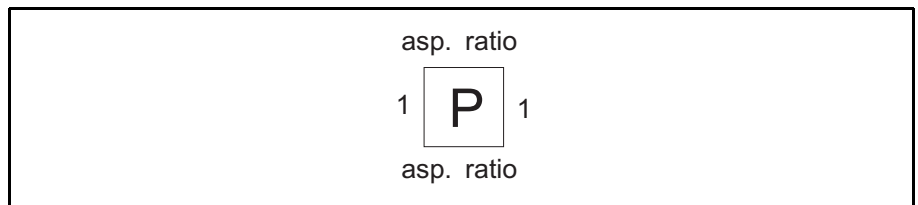
You can also calculate an approximation of the convex perimeter (M\_CONVEX\_PERIMETER) of the blobs. The convex perimeter is the perimeter of the convex hull (see below).



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:

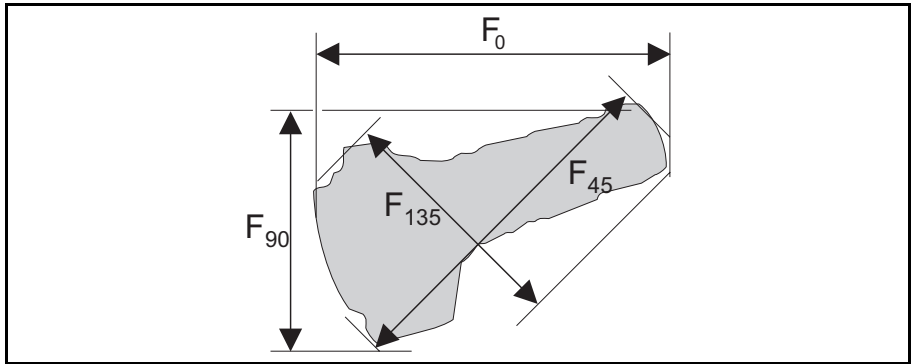


This affects all calculated features as if you had actually stretched the image (from the top-left corner in the x-direction only), by a factor equal to the pixel aspect ratio. We can no longer say that results are in "pixel" units. In fact, results are really in units of "pixel height" since the height 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.

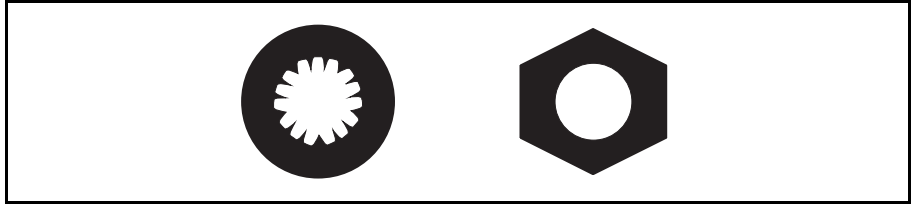
These features are derived from the area and perimeter, using the assumption that the blob area is equal to the [length x breadth] and the perimeter is equal to  $2(\text{length} + \text{breadth})$ . These relations only hold if the length and breadth are constant throughout a blob. However, long, thin 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.



```

/* 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\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.

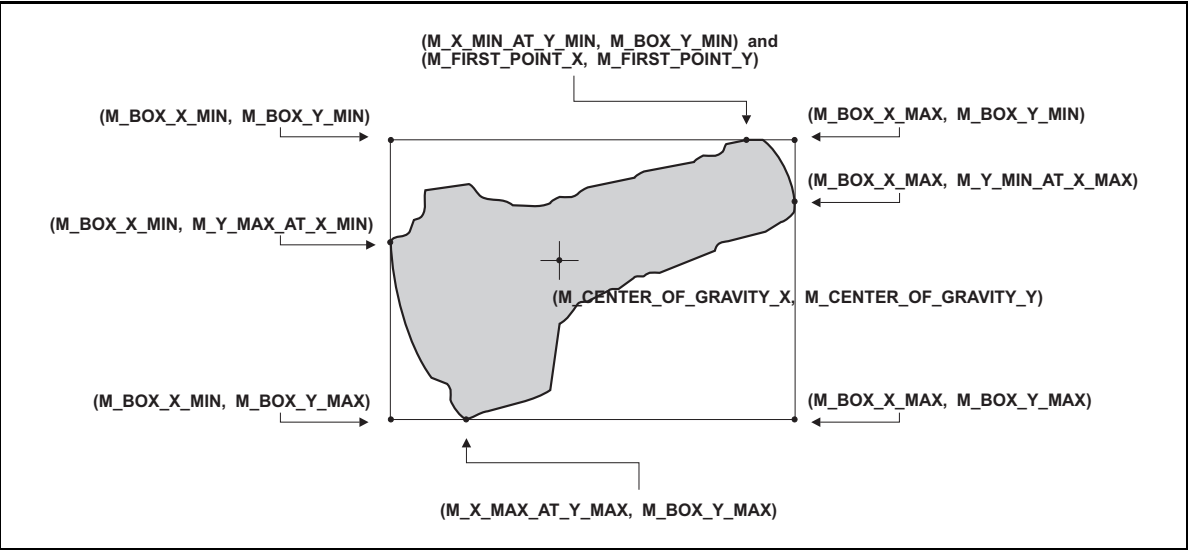
## Finding the blob location

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.

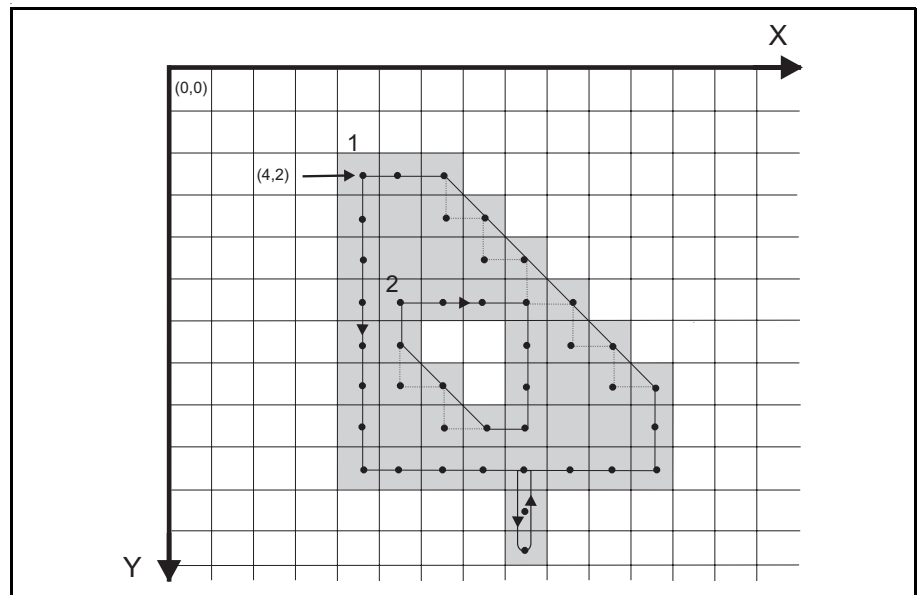
### Chained pixels

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 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**

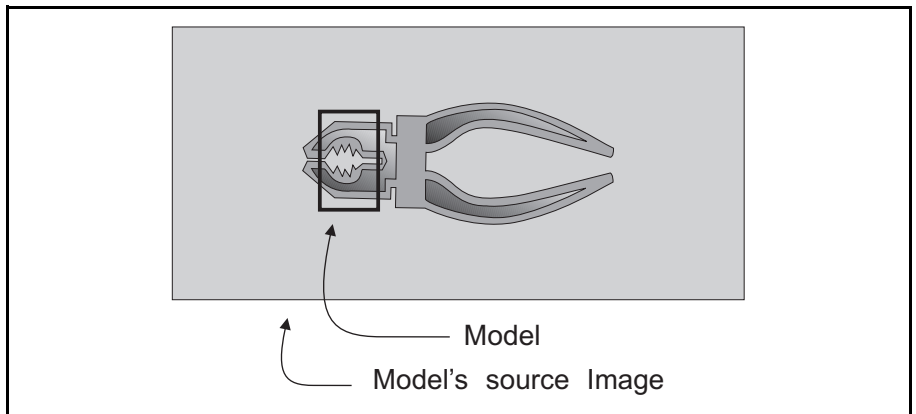
This chapter explains quick techniques to perform alignment operations, using the MIL pattern matching (recognition) module.

## 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.

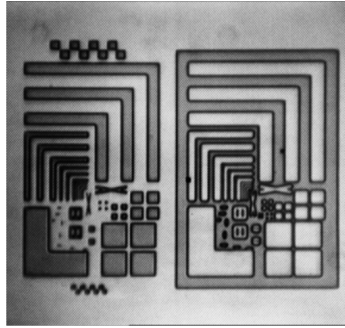
### Important

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`.

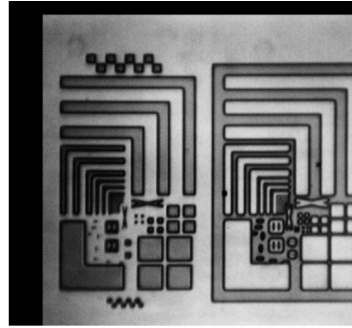
Note, once the model is defined, you can perform the search operation on an unlimited number of target images.

**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 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 Y.\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);
}

```

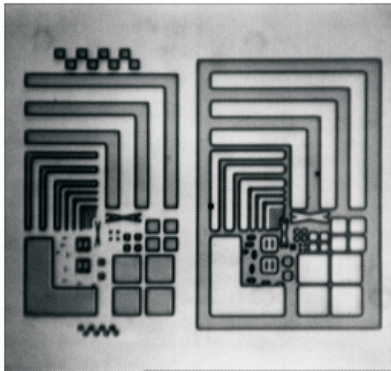
By executing *mshift.c*, you will find that *shfwafer.mim* is shifted by approximately 50 pixels horizontally and 20 pixels vertically.

### Angular alignment

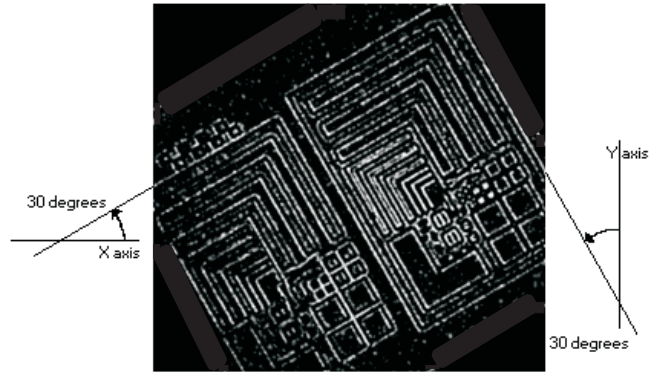
You can find the angular displacement of a target image, or of an object in that image, in a number of ways. The choice of method will depend on whether whole-image or object orientation is required, the shape and distinctiveness of the object, the complexity of the image background, and the degree of 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.

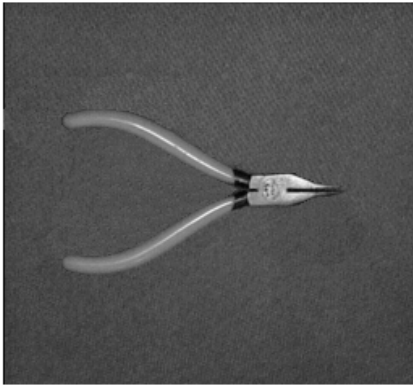
Whole-image orientation can be determined by following these steps:

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.

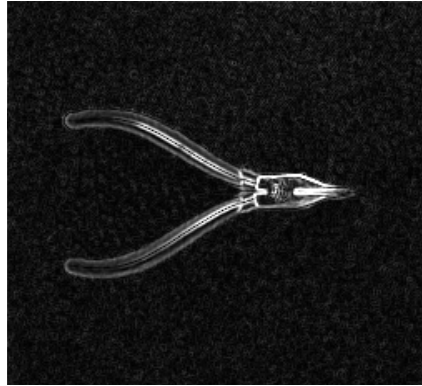
4. Call *MpatGetResult()* to get the angle of orientation, returned as a value in the specified result range.

### 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 with a user-defined model

---

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). 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.

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()* or *MpatDraw()*.

You can use *MpatDraw()* to draw the various model features. You can use a previously allocated graphics context (see Ch. 19 *Generating graphics*) to control the drawing color, or use the default graphics context (M\_DEFAULT). You can draw directly into the image buffer, or annotate the image non-destructively by drawing into its display overlay buffer (see Ch. 18 *Annotating the displayed image non-destructively*).

**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. To verify your mask, you can use *MpatDraw()* to draw the models' don't care pixels.

<b>Specify model parameters</b>	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 ( <i>MpatSetPosition()</i> ), restrict the number of occurrences to find ( <i>MpatSetNumber()</i> ), and set the level of acceptance ( <i>MpatSetAcceptance()</i> ).
<b>Preprocess the model</b>	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.
<b>Allocate a result buffer</b>	Before performing the search, you must allocate a result buffer, using <i>MpatAllocResult()</i> . This buffer is used to store the result values for subsequent search operations. You can delete the result buffer, using <i>MpatFree()</i> .
<b>Acquire the new target image</b>	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.
<b>Find the model</b>	<p>You can now search in the target image for the coordinates of model occurrences, using <i>MpatFindModel()</i>. The search is performed according to the defined model parameters.</p> <p>You can also search for several models of the same size and search region in the same image, using <i>MpatFindMultipleModel()</i>. 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.</p>
<b>Read the search results</b>	<p>To read results, use <i>MpatGetNumber()</i> and <i>MpatGetResult()</i> to get, respectively, the number of model occurrences found in the target, and the required results. You can use <i>MpatDraw()</i> to draw the occurrences' bounding box and/or a cross at the occurrences's reference position.</p> <p>The following sample program (<i>msearch.c</i>) shows how to define a model and then find this model in a target image. It also demonstrates the sub-pixel accuracy of <i>MpatFindModel()</i>.</p>

```

/* 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.      */

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

    (cont...)

```

```

/* 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);

/* 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);
}

(cont...)

```

```

/* 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);
}

```

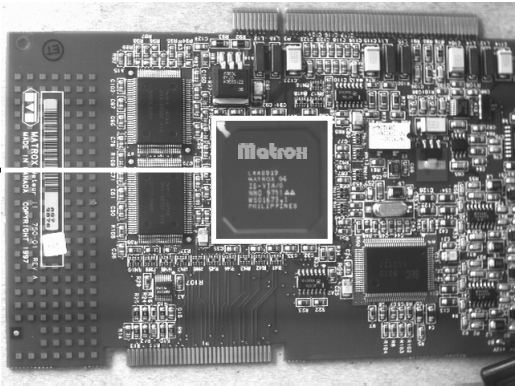
# Rotation

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




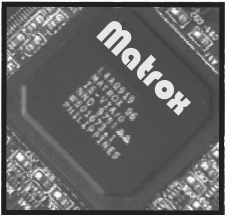
Model

An M\_NORMALIZED model internally rotated

An M\_NORMALIZED+M\_CIRCULAR\_OVERSCAN model internally rotated







= resulting M\_DONT\_CARE pixels



The following describes how to create the models to perform these types of searches.

### Creating rotated versions of the models

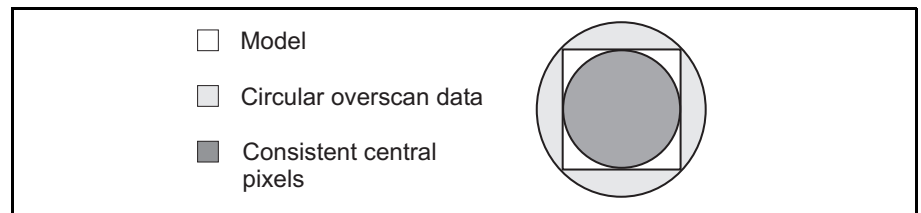
To implement the first type of search, allocate an `M_NORMALIZED` model with *MpatAllocModel()*. Then, enable and specify the angular range in which it can appear with *MpatSetAngle()*. When you call *MpatPreprocModel()*, it will internally create different models by rotating the original model at the 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 *MpatPreprocModel()*, 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.



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  $0^\circ$ . However, you can specify a rotational range of up to  $360^\circ$ , 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()*, *MpatPreprocModel()* 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.

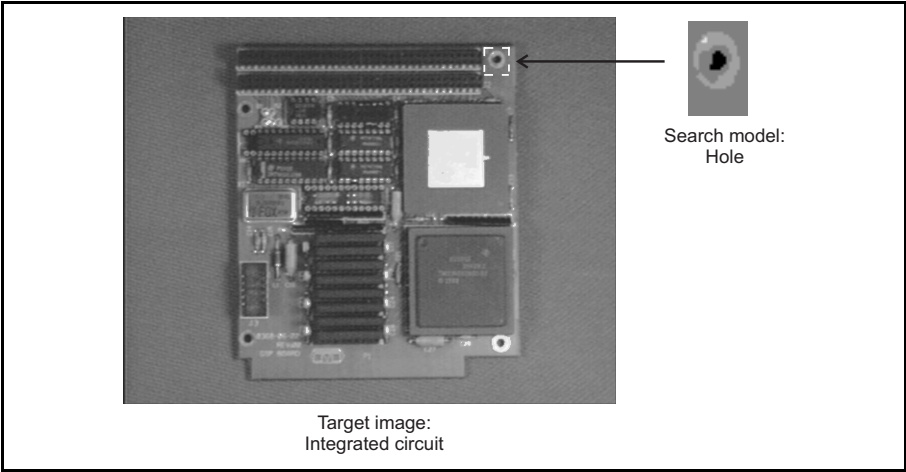
## Masking the model

---

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

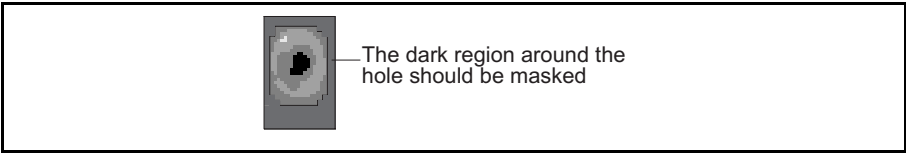
For instance, in a machine guidance application, a mechanical device might need to know where mounting holes are located on a circuit board so that screws can be properly inserted. In this case, a mounting hole would be the search model and the circuit board would be the target image.



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 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, use *MpatDraw()*, with `M_DRAW_DONT_CARES`. Alternatively, you can 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.

Typical match has an  
80 to 100%  
correlation

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%. 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.

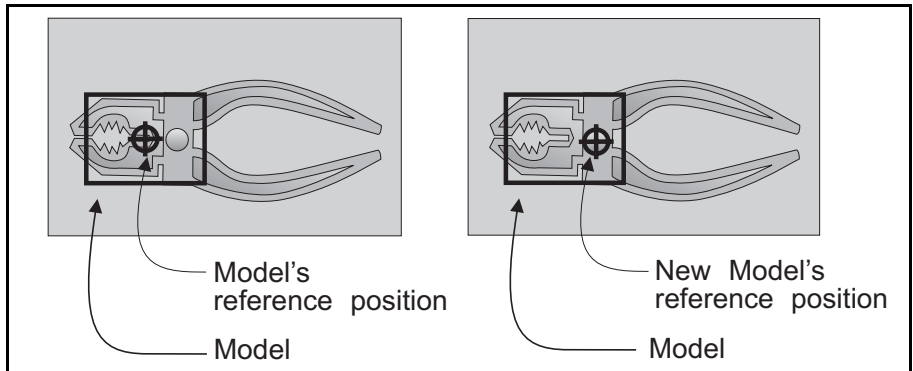
Often, you know that the pattern you want is unique in the image, so anything that reaches the acceptance level must be the match you want; therefore, you can set the certainty and acceptance levels to the same value.

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

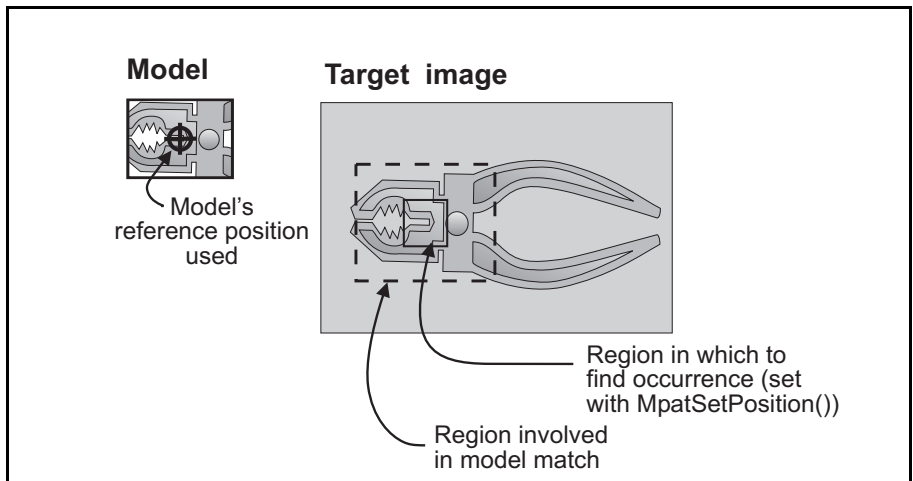
The coordinates returned by *MpatGetResults()*, after a call to *MpatFindModel()* are the coordinates of the model's reference position (in pixel or real-world units, depending on whether the imaging setup is calibrated; see *Chapter 7 Calibration*). By default, this reference position is defined to 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  $\pm 0.20$  pixel)
- M\_MEDIUM (typically  $\pm 0.10$  pixel)
- M\_HIGH (typically  $\pm 0.05$  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:

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

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"):

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

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

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:

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

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:

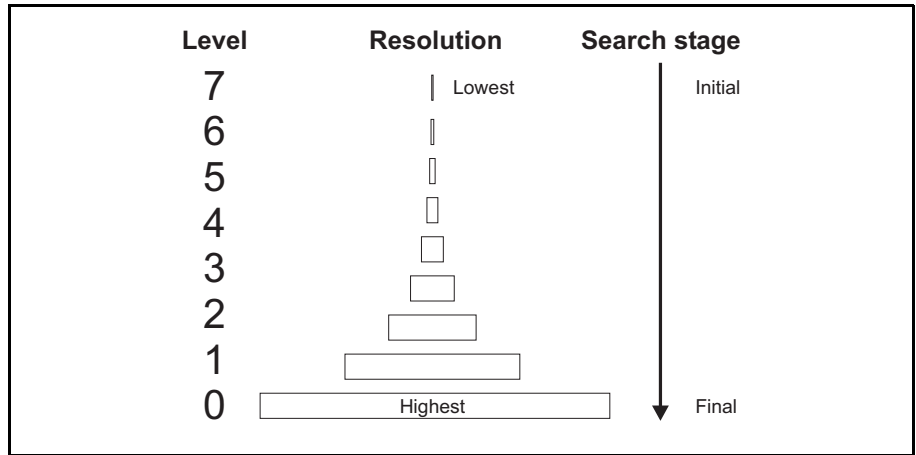
$$\sum I, \sum I^2, \sum IM$$

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  $512 \times 512$  at level 0, then at level 1 it is  $256 \times 256$ , at level 2 it is  $128 \times 128$ , 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 ( $128 \times 128$  model and  $512 \times 512$  image), it might start the search at level 4, which would mean using an  $8 \times 8$  version of the model and a  $32 \times 32$  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 as 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

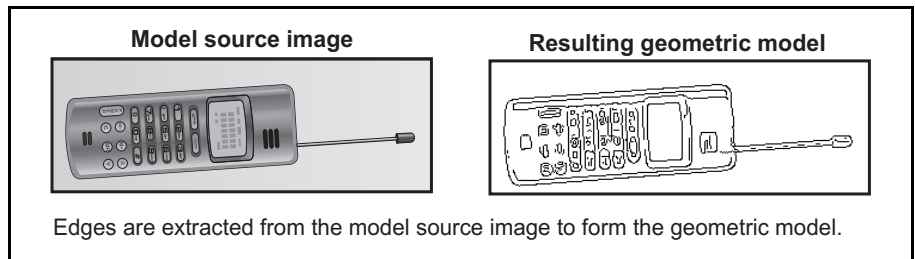
## **Geometric Model Finder**

This chapter explains how to use the MIL Geometric Model Finder to locate occurrences of models in your target image.

## The MIL Geometric Model Finder module

---

The MIL Geometric Model Finder module is a set of functions to find patterns, or models, based on geometric features. The algorithm finds models using edge-based geometric features instead of a pixel-to-pixel correlation. As such, the Geometric Model Finder module offers several advantages over correlation pattern matching, including greater tolerance of lighting variations (including specular reflection), model occlusion, as well as variations in scale and angle.



The MIL Geometric Model Finder module allows you to tailor your search to fit the requirements of your application. You can search for any number of different models simultaneously, through a range of angles and scale. The module also provides complete support for calibration. Searches can be performed in the calibrated real-world such that, even without physically correcting your images, occurrences can be found even in the presence of complex distortions, and results returned in real-world units.

## Steps to performing a search

---

The following steps provide a basic methodology for using the MIL Geometric Model Finder module:

1. Allocate your model finder context, using *MmodAlloc()*.
2. Define and add your model(s) to this model finder context, using *MmodDefine()*.
3. If necessary, mask any irrelevant, inconsistent, or featureless areas of your models, using *MmodMask()*.
4. Specify your required search settings for both the context and the individual model(s), using successive calls to *MmodControl()*.
5. Preprocess your model finder context, using *MmodPreprocess()*.
6. Allocate a result buffer to hold the results of your search, using *MmodAllocResult()*.
7. Search the target image for occurrences of models in your model finder context, using *MmodFind()*.
8. Retrieve the required results from the result buffer, using *MmodGetResult()*.
9. If necessary, save your model finder context, using *MmodSave()*.
10. Free all your allocated objects using *MmodFree()*.

If you are using calibrated model source and target images, refer to the *Calibration* section at the end of this chapter.

## Basic concepts

---

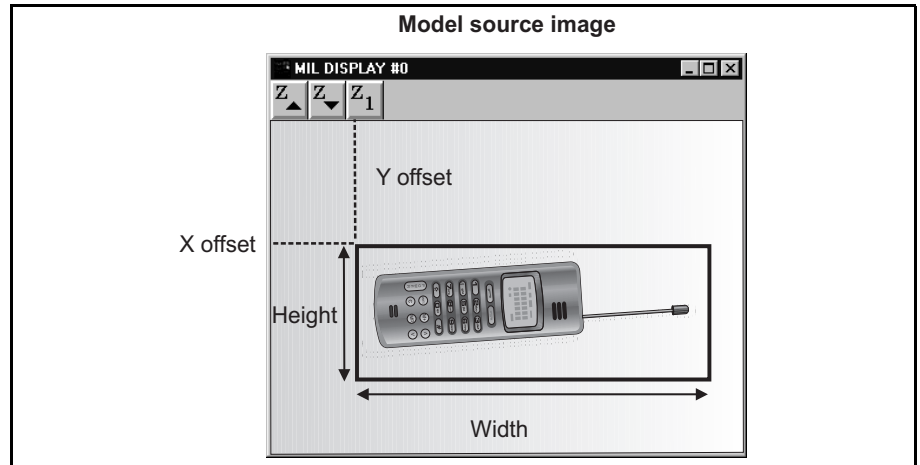
The basic concepts and vocabulary conventions for the MIL Geometric Model Finder module are:

- **Edges.** Transitions in grayscale value over several adjacent pixels. Well-defined edges have sharp transitions in value. The smoother the image, the more gradual the change, and the weaker the edge.
- **Active edges.** Edges which are extracted from the model source image to compose the geometric model, and searched for in the target image.
- **Model.** The pattern of active edges to find in the target image.
- **Occurrence.** An instance of the model found in the target image.
- **Bounding box.** The boundary of the square or rectangular region which defines the height and width of the model or occurrence.
- **Model source image.** The image from which to extract the model's active edges. In MIL, a model can be defined from any 1-band, 8-bit unsigned image.
- **Model finder context.** The container for all models you want to find. The model finder context allows you to set global search settings for all the models contained within the context.
- **Mask.** A binary image used to define irrelevant, inconsistent, or featureless areas in the model, so that only the pertinent model details are used for the search. Non-zero values indicate masked regions.
- **Target image.** The image which will be searched for occurrences of the model. In MIL, the target image can be any 1-band, 8-bit unsigned image.

## Defining and adding models to your model finder context

Once you have allocated a model finder context using *MmodAlloc()*, you can begin to add models to your model finder context, using *MmodDefine()*. The *MmodDefine()* function requires that you specify the model source image; models can be added to your model finder context from different 8-bit unsigned images.

You can use the entire image (if you have already cropped the area intended for the model), a child buffer, or you can designate the model region using the offset and size parameters of *MmodDefine()*. The minimum size of a model is limited to 16x16 pixels, while a maximum size of 1024x1024 pixels is permitted.



### Model indices and labels

Each model added to a model finder context can be accessed by its index. The first model is assigned an index of 0, and for each subsequent model added, the index is increased by one. You can also apply setting changes to all models within a model finder context by using *M\_ALL* as the index. When a model is deleted, all model indices greater than the deleted model are shifted down one. MIL searches for all models within a context in parallel, therefore the model index does not have any bearing in terms of a search order.

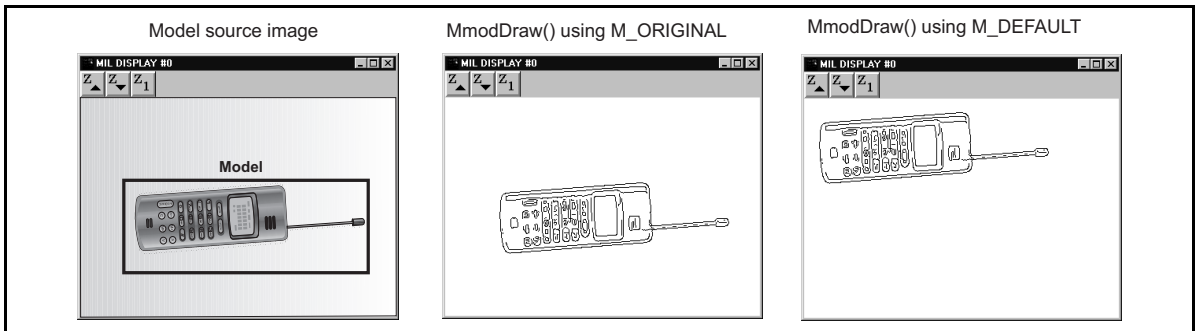
Models can also be given a user-defined numeric label, using *MmodControl()* with *M\_USER\_LABEL*. A user label can be used as a means of identifying your model, independently from its index in the model finder context. However, user labels

cannot be used as a direct replacement for the index; to retrieve the index of a model from the user label, use *MmodInquire()* with `M_INDEX_FROM_LABEL`. The user label, once converted to an index value, can be used with any *Mmod...*() function that takes an index value. All user labels must be unique integers; that is, no two user labels can have the same integer. To remove a label from a model, set the user label to `M_NO_LABEL`.

### Drawing the model's active edges

When testing different model source images for the best model candidate, use *MmodDraw()* to draw the model's active edges. The resulting edge map can reveal whether image processing control types (see the following section) need to be adjusted or if unwanted edges need to be masked.

If you have used offsets to define the model region in the model source image, you can draw the active edges at the position where the model was defined using *MmodDraw()* with `M_ORIGINAL`; otherwise the active edges will be drawn at the top-left corner.

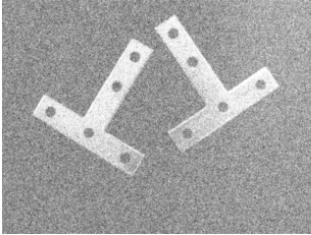


### Extracting edges

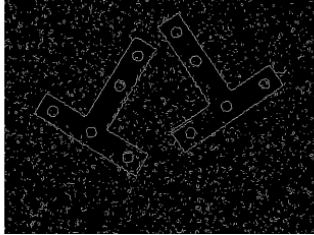
The MIL Geometric Model Finder module uses custom image processing algorithms to smooth, reduce noise, and extract edges in your model source and target images. The *MmodControl()* `M_SMOOTHNESS` and `M_DETAIL_LEVEL` control types control these image processing algorithms, determining which active edges are extracted from the model source and target images.



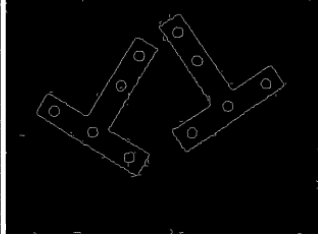
The `M_SMOOTHNESS` setting allows you to control the smoothing level of the edge extraction filter. The smoothing operation evens out rough edges and removes noise. The range of this control varies from 0 (no smooth) to 100 (a very strong smooth). The default setting is 50.



Target image with  
considerable noise



Edge map obtained with  
`M_SMOOTHNESS` set to 50

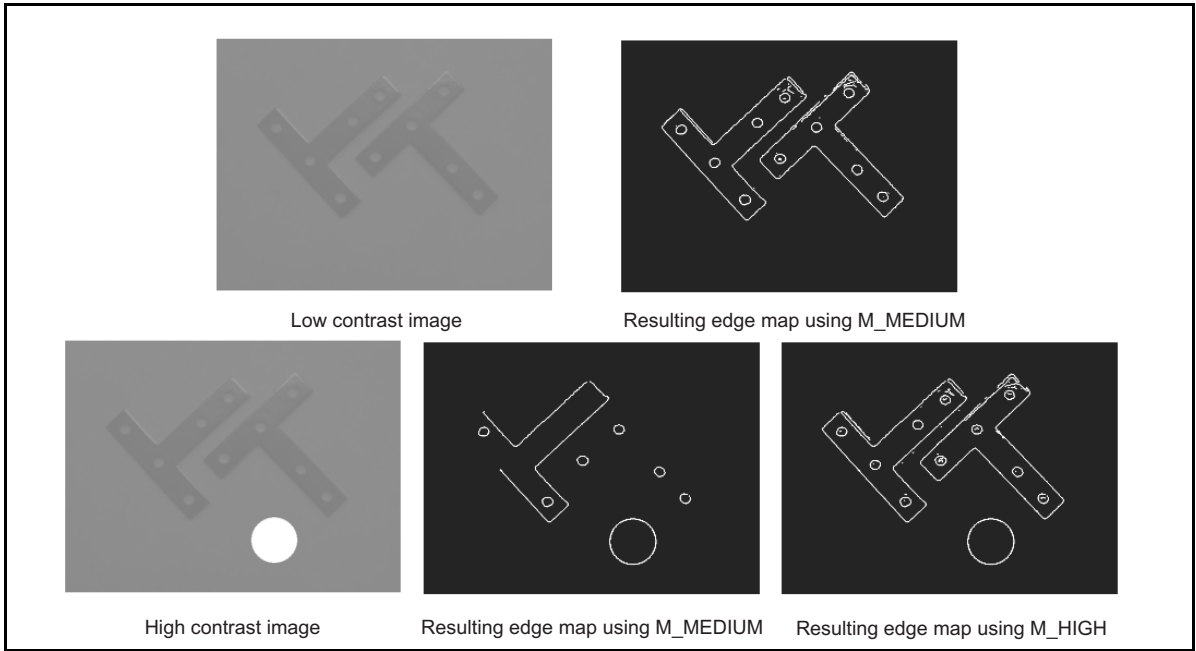


Edge map obtained with  
`M_SMOOTHNESS` set to 70

Note that using a very high smoothing level can result in a loss of important detail and a decrease in precision.

The `M_DETAIL_LEVEL` setting determines what is considered an edge. Edges are defined by the transition in grayscale value between adjacent pixels. The default mode (`M_MEDIUM`) offers a robust detection of active edges from images with contrast variation, noise, and non-uniform illumination. Nevertheless, in cases where objects of interest have a very low contrast compared to high contrast areas in the image, some low contrast edges can be missed.

The following examples show the use of the `M_DETAIL_LEVEL` control:



If your images contain low-contrast objects, a detail level setting of `M_HIGH` should be used to ensure the detection of all important edges in the image. The `M_VERY_HIGH` setting performs an exhaustive edge extraction, including very low contrast edges. However, it should be noted that this mode is very sensitive to noise.

The `M_SMOOTHING` and `M_DETAIL_LEVEL` control types are applied to both the model source and the target images for the specified model finder context. Note that the model source and target images are not directly modified; these controls merely extract the edge-based information from the images.

Generally, the default settings for these control types are sufficient for the majority of images; you should adjust these settings only when dealing with very noisy images, extremely low or high contrasted images, or images with very thin, refined features. In such cases, it is recommended that you experiment with different settings to achieve the necessary level of accuracy and speed required by your application.

## Guidelines for choosing models

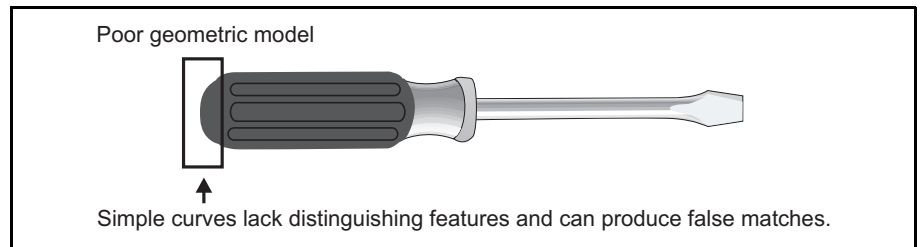
While finding models based on geometric features is a robust, reliable technique, there are a few pitfalls which you should be aware of when allocating your models so that you choose the best possible model.

### Make sure your images have enough contrast

Contrast is necessary for identifying edges in your model source and target image with sub-pixel accuracy. Images with weak contrast should be avoided since the MIL Geometric Model Finder module uses a geometric searching algorithm; the weaker the contrast, the less the amount and accuracy of edge-based information with which to perform a search. For this reason, it is recommended that you avoid models that contain only slow gradations in grayscale values. You should maintain a difference in grayscale values of at least 10 between the background and the edges in your images.

### Avoid poor geometric models

Poor geometric models suffer from a lack of clearly defined geometric characteristics, or from geometric characteristics that do not distinguish themselves sufficiently from other image features. These models can produce unreliable results.



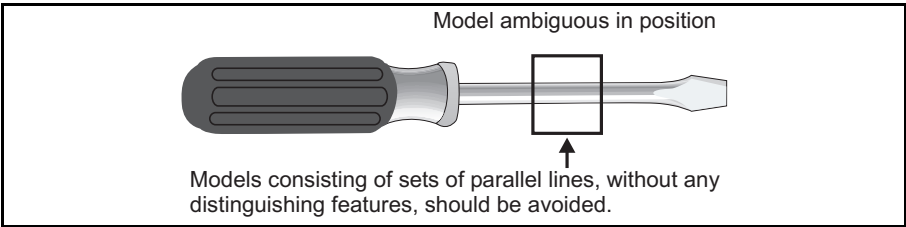
### Avoid ambiguous models

Certain types of geometric models provide non-unique, or ambiguous, results in terms of position, angle, or scale.

#### Models ambiguous in position

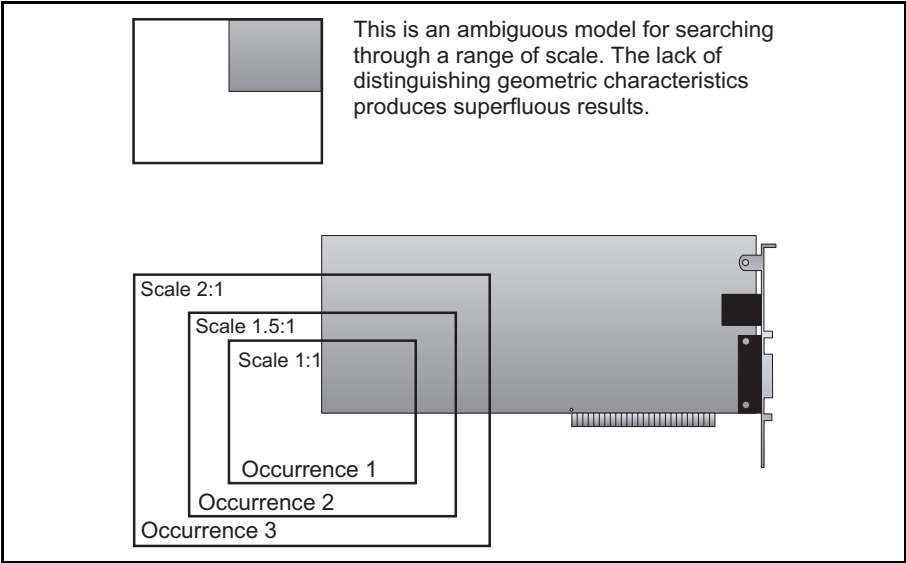
Models which are ambiguous in position are usually composed of one or more sets of parallel lines only. For example, models consisting of only parallel lines should be avoided since it is impossible to establish an accurate position for them. An infinite number of matches can be found since the actual number of line

segments in any particular line is theoretically limitless. Line segments should always contain some distinguishing contours that make them distinct from other image details.



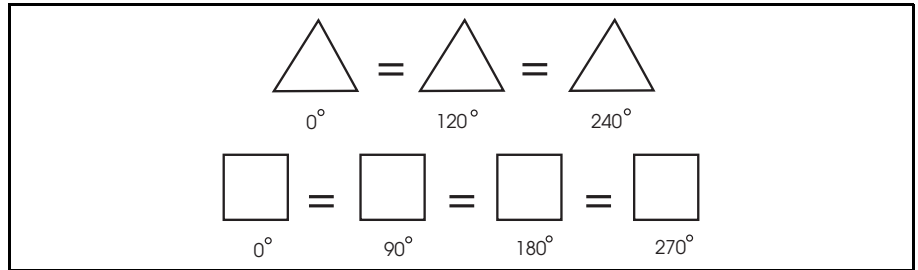
Models ambiguous in scale

Models which consist of small portions of objects should be tested to verify that they are not ambiguous in scale. For example, models which consist of isolated corners are ambiguous in terms of scale.



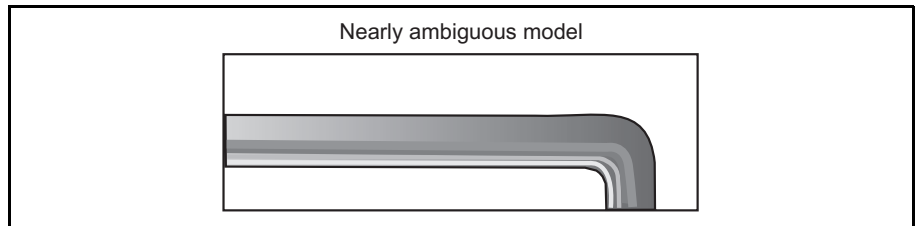
### Models ambiguous in angle

Symmetric models are often ambiguous in angle due to their similarity in features. For example, circles are completely ambiguous in terms of angle. Other simple symmetric models, such as squares and triangles, are ambiguous with respect to certain angles:



### Nearly ambiguous models

When the major part of a model contains ambiguous features, false matches can occur because the percentage of the occurrence's edges involved in the ambiguous features is great enough to be considered a match (see the *Determining what is a match* section later in this chapter). To avoid this, make sure that your models have enough distinct features to be found among other target image features. This will ensure that only correct matches are returned as results. For example, the model below can produce false matches since the greater proportion of active edges in the model is composed of parallel straight lines rather than distinguishing curves.



## Masking your model

---

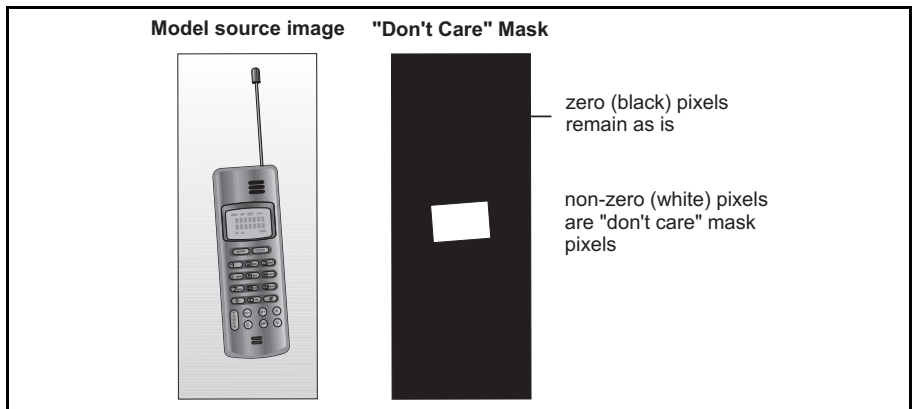
Once you have added a model to your model finder context, you can then mask any irrelevant, inconsistent, or featureless areas of your model using *MmodMask()*. A mask can be one of two types: a "don't care" or "flat region" mask.

With a "don't care" mask, MIL ignores the masked regions when searching for occurrences of the model. These regions might be noise edges, unwanted edges, inconsistent features, or simply regions which are irrelevant to your search.

With a "flat region" mask, the masked region is expected to be featureless in the found occurrence, and any edges present in these regions will reduce the target score. For example, in a quality control application, features that are present in a region where none are expected can indicate that an error has occurred in the manufacturing process.

A separate image is used to create each mask, and it must be the same size as the model. The mask image is treated as a simple binary image, where any non-zero pixel value is considered a masked pixel.

In the example below, a mobile phone is used as the model. However, occurrences of this model have displays containing inconsistent characters from target image to target image. Since these characters are variable, they should be masked in the model:



Note that the masked region should be slightly larger than the unwanted edges to ensure that they are masked.

If you want to verify the mask which has been applied to a model, you can use *MmodDraw()* to draw the model's "don't care" or "flat region" pixels. If necessary, you can clear a model's current mask by setting the mask's image buffer to `M_NULL`.

Finally, when you change the masked pixels of a model, you must preprocess the search model again.

## Determining what is a match

---

Before customizing your search settings, it is necessary to understand how a match between your model and occurrences in the target image is determined. The score (`M_SCORE`) and the target score (`M_TARGET_SCORE`) are the primary factors in determining which occurrences are considered matches with the models in your model finder context.

The score is a measure of the active edges in the model found in the occurrence, weighted by the deviation in position of these common edges. The target score is a measure of edges found in the occurrence that are not present in the original model (that is, extra edges), weighted by the deviation in position of the common edges. Edges found in the occurrence that are not present in the model will reduce the target score. These scores are calculated as follows:

Score = Model coverage x (1 - Fit error weighting factor x Normalized fit error)

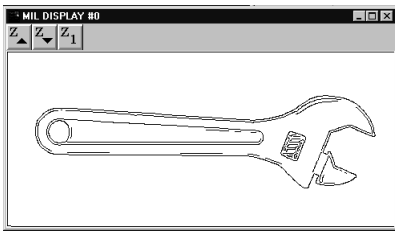
Target Score = Target coverage x (1 - Fit error weighting factor x Normalized fit error)

Note: the normalized fit error is the fit error converted to a number between 0.0-1.0.

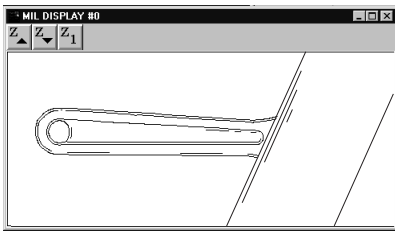
The model coverage, target coverage, and fit error components of the score and target score are explained below.

- **Model coverage.** The model coverage is the percentage of the total length of the model's active edges found in the occurrence. 100% indicates that for every edge in the model, a corresponding edge was found in the occurrence.

- **Target coverage.** The target coverage is the percentage of the total length of the model's active edges found in the occurrence, divided by the total length of edges present in the the occurrence's bounding box. Thus, a target coverage score of 100% means that no extra edges were found. Lower scores indicate that features or edges found in the target (result occurrence) are not present in the model.



Model's active edges



Occurrence edge map

**Model coverage** ( x 100 for percentage)

$$\left( \frac{\text{Length of the model's active edges found in the occurrence}}{\text{Length of the model's active edges}} \right) = \left( \frac{\text{Wrench handle edge map}}{\text{Wrench model}} \right)$$

**Target coverage** ( x 100 for percentage)

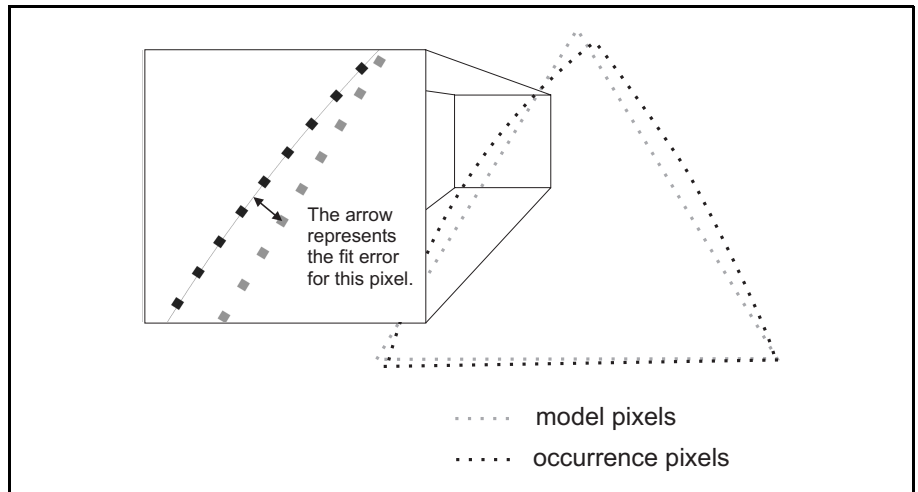
$$\left( \frac{\text{Length of the model's active edges found in the occurrence}}{\text{Total length of edges found in the occurrence}} \right) = \left( \frac{\text{Wrench handle edge map}}{\text{Wrench occurrence edge map}} \right)$$



- **Fit error.** The fit error is a measure of how well the edges of the occurrence correspond to those of the model. The fit error is calculated as the average quadratic distance, in pixels or calibrated units, between the sub-pixels in the occurrence and the corresponding active edges in the model:


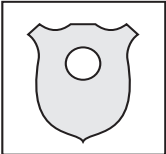
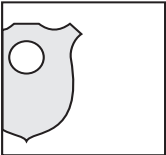
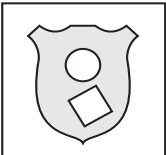
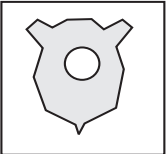
$$\text{Fit error} = \frac{\sum_{\text{All common pixels}} (\text{Error in X})^2 + (\text{Error in Y})^2}{\text{Number of common pixels}}$$

A perfect fit gives a fit error of 0.0. The fit error weighting factor (between 0.0 - 100.0) determines the importance to place on the fit error when calculating the score and target score.



Interpreting results

The following diagram illustrates how the model coverage, target coverage, and fit error work together to provide details about the nature of a result occurrence.

Model		
		
The image on the left will be used as our sample model to explain how the different scores can provide information about particular occurrences found. Note that values given are for illustrative purposes only, and as such are only qualitative.		
Target images		
	Model Coverage = 100% Target Coverage = 100% Fit Error = 0.0	This occurrence has a model coverage of 100% meaning that a perfect match has been found. All the edges in the model are found in the occurrence, all the edges in the occurrence are found in the model, and are a perfect fit.
	Model Coverage = 70% Target Coverage = 100% Fit Error = 0.0	This occurrence has a model coverage of 70%. The target coverage is still 100% since all the edges in the occurrence are found in the model. The fit error is 0.0 because all the model edges found in the occurrence are a perfect fit.
	Model Coverage = 100% Target Coverage = 70% Fit Error = 0.0	This occurrence has a target coverage of 70% because of the presence of extra edges (the white square) found in the occurrence. The model coverage is still 100% since all the edges in the model have been found in the occurrence.
	Model Coverage = 100% Target Coverage = 100% Fit Error = 1.5	Finally, in this occurrence, we have a model coverage of 100%, since all the edges in the model have been found in the occurrence. The edges found in the occurrence do not conform perfectly in shape to those in the model, resulting in a higher fit error score.

## Customizing search settings

---

With successive calls to *MmodControl()*, you can customize all of your model finder context settings and individual model search settings, using either `M_CONTEXT` or the individual model's index as the index parameter. You can also apply settings to all the models within your context by using `M_ALL` as your index parameter. However, when using `M_ALL` as your index parameter, ensure that the specified settings are appropriate for all models.

You can inquire about any particular model finder context setting or individual model setting, using *MmodInquire()*. All of the model search settings can affect the speed and robustness of your application. It is recommended that you begin with the default settings, and then, as your application demands, adjust the individual settings one by one as required.

### Acceptance levels

Acceptance levels can be set for both the score and target score. The acceptance levels determine the minimum scores required for an occurrence to be considered a match.

MIL will search the target image for the required number of occurrences, returning the occurrences with the best scores above the acceptance levels. If the score or target score of an occurrence is less than the acceptance levels, it is not considered a match and no result will be returned.

You can set the acceptance level for the score of the specified model, using *MmodControl()* with `M_ACCEPTANCE`. For example, a setting of 100% means that you will only accept occurrences that contain every active edge in the model. That is, for every edge in the model, a equivalent edge must be found in the occurrence. However, perfect matches are generally unobtainable because of noise introduced when grabbing images and the influence of the fit error weighting factor. You should use a reasonable acceptance level that is high enough to avoid false matches, but not so high that occurrences are missed. If your images have considerable noise and/or distortion, or if occlusion of occurrences is expected, you might have to set the level below the default value of 60%.

You can set the target score required using `M_ACCEPTANCE_TARGET`. A setting of 100% for the target score means that you will not tolerate any extra edges, while a setting of 0% allows for any number of extra edges. Essentially, the target score acceptance level allows you to control how tolerant your search is to details not present in the original model. The default value is 0%.

### **Certainty levels**

The certainty levels are used to speed up the search when very good matches are expected; any occurrence found above the certainty level is considered a "certain" match.

The certainty levels determine the score and target score above which the algorithm can assume that it has found a match. Both of these scores must be above their respective certainty levels for an occurrence to be considered a certain match. If the required number of occurrences has been found above these certainty levels, MIL stops searching the target image for better ones. This can accelerate the search by avoiding exhaustive checking of all possible candidates.

You can set the certainty level for the score, using *MmodControl()* with `M_CERTAINTY` (the default setting is 80%). You can set the certainty level for the target score using `M_CERTAINTY_TARGET` (the default value is 0%).

If necessary, when searching for a fixed number of occurrences in a target image, you can ensure that MIL always returns those occurrences with the highest match score among those found, by setting the certainty level of each individual model to 100% (however, so doing will increase the search time). Otherwise, the search will stop after it has found the required number of occurrences above the certainty level, regardless of whether better matches can be found in the image.

### **Fit error weighting factor**

The fit error weighting factor, *MmodControl()* with `M_FIT_ERROR_WEIGHTING_FACTOR`, determines the relative importance of the fit error when calculating the match score and the target score. The higher the factor, the greater the influence that the fit error has on the resulting score and target score for an occurrence (see the previous section, *Determining what is a match*, for more information). For example, setting this factor to 0.0 means that the fit error is not considered in the score calculation. Setting this factor to 100.0 means that the fit error is given the same weight as the model or target coverage when calculating the score or target score. The default value is 25%.

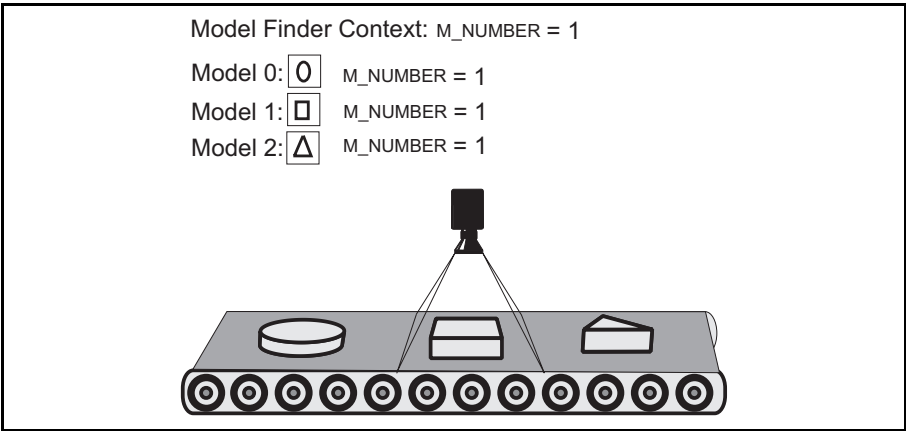
### Expected number of occurrences

You can set the expected number of occurrences for the model finder context, and for each individual model in the model finder context. For each individual model in the context, you can set the expected number of occurrences of the model to find in the target image. To do so, use *MmodControl()* with `M_NUMBER`, specifying the index of the particular model. The default value is 1. To find all occurrences of a model, set `M_NUMBER` to `M_ALL`. However, it is important to note that setting a model's expected number of occurrences to `M_ALL` can significantly slow the *MmodFind()* operation, depending on the complexity of your model and the target image (operation speed will only be slightly affected for simple models on a uniform background). It is recommended that you specify the exact number of expected occurrences whenever possible.

The number set for the context specifies the maximum total of all model occurrences (for all models within the context together). To set the number for a model finder context, use *MmodControl()* with `M_NUMBER`, specifying `M_CONTEXT` as the index. The default value is `M_ALL`, which finds the expected number of occurrences specified for each model.

MIL searches for all models within a context in parallel, therefore the model index does not have any bearing in terms of a search order. Once the number of occurrences for the context has been found, with scores above the certainty levels, the search will stop.

For example, in an application involving a number of different objects being examined one at a time on a conveyor belt, the actual object present in the target image at any time is unknown. Since only one occurrence is expected at any one time, the model finder context's number would be set to one, as would the number for all the models within the context.



MIL will search the target image for all three models; the first occurrence of any of these models, found above the certainty level, will stop the search.

To further illustrate the relationship between the M\_NUMBER setting for the context and that of the individual models, the table below shows the maximum possible results which can be returned for different context M\_NUMBER settings.

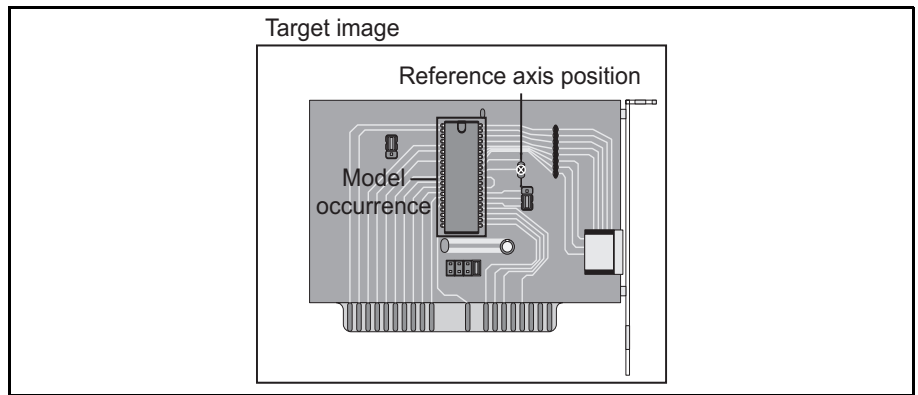
Model Finder Context Element	Setting for M_NUMBER	Maximum possible results						
Context	M_ALL	All						
Model 0	1	1						
Model 1	M_ALL	All						
Model 2	2	2						
Context	6	6						
Model 0	1	0	1	0	1	0	1	
Model 1	M_ALL	6	5	5	4	4	3	
Model 2	2	0	0	1	1	2	2	

Reference axis

When an occurrence is found, the model’s reference axis determines the coordinates and angle returned as the actual occurrence’s found position and angle. These coordinates are relative to the target image origin. By default, the reference axis origin is at the center of the model and is aligned with the axis of the model source image. You can change the position of this reference axis origin using *MmodControl()* with M\_REFERENCE\_X and M\_REFERENCE\_Y, specifying

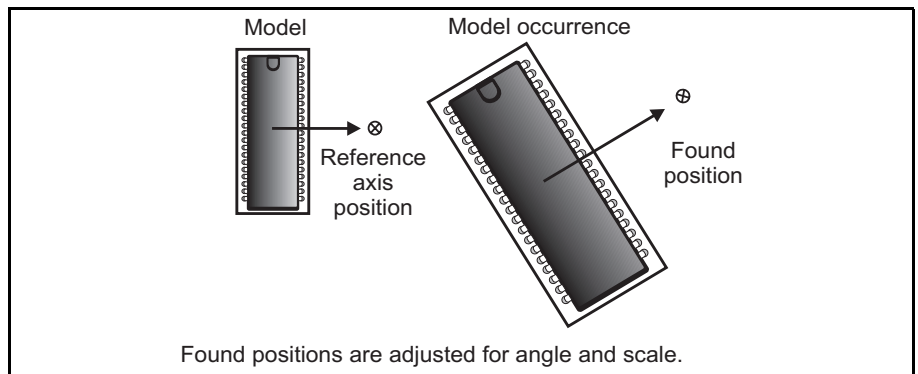
coordinates relative to the top-left corner of the model. Modifying the reference axis position can be useful when the coordinates of a specific position are necessary, but the most easily found model occurs at a position elsewhere in your image; you can find this model occurrence and have MIL return the coordinates of the required position using a custom reference axis position.

For example, in the diagram below, the easily found chip is used as the model, and the model's reference axis position is shifted to the location of the diode.



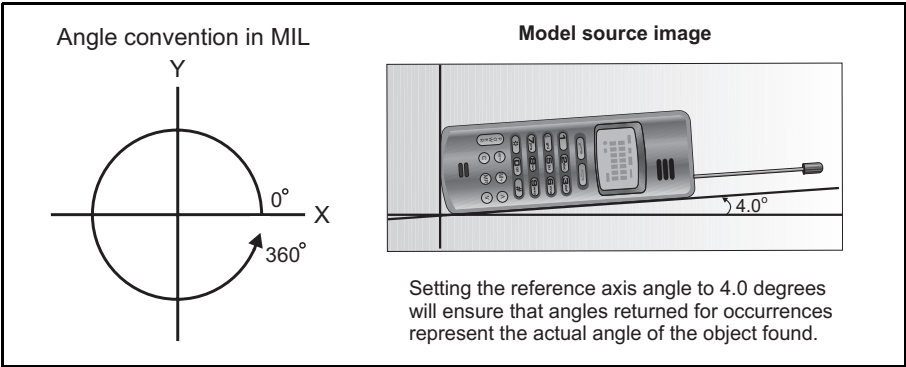
The reference axis position does not have to reside within the model, meaning that you can place the reference axis position outside of the model boundaries.

Note that the found position of an occurrence respects differences in angle and scale, meaning that the found position is relative to the angle and scale of the occurrence.



Reference angle

Often, when allocating models, the angle of the object in your model source image is not aligned with the horizontal image axis. When occurrences in your target image are successfully located, the angle of the occurrence returned is with respect to model source image axis. The actual angle of the object in the model is not taken into account. To correct this, you can set the reference angle of the model's reference axis using *MmodControl()* with M\_REFERENCE\_ANGLE. Angle results will now reflect the actual angle of the object, instead of the angle of the model image source axis.



Forward and reverse transformation coefficients

Forward and reverse transformation coefficients are available (as results) for mapping additional points of interest other than the reference axis position. These allow you to map any point in the model to that of an occurrence found in the target image (forward), or vice versa (reverse) with the following equations:

$$x_d = (a x_s) + (b y_s) + c$$
$$y_d = (-b x_s) + (a y_s) + d$$

where:

a, b, c, d: Transformation coefficients (forward or reverse).

$x_d, y_d$  : Destination coordinate (in target image for forward transformation or in the model region extracted from the model source image for reverse transformation).

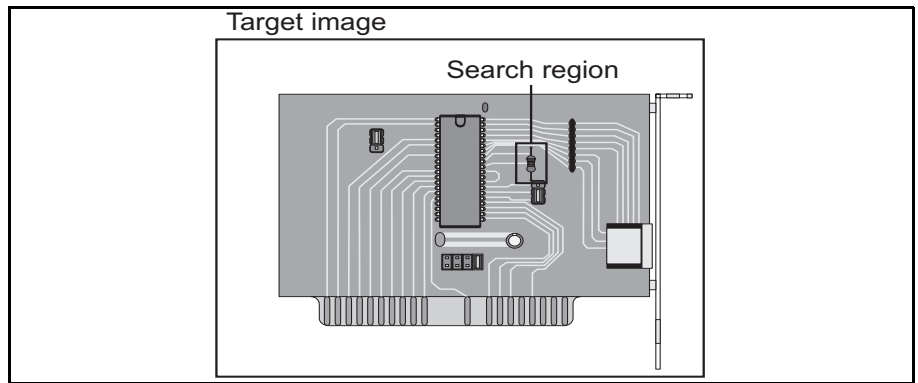
$x_s, y_s$  : Source coordinate (in the model region extracted from model source image for forward transformation or in target image for reverse transformation).



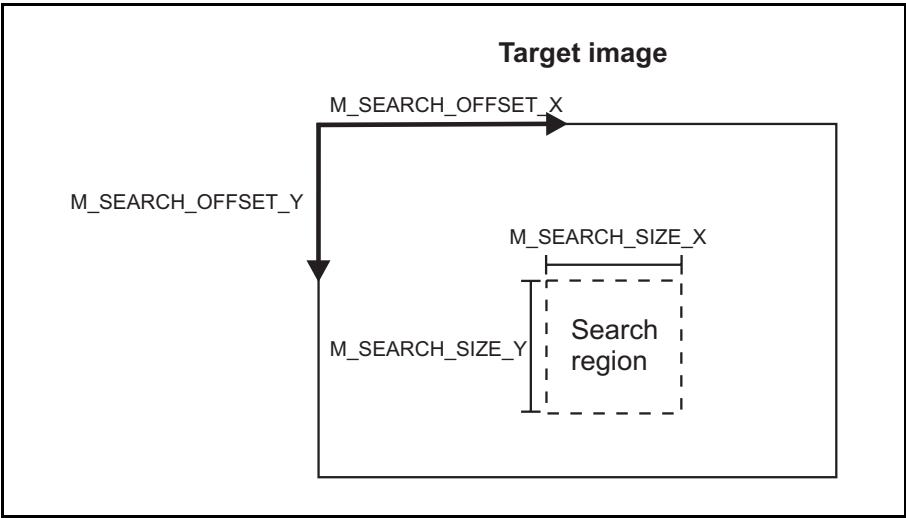
These coefficients can be useful when it is necessary to map several points, other than the reference axis position. These parameters take into consideration translation, rotation, and scale, allowing for easier mapping of critical points between model and occurrence.

### Search region

The search region limits the possible positions which can be returned for a model occurrence; position coordinates which fall outside of this region cannot be returned as results. If you have modified the reference axis position of your model, keep this in mind when selecting your search region, particularly when searching through a range of angles and/or scales. The search region can lie partially, or totally, outside the target image; this can be necessary, depending on the position of your reference axis position. For instance, in the example presented in the previous section, the search region must include, and can be limited to, the area including the diode for the chip to be found.



The default search region is the entire image plane (M\_ALL), meaning that all coordinates (even outside the target image) can be returned as results. Use the *MmodControl()* M\_SEARCH\_OFFSET\_X and M\_SEARCH\_OFFSET\_Y control types to set the search region's offset from the target image's origin, and M\_SEARCH\_SIZE\_X and M\_SEARCH\_SIZE\_Y to set the width and height, respectively, of the search region. You can, if necessary, set a different search region within the target image for each of the models within your context.



Depending on the level of details present in the target image, using a small search region generally decreases the search time. Always set the search region to the minimum required when speed is a consideration; there is no minimum search region size.

## Advanced search settings

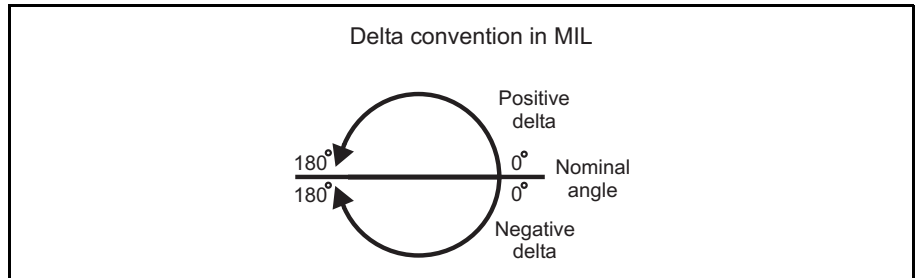
In addition to the fundamental search settings, the MIL Geometric Model Finder also provides advanced settings that allow you to search through ranges of angle and scale, to determine how much or little separation between occurrences is permitted, and to supply other features as well.

### Angle

You can search for each model in a model finder context at a specific angle, or through an angular range. For each model in your context, you can specify the angle of the search, using *MmodControl()* with `M_ANGLE`. By default, the search angle is 0°.

You can search through a full angular range of 360° from the nominal angle specified with `M_ANGLE`. The *MmodControl()* `M_ANGLE_DELTA_POS` and `M_ANGLE_DELTA_NEG` control types specify the angular range in the

counter-clockwise and clockwise direction, respectively. This angular range limits the possible angles which can be returned as results for an occurrence. Note that the actual angle of the occurrence does not affect search speed.



By default, searching through a range of angles is enabled. To search at a specific angle (M\_ANGLE) only, you must disable a search within an angular range for the model finder context, using *MmodControl()* with M\_CONTEXT as the index parameter and with the M\_SEARCH\_ANGLE\_RANGE control type set to M\_DISABLE. Note that, even with the search angle range disabled, MIL can typically still find occurrences within the specified angular range (depending on the characteristics of the actual model). If you want to ensure that MIL finds occurrences within a specific range of angles only, you should specify the required positive and negative deltas (the default for both is 180°).

### Scale

The scale of the model establishes the size of the model that you expect to find in the target image. If the expected occurrence is smaller or larger than that of the model, you can set the scale according to the supported scale factors. The scale is set for each individual model, using *MmodControl()* with M\_SCALE (0.5 - 2.0).

By default, searching through a scale range is disabled. If necessary, you can also enable a search through a range of scales for your model finder context using *MmodControl()* with M\_CONTEXT as the index parameter and the M\_SEARCH\_SCALE\_RANGE control type set to M\_ENABLE. This allows you to find models in the target image through a range of different sizes from the specified scale, both smaller or larger. To specify the range of scales, use *MmodControl()* with M\_SCALE\_MAX\_FACTOR and M\_SCALE\_MIN\_FACTOR. The minimum factor (0.5-1.0) and the maximum factor (1.0-2.0) together determine the scale range from the nominal scale (M\_SCALE).

These maximum and minimum factors are applied to the M\_SCALE setting as follows:

$$\text{Maximum scale} = (\text{M\_SCALE}) \times (\text{M\_SCALE\_MAX\_FACTOR})$$
$$\text{Minimum scale} = (\text{M\_SCALE}) \times (\text{M\_SCALE\_MIN\_FACTOR})$$

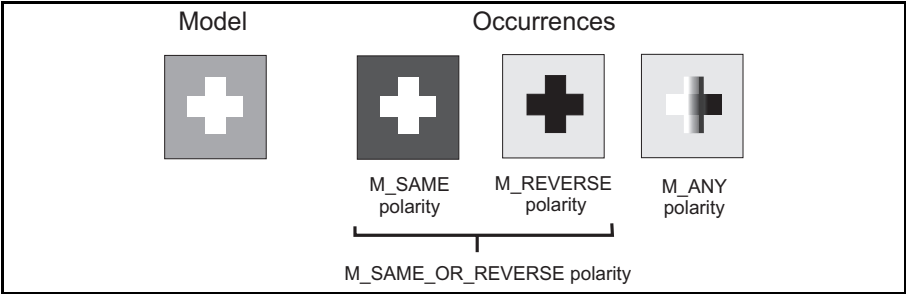
Note that the range is defined as factors so that if you change the expected scale (M\_SCALE), you do not have to modify the range. If you require a greater scale range than 0.25-4.0, you must allocate another larger or smaller version of the model.

A search through a range of scales is performed in parallel, meaning that the actual scale of an occurrence has no bearing on which occurrence will be found first. However, the greater the range of scale, the slower the search.

Note that if M\_SEARCH\_SCALE\_RANGE is disabled, MIL will typically continue to find occurrences within a scale factor of 0.95 to 1.05 times the specified model scale (M\_SCALE), depending on the characteristics of the actual model. If you want to ensure that MIL finds occurrences within a specific range of scale only, you should specify the required minimum and maximum factors (the defaults are 0.5 and 2.0, respectively).

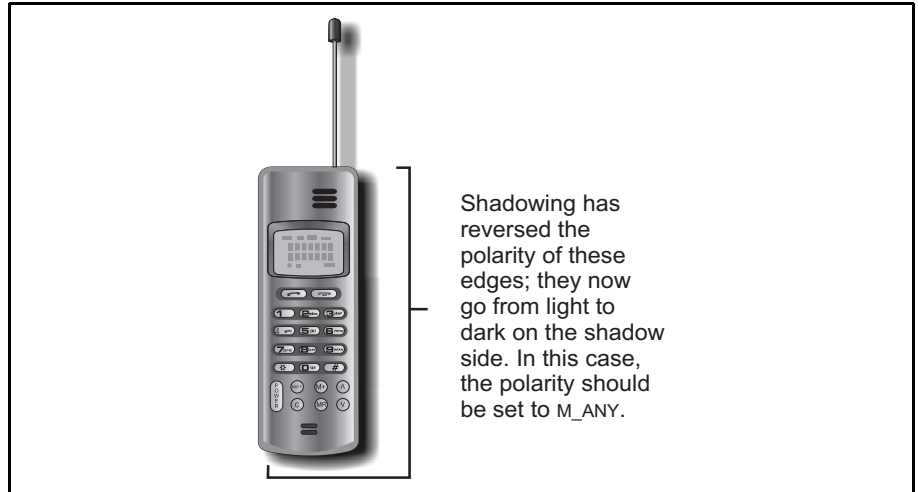
**Polarity**

The polarity of an edge indicates whether edges occur as transitions from light to dark or vice versa. When the polarity of edges in the target image might be different from those of the model (for example, due to shadows), you should specify the expected polarity using *MmodControl()* with M\_POLARITY. This control type allows you to specify whether edges present in the model and in the occurrence must have the same, the reverse, either of these, or a mixture of both polarities.



The default polarity setting is `M_SAME`.

The polarity setting can be useful when dealing with adverse lighting conditions where dark shadows can cause edges to vary in polarity.



## Separation

You can specify the minimum amount of separation from other occurrences (of the same model) necessary for an occurrence to be considered distinct (a match). In essence, this determines what amount of overlap by the same model occurrence is possible.

You can set the minimum separation for four criteria, which are: the X position, Y position, angle, and scale. These can be set for each individual model in your model finder context. For an occurrence to be considered distinct from another, only one of the minimum separation conditions needs to be met. For example, if the minimum separation in terms of angle is met, then the occurrence is considered distinct, regardless of the separation in position or scale. However, each of these separation criteria can be disabled (`M_DISABLE`) so that it is not considered when determining a valid occurrence.

The minimum positional separation (*MmodControl()* with `M_MIN_SEPARATION_X` and `M_MIN_SEPARATION_Y`) determines how far apart the found positions of two occurrences of the same model must be. This separation is specified as a percentage of the model size at the nominal scale (`M_SCALE`). The

default value is 10%. For example, if your model is 100 pixels wide at the nominal scale, setting M\_MIN\_SEPARATION to 10% would require that occurrences be 10 pixels apart in the X direction to be considered distinct and separate occurrences.

The minimum angular separation (M\_MIN\_SEPARATION\_ANGLE) determines the minimum difference in angle between occurrences. This value is specified as an absolute angle value. The default value is 10.0°.

The minimum scale separation (M\_MIN\_SEPARATION\_SCALE) determines the minimum difference in scale between occurrences, as a scale factor. The default value is 1.1.

The four criteria are summarized below in equation form.

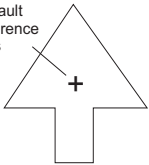
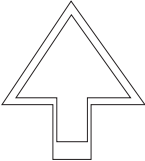
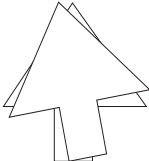
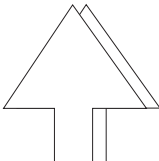
$$\frac{|\text{PositionX}_1 - \text{PositionX}_2|}{\text{Model width at M\_SCALE}} \geq \text{M\_MIN\_SEPARATION\_X}$$

$$\frac{|\text{PositionY}_1 - \text{PositionY}_2|}{\text{Model height at M\_SCALE}} \geq \text{M\_MIN\_SEPARATION\_Y}$$

$$|\text{Angle of Occurrence1} - \text{Angle of Occurrence2}| \geq \text{M\_MIN\_SEPARATION\_ANGLE}$$

$$\begin{array}{c} \text{Scale of Occurrence1/Scale of Occurrence2} \\ \text{or} \\ \text{Scale of Occurrence2/Scale of Occurrence1} \end{array} \geq \text{M\_MIN\_SEPARATION\_SCALE}$$

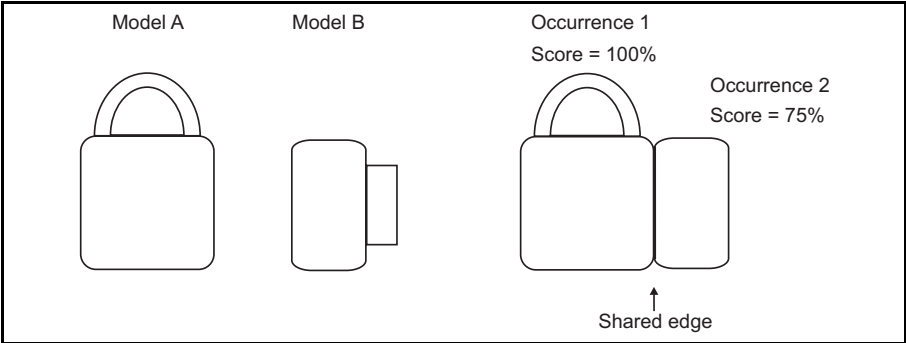
The following example illustrates the four separation criteria; an arrow represents two occurrences of the same model and the various types of separation possible.

 <p>Default reference axis</p>	<p>The model to the left has the following default settings:</p> <table border="0"> <tr> <td>M_MIN_SEPARATION_X</td> <td>= 10%</td> </tr> <tr> <td>M_MIN_SEPARATION_Y</td> <td>= 10%</td> </tr> <tr> <td>M_MIN_SEPARATION_ANGLE</td> <td>= 10°</td> </tr> <tr> <td>M_MIN_SEPARATION_SCALE</td> <td>= 1.1</td> </tr> </table>	M_MIN_SEPARATION_X	= 10%	M_MIN_SEPARATION_Y	= 10%	M_MIN_SEPARATION_ANGLE	= 10°	M_MIN_SEPARATION_SCALE	= 1.1
M_MIN_SEPARATION_X	= 10%								
M_MIN_SEPARATION_Y	= 10%								
M_MIN_SEPARATION_ANGLE	= 10°								
M_MIN_SEPARATION_SCALE	= 1.1								
	<p>This example illustrates a case with occurrences that have the same angle and reference axis position, but are separated by the minimum scale and therefore are considered distinct.</p>								
	<p>This example illustrates a case with occurrences that have the same scale and reference axis position, but are separated by the minimum angle and therefore are considered distinct.</p>								
	<p>This example illustrates a case with occurrences that have the same scale and angle, but are separated in the X direction by the minimum distance and therefore are considered distinct.</p>								

It should be noted that for a model to be found, the number of visible edges in the occurrence must be sufficient to provide a match according to your acceptance levels.

Shared edges

You can choose to allow occurrences to share edges, using *MmodControl()* with `M_SHARED_EDGES` set to `M_ENABLE`. Otherwise, edges that can be part of more than one occurrence are considered part of the occurrence with the greatest score. For example, in the illustration below, two occurrences of two simple models share a common edge. With shared edges enabled, these occurrences would have the following scores:



However, with shared edges disabled (default), the shared edge would be considered part of occurrence 1, since it has the greater score; the score of occurrence 2 would be subsequently reduced by the loss of the shared edge in the score calculation.



## Global context settings

---

The model finder context settings, which apply to every model, also determine the speed and robustness of your search, and can be adjusted to fit your individual application's needs. To specify these settings, use *MmodControl()* with the index parameter set to M\_CONTEXT. We recommend experimenting with different settings to find the particular settings that provide your application with the necessary robustness.

### Setting the search speed

You can specify the algorithm's search speed, using *MmodControl()* with M\_SPEED. The speed can be set to M\_MEDIUM (default), M\_HIGH, OR M\_VERY\_HIGH. Increasing the speed can adversely affect the robustness, as well as the accuracy, of the search.

### Accuracy

You can set the positional accuracy for your search, using *MmodControl()* with M\_ACCURACY. It can be set to:

- M\_MEDIUM (typically  $\pm 0.03$  pixel)
- M\_HIGH (typically  $\pm 0.02$  pixel)

Note, the actual precision achieved is dependent on the quality of the model and of the image (the values listed above are typical for high-quality, well-contrasted, low-noise images). The default setting is M\_MEDIUM.

Note that increasing the accuracy can adversely affect the speed.

### Timing out your search

In time critical applications, you can set a time limit in milliseconds, using *MmodControl()* with M\_TIMEOUT, for the model finder to find occurrences of the specified models. If the required number of occurrences is not found before the time limit is up, the search will stop. Results are still returned for those occurrences found. However, it is not possible to predict which occurrences will be found before the time limit is reached. The default value is 2000 msec.

## Retrieving and analyzing results

---

After having successfully located your model occurrences in your target image using *MmodFind()*, you can extract the required results from your result buffer using *MmodGetResult()*.

### Possible results

The MIL Geometric Model Finder provides several types of results which provide considerable information on the nature of the occurrence found. In addition to the score, target score, model coverage, target coverage, fit error, and forward/reverse transformation coefficients discussed previously in this chapter, results can be returned for:

- Number of occurrences
- Index or user label of occurrences
- Position
- Scale
- Polarity
- Angle

Results are returned in descending order of match score, such that the highest score is returned first. Generally, you should first retrieve the total number of occurrences found for all the models in the model finder context, to ascertain the size of the result array needed. Note that results are returned for the entire context; the model index (M\_INDEX result type) is used to differentiate results between models. For a complete description of all possible results, refer to the description of *MmodGetResult()* in the MIL Command Reference.

### Annotating results

The *MmodDraw()* function provides several operations for annotating your results. You can draw a bounding box surrounding the model occurrence, a cross-like symbol at the reference axis position and angle, or draw the occurrence's active edges. You can use a previously allocated graphics context (see Ch. 19 *Generating graphics*) to control the drawing color, or use the default graphics context (M\_DEFAULT). You can draw directly into the selected image buffer, or

annotate non-destructively into the image's overlay buffer (see Ch. 18 *Annotating the displayed image non-destructively*, or the *Mmodfind.c* example at the end of this chapter).

## Calibration

---

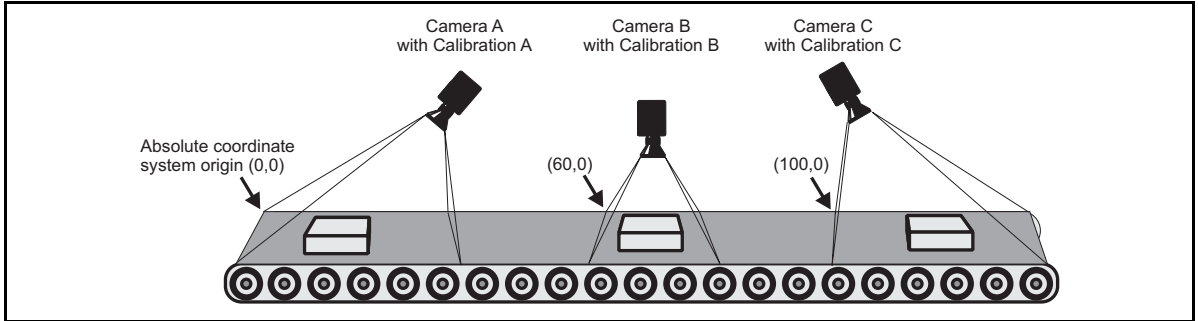
The MIL Geometric Model Finder module supports the use of calibrated images, meaning that searches will be performed in the real-world, compensating for image and/or perspective distortions, and results returned in real-world units. Calibration is performed transparently, without the need for any setting adjustments. Note that defining models from calibrated images and adjusting individual model settings is done in pixels, as with non-calibrated images.

To search in a calibrated target image with a model finder context, all models of the context must also be calibrated. To do so, use calibrated model source images or associate a calibration object to the models, using *MmodControl()* with *M\_ASSOCIATED\_CALIBRATION*. The model source images can be associated with different calibration objects, however, you should ensure that the same absolute coordinate system is used for all calibration objects (otherwise, results will be skewed).

Target images must also be calibrated using the same absolute coordinate system as the models in the model finder context. Images can be either physically corrected or not, without affecting the robustness of the search. By default, if models are defined from calibrated model source images and the target images are calibrated, results will be calibrated as well (see *Chapter 7: Calibration*, for more information on using calibrated images in MIL).

Target images can be grabbed from any number of cameras, each with its own calibration object, as long as the same absolute coordinate system has been used. For example, if a model has been extracted from a model source image taken using camera A, the model can be used to locate occurrences in a target image grabbed

by camera B or camera C (whether or not the image has been physically transformed) since the same absolute coordinate system has been used to calibrate all images; MIL will transparently convert between calibration objects.



Calibration objects are not saved with the model finder context. When restoring a model finder context from disk, you must re-associate your models with their calibration objects (which must also have been saved), using *MmodControl()* with `M_ASSOCIATED_CALIBRATION`.

If necessary, you can disassociate a calibration object from a model, by using *MmodControl()* with `M_ASSOCIATED_CALIBRATION` set to `M_NULL`. This can be useful if you want to use a calibrated model with an uncalibrated target image.

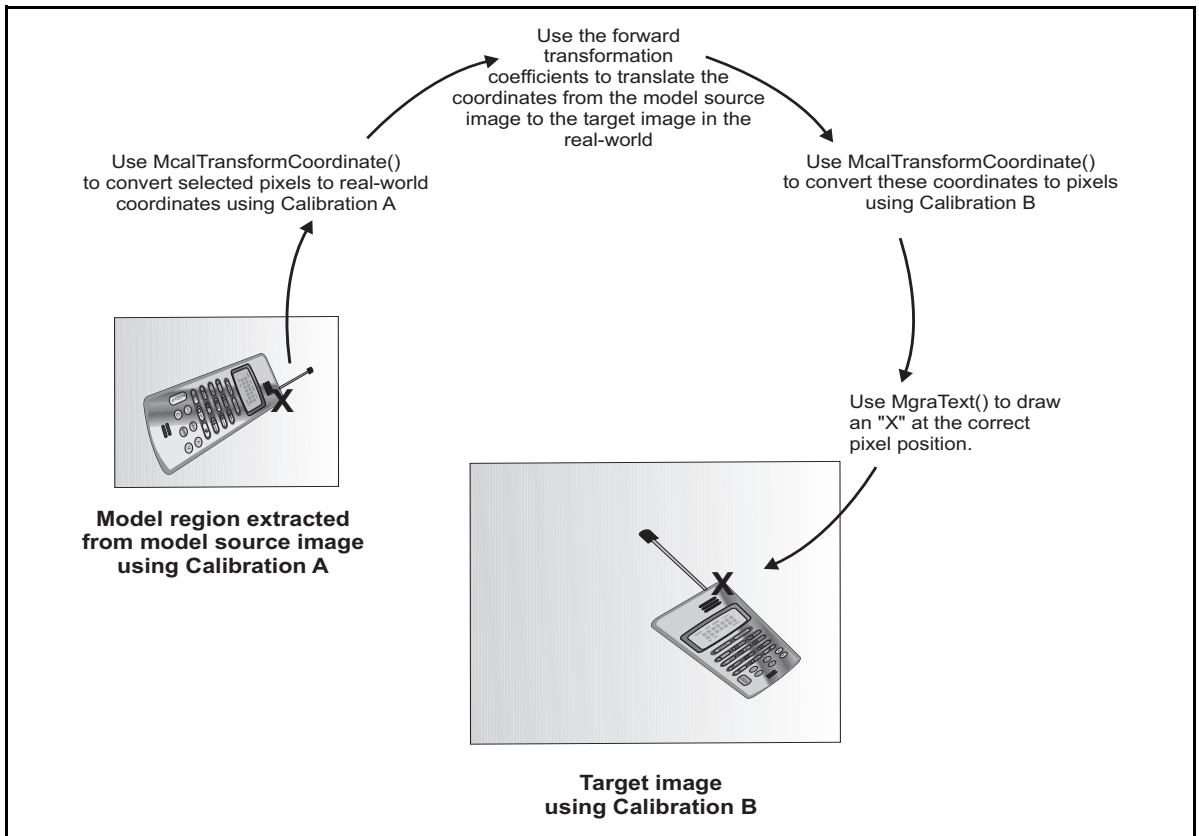
Finally, when using *MmodDraw()* to draw features from a calibrated model or results from a calibrated target image, the destination drawing buffer must also be calibrated, using the same absolute coordinate system, otherwise annotations will be skewed. MIL transparently takes into account the calibration when drawing these annotations.

### Using transformation coefficients with calibrated images

When using calibrated model source and target images, forward or reverse transformation coefficients are given in real-world units, meaning that they can be used to convert any model world coordinate to the corresponding occurrence world coordinate (or vice versa). These coefficients handle variations in scale, rotation, and angle in the real-world.

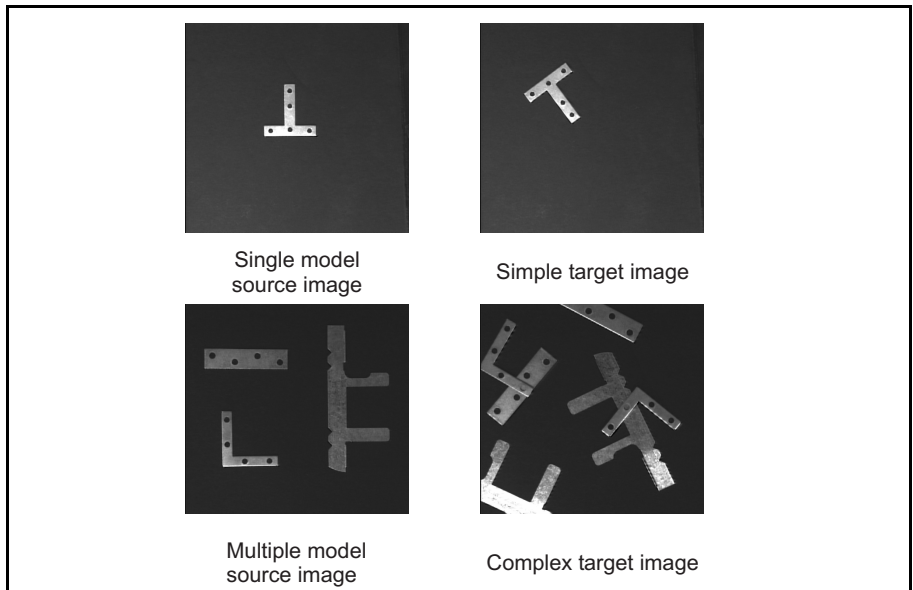
If you want to draw annotations (with the *Mgra...()* functions) in your calibrated image using real-world coordinates, you must convert them into image pixel coordinates. Since the transformation is applied in the real-world only, you need to use *McalTransformCoordinate()* to handle the effects of calibration in the image.

That is, you must convert the required pixel coordinates in the source image to real-world coordinates, apply the transformation coefficients, and convert these coordinates back to pixels in the target image. Using these coordinates, you can correctly draw the annotations in your calibrated target image. In this manner, provided that the same absolute coordinate system has been used, transformations can be performed using any two calibration objects.



## An example

The following sample program (*mmodfind.c*) shows how to define a model finder context with a single model, as well as with several models, and find these models in a target image.



```

/*****
/* File name: MModFind.c
/* Synopsis: This program defines models in a geometric context and searches
*            for them in a target image. A simple single model example
*            (1 model, 1 occurrence, good search conditions) is presented first,
*            followed by a more complex example (multi-model, multi-occurrences
*            in a complex scene with bad search conditions).
*/

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

/* Example selection. */
#define RUN_SINGLE_MODEL_EXAMPLE      1
#define RUN_MULTIPLE_MODELS_EXAMPLE  1

(cont...)

```

```

/* Example functions declarations. */
void SingleModelExample(MIL_ID MilSystem, MIL_ID MilDisplay);
void MultipleModelsExample(MIL_ID MilSystem, MIL_ID MilDisplay);

/*****
Main.
*****/
void main(void)
{
    MIL_ID MilApplication,      /* Application identifier. */
        MilSystem,             /* System Identifier.      */
        MilDisplay;            /* Display identifier.     */

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

    #if (RUN_SINGLE_MODEL_EXAMPLE)
        SingleModelExample(MilSystem, MilDisplay);
    #endif
    #if (RUN_MULTIPLE_MODELS_EXAMPLE)
        MultipleModelsExample(MilSystem, MilDisplay);
    #endif

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

/*****
Single model example.
*****/

/* Source MIL image file specifications. */
#define SINGLE_MODEL_IMAGE          M_IMAGE_PATH MT("SingleModel.mim")

/* Target MIL image file specifications. */
#define SINGLE_MODEL_TARGET_IMAGE    M_IMAGE_PATH MT("SimpleTarget.mim")

/* Search speed: M_VERY_HIGH for faster search, M_MEDIUM for greater
 *precision and robustness
 */
#define SINGLE_MODEL_SEARCH_SPEED    M_VERY_HIGH

/* Model specifications. */
#define MODEL_OFFSETX                176L
#define MODEL_OFFSETY                136L
#define MODEL_SIZEX                  128L
#define MODEL_SIZEY                  128L
#define MODEL_MAX_OCCURRENCES        16L
#define MODEL_DRAW_COLOR              M_RGB888(255, 0, 0)          /* Red */

(cont...)

```

```

void SingleModelExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID MilImage,                /* Image buffer identifier */
        MilOverlayImage;           /* Overlay image */
    MIL_ID MilSearchContext,         /* Search context */
        MilResult;                 /* Result identifier */
    long Model[MODEL_MAX_OCCURRENCES], /* Model index */
        ModelDrawColor = MODEL_DRAW_COLOR; /* Model draw color */
    long TransparentColor,           /* Overlay clear color */
        NumResults = 0L;            /* Number of results found */
    double Score[MODEL_MAX_OCCURRENCES], /* Model correlation score */
        XPosition[MODEL_MAX_OCCURRENCES], /* Model X position */
        YPosition[MODEL_MAX_OCCURRENCES], /* Model Y position */
        Angle[MODEL_MAX_OCCURRENCES], /* Model occurrence angle */
        Scale[MODEL_MAX_OCCURRENCES], /* Model occurrence scale */
        Time = 0.0;                 /* Bench variable. */
    int i;                           /* Loop variable */

    /* Restore the model image and display it */
    MbufRestore(SINGLE_MODEL_IMAGE, MilSystem, &MilImage);
    MdispSelect(MilDisplay, MilImage);

    /* Prepare for overlay annotations. */
    MdispControl(MilDisplay, M_WINDOW_OVR_WRITE, M_ENABLE);
    MdispInquire(MilDisplay, M_WINDOW_OVR_BUF_ID, &MilOverlayImage);
    MdispInquire(MilDisplay, M_KEY_COLOR, &TransparentColor);
    if (MbufInquire(MilOverlayImage, M_SIZE_BAND, M_NULL) == 1)
        ModelDrawColor = 0xFF;

    /* Allocate a geometric model finder. */
    MmodAlloc(MilSystem, M_GEOMETRIC, M_DEFAULT, &MilSearchContext);

    /* Allocate a result buffer. */
    MmodAllocResult(MilSystem, M_DEFAULT, &MilResult);

    /* Define the model */
    MmodDefine(MilSearchContext, M_IMAGE, MilImage, MODEL_OFFSETX, MODEL_OFFSETY,
        MODEL_SIZEEX, MODEL_SIZEY);

    /* Set the search speed */
    MmodControl(MilSearchContext, M_CONTEXT, M_SPEED, SINGLE_MODEL_SEARCH_SPEED);

    /* Preprocess the search context. */
    MmodPreprocess(MilSearchContext, M_DEFAULT);

    /* Draw a box and position it in the source image to show the model. */
    MgraColor(M_DEFAULT, ModelDrawColor);
    MmodDraw(M_DEFAULT, MilSearchContext, MilOverlayImage, M_DRAW_BOX+M_DRAW_POSITION, 0,
        M_ORIGINAL);

```

(cont...)



```

/* Pause to show the model. */
printf("A model finder context was defined with the model in the");
printf(" displayed image.\n");
printf("Press <Enter> to continue.\n");
getchar();

/* Clear the overlay image. */
MbufClear(MilOverlayImage, TransparentColor);

/* Load the single model target image. */
MbufLoad(SINGLE_MODEL_TARGET_IMAGE, MilImage);

/* Dummy first find for better function timing accuracy (model cache effect,...). */
MmodFind(MilSearchContext, MilImage, MilResult);

/* Search for the model. */
MappTimer(M_TIMER_RESET, M_NULL);
MmodFind(MilSearchContext, MilImage, MilResult);
MappTimer(M_TIMER_READ, &Time);

/* Get the number of models found. */
MmodGetResult(MilResult, M_DEFAULT, M_NUMBER+M_TYPE_LONG, &NumResults);

/* If a model was found above the acceptance threshold. */
if ( (NumResults >= 1) && (NumResults <= MODEL_MAX_OCCURRENCES) )
{
    /* Get the results for each model. */
    MmodGetResult(MilResult, M_DEFAULT, M_INDEX+M_TYPE_LONG, Model);
    MmodGetResult(MilResult, M_DEFAULT, M_POSITION_X, XPosition);
    MmodGetResult(MilResult, M_DEFAULT, M_POSITION_Y, YPosition);
    MmodGetResult(MilResult, M_DEFAULT, M_ANGLE, Angle);
    MmodGetResult(MilResult, M_DEFAULT, M_SCALE, Scale);
    MmodGetResult(MilResult, M_DEFAULT, M_SCORE, Score);

    /* Print the results for each model found. */
    printf("The model finder context was used to find the model in the");
    printf("target image.\n\n");
    printf("Result   Model   X Position   Y Position   Angle   Scale");
    printf("   Score\n\n");
    for (i=0; i<NumResults; i++)
    {
        printf("%-9d%-8d%-13.2f%-13.2f%-8.2f%-8.2f%-5.2f%%\n",
            i, Model[i], XPosition[i], YPosition[i], Angle[i], Scale[i], Score[i]);
    }
    printf("\nThe search time is %.1f ms\n\n", Time*1000.0);

    /* Draw edges, position and box over the occurrences that were found. */
    for (i=0; i<NumResults; i++)
    {
        MgraColor(M_DEFAULT, ModelDrawColor);
        MmodDraw(M_DEFAULT, MilResult, MilOverlayImage,
            M_DRAW_EDGES+M_DRAW_BOX+M_DRAW_POSITION, i, M_DEFAULT);
    }
}

(cont...)

```

```

    else
    {
        printf("The model was not found or the number of models found");
        printf(" is greater than\n");
        printf("the specified maximum number of occurrence !\n\n");
    }

    /* Wait for a keystroke. */
    printf("Press <Enter>.\n");
    getchar();

    /* Free MIL objects. */
    MbufFree(MilImage);
    MmodFree(MilSearchContext);
    MmodFree(MilResult);
}

/*****
Multiple models example.
*****/

/* Source MIL image file specifications. */
#define MULTI_MODELS_IMAGE      M_IMAGE_PATH MT("MultiModel.mim")

/* Target MIL image file specifications. */
#define MULTI_MODELS_TARGET_IMAGE M_IMAGE_PATH MT("ComplexTarget.mim")

/* Search speed: M_VERY_HIGH for faster search, M_MEDIUM for greater
*precision and robustness
*/
#define MULTI_MODELS_SEARCH_SPEED  M_VERY_HIGH

/* Number of models. */
#define NUMBER_OF_MODELS          3L
#define MODELS_MAX_OCCURRENCES    16L

/* Model 1 specifications. */
#define MODEL0_OFFSETX            34L
#define MODEL0_OFFSETY            93L
#define MODEL0_SIZEX              214L
#define MODEL0_SIZEY              76L
#define MODEL0_DRAW_COLOR         M_RGB888(255, 0, 0) /* Red */

/* Model 2 specifications. */
#define MODEL1_OFFSETX            73L
#define MODEL1_OFFSETY            232L
#define MODEL1_SIZEX              150L
#define MODEL1_SIZEY              154L
#define MODEL1_REFERENCECX        23L
#define MODEL1_REFERENCECY        127L
#define MODEL1_DRAW_COLOR         M_RGB888(0, 255, 0) /* Green */

(cont...)

```

```

/* Model 3 specifications. */
#define MODEL2_OFFSETX          308L
#define MODEL2_OFFSETY          39L
#define MODEL2_SIZEX            175L
#define MODEL2_SIZEY            357L
#define MODEL2_REFERENCECX      62L
#define MODEL2_REFERENCECY      150L
#define MODEL2_DRAW_COLOR       M_RGB888(0, 0, 255) /* Blue */

/* Models array specifications. */
#define MODELS_ARRAY_SIZE       3L
#define MODELS_OFFSETX          {MODEL0_OFFSETX, MODEL1_OFFSETX, MODEL2_OFFSETX}
#define MODELS_OFFSETY          {MODEL0_OFFSETY, MODEL1_OFFSETY, MODEL2_OFFSETY}
#define MODELS_SIZEX            {MODEL0_SIZEX, MODEL1_SIZEX, MODEL2_SIZEX}
#define MODELS_SIZEY            {MODEL0_SIZEY, MODEL1_SIZEY, MODEL2_SIZEY}
#define MODELS_DRAW_COLOR       {MODEL0_DRAW_COLOR,
MODEL1_DRAW_COLOR, MODEL2_DRAW_COLOR}

void MultipleModelsExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID MilImage,          /* Image buffer identifier. */
    MilOverlayImage;          /* Overlay image. */
    MIL_ID MilSearchContext    /* Search context */
    MilResult;                /* Result identifier. */
    long   Models[MODELS_MAX_OCCURRENCES], /* Models index. */
    ModelsOffsetX[MODELS_ARRAY_SIZE] = MODELS_OFFSETX,
    /* Models X offsets array. */
    ModelsOffsetY[MODELS_ARRAY_SIZE] = MODELS_OFFSETY,
    /* Models Y offsets array. */
    ModelsSizeX[MODELS_ARRAY_SIZE] = MODELS_SIZEX,
    /* Models X sizes array. */
    ModelsSizeY[MODELS_ARRAY_SIZE] = MODELS_SIZEY;
    /* Models Y sizes array. */
    long   ModelsDrawColor[MODELS_ARRAY_SIZE]=MODELS_DRAW_COLOR,
    /* Models draw colors */
    TransparentColor,          /* Overlay clear color. */
    NumResults = 0L;           /* Number of results found. */
    double Score[MODELS_MAX_OCCURRENCES], /* Models correlation score.*/
    XPosition[MODELS_MAX_OCCURRENCES],
    /* Models X position. */
    YPosition[MODELS_MAX_OCCURRENCES],
    /* Models Y position. */
    Angle[MODELS_MAX_OCCURRENCES], /* Models occurrence angle. */
    Scale[MODELS_MAX_OCCURRENCES], /* Models occurrence scale. */
    Time = 0.0;                /* Time variable. */
    int    i;                  /* Loop variable */

```

(cont...)

```

/* Restore the model image and display it */
MbufRestore(MULTI_MODELS_IMAGE, MilSystem, &MilImage);
MdispSelect(MilDisplay, MilImage);

/* Prepare for overlay annotations. */
MdispControl(MilDisplay, M_WINDOW_OVR_WRITE, M_ENABLE);
MdispInquire(MilDisplay, M_WINDOW_OVR_BUF_ID, &MilOverlayImage);
MdispInquire(MilDisplay, M_KEY_COLOR, &TransparentColor);
MbufClear(MilOverlayImage, TransparentColor);

/* If displaying in 8 bit resolution, set the annotation color to white. */
if (MbufInquire(MilOverlayImage, M_SIZE_BAND, M_NULL) == 1)
{
    for (i=0; i<MODELS_ARRAY_SIZE; i++)
        ModelsDrawColor[i] = 0xFF;
}

/* Allocate a geometric model finder. */
MmodAlloc(MilSystem, M_GEOMETRIC, M_DEFAULT, &MilSearchContext);

/* Allocate a result buffer. */
MmodAllocResult(MilSystem, M_DEFAULT, &MilResult);

/* Define the models. */
for (i=0; i<NUMBER_OF_MODELS; i++)
{
    MmodDefine(MilSearchContext, M_IMAGE, MilImage, ModelsOffsetX[i], ModelsOffsetY[i],
        ModelsSizeX[i], ModelsSizeY[i]);
}

/* Set the desired search speed */
MmodControl(MilSearchContext, M_CONTEXT, M_SPEED, MULTI_MODELS_SEARCH_SPEED);

/* Increase the smoothness for the edge extraction in the search context. */
MmodControl(MilSearchContext, M_CONTEXT, M_SMOOTHNESS, 75);

/* Modify the acceptance and the certainty for all the models that were defined. */
MmodControl(MilSearchContext, M_DEFAULT, M_ACCEPTANCE, 40);
MmodControl(MilSearchContext, M_DEFAULT, M_CERTAINTY, 60);

/* Set the number of occurrences to 2 for all the models that were defined.*/
MmodControl(MilSearchContext, M_DEFAULT, M_NUMBER, 2);

#ifdef NUMBER_OF_MODELS>1
/* Change the reference point of the second model. */
MmodControl(MilSearchContext, 1, M_REFERENCE_X, MODEL1_REFERENCE_X);
MmodControl(MilSearchContext, 1, M_REFERENCE_Y, MODEL1_REFERENCE_Y);

```

(cont...)

```

    #if (NUMBER_OF_MODELS>2)
    /* Change the reference point of the third model. */
    MmodControl(MilSearchContext, 2, M_REFERENCE_X, MODEL2_REFERENCE_X);
    MmodControl(MilSearchContext, 2, M_REFERENCE_Y, MODEL2_REFERENCE_Y);
    #endif
    #endif

    /* Preprocess the search context. */
    MmodPreprocess(MilSearchContext, M_DEFAULT);

    /* Draw boxes and positions in the source image to show the models. */
    for (i=0; i<NUMBER_OF_MODELS; i++)
    {
        MgraColor(M_DEFAULT, ModelsDrawColor[i]);
        MmodDraw (M_DEFAULT, MilSearchContext, MilOverlayImage, M_DRAW_BOX+M_DRAW_POSITION,
                  i, M_ORIGINAL);
    }

    /* Pause to show the models. */
    printf("A model finder context was defined with the models in the");
    printf("displayed image.\n");
    printf("Press <Enter> to continue.\n");
    getchar();

    /* Clear the overlay image. */
    MbufClear(MilOverlayImage, TransparentColor);

    /* Load the complex target image. */
    MbufLoad(MULTI_MODELS_TARGET_IMAGE, MilImage);

    /* Dummy first find for better function timing accuracy (model cache effect,...). */
    MmodFind(MilSearchContext, MilImage, MilResult);

    /* Find the models. */
    MappTimer(M_TIMER_RESET, M_NULL);
    MmodFind(MilSearchContext, MilImage, MilResult);
    MappTimer(M_TIMER_READ, &Time);

    /* Get the number of models found. */
    MmodGetResult(MilResult, M_DEFAULT, M_NUMBER+M_TYPE_LONG, &NumResults);

    /* If the models were found above the acceptance threshold. */
    if ( (NumResults >= 1) && (NumResults <= MODELS_MAX_OCCURRENCES) )
    {
        /* Get the results the single model. */
        MmodGetResult(MilResult, M_DEFAULT, M_INDEX+M_TYPE_LONG, Models);
        MmodGetResult(MilResult, M_DEFAULT, M_POSITION_X, XPosition);
        MmodGetResult(MilResult, M_DEFAULT, M_POSITION_Y, YPosition);
        MmodGetResult(MilResult, M_DEFAULT, M_ANGLE, Angle);
        MmodGetResult(MilResult, M_DEFAULT, M_SCALE, Scale);
        MmodGetResult(MilResult, M_DEFAULT, M_SCORE, Score);
    }

```

(cont...)

```

/* Print information about the target image. */
printf("The model finder context was used to find the models");
printf(" in the target image.\n");
printf("This illustrates the following search capabilities:\n\n");
printf("  Full rotation\n  Small scale change\n");
printf("  Contrast variation\n  Specular reflection\n");
printf("  Occlusion\n  Multiple models\n  Multiple occurrences\n\n");

/* Print the results for the found models. */
printf("Result  Model  X Position  Y Position  Angle  Scale");
printf("  Score\n\n");
for (i=0; i<NumResults; i++)
{
    printf("%-9d%-8d%-13.2f%-13.2f%-8.2f%-8.2f%-5.2f%\n",
           i, Models[i], XPosition[i], YPosition[i], Angle[i], Scale[i],
           Score[i]);
}
printf("\nThe search time is %.1f ms\n\n", Time*1000.0);

/* Draw edges and positions over the occurrences that were found. */
for (i=0; i < NumResults; i++)
{
    MgraColor(M_DEFAULT, ModelsDrawColor[Models[i]]);
    MmodDraw(M_DEFAULT, MilResult, MilOverlayImage,
             M_DRAW_EDGES+M_DRAW_POSITION, i, M_DEFAULT);
}
}
else
{
    printf("The models were not found or the number of models found");
    printf(" is greater than\n");
    printf("the defined value of maximum occurrences !\n\n");
}

/* Wait for a key press to end. */
printf("Press <Enter> to end.\n");
getchar();

/* Free MIL objects. */
MbufFree(MilImage);
MmodFree(MilSearchContext);
MmodFree(MilResult);
}

```

**Chapter**

# 14

## **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, with a few 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**

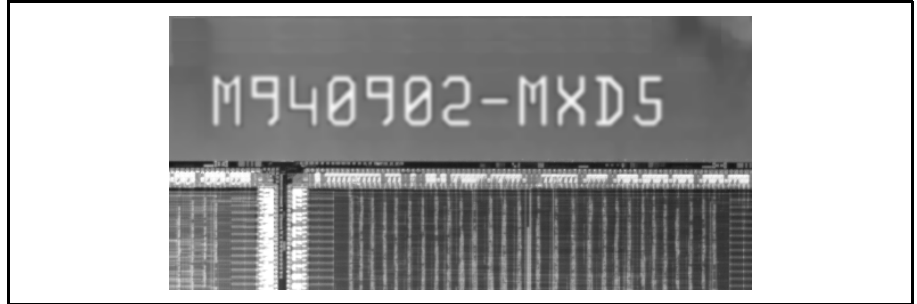
You can now read or verify the string in the target image using *MocrReadString()* or *MocrVerifyString()*. These operations are performed according to the defined character constraints and processing controls.

**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"); /* M only */
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, the font 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.

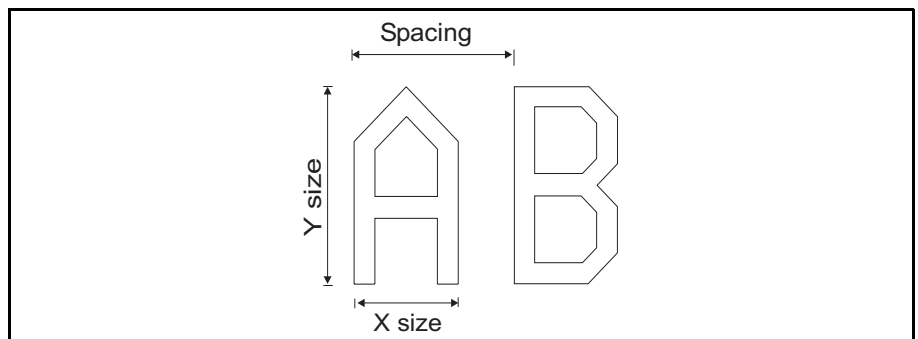
Generally, a font is calibrated once, at the beginning of an application. If your target image characters' size and/or spacing change, you should calibrate the font again. If the font is not calibrated, *MocrReadString()* or *MocrVerifyString()* might not be able to find the string in the target image 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 "semil292.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);
}

```

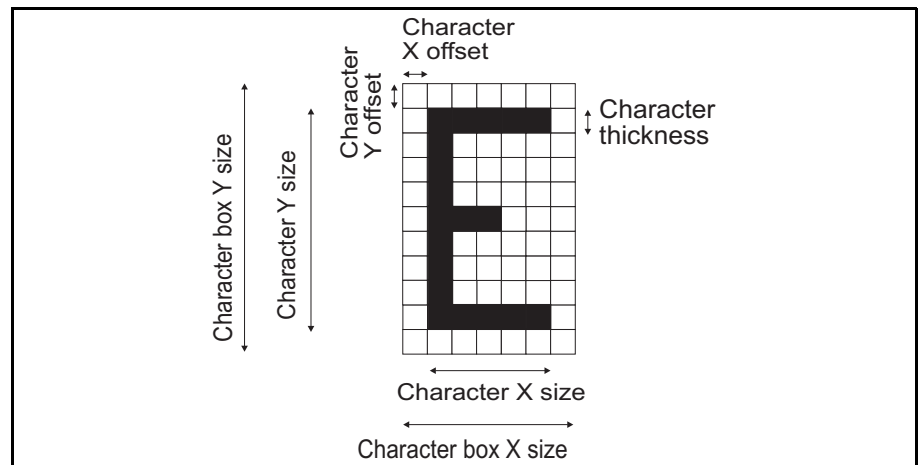
## Creating custom fonts

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.



The `CharBoxSizeX`, `CharBoxSizeY`, `CharOffsetX`, `CharOffsetY`, `CharSizeX`, and `CharSizeY` parameters must comply with the following restrictions:

- $2 * \text{CharOffsetX} + \text{CharSizeX} \leq \text{CharBoxSizeX}$
- $2 * \text{CharOffsetY} + \text{CharSizeY} \leq \text{CharBoxSizeY}$

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"

/* 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);

    (cont...)

```

```

/* 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);

/* 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);

```

(cont...)



```

/* 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**

# 15

## **1D and 2D code types**

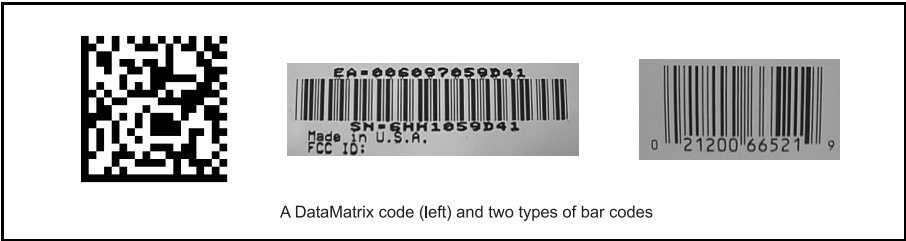
This chapter describes how to read and write 1D and 2D codes.

# Overview of codes

Many industries label products using symbols from symbologies such as bar codes and Data Matrix codes. This is done for identification purposes during different stages of production and distribution. Each *symbology*, known as a *code type* in MIL, follows a different set of rules to encode the data into light and dark patterns; data is encoded into units called *cells*.

MIL can both read symbols from and write symbols to images. To read a symbol, MIL searches for a specified code type in an image and decodes it. The decoded string can then be used to identify the object in the image. To write a symbol, MIL encodes a string into a symbol using the specified code type and encoding scheme. The resulting image of the symbol can then be rotated, combined with text on a logo, and then printed. In MIL, symbols are known as codes.

MIL supports both 1D and 2D code types. One-dimensional (1D) code types are linear bar codes, and are used for small amounts of data, for example, the identification number for a grocery store item, or a stock room part number (data is represented by bars and spaces). MIL also supports Data Matrix, Maxicode, and stacked bar codes, which are examples of two-dimensional (2D) code types. A Data Matrix code appears like a “checker board”, with light and dark cells. Maxicode appears like a series of interlocking hexagons that surround a central “bull’s eye”. Stacked bar codes, such as PDF417, are literally stacks of bar codes. Two-dimensional code types can contain much more data than 1D code types, for example, shipment information or medical records. Examples of symbols of 1D and 2D codes are illustrated below:



Some code types support several methods of encoding, known as *encoding schemes*. This means a code type might support an encoding scheme of alpha characters only, numeric characters only, or both alpha and numeric characters. In addition, some code types can support any number of characters, while others require a fixed number.

If the image is degraded, codes can still be read if an *error correction* scheme was used when the code was generated. Error correction is essentially redundant data included in the encoding scheme of some code types. Some error correction schemes are used only for error *detection*, while others are used for error *detection* and *recovery*. These terms are discussed in further detail in the section, *Supported code types and noteworthy characteristics*.

**Supported buffer types**

The code module only supports 8-bit unsigned buffers. Buffers in other formats will produce an error.

**More information**

For technical information about Data Matrix, PDF417, Maxicode, or bar codes, see the *AIM International Symbolology Specification - Data Matrix*, *AIM International Symbolology Specification - PDF417*, *AIM International Symbolology Specification - Maxicode*, or *The Bar Code Book*, Roger C. Palmer, *Helmets Publishing, United States*, 1995.

## Using MIL to read and write codes

---

The steps below describe, in general, how to perform read and write operations using MIL. Note that you must take into consideration any particularities of the chosen code type. More detailed and code type-dependent information is described in subsequent sections in this chapter, as well as the *MIL Command Reference* manual.

To perform a read or write operation:

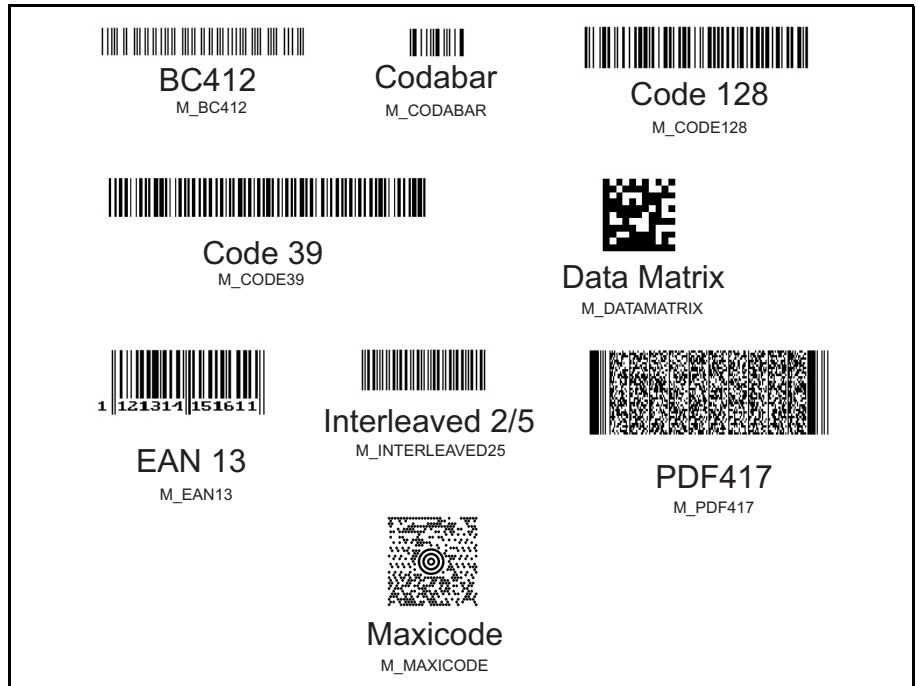
1. If reading a code, grab or load an image that contains the code into an allocated image buffer. If the image is large, or if there are multiple codes in the image, create a child buffer to isolate the code from the rest of the image. If your image contains other important information besides the code, you might want to use the pre-search feature using *McodeControl()*, which will help MIL to localize the code.

If writing a code, allocate the image buffer in which to generate the code. For more information on how to determine the required size of the image buffer, see the subsection, *Write operations - issues to consider*, of the *Controlling read and write operations* in MIL section.

2. Allocate a code object, using *McodeAlloc()*. A code object specifies the type of code to read or write and how to perform the operation.
3. If necessary, change control settings of the code object, using *McodeControl()*.
4. To perform a read operation, use *McodeRead()*. To perform a write operation, use *McodeWrite()*.
5. Retrieve results, using *McodeGetResult()*.
6. Free the memory allocated to the code object, using *McodeFree()*.

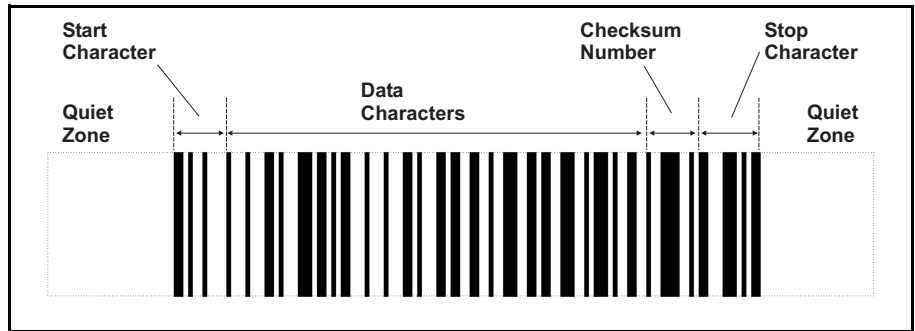
## Supported code types and noteworthy characteristics

Whether you plan on reading or writing codes, you must specify the code type when allocating the code object with *McodeAlloc()*. Below is a list of the supported code types and their MIL predefined constants:



### Anatomy of 1D and 2D codes

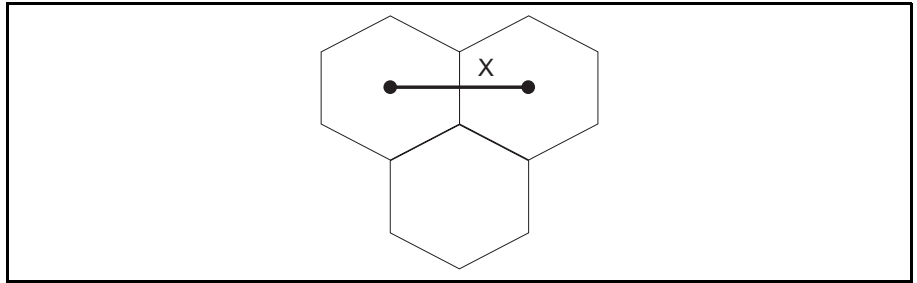
To perform more sophisticated read and write operations using MIL, it will be useful to understand the following terms that are illustrated in the example Code 128 code below.



- **Cell.** The narrowest dark or white space into which data is encoded. Cells have different shapes, depending on the symbology: cells in 1D code types are bars, some 2D codes have cells that are approximately-square blocks, while Maxicode cells are hexagonal. In 1D codes, thicker bars (and therefore spaces) occupy several contiguous cells.
- ❖ Industry uses slightly different terminology for PDF417 codes. For example, industry uses the term *modules* to refer to cells. There are 17 modules (cells) in each *column* of a PDF417 code, and each column has four bars and four spaces. Industry also refers to *codewords*; a codeword is a column within one row.
- **Cell size (also known as x-width).** Width, in pixels, of the cell, the code's narrowest unit.

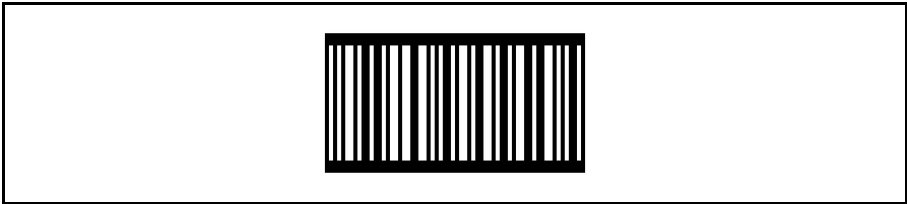


- ❖ The cell size of a Maxicode is defined as the distance between the center points of two hexagons in the same row.



- **Encoding scheme.** The method used to encode a string into a code. Not all code types support all encoding schemes, so you must be aware of the characteristics of your code type. The encoding schemes supported for a given code type are listed under *McodeControl()* in the *MIL Command Reference* manual.
- **Error correction.** Error correction schemes prevent read operations from returning incorrect results. During read operations, their purpose is for *error detection* and *error recovery*. Error detection discovers bit errors (when a 1 is read as a 0 or vice versa). Error recovery both detects and corrects bit errors; depending on the error correction scheme used, questionable data can be corrected based on the remaining data.
- **Checksum number.** Number included as part of a 1D or 2D code, which is calculated based on the other characters in the code; it is a simple form of error detection. Typically during a read operation, the characters that make up the string will be put into a checksum calculation whose results are compared against the checksum number. If a discrepancy exists between the calculated and stored checksum, the read operation will yield no results. Not all code types use a checksum.
- **Start and stop characters.** Characters which inform the reader or scanner of the beginning and end of a code, analogous to the start and stop bits which are specified when transferring data across a modem. Some code types, such as Code 39, use identical stop and start characters, while others use different stop and start characters. The PDF417 code type refers to these characters as *start and stop patterns*.

- *Quiet zone.* Quiet zones are areas with no marks and are used to aid the scanning process. For 1D codes, the quiet zone immediately precedes the start character and immediately follows the stop character. For 2D codes, the quiet zone must be present on all four sides of the code. The quiet zone must contain no marks; even hand-written scribbles in the quiet zone will impede the read operation significantly.
- *Bearer bars.* Bars that run along the tops and bottoms of a code, which are illustrated in the diagram below.



If you are reading a code with bearer bars, you must specify the search angle at which to read the code. MIL does not generate bar codes with bearer bars. If you want to generate a code with bearer bars, generate the required code and then draw a rectangle at the top and bottom of the code using *MgraRectFill()*; the minimum width of the bearer bar (height of the rectangle) must respect the specifications of your code type.

## Controlling read and write operations in MIL

This section provides information to consider and describes settings that you might have to change for both read and write operations.

### Issues pertaining to both read and write operations

Code type	You must know the code type when performing a read or write operation. If your read operation is unsuccessful, ensure that you have allocated the code object with <i>McodeAlloc()</i> using the correct code type.
Encoding scheme	MIL can automatically detect the type of encoding scheme for read operations for most code types, but it must be specified for write operations; use the <i>McodeControl()</i> <code>M_ENCODING</code> control type. The list of supported encoding schemes is found under <i>McodeControl()</i> in the <i>MIL Command Reference</i> manual.

Note the following when using this control type and some code types:

- You must specify the encoding scheme when reading a Code 39.

#### Error correction

In most cases, MIL can automatically detect the error correction scheme for read operations. Similarly, MIL can automatically choose the best error correction scheme for most write operations. If your code type supports more than one error correction scheme and does not support `M_ANY`, you must specify the error correction scheme. Specify error correction using the *McodeControl()* `M_ERROR_CORRECTION` control type. See *McodeControl()* in the *MIL Command Reference* to determine supported error correction schemes for your code type. If you specify a scheme that is not supported by your code type, you will get a MIL error.

#### Foreground color

During a read or write operation, you must specify the foreground color of the code; use the *McodeControl()* `M_FOREGROUND_VALUE` control type.

### Read operations - issues to consider

#### Search region

It is strongly recommended to use child buffers for read operations, especially if your image contains multiple codes of the same type and you want to read more than one, or if your image contains other information besides the code. MIL's code module is designed to read one code at a time, therefore if you do not use child buffers and your image contains multiple codes, results will be unpredictable.

A quiet zone is part of each code type's specification. During read operations, therefore, your image buffer must be large enough for your code to have a quiet zone. MIL can, in some cases, successfully read a code that does not meet the minimum requirements for the quiet zone; however, it is strongly recommended that codes have one in order to perform robust operations and return accurate results.

There are some situations where you are not able to create a child buffer for your read operation. Your image could be very complex and contain other information besides the code. In this case, you could use the code module's pre-search algorithm, which helps MIL to localize the code in the image prior to the read operation. The pre-search algorithm is only supported with 2D codes, and is disabled by default. This algorithm can also be used if your image contains several codes of different types. Note that if your image contains more than one code of the same type, MIL will randomly select only one to decode. Set the pre-search option by using the *McodeControl()* `M_USE_PRESEARCH` control type.

- ❖ It is highly recommended that when using this control type, you also specify the best cell size range possible, especially if the cell size is greater than 10 or smaller than 6.

### Cell size

In most cases, you should not have to specify the cell size when reading codes; the default setting (M\_DEFAULT) is sufficient. However, for some 2D code types, read operations can be more robust if you specify the cell size. Specify the cell size as a range using the *McodeControl()* M\_CELL\_SIZE\_MIN and M\_CELL\_SIZE\_MAX control types. MIL will search for codes with cells that fall within this range. If the cell size is not within the specified range, the code will not be found. Note that MIL might have difficulty reading codes if the cell size is less than 2 pixels, even if the size is specified.

Note the following when using these control types and some code types:

- Data Matrix and Maxicode read operations can be more robust if you specify the cell size.

### Number of cells

During read operations, MIL can automatically detect the number of cells, and will search for the specified 2D code type with any number of cells in the X and Y direction; the X direction representing the number of columns, and the Y direction representing the number of rows. You can specify the number of cells in the X and Y directions to increase the speed and robustness of the operation for 2D code types only. To specify the number of cells, use both *McodeControl()* M\_CELL\_NUMBER\_X and M\_CELL\_NUMBER\_Y control types.

Note the following when using these control types and some code types:

- Specifying the number of cells is mostly useful when reading PDF417 and Data Matrix code types; the Maxicode code type has a fixed number of cells.
- For the PDF417 code type,  $M\_CELL\_NUMBER\_X = 17c + 35$ , where  $c$  is the number of columns, and 35 represents the number of cells required for the start and stop patterns.

### Search angle

By default, MIL can read codes if they fall within the angular range of  $0 \pm 5^\circ$ . It is strongly recommended that you change the search angle if the code appears rotated in the image, and you should increase the angular range if you are unsure of the code's exact orientation. Note, however, that when searching for 1D codes, the operation speed is slower for angular ranges greater than  $\pm 5^\circ$ . The angular

range is the range of angles defined by the *McodeControl()* `M_SEARCH_ANGLE - M_SEARCH_ANGLE_DELTA_NEG` and `M_SEARCH_ANGLE + M_SEARCH_ANGLE_DELTA_POS` control types.

Note the following when using these control types and some code types:

- If the code has bearer bars, you must specify the search angle.
- For the PDF417, Maxicode, and Data Matrix codes, the angular range does not affect the speed of the operation.

### Search speed

You can specify the speed at which to perform a read operation; the faster the 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 speed (`M_MEDIUM`). Specify the search speed using the *McodeControl()* `M_SPEED` control type. 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 automatically chosen and is suitable in most cases. However, if you think that a different value would result in a better separation (and therefore in a more efficient operation), you can manually adjust the threshold level by setting the *McodeControl()* `M_THRESHOLD` control type to another value.

- ❖ 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.

### String size

By default, the operation searches for a code that encodes a string of any size. However, if you know the exact size of the string, you might want to specify this value to increase the robustness of the operation using the *McodeControl()* `M_STRING_SIZE` control type.

Note the following when using this control type and some code types:

- You must specify the string size when reading the BC412 code type.

### Write operations - issues to consider

It is strongly recommended that you define your codes as clearly as possible, as it will facilitate reading them later on. When writing a code ensure that you are familiar with the requirements and restrictions of your chosen code type, otherwise the code will not be generated; a MIL error will result.

#### Size of destination image

The destination image of the write operation should be large enough to hold the encoded string. For a given code type, cell size, 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()*.

Note that some code types have specific requirements for the encoded string:

- The encoded string for a Codabar code type must be numeric, but can contain the following ASCII marks: -, \$, :, /, ., and +. In addition, the string must start and end with any of the following characters: a, b, c, or d.
- The Interleaved 2 of 5 code type requires a string that has an even number of characters.
- The encoded string for a Maxicode code type with the `McodeControl()` `M_ENC_MODE2` or `M_ENC_MODE3` control type, must respect the structured carrier message format (the portion of the string that contains postal code, country code, and class of service information).

#### Encoding non-printing characters

If you need to encode non-printing characters such as a carriage return or tab, use a code type that supports them. If you set the *McodeWrite()* control flag to `M_ESCAPE_SEQUENCE`, you can use the ASCII codes to encode all non-printing characters. In the string to encode, the non-printing character's ASCII code must be in hexadecimal format and be preceded by the `\x`, for example, `\x0D` for ASCII 13, which is the carriage return.

- ❖ If you want to encode the “\” character in escape sequence mode, type “\\”.

#### Cell size

In most cases, you should not have to specify the cell size when writing codes; the default setting (*McodeControl()* `M_CELL_SIZE_MIN` set to `M_DEFAULT`) is sufficient. In this case, the code is resized so as to just fit into the destination image of the operation, when possible. During a write operation, you can use the *McodeControl()* `M_CELL_SIZE_MIN` control type to force the cell size; the size is

used to determine the size of the generated code. If you specify the required cell size with *McodeControl()*, you should ensure that the destination image has an appropriate size. See the discussion earlier in this subsection.

❖ The `M_CELL_SIZE_MAX` control type is not used in write operations.

#### Number of cells

During a write operation, the number of cells will be automatically chosen to minimize the code written. To specify the number of cells, use both *McodeControl()* `M_CELL_NUMBER_X` and `M_CELL_NUMBER_Y` control types. If you specify more cells than are necessary to generate the code, fill characters will automatically be added. If you specify fewer cells than are necessary to generate the code, you will get a MIL error.

Note the following when using these control types and some code types:

- Specifying the number of cells is mostly useful when writing PDF417 and Data Matrix code types; the Maxicode code type has a fixed number of cells.
- Note that for the PDF417 code type,  $M\_CELL\_NUMBER\_X = 17c + 35$ , where  $c$  is the number of columns, and 35 represents the number of cells required for the start and stop patterns.

## Retrieving results

---

You can retrieve results by calling *McodeGetResult()* with the control type for the result that you want returned.

You can retrieve the string or the length of the string in escape sequence format. Using the `M_ESCAPE_SEQUENCE` flag with `M_STRING` will return the string showing the ASCII codes for non-printing characters. Using the `M_ESCAPE_SEQUENCE` flag with `M_STRING_SIZE` will return the length of the encoded string, including all ASCII characters used to encode all non-printing characters in the count.

- ❖ The ASCII codes are returned in hexadecimal format.



**Chapter**

# 16

## **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*).

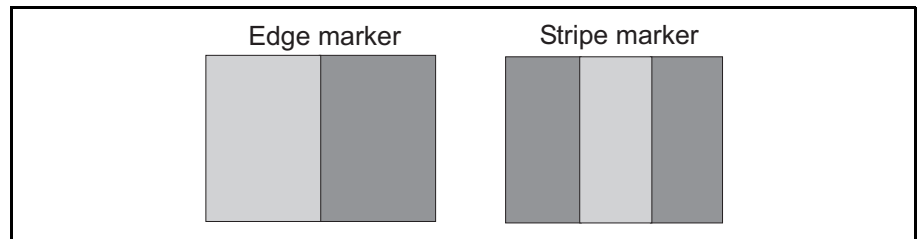
This chapter discusses finding and obtaining measurements of markers, how to define and set marker characteristics, and then the steps that are generally followed when taking measurements between markers.

## 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 characteristic 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.

### Viewing the marker

You can use *MmeasDraw()* to draw specific features of your marker as a means of verifying your marker settings. For example, you can draw the measurement box, the position, possible position variation, width variation, and edge of the expected marker. For stripe markers, you can draw both the first and second edges. For multiple markers, you also draw the expected spacing. You can use a previously allocated graphics context (see Ch. 19 *Generating graphics*) to control the drawing color, or use the default graphics context (M\_DEFAULT). You can draw directly into the image buffer, or annotate the image non-destructively by drawing into its display overlay buffer (see Ch. 18 *Annotating the displayed image non-destructively*).

### 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.

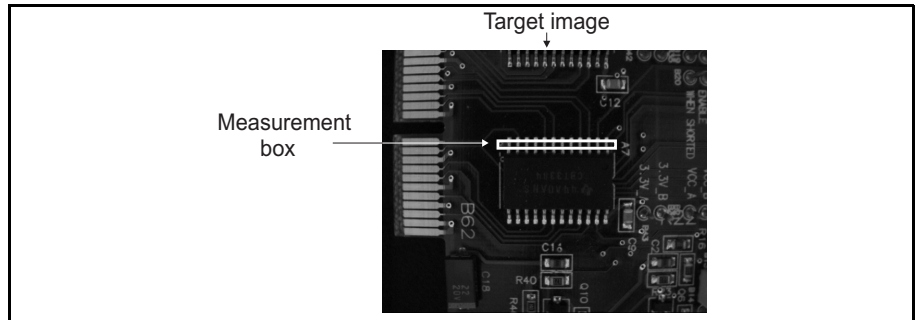
For a multiple marker, the *MmeasFindMarker()* function takes the required measurements for all the located edges or stripes. The number of edges or stripes found can be obtained using `M_NUMBER` as the result type with the *MmeasGetResult()* function. Global results, such as the maximum, minimum, mean, or standard deviation of any characteristic of the result group can be returned.

### Annotating results

In addition to the *Mgra...()* functions available for annotating your results, the *MmeasDraw()* function provides several additional operations for drawing features of the marker occurrences. For example, you can draw the edge profile of the occurrence, a cross at the found position, the width of the marker, among other features.

## 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

(cont...)

```

```

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

/* Main application function */
void main(void)
{
    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);
}

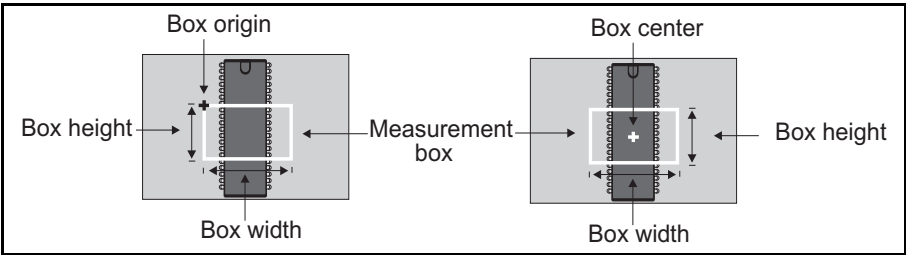
/* Draw a cross at the specified position. */
void DrawCross(MIL_ID ImageId, double CenterX, double CenterY, long Color)
{
    MgraColor(M_DEFAULT, Color);
    MgraLine(M_DEFAULT, ImageId, (long)(CenterX+0.5)-(CROSS_SIZE/2), (long)(CenterY+0.5),
              (long)(CenterX+0.5)+(CROSS_SIZE/2), (long)(CenterY+0.5));
    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.

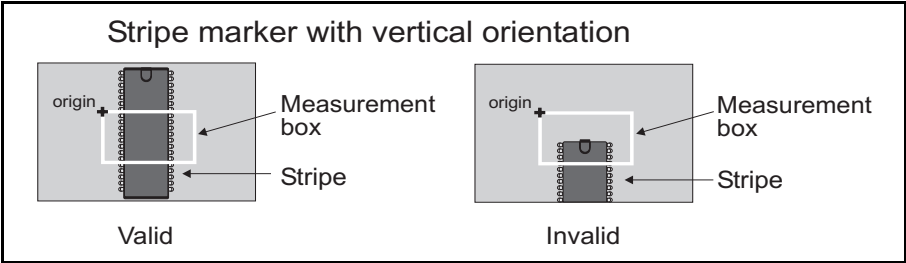
There are two ways to specify the measurement box. To specify the box's position, you can set either the origin (M\_BOX\_ORIGIN) or the center (M\_BOX\_CENTER) coordinates. You must also set the box's width and height (M\_BOX\_SIZE).



Note that the origin is always the top-left corner of the unrotated box and all width and height values are positive.

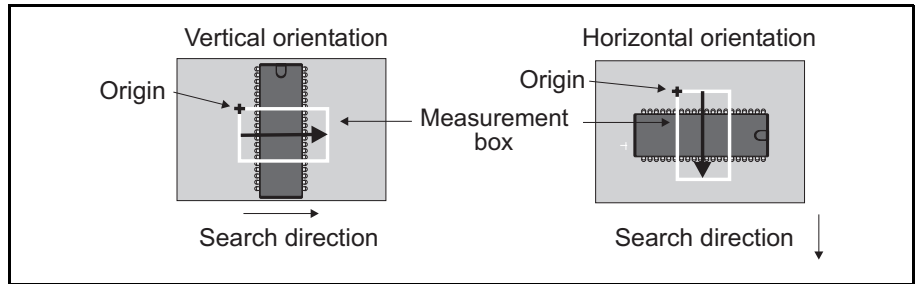
## Obtaining valid results

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.



## Orientation

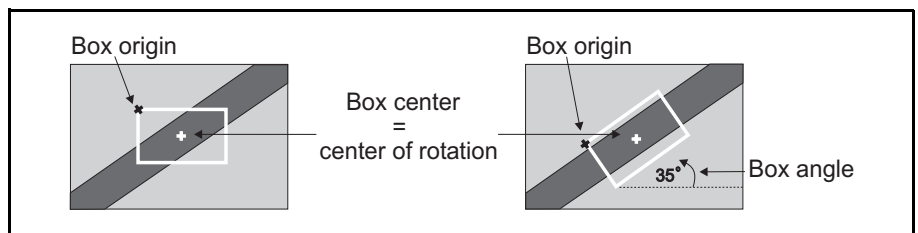
The orientation (M\_ORIENTATION) specifies the angle of the edge or stripe in relation to that of the measurement box and can be set to either M\_VERTICAL or M\_HORIZONTAL. More importantly, the orientation determines the direction in which the search will proceed.



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.

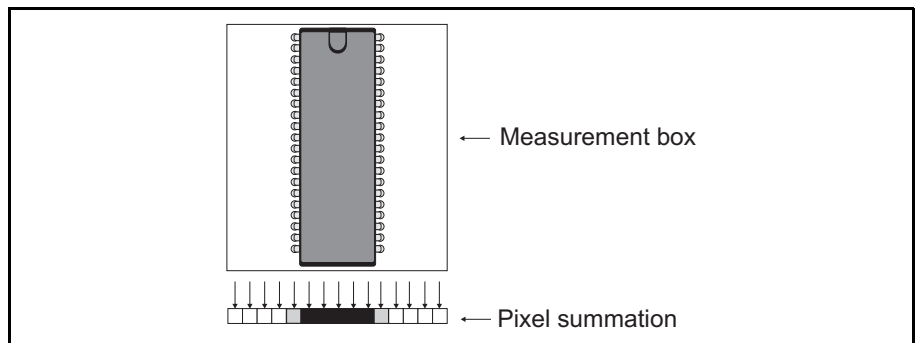
5. Take the minimum of the absolute value of these angles. Double this angle and set it as the rotation tolerance for the angular search.

### 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*.

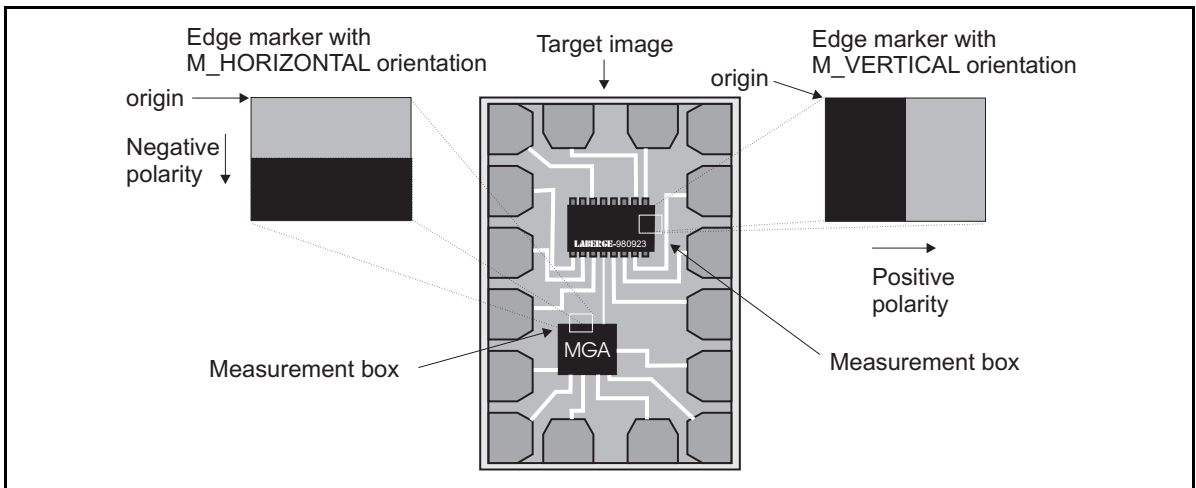
## Edge markers: fundamental characteristics

This section describes the fundamental characteristics of edge markers that are set using *MmeasSetMarker()* or returned as measurement results.

### Polarity of an edge

#### Polarity

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

#### Position and position variation

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**

**Contrast and contrast variation**

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.



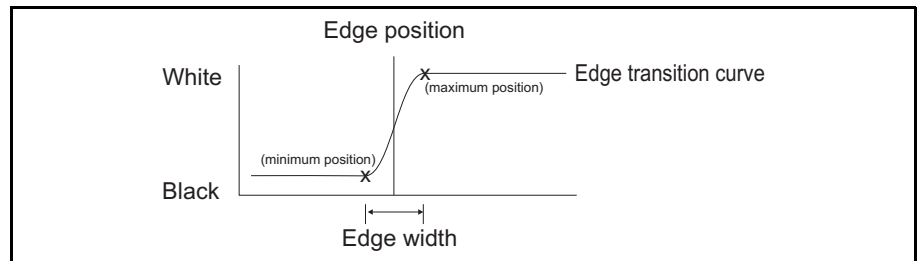
## Edge markers: advanced characteristics

This section describes the advanced characteristics of edge markers that are set using *MmeasSetMarker()* or returned as measurement results.

### Width of an edge marker

#### Width

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

The minimum and maximum positions (M\_POSITION\_MIN and M\_POSITION\_MAX) indicate the start and end positions of the edge of the occurrence. The minimum and maximum positions are not limited by the measurement box, but by the size of the target image. The minimum position is always closest to the target image origin, regardless of how the measurement box is rotated.

The minimum and maximum positions of an edge cannot be set but can be found with *MmeasGetResult()*.

Edge strength

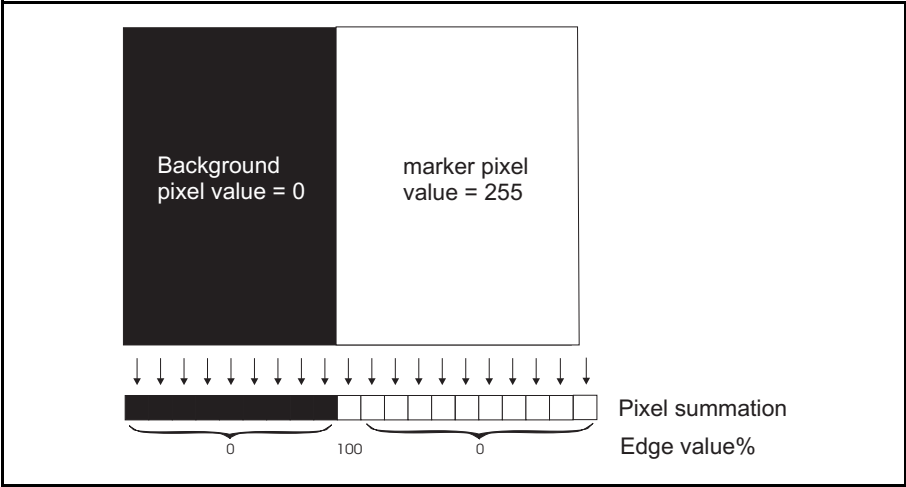
A marker's edge strength (M\_EDGE\_STRENGTH) is the minimum/maximum edge value along the width of the edge (depending on the polarity of the edge). The edge value is represented as a normalized percentage of the maximum pixel value possible for the specific image buffer. The sign of the edge value represents the polarity of the edge. For example, for a measurement box with a vertical orientation, the equation for an edge value is:

$$\text{edge value\%} = \frac{\Delta \text{ adjacent profile values}}{(\text{box height}) (2^{\text{buffer depth}} - 1)} * 100$$

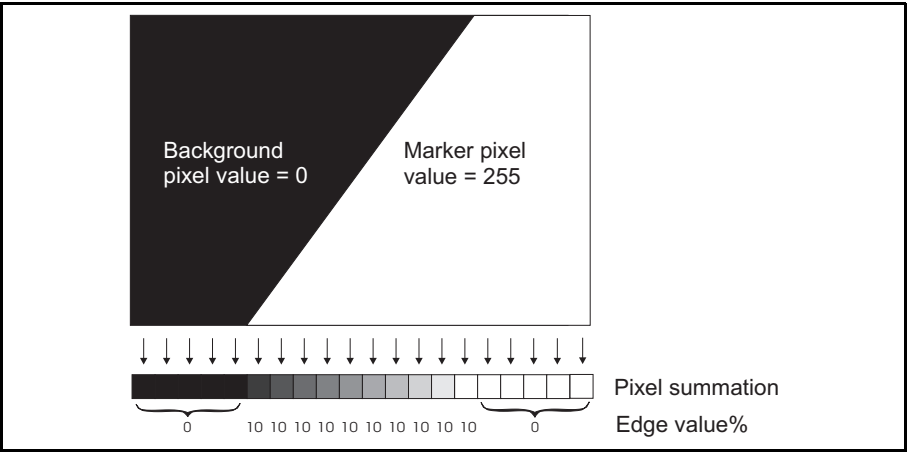
For instance, in an 8-bit image buffer the maximum possible pixel value is 255. Therefore, a 50% edge strength in this buffer represents a maximum difference in average adjacent profile values of 128 and a rising edge. The larger the absolute edge value, the greater the edge strength. The default setting 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%:



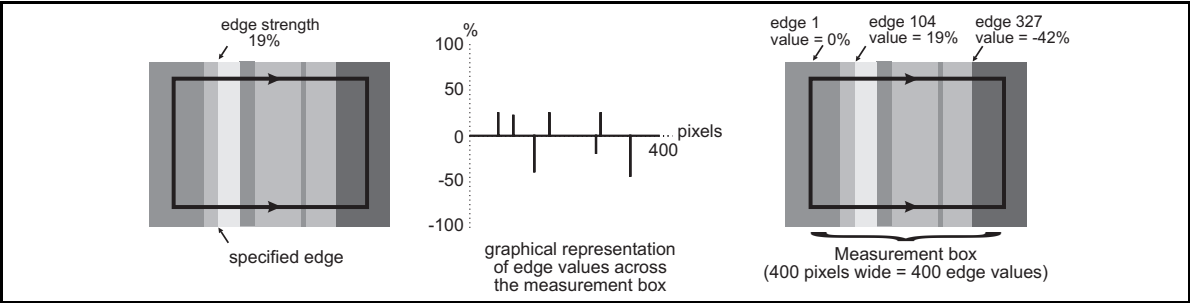
When the edge strength is not very high, you should specify the contrast. By using the contrast characteristic, changes in consecutive edge values are summed, from where the edge starts and ceases (that is, from one region of zero edge value to another), allowing the marker to be located. In general, 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

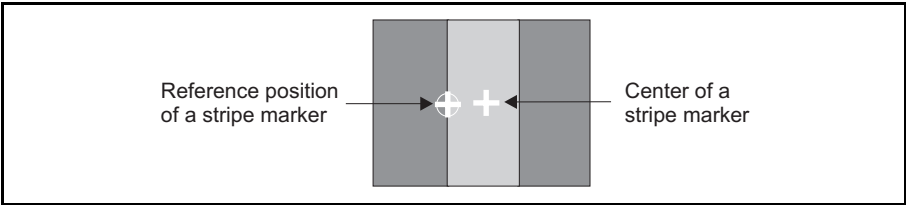
To determine the edge strength of the required marker, use `M_BOX_EDGE_VALUES`. THIS CALCULATES the edge values for every profile value of the measurement box. The illustration below shows a specific edge strength measurement, a graphical representation of box edge value measurement, and a sampling of 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

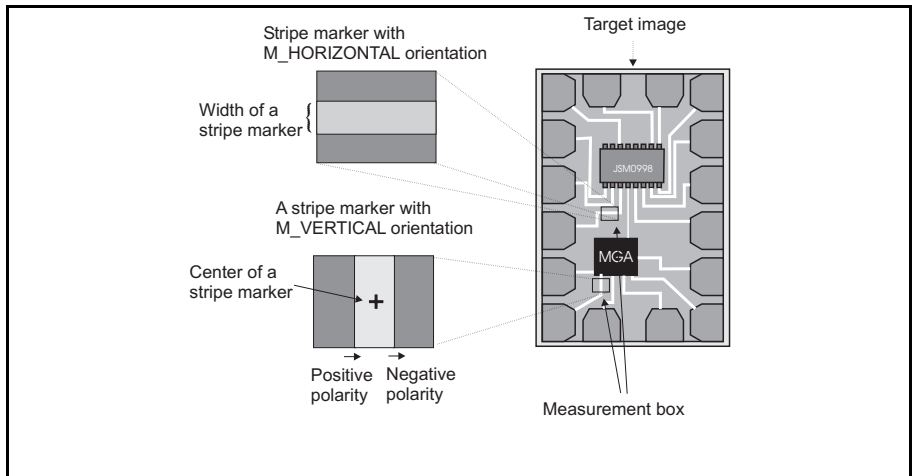
When searching for a marker, the relative importance (weight) assigned to each of the marker characteristics is crucial to the robustness of the operation. By default, 50% of the search weight is assigned to the edge strength; the remaining 50% is equally divided among all characteristics that can have 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

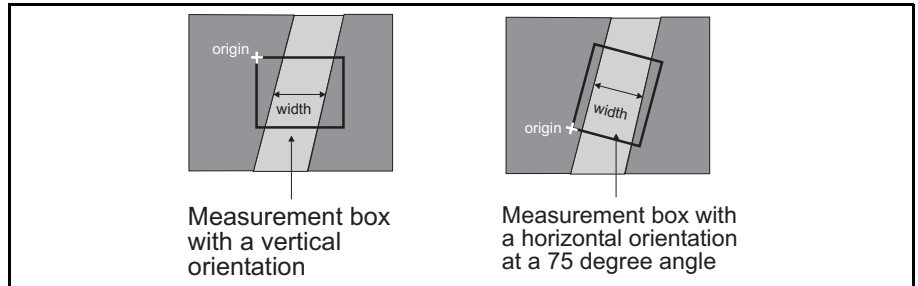
The contrast for a stripe marker requires two values, one for each of the stripe's edges. The contrast of the second edge can be set to M\_SAME if both edges of the stripe have approximately the same contrast. Both values can be set to M\_ANY if the contrast is unknown (default). The contrast variation 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.



## 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).

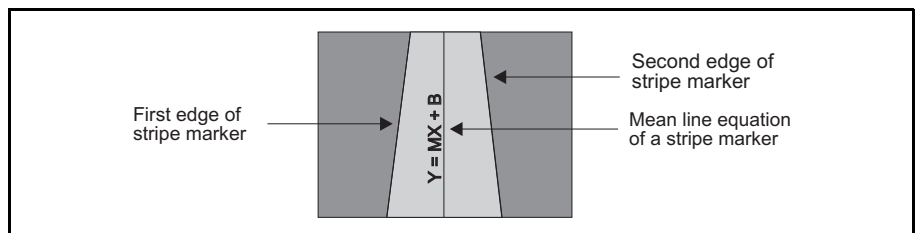
## Length

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.



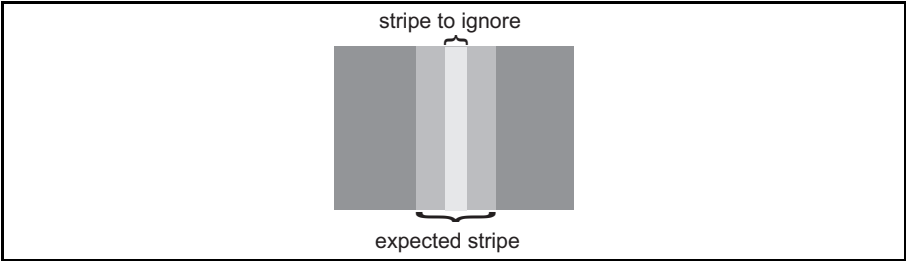
Note, the line equation of a stripe marker cannot be set, but can be returned with *MmeasGetResult()*. The slope of the line equation (M\_LINE\_EQUATION\_SLOPE) and the Y intercept of the line equation (M\_LINE\_EQUATION\_INTERCEPT) can also be returned separately.

## Stripe markers: advanced characteristics

Inside edge

**Inside edge and inside-edge variation**

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

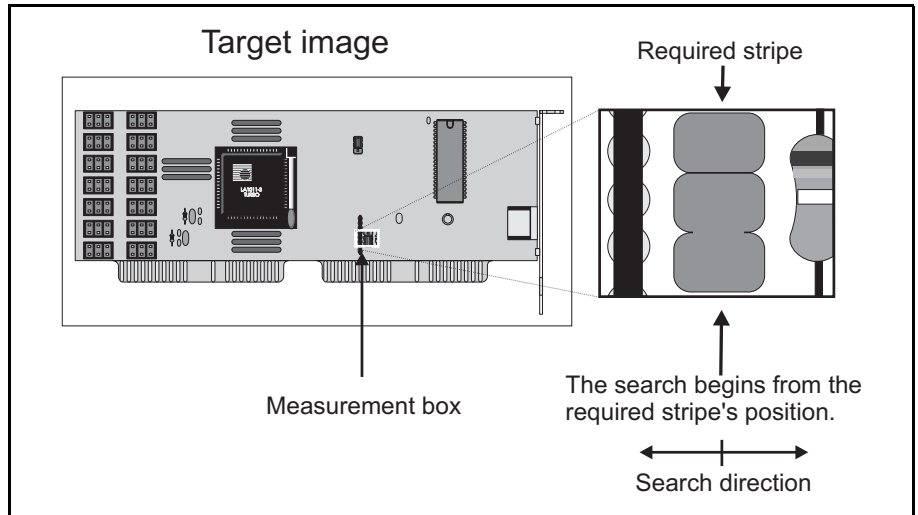
You can also specify the tolerance in the number of inside edges of a stripe (M\_EDGE\_INSIDE\_VARIATION). Note, this tolerance should be in increments of two if stripes are contained within stripes, since two edges are recognized for each stripe.

**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.



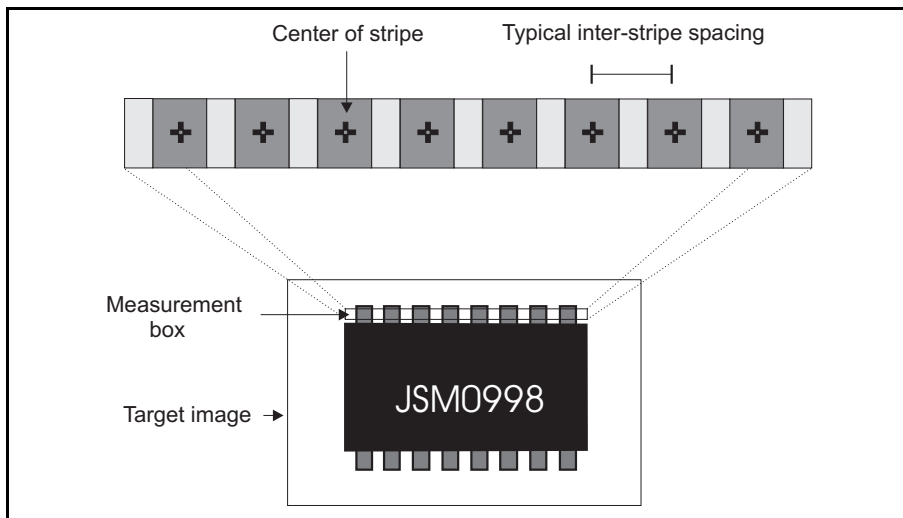
M\_POSITION\_INSIDE\_STRIPE is useful when the required stripe does not have the greatest edge strength or when it is difficult to set the measurement box without clipping other similar image characteristics. If defined as outside, no stripe including the position is considered.



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 edge or stripe marker, the marker edge or stripe characteristics are set just as with any marker; only a few additional *MmeasSetMarker()* characteristics need to be set. In addition, the `M_POSITION` control type and its weight factor are 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 specified 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.

- ❖ Note that the `M_SAME` SETTING is recommended only when several occurrences are expected; if the number of occurrences is too few, matches for width or spacing might not correspond to those of the required stripes.

A multiple point marker can also be specified for performing calculations between markers. A multiple point marker must have its initial position set using *MmeasSetMarker()* with `M_POSITION`. The spacing between these points must also be set using *MmeasSetMarker()* with `M_SPACING`. A multiple point marker can only be defined to use the same spacing between points; if irregularly spaced points are needed, you must specify each one individually as a point marker.

## 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()*.

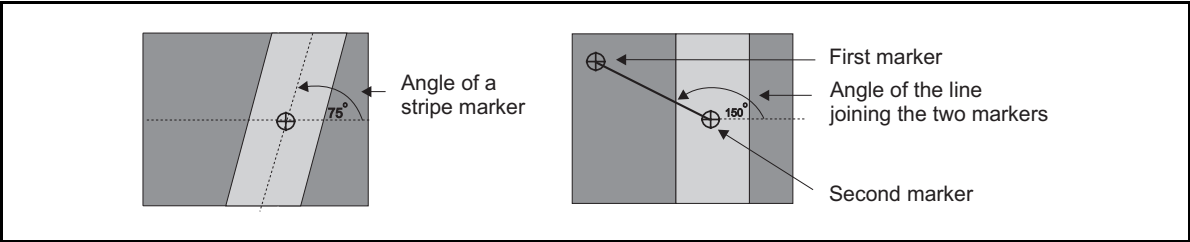
### Calculating with multiple markers

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.

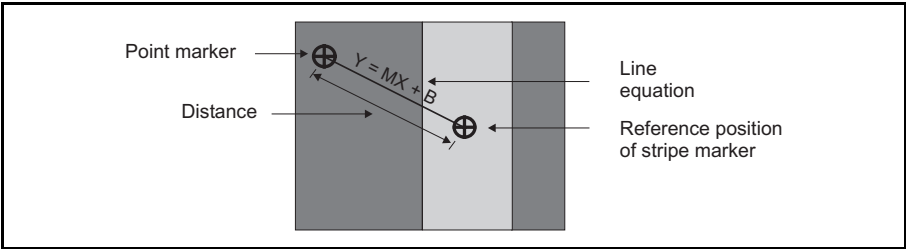


The angle is measured in a counter-clockwise direction relative to the positive x-axis, and can be a value from 0 to 360 degrees.

Line equation and distance

The line equation and distance can be calculated for the line joining two markers (see diagram below).

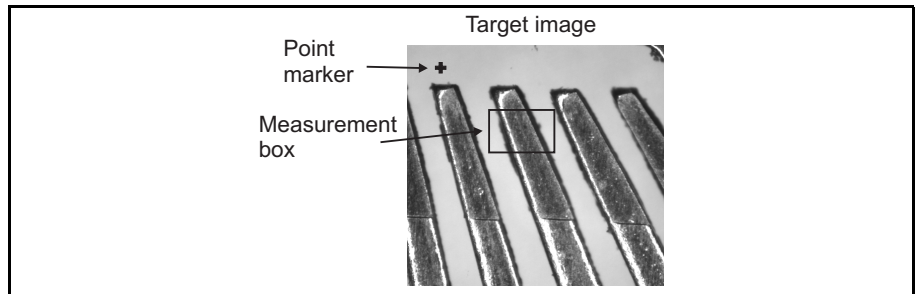
The distance between two markers is the distance between both of their reference positions, as illustrated in the diagram below.



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...)

```

```

/* Size and color of the cross to mark the positions. */
#define CROSS_SIZE          10L
#define CROSS_COLOR        240L
#define MARK_COLOR          0L

/* Utility functions prototype */
void DrawCross(MIL_ID ImageId, double CenterX, double CenterY, long Color);
void DrawMark(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. */
          StripeMarker,   /* Stripe marker identifier. */
          PointMarker,    /* Point marker identifier. */
          CalcResult;     /* Result buffer identifier. */
    double StripeCenterX, /* Stripe X center position. */
          StripeCenterY, /* Stripe Y center position. */
          StripeWidth,   /* Stripe width. */
          StripeAngle,   /* Stripe angle. */
          StripeFirstEdgeX, /* Stripe left edge X position. */
          StripeFirstEdgeY, /* Stripe left edge Y position. */
          StripeSecondEdgeX, /* Stripe right edge X position. */
          StripeSecondEdgeY, /* Stripe right edge Y position. */
          ReferenceDistance, /* Reference to stripe distance. */
          ReferenceAngle;    /* Reference to stripe angle. */
    long Box00X, Box00Y, /* Location box top left position. */
         Box10X, Box10Y, /* Location box top right position. */
         Box01X, Box01Y, /* Location box bottom left position. */
         Box11X, Box11Y; /* Location box bottom right position. */

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

    /* Restore the source image */
    MbufRestore(IMAGE_FILE, MilSystem, &MilImage);
    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 determines the position, width and angle of the\n");
    printf("stripe in the highlighted box and marks its center and edges.\n");
    printf("It also determines the length and angle of a line going from\n");
    printf("a reference point in the image to the center of the stripe.\n\n");
    printf("Press <Enter> to continue.\n");
    getchar();
}

```

(cont...)

```

/* Read the source image again to remove previously drawn rectangle */
MbufLoad(IMAGE_FILE, MilImage);

/* 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);
MgraLine(M_DEFAULT, MilImage, Box00X, Box00Y, Box10X, Box10Y);
MgraLine(M_DEFAULT, MilImage, Box10X, Box10Y, Box11X, Box11Y);
MgraLine(M_DEFAULT, MilImage, Box11X, Box11Y, Box01X, Box01Y);
MgraLine(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);

/* 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();

```

(cont...)

```

/* 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);
}

/* 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));
}

(cont...)

```



```
/* 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**

# 17

## **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. It shows you how to grab images with a Bayer camera and restore the color information.

## 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).

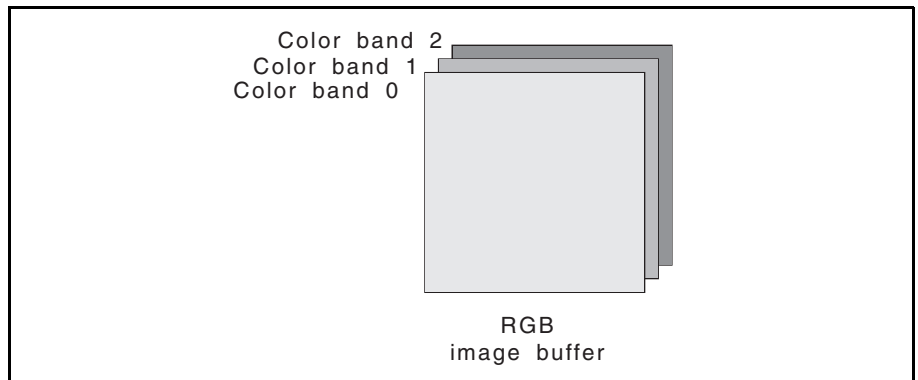
In addition, any operation involving one or more buffers will be performed by the most appropriate system that is associated with one of the buffers. By default, if none of these systems is more appropriate than the Host, the Host is used to perform the operation.

## 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 22: 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 (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 is automatically updated. When displaying a buffer, both the buffer and the display should be 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 and then at the end of the grab, the Host buffer is updated.

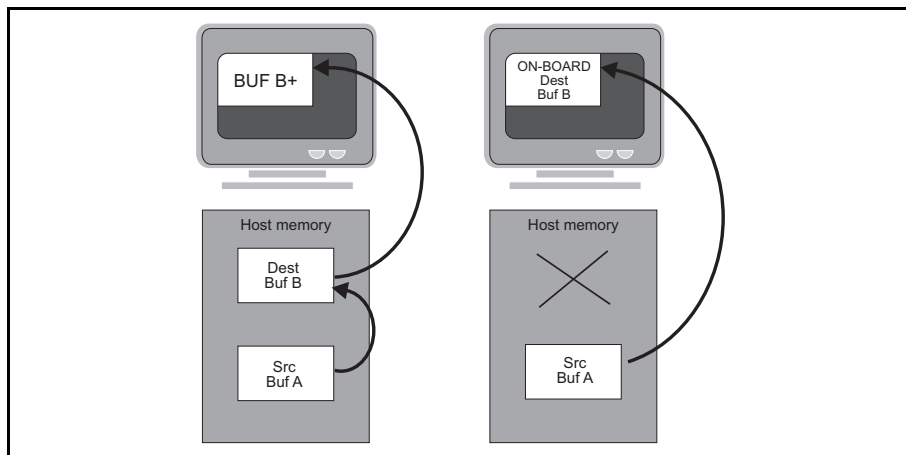
### **Overriding the default allocation sequence**

On boards with a display section, you can override the default buffer allocation sequence and force allocation only in the frame buffer using the *MbufAlloc...*() `M_ON_BOARD` attribute. In general, the buffer is allocated in the non-displayable area of the frame buffer. When the `M_DISP` attribute is specified for auxiliary displays, the buffer will be in the displayable area. 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.

- ❖ If you need to allocate an on-board image buffer, it is important to note that, since MIL selects which device will be used to display your image, you should only allocate this buffer (*MbufAlloc()*) after allocating the display to which it will be selected (*MdispAlloc()*).

Overriding the default allocation sequence is useful when allocating a displayable buffer for any auxiliary display. 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, for auxiliary displays, 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 is signed and the destination depth is greater than 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.

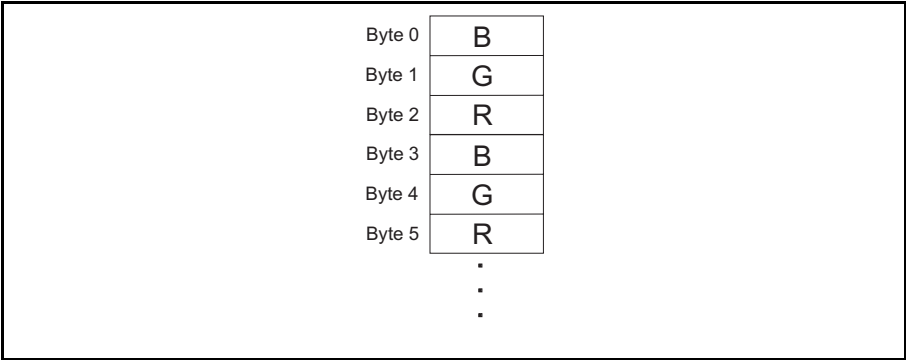
See the *MIL/MIL-Lite Board-Specific Notes* manual to determine which formats are supported on your board.

## RGB buffers

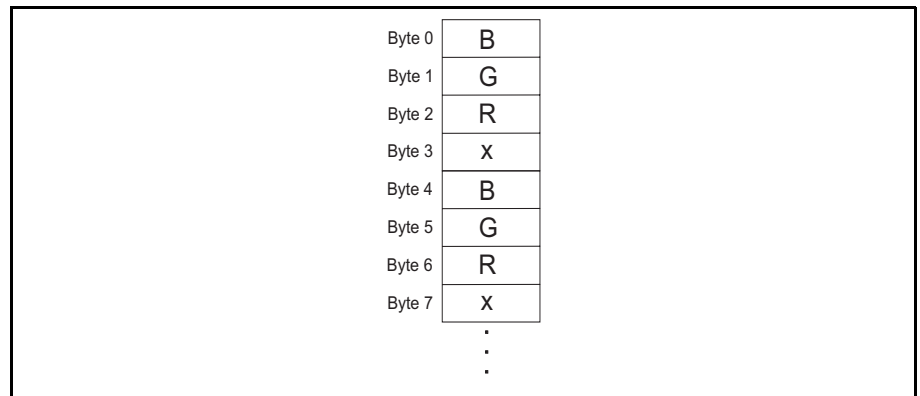
By default, MIL allocates color image buffers in an RGB color format. The pixels are internally stored in little-endian order, that is, they are stored in memory from their least-significant to the most significant bytes. The definitions of the RGB formats that are available are shown here. The corresponding MIL constant is shown in brackets beside the common format name.

RGB data formats

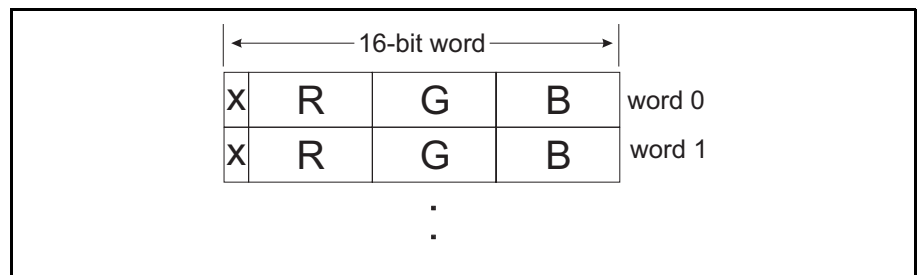
BGR24 packed (M\_BGR24+M\_PACKED) is a format whereby each pixel is internally stored as three consecutive bytes in little-endian order, that is:



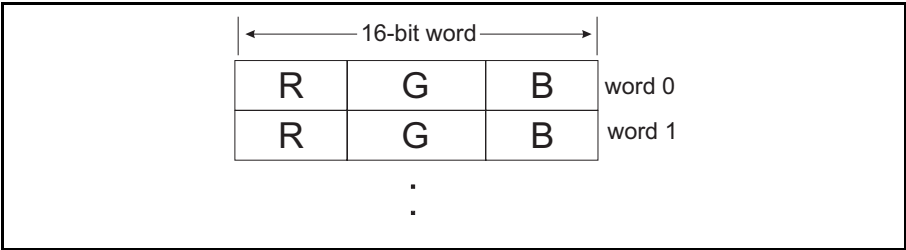
**BGR32 packed** (`M_BGR32+M_PACKED`) is a format whereby each pixel is internally stored as four consecutive bytes, in little-endian order. The most-significant byte is a "don't care" byte, as shown below:



**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

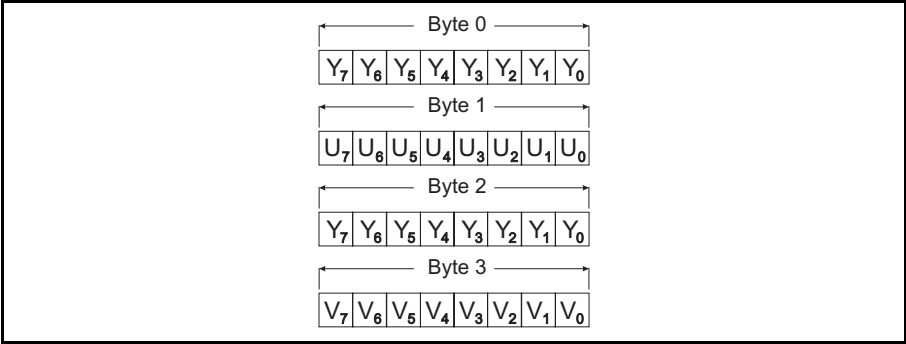
### **YUV16 Packed**

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.

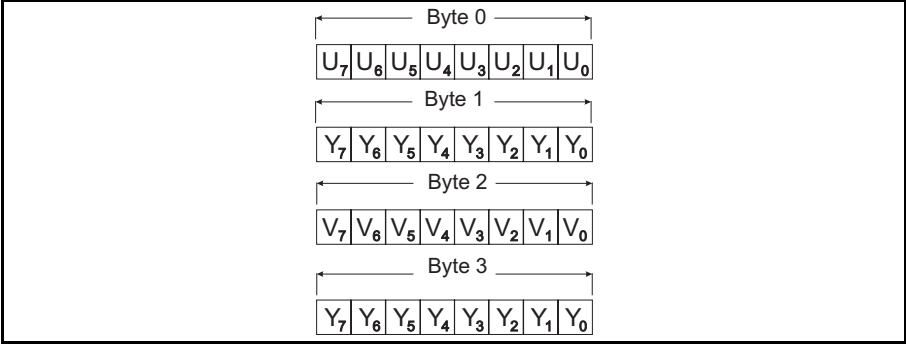
The YUV16 packed data format has two available formats: YUYV and UYVY. The only difference between these two YUV formats is the ordering of data in the buffer. Certain digitizer boards grab data in exclusively YUYV or UYVY packed data format. Note that, for display, certain operations are optimized to handle the YUYV format; for example, displaying a decompressed buffer.

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.

YUYV

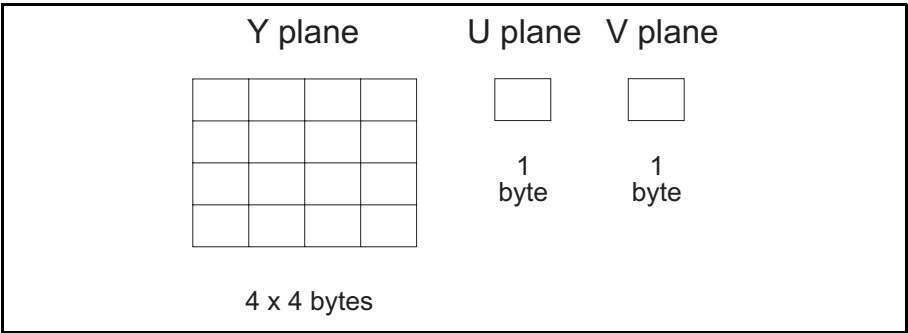


UYVY



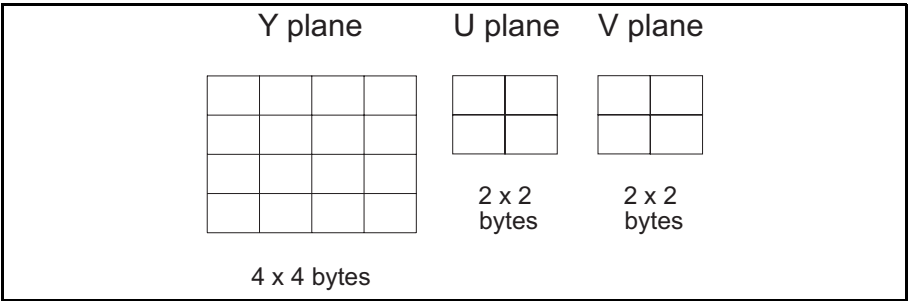
YUV9 Planar

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.



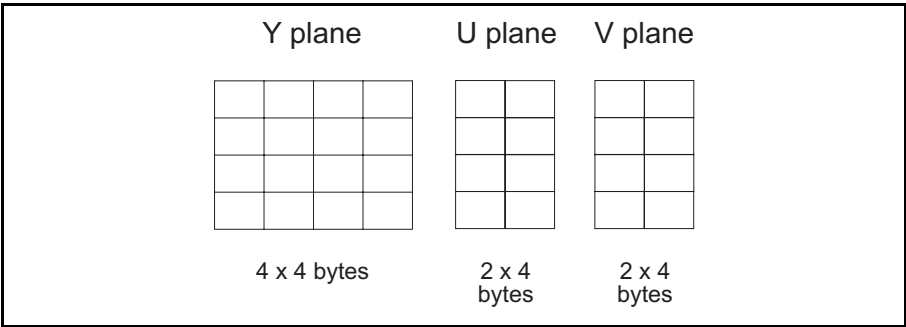
**YUV12 Planar**

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.



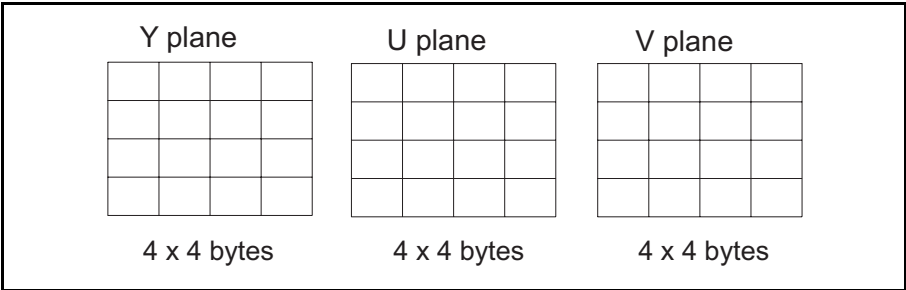
**YUV16 Planar**

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.



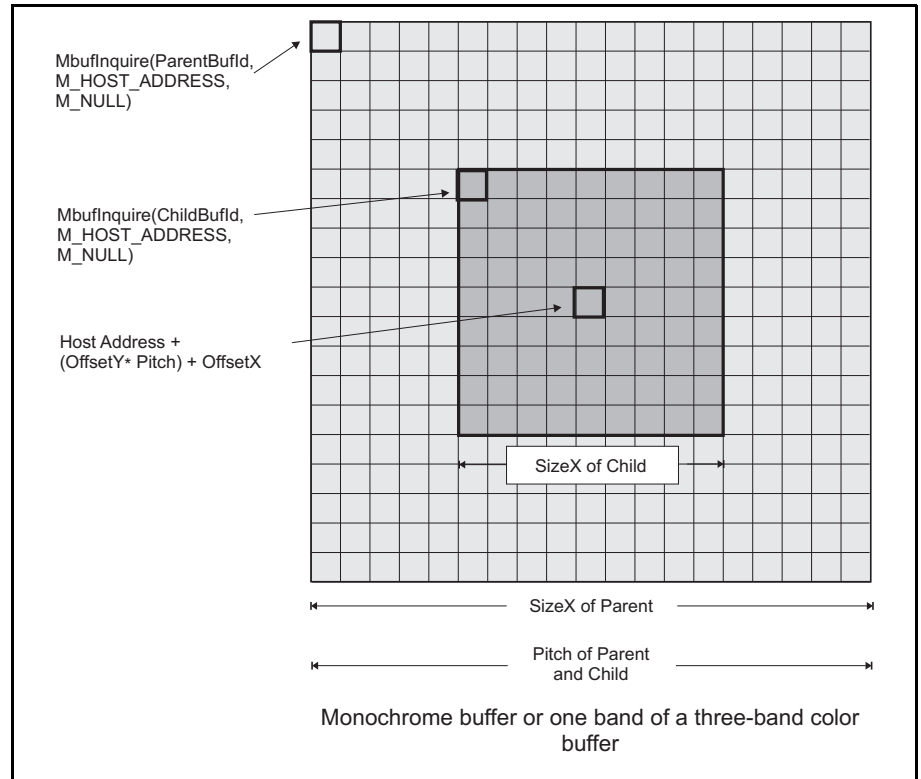
**Child YUV buffers**

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 <i>MbufInquire()</i> . Selecting <code>M_HOST_ADDRESS</code> will return a logical address, while <code>M_PHYSICAL_ADDRESS</code> 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.

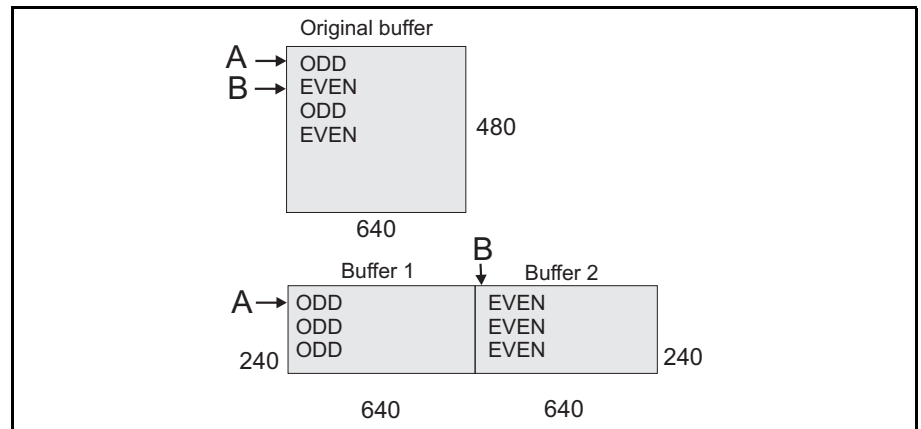
When creating a buffer with *MbufCreateColor()*, you must pass an array of pointers to the addresses of the data. For packed color buffers, you must pass an array of one pointer; for planar buffers, you must pass an array with the same number of pointers as the number of bands in the buffer. When creating 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` inquire 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 of the displayable buffer (Buffer 2).

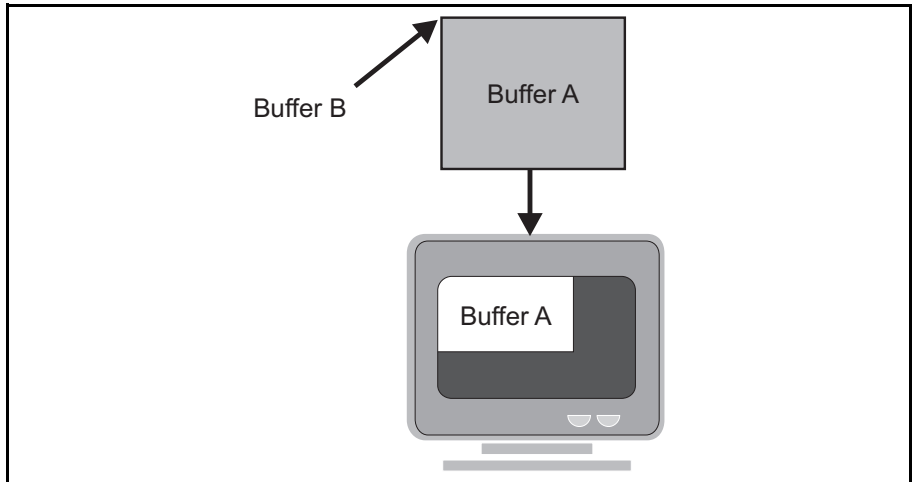
Create Buffers 1 and 2 as follows:



- 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 24: Data Manipulation with multiple systems* for another instance where creating buffers is useful.



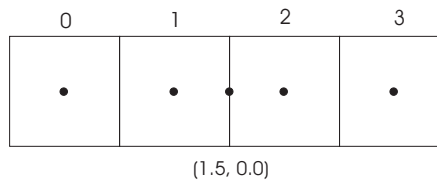
## Pixel conventions

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:

$$\left(\frac{width-1}{2}, \frac{height-1}{2}\right)$$

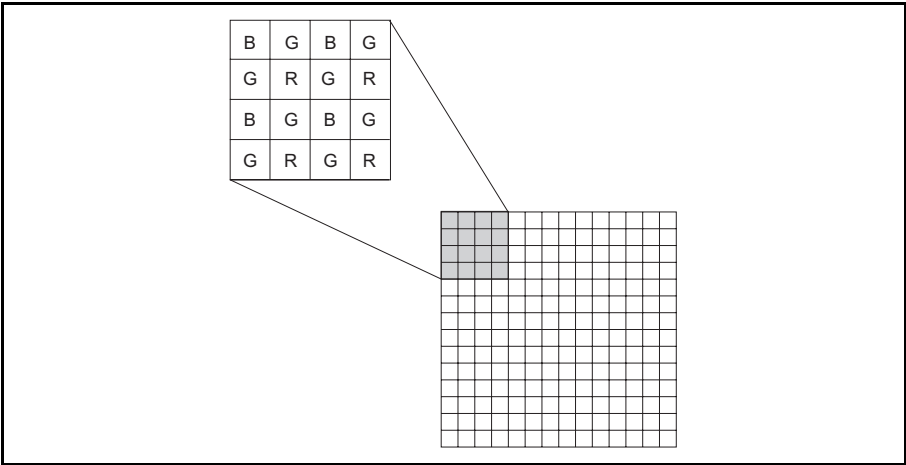
For example, the following image contains 4 pixels. If the formula is applied, the center of the image is found at (1.5, 0).



## Using buffers with the Bayer color filter

Cameras that feature a Bayer color filter can be used with MIL to provide a cost-effective method for grabbing color images: the camera grabs a single-band color-encoded image, and then MIL converts it to a multi-band color image, using the *MbufBayer()* function. Bayer images are distinct from standard single-band images because of the color information contained in their pixels, which is extracted by the *MbufBayer()* function.

When grabbing from these cameras, each pixel quantifies only one of the color components of the image in the camera’s field of view at the corresponding location. Within a group of 2x2 pixels, there are two pixels containing color information for the green component, and one pixel for each the red and blue components; Bayer images contain more green pixels because the human eye is more sensitive to this color. The pixels are arranged in the following pattern: green pixels are always diagonal to each other, as are the red and blue pixels.



The *MbufBayer()* function can also white balance the Bayer image during conversion. White balancing adjusts an image for color variations introduced by the lighting conditions when the image was grabbed. The function converts pixels that represent white so they appear as close to white as possible, and adjusts other pixels accordingly. White balancing is discussed in greater detail later in this section.

## Using MIL to convert the image

The steps below describe, in general, how to convert a Bayer image using MIL:

1. Determine the white balance coefficients (optional). For information on how to calculate the white balance coefficients, see the subsection, *White balancing your Bayer images*.
2. Grab or load a Bayer image into your source buffer.
3. Apply the MIL Bayer filter on the image using *MbufBayer()*, including the white balance coefficients, if using.

Below is an example of how to grab a Bayer image and convert it to a 3-band color image. This example also shows how to correct the white balance, which will be discussed later in this section.

```
/*
 * Synopsis: This program shows how to perform Bayer-to-Color conversion.
 */
#include <mil.h>
#include <conio.h>

void main(void)
{
    MIL_ID MilApplication,
           MilSystem,
           MilDigitizer,
           MilDisplay,
           MilWBCoefficients,
           MilImageDisp,
           MilImageGrab;

    /* User array for white balance coefficients. */
    float WBCoefficients[3];

    /* Specify the Bayer pattern of your camera. */
    long ConversionType = M_BAYER_GR;

    /* Buffer characteristics. */
    long XSize;
    long YSize;

    /* Allocate an application. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
                    &MilDigitizer, M_NULL);
```

(cont...)

```

XSize = MdigInquire(MilDigitizer, M_SIZE_X, M_NULL);
YSize = MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL);

/* Allocate a display buffer. */
MbufAllocColor(MilSystem, 3, XSize, YSize, 8L+M_UNSIGNED, M_PROC+M_IMAGE,
               &MilImageDisp);

/* Allocate a grab buffer. */
MbufAllocColor(MilSystem, 1, XSize, YSize, 8L+M_UNSIGNED,
               M_IMAGE+M_DISP+M_GRAB+M_PROC, &MilImageGrab);

/* Allocate an array for the white balance coefficients. */
MilWBCoefficients = MbufAllocId(MilSystem, 3, 32+M_FLOAT, M_ARRAY, M_NULL);

/* Display the image. */
MbufClear(MilImageDisp, M_RGB888(0, 0, 0));

MdispSelect(MilDisplay, MilImageDisp);

/* Ask the user for a white image for white balance. */
printf("Place a white paper in front of the camera and " \
       "press <ENTER> when ready.\n");

do
{
    /* Grab a white Bayer image. */
    MdigGrab(MilDigitizer, MilImageGrab);

    /* Convert the white Bayer image to color without white balance. */
    MbufBayer(MilImageGrab, MilImageDisp, M_DEFAULT, ConversionType);
}
while (!kbhit());
getch();

/* Determine the white balance coefficients. */
MbufBayer(MilImageGrab, MilImageDisp, MilWBCoefficients,
          ConversionType+M_WHITE_BALANCE_CALCULATE);

/* Print the computed coefficients. */
MbufGet(MilWBCoefficients, (void *) &WBCoefficients[0]);

printf("\nWhite balance correction coefficients : %f, %f, %f\n\n",
       WBCoefficients[0], WBCoefficients[1], WBCoefficients[2]);

/* Grab a new Bayer image with white balance correction. */
printf("Press <ENTER> to grab an image\n");

getchar();

```

(cont...)

```

do
{
/* Grab a Bayer image. */
MdigGrab(MilDigitizer, MilImageGrab);

/* Convert the Bayer image to color. */
MbufBayer(MilImageGrab, MilImageDisp, MilWBCoefficients, ConversionType);
}
while (!kbhit());

getch();

printf("Press <ENTER> to end\n");

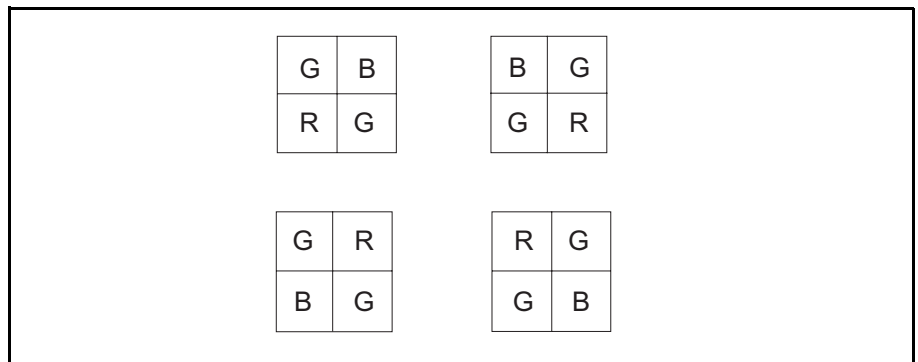
getchar();

/* Terminate and free everything. */
MbufFree(MilImageGrab);
MbufFree(MilImageDisp);
MbufFree(MilWBCoefficients);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, MilDigitizer,
                M_NULL);
}

```

### How the Bayer image gets converted

Bayer images are arranged in groups of 2x2 pixels. Each group contains one blue pixel, one red pixel, and two green pixels; the values of which are used in calculating the corresponding bands of the destination pixel. Your camera will grab an image with one of the following four patterns:



You must specify the pattern that is used by your camera when calling *MbufBayer()*. Since the green pixels are always diagonal to each other, specifying the starting two pixels of the pattern defines the pattern uniquely. Consult your camera's documentation or contact the manufacturer if you are unsure; the Bayer image

will not be converted properly if you specify the wrong pattern. If you cannot obtain information regarding the pattern of your camera, try all of *MbufBayer()*'s supported patterns to find the correct one.

The value of a source pixel is used in the corresponding band of its destination pixel. The two remaining color components use the average value of the source pixel's corresponding neighbors. If the source pixel is green, then the average value for the remaining two components (red and blue) is based on two neighboring pixels.

B <sub>1</sub>	G <sub>2</sub>	B <sub>3</sub>	G <sub>4</sub>	:
G <sub>11</sub>	R <sub>12</sub>	G <sub>13</sub>	R <sub>14</sub>	:
B <sub>21</sub>	G <sub>22</sub>	B <sub>23</sub>	G <sub>24</sub>	:
G <sub>31</sub>	R <sub>32</sub>	G <sub>33</sub>	R <sub>34</sub>	:
:	:	:	:	:

B <sub>1</sub>	G <sub>2</sub>	B <sub>3</sub>	G <sub>4</sub>	:
G <sub>11</sub>	R <sub>12</sub>	G <sub>13</sub>	R <sub>14</sub>	:
B <sub>21</sub>	G <sub>22</sub>	B <sub>23</sub>	G <sub>24</sub>	:
G <sub>31</sub>	R <sub>32</sub>	G <sub>33</sub>	R <sub>34</sub>	:
:	:	:	:	:

If the source pixel is either red or blue, the average value for the remaining two components is based on four neighboring pixels.

B <sub>1</sub>	G <sub>2</sub>	B <sub>3</sub>	G <sub>4</sub>	:
G <sub>11</sub>	R <sub>12</sub>	G <sub>13</sub>	R <sub>14</sub>	:
B <sub>21</sub>	G <sub>22</sub>	B <sub>23</sub>	G <sub>24</sub>	:
G <sub>31</sub>	R <sub>32</sub>	G <sub>33</sub>	R <sub>34</sub>	:
:	:	:	:	:

B <sub>1</sub>	G <sub>2</sub>	B <sub>3</sub>	G <sub>4</sub>	:
G <sub>11</sub>	R <sub>12</sub>	G <sub>13</sub>	R <sub>14</sub>	:
B <sub>21</sub>	G <sub>22</sub>	B <sub>23</sub>	G <sub>24</sub>	:
G <sub>31</sub>	R <sub>32</sub>	G <sub>33</sub>	R <sub>34</sub>	:
:	:	:	:	:

Note that if the source pixel is on an edge of the image, MIL will use as many neighbors as possible when determining the average pixel value for the remaining components.

When the destination buffer is in YUV format, MIL converts the Bayer image first to RGB, and then to YUV.

### White balancing your Bayer images

Sometimes grabbed images appear with “the wrong colors”. This is due primarily to color distortions introduced by the light source or lighting conditions. Such distortions can be corrected by white balancing the image.

On the premises that white pixels should contain no chrominance, white balancing applies a coefficient to each band of the image so that “white” pixels contain no chrominance. For RGB images, this means that a given white pixel’s value in all 3 bands is equal. For YUV images, this means that the U and V bands of a white pixel are equal to 0. After white balancing an image, pixels that are white appear white (or a shade of gray), and the other pixels also appear with the correct colors. The result is an image that more accurately reflects the colors of the object that was grabbed.

The steps below show how to white balance Bayer images using the *MbufBayer()* function:

1. Grab an image that is entirely white. This can be done by holding a white object, such as a piece of paper, in front of the camera. Ensure that the image is grabbed in the same lighting conditions as subsequent source images. Note that it is unlikely you will be able to grab an image whose pixel values are exactly 255.
2. Allocate a 3 x 1 MIL array of type M\_FLOAT using *MbufAlloc2d()*.
3. Call *MbufBayer()*, using the white image as the source image, and adding M\_WHITE\_BALANCE\_CALCULATE to the control flag. This call calculates the coefficients required to white balance the specified image and passes them to the array.
4. Grab the image required for the Bayer conversion.
5. Call *MbufBayer()*, using the new source image and the white balance coefficient array.

The white balance coefficients are calculated differently, depending on whether your destination image is RGB or YUV.

**RGB images**

For an RGB destination image, three white balance coefficients,  $a$ ,  $b$ , and  $c$ , are calculated and passed to the array as the first, second, and third values, respectively. These coefficients are for a given lighting condition, and calculated such that given an image of a flat white surface in that lighting:

$$a\bar{R} = b\bar{G} = c\bar{B}$$

where  $R$ ,  $G$ , and  $B$  with macrons ( $\bar{\phantom{x}}$ ) are the average values of the red, green, and blue color components, respectively. When subsequent source images are converted, the pixels of each color component are multiplied by their corresponding coefficient.

**YUV images**

For a YUV destination image, the coefficient of the  $Y$  component is set to 1 by default. The remaining two white balance coefficients,  $b$  and  $c$ , are calculated and passed to the array as the second and third values, respectively. These coefficients are for a given lighting condition, and calculated such that given an image of a flat white surface in that lighting:

$$b + \bar{U} = c + \bar{V} = 0$$

where  $U$  and  $V$  with macrons ( $\bar{\phantom{x}}$ ) are the average values of the  $U$  and  $V$  components, respectively. When subsequent source images are converted, the pixels of the  $Y$  component are multiplied by the first value of the array (1 by default), and the pixels of the  $U$  and  $V$  bands are summed with the second and third values in the array.

**Monochrome images**

If the format of the destination buffer is 8-bit monochrome, the pixels of the image are multiplied by the first value in the array, which is 1 by default; the last two values in the array are ignored.

Note that if your image is grabbed in dark conditions, you might not only want to white balance your image, which will adjust the colors in your image, but you might also want to adjust the intensity. For monochrome and YUV images you can pass a greater value as the first element of the array if your image is dark; if your image is bright, pass a value greater than 0 and less than 1.



**Chapter**

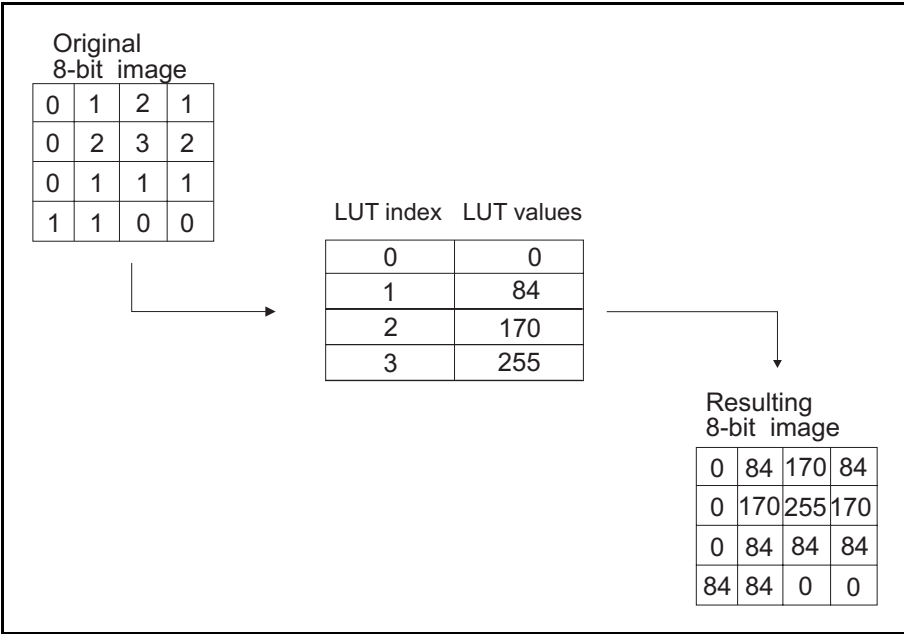
# 18

## **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

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 difference between the start value and the end value divided by the number of entries specified by the index range produces the increment. The increment is then used to generate the remaining entries of the index range.

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 a value for each LUT index within the specified index range according to a specified mathematical function. The functions available are: M\_LOG, M\_EXP, M\_SIN, M\_COS, M\_TAN, and M\_QUAD. The specified start value is used as the initial X value in the equation. The remaining entries of the index range are generated by incrementing the value of X by 1 for each index.

The *MimHistogramEqualize()* command can be used to create a LUT for intensity correction.

## Color LUTs

When generating data in a color LUT buffer, the same data is written to all bands.

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()*.

Alternatively, you can allocate three separate one-dimensional child buffers into which the values for each color band will be generated. The use of child buffers will cause the values for each color band in the LUT buffer to be automatically updated and no copying is necessary.

### **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 19: Displaying an image*.

**Associating a LUT to a digitizer**

**LUTs and digitizers**

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**

# 19

## **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

---

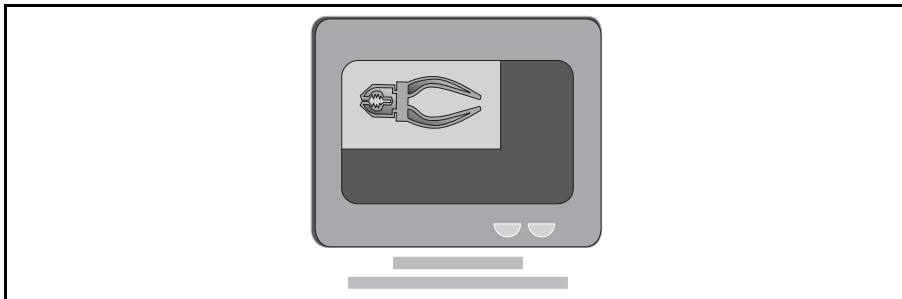
Whether or not your imaging board has a display section, MIL can display images. It will use the most appropriate graphics controller in your computer for display purposes. If your imaging board has a display section, and it is available, MIL will typically use it for display purposes.

### 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()*. Note that the buffer and the display should be allocated on the same system.

### Selecting a buffer for display

Once a buffer and a display have been allocated, use *MdispSelect()* to select the image buffer to display. The buffer is displayed in a dedicated window or at the top-left corner of an auxiliary screen. If the specified image buffer is smaller in size than the display, the border outside the buffer is blacked 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.



Note that a buffer, or any of its child regions, can be selected on more than one display.

### Frame buffer

This manual uses the term frame buffer to refer to *physical* display (graphics controller) memory (not a buffer, *per se*).



## Types of displays

---

You can allocate a display that, when an image is selected to this display, it is either displayed:

- With a windowed border, which is called a **windowed display** (M\_WINDOWED).
- Without a windowed border on a dedicated screen, which is called an *auxiliary display* (M\_AUXILIARY).

You must specify either one of these two types of displays upon allocating the display, with *MdispAlloc()*.

- ❖ Typically, the default display type (M\_DEFAULT) is M\_WINDOWED. However, a Matrox imaging board might be dedicated for MIL auxiliary display, which can make the default M\_AUXILIARY. For more information, see the board's installation and hardware reference manual.

### Windowed display

An image selected to a windowed display is displayed with a windowed border on the Windows desktop screen(s). To choose a windowed display, set the initialization flag for *MdispAlloc()* to M\_WINDOWED.

#### Extended desktop

A windowed display is not affected by whether or not your desktop is displayed using one screen or multiple screens. We refer to these screens, either one or many, as the **Windows desktop screen(s)**. Under Windows 98, 2000, and Me, your desktop can be extended over screens of different resolutions. However, under Windows NT, you must observe the following restrictions: all your monitor settings (resolution) must be the same as your least-capable monitor, a maximum of 4 boards (Matrox imaging boards and/or Matrox MGA boards) can be used, and the extended desktop must be on screens that are positioned in a horizontal, vertical, or tiled fashion.

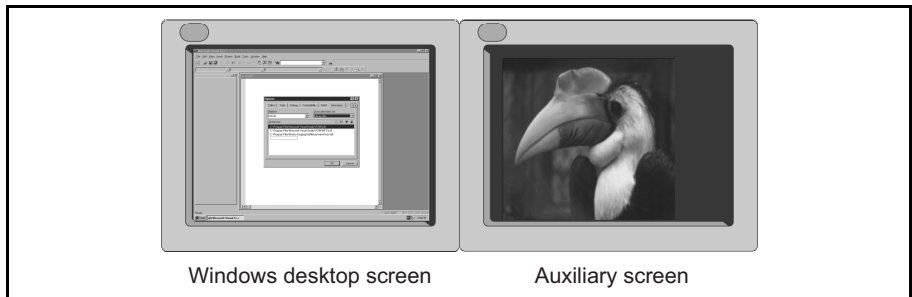
**All windowed displays** (M\_WINDOWED) are displayed in their own MIL default window (or, as will be seen later, in a user-allocated window). This window is transparently tracked and updated with the image buffer selected to the display; that is, if the window moves or is occluded, the window is automatically updated with the image buffer accordingly.

Multiple windowed displays can be allocated and selected for display; the display device number should always be set to `M_DEFAULT`.

For windowed displays, MIL does not typically communicate directly with the graphics controller, but uses the normal Windows mechanisms (Windows API functions and extensions) to display images. Upon selecting a windowed display, MIL allocates a second image buffer in a Windows Device Independent BITMAP (DIB) or DirectDraw format, passes Windows the address of this buffer, and copies the contents of the selected buffer to this displayed buffer. MIL also loads display LUT buffers into Windows' logical palettes (only in 256 color display resolution). Refer to the Microsoft SDK Programming Guide for information on Windows' DIBs, DirectDraw, and logical palettes.

### **Auxiliary display**

An image selected to an auxiliary display is displayed without a windowed border or frame, at the top-left corner of a screen that is not used to display the Windows desktop. This screen is referred to as an **auxiliary screen**.



Note that in this context, the term screen refers to any device that supports video output data, such as a high-resolution monitor, TV, or VCR.

Auxiliary displays are supported with the following minimum resources:

- Two graphics controllers.
- One DualHead graphics controller that integrates two CRT controllers (for example, Matrox G400, G450, or G550).

- ❖ Note that the graphics controller can be on, or apart from, the Matrox imaging board. Also, some boards might have special features or limitations regarding auxiliary displays; please see the *MIL/MIL-Lite Board Specific manual*.

You can only allocate one auxiliary display at a time on a given auxiliary screen. Moreover, you are responsible for moving and tracking an auxiliary display, if required. To choose this type of display, set the initialization flag for *MdispAlloc()* to *M\_AUXILIARY*.

- ❖ When using an imaging board with a display section (for example, Matrox Genesis), it might be necessary to set a Dip switch to display on an auxiliary screen. For more information, see the board's installation and hardware reference manual.

#### Video output format

When allocating an auxiliary display, MIL does not impose any restrictions on the video output format of the auxiliary screen. The format can be:

- A high-resolution format (for example, 1024x768x32@70 hz).
- An encoded video format (for example, NTSC/PAL).
- ❖ For all the supported formats, see the *MIL/MIL-Lite Board Specific Notes*.

The video output format of the auxiliary screen is set with the display format parameter of *MdispAlloc()*. For example:

```
MdispAlloc(MilSystem, M_DEFAULT, "M_NTSC", M_AUXILIARY, &MilDisplay1);
MdispAlloc(MilSystem, M_DEFAULT, "1024x768x32@70", M_AUXILIARY, &MilDisplay2);
```

The maximum number of auxiliary displays that can be allocated is determined by the number of CRT controllers that support the specified format. For example, two auxiliary displays with high-resolution formats can only be allocated if there are two available CRT controllers that support high-resolution formats.

- ❖ When allocating a display, MIL checks for an appropriate CRT controller, and not an appropriate device attached to it. It is therefore possible to allocate an auxiliary display without an auxiliary screen connected to the CRT controller.

#### Windows NT

Under Windows NT, if the auxiliary screen is driven by the same board as the Windows desktop screen, the desktop's resolution must be larger than the resolution of the auxiliary display. For example, to allocate an auxiliary display

with a resolution of 1024x768x32, the Windows desktop's resolution must be at least 1152x864x32. Note that if the Windows desktop screen and the auxiliary screen are controlled by two separate graphics controllers that have their own frame buffers, this restriction does not apply.

**Matrox Millennium  
G400, G450, G550**

To use the second CRT controller of Matrox Millennium G400, G450, or G550 for MIL auxiliary display, your display driver's DualHead mode must be disabled; otherwise both the display driver and the MIL driver will attempt to access the second CRT controller. In addition, the G400's second CRT controller does not support encoded video formats, but the G450 and G550 do.

- ❖ If the performance of the display is slow, make sure your display driver's DualHead mode is set correctly.

**Display number**

The display number parameter of *MdispAlloc()* should always be set to M\_DEFAULT. Based on the specified format, MIL will find the best device to use when displaying an image. If your imaging board has a display section, and it is available, MIL will typically use it for display purposes.

- ❖ If you need to allocate an on-board image buffer, it is important to note that, since MIL selects which device will be used to display the image, you should only allocate this buffer (*MbufAlloc()*) after allocating the display to which it will be selected (*MdispAlloc()*).

**Display size and depth**

For windowed displays, the display format of the on-screen portion of the frame buffers is set using the selected Windows display resolution. In this case, the display format parameter of *MdispAlloc()* should be set to M\_DEFAULT.

For auxiliary displays, you set the display format with the display format parameter of *MdispAlloc()*.

When you select a buffer to a windowed display, Windows will create a display of the same size as the buffer, unless such a display cannot fit in the Windows desktop. If the image is too large, there will be scroll-bars to view other parts of the image, and the initial view of the image will be the upper-left corner. If the image is too small, it will be centered in the buffer, and the surrounding area will be blacked out.

### Displaying buffers of different data depths

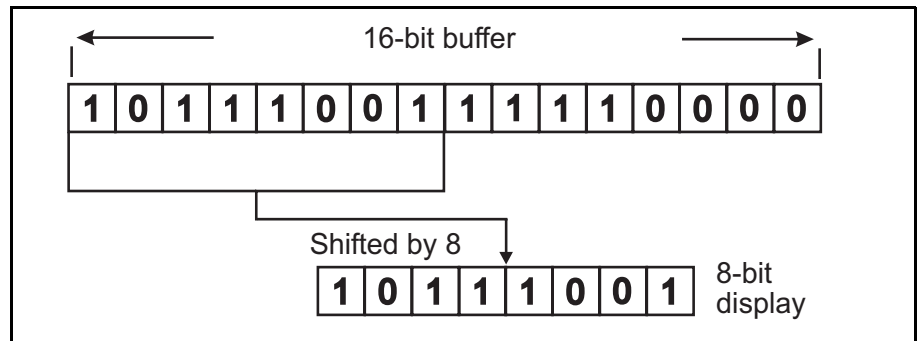
Displayable image buffers usually have a depth of 8-bits (or 3-band 8-bits, in the case of color images). For windowed displays, you can display images of other depths (for example, 1-bit or 16-bit images). By using *MdispControl()* with the `M_VIEW_MODE` control type, you can control the way such buffers are actually displayed.

The `M_VIEW_MODE` control type provides different modes of displaying non 8-bit images:

- `M_BIT_SHIFT`
- `M_AUTO_SCALE`
- `M_MULTI_BYTES`
- `M_DEFAULT`

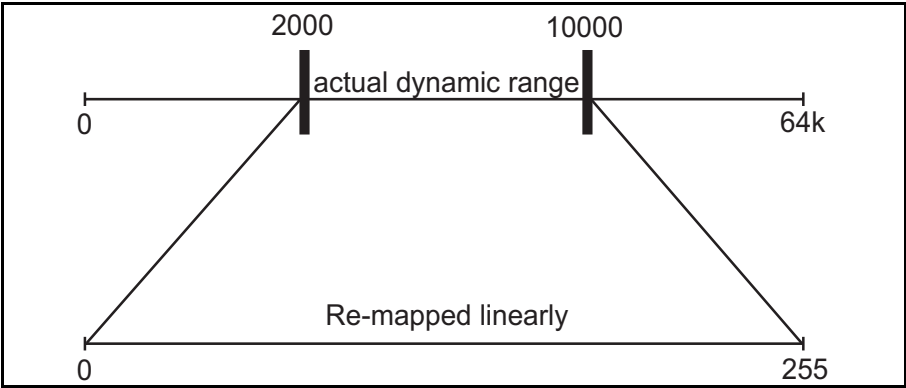
#### `M_BIT_SHIFT`

The `M_BIT_SHIFT` setting will bit shift the pixel values of the image by the specified number of bits upon updating the display.

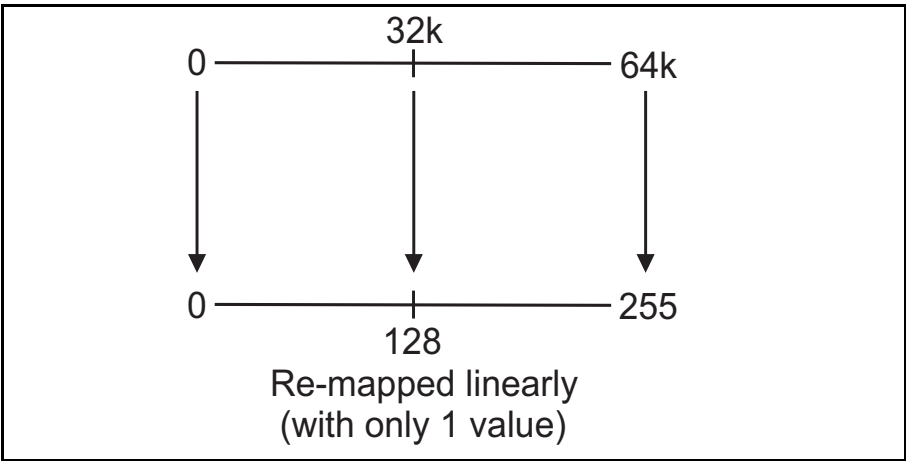


M\_AUTO\_SCALE

The M\_AUTO\_SCALE setting remaps the pixel values to the display range such that the minimum and maximum values in the image (not the full range of the buffer) are set to 0 and 255, respectively.

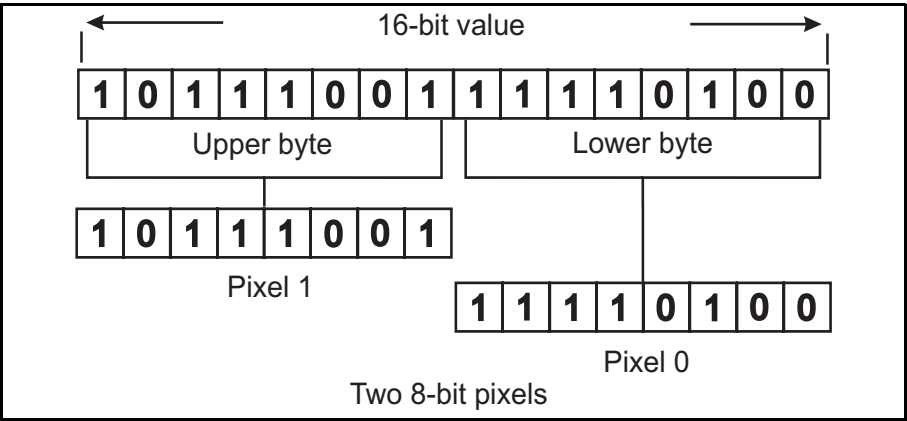


If the image buffer contains a single value, its corresponding displayed value is determined by linearly re-mapping the full range of the buffer; for example, (0 to 64K) to (0 to 255).



M\_MULTI\_BYTES

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.



M\_DEFAULT

The M\_DEFAULT setting will automatically select the appropriate mode, depending on the image depth.

## Removing a buffer from the display

After displaying an image buffer (with *MdispSelect()*), you can remove it from the display and close the associated window (for windowed displays), or leave the display blank (for auxiliary displays), 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.

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 (for windowed displays) or is left blank (for auxiliary displays).

## Displaying multiple buffers

---

*MdispSelect()* allows you to view one buffer at a time in one display. You can, however, use many windowed displays (up to a maximum of 64) and therefore view more than one buffer at the same time on the Windows desktop screen(s).

This is not the case for auxiliary displays, where you can only display one display at a time on a given auxiliary screen. However, you can still 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 using *MbufAlloc2d()* or *MbufAllocColor()*. This buffer will be known as the parent buffer.
2. Allocate two non-overlapping child buffers within it, using *MbufChild2d()* or *MbufChildColor()*.
3. Select the parent buffer for display using *MdispSelect()*.
4. Use one of the child buffers as the source image buffer and the other as a destination image buffer of the operation.

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();

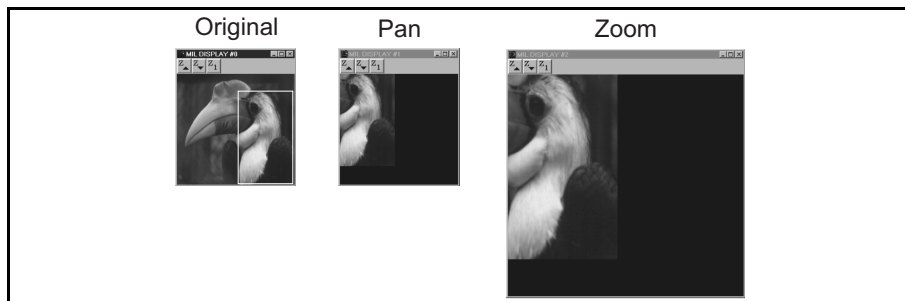
/* Close the display. */
MdispDeselect(MilDisplay, MILParentImage);

/* 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.



### 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 (for windowed displays) or screen (for auxiliary displays), using *MdispPan()*.

### Zooming

Zooming is the horizontal and/or vertical replication of each pixel by a given integer factor. You can zoom the display by an integer factor using *MdispZoom()*. 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, 2, 2);
```

Note that zooming by a large factor might cause a "blocky" effect. Also, you can reduce the size of an image on the display by passing a negative zoom factor to *MdispZoom()*.

### Image placement

For auxiliary displays, to view 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. For windowed displays, this is automatically handled.

## Annotating the displayed image non-destructively

---

For either windowed displays or auxiliary displays, you can annotate the displayed image non-destructively using MIL's overlay-display mechanism. To make use of this functionality, 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 also associate a temporary overlay buffer with the display. This buffer is referred to as the *display's overlay buffer*. This overlay buffer can be used to annotate the underlying image with an effect called keying, which replaces pixels of one image that are of the specified keying color with the underlying areas of another image. Therefore, anything that you draw in this overlay buffer in a color other than the keying color, will annotate the image selected to the display.

3. To access the display's overlay buffer, use the following call to determine the MIL identifier of the buffer:

```
MdispInquire(DisplayID, M_WINDOW_OVR_BUF_ID, &OverlayBufferID)
```

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()*. Note that since this temporary overlay buffer is a real buffer, any function (except grabbing) can be used.
  - ❖ You can also annotate the displayed image buffer with Windows GDI annotations, which is discussed later.

Typically, the overlay buffer will have the same number of bands and will be the same size as the buffer selected to the display (not the size of the display). However, if you are using a non 8-bit display resolution (15-bit, 16-bit, 24-bit, or 32-bit color resolution), and the image selected to the display is 1 band, then the overlay buffer is 3 bands.

- ❖ Note that if the graphics controller does not have non-destructive overlay capabilities, and you are using a non 8-bit display resolution (15-bit, 16-bit, 24-bit, or 32-bit color resolution), a 1-band image will have a 1 band overlay buffer.

#### Overlay buffer behavior

When an image is selected to a display that has an overlay buffer associated with it, and you select another image to that display, which:

- Has the same dimensions as the image currently selected to that display, the current overlay buffer is not freed. Any annotations will, therefore, remain until you clear the overlay buffer, with *MbufClear()*.
- Has different dimensions than the image currently selected to that display, the current overlay buffer is freed, and another overlay buffer is created. The annotations of the old overlay buffer are copied into the new one. Note that the overlay buffer is now the size of the new image selected to the display.

#### CPU-assisted overlay

The ability to annotate the displayed image non-destructively by using MIL's overlay-display mechanism is always available, and is typically accomplished by your hardware (that is, your graphics controller). However, if your hardware limits have been reached, MIL produces a simulated version of the overlay effect by using the CPU; the display is therefore said to be CPU-assisted.

#### Keying

When allocating a display (*MdispAlloc()*), keying is automatically enabled, if required, and the keying color is automatically set to a default color, which is generally appropriate. This keying color can be read with the `M_KEY_COLOR` inquire type of *MdispInquire()*. If required, select another keying color with *MdispOverlayKey()*.

If you are using an 8-bit display resolution (256 colors), set the keying 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 color resolution), call the macro `M_RGB888` and specify the RGB value. For example:

```
MdispOverlayKey(..., M_RGB888(20,32,24), ...).
```

When the display's overlay buffer is created, it is cleared to the effective keying color. If the keying color is changed after the overlay buffer is created, the buffer will not be cleared.

### Using GDI annotations

If the display has been selected, you can also annotate the displayed image buffer with 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).

After either buffer is changed, signal MIL by calling *MbufControl*(..., `M_MODIFIED`,...). When you have finished using the DC, free it immediately by calling *MbufControl*(..., `M_WINDOW_DC_FREE`,...).

- ❖ If using a `DDRAW` buffer, you must free the DC before signalling MIL.
- 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(MiOverlayImage, M_WINDOW_DC_ALLOC, M_DEFAULT);

/* Inquire the device context. */
hCustomDC = ((HDC)MbufInquire(MiOverlayImage, 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(MiOverlayImage, M_WINDOW_DC_FREE, M_DEFAULT);

/* Signal MIL that the overlay buffer was modified. */
MbufControl(MiOverlayImage, M_MODIFIED, M_DEFAULT);
```

## Displaying an image in a user-defined window

---

For windowed displays, images are automatically displayed in a default window created by MIL, using *MdispSelect()*. This function dynamically creates a window on 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.

### Selecting a buffer into a specific display window

However, for windowed displays, you can choose to display a specific image buffer in a user-defined window, using *MdispSelectWindow()*. Note that typically, the display need not have the same resolution as the image buffer. If the defined window is of a different dimension than the image buffer, any excess window area will be left untouched or any excess image area will be cropped.

### Using *MdispSelectWindow()*

The *MdispSelectWindow()* function is similar to *MdispSelect()*, except that it allows you to specify the handle of the user-defined window or child window to use for display, rather than displaying into a MIL created window. This user-defined 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.

Note that the user-defined window must have been created with Windows API functions. In addition, if the handle parameter of *MdispSelectWindow()* is set to zero, this function behaves like *MdispSelect()*.



The following portion of MIL code from the *mwindisp.c* example shows 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 BUFFERSIZEX      640
#define BUFFERSIZEY      480
#define BUFFERSIZEBAND   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;
    }

    /* Only allow example to run for windowed displays */
    if (MdispInquire(MilDisplay, M_DISPLAY_MODE, M_NULL) != M_WINDOWED)
    {
        MessageBox(0, "" "This example only runs for windowed displays.",
        "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);
}

```

## Palettes and output LUTs for windowed display (256-color)

For windowed displays, when displaying in a 256-color Windows display resolution, images are mapped through physical output LUTs on display. Windows uses these LUTs to achieve color; in addition, Windows uses the concept of a palette to load these LUTs.

MIL provides Windows with good default logical palettes for the realization of the physical output LUTs in a 256-color display resolution. However, since not all colors are available at this resolution, the default might not always be optimal for certain atypical cases. Therefore, the physical output LUTs are always available and programmable, by a user, to achieve the best display effect for your images.

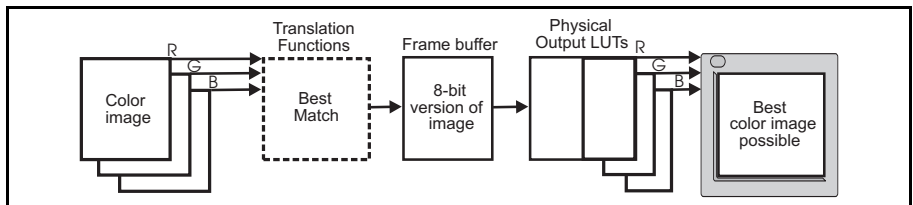
### Reference material: Windows palettes and physical output LUTs

Before describing the default and explaining how and why to change it, a basic explanation of palettes and output LUTs is discussed. For more detailed information, consult the following reference:

Halibard, Moishe. Windows Developer's Journal. "Palettes and 256 Colors." July, 2000.

#### Color images

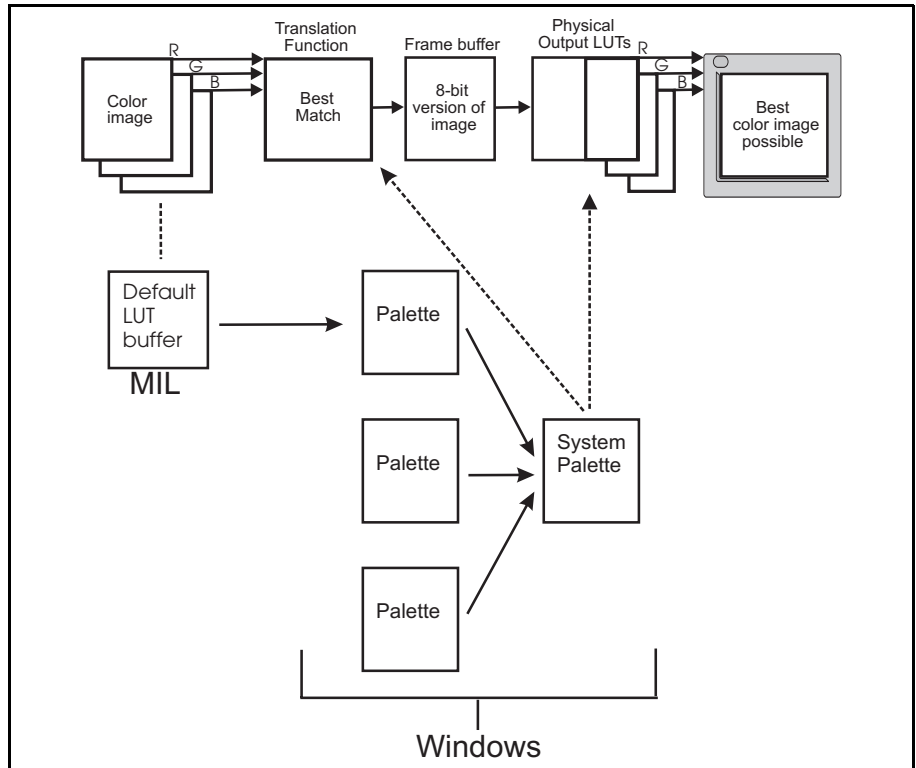
For windowed displays, in a 256-color display resolution, the following example shows how a color image is consequently displayed as the best color image possible:



1. The color image goes through Windows translation functions. Windows searches the physical output LUTs for the entry that best matches the color of the image's pixels. Depending on what colors are available, Windows translates the image to an 8-bit index image.
2. The physical output LUTs subsequently translate that index image and display it as the best color image possible.

## Palettes and physical output LUTs

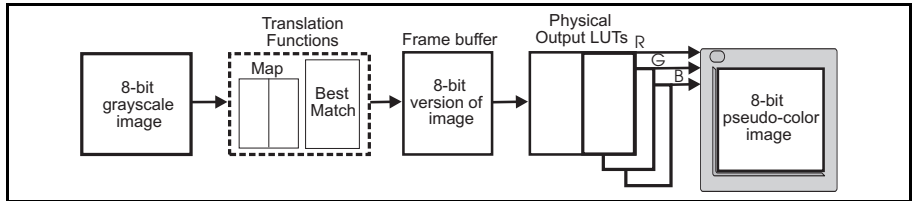
Each of the color images' palette is used to specify the required colors. Windows uses these palettes to realize a system palette. The system palette is ultimately loaded into the physical output LUTs and is searched by the translation functions when generating an 8-bit (index) image.



Note that the palette of the active window has priority over available system palette entries and is therefore loaded in the system palette first. Also, Windows tries to map colors from the logical palette into the currently realized system palette to reduce the number of requested new entries. This reduces the chance of a color occurring more than once in the system palette.

**Monochrome images**

Since LUTs are available, you can use them to create special effects for 8-bit images. However, unlike color images, grayscale values map to values at the corresponding index in their associated palette.

**Default palette settings**

By default, windowed displays use the default MIL palette. MIL provides good default logical palettes for the realization of the physical output LUTs (*MdispLut(..., M\_DEFAULT, ...)*). To accomplish this, the MIL default takes into consideration the number of bands of the image, and produces the best performance versus visual quality compromise possible.

**Why change the default LUT values**

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 infrared camera.
- When displaying monochrome images, you might want to invert the image values. For example, when grabbing a film negative, you can negate the video and display the film as it will be printed.
- When displaying color images in a 256-color Windows display resolution, you might want to reduce the loss of color resolution. Generally, the default LUT values will not optimize and distinguish between subtle differences in one color. For example, when displaying a color image with many shades of red, you might want to select a LUT so that all shades of red in that particular image are being represented.

Changing the default LUT values

Change the default LUT values by associating a LUT to either the display, using *MdispLut()*, or to the image buffer, using *MbufControl()*, with M\_ASSOCIATED\_LUT. Changing the default LUT values by associating a LUT to the display affects all images displayed in that display. On the other hand, changing the default LUT values by associating a LUT to the buffer affects only the display of that buffer. In this case, when the buffer is saved, the LUT is saved with it.

Viewing each gray intensity in a different color

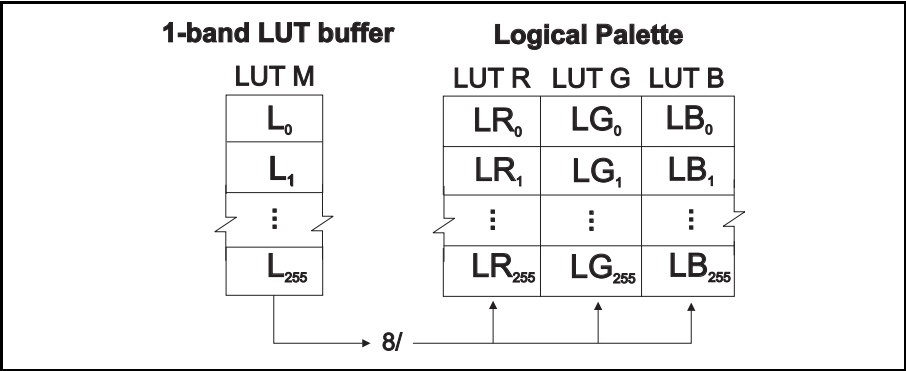
To view an 8-bit image buffer with each gray intensity in a different color, associate the default pseudo-color LUT buffer (M\_PSEUDO) with the display of the image. In this case, the data is loaded in each component of the logical palette.

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 1-band 256-entry LUT buffer using *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 using *MbufControl()* with M\_ASSOCIATED\_LUT.

If you associate a 1-band LUT buffer with the display or buffer, the same data is loaded into each component of the logical palette.



A 3-band custom LUT buffer

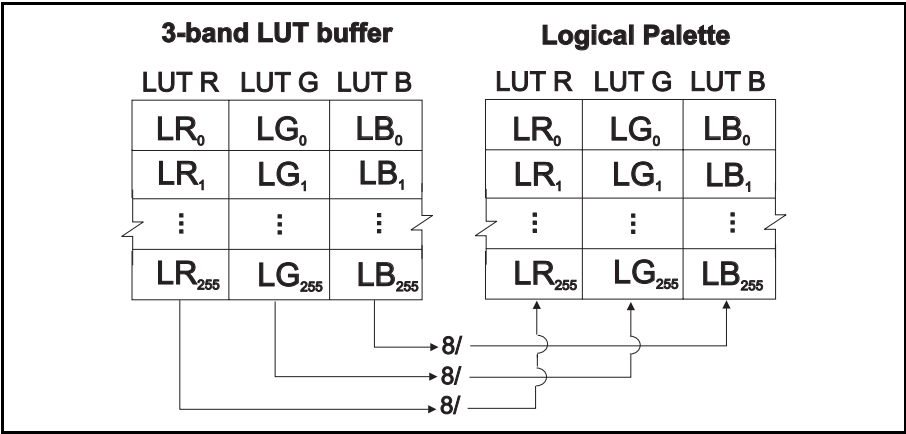
To reduce the loss of color resolution when displaying an image with a specific range of colors, you would need physical output LUTs that contain all the required colors so that, when Windows creates a translation table for the image, most colors are mapped to their exact values. To do so, allocate a 3-band 256-entry LUT buffer, and:

- 1. Take a histogram of the image.
- 2. Order and load frequently used colors in the LUT buffer according to popularity (the most popular color first).

Be careful not to remove infrequent colors that illustrate critical information.

- 3. Associate the LUT buffer with the required display using *MdispLut()*, or to a particular image using *MbufControl()* with M\_ASSOCIATED\_LUT.

When you associate the 3-band LUT buffer (RGB) with a display and then select the display, each band of the LUT buffer is loaded into its corresponding component of the logical palette.



❖ A 3-band LUT buffer can also be used to create a custom pseudo-color LUT for 8-bit images.



Note that, when using physical output LUTs, you must keep the following points in mind:

- If the contents of a LUT buffer changes while the image is selected on the display, the changes will not take effect until calling *MdispLut()* again.
- When displaying in a non-256-color display resolution, MIL can simulate a display LUT in software only for 8-bit 1-band images.
- The LUT buffer must have one or three bands. The number of LUT buffer entries must be the same as the maximum number of intensities that can be represented in the displayed image 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.

## CPU-assisted display

---

All displays are typically accelerated by the graphics controller, which means that CPU usage is low since the graphics controller is handling the display. However, if your graphics controller's limits have been reached, MIL compensates by making the display CPU-assisted. This results in higher than expected CPU usage.

In addition to higher than expected CPU usage, you might experience the following behaviors when your display is CPU-assisted:

- Overlay flickering.
- Pseudo-live continuous grabbing.

### Overlay flickering

When your graphics controller's limits have been reached, MIL uses your CPU to implement the overlay-display mechanism. Annotations in this overlay buffer might flicker, though they are still non-destructive.

**Grabbing  
continuously**

When your overlay buffer is CPU-assisted, the display can be slower and a continuous grab operation is performed only in pseudo-live. This is due to an additional operation needed to combine the grabbed image with the display's overlay buffer in an intermediate buffer. Note that the actual image buffer selected on the display is not overwritten by the contents of the overlay buffer.

**Avoiding  
CPU-assisted  
displays**

To avoid a CPU-assisted display, you can attempt to:

- Lower your screen resolution or refresh rate.
- Free all on-board buffers or displays that are no longer being used.

**Chapter**

# 20

## **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

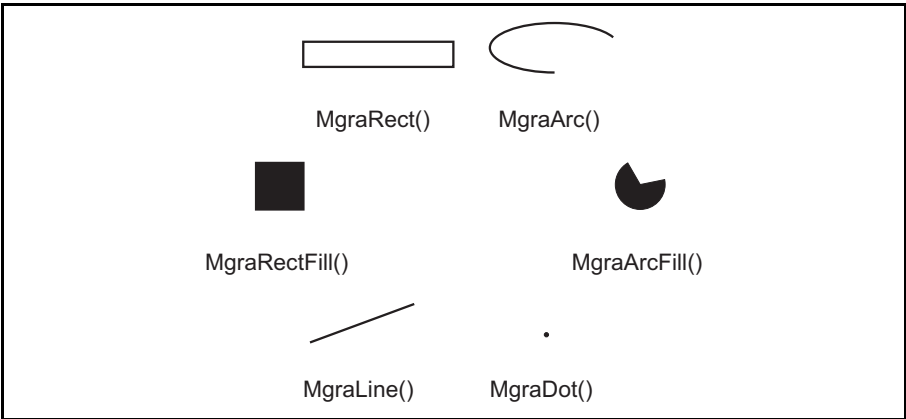
---

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.

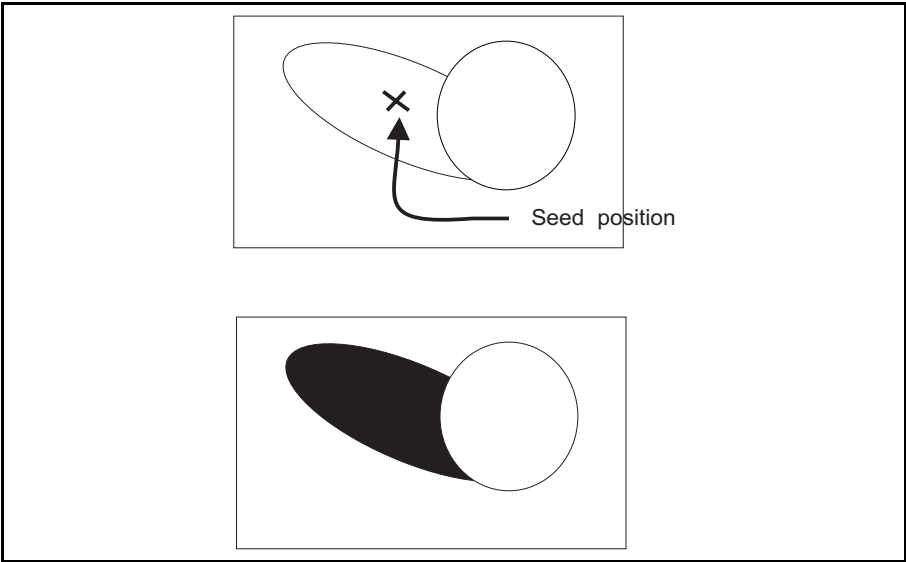
In addition, the MIL package includes *MgraRectFill()* and *MgraArcFill()* so you can draw solid rectangles and arcs.



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.



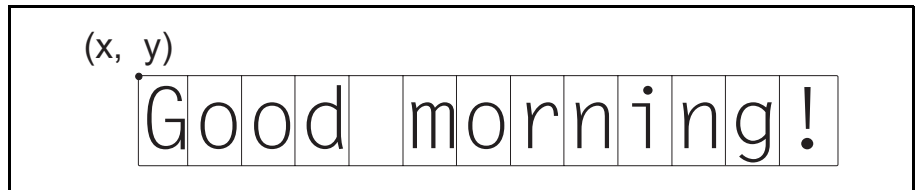
Note, any drawing is clipped outside the boundaries of the buffer.

## 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.



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**

# 21

## **Grabbing with your digitizer**

This chapter discusses the cameras supported with MIL; it also discusses 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 through the digitizer, using *MdigGrab()* or *MdigGrabContinuous()*, is stored into an image buffer. 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 camera (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 because, perhaps, you have a non-standard video source (such as a strobe or trigger device), you can create your own DCF file, 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

---

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 in 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. This means that 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 connected to 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 Corona-II) 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.

## **Grabbing a single field**

---

With interlaced scanning cameras, 2 fields are grabbed by default; therefore one call to *MdigGrab()* will grab both the odd and even fields. You can change the number of fields to 1 and have MIL treat each field as one frame using *MdigControl()* with `M_GRAB_FIELD_NUM`. Therefore, the grab time is reduced by half. This control type can only be set to 1 or 2, and should only be used for interlaced video. When set to 1, each field is treated like a frame and the following digitizer hooks are aligned with the field: `M_GRAB_FRAME_START`, `M_GRAB_END`, and `M_GRAB_FRAME_END`. To achieve 60 fps in NTSC or 50 fps in PAL, control type `M_GRAB_START_MODE` must be set to `M_FIELD_START`.

## **Line-scan cameras**

---

If your target digitizer supports it, you can grab from a line-scan camera as 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. 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).

See the *Attribute* section in *Chapter 17: Specifying and managing your data buffers* for more information on displayable buffers.

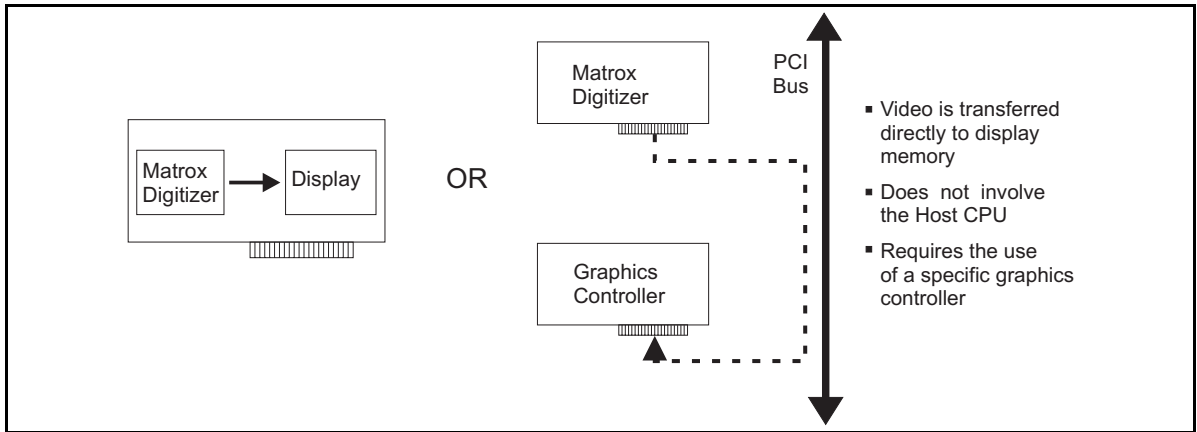
When grabbing, the digitizer (for example, Matrox Meteor-II) always acts as the bus master.

In general, a continuous grab is live, and a monoshot grab is pseudo-live. However, for auxiliary displays, it is possible to perform a live monoshot grab by allocating your buffer directly on the graphics controller with `M_ON_BOARD`.

By default, at the end of a continuous grab (live or pseudo-live), 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 graphics controller that supports fast linear-memory accesses to its frame buffer.



### Pseudo-live transfers to the display

If your graphics controller does not have non-destructive overlay capabilities, a continuous grab will automatically switch to pseudo-live if one of the following cases applies:

- Your graphics controller does not support fast linear-memory accesses (discussed later in this section).
- The format of the grabbed data is not compatible with your display resolution. For example, performing a color grab in a 256 color display resolution.
- You are grabbing to a buffer selected to a windowed display that is overlapped by another window.
- You are displaying a windowed display, and the grab display window does not have the focus (that is, it is not active).
- Your windowed display occupies more than one screen.



- ❖ For windowed displays, a continuous grab with overlay can be moved from one Windows desktop screen to another and be displayed live when it has the focus, if your graphics controller supports non-destructive overlay capabilities.
- ❖ When the Windows desktop is extended, very little CPU is typically used to perform the pseudo-live grab.

### Using an MGA graphics controller

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 graphics controller other than MGA

If your graphics controller 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 graphics controller vendor for this information). The Matrox vendor identifier is 102B, the MGA Millennium-II device identifier is 051B, the 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 graphics controller'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\_RGB15+M\_PACKED

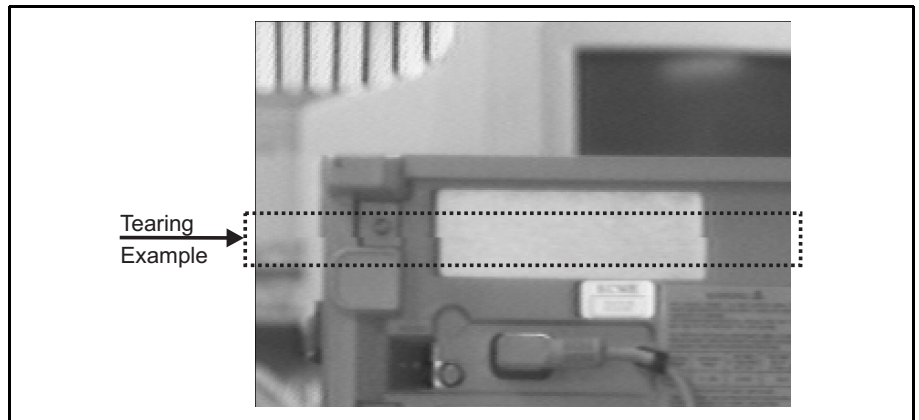
❖ All values are hexadecimal.

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.

## Screen Tearing

---

Screen tearing occurs when the grab and the display are not updated synchronously, for example:



The non-synchronous update between the grab and the display causes two images to temporarily appear simultaneously, with one of the images drifting down the screen line by line. When the lag between the grab and the display is high, the drift is fast; when the lag is slow, so is the drift.

Your monitor's vertical frequency must be a multiple of your camera's vertical frequency. In North America, this does not present a problem since both monitors and cameras usually function at 60Hz, although a phase shift can still cause tearing. However in Europe, monitors function at 50Hz, while cameras operate at 60Hz, therefore creating the disturbing visual effect.

MIL supports live grab with no tearing. The `M_LIVE_GRAB_NO_TEARING` control type in *MsysControl()* sets whether or not the no-tearing mode is enabled. This mode requires special hardware, such as a Matrox Millennium G400, G450, or G550 graphics controller. Note that if the `M_LIVE_GRAB_NO_TEARING` control type is used and is not supported, an error will be produced.

## **Reference levels, lookup tables, and scaling**

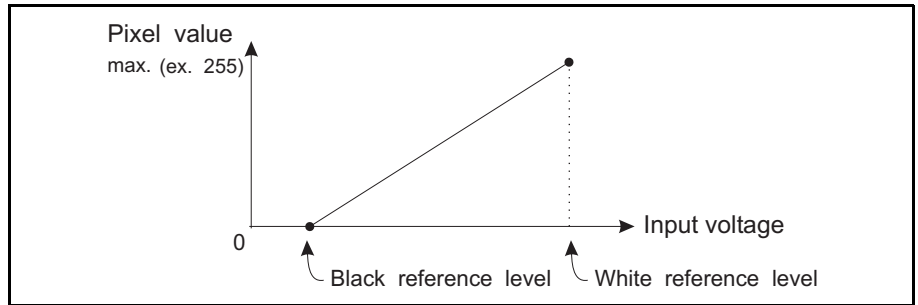
---

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.

$$\text{Value to pass to } MdigReference() = \left( \frac{\text{Voltage needed} - \text{minimum voltage}}{\text{maximum voltage} - \text{minimum voltage}} \right) \left( M\_MAX\_LEVEL - M\_MIN\_LEVEL \right)$$

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()*.

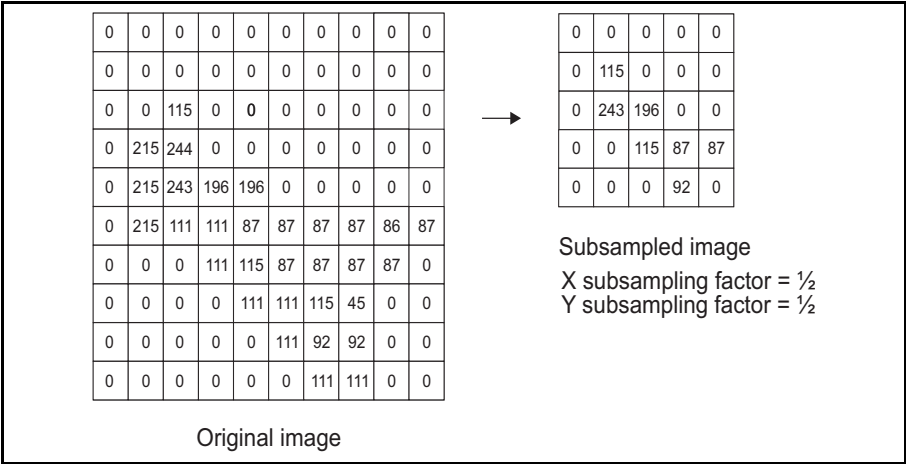
You can copy a LUT buffer that has the same number of color bands as the digitizer’s physical input LUTs. If you copy a one-band LUT buffer to a digitizer that has more than one physical input LUT, each of the digitizer’s LUTs is loaded with the same LUT buffer data.

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.



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 (*mdbproc.c*) of how to perform double buffering:

```

/* File name: Mdbproc.c
 * This example does double buffered grab with real time processing.
 * Note: This assumes 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]);
}
```

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

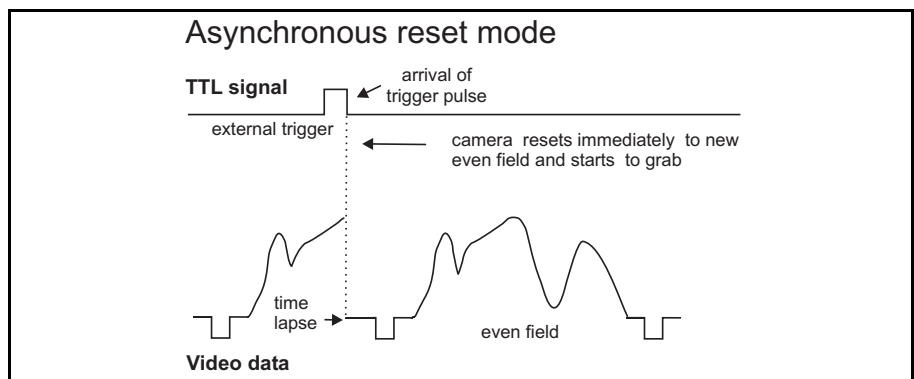
If your Matrox digitizer supports trigger input, this allows you to grab a frame upon the occurrence of an event; that is, nothing is grabbed when you call **MdigGrab()** or **MdigContinuousGrab()**, until a specified event occurs. When grabbing continuously, the digitizer waits for a trigger before grabbing each frame; you must still call **MdigHalt()** after grabbing all required frames.

The camera's digitizer configuration 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.

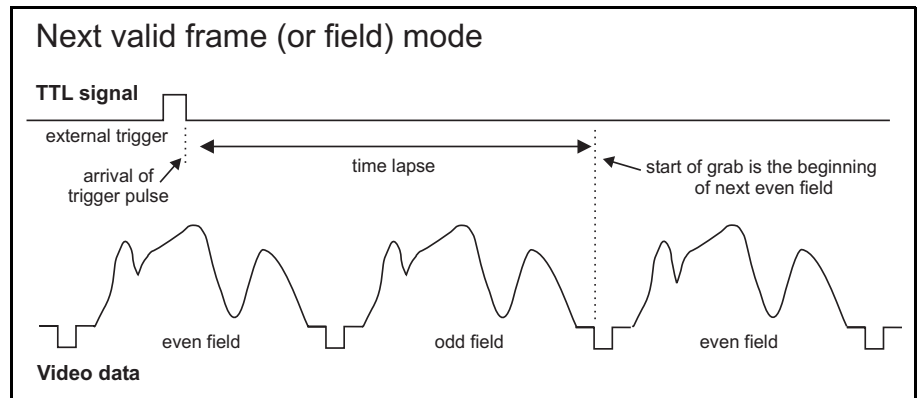
You can use MIL commands to override the DCF trigger settings. You can enable/disable whether **MdigGrab()/MdigContinuousGrab()** performs a triggered grab using **MdigControl()** with `M_GRAB_TRIGGER`. You can also specify the source and activation mode of the event upon which to grab using **MdigControl()** 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.



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.

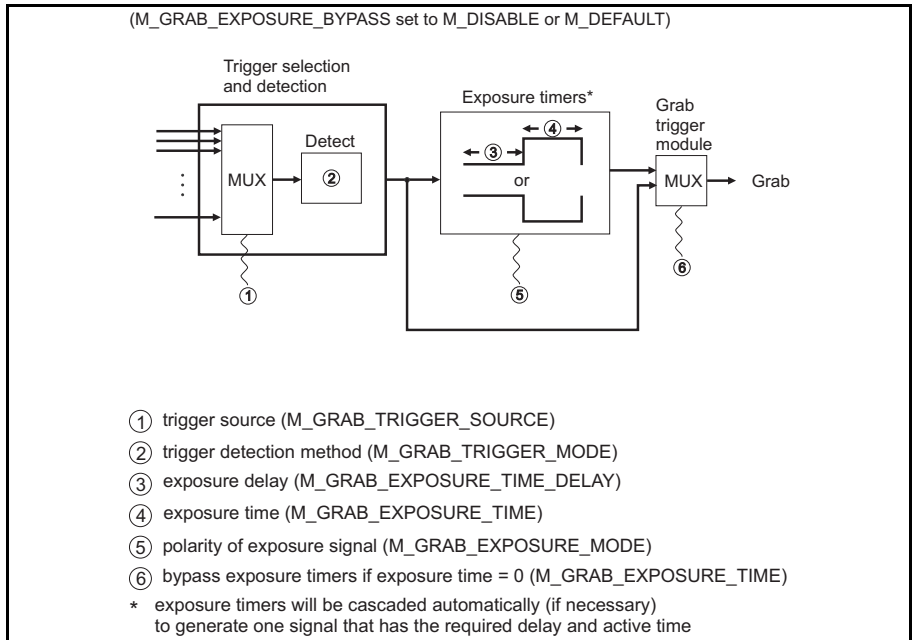


### Triggers and exposures

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).



To summarize:

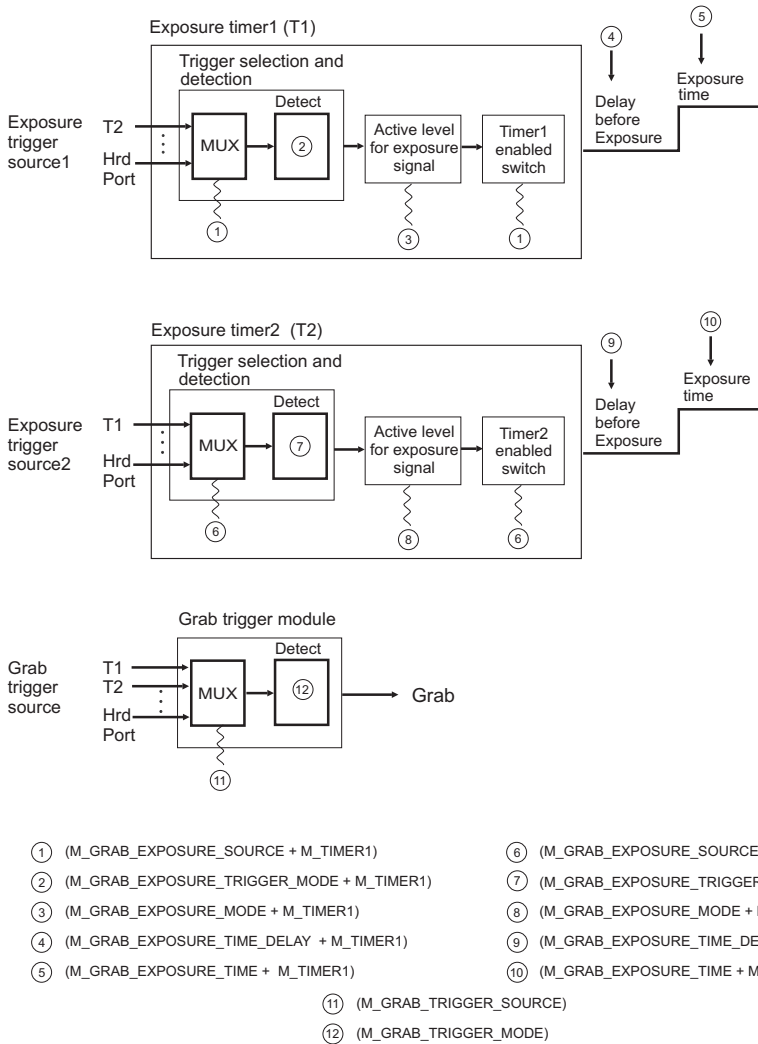
- **MdigControl()** with M\_GRAB\_TRIGGER\_SOURCE selects which signal to use as the source of the trigger (for example, M\_HARDWARE\_PORT0). **MdigControl()** with M\_GRAB\_TRIGGER\_MODE, selects the trigger detection method (for example, trigger on the rising edge of the signal).
- If the exposure 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.

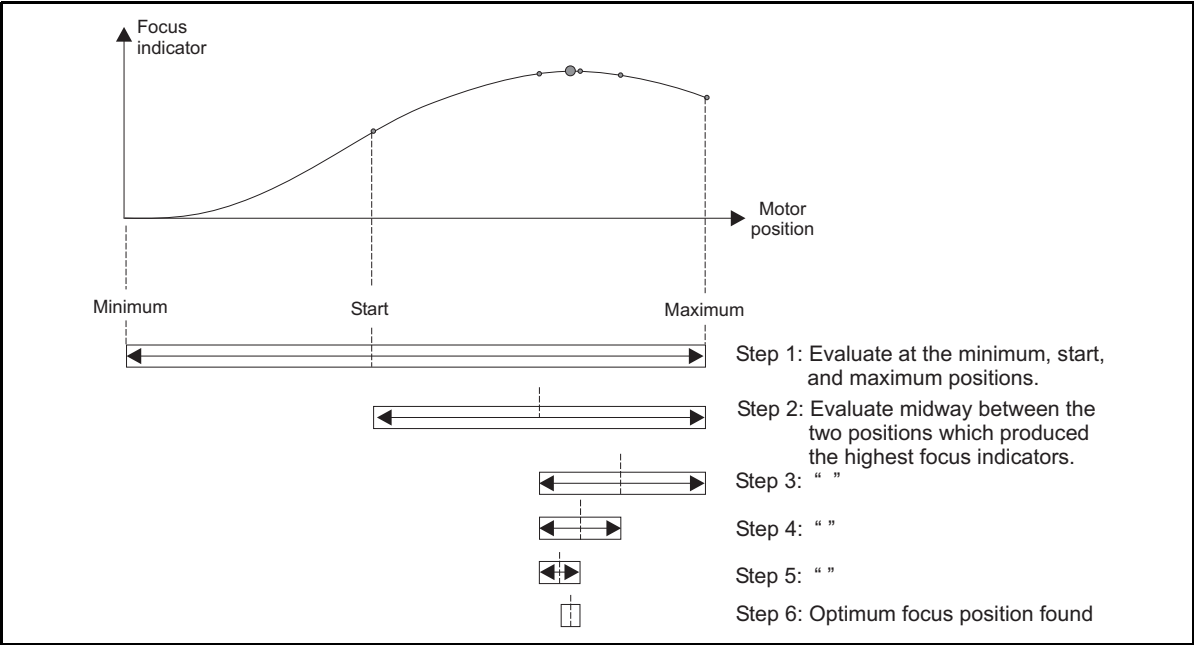
By default, *MdigFocus()* subsamples and filters each grabbed image before analyzing it. This makes it easier to analyze the image. If necessary, you can specify that the subsampling and/or filtering be skipped. Skipping these operations will result in a more accurate analysis of the image's focus quality. 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.

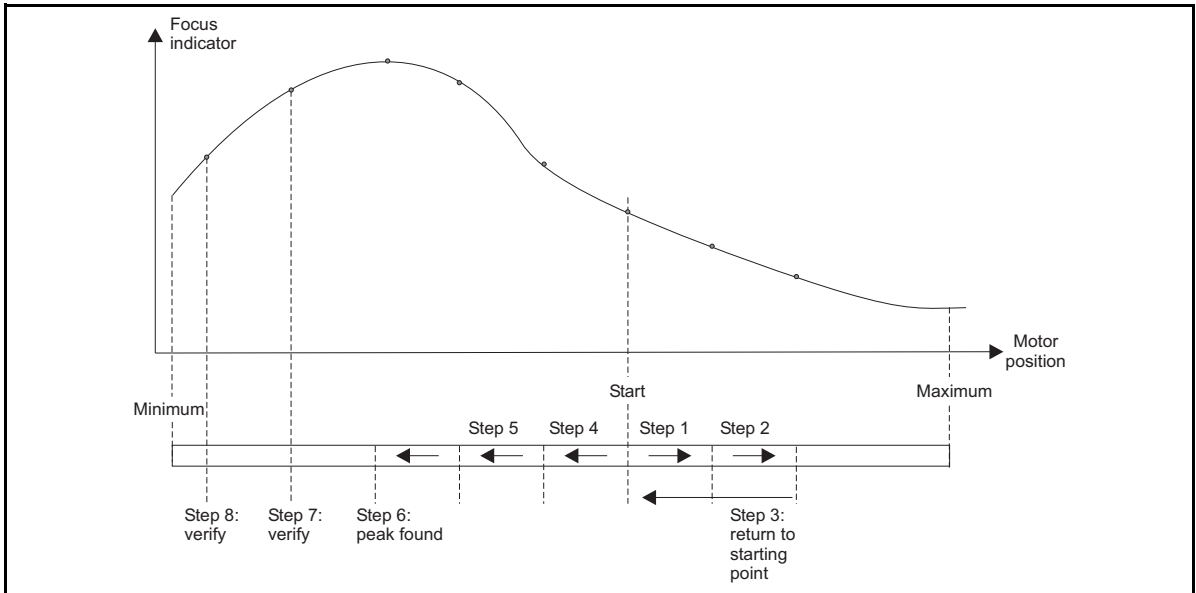


In general, the bisection strategy processes the fewest amount of images. However, it is most sensitive to noise and requires that the lens motor travel the greatest distance.

**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



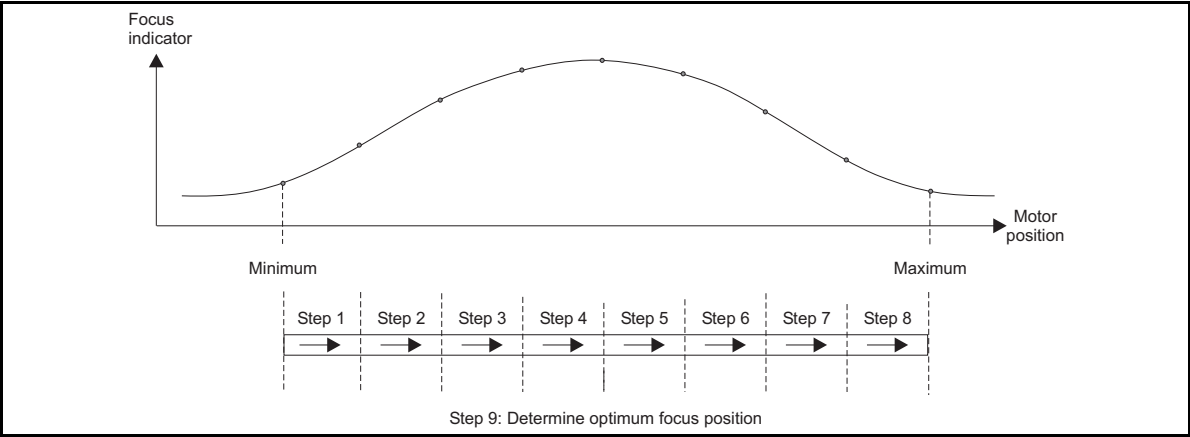
values is found, the next two positions are scanned to make sure the peak is truly the optimum. If necessary, you can change the number of positions used to verify a peak.



The refocus strategy is the best strategy to use when the current focus position is close to optimum.

Scan-All strategy

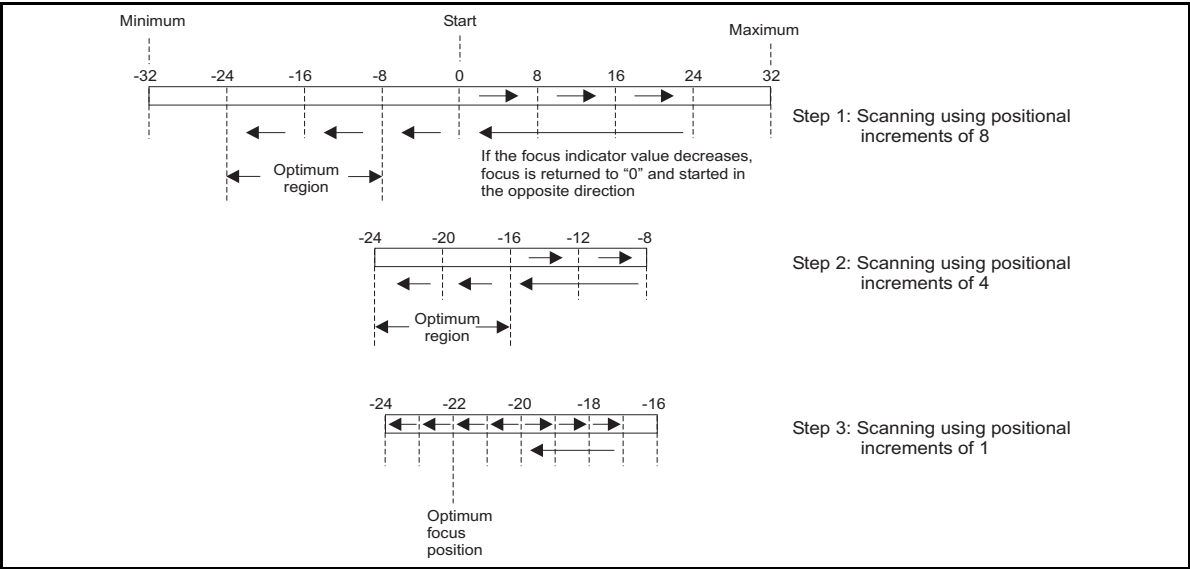
The scan-all strategy scans, by 1, all positions between the minimum and maximum and returns the position which produced the highest focus indicator.



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**

# 22

## **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 should 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**

# 23

## **JPEG and JPEG2000 compression**

This chapter describes how to compress and decompress images.

## Introduction

---

MIL allows you to compress and decompress images and sequences. Compression allows you to store more images in memory than would normally be possible. In addition, compression allows images to be transferred more quickly, since it reduces the amount of data that must be transferred. MIL supports both lossy and lossless JPEG and JPEG 2000 compression algorithms.

- ❖ MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression.

<b>JPEG lossless</b>	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.
<b>JPEG lossy</b>	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.
<b>Interlaced JPEG</b>	MIL can perform a JPEG compression such that the image data is stored in separate fields. This is referred to as an <i>interlaced JPEG compression</i> . Unless otherwise stated, everything that applies to a JPEG compression also applies to an interlaced JPEG compression.
<b>JPEG2000 lossless and lossy</b>	JPEG2000 is another standard for image compression. When compared with regular JPEG lossy compression at the same compression ratio, JPEG2000 lossy compression provides better image quality. JPEG2000 lossless compression permits a smaller buffer size while retaining the same image quality as regular JPEG lossless compression. The JPEG2000 lossy and lossless algorithms can compress 8- or 16-bit buffers with 1 or 3 bands (RGB or YUV). For a more detailed description of supported features, see the section, <i>JPEG2000</i> .
<b>Control options</b>	MIL allows you to control certain aspects of a compression. For example, you can use your own compression tables, although the default tables are suitable for most applications.

## 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.
  - ❖ Compressed buffers that are created using the *MbufCreate...()* functions should not be used as the destination buffer of a MIL function. If a buffer with an `M_COMPRESS` specifier is used as a source buffer for an operation, the data will be decompressed depending on the attributes of the destination buffer.
2. If necessary, change the control settings of the buffer, using *MbufControl()*.

For example, for a JPEG or JPEG 2000 lossy compression, you might want to change the quantization factor (`M_Q_FACTOR`), which is one of the factors that determine the amount of compression. The default value of the quantization factor is 50; setting a lower value will produce marginal improvement in image quality and will result in a larger file size; setting a higher value will produce a smaller file, and therefore a poorer quality image.

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 in a 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. Note that due to the computational complexity of JPEG 2000 compression, grabbing into such a buffer presents a risk of missing frames.

- ❖ Compression operations are optimized when the uncompressed source buffer and the compressed destination buffer are in the same format. Typically, buffers in YUV16 format produce the best compromise for quality and speed.

**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; the same controls must be used for decompression, otherwise the image data will be lost. The only exception to this rule is for JPEG 2000 lossy compression, where you can change the target size of the image (the `M_TARGET_SIZE` control type).

- ❖ Decompression operations are optimized when the compressed source and uncompressed destination buffers are in the same format. Typically, buffers in YUV16 format produce the best compromise for quality and speed.
- ❖ Decompressing a JPEG buffer into a YUV16 packed (YUYV) buffer might accelerate transfer to the display.

**Sequences** When compressing sequences, you can use *MbufImportSequence()* to import a sequence of images from an audio video interleave (AVI) file into separate compressed buffers. You can use *MbufExportSequence()* to export a sequence of compressed image buffers to an AVI file.

**Multi-band buffers, color formats, and control settings - JPEG** When you allocate a multi-band buffer for a JPEG lossy compression, you can specify that the compressed image be stored in an RGB or YUV format. YUV is convenient because most image-viewing software support compressed color images in YUV16 format.

If you are performing a JPEG lossy compression on a YUV image, you can use the `XX_LUMINANCE` and `XX_CHROMINANCE` control types to control the Y band and the U and V bands, respectively. The control types without these suffixes control all bands. See the MIL Command Reference for the list of YUV-specific control types.

When the specified compressed buffer format differs from that of the source image, MIL will internally convert the source image to the specified format before performing the compression.

**Multi-band buffers, color formats, and control settings - JPEG 2000** When you allocate a multi-band buffer for a JPEG 2000 compression, you can specify that the compressed image be stored in an RGB or YUV format. If you are compressing a multi-band buffer in JPEG2000, you can specify different control



settings for each band. To do so, add M\_RED, M\_BLUE, or M\_GREEN to your control type (for a YUV buffer, add M\_Y, M\_U, or M\_V). Using the control type alone or with M\_ALL\_BAND will control all bands.

### Application-specific markers

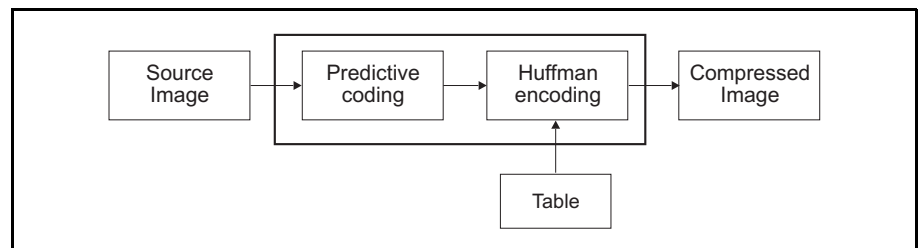
During a compression, MIL adds some application-specific markers to the resulting image. Most other packages will ignore these markers and therefore be able to decompress the file. MIL itself ignores unrecognized markers when it decompresses files.

## 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 *Information technology -- Digital compression and coding of continuous-tone still images: Requirements and guidelines*, which is available from the *International Standards Organization* ([www.iso.ch](http://www.iso.ch)). The section, *Improving results*, summarizes techniques to use to improve compression operations.

### 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.

MIL supports three types of predictive coding: predictor #0 (no predictor), predictor #1 (the “pixel-to-the-left” predictor), and predictor #2 (the “pixel-above” predictor). By default, MIL uses the pixel-to-the-left to predict values, which is suitable for most images. In some applications, you might prefer to use the pixel-above predictor. You can also specify no predictor (predictor #0), but note that in this case, the values after predictive coding will be the same as the original values. This predictor 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. You can specify the predictor with the *MbufControl()* M\_PREDICTOR control type.

**Huffman encoding**

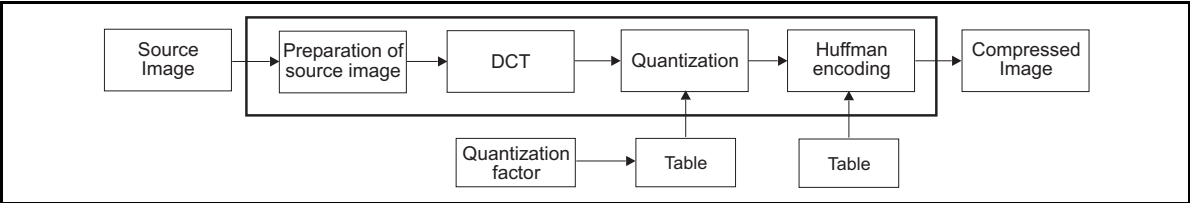
After an image has been predictive coded, Huffman encoding assigns a variable-length “code word” to each value. This code is based on the number of bits by which adjacent values differ. Values are assigned code words according to a DC Huffman table. You can use the default DC Huffman table or you can create your own table. If you want to use your own table, refer to the section *Working with tables*.

**JPEG lossy**

The JPEG lossy algorithm is outlined below. First the source image must be in the correct format before it can be compressed. Although the JPEG algorithm requires signed source data, the algorithm accepts both signed and unsigned data. Initially the algorithm internally treats all data as unsigned. Then a computational shift is performed to set all the values to signed.

After the computational shift, a color conversion is performed if the source and destination buffers are in different formats, for example an RGB source buffer and a YUV destination buffer. Note that conversion to YUV introduces some loss.

Afterwards, 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.

### 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 the *MbufControl()* `M_RESTART_INTERVAL` control type 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 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.

## JPEG2000

---

This section provides a brief overview of the JPEG2000 lossy and lossless algorithms and the control you have over these algorithms. Everything applicable to lossy compression is applicable to lossless compression, unless otherwise stated. In general, you should only change compression controls if you are familiar with the algorithm (except the quantization factor, the `M_Q_FACTOR` control type). The section, *Improving results*, summarizes techniques to use to improve compression operations. For more detailed information about the JPEG2000 algorithm, see [www.jpeg.org](http://www.jpeg.org).

There are two fundamental differences between the JPEG2000 and regular JPEG compression algorithms. The first is the use of a discrete wavelet transform (DWT), instead of a discrete cosine transform. The second is the use of arithmetic encoding instead of Huffman encoding as the entropy encoding technique; arithmetic encoding reduces the number of bits required to encode data, based on how frequently that data occurs in the image.

When compared to regular JPEG compression, JPEG2000 supports much higher ratios of compression without compromising image quality. Note however, that JPEG2000 is best suited for archiving purposes because of the processing time required to compress and decompress images; therefore, JPEG2000 can be used for grabbing, but it presents a risk of missing frames.

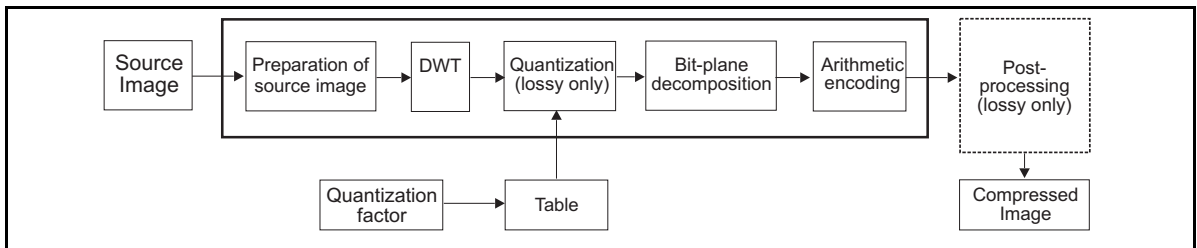
When compressing images with the JPEG 2000 algorithm, MIL supports:

- Custom quantization tables.
- Custom wavelet settings (setting the number of decomposition levels).
- Saving files in JPEG 2000 format with a `*.jp2` extension (includes only the JPEG 2000 bit-stream), as well as saving AVI sequences in JPEG 2000 format.
- Band-specific compression settings.
- Easy control over image size and quality (using the `M_Q_FACTOR` control type).
- Specifying a target size for the compressed image (when performing a JPEG 2000 lossy compression).

Note that MIL does not support the following JPEG 2000 features:

- Progressive encoding/decoding (layering).
- Random bit-stream access (tiling and/or ROI coding).
- The JP2 file format optional features.
- Error resilience (not typically used in imaging applications).

The JPEG2000 compression consists of the following steps:



1. Preparation of source image.
2. Discrete wavelet transform.
3. Quantization (lossy only).
4. Bit-plane decomposition.
5. Arithmetic encoding.
6. Post-processing (lossy only).

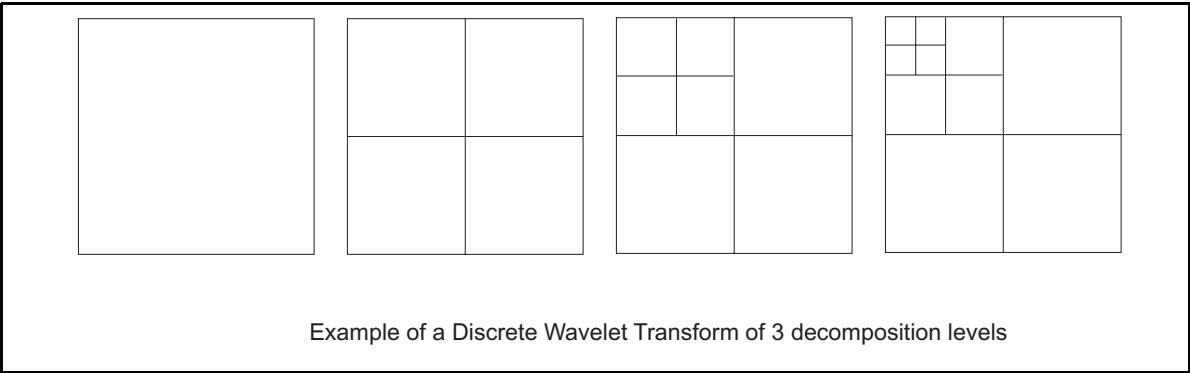
### **Preparation of source image**

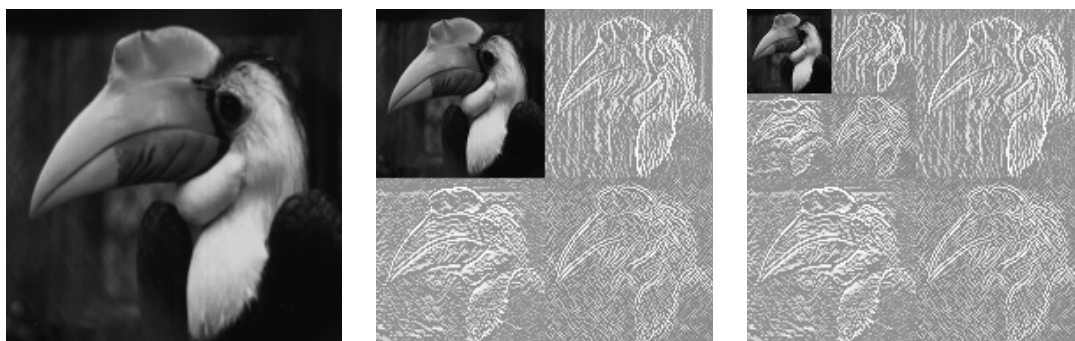
Although the JPEG2000 algorithm requires signed source data, the algorithm accepts both signed and unsigned data. Initially the algorithm internally treats all data as unsigned. Then a computational shift is performed to set all the values to signed.

After the computational shift, a color conversion is performed if the source and destination buffers are in different formats, for example an RGB source buffer and a YUV destination buffer. Note that conversion from RGB to YUV naturally introduces some loss.

**Discrete wavelet transform (DWT)**

After preparation of the source image, a lossy or lossless discrete wavelet transform (DWT) is passed on the image. The discrete wavelet transform both subsamples and spatially filters the image. Depending on the type of compression, the DWT uses one of two sets of filters: for lossy compression, a Daubechies 9-tap/7-tap filter is used, while a 5-tap/3-tap filter is used for lossless compression. The DWT subsamples and then separates the data into high frequency areas and low frequency areas; these areas are referred to as *sub-bands*. Each iteration, which consists of one pass of both the high and low frequency filters in both the horizontal and vertical directions, results in four sub-bands. Subsequent iterations of the transform are always passed on the top-left (low frequency) sub-band. MIL refers to each iteration of the DWT as a *decomposition level*.





Example of a Discrete Wavelet Transform of 2 decomposition levels

By default, the wavelet transform is complete when the last pass renders four sub-bands that are, if possible,  $16 \times 16$  pixels. Depending on the size of your image,  $16 \times 16$  might not be possible. In this case, the wavelet transform would pass until the last iteration renders sub-bands that are the next largest size closest to  $16 \times 16$ . For most compressions, the default number of decomposition levels produce good results.

#### Changing the size of the smallest sub-band

For JPEG 2000 lossy, you can, however, change the number of decomposition levels to try to improve the compression (depending on the image data) using the *MbufControl()* `M_DECOMPOSITION_LEVEL` control type, but this can also introduce more error. By changing the number of decomposition levels, you indirectly control the size of the smallest sub-band produced by the transform. For example, if your original image is  $480 \times 480$ , by default the number of decomposition levels is 4, and the size of the four smallest sub-bands will be  $30 \times 30$ . If you want the smallest sub-bands to be  $15 \times 15$ , you can set the `M_DECOMPOSITION_LEVEL` control type to 5. Note that for JPEG 2000 lossless, changing the default size of the smallest sub-band usually has no benefit.

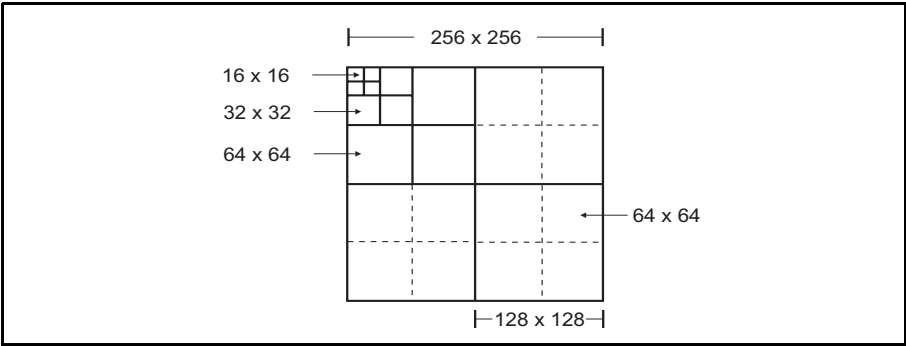
- ❖ It is recommended that you change the number of decomposition levels only after trying the other techniques described in the section, *Improving results*.

After the wavelet transform, each pixel in the original image becomes a *wavelet coefficient*. There will always be as many wavelet coefficients as there were pixels in the original image.

- ❖ After the wavelet transform is applied, MIL no longer considers the data as an image.

Segmentation

Sub-bands that are larger than 64 x 64 are further segmented into as many 64 x 64 regions as possible. These regions are referred to as *code-blocks*. Smaller sub-bands are not segmented, and each contains one code-block. In the diagram below, the sub-band that is 128 x 128 contains four code-blocks, while the sub-bands that are 64 x 64, 32 x 32, and 16 x 16 each contain one code-block.



Segmentation optimizes compression by grouping areas with similar values; when the values to compress are similar, the result is a better compression ratio. Segmentation also optimizes access to Host memory, thereby improving the performance of the quantization (for JPEG2000 lossy compression), bit-plane decomposition, and arithmetic encoding steps in the algorithm.

Quantization

During JPEG 2000 lossy compression, the wavelet coefficients contained in each code-block are quantized based on both the sub-band in which they fall, and an entry in the quantization table. Quantization multiplies each wavelet coefficient by its sub-band’s corresponding entry, the *quantization coefficient*, in the table. The sub-band’s rank in significance, which is illustrated below, determines which entry in the quantization table is used. Note that the top-left sub-band is the most significant.



0	1	4
2	3	
5		6

❖ JPEG 2000 lossless compression does not use the quantization step.

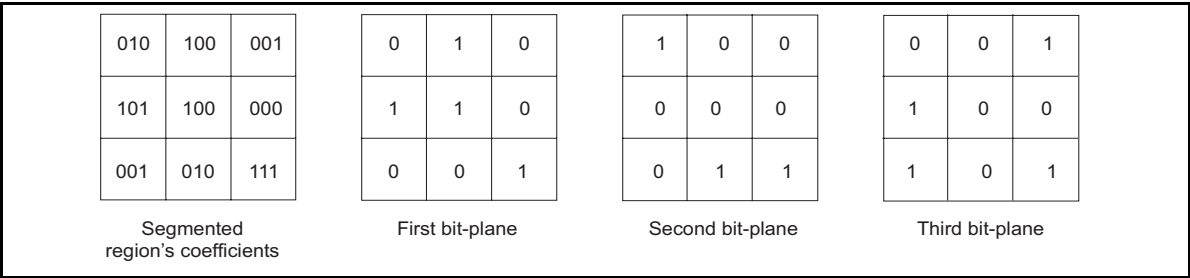
MIL automatically determines the values for the quantization table based on the number of sub-bands resulting from the DWT transform; that is, all images of the same depth that have five decomposition levels will use the same quantization table. The first entry (for sub-band 0) always has the largest quantization coefficient.

When you adjust the quantization factor (*MbufControl()* M\_Q\_FACTOR control type), MIL scales all the quantization coefficients in the quantization table according to the specified factor. If you want more control, you can pass your own custom quantization table. To establish good values for a custom quantization table, you can inquire the default quantization coefficients. See the section *Working with tables* for more detailed information.

## Decomposition

After the wavelet coefficients are multiplied by the quantization coefficients, the code-blocks are decomposed into *bit-planes*. A bit-plane contains the *n*th bit of all the multiplied wavelet coefficients that occupy a specific code-block. Decomposition into bit-planes is necessary for arithmetic encoding and post-processing.

A specific code-block might not have the same number of bit-planes as another code-block. Decomposition starts from the most-significant non-zero bit-plane of a code-block, and all subsequent bit-planes are decomposed, even eventual zero bit-planes. For example, the code-block in the diagram below contains coefficients that are 3 bits, and therefore will be decomposed into 3 bit-planes.



Arithmetic encoding

The bit-planes are then passed to the arithmetic encoder. The arithmetic encoder analyzes the bit-plane data to find redundancies, in other words, patterns of bits that occur frequently in the bit-planes. Using this information, the arithmetic encoder replaces repeating bit patterns, also called symbols, by shorter bit sequences. The more often a symbol occurs in the bit-plane data, the fewer bits into which that symbol will be coded.

During lossless compression, the arithmetic encoder processes each bit-plane to produce a continuous bit-stream. Since arithmetic encoding is the final stage of compression, this bit-stream is actually the compressed image.

During lossy compression, the arithmetic encoder produces several independent bit-streams; the post-processing step determines which bit-streams are concatenated in the final image.

Post-processing

Once the most-significant bit-plane of a code-block is encoded, its size in bytes is stored in memory, and MIL calculates how much error would be present in the decompressed image if the remaining bit-planes, for that code-block, were not arithmetically encoded.

Once the second most-significant bit-plane of the code-block is arithmetically encoded, MIL calculates the error introduced in the decompressed image if only the first two bit-planes were compressed. As each remaining bit-plane is encoded, MIL repeats the same calculation until all the bit-planes have been processed.

As an example only, the values in the following table represent the amount of distortion (error) with respect to the size in bytes of the encoded bit-planes of one code-block.

Bit-plane (in order of significance)	Compressed size in bytes	Distortion
1	5	58817925.11
2	54	34495584.39
3	159	11160557.81
4	296	2633195.13
5	436	594555.28
6	574	94112.98
7	717	21139.81
8	863	4080.74
9	980	0.00

Then, the best compromise between decompressed image quality and requested size in bytes (*MbufControl()* M\_TARGET\_SIZE) is determined. Any encoded bit-planes that are not required are discarded after the best compressed image is determined for the specified size. The remaining encoded bit-planes are then concatenated in the final image.

Determining an appropriate target size will require some trial and error, but a logical guess can be calculated by dividing the image's original size (in bytes) by the required compression ratio. For example, if you want to compress a 256-byte image by a factor of 16:1, you would specify a target size of 16 bytes.

- ❖ Post-processing is not performed in JPEG2000 lossless, and none of the encoded bit-planes are discarded; therefore, you cannot specify a target buffer size when performing a lossless compression.

## Improving results

---

If the defaults do not meet your application requirements, you can try to improve your compression ratio using the following techniques. We recommend trying these techniques in the order they appear.

- ❖ Regardless of the type of your compression operation, you should first remove extraneous noise from the image (if possible) using MIL processing functions.

For JPEG lossy compression:

- Allocate a YUV buffer for compression.
- Increase the quantization factor with the *MbufControl()* M\_Q\_FACTOR control type.
- Decrease the restart interval.
- Change the quantization table. See the section, *Working with tables*.
- Change the Huffman table. See the section, *Working with tables*.

For JPEG lossless compression:

- Try the other supported predictors with the *MbufControl()* M\_PREDICTOR control type.
- Decrease the restart interval.

For JPEG 2000 lossy compression:

- Allocate a YUV buffer for compression.
- Increase the quantization factor with the *MbufControl()* M\_Q\_FACTOR control type.
- Decrease the target size with the *MbufControl()* M\_TARGET\_SIZE control type.
- Change the number of decomposition levels of the DWT with the *MbufControl()* M\_DECOMPOSITION\_LEVEL control type.

- Pass another quantization table. See the section, *Working with tables*.

For JPEG 2000 lossless compression:

- Change the number of decomposition levels of the DWT with the *MbufControl()* `M_DECOMPOSITION_LEVEL` control type.

## Working with tables

---

In some applications, the default quantization or Huffman tables might not be suitable. MIL allows you to create your own. You can inquire the default table to help you determine appropriate values. You might have to select values by trial and error to determine the best ones for your application.

- ❖ For JPEG 2000 compression, quantization multiplies coefficients, while in JPEG compression, quantization divides values.

Whether you are inquiring the default tables or customizing your own, you must allocate arrays that are large enough to contain the data. The table below lists the tables that you can manipulate and their required size for each compression type.

Compression type	Table type	Buffer type, size, and attribute
JPEG lossless	DC Huffman	1-dimensional, 8+M_UNSIGNED, 28 entries, M_ARRAY
JPEG lossy	DC Huffman	1-dimensional, 8+M_UNSIGNED, 28 entries, M_ARRAY
	AC Huffman	1-dimensional, 8+M_UNSIGNED, 178 entries, M_ARRAY
	Quantization	2-dimensional, 8+M_UNSIGNED, 8 x 8 entries, M_ARRAY
JPEG 2000 lossy	Quantization	1-dimensional, 32+M_FLOAT, 3N + 1 entries, where N is the number of decomposition levels, M_ARRAY

### Inquiring values in default tables

Inquiring the default values of a table is useful to determine values for your custom tables. The steps below outline this procedure.

1. First, inquire the MIL identifier of the default table using *MbufInquire()*. Then, inquire the size of the table using the same function.
2. Allocate a user array of the appropriate size for storing the default table values.

3. Get the values from the inquired table in Step 1 into the user array using *MbufGet()*.

❖ You can only inquire all values in the table. You cannot inquire specific table entries.

### **Using your own table**

To use your own table:

1. Allocate a buffer with an M\_ARRAY attribute and of the required data type specified in the table earlier in this section.
2. Transfer the user array containing the custom table values to the array buffer, using *MbufPut1d()* or *MbufPut2d()*, depending on the type of table.
3. Associate the M\_ARRAY buffer to the required M\_COMPRESS image buffer, using the *MbufControl()* control types specific to your table. Specifying these control types as-is, or combined with M\_ALL\_BAND, controls all bands.

For a JPEG2000 compression, you can associate a different table to each band of a multi-band buffer. To do so, add M\_RED, M\_BLUE, or M\_GREEN to your control type (for a YUV buffer, add M\_Y, M\_U, or M\_V).

For JPEG lossy compressions of YUV images, use the XX\_LUMINANCE and XX\_CHROMINANCE control types. The control types without these suffixes control all bands.

❖ If you set the M\_Q\_FACTOR control type after specifying a custom table, the custom table will be scaled.

Chapter

24

# **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 17: 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

# 25

## **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	<p>Under multi-thread operating systems, you can create as many threads as you require. The MIL commands in any thread are executed as follows:</p> <ul style="list-style-type: none"> <li>• If the target processor is the Host CPU, processing in each thread is determined by the operating system.</li> <li>• 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.</li> </ul>
MIL application context	<p>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 <i>MappAlloc()</i>). Its purpose is to handle the context of the new thread, such as error reporting.</p> <p>You can have the thread's application initialized to its initial state by allocating a new application using <i>MappAlloc()</i>; note that this must be the first call to MIL in the thread.</p>
Synchronization	<p>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.</p> <p>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, <i>MbufGet()</i> after an <i>MdigGrab()</i>), MIL automatically synchronizes the threads to force the Host to wait for completion of the earlier command(s).</p> <p>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, <i>MappControlThread()</i>, to control the flow of such commands.</p>

**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 the function is executed.

The most common way to check for errors is to use the *MappGetError()* function. In multi-thread environments, an *MappGetError()* call returns the error of the current thread or, if none, checks for errors in the other threads running MIL. To return only errors in the current thread, add M\_THREAD\_CURRENT to the **ErrorType** parameter (M\_CURRENT+M\_THREAD\_CURRENT).

**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:    %41d.\n", NumberOfTopLeft);
    printf("Bottom left iterations done:  %41d.\n", NumberOfBotLeft);
    printf("Top right iterations done:         %41d.\n", NumberOfTopRight);
    printf("Bottom right iterations done:       %41d.\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);
}

/* Top left and top right functions (Add an offset): */
/* ----- */
unsigned long MFTYPE TopThread(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;
    MIL_ID EventEndBotId   = ((THREAD_PARAM *) TParam)->EventEndBotId;
    char Text[STRING_LENHT_MAX] = STRING_TOP;
    long Exit=0;

    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);
}

```

(cont...)

```

/* Minimize impact of Window's thread scheduling adjustment mechanisms. */
BalanceThreadScheduling();

/* Wait before freeing MIL objects that all threads are finished. */
while (MappControlThread(EventEndBotId, M_EVENT_STATE,
                        M_DEFAULT, M_NULL) == M_SIGNED)
;

/* Make sure exit of thread is synchronized with HOST. */
MappControlThread(EventEndId, M_EVENT_SET, M_NOT_SIGNED, M_NULL);
return(1L);
}

/* 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);
    }
}
(cont...)

```

```

/* Make sure that exit of thread is synchronized with HOST. */
MappControlThread(EventEndId, M_EVENT_SET, M_NOT_SIGNED, M_NULL);
return(1L);
}

/* 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_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);
}

```

**Chapter**

# 26

## **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 to 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.

```

/* 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);
}

```

**Chapter**

**27**

# **Distribution and licensing**

This chapter presents how to distribute and license MIL applications.

## **Distribution of MIL-based applications**

---

There are some details that you must consider before you can distribute a MIL-based application, either for your customers' use or for your own. When doing this distribution, you must redistribute MIL run-time DLLs and device drivers. In addition, when using a MIL processing module, you must purchase an appropriate run-time license, otherwise, your MIL-based application will stop functioning after 30 days. This chapter deals with the different ways to distribute your application, purchase a license, and handle licensing concerns for your customers.

- ❖ When distributing an application that uses MIL-Lite, you do not require a run-time license.

## **Redistributing MIL run-time DLL files and device drivers with your application**

---

To distribute your MIL application, you will have to redistribute MIL run-time DLL files and the necessary device drivers with your application.

It is important to remember that only one copy of MIL can be present on a computer at a time. When installing an application that uses MIL or MIL-Lite on a computer with a more recent or equally recent version of MIL or MIL-Lite, the application's set-up program must not install MIL. The application should use the version of MIL already installed on the computer.

Conversely, if the version of MIL or MIL-Lite on the computer is less recent than the application's required version, a decision must be made. Either the version of MIL or MIL-Lite already on the computer must be removed before installing the newer version, or the application cannot be installed on that computer.

### **Redistributing directly from the MIL CD**

If the target computer (on which you want to install the MIL run-time DLLs and device drivers) is 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.

### Redistributing using your own setup program

Alternatively, to redistribute your MIL run-time DLLs and device drivers, you can have your application's setup program call MIL's redistribution setup program in one of two ways:

- **Normal redistribution.** Prompts your customer for setup information.
- **Silent redistribution.** Uses a custom response file instead of prompting your customer for information.

These are described in detail in the next sections.

## Normal redistribution using your custom CD

---

To distribute your MIL run-time DLLs and device drivers, you can have your application's setup program call MIL's redistribution setup program. If you use the normal redistribution mechanism, your customer will be prompted for information during the setup. To use your setup program, do as follows:

1. Copy the `\REDIST` directory from the MIL CD to your installation directory.
2. If you want to save space on your target CD, you can remove certain components of the `\REDIST` directory if they are not required for your particular application. For example, the `\REDIST\GENESIS` and `REDIST\MATROX\DRIVERS\GENESIS` directories can be removed if you are not using a Matrox Genesis board. Refer to the `redist.txt` file for more examples.
3. Note that the driver-specific `*.inf` files must be modified. These files can be found in the appropriate driver directory located in the `\REDIST` directory. If you do not modify these files there will be a reference to MIL and the MIL CD each time a driver is being installed. You would likely want to replace these references with references to your product and CD. As well, under Windows 98/Me/2000, it is only possible to install the Matrox frame grabber drivers if the boards are physically present in the computer at the time of installation.
4. Have your redistribution program call the `setup.exe` file in the `\REDIST\MATROX` directory. The `setup.exe` program installs the required run-time MIL DLL files and device drivers on your client's computer. An example call would be:

```
C:\Redist\Matrox\setup.exe REDISTRIBUTION
```

## Silent redistribution

---

A silent redistribution does not prompt your user for any MIL information; instead, it uses a response file to provide the necessary setup parameters for the intended computer. You would use a silent redistribution when you are including MIL within your application and you do not want to have any Matrox Imaging setup dialog boxes appear. You could also use a silent redistribution if you wanted to control the setup parameters for your client.

It is not possible to create a response file for only a few of the setup parameters and ask the customer for the rest of the setup information. If you are going to use a response file, you must answer all of the setup questions in the response file.

To redistribute the MIL run-time DLLs and device drivers using a silent redistribution:

1. Follow the steps for a normal MIL redistribution.
2. Create a response file that provides the setup questions with the answers you want. Refer to the next subsection for details.
3. Have your redistribution program call the *redist.exe* program with the additional 'REDISTRIBUTION RESPONSEFILE = "<filename>" -s' parameter to specify the name and the location of your response file. For example:

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

- ❖ In a silent redistribution, if a copy of MIL is found on the target computer, the installation will not occur. MIL will not be overwritten or uninstalled. In this instance, there will be an error code in the registry key *HKEY\_CURRENT\_USER\SOFTWARE\MATROX\ENTRY: STATUS*, signifying that a previous version of MIL has been found.

### Response file parameters

The response file's format parameters follow; the error codes can be found in the *Redist.txt* file. The order in which you place the parameters in the response file is not important. What is important is that each parameter, and its settings, is written

as one unbroken line of code, with a carriage return at the end of the line. Any errors in the response file will cause the silent distribution to stop. Here is an example of a typical response file for an installation under Windows 98/Me.

```
SILENTMODE = 1
INSTALLATION_DIRECTORY = C:\Program Files\Matrox Imaging
DRIVER1 = METEOR_II
DRIVER2 = ORION
DMA_METEOR_II = 4
DMA_ORION = 4
DMA_VGA = 0
MGA_DRIVER = INSTALL
SUPERPRO = IGNORE
```

These are the potential parameters for a response file.

- SILENTMODE.

This parameter designates silent installation and is a required parameter. SILENTMODE must be set to 1.

- INSTALLATION\_DIRECTORY.

This parameter specifies the target installation directory and is a required parameter.

- DRIVER $x$ .

This parameter designates the drivers that are to be installed. Replace  $x$  with the number to assign to the driver. More than one driver can be installed, but each driver must be assigned a different number; these numbers do not have to be consecutive. The available settings for this parameter are: CORONA\_II, METEOR\_II, METEOR\_II\_1394, ORION, and VGA. Use the METEOR\_II setting for both the Matrox Meteor-II /Standard and Meteor-II /Multi-Channel boards.

GENESIS, METEOR\_II\_DIG, and METEOR\_II\_CL are also possible settings for a response file, but in this case the redistribution will not be completely silent. Completely silent redistribution is not possible with the Matrox Genesis, Meteor-II /Digital, and Meteor-II /Camera Link boards because they call on another library which does not support silent redistribution.

Also, note that for backwards compatibility, response file settings METEOR2 and METEOR2D can also be used for **DRIVERx**. However, we recommend using the new settings, METEOR\_II and METEOR\_II\_DIG.

- DMA\_MEMORY\_SIZE.

This parameter is used only for Windows NT/2000 and designates the non-paged memory size in Mbytes. The memory set aside is not board-specific. This must be a minimum of one Mbyte; if you only have a graphics controller, or a graphics controller and a Matrox Meteor-II /1394 board, you can set this parameter to a minimum of 0 Mbytes.

- DMA\_CORONA\_II, DMA\_METEOR\_II, DMA\_ORION, DMA\_VGA.

These parameters are used with Windows 98/Me and they designate the non-paged memory sizes in Mbytes for each board. This must be done for each board installed under Windows 98/Me except for Matrox Meteor-II /1394.

The **DMA\_VGA** parameter designates the non-paged memory sizes in Mbytes for the graphics controller. **DMA\_VGA** must have a value if you are creating a response file under Windows 98/Me; in general, this should be set to zero. This memory can be used to store data across processes (using MbufCreate()), although this should be used with caution; it is generally better to leave memory control to MIL.

- *MGA\_DRIVER*.

This parameter has two settings: INSTALL and IGNORE. The INSTALL setting will install the MGA driver in silent mode, while the IGNORE setting will not install the MGA driver.

### **Warning!**

If the MGA driver is installed and a previous version of the MGA driver is on the computer, the installation program will overwrite the previous version rather than uninstall it.

- SUPERPRO.

This parameter determines whether or not the SuperPro driver is installed. SuperPro is required to communicate with the MIL run-time hardware license-key, discussed in the next section; it is not required for MIL-Lite. There are four settings for the SuperPro parameter: INSTALLPARALLEL,



INSTALLUSB, INSTALLBOTH and IGNORE. The INSTALL<sub>xxx</sub> settings will install the SuperPro driver for the appropriate port, while the IGNORE setting will not install it. Note that the USB port is not supported under Windows NT.

Note, in the *RESPONSE.txt* provided, you can comment out or disable a setting by preceding the line with two slashes: // (putting "/" anywhere on the line will also disable the setting).

### Debugging the response file

The response file needs to be error-free. Some common errors to avoid are:

- **Missing underscores.** The underscores must be present in the parameter name and the parameter settings.
- **Incorrect case.** Every character in a parameter or setting must be uppercase, unless it is part of a file path.
- **Incorrect parameter setting.** If a setting is defined in units, those units must not be written in the response.txt file (for example, DMA\_CORONA\_II = 4, not DMA\_CORONA\_II = 4Mbytes)
- **Broken line of code.** Each parameter and its setting must be written as one unbroken line of code finished by a carriage return.
- **Unknown location for response file.** The response file is not in the location specified by your redistrib.exe command line parameter **Responsefile**.
- **Unknown location for \Redist\.** The \REDIST\ directory is not in the correct location on the installation CD. Make sure when writing the redistribution program that the path is correct when calling the *redist.exe*.
- **Missing INF and DLL files (for Windows 98/Me).** The *MtxImage.inf* and *Mtximgci.dll* files are not in the root of the installation CD. Make sure you copy them with the \REDIST\ directory.

To debug your response file, you will have to run the redistribution executable file, and fix any errors that occur. Should an error occur, there will be an error code in the registry key, *HKEY\_CURRENT\_USER\SOFTWARE\MATROX ENTRY: STATUS*. Refer to the *redist.txt*, where the entire error code list is stored.

**Important notes for Windows 98/Me users**

- Ensure that the O/S has a standard VGA display driver installed prior to the installation of MGA drivers.
- The MGA drivers are only installed for boards that have an on-board graphics controller (Matrox Corona-II and Matrox Orion only).
- To avoid an MGA diagnostic message when installing the MGA driver, set the RUN\_DIAG parameter to NO in the *MGA.ini* file prior to the installation. This file is located in the \REDIST\MGADRV\WIN98\ directory.

**Important notes for Windows NT/2000 users**

- Ensure that the O/S has a standard VGA display driver installed prior to the installation of the MGA drivers.
- The MGA drivers are only installed for boards that have an on-board graphics controller (Matrox Corona-II, Matrox Genesis and Matrox Orion only).

**Uninstallation**

---

To uninstall previous versions of MIL, or other Matrox Imaging Products, call the Matrox Imaging Products Uninstallation program and specify the products that you want to uninstall. To make the uninstallation silent, make the call with the silent mode parameter. The following command-line parameters are available:

Parameter	Description
/s	Silent mode (no dialog box).
/n	No reboot at the end of uninstallation.
/m	UnInstall MIL or MIL-Lite.
/a	UnInstall ActiveMIL or ActiveMIL-Lite.
/i	UnInstall Intellicam.
/h	Display parameters.

For example, if you want to uninstall MIL and ActiveMIL in silent mode without rebooting your system, call:

C:\Winnt\UnInstallMIP.exe/s/n/m/a

- ❖ The Matrox Imaging Products Uninstallation program is copied to the Windows directory at the time of installation.

You will find the uninstallation status in the following registry key:  
*HKEY\_CURRENT\_USER\ SOFTWARE\ MATROX ENTRY:  
 UNINSTALL\_STATUS*

To see the error codes, refer to the *redist.txt*.

## **MIL and MIL-Lite licenses**

---

MIL and MIL-Lite have different licensing terms and mechanisms.

When you purchase MIL-Lite, you receive a development license and a registration number. MIL-Lite also comes with a royalty-free run-time license, which allows you to redistribute applications based on MIL-Lite without paying additional royalty fees.

With MIL, you can purchase two types of permanent licenses: a development license and/or a run-time license. Licenses are always verified when a MIL application is allocated, as well as when the application is running; this verification has negligible overhead. You can use a temporary license while waiting for a permanent license.

Please refer to the Matrox Software License Agreement for the legal provisions of using/redistributing MIL or MIL-Lite.

The rest of this chapter deals exclusively with MIL and its licensing mechanisms.

### **Development license**

A development license permits you to run and debug MIL-based applications with Microsoft Visual C++ or Visual Basic. For the license to take effect, a development hardware license-key must be attached to your computer's parallel or USB port. A development hardware license-key is included with the MIL starter or additional developer package.

### **Temporary license**

A temporary license is a 30-day development license. This type of license is most useful for developers who are performing a development or redistribution installation while waiting for a permanent license.

The temporary license is used when MIL is first executed, unless a development license is already present. However, a temporary license only lasts 30 days from the first launch of MIL on the computer. After allocating an application, if the temporary license is used, a dialog box appears indicating the number of days until the temporary license expires.

If a development license is removed, a temporary license dialog box will appear indicating the number of days until the license expires, if it is less than 30 days from the first launch of MIL on the computer. If it is more than 30 days, a dialog box will appear asking you if you want to purchase a run-time license. If you do not purchase a run-time license, you won't be able to use the application since MIL will have been disabled.

- ❖ Once the temporary license has expired, it is not possible to prolong or restart the 30-day trial period. Attempting to alter the computer's calendar before the temporary license expires will immediately disable MIL. In that event, MIL can only be re-used once an appropriate license is installed.

### **Run-time license**

A run-time license is required for each computer that is going to run a MIL-based application. It does not allow for development (debugging) of MIL applications.

- ❖ MIL-Lite applications do not require a run-time license. They can be distributed royalty free.

MIL run-time licenses are purchased according to the requirements of the application. A MIL run-time license consists of different MIL modules, grouped in *packages*, for specific uses. Consult the MIL datasheet or contact Matrox Imaging sales or the local representative regarding the available run-time packages.

A MIL run-time license can be in one of two formats:

- A hardware license-key.
- A software license-key.

If you want to, you can hide the MIL license from your customer using *Gencode*, a modified version of the MIL License Manager. Gencode is located in the same directory as the MIL License Manager.

## Hardware license-key

---

A hardware license-key is a type of run-time license. For the license to take effect, the key must be attached to the parallel or USB port of your computer. This license can be bought by calling Matrox Imaging or your local representative.

If you decide that another computer needs to run a MIL application, you can shift the hardware license-key. This will allow the second computer to use MIL, but it will mean that the first computer will have the temporary license dialog box appearing again. If this is done more than 30 days after MIL was first used on the computer, MIL will be disabled.

To have MIL run-time DLLs recognize the hardware license-key, you must install the SuperPro driver for the hardware license-key.

To install the hardware license-key driver, if you did not install it when you were installing MIL, use one of the following procedures:

- **For Windows NT:** Call the `\MATROX\DRIVERS\SUPERPRO\WINNT\SETUPX86.EXE` program.
- **For Windows 98:** Call the `\MATROX\DRIVERS\SUPERPRO\WIN98\SETUPW9X.EXE` program if you are using a parallel port. If you want to use a USB port, call `\SETUPW9X.EXE /USB` from the `\MATROX\DRIVERS\SUPERPRO\WIN98` directory. After calling the appropriate program, select the **Install Sentinel Driver** menu command.
- **For Windows 2000/Me:** Call the `\MATROX\DRIVERS\SUPERPRO\WDM\RAINBOWSSD539.EXE` and select the **Install Sentinel Driver** menu command.

For more information about Sentinel driver installation for Windows 98/NT, refer to the *Readme.txt* file in the `\MATROX\DRIVERS\SUPERPRO\WINNT` directory. For Windows 2000/Me, refer to the `\DRIVERINFO.HTM` file located in the `\MATROX\DRIVERS\SUPERPRO\WDM` directory. These files also include information on how to install the Sentinel Driver silently.

## Software license-key

---

A software license-key is another type of run-time license. The software license-key uses a fingerprint, generated from one of several components on the target computer, to produce the license.

You can obtain a license as follows:

1. Install your MIL-based application on the target computer as described in Chapter 1. A temporary license is assigned to your computer, allowing use of MIL for 30 days. Each time your application runs MIL, a dialog box appears indicating the number of days until the temporary license expires and a setting to obtain a permanent run-time license is displayed. If you click "Yes" to get the license, the *MIL License Manager* program will be launched. The License Manager program will also be launched if you attempt to use a MIL function that is not available in the purchased package.
2. In the License Manager program, choose any combination of the different MIL packages for purchase, by enabling the required package's check box.
  - ❖ Since the MIL Image Analysis package is included in the MIL Machine Vision package, these two packages are mutually exclusive.
3. Select the hardware component, from which to generate a fingerprint, from the **System Fingerprint** drop-down list box. After choosing the hardware component, click the **Generate** button. This will create a lock code.
  - ❖ If you replace the component you have chosen, you will need a new license.

If your computer does not have any of the components listed, or for portability, you can purchase a hardware ID-key instead. This gets attached to your computer's parallel or USB port, and it will allow you to generate a lock code that you can use to create a license. If you attach the hardware ID-key to a different computer, the run-time license will only operate on the computer on which the hardware ID-key is installed. The hardware ID-key allows you to quickly shift the MIL license to another computer without removing a hardware component. It also allows you to upgrade to other ActiveMIL packages at a later point in time; a hardware ID-key allows you flexibility in choosing the package you want, unlike a hardware-license key.

You also have to copy over the *lservrc* file, which is located in the *WIN NT\SYSTEM 32* directory under Windows NT/2000 or the *WINDOWS\SYSTEM* directory under Windows 98/Me, to the identical directory on the target computer.

4. After you have generated a lock code, the software license-key can be obtained at Shopmatrox.com or by contacting Matrox Imaging sales or your local representative.
5. Return to the License Manager program, type or paste the software license-key into the **Software License Key** edit field, and click on the OK button. The licensing process is finished.

## Hiding the MIL license

---

It is possible to make the presence of MIL licensing non-invasive so that your customer can install your MIL-based application and not have to deal with MIL licensing issues.

Perhaps the simplest way to hide the MIL licensing is to purchase and redistribute hardware license-keys. You will have to purchase a hardware license-key for each application that you are distributing.

If this is not possible or preferable, there is a utility, Gencode, that allows you to get a software license-key while hiding the MIL licensing process. It is essentially the silent command-line version of the MIL License Manager. Gencode can generate the lock code needed to get a MIL software license-key, as well as register the license.

### Generating the lock code

There are many different ways of getting the information that the utility needs. You could create an interactive interface, or have a series of prompts during the setup of your application. If you know the hardware and the packages your client will be using, you could build a batch file which would call Gencode with the predetermined information. For example, if you know that your client will be using a Matrox Corona-II board, you could hardcode Gencode's command-line parameter **SystemFingerprint** to the constant that means Matrox Corona-II.

However implemented, your application must call Gencode as follows to generate the lock code:

```
Gencode /G Filename SystemFingerprint Packages
```

See the next section for details.

**Getting a software  
license-key**

Once the code is actually created, there are some steps that must be followed.

1. Your customer must send you the lock code, by whichever means you designate.
2. Take the lock code, and obtain the software license-key at Shopmatrox.com or by contacting Matrox Imaging Sales or your local representative.
3. Your customer must input the software license-key into whatever registration interface you have created.

**Registering the  
software license-key**

Your registration interface must again call Gencode, this time as follows:

```
Gencode /L LicenseCode
```

and input the license code. This will register your application with MIL.

Gencode's command-line parameters are described in the following section.



# Gencode Utility

Synopsis	Generates a lock code for licensing, and activates a software license-key for MIL.
Protype	<p>GENCODE /H or</p> <p>GENCODE /G &lt;Filename&gt; &lt;SystemFingerprint&gt; &lt;Packages&gt; or</p> <p>GENCODE /L &lt;SoftwareKey&gt;</p>
Description	<p>The Gencode utility does one of the following:</p> <ul style="list-style-type: none"><li>• With the /H switch, it displays a help screen.</li><li>• With the /G switch, it generates a lock code that is necessary for getting a software license-key from the Matrox web site.</li><li>• With the /L switch, it registers the license-key for running MIL on the current computer.</li></ul>
Parameters	<p>The <b>Filename</b> parameter is the name of the file that will be created to store the lock code. You can specify any file name; if the file name already exits, the lock code information will be appended to the existing file. The lock code can also be retrieved from the registry under:</p>

HKEY\_LOCAL\_MACHINE\SOFTWARE\Matrox\MIP\MilRuntime\Redist\Licenses\ManagerOutput

The **SystemFingerprint** parameter designates the hardware component upon which the system fingerprint is based. This parameter can be set to one of the following numeric values.

SystemFingerprint setting	Description
2	MGA Graphics Controller
7	Ethernet Controller
8	Hardware ID Key
31	Matrox Board

The **Packages** parameter identifies which MIL packages are going to be purchased. This is something you would likely set for your customer, since you know which MIL modules your application uses.

The software license-key packages correspond to the following numeric values. You can set the **Packages** parameter to any combination of the numeric values given. If you need to buy more than one package, add their corresponding numeric values together. For example, to purchase the Machine Vision and Identification packages, set the **Packages** parameter to 6.

Packages setting	Description
1	Image Analysis
2	Machine Vision
4	Identification
8	Compression/Decompression
16	Geometric Model Finder

The one exception is that the Image Analysis and Machine Vision packages can not both be included in the value of the **Packages** parameter.

The **SoftwareKey** parameter is the software license-key code that you receive from Matrox or your local representative to finalize registration.

# Index

## A

- absolute value
  - image 67
  - result of operation 79
- absolute world coordinate system 120
- AC Huffman table 420
- accentuating edges 63
- acceptance level
  - definition 184
- acquisition
  - attribute 313
  - continuous 39
  - image 38, 380
  - input LUT 409
  - precondition 390
- active edges 200
- adding, image 67
- address
  - Host 328
  - logical 328
- alignment
  - angular 166
  - fiducial marks 172
  - image rotation 53
  - vertical and horizontal 163
- allocate
  - application 28
  - buffers 35
  - child buffer 316, 410
  - data buffer 310
  - defaults 29
  - digitizer 38, 380
  - display 346
  - graphics context 374
  - image buffer 33, 313
  - LUT buffer 69, 341–342
  - measurement marker 278
  - multi-band buffer 408
  - OCR result buffer 244
  - pattern matching model, manual 173
  - pattern matching result buffer 174
- allocation error 315

- angle
  - marker 302
  - orientation 168
- angular alignment 166
- annotation
  - display 358
  - image 374
- application
  - building 28
  - processing, typical 43
  - simultaneous processing 436
- application context 438
- arcs, draw 375
- arithmetic encoding 428
- arithmetic operations 67
- aspect ratio 48, 139, 141, 149
  - definition 140
- auto-focusing 400
- auxiliary display 348
  - Matrox Millennium G400, G450, G550 350
  - video output format 349
- auxiliary screen 348
- average, input data 49–50

## B

- background color
  - associate to graphics context 374
- background, blobs 128
- bar code
  - supported buffer types 263
- bar codes 262
- Bayer cameras
  - formats 335
  - grabbing with 332–333
  - using 332–333
- Bayer images
  - converting to color images 332–333, 335
  - white balancing 337
- BC412 265
- Bearer bars 268
- bicubic interpolation 107
- bilinear interpolation 107
- binarize 43, 56, 60, 138
- Binary buffers 322
- binary buffers, packed 312
- binary measurements, blobs 129, 146
- bisection strategy 400, 402

- bit-plane decomposition 427
- blanking, display 353
- blob analysis 127–128, 137, 145
  - all-blob interpretation 142
  - area 148
  - binary measurements 129, 146
  - blob location 156
  - blob-group interpretation 142
  - calculating with blob runs 159
  - capabilities 21
  - compactness 152
  - controls 130, 139
  - coordinates 141
  - count by label 59
  - example, compactness 153
  - feature list 130, 146, 150
  - feature selection 128
  - foreground 138–139
  - grayscale measurements 129, 146
  - module 129
  - non-calibration of results 148
  - number of holes 155
  - perimeter 148
  - result buffer 130, 143
  - results 142
  - roughness 152
  - selecting blob features 146
  - selecting blobs 143
  - single-blob interpretation 142
  - speed 143, 146–147, 151
  - steps to performing 129
  - transformation 135
- blob identifier image 139
  - acquisition 138
  - background 128
  - blob group 139, 142
  - definition 130
  - foreground 128, 138
  - identifier type 139
  - interpretation 139
  - labelled 142
  - lattice 139–140
  - noise 139
  - pixel aspect ratio 139–140, 149
  - preprocess 130, 139
  - purpose 129, 138
  - segmentation 138
- blobs
  - area 75, 148
  - binary, feature extraction 129
  - border-touching 135
  - breadth 151
  - break apart 52, 69, 130, 139
  - calculate features 142
  - center of gravity 156
  - compactness 152
  - convex perimeter 149
  - counting 132
  - definition 21, 128
  - dimensions 150
  - distinguish 130
  - feature extraction 146
  - feature ordering 128
  - features 129–130, 145–146
  - Feret diameter 139, 141, 150
  - Feret diameter, See also Feret diameter 146
  - grayscale, feature extraction 129
  - grouping 142
  - holes 139, 148, 155
  - identifier types 129
  - identifying 138
  - including See blob identifier image 146
  - label 75, 158
  - locate 59
  - location 156
  - moments 146, 158
  - noise 139
  - number of 75
  - perimeter 148
  - reconstruction 135
  - roughness 152
  - runs 159
  - selecting 143
  - shape 152
  - sizing 151
  - touching 138, 140
- border
  - search accuracy 186
- border handling
  - neighborhood operations 79, 83
- borders
  - blobs touching 135
- brightness, adjust on input 388

- buffer
  - accessing a 327
  - RGB 320
  - storage format 319—322, 325
  - user-allocated 328
- buffers
  - address 328
  - binary 322
  - displayable 314
  - grab 314
  - pitch of 328
  - supported 129
  - YUV 322

## C

- calibrating images 112
- calibration 148
- camera
  - acquisition from 38, 380
  - adjusting/focusing 39, 400
  - sophisticated 380
  - specification 380
- catchment basins 96
- cellular mapping 90
- center of gravity 132, 156
- central moments 158
- certainty level
  - definition 184
- Chained pixels 156
- Chamfer 3-4 transform 74
- characters, text 377
- checksum number, code 267
- Chessboard transform 73
- child buffers 316
  - allocate 316
  - color 410
  - data buffer attributes 316
  - definition 310
  - dimensions 316
  - display 316, 357
  - display multiple 354
  - inheriting parent features 316
  - LUT 341
  - offset from parent 316
  - returned coordinates 316
  - size 316
- circles, draw 375

- City Block transform 73
- clear
  - display 353
  - graphics image buffer 375
- clipping
  - borders 317
  - graphics 376
  - pixel values 62, 138
- closing operation 43, 49, 52
- Codabar 265
- code 262
  - cell 266
  - cell size 266, 270, 272
  - checksum number 267
  - code type 268
  - encoding non-printing characters 272, 274
  - encoding scheme 267—268
  - error correction 263, 267, 269
  - escape sequence format 272, 274
  - image size 272
  - number of cells 270, 273
  - quiet zone 268
  - read operations 262—263, 268—269, 272
  - search angle 270
  - search speed 271
  - start and stop characters 267
  - string size 271
  - supported buffer types 263
  - threshold value 271
  - write operations 262
- Code128 265
- Code39 265
- codes 262
- coefficients, warping 105
- color
  - handling techniques 407
  - input LUT 409
- color band 311
- color images
  - allocate buffer 408
  - allocate child buffer 410
  - color conversion 410
  - copy 410
  - copy single band 317
  - dealing with 408
  - displaying 409
  - grabbing 408
  - loading 413

- processing 410
- processing restrictions 410
- put data in band 318
- reference levels 389
- saving 413
- column profile 59
- commands
  - functions 29
  - pseudo-MIL 448
- communication channels 28, 31
- compactness, blob 152
- comparative operations 67
- compass gradient 65–66
- compiling 29
- complex operations 69
- compressing images 416
- conditional buffer, creating 374
- connectivity
  - code 90
  - mapping 90
- constant thresholding 42
- continuous grab 39
- contrast 389
  - image, adjusting 56, 62, 388
  - marker/background 290
- control
  - areas processed 316
  - neighborhood center 83
  - neighborhood operation 79
- conversion
  - color 410
  - data format 319
- convex perimeter 149
- convex perimeter, find 88
- convolution 64
- coordinates
  - child buffer 316
  - measurement marker 286
  - model 173
  - of a pixel 331
  - search result 163
  - text writing 377
- copy
  - bit truncation/extension 317
  - clip, and 317
  - color band 317, 342, 410
  - conditional 317
  - data 317

- data to LUT 342
- mask 317
- model 173
- specific buffer areas 317
- Corona-II
  - exposure 396, 398, 400
    - automatic model 397
    - bypass model 398
  - triggers 396, 398, 400
- counting
  - dark particles 60
  - objects 43, 75, 153
- CPU
  - CPU-assisted display 359
- custom
  - morphological operations 82
  - spatial filters 78
  - structuring element 83
  - window, VGA 362

## D

- data allocation and access module 309
- data average 50
- data buffer
  - attributes 313, 315
  - automatically allocated 318
  - child 310, 316
  - clear 375
  - clip border 317
  - color band 311, 408
  - defined 310
  - depth 312
  - dimensions 311
  - display 345
  - export data 319
  - free 311
  - get data, put in array 318
  - handling 309
  - import 318
  - incorrect usage 315
  - integer 312
  - intended usage 313
  - location 313
  - LUT, see LUT buffer 341
  - management 318
  - multiple, display 316
  - multiple, handling 316

- packed binary 312
- put data 318
- range 312
- restore 318
- save 319
- type 62, 313
- data format, input device 380
- data generation
  - LUT 341
- data type 35
  - changing 62
- data, overwriting 35
- Datamatrix 265
- DataMatrix codes 262
- DC Huffman table 420
- dcf files 381
- decompressing images 416
- default graphics context 374
- defaults
  - display 29
  - image buffer 25, 29, 35, 37
  - initializing 25
  - input device 380
  - input LUT 390
- defect highlighting 63
- depth
  - data buffer 312
- destination buffer 35
- device
  - control module 379
- differences
  - image, count 56
- digitization, definition 42
- digitizer
  - allocate 38, 380
  - color data format 408
  - configuration format 380
  - frame averaging 50
  - free 380
  - input channel 381
  - inquire 380
  - LUT 344, 390, 409
  - reference levels 388
- digitizer configuration files 381
- dilation
  - advanced 83
  - basic 43, 69–70
  - binary algorithm 84
  - conditional 84, 135
  - grayscale algorithm 84
  - opening/closing operation 52
- dimensions, blob 150
- displacement
  - search maximum 186
- display
  - allocation 28
  - annotation 358
    - Windows GDI 358, 360
  - border handling 346
  - buffer 35, 316
  - clear 353
  - color 368
  - color image 409
  - control module 345
  - CPU-assisted 359, 371
  - default 29
  - device number 350
  - example 363
  - free 363
  - image location 346
  - keying 359
  - LUT 343, 368, 409
  - memory 346
  - monochromatic effect 343
  - monochrome buffer 34
  - multiple buffers 354
  - non 8-bit buffers 351
  - pan 357
  - pseudo-color effect 343
  - pseudo-color LUT 369
  - scroll 357
  - size and depth 350
  - true color effect 344
  - user-defined window 362
  - zoom 357
- display type 347
  - auxiliary 348
    - Matrox Millennium G400, G450, G550 350
    - video output format 349
  - windowed 347
    - extended desktop 347
- Displayable buffers 314
- distance transforms 73
- distinguishing edges 63
- distortions 48–49, 53, 112
- dividing, image 67

- dontcare, kernel value' 83
- dots, draw 375
- double buffering, definition 391
- drawing 374–375
- dynamic range 388

## E

- EAN13 265
- edge
  - enhance for contrast 64
  - enhancers 43, 63–64, 78
  - inside stripe marker 298
  - markers 277
- edge extractors 66
  - compass gradient 65–66
  - definition 63–64
  - horizontal 63, 65
  - Laplacian 64–66
  - oblique 63
  - predefined kernels 65
  - vertical 63, 65
- erosion 43
  - advanced 83
  - basic 69–70, 82
  - binary algorithm 84
  - grayscale algorithm 84
  - opening/closing operation 52
- error reporting
  - automatic 37
  - memory, insufficient 315
  - message control 29
  - thread 438–439
- event, locate 56
- examples
  - blob analysis, compactness 153
  - change data type 62
  - color image manipulation 411
  - color, run with 408
  - custom structuring element 86
  - display in user-defined window 363
  - display multiple buffers 354
  - extract background 67
  - filter with custom kernel 81
  - find perimeter of object 71
  - general information 25, 31
  - grab 38
  - image allocation/display 36
  - image analysis 43
  - installing 25
  - kernel 78
  - mblobcnt.c 153
  - mcolor.c 411
  - mconvol.c 81
  - MIL sample program 31
  - mmultdis.c 354
  - mocrfont.c 257
  - mocrread.c 245
  - mocrview.c 254
  - modify for color 34
  - mopen.c 86, 88
  - mperim.c 71
  - msearch.c 174
  - mstart.c 31
  - msurvey.c 67
  - mwindisp.c 363
  - Native Mode ProgrammersToolkit' 449
  - OCR, calibrate a font 245
  - OCR, create custom font 257
  - OCR, visualize font characters 254
  - optical character recognition 245, 250
  - pattern matching, define model and search 174
  - standard defaults 34
  - structuring element 84
  - wafer alignment 164
- excluding blobs 143, 148
- export data buffer 319
- exposure 400
  - automatic model
    - Corona-II 397
  - bypass model
    - Corona-II 398
  - Corona-II 396
- extreme value, find 44, 56, 58

## F

- false matches 184
- feature list, blob analysis 130
- Feret diameter
  - angle 151
  - aspect ratio 141
  - average 150
  - blob feature 146
  - convex perimeter 149
  - general 150



- illustrated 150
- minimum/maximum 150
- number of 139
- fiducial marks 172
- field
  - grabbing 382
- field-of-view 123
- file format 319
- files
  - semi1292.mfo 248
  - semi1388.mfo 248
- fill
  - holes 52
- filled-in shapes 376
- filter
  - temporal 50
- find
  - buffer extremes 56
  - marker 280, 288
  - model 174
- first-order polynomial warping 105
- focus indicator 400
- font
  - associate to graphics context 377
  - predefined 377
  - scale 377
  - size 377
- foreground color
  - associate to graphics context 375
  - fill with 376
- foreground, blobs 128, 139
- frame
  - averaging 50
  - grabbing 39, 382
- frame buffer 346
- free
  - buffer, data 311
  - buffer, image processing result 59
  - graphics context 374
- full range digitization 58
- function
  - development 449
  - execution success 29
  - user-created 22
- functions
  - commands 29

## G

- gamma correction 344
- Gaussian noise 48–51
- generating warping coefficients 105
- geometric model finder
  - adding models 201
- Geometric Model Finder 198
- geometric model finder
  - acceptance levels 213
  - accuracy 227
  - angular search 220
  - annotating results 228
  - calibration 229
  - certainty levels 214
  - choosing models 205
  - determining matches 209
  - fit error 211
  - fit error weighting factor 214
  - image processing controls 202
  - interpreting results 212
  - masks 208
  - model coverage 209
  - model indices and labels 201
  - number of occurrences 215
  - polarity 222
  - reference angle 218
  - reference position 216
  - scale 221
  - score 209
  - search speed 227
  - separation 223
  - target coverage 210
  - target score 209
  - transformation coefficients 218
- geometric transforms 104
- Grab 314
- grab 390
  - color images 408
  - continuous 39, 372
  - data average 49–50
  - data buffer 313, 408
  - example 38
  - frames 39, 382
  - halt 39
  - image 38, 43, 380
  - mode 391
  - monochrome 38

- multi-dimensional buffers 408
  - sequence 50
  - synchronization 391
- Grab buffers 314
- GrabAndWarp()
  - example 449
- grabbing
  - single frame 382
- graphics 374
  - arcs, draw 375
  - boundary type seed fill 376
  - buffer, clear 375
  - capabilities 20
  - circles, draw 375
  - clipping 376
  - dots, draw 375
  - filled elliptic arcs, draw 375
  - filled rectangles, draw 375
  - filled-in shapes 376
  - lines, draw 375
  - module 373
  - outline, draw 375
  - parameters 374
  - plot a histogram 58
  - rectangles, draw 375
  - text, write 377
- graphics context
  - allocate 374
  - background color, associate 374
  - default 374
  - definition 374
  - font scale, associate 377
  - foreground color, associate 375
  - free 374
  - object parameters 374
  - text font, associate 377
- graphics controller
  - memory 346
- grayscale images, modules using only 20

## H

- halt grabbing 39
- header file 29
- hierarchical search 192
- histogram
  - equalization 62
  - generate 56
  - plotting 58
  - statistical operation 42
- hit or miss pattern matching 90
- holes
  - blobs, in 148
  - blobs, to distinguish 155
  - extract 135
  - fill 49, 52, 135
- horizontal edges 63, 65
- host
  - communication 28
  - CPU 20
  - default system 311
  - screen 29
  - system 342
- hue 410
- Huffman encoding 419—420

## I

- Image
  - placement 357
- image
  - addition 67
  - analysis 42
  - arithmetic operations 67
  - binarize 60
  - column profile 59
  - comparison 42
  - contrast 56
  - data type 62
  - definition 42
  - differences, count 56
  - dilation 69
  - distortion 49, 53, 184
  - division 67
  - edge enhancement 63
  - enhancement 42
  - erosion 69
  - extreme values 56, 58
  - grabbing 38

- histogram 56
- histogram equalization 62
- inversion 69
- manipulation, advanced 77
- manipulation, basic 42
- manipulation, order 56
- mapping 42, 69
- multiplication 67
- noise reduction 43
- pixel value clipping 62
- projection 56, 59
- quality 48–49
- row profile 59
- sharpening 43
- smoothing 43
- statistics 56
- subtraction 42, 67
- thresholding 60
- transformation 135
- image buffer
  - acquisition 313
  - allocation 33, 35–36, 313
  - color 34, 38
  - conditional 374
  - default 29, 35, 408
  - defined 35
  - destination 35
  - display 36, 313
  - display border 346
  - display multiple 354
  - display position 346
  - free 311
  - map through LUT 340
  - multi-band 410
  - processing 313
  - removing from display 353
  - select for display 346
  - size 35
  - source 35
  - two-dimensional 34–35, 38
  - uses 35
- image coordinate system 120
- image digitization 42
- image processing 42
  - application 43
  - capabilities 20
  - module 41, 47, 55
  - operation ordering 56
  - result buffer 57, 59
- imIntDistance() 73
- impulse noise 48
- include file 29
- initialization
  - input device 380
  - system 25, 380
- input device
  - brightness 388
  - contrast 388
  - defaults 380
  - frequency 380
  - line-scan 382
  - LUT 390
  - resolution 380
  - subsampling 390
  - using 38
- inquire
  - digitizer 380
  - font, OCR 253
- installation
  - MIL 25
  - test program 31
- integer buffers 312
- Intellicam 381
- intensity
  - correction 342
  - distribution 56, 62
  - histogram 56
  - HLS 410
- interlaced JPEG compression 416
- Interleaved 2/5 265
- interpolation modes 107
- isthmuses 49, 52

## J

- JPEG
  - discrete cosine transform 420
- JPEG compression 416
- JPEG2000
  - discrete wavelet transform 424
- JPEG2000 compression 416, 422, 425

## K

### kernels

- buffer allocation 79, 313
- defined 51
- example, sharpen/smooth 78
- predefined 51, 64
- usage 64, 66, 78
- user-defined 79

### keying 359

## L

### labelling

- blobs 43–44, 158
- blobs, use 59, 75

### Laplacian edge detection 64–66

### lattice, blobs 139

### length

- measurement marker 290, 297

### lens curvature 48

### lens motor 400

### line equation

- measurement marker 290, 297

### linear interpolation function 120

### lines

- draw 375

### line-scan device 382

### link program with library 29

### load

- color image 413
- data 318
- LUT data 342

### locating blobs in an image 59, 156

### logical operations 67

### look-up table

- 1-band custom 369
- 3-band custom 370
- changing default 369
- control loading into physical output LUTs 366
- pseudo-color 369

### lossless compression, JPEG 416

### lossless compression, JPEG2000 416, 422, 425

### lossy compression, JPEG 416

### lossy compression, JPEG2000 416, 422, 425

### low-pass spatial filters 49

### luminance

- HLS 410

## LUT

### 1-band custom 369

### 3-band custom 370

### changing default 369

### control loading into physical output LUTs 366

### pseudo-color 369

## LUT buffer

### allocation 313, 341

### child buffer 341

### color bands 341, 409

### data generation 341–342

### dimensions 341

### load 342

### management 341

### one-dimensional 341

### restore 342

## LUTs

### allocate 69

### connectivity mapping 90

### custom 369

### definition 340

### display 343, 409

### display color, change 368

### general information 339

### index 341

### input 388, 390, 409

### input mapping 69, 344

### intensity correction 342

### monochromatic effect 343

### multiple-color-band 370

### one-color-band 369

### processing 69, 343

### pseudo-color effect 343

### ramp 341

### transformation 63

### true-color effect 344

### usage 343

## M

### M\_DISP 314

### M\_GRAB 314

### machine guidance 156, 172

### MappAlloc() 29, 374, 438

### example 363

- MappAllocDefault() 25, 28, 35, 37–38, 346, 374, 380, 408–409
  - example 31, 36, 38–39, 45, 67, 72, 81, 86, 174, 245, 254, 257, 354, 449
- MappControl() 29
- MappControlThread() 438–439
- MappFree() 29
  - example 363
- MappFreeDefault() 28
  - example 31, 36, 38–39, 45, 58, 67, 72, 81, 86, 153, 164, 174, 254, 257, 354, 411, 449
- MappGetError() 29, 439
  - example 36, 164
- MappHookFunction() 29
- mapping 69, 90
- mapping pixels to world 112
- marker
  - allocation 278
  - center 289, 294
  - characteristics 279, 288
  - contrast 290
  - edge 277
  - find 280, 288
  - find, speed 284, 290
  - find, tolerance 285
  - managing 277
  - measure 280
  - measurement box 284–285, 297
  - parameters 279, 288
  - processing area 279
  - reference 289
  - reference position 294
  - region of interest 284
  - stripe 277
  - types 277
- mask, copy 317
- matrix-defined warping 106
- matrix-defined warpings
  - generating LUTs for 106
- Matrox READER 242
- Maxicode 265
- maximum
  - pixel value 42, 58
- MblobAllocFeatureList() 130, 146
  - example 153
- MblobAllocResult() 130
  - example 153
- MblobCalculate() 130–131, 143, 146, 156
  - example 153
- MblobControl() 130, 139–142, 149–150
  - example 153
- MblobFill() 136, 143, 147
- MblobFree()
  - example 153
- MblobGetLabel() 131, 158
- MblobGetNumber() 131
  - example 153
- MblobGetResult() 131
- MblobGetResultSingle() 131, 158
- MblobGetRuns() 131, 158
- MblobLabel() 136, 147
- MblobReconstruct() 84, 135
- MblobSelect() 131, 143, 147
  - example 153
- MblobSelectFeature() 130, 146, 150–151, 158
  - example 153
- MblobSelectFeret() 130, 146, 150
- MblobSelectMoment() 130, 146, 158
- MbufAlloc() 327
- MbufAlloc1d() 69, 310, 341, 369
- MbufAlloc2d() 35, 79, 83, 310, 354
  - example 36, 67, 72, 86, 363
- MbufAllocColor() 29, 310, 319, 341, 408
- MbufBayer() 332–333, 335, 337
- MbufChild1d() 316
- MbufChild2d() 316, 354
  - example 45, 81, 86, 174, 245, 257, 354, 411
- MbufChildColor() 316, 410
  - example 411
- MbufClear() 375
  - example 257, 354, 363
- MbufControl 360
- MbufControl() 360, 369–370, 417, 432
  - example 449
- MbufControlNeighborhood() 79, 83
  - example 81
- MbufCopy() 342, 417
  - example 67, 86
- MbufCopyClip() 317
- MbufCopyColor() 317, 342, 410
- MbufCopyCond() 317
- MbufCopyMask() 317
- MbufCreate2d() 328
- MbufCreateColor() 328
- MbufExport() 319, 413, 417

- MbufExportSequence 418
- MbufFree() 29, 311, 316, 341
  - example 45, 58, 67, 72, 81, 86, 153, 164, 174, 257, 354, 411
- MbufGet() 318, 327
- MbufGet1d() 318
- MbufGetColor() 318
- MbufImport() 318, 413, 417
- MbufImportSequence() 418
- MbufInquire() 328–329, 360
  - example 254, 449
- MbufLoad() 318, 342, 413
  - example 45, 72, 86, 174, 245, 354, 411
- MbufPut() 79, 83, 318, 327, 342, 369
  - example 81
- MbufPut1d() 69, 318, 342
- MbufPut2d() 79, 83
  - example 86
- MbufPutColor() 318, 342
- MbufRestore() 318, 342, 413
- MbufSave() 319, 413
- McalAlloc() 113
- McalAssociate() 114
- McalControl() 117
- McalGrid() 113, 117, 125
- McalList() 113, 117, 119, 125
- McalRelativeOrigin() 123
- McalTransformCoordinate() 113
- McalTransformImage() 113
- McalTransformResult() 113
- McodeAlloc() 264
- McodeControl() 264
- McodeFree() 264
- McodeGetResult() 264, 274
- McodeRead() 264
- McodeWrite() 264
- mconvol.c 81
- MdigAlloc() 29, 38, 380–381, 408
  - example 363
- MdigChannel() 381
- MdigControl() 39, 391, 437
- MdigFocus() 400
- MdigFree() 29, 380, 408
  - example 363
- MdigGrab() 39, 391, 408, 417
  - example 38, 67
- MdigGrabContinuous() 39, 408
  - example 39, 67, 363
- MdigGrabWait() 391
- MdigHalt() 39, 391
  - example 39, 67, 363
- MdigHookFunction() 437
- MdigInquire() 380
  - example 67, 449
- MdigLut() 63, 344, 390, 409
- MdigReference() 389, 408
- MdispAlloc() 29, 35, 346–347, 359, 409
  - device number 350
  - example 363
  - M\_AUXILIARY 349
  - M\_WINDOWED 347
- MdispControl() 351, 360
- MdispDeselect() 35–36, 353, 409
  - example 354, 363
- MdispFree() 29, 353
  - example 363
- MdispHookFunction() 360
- MdispInquire() 360, 371
- MdispLut() 63, 343, 369–370, 409–410
- MdispOverlayKey() 359
- MdispPan() 357
- MdispSelect() 35, 314, 317, 346, 354, 409
  - example 354, 411
  - VGA 362
- MdispSelectWindow() 362
  - example 363
  - VGA 362
- MdispZoom() 357
  - example 354
- measurement context
  - free 279
- measurements
  - angle, marker 302
  - capabilities 21
  - context 279
  - control settings 279
  - length, marker 290, 297
  - line equation 290, 297
  - markers, multiple 296
  - markers, using 280
  - module 275
  - position 289
  - preprocess image 280
  - region of interest 284
  - steps 278
  - width, stripe 297

- measurements, blob
  - binary 129
  - compactness 152
  - dimensions 151
  - discriminate, to 128
  - general 146
  - grayscale 129, 146
  - pixel units 149
  - roughness 152
- median filter 51
- memory
  - insufficient 315
  - resources 28–29
- messages, error 29, 37
- MgenLutFunction() 69, 341
- MgenLutRamp() 69, 341–342, 369
- MgenWarpParameter() 105
- MgraAlloc() 374
- MgraArc() 375
- MgraArcFill() 375
- MgraBackColor() 374
- MgraClear() 375
- MgraColor() 375
- MgraDot() 375
- MgraFill() 376
- MgraFont() 377
  - example 257
- MgraFontScale() 377
  - example 257
- MgraFree() 374
- MgraLine() 375
- MgraRect() 375
  - example 164, 174
- MgraRectFill() 375
- MgraText() 377
  - example 31, 257, 363
- MIL
  - file format 319
  - header file 29
  - include file 29
  - objects 22
  - running application 31
- MIL Configuration utility 27
- MIL modules
  - blob analysis 127, 137, 145
  - display control 345
  - geometric model finder 197
  - graphics 373
  - I/O device control 379
  - image processing 41, 47, 55
  - measurements 275
  - pattern matching 161, 171
- mil.h 29
- mil.ini
  - Meteor-II 386
- MILINTER 243
- milsetup.h 25, 28–29, 34, 380, 408
- MimAllocResult() 57, 59
  - example 45
- MimArith() 67, 97, 342
  - example 45, 67, 72, 411
- MimBinarize() 60–61, 138
  - example 45, 72, 86, 153, 354
- MimClip() 60, 62, 75, 138
  - example 62, 411
- MimClose() 52
  - example 153
- MimConnectMap() 90
- MimConvert() 410
  - example 411
- MimConvolve() 51, 64–66, 78
  - example 45, 81, 86
- MimCountDifference() 56
- MimDilate() 52, 70–71, 84–85
  - example 72
- MimDistance() 97
- MimErode() 52, 70–71, 85
- MimFindExtreme() 56, 58–59
  - example 45
- MimFlip() 104
- MimFree() 57, 59
  - example 45, 58
- MimGetResult() 59
  - example 45, 58
- MimHistogram() 56–57
- MimHistogramEqualize() 62–63, 342
- MimLabel() 59, 75
  - example 45, 75
- MimLocateEvent() 56
- MimLutMap() 63, 69, 340
- MimMorphic() 82, 85, 88
  - example 86
- MimOpen() 52, 86
  - example 45, 72, 86, 153
- MimPolarTransform() 104
- MimProject() 56, 59

- MimRank() 51
- MimResize() 49, 53, 104, 141
- MimRotate() 53, 104
- MimThick() 88
- MimThin() 88
- MimTranslate() 104
  - example 174
- MimWarp() 104
- MimWatershed() 96
- minimum pixel value 42, 58
- MmeasAllocMarker() 278, 288
- MmeasCalculate() 279, 289
- MmeasFindMarker() 279–280
- MmeasFree() 279
- MmeasInquire() 279
- MmeasRestoreMarker() 278
- MmeasSetMarker() 279, 288, 298
- MmodAlloc() 201
- MmodControl() 202
- MmodDefine() 201
- MmodDraw() 202
- mmultdis.c 354
- MMX Technology, Intel 23
- mnatgen.c 449
- MocrAllocFont() 255
  - example 257
- MocrAllocResult() 244
  - example 257
- MocrCalibrateFont() 249
- MocrControl() 244, 251
  - example 257
- MocrCopyFont() 243, 253, 255
  - example 254, 257
- mocrfont.c 257
- MocrFree()
  - example 254, 257
- MocrGetResult() 244
  - example 257
- MocrImportFont() 243, 255
- MocrInquire() 250–251, 253
  - example 254
- MocrModifyFont() 249, 253
- mocrread.c 245
- MocrReadString() 244, 249
  - example 257
- MocrRestoreFont() 243, 253, 256
  - example 245, 254
- MocrSaveFont() 249, 253, 256
  - example 257
- MocrSetConstraint() 244, 250, 259
  - example 250, 257
- MocrVerifyString() 244, 249
- mocrview.c 254
- model
  - acceptance level 184
  - allocation, automatic 163
  - allocation, manual 173
  - center 163, 186
  - coordinates 163, 173
  - copy to image buffer 173
  - default search parameters 174
  - dontcarepixels' 191
  - find 163, 183
  - load 174
  - number of matches 184
  - positional accuracy 187, 189
  - preprocess 188–189
  - reference point 186
  - rotate 173, 184
  - search parameters 174
  - size 173, 189
  - storage location 173
  - view 173
- moments 146, 158
  - central 158
  - ordinary 158
- monochromatic effect 343
- monochrome image buffer 35
- mopen.c 86
- morphological operations
  - binary 83
  - custom 82
  - erosion/dilation 70
  - grayscale 83
  - standard 43, 52
- MpatAllocAutoModel() 163, 173
- MpatAllocModel() 168, 173
  - example 174
- MpatAllocResult() 163, 167, 174
  - example 164, 174
- MpatCopy() 173
- MpatFindModel() 163, 174, 184–185, 193
  - example 164, 174
- MpatFindOrientation() 167, 169



- MpatFree() 174
  - example 164, 174
- MpatGetNumber() 164, 174, 184
  - example 164
- MpatGetResult() 168–169, 174, 192
  - example 164, 174
- MpatInquire() 163
  - example 164, 174
- MpatPreprocModel() 163, 188–189, 194
  - example 174
- MpatSave()
  - example 174
- MpatSetAcceptance() 184
- MpatSetAccuracy() 187, 190, 195
  - example 174
- MpatSetCenter() 185–186
- MpatSetCertainty() 185
- MpatSetDontCare() 173
- MpatSetNumber() 184
- MpatSetPosition() 164, 186, 190
- MpatSetSearchParameter() 193–194
- MpatSetSpeed() 187, 189, 195
  - example 174
- mstart.c 31
- MsysAlloc() 29
  - example 363
- MsysControl()
  - example 449
- MsysFree() 29
  - example 363
- MsysInquire()
  - example 363, 449
- multi-dimensional buffers 408
- multiple buffers
  - displaying 354
- multiple fields of view 123
- multiplying, image 67
- multi-processing 435–436
  - definition 436
- multi-threading 435, 437
  - definition 437
- mwindisp.c 363

## N

- native mode 447
  - example code 449
  - integrating with MIL 448
  - interface 448
- nearest-neighbor interpolation 107
- neighborhood
  - bordering pixels 80
  - center pixel 79, 83
  - convolution method 79
  - dontcarepixels' 83
  - operations 43, 49, 51–52, 64, 70, 78, 82, 88
  - overscan 79, 83
- noise 43
  - blobs 139
  - Gaussian 48–50
  - impulse 48
  - random 48, 50–51
  - reduction 51, 78, 129, 139
  - salt-and-pepper 48–49, 51
  - shot 48
  - systematic 48, 51
- non 8-bit buffers
  - displaying 351
- normalization factor 79
- normalized grayscale correlation 191
- number of
  - objects 59

## O

- object orientation 168
- oblique edges 63
- open communication 28, 31
- opening operation 43, 49, 52
- optical character recognition 242
  - acceptance levels 251
  - allocating result buffer 244
  - calibrating fonts 243, 249
  - character constraints 244, 250
  - character dimensions 249
  - contrast enhancement 252
  - creating fonts 243, 255
  - erasing characters 252
  - examples 245, 250, 254, 257
  - font files 248–249
  - inquiring about fonts 253
  - inverting fonts 253

- loading fonts 243
- managing fonts 253
- match scores 251
- processing controls 244, 251
- READER 242
- reading results 244
- reading strings 244
- restoring fonts 253, 256
- saving fonts 253, 256
- semi1292.mfo font file 248
- semi1388.mfo font file 248
- speeding up 252, 259
- steps 243
- string location 252
- target images 244
- unrecognized characters 252
- verifying strings 244
- visualizing fonts 253
- ordinary moments 158
- orientation
  - image 53, 166
  - object 166, 168
  - whole-image 167
- overlay
  - buffer 358
    - behavior 359
  - flickering 371
  - simulated 372
- overscan 108
  - mirror 80
  - pixels 79, 83
  - transparent 80
- overwriting data 35

## P

- packed binary buffers 312
- palette
  - image 341
- panning, display 357
- parent buffer 310, 316
  - display 354, 357
- pattern matching
  - algorithm 191
  - alignment 163
  - basic 60
  - capabilities 21
  - example 164

- module 161, 171
- morphological 43, 89
- peaks 194
- result buffer 163, 174
- score 184
- search steps 172
- usage 162
- PDF417 265
  - reading 266
  - writing 266
- perimeter
  - convex 149
  - normal 148
- perspective transformation function 120
- perspective warping 106
- physical memory 35
  - buffer allocation 313
- pitch 328
- pixel
  - area 148
  - aspect ratio 48–49, 130, 139–141, 148–149
  - coordinates 331
  - depth 22
  - dontcare' 173, 191
  - height 149
  - location 56, 83
  - perimeter 148
  - real-world units 148
  - units 149
  - value density of diagonals 59
  - value distribution 62
  - value, minimum/maximum 42
  - value, minimum/maximum 388
- pixel reference position 280
- pixel-to-world mapping 112
- plotting histogram 58
- point-to-point operations 42, 67
- portability
  - native mode 448
- position of marker 289
- positional accuracy 187
- predictive coding 419
- preprocess
  - input data 390
  - measurement target image 279
  - model 163, 174, 188–189

- processing
  - attribute 313
  - limiting 316
  - single band 410
  - with LUT 343
- profile 59
- program examples 25
- projecting an image 56, 59
- pseudo-color
  - effect 343
- put data
  - array, from 318

## Q

- quantization
  - JPEG 421
- quiet zone 268

## R

- ramp, LUT 341
- random noise 48, 51
- rank filters 51
- read operation, code 262–263, 268–269
- read.me 25, 27, 30–31
- READER 242
- reading codes 262
- real-world results, obtaining 112
- reconstruct object from seed 135
- rectangles, draw 375
- reference level
  - analog 388
  - black/white 58, 388
  - controls 388
  - input channel 388
- reference point, model 186
- reference position 289
  - measurement marker 294
- refocus strategy 403
- relative camera position 121, 124
- relative coordinate system 120, 124
- remove
  - holes 49
  - small particles 52
- resize image 53
- resolution pyramid 192
- restart markers 421

- restore
  - fonts, OCR 253
  - LUT buffer 342
- result buffer
  - blob analysis 130–131, 142–143
  - image processing 57, 59
  - pattern matching 163, 174
- RGB
  - buffers 320
- rotate
  - image 53
  - model 173, 184
  - tolerance 189
- roughness, blob 152
- row profile 59

## S

- salt-and-pepper noise 48–49, 51
- sample program 31
- saturation
  - HLS 410
  - operation result 79
- save
  - color image 413
  - data 318–319
  - fonts, OCR 253
- scale, input 69, 390
- scaling 390
- scan\_all strategy 404
- scrolling, display 357
- search
  - acceptance level 184
  - accuracy 187
  - area 190
  - basic steps 172
  - border effects 186
  - certainty level 184
  - coordinates 163
  - heuristics 194
  - hierarchical 192
  - model 174
  - number of matches 184
  - parameters 174, 183, 193

- region 186
- results 184
- robustness 189
- speed 173–174, 184, 187–190, 192–193, 195
- subpixel accuracy 195
- segmentation 138
- semi1292.mfo file, OCR 248
- semi1388.mfo file, OCR 248
- sequence averaging 50
- sharpen image 64
- shift
  - operations 67
- shot noise 48
- size
  - child buffers 316
  - data buffer 311
  - image buffer 35
  - LUT buffer 341
  - model 189
  - text character 377
- skeleton, find 88
- smart\_scan strategy 404
- Smatch
  - pattern matching 171
- smoothing 43, 51, 78
- software triggers
  - Corona-II 400
- source buffer 35
- spatial filtering operations
  - algorithm 78
  - custom 78
  - edge enhancement/extraction 64
  - low-pass 49, 51
  - median 49
  - rank effect 51
  - usage 43, 51
- speed
  - marker, find 284, 290
  - model size 189
  - multi-threading 437
  - packed binary processing 312
  - search 174, 184, 187–190, 192–193, 195
  - search, high speed 189
  - search, low speed 190
  - search, medium speed 189
- spurious blobs 139
- square pixels 49
- stop grabbing 39

- storage area 35
- stripe
  - markers 277
  - rising/falling 289
- strobe device 381
- structuring elements
  - buffer allocation 83, 313
  - connectivity 90
  - custom 83
  - default 85
  - defining 83
  - example, custom opening 86
  - example, erosion/dilation 85
  - usage 82, 88–89
- subpixel accuracy 195
- subsampling input 390
- subtracting, image 67
- symbolologies, supported 265
- synchronization
  - of grab 391
  - thread 437–438
- system
  - allocation 28
  - buffers 35
  - configuration 25
  - default 22, 25
  - definition 20
  - device 343, 408–409
  - display criteria 35
  - grab criteria 38
  - initialization 25, 380
  - multiple 29
  - multi-processing capabilities 436
- systematic noise 48, 51

## T

- target system
  - system 22
- test installation program 31
- text
  - character font 377
  - graphics 377
  - support 374
- thickening 88
  - binary algorithm 89
  - grayscale algorithm 89

- thinning 88
  - binary algorithm 88
  - grayscale algorithm 88
- thread
  - application context 438
  - data sharing 437
  - error reporting 438–439
  - multi-threading 435, 440
  - synchronization 438
- thresholding 56, 60
- toolkit
  - Function Developers' 447
- transformation LUT 63
- transformations
  - generating LUTs for 106
- transforming data 319
- trigger device 381
- triggers 398, 400
  - Corona-II 396, 398
- true color effect 344
- typical application
  - binarizing 43
  - extreme value 44
  - grabbing 43
  - labelling 44
  - opening 43
  - smoothing 43

## U

- uniform distribution 62
- user-allocated buffer 328

## V

- vertical edges 63, 65
- view model 173

## W

- warping 104
  - interpolation modes 107
- warpings
  - generating LUTs for 106
- watershed lines 96
- watershed transforms 96
- White balance
  - and Bayer images 337
  - determining coefficients
    - monochrome 338
    - RGB 338
    - YUV 338

- width

- stripe 297

- Window occlusion

- Meteor-II 388

- windowed display 347

- extended desktop 347

- Windows

- custom window, VGA 362

- Windows desktop screen(s) 347

- Windows GDI annotations 358, 360

- Windows NT

- display

- size and depth 350

- extended desktop restriction 347

- working area 118

- world coordinate system 120

- write operation, code 262–263, 268, 272

- writing codes 262

## Y

- YUV buffers 322

## Z

- zoom

- display 357



# Product Support

## Product Assistance Request Form

[illegible]

